PERFORMANCE ENHANCEMENTS FOR FLOATING POINT MODULE FOR SHAKTI PROCESSOR

A Project Report

Submitted by

PRABHANSH PRATAP (EE19M082)

In partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF

ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2021

APPROVAL SHEET

This thesis entitled **PERFORMANCE ENHANCEMENTS FOR FLOATING POINT MODULE FOR SHAKTI PROCESSOR** by **Prabhansh Pratap** is approved for the Degree of Master of Technology, Electrical Engineering.

Examiners:

____

____

____

____

____

Supervisor:

Prof. V. Kamakoti
Dept. of Computer Science & Engineering,
IIT Madras, 600036.

Date: ________________________________

Place: ________________________________

Certificate

This is to certify that the thesis titled PERFORMANCE ENHANCEMENTS FOR FLOATING POINT MODULE FOR SHAKTI PROCESSOR, submitted by Prabhansh Pratap to the Indian Institute of Technology, Madras, for the award of the degree of Master of Technology is a bonafide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V. Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT Madras, 600036

Place: Chennai

Date: June 23, 2021

DECLARATIONS

I declare that this project work titled "PERFORMANCE ENHANCEMENTS FOR FLOATING POINT MODULE FOR SHAKTI PROCESSOR" submitted in partial fulfilment for the award of the degree of Master of Technology is a record of original research work carried out by me under the supervision of Prof. V. Kamakoti, and has not formed the basis for the award of any degree, diploma, associateship, fellowship, or other titles in this or any other Institution or University of higher learning. In keeping with the ethical practice in reporting scientific information, due acknowledgments have been made wherever the findings of others have been cited.

Prabhansh Pratap
M. Tech
Dept. of Electrical Engineering
IIT Madras, 600036

Date: June 23, 2021

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my dear guide Prof. V. Kamakoti for allowing me to undertake research under him. I would like to express my deepest appreciation for his suggestions and motivation.

I convey my sincere gratitude to Dr. Nitya Ranganathan for all her suggestions, patience and support during the entire course of the project. Throughout the course of project she offered immense help and provided valuable suggestions which helped me.

My special thanks go to Shalender Kumar for support during the project work.

I would like to extend my appreciation to all my friends for their help and support during project work.

Prabhansh Pratap

IITM, Chennai.

ABSTRACT

KEYWORDS: SHAKTI class of processors; Modules; FPU; TestFloat

The SHAKTI I-class processor is an open-source processor, based on the RISC-V ISA, developed using BSV (Bluespec System Verilog). I-Class is a 64bit superscalar out-of-order (OoO) processor. FPU of the I-CLASS processor is not fully designed. Some modules need optimization because they don't satisfy frequency constraints. Floating Point division is an integral part of FPU and in this project work it is supposed to design using the SRT Radix-4 division algorithm for mantissa part of Floating point number .
So we want to use here integer divider for mantissa part of Floating point operands.
Integer divider is already developed. Modules were tested against few directed test cases. While designing FPU's main challenging test is some modules are single cycle and some are multi-cycles. So it was very hard to use them without any conflict.

In this work, we are working on to develop Floating point divider using Bluespec, the Floating Point Unit for I-class, and perform verification of all modules of the FBOX. This Floating point divider should work for both single precision and double precision Floating point. According to the operation, it calls the appropriate module. If some modules need only one operand then the rest of the two operands are set to zero in the input. There might be conflict while calling modules from FPU because modules take the variable number of clock cycles. So FPU should get instructions in a conflict-free manner. For verification testbench modules are written BSV. Testcases for verification can be generated by TestFloat.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

SHAKTI is an open-source initiative based on RISC-V ISA to build open-source production grade processors, complete System on Chips (SoCs), development boards.

Over the past few years, the work in the IEEE floating-point standard that needs a full accuracy division has developed a great curiosity in the fast division algorithms that are appropriate in VLSI microprocessor circuits. The application of these high-speed division algorithms is in computer graphics applications, cryptography, and digital signal processing. The main problem with high-speed division has been that every bit of the quotient is found out by the carrying out a bit from a full subtraction or addition carry out on the n bit divisor and partial remainder conventionally. The n bit divider has a complexity of O(n2). Verification is a process carried out to find out bugs in FBOX and fix them if any. As we know a buggy module is useless. Test cases are generated by TestFloat by using QEMU as an emulator. FPU accepts a decoded instruction and based on operation return the result.

This chapter explains the SHAKTI I-class processor and the Floating Point Unit, motivation, the contribution of this work, and the organization of the rest of the report.

1.1 Processors

SHAKTI class of processors have been broadly categorized into “Base Processors”, “Multi-Core Processors” and “Experimental Processors”. The “Base Processors” category consists of E-class, C-class, and I-class processors.

The processor design is open-sourced under the BSD-3 License. A brief overview of the I-CLASS of the processor is described below.

1.1.1 I-class:

The I-class is a 64-bit processor that targets the computing, mobile, storage, and net-working platforms. Its features include out-of-order execution, multithreading, aggressive branch prediction, non-blocking caches, and deep pipeline stages. The operational clock frequency of this processor is 1.5-2.5 GHz. SHAKTI I-CLASS not yet fabricated. Some issues are arising like timing constraints in FPU, functional correctness of some modules, a large number of clock cycles for some modules. Some of these issues are resolved and some are yet to be resolved.

I	Base Integer Instruction Set
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
C	Standard Extension for Compressed Instructions

Table 1.1: RISC-V ISA extensions in SHAKTI

1.2 Motivation

The Floating Point Unit is a very important part of any processor. SHAKTI I-CLASS is using FBOX for Floating-Point operations. So main motto of this project has been enhancing the performance during Floating point operation.

- Optimize modules so that the critical path takes lesser time and high frequency can be achieved.
- Working on improving the latency/frequency of Floating Point divider
- Working on improving the latency/frequency of Floating Point square root
- Rewriting simpler modules to increase performance, timing analysis and showing improvements on latency and frequency compared to the current modules in Fbox.

1.3 Organization of the Report

The rest of the report is organized as follows. Chapter 2 states the background, where integer divider, Floating point presentation and FBOX are explained. Chapter 3 presents the details of the implementation of the integer divider, process of floating point division and FPU DESIGN. Chapter 4 concludes my work with possible future work.

CHAPTER 2

BACKGROUND

In this chapter, we explore the necessary background of implementation. As we are using integer division algorithm for mantissa, which is the main operation of Floating point division .So here we understand the present implementation of integer divider (in section 2.1) and verification of FBOX (in section2.2). Later we will discuss about Floating point division representation and then will discuss FPU for SHAKTI I-CLASS processor (in sectoin2.3).

2.1 Integer Divider

Integer divider takes two input and produce one output. Output of divider may be defined as: $D = M*q + R$. Here

- D is Dividend
- M is divisor
- q is quotient
- and R is remainder

And R should satisfy following condition $0 R < M$.

The division is an iterative process in each iteration if the dividend is greater than the divisor then it is subtracted from the dividend. And iterations stop when $R < M$. In each iteration of division one quotient digit is decided. In the implementation section, we will discuss the implementation of integer division algorithms.

2.2 FBOX verification

FBOX have modules to perform floating-point operations. A module, like an object in Object-Oriented languages, has a well-defined interface. FBOX modules are not fully verified for functional correctness. Some modules have bugs. At present FBOX modules are used in the SHAKTI I-CLASS processor. Later after full verification, they may be used in other classes as well. Verification involves the functional correctness of the module. Verification of FBOX is done by writing testbench modules in BSV and test cases are generated by TestFloat.

2.3 FPU for SHAKTI I-CLASS

In the early days' none of the processors had a built-in floating-point capability. If you wanted floating-point processing, you emulated it with software. Needless to say, this was slow and it was difficult for the average programmer! Eventually, it was decided that there should be hardware floating-point processing. So it was decided to build a co-processor (also called a floating-point unit, or FPU) to do the floating-point instructions. For the SHAKTI I-CLASS processor, an FPU unit is required that performs floating-point operations.

2.4 Floating Point Representation

In A number representation specifies some way of encoding a number usually a string of digits. Floating point representation is similar to scientific notation. Logically, a floating-point number consists of:

- A signed (meaning positive or negative) digit string of a given length in a given base (or radix). This digit string is referred to as the significand *mantissa*, or *coefficient*. The length of the significand determines the *precision* to which numbers can be represented. The radix point position is assumed always to be somewhere within the significand—often just after or just before the most significant digit, or to the right of the rightmost (least significant) digit. This article generally follows the convention that the radix point is set just after the most significant (leftmost) digit.
- A signed integer exponent (also referred to as the *characteristic*, or *scale*), which modifies the magnitude of the number.

To derive the value of floating-point number, the signified is multiplied by the base raised to the power of exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent to the right if the exponent is positive or to the left if the exponent is negative.

A floating-point number is a rational number, because it can be represented as one integer divided by another. The way in which the significand (including its sign) and expected are stored in a computer is implementation dependent.

IEEE 754: Floating Point in modern computers

In The IEEE standardized the computer representation for binary floating-point numbers in IEEE-754 in 1985. This first standard is followed by almost all modern machines. It was revised in 2008.

Here we are using two widely used formats for Floating point divider.

- Single precision (binary 32), usually used to represent the float type in the C language. This is binary format that occupies 32 bits (4 bytes) and its significand has a precision of 24 bits (about 7 decimals digits).

- Double precision (binary 64), usually used to represent the double type in the C language family. This is a binary format that occupies 64 bits (8 bytes) and its significand has a precision of 53 bits (about 16 decimal digits).

Type	**Sign**	**Exponent**	**Significand field**	**Total bits**	**Exponent bias**	**Bits precision**
Half (IEEE 754-2008)	1	5	10	16	15	11
Single	1	8	23	32	127	24
Double	1	11	52	64	1023	53

Table: 1.2 IEEE-754 Floating point representation

While the exponent can be positive or negative, in binary formats it is stored as an unsigned number that has a fixed "bias" added to it. Values of all 0s in this field are reserved for the zeros and subnormal numbers; values of all 1s are reserved for the infinities and NaNs. The exponent range for normalized numbers is [−126, 127] for single precision, [−1022, 1023] for double. Normalized numbers exclude subnormal values, zeros, infinities, and NaNs.

CHAPTER 3

IMPLEMENTATION

In this section, we will understand the implementation of three present integer dividers (Non-restoring, SRT radix-2, and SRT radix-4), division algorithms of these methods. So that the best of it can be used for floating point division (for mantissa part of operands). Then the Floating point division process using BSV(Bluespec System Verilog) with testbench related to it Then we will look at the FPU of I-class.

3.1 Integer Divider

As we know integer divider takes two operands (dividend and divisor) and returns either quotient or remainder. The result is quotient or the remainder depends on the type of instruction. Input is always 64bit number and the result is also 64bit number. Dividers are implemented using BSV.

3.1.1 Non-restoring Divider

It is the simplest algorithm for division. It generates one quotient bit at a time based on a sign bit of partial remainder. Following is an algorithm for non-restoring division:

Flow Diagram

Figure 3.1 shows flow diagram of non-restoring division. Diagram shows there are only two comparison operation, one shift operation and one add/sub operation. Register ’A’ contains final remainder, if ’A’ is negative then make it positive by adding divisor after all steps.

Implementation:

- 1st clock cycle

Algorithm 1 Non Restoring Division Algorithm

Input: Dividend (64bit integer number), Divisor (64bit integer number), opcode and funct3

Output: Quotient or Remainder (64bit integer number)

```
 1: procedure START
 2:     Initialization : Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend
 3:     while N > 0 do
 4:         if A < 0 then
 5:             Shift left AQ.
 6:             A=A+M
 7:         else
 8:             Shift left AQ.
 9:             A=A-M
10:         end if
11:         if A < 0 then
12:             Q[0] = 0
13:         else
14:             Q[1] = 1
15:         end if
16:         N=N-1
17:     end while
18:     if A < 0 then
19:         A=A+M
20:     end if
21:     Register Q contain quotient and A contain remainder
22: end procedure
```

  - – Put the dividend and divisor into the registers for next clock cycle.
  - – Name of operation contains two information: 1st type of division (signed or unsigned) and 2nd what to return (quotient or remainder). Put the both information into the registers.
  - – If inputs are 32 bits operands then sign bits are padded.
  - – Special case (divide by zero, sign overflow...) checked.

- 2nd clock cycle
  - – In 2nd clock cycle if division is signed and any operand is negative then its 2’s complement is taken. After their complement, put the dividend and divisor into the registers so later cycles can use them. For signed division sign of the result decided in the same clock cycle.
- Next 65 clock cycles

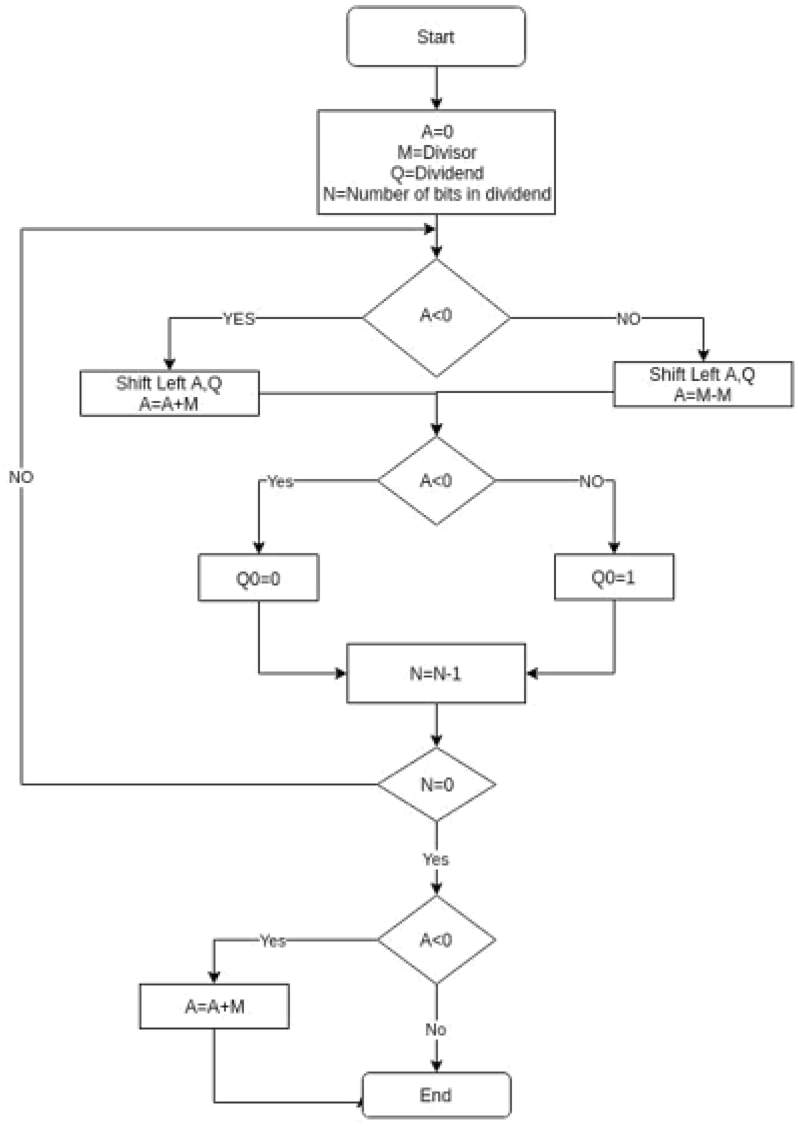


Figure 3.1: Flow diagram

– Step 2 to 5 of the non-restoring algorithm are performed. In each clock cycle, partial remainder and divisor are taken from register manipulated and put back into the registers.

• Last clock cycle

– Returns result. According to the type of instruction, the result is decided. It may be quotient or remainder and further negative or positive.

– If the result is negative, then 2’s complement is taken.

3.1.2 SRT radix-2

Idea:

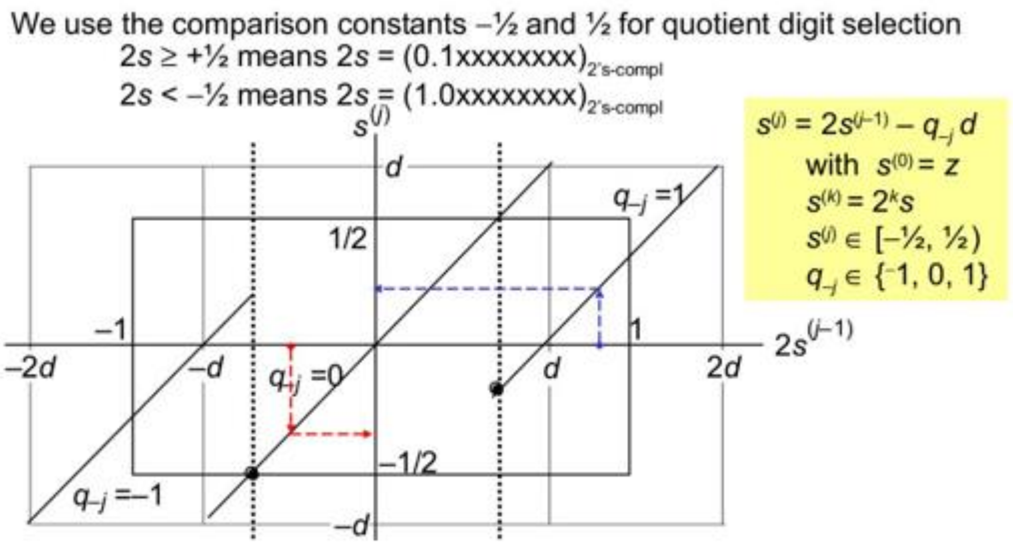
In SRT radix-2 divider initially start with dividend which is treated as partial remainder. And in each iteration, one quotient bit is generated. Choose quotient bit such that partial remainder remains in some particular range (range depends on base).

Quotient digit selection:

$s^{(j-1)}$ - partial remainder at $(j-1)^{th}$ step

$s_{(j)}$ - partial remainder at j_{th} step

q_{-j} - quotient digit at jth step. The quotient selection logic diagram shows quotient digit



We use the comparison constants –½ and ½ for quotient digit selection
2s ≥ +½ means 2s = (0.1xxxxxxxx)2's-compl
2s < –½ means 2s = (1.0xxxxxxxx)2's-compl
s(j)
d
1/2
–1
1
–2d
–d
d
2d
2s(j–1)
q–j =1
q–j =0
q–j =–1
–1/2
–d
s(j) = 2s(j–1) – q–j d
with s(0) = z
s(k) = 2ks
s(j) ∈ [–½, ½)
q–j ∈ {-1, 0, 1}

Figure 3.2: Quotient selection logic[1]

Selection for fractional numbers, but the same logic can be applied for integer division.

In SRT radix-2 divider in each iteration quotient bit is selected based on two MSBs.

If two MSBs are 10 at j^{th} step $q_{-j} = -1$.

If two MSBs are 01 at j^{th} step $q_{-j} = 1$.

If two MSBs are 11 or 00 at j^{th} step $q_{-j} = 0$.

Implementation:

This divider takes the variable number of clock cycles. The number of clock cycles depends on the difference between the number of leading zeros in divisor and dividend. If the difference is negative, then it takes 2 clock cycles only, because the dividend becomes less than the divisor. For using this divider in the pipeline processor, one wrapper is added so that every division takes the fixed number of clock cycles.

- 1st clock cycle:
  - Put the dividend and divisor into registers for the next clock cycle.

Algorithm 2 SRT radix-2 Algorithm

Input: Dividend (64bit integer number), Divisor (64bit integer number), opcode and funct3

Output: Quotient or Remainder (64bit integer number)

```
 1: procedure START
 2: Initialization: S = dividend, d = divisor, q_pos = 0, q_neg = 0, n = number
    of bits in dividend.
 3:     while n > 0 do
 4:         Generate q(the quotient bit)
 5:         S = 2*S - d*q
 6:         shift left q_pos and q_neg by one
 7:         if q == -1 then
 8:             q_neg = q_neg + 1
 9:         else
10:             if q == 0 then
11:                 q_pos = q_pos + 1
12:             end if
13:         end if
14:         n = n-1
15:         q_pos = q_pos + 1
16:     end while
17:     S contain remainder and Quotient = (q_pos - q_neg)
18: end procedure
```

  - The name of operation contains two information 1st type of division (signed or unsigned) and 2nd what to return (quotient or remainder). Put both into the registers.
  - If inputs are 32 bits operands then sign bits are padded.
  - Special case (divide by zero, sign overflow.) checked.

- 2nd clock cycle:
  - – For signed division complement of negative number is taken.
  - – Sign information put into the register so that while returning, the result for negative complement can be taken of the result.
  - – Initially dividend considered as partial remainder. Assume it is s_0 initially and at later cycles it is s_j in jth cycle.
- Next 63 clock cycles
  - – In next 63 clock cycles one quotient bit is decided in each cycle. And quotient bit may be {−1, 0, 1}. Assume this bit is q_{-j}.
  - – For next cycle (j+1) partial remainder becomes $s_{j+1}=s_j*2 - q_{-j}*$divisor.
- Last cycle
  - – According to the type of instruction, the remainder or quotient is returned.
  - – For signed division, if the result is negative, then its 2's complement is taken.

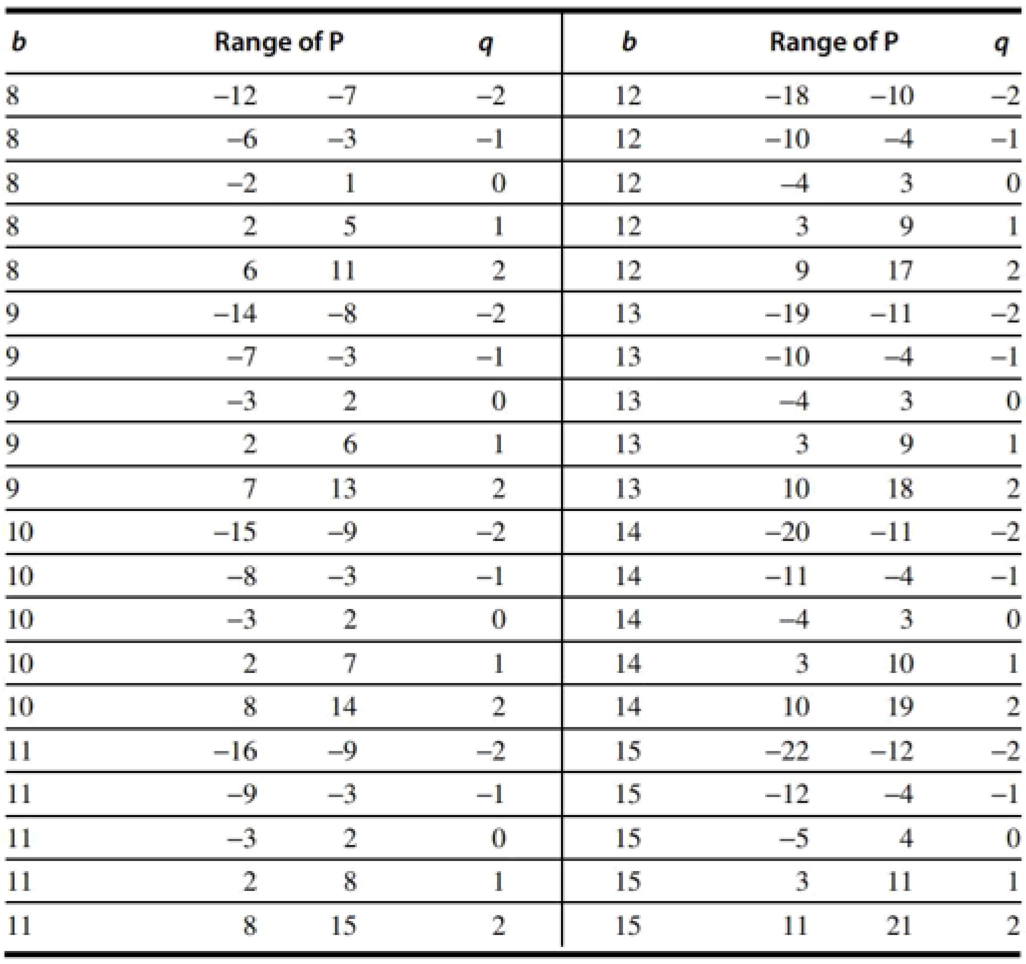
3.1.3 SRT radix-4

Idea:

The idea of SRT radix-4 is similar to SRT radix-2 the only difference is here 2 bits are generated in each cycle. More number of quotient bits are generated so that more number of bits are involved in quotient digit selection logic.

Quotient selection table

In table for Figure 3.3 'b' is integer value of four MSBs of divisor, 'P' is integer value of six MSBs of partial remainder and q is quotient digit.



b	Range of P		*q*	*b*	Range of P		*q*
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

Figure 3.3: Quotient selection table [2]

Implementation:

- 1st clock cycle
  - – Put the dividend and divisor into the registers for next clock cycle.
  - – Name of the operation contains two information: first, type of division (signed or unsigned) and second, what to return (quotient or remainder). Both information are put into registers.
  - – If the inputs are 32 bits operands then the sign bits are padded.
  - – Special case (divide by zero, sign overflow.) checked.
  - – Take two 64 bits register rg_q_pos and rg_q_neg to hold quotient digits.
- 2nd clock cycle
  - – For signed division if any input is negative then take its 2’s complement.

Algorithm 3 SRT radix-4 Algorithm

Input: Dividend (64bit integer number), Divisor (64bit integer number), opcode and funct3

Output: Quotient or Remainder (64bit integer number)

```
 1: procedure START
 2: Initialization (rg_p_a = dividend, d = divisor, rg_q_pos, rg_q_neg, n = 0).
    rg_p_a and d are both 64 bits registers.
 3:     Left shift d by 64.
 4:     Count number of leading zeros in divisor and put in to c.
 5:     Left shift rg_p_a and d by c.
 6:     while n < 32 do
 7:         Select quotient digit q by using quotient digit selection table.
 8:         Left shift rg_p_a, rg_q_pos and rg_q_neg by 2.
 9:         rg_p_a = rg_p_a - q * d.
10:         if q < 0 then
11:             rg_q_neg = rg_q_neg + a
12:         else
13:             if q > 0 then
14:                 rg_q_pos = rg_q_pos + q
15:             end if
16:         end if
17:         n = n+1.
18:     end while
19: (rg_q_pos - rg_q_neg) have quotient. Left shift rg_p_a by c and it will be
    remainder.
20: end procedure
```

  – Sign of the final answer decided.

- 3rd clock cycle
  – Count number of leading zeros in the divisor and put into some register.
- 4th clock cycle
  – Take a register rg_p_a of size double of dividend length, and put dividend into it. rg_p_a contain partial remainder.

  – Left shift rg_p_a and divisor by number of leading zero count of divisor.
- Next 32 cycles
  – Left shift q_pos, q_neg and rg_p_a by 2;

  – Quotient digit is decided by using table of Figure 3(assume it q). If quotient digit is negative then rg_q_neg = rg_q_neg + q. It it is positive then rg_q_neg = rg_q_neg + q.

  – rg_p_a = rg_p_a - divisor * q.
- Last 3 clock cycles
  – Shift back partial remainder.

  – If result is negative then take it’s 2’s complement.

Basically radix-4 takes 32 clock cycles but some initial initial and final work has to be done. So it takes to 39 clock cycles.

3.2 Floating Point Divider

As we know integer divider takes two operands (dividend and divisor) and returns either quotient or remainder. The result is quotient or the remainder depends on the type of instruction. This same algorithm can be used for floating-point divider. As SRT radix-4 method is fastest among all three integer divider (Non-restoring, SRT radix-2, and SRT radix-4) described here. So the SRT Radix-4 method has been used for the mantissa of the given two operands.

Which is the main part of floating-point division, then taking the difference of exponent of the operands provides the exponent of quotient and XNOR the sign bit of the operands gives sign bit of the resultant quotient.

3.2.1 Writing BSV testbench

Testbench have mainly three things: reading input from testcase file, calling appropriate module and check whether module output is same as expected output or not. For writing BSV testbench module we have to understand interface of modules. Interface of modules of FBOX is as follow:

– Input

op1 : 32bit for single precision and 64bit for double precision of type Floating Point

op2 : 32bit for single precision and 64bit for double precision of type Floating Point

Rounding mode: 3bit for founding mode

– Output valid: one bit information for output is valid or invalid.

value: 32bit for single precision and 64bit for double precision result of Type Floating Point

ex: three bit information for exception flags.

3.3 FPU for I-class

Stands for "Floating Point Unit." An FPU is a processor or part of a processor that performs floating-point calculations. An FPU provides a faster way to handle calculations with non-integer numbers. Any mathematical operation, such as addition, subtraction, multiplication, or division can be performed by FPU. FPU for SHAKTI I-CLASS is shown in figure 3.4. FPU calls modules of FBOX. Modules of FBOX can execute operations in a pipelining manner, while FPU is processing some input at that time it can accept new input. Modules takes the different number of clock cycles. So before feeding instruction to FPU, there must be proper scheduling, because there may be a conflict when results are produced.

- Input for FPU: op1, op2, and op3 are three input operands for computation. ’operations’ is of type Fpu_type it is a simple operation. ’funct3’ is rounding mode .’use_data’ is of type Usr_data, it has current_rob_id, dest_tag, and pc.
- Output: FPU provides results in each clock cycle. If some valid result is available then return it with the valid bit as ’1’ otherwise return ’0’ with the valid bit as ’0’. ’ans’ is actual output. ’flags’ are exceptional flags. ’use_data’ is the same thing as input.

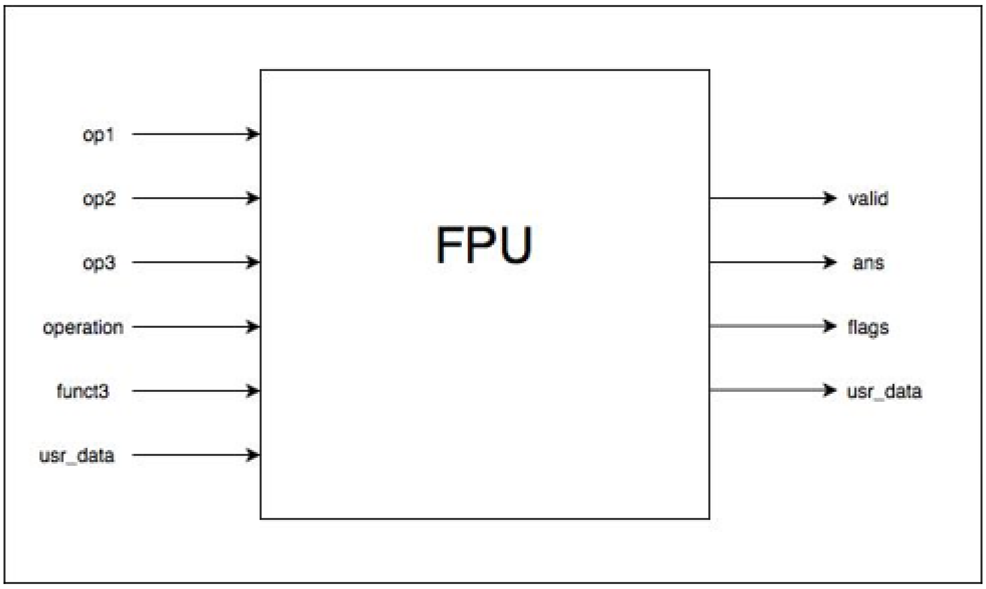


Figure 3.4: FPU

CHAPTER 4

CONCLUSION AND FUTURE SCOPE

4.1 Summary of Work Done

SHAKTI is an open-source work based on RISC-V ISA. This work revolves around Floating Point operations. I have studied the three integer dividers non-restoring, SRT radix-2, and SRT radix-4 . Among all three dividers, SRT radix-4 takes fewer clock cycles but its implementation is difficult. As SRT Radix-4 integer divider is the fastest among all the three divider, So using the SRT radix-4 integer divider algorithm for division of mantissa of Floating-point operands using the BSV. Presently it has some bugs and we are working on it. The Floating-point divider can be modified for the square root of floating point in future.

APPENDIX

TOOL INSTALLATION

A.1 Installation of Bluespec Compiler

An open-source version of the Bluespec Compiler is available online [4]. By in-stalling the open-source Bluespec compiler, one will be able to generate the synthesiz-able Verilog compatible with FPGA targets.

- Open a new terminal and move to the Home folder. Copy-paste the below com-mands in the terminal and press enter.

```
  sudo apt install ghc libghc-regex-compat-dev libghc-syb-
dev iverilog
```

```
sudo apt install libghc-old-time-dev libfontconfig1-dev libx11-dev
```

```
sudo apt install libghc-split-dev libxft-dev flex bison libxft-dev
```

```
sudo apt install tcl-dev tk-dev libfontconfig1-dev libx11-dev gperf
```

```
sudo apt install itcl3-dev itk3-dev autoconf git
```

- Download the repository

```
git clone --recursive https://github.com/B-Lang-
org/bsc cd bsc
make PREFIX=/path/to/installation/folder
```

- After you have done the above steps, add the path you have installed the bsc compiler to your $PATH in the bashrc or cshrc.

```
export PATH=$PATH:/path/to/installation/folder/bin
```

A.2 Setup for test cases generation

TestFloat and QEMU Both are available on SHAKTI tools.

```
mkdir /sw_tools
sudo mount 192.168.1.112:/home/rise/sw_tools
/sw_tools source /sw_tools/qemu.sh
sudo mount 192.168.1.11:/scratch/gitlab-builds/releases /scratch/bb-
releases/ cqemu-riscv64 /scratch/bb-releases/common-
verif/testfloat/testfloat_gen [round-ing_mode] [function] > input.txt
```

REFERENCES

[1] Behrooz Parhami. Computer Arithmetic.

[2] David Goldberg, Xerox Palo Alto Research Centre. Computer Arithmetic. Appendix J.

[3] SHAKTI Processor Program Open-source Processor Development Ecosystem, https://shakti.org.in.

[4] Bluespec Compiler, https://github.com/B-Lang-org/bsc.