Designing an IEEE Floating-Point Unit with Configurable Compliance Support and Precision for FPGA-Based Soft-Processors

by

Yuhui Gao

B.A.Sc., The Pennsylvania State University, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY

Fall 2022

Declaration of Committee

Name:	**Yuhui Gao**
Degree:	**Master of Applied Science (Computer Engineering)**
Title:	**Designing an IEEE Floating-Point Unit with Configurable Compliance Support and Precision for FPGA-Based Soft-Processors**
Committee:	**Chair:** Andrew Rawicz Professor, Engineering Science
	Lesley Shannon Supervisor Professor, Engineering Science
	Zhenman Fang Committee Member Assistant Professor, Engineering Science
	Mieszko Lis Examiner Associate Professor, Electrical and Computer Engineering University of British Columbia

Abstract

Field Programmable Gate Arrays (FPGAs) are commonly used to accelerate floating-point applications. The advancements in FPGA technology and the introduction of the RISC-V Instruction Set Architecture (ISA) have collectively enabled a number of soft-processor designs. Although researchers have extensively studied FPGA-based floating-point implementations, existing work has largely focused on standalone, and frequency-optimized data-path designs. They are not suitable for soft-processors targeting FPGAs due to the units' long latency, and soft-processors' innate frequency ceiling. Furthermore, the few existing integrated Floating Point Unit (FPU) hardware implementations targeting FPGA-based soft-processors are not IEEE 754 compliant. We present a floating-point unit for FPGA-based RISC-V soft-processors that is fully IEEE compliant and configurable. Our design focuses on maximizing runtime performance with efficient resource utilization. We allow the users to configure the FPU to four varying levels of compliance, or to select reduced precision configurations. Benchmarking against a set of real-world floating-point applications, we evaluate the FPU variants in term of resource usage, operating frequency, runtime performance, and performance efficiency. We also present trade-off analyses of two microarchitecture design choices.

Our fully compliant FPU uses 5423 Look-Up Tables (LUTs), and achieves an operating frequency of 105 MHz. The key results from our work demonstrate the effect of running floating-point workloads using reduced compliance FPUs. Our experimentation shows that decreasing the Fused Multiply-Add (FMA)'s intermediate representation leads to a 25% reduction in LUT usage that translates to an average 46% increase in performance-efficiency. Additionally, disabling denormal support reduces the resource utilization by 10% and improves the clock frequency by 6%, which results in a 14% higher performance efficiency, while having no impact on the result accuracy for our benchmark applications. Furthermore, we find that running applications in reduced precision can improve runtime performance by up to 75%, although applications may suffer from significant loss of precision.

Keywords: FPGA; Floating-Point; Computer Architecture; Soft-Processor

Acknowledgements

I could not have undertaken this journey without the help of many. I would like to thank my supervisor, Dr. Lesley Shannon, for her invaluable guidance and patience. Your mentorship was essential in completing this thesis. Additionally, I would like to express my deep gratitude to Eric. Thank you for always going out of your way to help and share your wealth of knowledge.

Words cannot express my gratitude to my parents and family, whose continuous support and sacrifice made graduate school possible. I am also grateful to my friends and my partner, Drina, for their support and abundant words of encouragement. Lastly, I would be remiss in not mentioning the people of the Reconfigurable Computing Lab. From the Friday Catan games to the Zoom bantering sessions, your comradery is greatly appreciated.

Table of Contents

List of Tables

List of Figures

Glossary

ALU Arithmetic Logic Unit. 57

ASIC Application Specific Integrated Circuit. v, 2, 11, 15, 80

BRAM Block-RAM. x, 4, 5, 12, 44, 53, 55–57, 76

CLB Configurable Logic Block. x, 4, 5

CSR Control and Status Register. 52

DP Double-Precision. x, xi, 6, 9, 12–14, 22, 29, 36, 51, 59, 68

DSP Digital Signal Processing. ix, x, 4, 5, 12, 13, 44, 53, 55–57, 66, 68, 76

FCSR Floating-Point Control Status Register. 9, 19, 20

FEQ Floating-Point Equal to. 39

FF Flip-Flop. x, 4, 5, 24, 44, 55–58, 64, 68, 69, 75, 76, 80

FIFO First-In First-Out. xi, 25, 26, 32, 34, 35, 49, 50, 76

FLE Floating-Point Less than or Equal to. 39

FLT Floating-Point Less than. 39

FMA Fused Multiply-Add. iii, vi–viii, x, xi, 2, 3, 9, 10, 14, 19, 24, 25, 28–33, 43–46, 48–51, 55, 56, 59, 60, 62–64, 76, 79–81

FPGA Field Programmable Gate Array. iii, vi, x, 1–5, 10–16, 53, 55, 58, 68, 75, 80, 81

FPU Floating Point Unit. iii, v, vi, viii, ix, xi, xii, 1–5, 10–18, 20, 21, 28, 29, 33, 40, 43, 44, 47–53, 55–58, 60, 62–64, 68–70, 73–76, 80, 81

G Guard Bits. 8, 42, 43

HDL Hardware Description Language. 13, 14

HLS High-Level Synthesis. 13

ILP Instruction Level Parallelism. 10, 12, 17

IO Input-Output. x, 4

IPC Instructions Per Cycle. ix, xi, xii, 12, 28, 29, 40, 52, 53, 61–64, 71–75, 77–81

ISA Instruction Set Architecture. iii, 1, 2, 5, 9, 13, 14, 16, 18, 20, 52

LE Logic Element. 13

LSB Least-Significant-Bit. 8, 42

LUT Look-Up Table. iii, x–xii, 1, 2, 4–6, 11–14, 24, 40, 44, 45, 53–58, 61, 62, 64, 68, 69, 74–81

MAC Multiply-Accumulate. 12

MIPS Millions of Instructions Per Second. xi, xii, 53, 54, 61–64, 73–75, 77–80

NaN Not-a-Number. 6, 39

OOO Out-of-Order. 10

R Round Bit. 8, 42, 43

RAS Return Address Stack. 47

RTL Register Transfer Language. 13

S Sticky Bit. 8, 42, 43

SP Single-Precision. x, 6, 7, 9, 12–14

UART Universal Asynchronous Receiver/Transmitter. 53

WB2FP Floating-Point Units that Write-back to Floating-Point Register. vi, 23, 24, 27–29, 36, 38

WB2INT Floating-Point Units that Write-back to Integer Register. vi, 23, 24, 27, 28, 38

1 Introduction

Floating-point computation is ubiquitous in the modern digital world. It is used in a wide range of applications, from automotive [4], to communications [5], to graphics [6]. Despite the introduction of various custom floating-point standards [7] [8] [9] for domain specific applications, IEEE 754 [10] continues to be the most popular floating-point format since its inception in 1985. However, due to the complex nature of floating-point operations, the inclusion of a hardware Floating Point Unit (FPU) is often cost prohibitive compared to integer-only platforms. In many embedded environments, engineers often choose to emulate floating-point computations using software libraries such as softfloat [11], resulting in significant performance degradation. Table 1.1 summarizes the design tradeoffs between IEEE 754 hardware FPU and software emulation. Software emulation of floating-point is slower than hardware floating-point, but offers more flexibility, because it enables floating-point arithmetic on FPU-less platforms and multi-precision support [12] [13]. Hardware floating-point is more costly to design and verify, both in terms of resources (gates/Look-Up Tables (LUTs)), and engineering complexity.

Table 1.1: Tradeoff Between Software Emulation vs. Hardware FPU

Implementation	Speed	Flexibility	Cost
Soft-Float	Low	High	Low
Hard-Float	High	Low	High

IEEE 754 FPUs can be implemented on Field Programmable Gate Arrays (FPGAs). However, historically, hardware FPUs' resource requirements and complexity have led to very few implementations. Instead, the focus has been on implementing specific floating-point operations [14] [15] [16]. With today's high-capacity FPGAs, the size of an FPU is no longer the limiting factor it once was. Additionally, FPGAs' reprogrammability is ideally suited for designing adaptable floating-point hardware to satisfy varying application requirements.

Furthermore, the larger size of modern FPGAs and the recent rise in popularity of RISC-V have collectively enabled a number of soft-processor cores [3] [17] [18] [19]. Designed with the FPGA's flexibility in mind, these soft-processors often offer support the integration of custom accelerators. For applications that require floating-point operations, the inclusion of a hardware FPU would increase the runtime performance substantially. However, we are aware of only one fully-compliant IEEE 754 FPU design for FPGA-based soft-processors. Due to performance/resource constraints, existing integrated FPU implementations [17] [20] [21] often omit specific data-path features and exclude denormal number and non-default rounding mode support. Moreover, implementing and optimizing FPUs for existing soft-processor architectures is extremely complex. In this thesis, we create an easily-integrated FPU targeting the RISC-V Instruction Set Architecture (ISA) with user customizable compliance and precision levels. Furthermore, we investigate and analyze the design tradeoffs, including supporting IEEE 754 compliance features, reduced precision configurations, and two microarchitectural variations.

1.1 Motivation

Researchers have explored various optimization techniques for floating-point designs. Bertaccini et al. [22] experiments with extreme resource-saving, multi-cycle Application Specific Integrated Circuit (ASIC) FPU designs. Hockert et al. [23] implements an FPGA-based, resource-optimized FPU by selectively accelerating the soft-float [11] instructions in hardware, leaving others to emulation. Emulation has been a common solution to unavoidable floating-point workloads for integer-only systems. However, emulation can be 10x slower [24] than dedicated hardware. While having use cases for heavily resource limited systems, the resource-optimized implementations under-utilize the capabilities of modern FPGAs, and leave significant performance potential unused. For example, the FPU design proposed by Hockert et al. [23] is 9x slower than a performance-optimized FPU, while saving only 23% in LUT.

In addition, existing work is not suitable for integration to soft-processors. Much research effort has focused on optimizing the frequency of standalone floating-point function(s) implementations. This design methodology can lead to deeply-pipelined (e.g. 9 cycle for FADD [14]) and high clock frequency designs. Moreover, as standalone designs are unaware of their surroundings, they can be independently resource efficient, but are unable to share hardware with adjacent modules.

Implementing integrated FPUs requires a different approach. In a processor system, the floating-point components' maximum operating frequency is dictated by that of the base processor. Unfortunately, FPGA-based soft-processors typically achieve much lower frequencies than that of standalone, highly-pipelined data-paths. Furthermore, deeply-pipelined data-paths in a processor environment can induce long processor stalls, as dependencies inevitably arise — especially since the amount of instruction-level parallelism that compilers can extract is limited. Standalone data-paths are often resource intensive, and lack the scope of the overall system that can lead to further resource optimizations. As such, FPGA-based FPU designs for soft-processors should focus on minimizing latency and resource usage. The latter often comes with the added benefit of reduced routing congestion on FPGAs, which can lead to higher operating frequency. Moreover, integrated FPU designs enable additional resource sharing optimizations with adjacent hardware in a processor environment that are not available in standalone data-paths.

Furthermore, floating-point designs targeting FPGAs have the unique advantage of tailoring their hardware to the applications' constraints, including varying compliance and precision levels. As a result, existing FPGA-based, standalone [16] [25] and integrated [26] [21], floating-point implementations only partially support the IEEE 754 compliance features, with few exceptions [14] [27]. These academic and commercial implementations often omit compliance features including denormal support and full rounding modes. Moreover, the RISC-V [1] ISA mandates the inclusion of Fused Multiply-Add (FMA) instructions, which are resource intensive if full compliance is required (elaborated in Section 3.3.2). Therefore, existing RISC-V based FMA implementations are either partially compliant [28], deeply-pipelined [27], or multi-cycle [22]. In addition, FPGA-based FPUs enable users to run their applications in reduced precision to improve performance efficiency. It is expected that executing floating-point workloads in reduced compliance level or reduced precision may result in a degraded quality of results. Moreover, since most floating-point applications are written to be IEEE 754 compliant, software standards must be strictly upheld even if the hardware is not fully compliant. This

requires implementation considerations where the familiar software interface is supported, while the underlying hardware is abstracted from the users.

1.2 Objective

The goal of this work is to create an open-source, RISC-V FPU for FPGA-based soft-processors that provides: compile-time configurable compliance support and multi-precision support. Furthermore, the design is optimized for runtime performance and performance efficiency (performance per resource). In the course of implementing the FPU and its non-compliant variations, we wish to facilitate a quantitative discussion on the impact of running floating-point applications in reduced compliance levels and reduced precision. As part of our focus on runtime performance and performance-efficiency, we plan to explore two design variations for the most frequently used instructions, FADD, FMUL, and FMA. We seek to quantify the design tradeoffs among different FPU designs across the same set of metrics, and identify the most optimal set of design choices. The FPU configurations are evaluated in terms of accuracy, runtime performance, and performance-per-resource.

1.3 Contributions

In this work, we introduce an FPGA-optimized, optionally compliant, compile-time multi-precision FPU. We investigate and evaluate four configurations at different compliance levels based on accuracy, runtime performance and performance efficiency. Moreover, by leveraging the reconfigurability of FPGAs, we explore executing floating-point binaries in reduced precision. And lastly, we provide tradeoff analysis of two FPU microarchitectural variations. The contributions of this thesis can be summarized as follows:

- An implementation of a runtime performance and performance-per-resource directed, optionally IEEE 754 compliant, compile-time multi-precision FPU for FPGA-based soft-processors.
- A trade-off analysis of three IEEE 754 compliance features, such as reduced FMAs instruction intermediate representation, denormal support and full rounding modes support.
- A quantitative discussion on the effect of running application in reduced precision, in terms of accuracy, runtime performance, and performance efficiency.
- Comparisons of two FMA microarchitectural design choices.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents the relevant information on FPGAs, the IEEE 754 floating-point standard, and provide an overview of related research. Chapter 3 presents implementation details of the FPU. Chapter 4 covers our experimental environment, including the different FPU configurations, measurement techniques, and our benchmarking methods. The results of our experiments are then presented in Chapter 5. Lastly, Chapter 6 provide a summary of this work along with potential improvements to the FPU and future work.

2 Background

This chapter provides relevant background information for the thesis. We begin with a brief introduction of SRAM-based FPGAs, including the common components and hardware features. We then outline Taiga, the processor architecture with which we our FPU is integrated. Finally, we present an overview of related research.

2.1 FPGA Overview

An FPGA is an integrated circuit that allows users to adapt and program the device to the desired application and/or functionality requirements within the FPGA's resource constraints. As shown in Figure 2.1, an FPGA is made of an array of Configurable Logic Blocks (CLBs) that implement logic functions. They are connected via a hierarchy of programmable routing interconnects that interface each CLB with other CLBs, Input-Output (IO) signals, and various compute and memory components such as Digital Signal Processings (DSPs) and Block-RAMs (BRAMs).

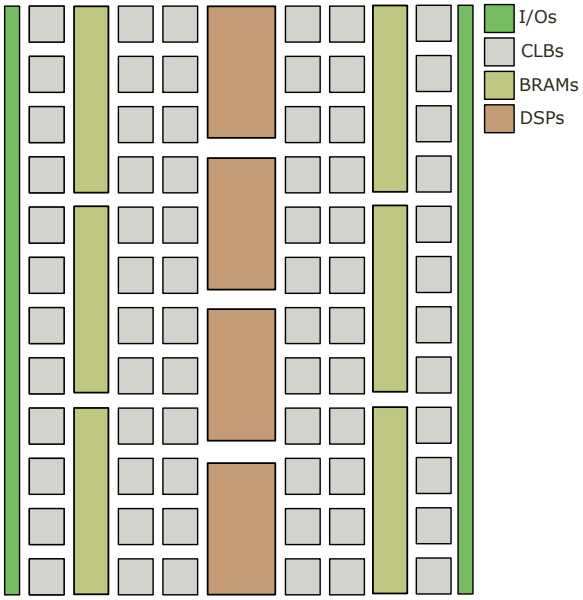

Figure 2.1: FPGAs consists of IOs, CLBs, BRAMs and DSPs

Logic blocks are the most abundant hardware resource in an FPGA. Even though there are differences among FPGA vendors and their respective device families. At the minimum, each CLB typically contains multiple K-input LUTs and storage elements (Flip-Flops (FFs)). Figure 2.2 provides a simplified view of the connections between LUTs, and FFs within the CLBs.

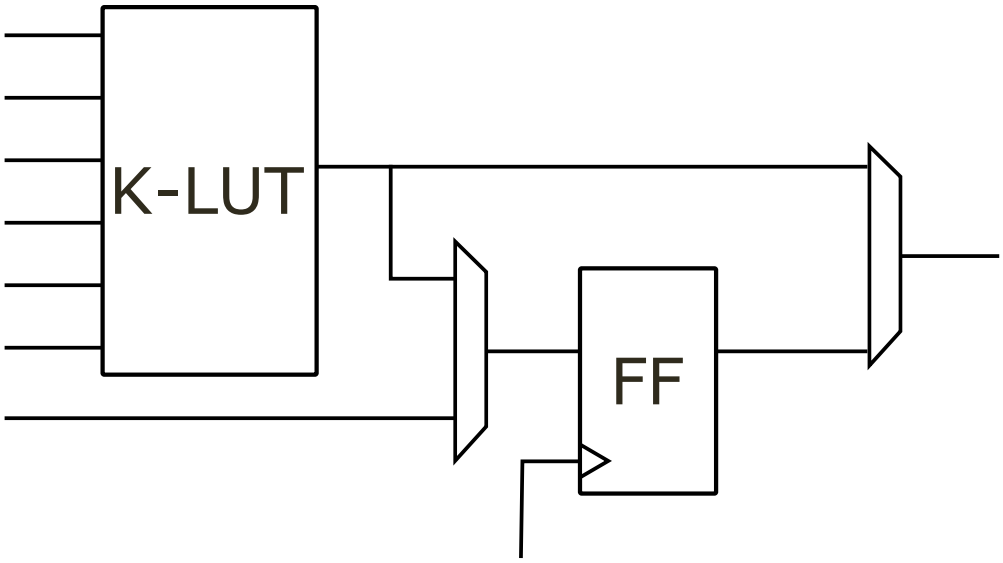

Figure 2.2: A simplified view of LUT and FF elements within an CLB.

As the foundational element of an FPGA, the K-input LUTs within a CLB are the function generators. Each K-input LUT can implement any function of K inputs and one output. Some CLB architectures support fracturing, where the K-input LUTs have two outputs if (K-1) inputs are shared. Wider functions can be created by combining multiple K-input LUTs and multiplexers. Furthermore, FPGA vendors provide dedicated fast carry logic to accelerate arithmetic operations. In addition to the configurable elements, modern FPGAs also include hard logic blocks that implement a set of common functionalities, such as DSPs that enable high-performance implementations of multipliers, and BRAMs for embedded memory. Notably, adjacent DSPs and BRAMs can be cascaded to implement larger arithmetic functions and memory blocks. Both DSPs and BRAMs are organized vertically within the FPGA fabric, as shown in Figure 2.1. Therefore, routing congestion may occur in designs with high utilization of DSPs and BRAMs due to placement constraints.

2.2 Floating-Point Overview

Our FPU targets RISC-V based soft-processors, and the RISC-V ISA specifies its floating-point instructions to be compliant with the IEEE 754 standard. Therefore, this section provides a brief overview of the IEEE 754 floating-point specification [10], including the floating-point format, denormal numbers, and exception and rounding. We also outline the RISC-V ISA's floating-point extensions.

2.2.1 Binary Floating-Point Format

The IEEE 754 floating-point format is conceptually similar to scientific notation. As shown in Figure 2.3, a k-bit floating-point number can be uniquely represented using three variables [10]:

- 1-bit sign S.
- w-bit biased exponent $E = e + bias$, whereas $bias = 2^{w-1} - 1$.

- t-bit trailing mantissa field T.

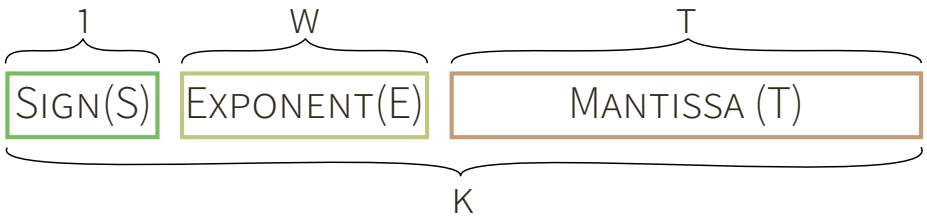

Figure 2.3: IEEE 754 Floating-Point Format: w=8, t=23 for Single-Precision (SP) numbers. w=11, t=52 for Double-Precision (DP) numbers.

The exponent field uses biased representation to mitigate the complexity of handling signed numbers in hardware. The signed exponent values are converted to unsigned values by adding a bias to the actual exponent. Like in scientific notation, the exponent field width *w* controls the range, i.e. the upper and lower limits of the format. The mantissa field width *t* dictates the precision of the representation, and fine-tunes the distance between successive numbers representable by the format. It is common to tradeoff *t* with hardware complexity, as we will discuss in Section 2.2.2. The value of a normal floating-point number encoded in IEEE 754 format can be expressed using:

$$(1)^S * 2^{E-bias} * (1.T), if 1 \leq E \leq 2^w - 2 \tag{2.1}$$

It can be seen that normal floating-point numbers contain an implicit leading one bit in the mantissa field. Moreover, the standard defines additional encodings to represent several special numbers:

- Not-a-Number (NaN): $E = 2^w - 1$ and $T \neq 0$
- $\pm\infty$: $E = 2^w - 1$ and $T = 0$
- Denormal numbers (more in Section 2.2.3): $(-1)^S * 2^{1-bias} * (0.T)$, if $E = 0$ and $T \neq 0$
- ± 0: $E = 0$ and $T = 0$

NaN can be interpreted as undefined floating-point values, or used to propagate specific debug information, i.e. multiply(0,∞). Denormal numbers are the subset of floating-point values that are too small to be representable by a given floating-point format. We provide a detailed discussion on denormal numbers in Section 2.2.3.

2.2.2 Reduced Floating-Point Format

To meet applications' performance, resource and power constraints, designers often choose to implement floating-point using non-standard floating-point formats. Compared to exponent processing, mantissa computation is substantially more complex and resource intensive, due to the large adders, multipliers, and shifters required. Our experiments show that truncating exponent field by 1 bit leads to <1% LUT usage reduction,

whereas decreasing the mantissa width by the same amount results in >4% saving in resource utilization. Therefore, it is common to trade-off improved hardware performance and efficiency with reduced computation precision and accuracy degradation. For example, by simply reducing SP floating-point format's mantissa width to 10 bits, Nvidia's tensor float format [7] provides a substantial 6x speedup in machine-learning applications while maintaining similar computation accuracy. Moreover, in applications where storage size is a bottleneck, such as the motion picture industry, compression algorithms with smaller mantissa widths [8] are employed to decrease the storage requirements.

2.2.3 Underflow and Denormal Floating-Point Numbers

In order to represent numbers with smaller magnitude than that of the smallest normal number, IEEE 754 uses a special format called denormal format. This section presents an overview of the denormal number format and the rationale behind its adoption.

IEEE 754 defines normal floating-point numbers as those floating-point numbers with biased exponent values greater than zero. Normal floating-point numbers have an implicit leading bit of 1, and their mantissa fields can be written as $1.m_0m_1...m_{p-1}$, where m_n are the mantissa digits. The normal number encoding covers the majority of the floating-point numbers. However, there exists a caveat for tiny floating-point values. Figure 2.4 illustrates the phenomenon called *abrupt underflow*. Abrupt underflow occurs when the tiny values around zero are not representable using normal encoding, leaving an abrupt gap between $\pm float_{min}$ and 0. The lack of representation of tiny values can cause various problems. For example:

- $x \pm y$ may underflow, where x and y are two neighboring normal floating-point numbers.
- $x = y$ cannot be safely deduced from $x - y = 0$ for normal, finite x and y.

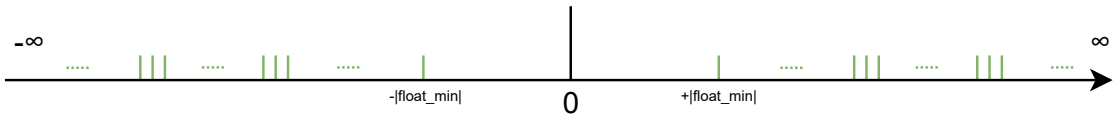

Figure 2.4: Normal Numbers Abruptly Jumps to 0 in Abrupt Underflow

To mitigate the aforementioned issues, IEEE 754 defines a new format specifically for tiny values called denormal. Denormal numbers are the subset of floating-point numbers whose implicit mantissa leading bit is 0. Mathematically, denormal numbers represent the small values around the interval of $(-float_{min}, +float_{min})$, filling the gap between normal numbers and ± 0, shown as the red bars in Figure 2.5. This scheme is also referred to as *gradual underflow*. Gradual underflow enhances software's numerical stability, though it comes at additional cost in hardware complexity and degraded performance. Many processors choose to handle denormal numbers by treating them as zeros (flush-to-zero). Others use software traps [29] which results in reduced runtime performance.

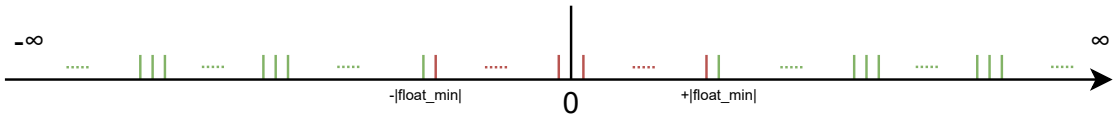

Figure 2.5: Normal Numbers Gradually Converge to 0 in Gradual Underflow

2.2.4 IEEE Rounding

IEEE-754[10] mandates that every operation must be carried out as if the intermediate results are correct to infinite precision and with unbounded range. Rounding is then performed to fit the ”infinitely precise” numbers in the destination’s finite representation. However, ”infinite precision” is physically impossible, therefore implementations only keep a subset of the total precision bits beyond the destination’s format for rounding purposes. Guard Bits (G) are the bits to the right of the least significant bit of the mantissa. Round Bit (R) is the bit to the right of the guard bit, and Sticky Bit (S) is the logical OR of all bits to the right of R.

IEEE 754 defines five rounding-direction attributes:

- **roundTiesToEven** (default mode): store the floating-point number nearest to the infinitely precise result; if two floating-point numbers are equally near, choose the one with the even Least-Significant-Bit (LSB).
- **roundTowardPositive**: store the floating-point number closest to and no less than the infinitely precise result.
- **roundTowardNegative**: store the floating-point number closest to and no greater than the infinitely precise result.
- **roundTowardZero**: store the floating-point number closest to and no greater in magnitude than the infinitely precise result.

Rounding modes *roundTiesToEven and roundTiesAway* deliver the floating-point number closest to the infinitely precise result. ”Closeness” is determined by the GRS bits: in decimal integer rounding, two rounding results are equally close to the infinitely precise value if the fraction is 0.5_{10}, which translates to 0.100_2 in binary, as shown in equation Section 2.2.

$$0.500_{10} = 0.100_2 \tag{2.2}$$

Thus, assuming that the mantissa fields are integers, we conclude that:

- Mant.100: there exists two values that are equally near to the infinitely precise result.
- Mant.0xx: Mant.000 is the nearest to the infinitely precise result.
- Mant.1xx: Mant+1 is the nearest to the infinitely precise result.

We note that floating-point instructions use either a static rounding mode that is encoded in the instructions, or a dynamic rounding mode encoded as 111 in the instructions' *rm* field. When an instruction requires dynamic rounding, the actual rounding mode is fetched from RISC-V's Floating-Point Control Status Register (FCSR).

2.2.5 IEEE Exception

The IEEE 754 standard specifies five kinds of exceptions that should be signalled when they arise, as well as the default handling for each exception in the absence of any explicit user specification. The RISC-V ISA defines a special register, called the FCSR, where exception flags are accumulated. However, RISC-V does not trap floating-point exceptions, and leaves exception handling to the software's discretion. The supported exceptions are:

- **Invalid Operation**: signalled if and only if there is no meaningfully definable result, e.g. multiply(0,∞).
- **Division by Zero**: signalled if and only if an exact infinite result is delivered for operations on finite operands.
- **Overflow**: signalled if and only if the result is larger in magnitude than the destination format's largest finite number.
- **Underflow**: signalled if and only if a tiny non-zero result is delivered.
- **Inexact**: signalled if and only if the delivered result is different from what would have been computed were the range and precision infinite.

2.2.6 RISC-V Floating-Point Instruction Extensions [1]

RV32F and RV32D are the RISC-V instruction extensions for SP and DP floating-point instructions. RISC-V's floating-point instruction set covers a wide range of instructions that have been supported by previous ISAs. Figure 2.6 illustrates different encodings of RV32FD floating-point instructions. The floating-point load/store instructions are encoded in I-type and S-type respectively, identical to their integer counterparts. FMA instructions uses the R4-type instruction encoding as they operate on three inputs. For those floating-point instructions that require rounding, the *funct3* field indicates the rounding attribute. Floating-point conversion, classification, and move instructions require only one operand, thus the I-type encoding is utilized, where the 12-bit immediate field functions as control signals. Not explicitly shown in the Figure 2.6, floating-point load/store, integer-to-floating-point conversion instructions read from the integer register file, while floating-point-to-integer conversion, comparison, and classification instructions write to the integer register file.
In addition to the traditional floating-point instructions, RISC-V floating-point extension mandates the inclusion of FMA instructions. In terms of implementation, unfused multiply-add (FMUL followed by FADD) requires a (t+3)-bit adder for mantissa addition, where three extra bits are used for rounding. However, compliant FMA implementations are carried out as if with unbounded range and precision, and rounding is only performed at the end of addition [10]. Therefore, the (2*t)-bit intermediate mantissa generated by the multiplier

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-tyep
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

Figure 2.6: RV32FD Floating-point instruction encoding.

is propagated to the adder for FMA instructions. This greatly increases the width of the alignment and post-normalization shifters, as well as the floating-point adder to (3*t) bits, which increase resource usage and presents additional challenges in timing closure.

2.3 Taiga Overview

An important consideration for the implementation of our FPU is that it should not be bottlenecked by the performance of the base processor. Heinz et al. [30] conducted a comparative study across an array of RISC-V processors, and found Taiga [3] to be the most performant, and performance-efficient. Taiga is an FPGA-optimized, highly configurable, open source 32-bit RISC-V processor supporting the RV32IMA extensions. Taiga features parallel execution units, which allows the execution stage to be decoupled from the fetch, decode and issue stages. To support execution units with different latencies, Taiga implements register renaming and tracks instructions throughout their life-cycle in the pipeline, from instruction-fetch to instruction-retire. These features enables Out-of-Order (OOO) execution which allows more Instruction Level Parallelism (ILP). Figure 2.7 provides an overview of Taiga's pipeline. A standard execution unit interface is provided to simplify the integration process. We discuss the FPU's integration scheme to Taiga in Section 3.1.

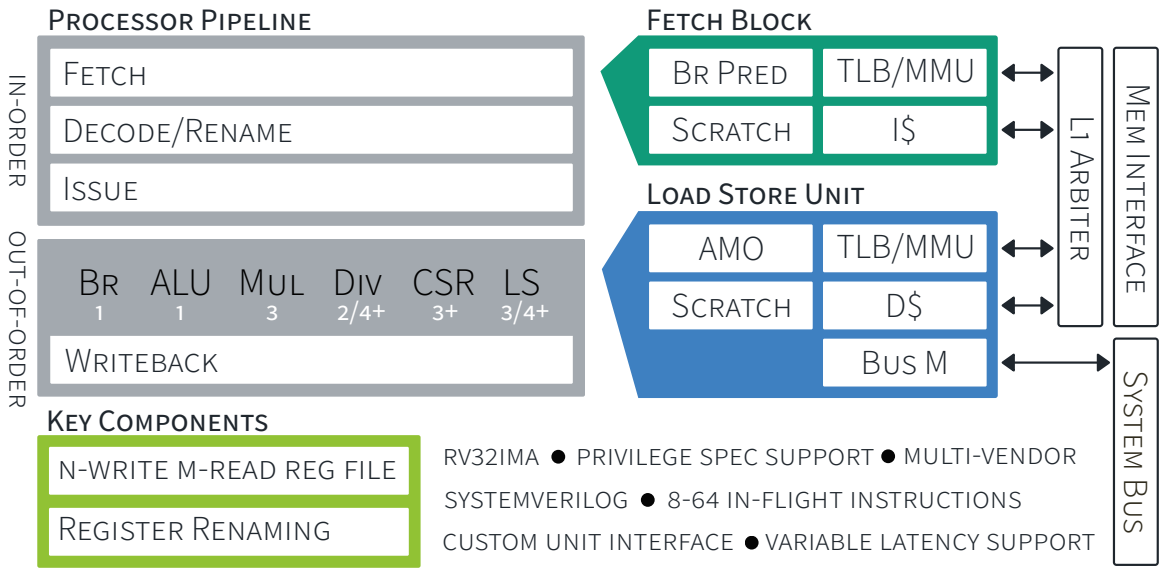

Figure 2.7: Taiga Pipeline Overview [3]

2.4 Related Work

In this section, we first highlight a few floating-point designs targeting ASICs, and demonstrate that the techniques and optimizations for ASIC are not directly transferable to FPGAs. Moreover, as this represents the bulk of floating-point research for FPGAs, we list existing work on standalone floating-point data-path designs, libraries and automated data-path generators. We then discuss several open-source and commercial integrated FPU designs. Finally, we outline the compliance deficiency of existing FPGA-based floating-point implementations in both standalone operators and integrated FPUs.

2.4.1 FPUs on ASICs

Extensive research has explored optimizing floating-point data-paths for ASICs, such as leading-one predictors [31] [32], rounding stage merging [33], and dual-path floating-point adders [34] [35] [36]. Bruguera et al. [32] [31] proposes and optimizes a leading-one prediction unit that detects the location of the leading-one concurrently with mantissa addition for floating-point adders. Bruguera et al. [33] introduces a compound adder that computes $A + B$ and $A + B + 1$ simultaneously, resulting in reduced latency due to the merging of mantissa addition and rounding stages. Moreover, Farmwald [34] introduces a dual-path (far-close-path) floating-point adder architecture. Dual-path adders partition the floating-point addition operation into two parallel paths depending on the inputs. The correct output is selected at the end of the pipeline. Since each path is only responsible for a subset of the possible inputs, the hardware required for one path can be removed from the other, resulting in reduced timing delay. Oberman et al. [35] improves the dual-path adders by allowing the back-loaded close-path to execute speculatively, achieving up to 33% reduction in latency. Seidel et al. [36] implements a delay-optimized dual-path adder by modifying the partitioning criterion. Other complex floating-point operators, such as divide and square-root [37] [38] [39], have also been extensively studied.

Unfortunately, the optimization opportunities afforded by ASIC's high-performance nature are not readily available for FPGAs due to technology differences. Malik et al. [40] point out that the adder optimizations [32] [34] previously discussed can increase resource utilization by 88% if implemented on FPGAs. Moreover, FPNew [41] is an ASIC-based FPU for RISC-V processors. A bare-bone FPNew core, i.e. no decode-and-issue, register file, or write-back logic, requires more than 6400 LUTs compared to our 5423 LUTs on a Zedboard. Last but not least, the general purpose LEON3 [42] configuration requires 7560 LUTs, whereas a similarly configured Taiga [3] core only needs 1950 LUTs, both targeting Xilinx 7-Series devices.

2.4.2 Standalone Floating-Point Cores on FPGAs

Much work touching upon floating-point on FPGAs has explored standalone data-paths [14] [16] [25]. This is likely due to FPGAs' usage as standalone accelerators, and the high parallelism that can be realized by replicating standalone operators. Furthermore, most work has focused on subsets of floating-point operations, i.e. just adders and multipliers [14] [15] [16], or dividers and square-root operators [25] [2]. For FPUs integrated into processors, we need to support all floating-point operations, and have additional constraints and opti-

mization goals over the standalone data-path designs. Since the base soft-processor constrains the operating frequency of the overall system , floating-point designs clocked at frequencies higher than that of the base processor offer no benefit in runtime performance compared to the ones clocked at the base-processor's frequency. Unfortunately, the additional pipeline stages required to achieve said high operating frequencies induce longer unit latency, and drastically degrade the processor's Instructions Per Cycle (IPC). Last but not least, standalone data-paths cannot take advantage of the resource sharing opportunities available in the processor system. For example, the post-normalization and rounding modules can be shared among all floating-point modules in an integrated FPU.

Early work has looked at the feasibility of implementing floating-point operators on FPGAs [43] [44]. As FPGA technology matured, their adoption in domain-specific applications drove a strong demand for fast, high-throughput floating-point operators. Hemmert et al. [14] explore and implement IEEE 754 compliant, FPGA-optimized floating-point adder (9 stages) and multiplier (14 stages). Dou et al. [15] proposes a 13-stage, DP floating-point Multiply-Accumulate (MAC) unit for accelerating floating-point matrix multiplication kernels. Jaiswal et al. [16] proposes a DSP efficient, DP floating-point multiplier. The low-latency configuration of the proposed multiplier has 6 pipeline stages, and achieves 310 MHz. Some work has also explored parameterized floating-point cores [45] [46].

These standalone, deeply-pipelined floating-point cores enable designers to offload demanding computations to FPGAs. However, they will likely perform poorly when integrated with general-purpose processors, as most software applications have limited ILP due to data dependencies. Albeit advanced processor architectures, such as Taiga, can mitigate the anti-dependencies (write-after-read and write-after-write) by implementing register renaming. In the event of true dependencies (read-after-write), deep pipelines will cause long processor stalls, and severely degrade the processor's runtime performance. Moreover, the high clock frequency achieved through additional pipelining provides little benefit, as the base soft-processor dictate the overall system's operating frequency.

Furthermore, standalone floating-point implementations are often too resource intensive for FPGA-based soft-processors. The DP floating-point multiplier proposed by Jaiswal et al. [16] utilizes 2071 LUTs, which is more than what Taiga [3] requires. Wang [25] et al. propose table-based, SP floating-point, pipelined division/reciprocal cores. However, the lookup table size increases exponentially with mantissa bit-width, making a DP version prohibitively large for FPGAs. Fang et al. [2] build upon Wang et al. [25] and propose variable-precision floating-point reciprocal, division and square root operator designs. Although the improved algorithm requires a smaller lookup table, the floating-point modules are still significantly larger than our base-processor Taiga, as shown in Table 2.1.

Table 2.1: VFLOAT's [2] reciprocal, divide and square-root operators require massive resources on Xilinx Virtex 6.

Operation	6-input LUT	BRAM
$1 \div x$	3449	8
$y \div x$	3487	8
$\sqrt{x}$	6285	58

Vendor supplied floating-point operators can have significant hardware footprint as well. Xilinx floating-point divide core [20] configured to high-throughput mode requires 3218 LUTs, while Intel's performance-optimized ALTFP_DIV core [47] requires 44 DSPs, significantly exceeding the resource requirement of Taiga (1948 LUTs and 2 DSPs). The extensive resource requirements add substantial constraints on the placer, and may further exacerbate the already frequency-constrained soft-processor system. In addition, researchers have explored automated floating-point libraries and generators in order to simplify the implementation process. VFLOAT [25] is an open-source library of variable-precision floating-point operators. FloPoCo [48] generates complex floating-point arithmetic cores in VHDL. Moreover, High-Level Synthesis (HLS) tools [49] [50] [51] abstract programmable logic development by enabling the synthesis of high-level programming code (C/C++) to Register Transfer Language (RTL). Although leveraging existing libraries and generators can accelerate the design process, users possess little control over the generated hardware besides the parameters the tools choose to expose (e.g. the pragmas in Xilinx HLS). Additionally, the libraries and automatic tools follow the same design methodology as that of the standalone cores, which are not suitable for soft-processors as discussed previously. Moreover, to our knowledge, no existing tools support generation of fully compliant floating-point data-paths.

2.4.3 Existing Integrated FPUs for FPGA-based Soft-Processors

There exists several SP [21] and DP [26] [17] [52] FPUs for soft-processors with support for all common floating-point operations. However, all FPUs targeting soft-processors are only partially compliant to IEEE 754. Moreover, there is a significant barrier to adopting these existing designs to other soft-processors, due to their closed-source nature, or the unconventional Hardware Description Language (HDL) used. In this section, we introduce the commercial and open-source FPUs targeting FPGA-based soft-processors, and provide an overview of their features (or lack thereof), as well as their adoptability to other soft-processors.

The vendor supplied MicroBlaze [26] and NIOS II [21] are proprietary soft-processors developed for Xilinx (recently bought by AMD) and Intel FPGAs respectively. Both implementations are equipped with optional FPUs that support floating-point addition, subtraction, multiplication, division, comparison, conversion and square root instructions. The FPUs share the floating-point register files with that of the integer ones. As such, only the 64-bit MicroBlaze configuration supports DP floating-point operation, and NIOS II only supports SP floating-point. Furthermore, both FPUs break IEEE 754 conformance for performance reasons, as neither MicroBlaze or NIOS II supports denormal number processing, full rounding modes, or compliant exception handling. As a result, both FPU designs achieve relatively low resource utilization. The DP MicroBlaze FPU can be clocked at 100 MHz and requires 1760 6-input LUTs on Xilinx Artix-7 FPGAs, and the SP NIOS II FPU can achieve 130 MHz with 2500 4-input Logic Elements (LEs) on Intel Cyclone IV devices.

GRFPU [52] is a paritially IEEE 754 compliant FPU designed for LEON [42] processors. The FPU supports both SP and DP floating-point instructions specified by the SPARC V8 ISA [53]. GRFPU features a shared multiplier that implements FMUL and non-blocking FDIV and FSQRT instructions. Similar to MicroBlaze and NIOS II, GRFPU does not support denormal numbers, therefore, it has a relatively low hardware footprint of 4700 LUTs on Xilinx 7-series devices. Although SPARC V8 is an open-source ISA, the free version of GRFPU only pro-

vides a pre-synthesized netlist for Xilinx Virtex II FPGAs, making it impossible to modify the GRGPU's backend implementation.

VexRiscv [28] is a highly configurable, open-source RISC-V processor. It supports both SP and DP floating-point operations through its extensive list of complementary plug-ins, however it deviates from the IEEE 754 standard. VexRiscv FPU does not fully support denormal number processing, as denormal numbers are not correctly promoted to normal numbers during rounding. Additionally, the FMA instructions' intermediate results are truncated to just (MANT_WIDTH+2), which can lead to complete loss of precision as we will demonstrate in Section 2.4.4. The omission of these compliance features leads to a relatively small FPU design of 3779 6-input LUTs clocked at 114 MHz on a Zedboard. Despite being fully open-source, VexRiscv is written in unconventional HDL SpinalHDL [17] that is compiled to Verilog and VHDL before synthesis. As the compilation engine flattens the design hierarchy, and discards any comments associated with the SpinalHDL source code, the generated Verilog and VHDL source code is unreadable. Therefore, it is difficult to migrate the FPU to other architectures.

NaxRiscv [27] is another open-source RISC-V processor developed using the SpinalHDL language. As an upgrade to the VexRiscv project, NaxRiscv's FPU propagates the full (2*MANT_WIDTH)-bit intermediate mantissa from FMA's multiplication stage to the addition stage. Expectedly, the fully compliant FPU requires 5591 LUTs, and achieves 98 MHz on a Zedboard. We note that despite its advancements, NaxRiscv still suffers from the same flaws of implemented using unconventional language SpinalHDL [17].

2.4.4 Compliance Status of Existing Research

A commonality across all existing floating-point work for FPGAs is that full IEEE 754 compliance is rarely achieved. Almost all standalone data-path designs are non-compliant with few exceptions [14]. Furthermore, existing open-source and commercial integrated FPUs targeting soft-processors are rarely compliant [28] [26] [21] [52]. The non-compliant implementations typically remove denormal number processing due to the significant hardware requirement. Moreover, further resource optimization is possible if only default rounding mode is supported (i.e. merging rounding addition with mantissa addition).

Previous work [45] [46] has investigated the effect of running benchmarks in reduced compliance levels, and found that supporting denormal numbers and full rounding modes can have a resource usage overhead of 200% and 15% respectively. However, previous research were not conducted in the scope of the processor's environment, especially RISC-V based soft-processors. Additionally, among the few RISC-V FPU designs, none implements the fully compliant FMA instructions mandated by the RISC-V ISA.

In order to demonstrate the negative impact of neglecting FMA compliance, we present the step-by-step computation of an FMA instruction in Figure 2.8. The example solves for $-100.15 * 0.99999999990 + 100.15$ in the following steps:

1. Multiplication stage generates a 2*T-bit intermediate result, where T is the mantissa field width (2.3). The residual bits are colored in green. The exponent fields are colored in gray, as we emphasize the interactions between mantissa fields.

2. The 2*T-bit intermediate result is propagated to the adder. Since the intermediate result and the third operand have opposite signs, the effective operation is subtraction.

3. The adder's inputs are swapped so that the minuend is larger in magnitude. The subtrahend is right-shifted so that its radix point is aligned with that of the minuend.

4. The two mantissas are subtracted, and normalized to obtain the final floating-point result.

It can be seen after alignment (step 3), the two upper mantissas (colored in red) are almost identical. As a result, the subtraction operation causes the most significant bits to cancel, and the actual mantissa is preserved in the residual bits (colored in blue). Post-normalization then left-shifts the residual bits to compose the final output. However, these precision bits would have been lost if the multiplication intermediate result (colored in green) is truncated, as is the case in VexRiscv [28], resulting in a catastrophic loss of precision.

In addition to maximizing portability and accuracy, users should be able to leverage FPGAs's flexibility and turn on/off specific compliance features when appropriate. However, existing implementations have fixed compliance features. Given the size of modern FPGAs, providing an optionally compliant FPU will allow the user to tradeoff performance and performance efficiency versus accuracy to meet the needs of their application.

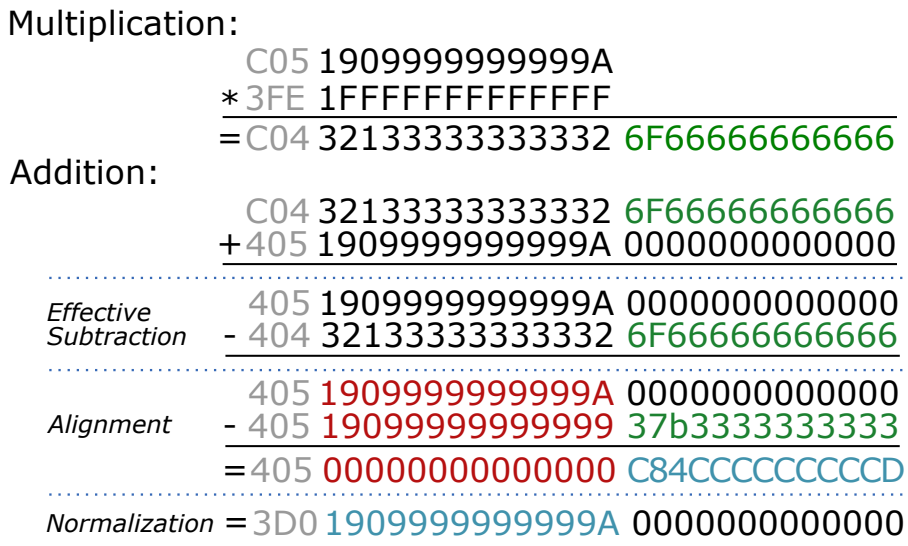

Figure 2.8: Step-by-step calculation of -100.15*0.999999990+100.15: addition step causes complete loss of precision unless the intermediate result is preserved. Note that the mantissa fields are 53-bits wide, thus shifting causes irregular changes to the hexadecimal representation.

2.4.5 Related Work Summary

As discussed, FPU designs targeting ASICs map poorly on FPGAs. Furthermore, existing FPGA-based floating-point standalone data-paths design are independently frequency-optimized. As such, they are not suitable for FPGA-based soft-processors due to long latency and soft-processor's clock frequency constraints. Moreover, the resource requirement of standalone cores makes them prohibitively large, as they are optimized

for throughput, and unable to leverage the resource sharing opportunities available in a processor's environment. In addition, existing integrated FPU designs for FPGA-based soft-processors omit important compliance features such as denormal and full rounding modes. Their proprietary nature makes it impossible to support additional functionalities. While we are aware of one RISC-V based, open-source FPU implementation, its programming environment is unconventional and difficult to adapt to other architectures. Crucially, no existing design supports the full set of compliance features mandated by the RISC-V ISA. Therefore, in this work we explore optionally compliant, compile-time multi-precision FPU designs.

3 FPU Implementation

In this chapter, we discuss the implementation details of our FPU. We begin with an overview of the FPU's integration to Taiga, including instruction decode-and-issue, ID management, and shared integer register file access. We then present the FPU's top-level processing stages, and provide insight into the FPU's micro-architecture designs. Lastly, we describe each instruction's pipeline in detail.

3.1 FPU Integration to Taiga

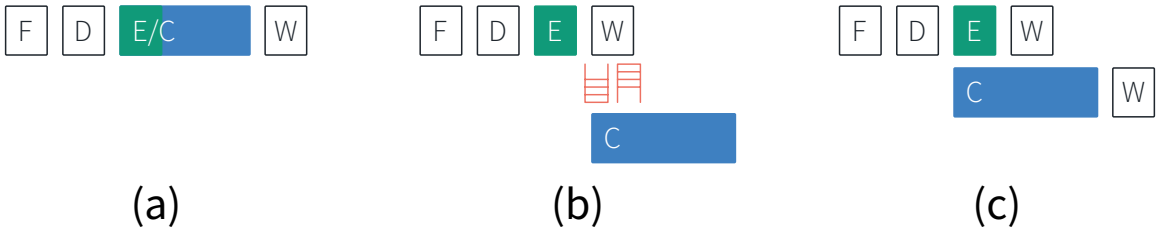

Figure 3.1: Custom Accelerator Integration Options

There exists several ways to integrate custom logic into a soft-processor, as visualized in Figure 3.1. Option (a) is typically used in fixed-pipeline designs where the custom accelerators are merged with the existing execution stage. This option has low overhead, but the ILP is limited due to the single pipeline. Option (b) enables higher parallelism than (a), but overhead and synchronization at the write-back stage can become the bottleneck — especially for low latency instructions. Option (c) visualizes the integration scheme that supports parallel execution units and custom accelerators. In this scheme, the custom logic interfaces with the processor as a base execution unit would. As a result, option (c) combines the advantages of (a) and (b), and enables low-overhead and high ILP custom logic integration. However, (c) has the additional complexity in managing pipelines and ordering. Fortunately, Taiga supports option (c) natively, and has conveniently provided a standardized unit interface that simplifies the integration process.

In this section, we present a detailed discussion on the integration of the FPU to Taiga. We then describe the modifications needed to the ID management and instruction decode-and-issue logic to support the floating-point instructions.

3.1.1 Instruction Fetch, Decode, Issue, and Management

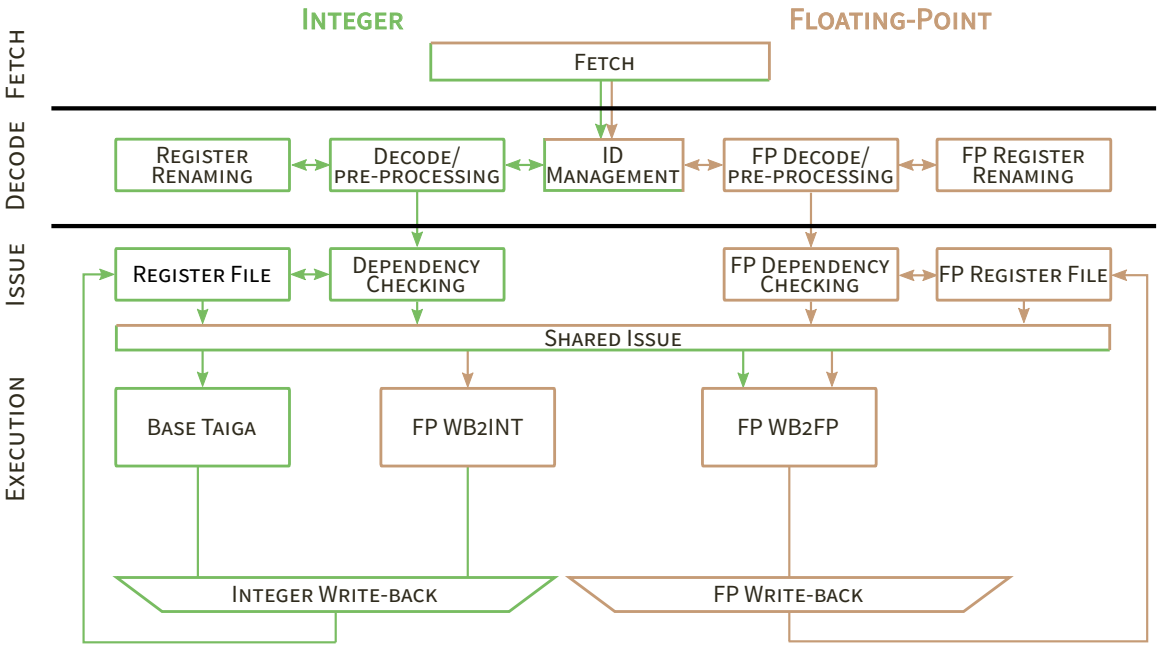

Figure 3.2: FPU Integration with Taiga.

Figure 3.2 provides an overview of Taiga with the integrated FPU. It can be seen that the integer and floating-point pipelines share the instruction fetch unit, as highlighted in both green and orange. The FPU also shares the pool of instruction IDs with the integer instructions. This enables a unified instruction management unit which simplifies the interaction between integer and floating-point instructions. As a result, we expand upon the existing instruction decode and ID management logic to handle floating-point instruction tracking. Additionally, we have implemented a separate floating-point register file as mandated by the RISC-V ISA [1]. Thus, the logic for integer register renaming (in the decode stage) is duplicated for the floating-point register file.

The pipeline splits once the instruction reaches the issue stage. However, some floating-point instructions read their operands from the integer register file, and vice versa. Table 3.1 outlines the floating-point instructions supported by our FPU, and their source and destination register types.

Table 3.1: Floating-point instructions supported. Source and destination columns outline the instructions' input and output types.

Instruction	Functional Description	Read	Write
FLD	Floating-point load	INT	FP
FSD	Floating-point store	INT, FP	No Output
FMA	Fused Multiply-Accumulate	FP	FP
FADD, FSUB	Add, subtract	FP	FP
FMUL	Multiply	FP	FP
FDIV	Divide	FP	FP
FSQRT	Square-root	FP	FP
FCVT.D.(U)W	(Unsigned) integer to floating-point conversion	INT	FP
FMIN, FMAX	Minimum, maximum	FP	FP
FSGNJ, FSGJN, FSGNJX	Move, negate, absolute value	FP	FP
FCVT.(U)W.D	Floating-point to (unsigned) integer conversion	FP	INT
FLE, FLT, FEQ	<=, <, ==	FP	INT
FCLASS	Classify	FP	INT

As such, the issue stage must be shared to enable the floating-point and integer data transfers. To facilitate such inter-register-file dependency checking and unified instruction management, we need to track additional instruction metadata:

- *read-int-data* and *read-fp-data*: signifies the source operand types.
- *write-int-data* and *write-fp-data*: signifies the destination types.
- *is-float*: asserted when the instruction is a floating-point instruction.
- *accumulate-csr*: asserted when the instruction may update the FCSR.

Read-int-data and read-fp-data are used to track the instructions' operand types, and facilitate operand dependency checking. The ability to distinguish the operand type is crucial as some floating-point instructions require integer operands. For example, floating-point store instructions reads the integer register file for base address and offset, while obtaining the out going data from the floating-point register file. Moreover, to further decouple the two data-paths, the integer and floating-point register files have independent register renaming units. Notably, renaming the integer registers should not affect the floating-point registers, and vice versa.

Write-int-data and write-fp-data attributes help the ID management and renamer modules identify the instructions' destination register type, and provide appropriate physical registers in the issue stage. Similarly, we also use these flags to ensure that the physical registers are returned to the correct free register pool during instruction retirement.

Is-float flag is asserted for all floating-point instructions, although it is only used for controlling memory operations. The integer and floating-point memory operations share a single memory bus. Therefore, the flag *is-float* is needed to distinguish the resulting data type. *Is-float* also controls the store byte-mask and load-store-queue output selection. Since the load store unit can commit data to both register files, we use *is-float* to assert the appropriate output valid bit during its write-back stage.

Accumulate-csr tracks instructions that may write to the FCSR. The RISC-V ISA does not trap for floating-point exceptions [1]. Instead, the standard specifies that floating-point exceptions should be accumulated in the FCSR and handled by software. The *accumulate-csr* flag therefore controls the floating-point exception status update logic.

Furthermore, Figure 3.2 also visualizes the shared issue interface between integer and floating-point execution units. The interface is responsible for propagating the integer operands and dependency status to those floating-point instructions that require integer operands, such as floating-point load/store, and integer-to-float conversion. Moreover, the integer write-back logic is extended to support the floating-point instructions that commit to the integer register file.

3.2 FPU Stages

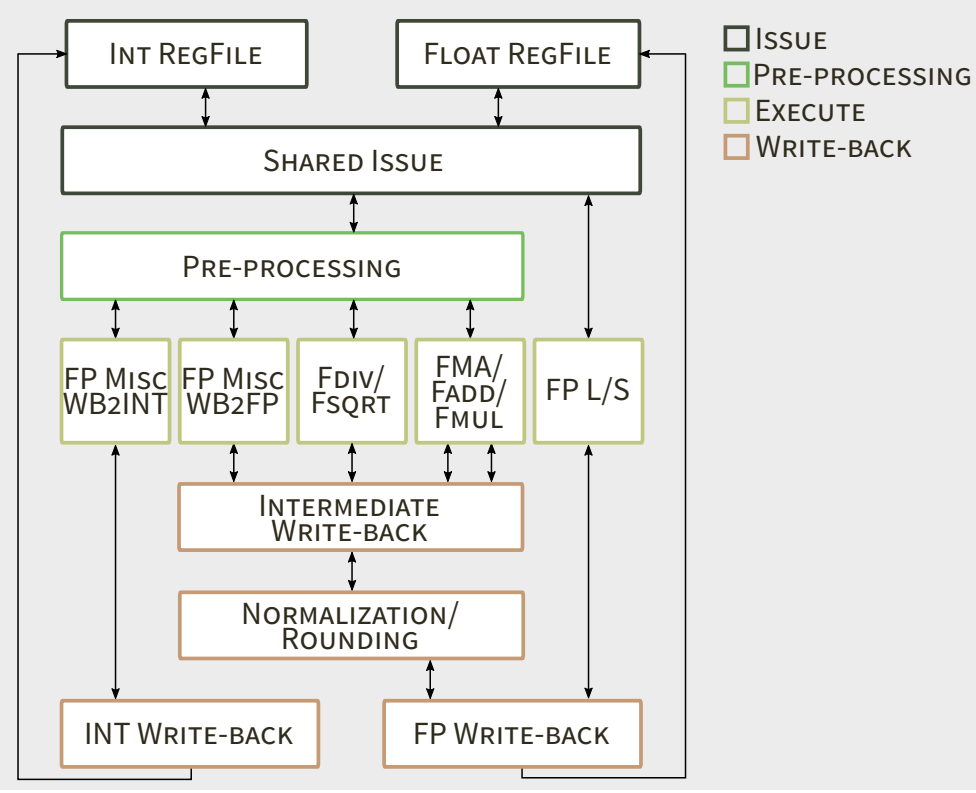

Figure 3.3: From the issue stage, floating-point instructions are issued in-order with execution being broken down into: pre-processing, execution and write-back stages.

In this section, we describe the top-level FPU stages. The four main stages, as highlighted in Figure 3.3, can be briefly summarized as:

- Issue Stage: reads integer and floating-point operands from the register files.
- Pre-processing Stage: pre-computes signals shared by the execution units.
- Execution Stage: processes the instructions.
- Write-back Stage: Commits results to the register file.

The following sections will discuss the interfaces between the stages, and provide an overview of the FPU's micro-architecture design. We note that these stages describe the high-level floating-point instruction execution sequence in the FPU, and are not a representation of the hardware pipeline (measured in clock cycles).

3.2.1 Issue Stage

The issue stage modules, marked in black in Figure 3.3, consist of the single-cycle issue logic and two register files. As discussed in Section 3.1, the shared issue interface coordinates dependency checking between both integer and floating-point register files. Moreover, it facilitates data transfer for those floating-point instructions that require integer operands. We note that the floating-point load-store's execution stage interfaces with the issue stage directly, as it requires no pre-processing.

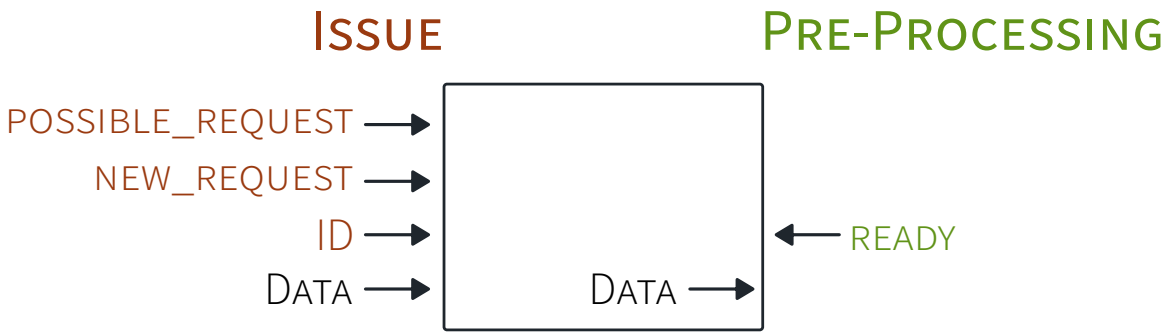

Figure 3.4: Unit issue interface

Figure 3.4 presents the interface between issue and pre-processing stages. The issue stage receives the *ready* signal from the pre-processing stage, and forwards the *new_request* signal when an instruction can be issued and the pre-processing stage is ready. The *ready* signal of the pre-processing stage is itself dependent on the status of the execution unit required by the current instruction. If the required execution pipeline stalls, the upstream pre-processing and issue stages may also stall. Moreover, The instruction ID, for ordering and tracking purposes, is propagated along with the data and other metadata for the instructions.

3.2.2 Pre-Processing Stage

The pre-processing stage, visualized in Figure 3.3 in green, is an intermediate stage that performs three important tasks:

- forwards the control signals and data from the issue stage to the execution stage.
- back-propagates the unit ready status signals from the execution units to the issue stage.
- computes various intermediate results that are shared by many instructions.

We have discussed the first two tasks in the previous section, therefore we focus on the third task in this section.

Many floating-point instructions require similar preparation steps before the operands can be processed. For example, FADD inputs are sorted in descending order before mantissa addition/subtraction. The sorted results can then be shared by floating-point compare and min/max instructions.

Table 3.2: The shared pre-processing stage performs sorting, pre-normalization, and special input detection.

Modules	Shared By
Sorting	FMUL, FADD, FCMP, FMIN/FMAX
Pre-normalization	FMUL, FDIV, FSQRT
Special-Input-Detection	All FP Instructions, except memory operations

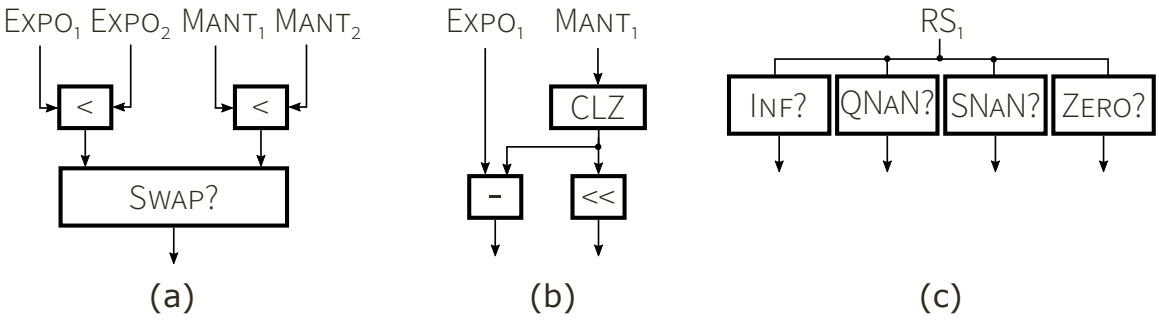

Figure 3.5: Implementation of the shared (a) sorting, (b) pre-normalization, and (c) special-input-detection modules.

Table 3.2 and Figure 3.5 provide an overview of the three shared hardware modules in the pre-processing stage. The *sorting* module (a) sorts two floating-point numbers in descending magnitude order. Sorting involves comparing the exponent and mantissa fields, and it is typically mapped to subtractors by the synthesis tool. As DP floating-point mantissa comparison requires a 53-bit subtractor, sharing this hardware allows us to save substantial resource and reduce routing congestion. Next, pre-normalization (b) refers to the operation of converting a denormal number to a normal one. This is implemented by first computing the count-leading-zero of the mantissa, and then left-shifting the mantissa by that amount so that the implicit leading bit is asserted. Floating-point multiplication, division, and square-root instructions require this step so that the resulting mantissa can be correctly scaled. However, instantiating separate *pre-normalization* modules for each instruction is too expensive, as the large 53-bit shifters can cause significant resource usage and routing congestion. Lastly, the *special-input-detection* module (c) determines whether the inputs are IEEE 754 special

values outlined in Section 2.2. Every arithmetic instruction must perform special input handling, thus, it is resource efficient to share this module.

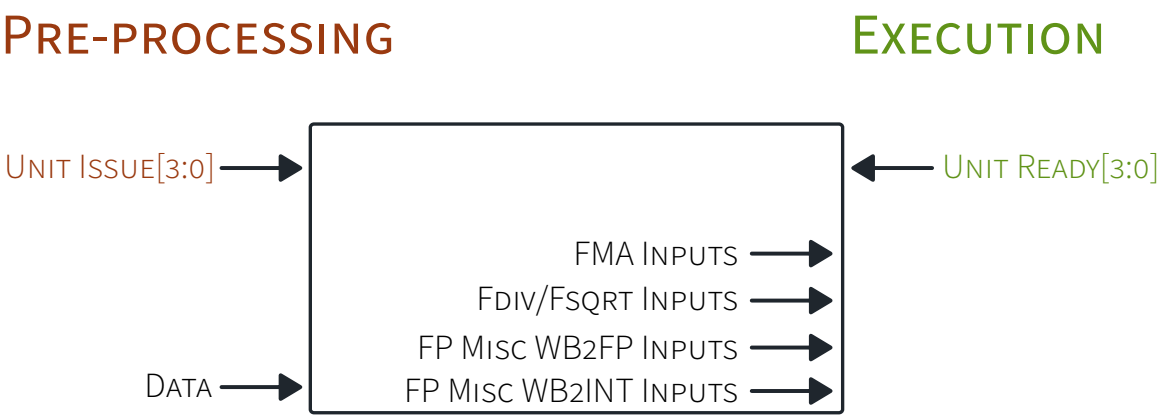

Figure 3.6: The pre-processing stage pre-computes intermediate results, packs and propagates execution unit inputs.

Figure 3.6 presents the interface connecting the pre-processing and execution stages. As mentioned in Section 3.2.1 and the beginning of this section, each execution unit, except floating-point load-and-store, is connected to the issue stage through the pre-processing stage via the shared issue interface visualized in Figure 3.4.

3.2.3 Execution Stage

The execution stage is visualized as the yellow blocks in Figure 3.3. It consists of five execution units:

- *FP L/S*
- *FMA/FADD/FMUL*
- *FDIV/FSQRT*
- *FP MISC Floating-Point Units that Write-back to Floating-Point Register (WB2FP)*
- *FP MISC Floating-Point Units that Write-back to Integer Register (WB2INT)*

Table 3.3: Each execution unit supports several floating-point instructions.

Unit	Instruction	Functional Description
FP L/S	FLD, FSD	Load, Store
FMA	FMA	Fused Multiply-Accumulate
	FADD, FSUB	Add, Subtract
	FMUL	Multiply
FDIV/FSQRT	FDIV	Divide
	FSQRT	Square-root
WB2FP	FCVT.D.W(U)	(Unsigned) Integer to Floating-Point Conversion
	FMIN, FMAX	Minimum, Maximum
	FSGNJ, FSGJN, FSGNJX	Move, Negate, Absolute Value
WB2INT	FCVT.W(U).D	(Unsigned) Floating-Point to Integer Conversion
	FLE, FLT, FEQ	<=, <, ==
	FCLASS	Classify

We present the supported floating-point instructions again in Table 3.3, but categorized by their respective execution units. We designed each execution unit to handle multiple instructions in order to reduce the number of issue interfaces needed. The instruction decode-and-issue stage has often been in the critical path during our implementation process, partially due to high fan-out. Additionally, Taiga heavily leverages the LUTRAM primitives for instruction and register file management, and LUTRAMs have longer delays than FFs. As a result, paths starting and ending at LUTRAMs, common in the decode stage, can become the critical path. The synthesis tool can sometimes remove redundant logic and duplicated registers. Nonetheless, we expect the tool to produce better designs if the issue interfaces are explicitly merged.

Furthermore, the write-back interfaces are also merged to promote better routing and resource sharing. Matthews et al. [54] explored various write-back and storage mechanisms, and investigated the scalability of Taiga in terms of adding extra units. They found that supporting 4 additional units could cause a 32% increase in LUT utilization and -5.7% decrease in clock frequency. The impact for floating-point units could be higher, as while the control logic will scale similarly, the floating-point data-path is significantly wider.

To simplify the control logic, we prioritize merging instructions with similar pipeline characteristics, e.g. unit latency, destination register type, and whether hardware sharing is possible. The FPU instantiates a total of four issue interfaces, five floating-point write-back interfaces, and one integer write-back interface. The following sections discuss each execution unit and their supported instructions in detail.

FP L/S Unit

The floating-point load/store unit shares its issue interface with the integer load/store, therefore bypassing the pre-processing stage as discussed in Section 3.2.1. One floating-point write-back interface is added for the floating-point load results. The floating-point load-and-store instructions are implemented leveraging the existing integer load-store-queue with minimal additional logic to track floating-point attributes and data.

FMA Unit

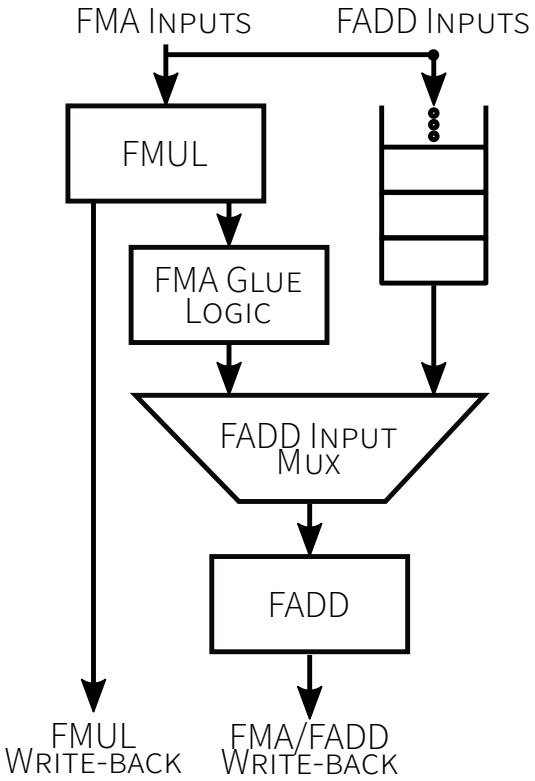

Figure 3.7: FMA unit overview: the FMA instructions are constructed using a floating-point multiplier and a floating-point adder.

Figure 3.7 provides an overview of the FMA unit pipeline. The Fused Multiply-Add (FMA) unit implements the floating-point FMA, FADD and FMUL instructions. While the FMUL and FMA instructions are both issued to the floating-point multiplier immediately, FADD instructions are first stored in a First-In First-Out (FIFO). We then propagate the FADD instructions to the floating-point adder only if no FMA instruction can claim the adder. This allows us to prioritize FMA instruction processing.

Two write-back interfaces are instantiated: one for the multiplier path (FMUL instruction) and one for the adder path (FMA and FADD instructions). Though it is possible to combine the two write-back interfaces, we maintain the current design, as multiplication occurs relatively frequently — as much as 24%. Thus, it can benefit from an independent write-back port. This design enables us to construct the FMA data-path using the add and multiply hardware, with only a small resource overheard for glue logic.

FP DIV/SQRT

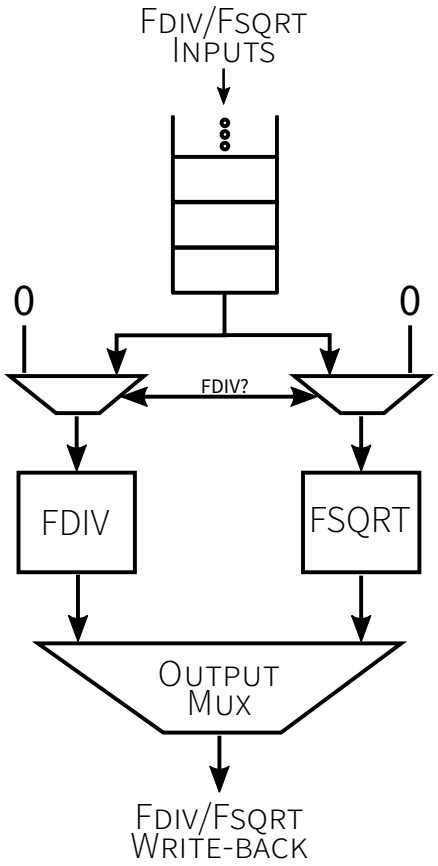

Figure 3.8: Floating-point divide and square-root pipeline overview.

The FP DIV/SQRT unit supports the floating-point division and square-root instructions, as shown in Figure 3.8. We use two FIFOs to store the inputs, as both instructions are multi-cycle. When both instructions are ready to commit simultaneously, the output multiplexer prioritizes the FDIV results since they occur more frequently — our benchmark applications contain up to 4% floating-point division instructions, while having at most 0.2% floating-point square-root ones.

WB2FP

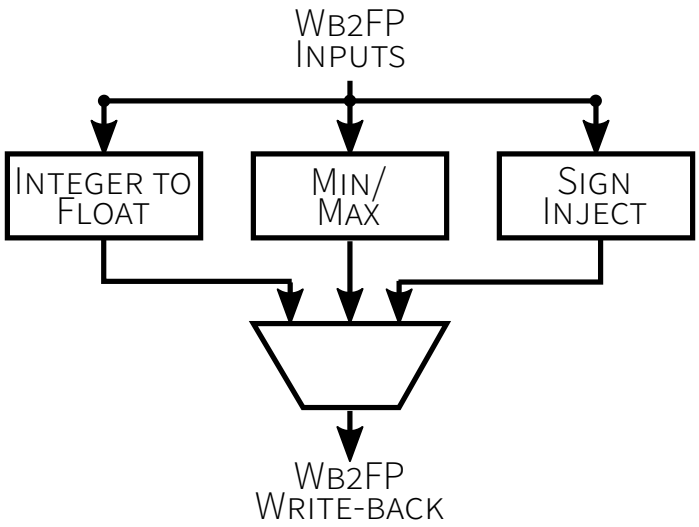

Figure 3.9: Miscellaneous floating-point instructions that commit to the floating-point register file are grouped together.

The WB2FP unit implements miscellaneous floating-point instructions that write to the floating-point register file. Figure 3.9 visualizes the unit organization. It supports integer to floating-point conversion, min/max, and sign injection instructions which realize floating-point move, negate and absolute value functions. We group these instructions together, since they all finish in two cycles, and commit back to the floating-point register file.

WB2INT

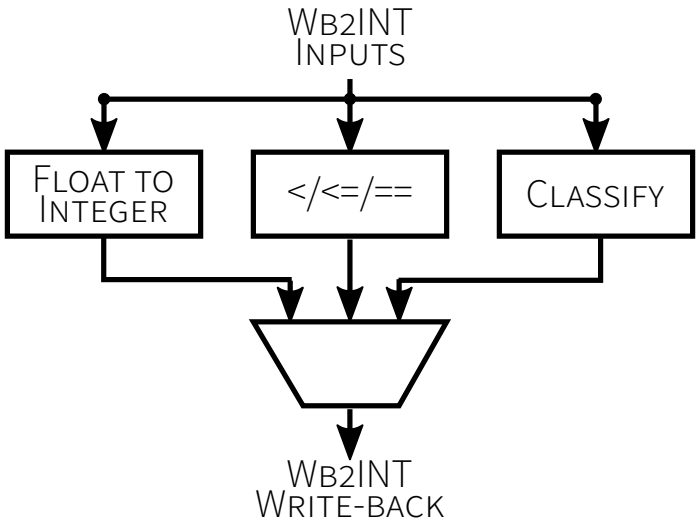

Figure 3.10: Miscellaneous floating-point instructions that commit to the integer register file are grouped together.

The WB2INT unit supports miscellaneous floating-point instructions that write to the integer register file, including floating-point to integer conversion, floating-point comparison, and classification instructions, as shown in Figure 3.10. The comparison instruction writes to the integer register file because RISC-V branch instructions operate on integer registers. The classification instruction returns a bit vector that indicates the input floating-point number's status. The floating-point to integer conversion data-path has a latency of two cycles, whereas floating-point comparison and classification operations finish in one cycle. To avoid the resource overhead of additional pipeline control logic, we delay the floating-point comparison and classification results by one cycle to synchronize with the floating-point to integer conversion instruction. The added latency has trivial impact on runtime performance (IPC) as all three instructions are not performance critical. Similar to the WB2FP unit, the results are multiplexed and share a common integer write-back interface.

3.2.4 Write-back Stage

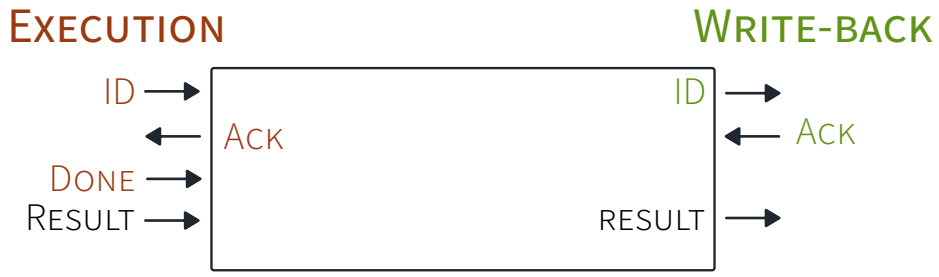

Figure 3.11: Interface connecting the execution units and the write-back stage.

The execution units connect to the write-back stage using a standardized interface as shown in Figure 3.11. The instruction ID passes through the execution units to the write-back stage, and is later used to accommodate instruction retirement. The *done* signal indicates to the write-back stage that the unit result is ready, and the *ack* signifies that if write-back stage has accepted the result.

Figure 3.3 presents two floating-point write-back modules: an intermediate write-back and a final write-back to the floating-point register file. The intermediate write-back selects one intermediate arithmetic result for post-normalization and rounding, as discussed in Section 2.2. It can be seen from Figure 3.3 that floating-point load results bypass the intermediate write-back, and connects to the final write-back stage directly. Memory operations can make up to 80% of floating-point workloads [55]. Therefore, this design allows the performance critical memory operation to commit earlier, and improves the FPU's IPC (7.9% compared to non-bypass floating-point load). The final write-back module then selects between the post-normalized and rounded arithmetic and floating-pint load results for write-back to the register file.

We note the current design can retire up to two instructions per cycle, with only one of the instructions writing back to either register file. Therefore, to handle simultaneous write-back requests, we commit floating-point results with the following fixed priority

1. Load.
2. FMA/FADD.

3. FMUL.
4. FDIV/FSQRT.
5. WB2FP instructions.

The ordering is based on the instructions' frequency of occurrence. As we have shown that memory operations dominate floating-point applications, prioritizing floating-point load over arithmetic results reduces the number of pipeline stalls, thus increasing IPC. Furthermore, the FMA/FADD share a single write-back interface which is ranked above FMUL, followed by FDIV/FSQRT. The WB2FP instructions have the lowest priority as they are used relatively infrequently, and have the least impact on IPC.

3.3 Instruction Pipeline Detail

In this section, we present implementation details of all instructions supported by each execution units shown in Figure 3.3. We note that some instructions require the intermediate results generated in the pre-processing stage as discussed in Section 3.2.2.

3.3.1 Floating-point Load and Store Unit

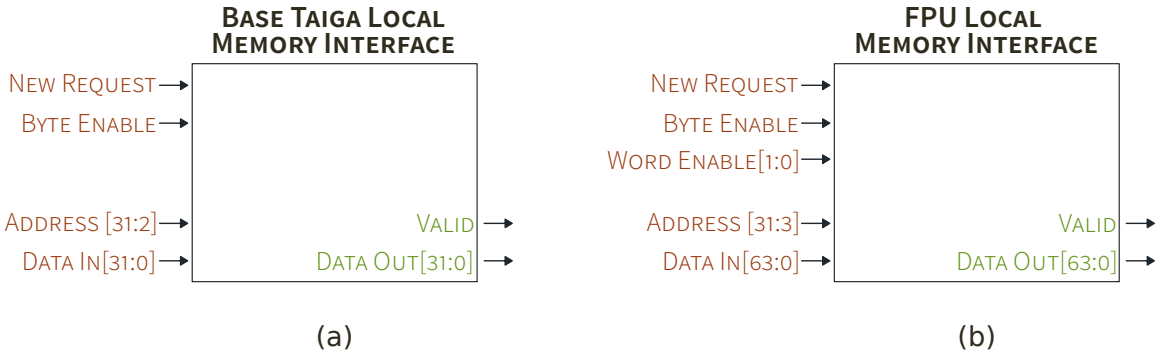

Figure 3.12: Floating-point memory interface widens the data bus, shrinks the address bus, and adds the 2-bit word enable vector.

We modified the 32-bit Taiga to support DP floating-point memory operations. Figure 3.12 visualizes the modifications done to the memory interface. In order to support DP floating-point numbers, the 32-bit memory blocks are widened to 64 bits which allows us to store two 32-bit integers per block. The 32-bit Taiga has a 30-bit address bus, with the least significant two bits used as byte masks. With the DP FPU integrated, the address bus shrinks to 29 bits where the address[2] acts as a word select. In addition to the address bus changes, we widen the data bus to 64 bits. To select the appropriate 32-bit data, we use the 2-bit *word enable* control signal to multiplex the two 32-bit words on the data bus.

While floating-point load-and-store shares the issue interface with the integer variant, additional data and control signals are needed to support floating-point instructions. The floating-point store data and forwarding instruction ID are propagated to the load-store-queue. The *is-float* instruction attribute (3.1) is also

included to distinguish between floating-point and integer types. As mentioned in Section 3.2.3, the existing load-store-queue is extended and tracks floating-point attributes, such as *is-float* and floating-point store forwarding information.

The integer load are committed through an integer write-back interface, and floating-point load has a floating-point write-back interface. The bus *valid* signal and the *is-float* flag are AND-ed to drive the correct output *done* signal.

3.3.2 FMUL, FADD, and Fused Multiply-Add (FMA) Units

We have provided an overview of the FMA unit, and discussed the hardware sharing scheme for the instructions implemented by the unit. In this section, we describe the floating-point multiplier, adder and the glue logic in detail.

Floating-Point Multiply

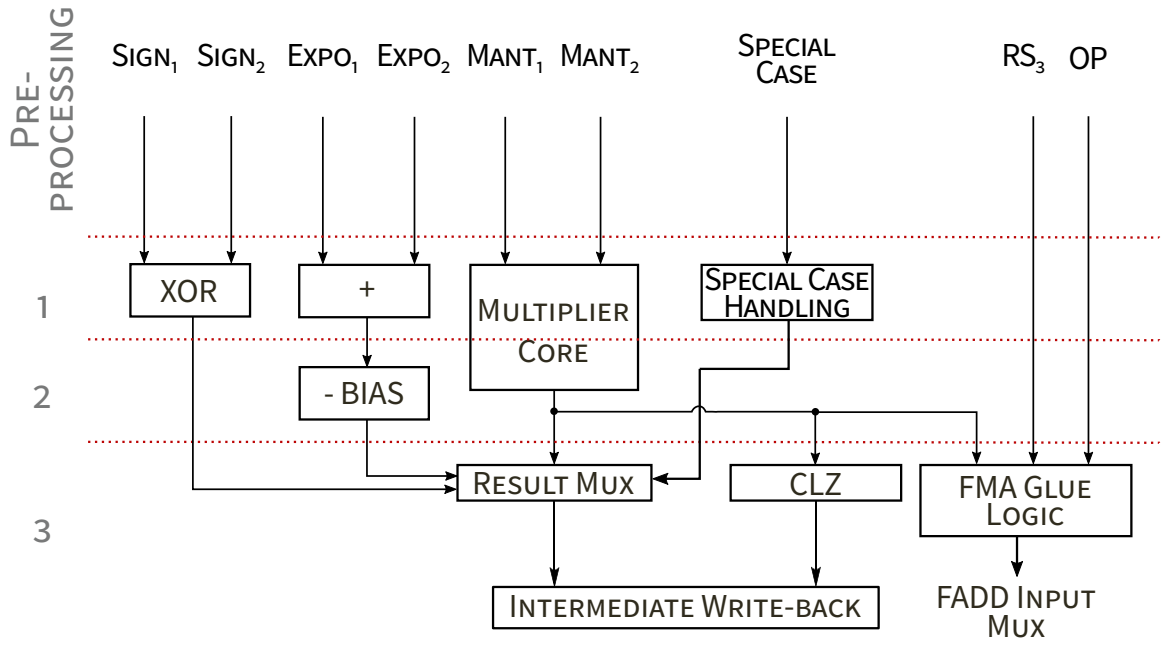

Figure 3.13: Cycle Breakdown of Floating-Point Multiply.

Typically, without considering exception handling, floating-point multiplication consists of the following steps:

1. Reorder the inputs so that RS_1 holds the larger input.
2. Pre-normalize RS_2 if it's denormal.
3. Calculate sign $Sign_{result} = XOR(Sign_1, Sign_2)$
4. Calculate exponent: $Exp_{result} = Exp_1 + Exp_2 - BIAS$.

5. Multiply mantissas.
6. Normalize/denormalize.
7. Round.

Figure 3.13 visualizes the floating-point multiplication pipeline. The input fields are ordered and pre-normalized in the pre-processing stage, as we have discussed. The *SpecialCase* signal, also calculated in the pre-processing stage, is a 4-bit vector, {*is_inf*, *is_QNaN*, *is_SNaN*, *is_zero*}, indicating the status of the input operands.

The result sign bit is calculated by XOR-ing the input sign bits, and we obtain the result exponent using the equation in step 3. We note that a negative exponent after step 3 indicates a denormal result which can be post-normalized by left-shifting. The multiplier core produces the (2*MANT_WIDHT)-bit intermediate result of the two mantissas. The most significant (MANT_WIDTH+3) bits are the preserved as the FMUL result mantissa, and the rest are OR-ed to generate the sticky bit. The selected result, either the arithmetic or the special case output, is packed and propagated to the intermediate write-back module for post-normalization and rounding. The full (2*MANT_WIDTH)-bit intermediate mantissa, RS3, and control signals are forwarded to the glue logic to prepare for the FMA's addition stage.

Floatin-Point Add

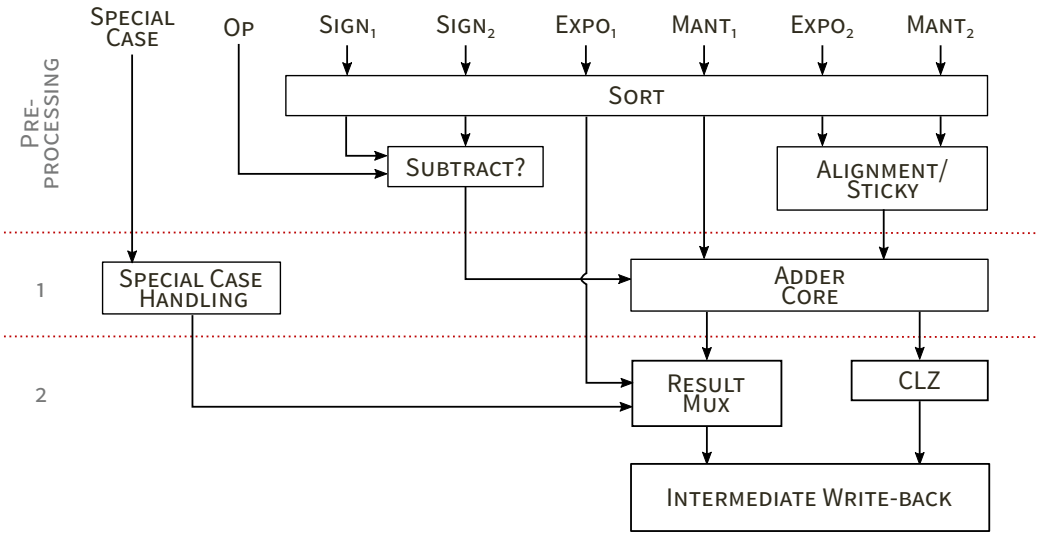

Figure 3.14: Cycle Breakdown of Floating-Point Add.

A basic floating-point add algorithm can be summarized as follows:

1. Sort the inputs in descending order (magnitude).
2. Right shift $Mant_2$ by the absolute value of the exponents.
3. Set $Expo_{result} = Expo_1$, $Sign_{result} = Sign_1$.

4. Add/Subtract the mantissas.

5. Normalize/denormalize the intermediate result.

6. Round.

Figure 3.14 visualizes the floating-point adder's pipeline. Similar to the floating-point multiplier, the adder inputs must be sorted in descending order. However, a separate *sorting* module is instantiated as the adder must handle both FMA and FADD instructions. We decided against sharing the existing sorting module in the pre-processing stage, as complex control logic would be needed to facilitate FADD operands back-propagation. The effective mantissa operation, either addition or subtraction, is then determined using the sorted sign bits. $Mant_2$ is right-shifted to align $Expo_2$ and $Expo_1$. The bits that are shifted out of the $Mant_2$ are used to calculate the rounding bits.

The result sign and exponent are equal to that of $Sign_1$ and $Expo_1$ since it is larger in magnitude. Depending on the effective operation, the adder core adds/subtracts the input mantissas. We note that a typical FADD/FSUB algorithm requires a (MANT_WIDTH+3)-bit adder/subtractor. However, since the floating-point adder is shared with the FMA instructions, we implement a (3*MANT_WIDTH)-bit adder to account for the full intermediate multiplication bits (details in Section 3.3.2). The computed fields are packed and multiplexed with the special case outputs. The appropriate result is then sent to intermediate write-back module for post-normalization and rounding.

FMA

Table 3.4: FMA Unit Supported Instruction List

Instruction	Functionality
FMUL	$rs1 * rs2$
FADD	$rs1 + rs2$
FMADD	$rs1 * rs2 + rs3$
FMSUB	$rs1 * rs2 - rs3$
FNMADD	$-rs1 * rs2 - rs3$
FNMSUB	$rs1 * rs2 - rs3$

The glue logic is responsible for preparing the intermediate multiplication signals for the addition stage as shown in Figure 3.7. This mainly includes repacking the intermediate multiplication signals, and RS_3 to a valid FADD input packet. Moreover, the control signal *OP* must reflect the effective FMA instruction addition/subtraction as outlined in Table 3.4.

The glue logic also contains a FIFO responsible for storing FADD instruction inputs, and a multiplexer for selecting one of the FADD input packets. This allows us to prevent the structural hazard on the floating-point adder when both FMA and FADD instructions may enter the addition stage. The multiplexer prioritizes FMA instructions as they occur more frequently in our benchmarks (avg. 23% vs FADD's 10%).

We mentioned in the previous section that a larger adder/subtractor facilitates the FMA instructions. IEEE 754 mandates that FMA instructions can only be rounded once at the end of the addition stage [10]. Therefore, the glue logic must propagate the full (2*MANT_WIDTH)-bit intermediate multiplication result to the adder.

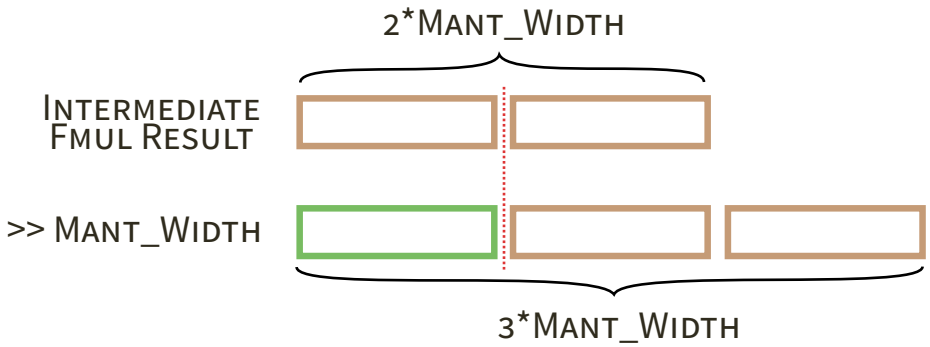

Figure 3.15: Floating-point adder's alignment may completely right-shift out the most significant mantissa bits.

Since the adder's alignment stage may right-shift $Mant_2$ by more than MANT_WIDTH bits, as visualized in Figure 3.15. We implement a (3*MANT_WIDTH)-bit adder/subtractor in order to account for both the intermediate and rounding bits. We have demonstrated in Section 2.4.4 that failure to include all intermediate and rounding bits in the addition stage can lead to catastrophic loss of precision. However, users may optionally configure our FPU to reduce the number of bits propagated to the adder, assuming they can ascertain application numerical accuracy is unaffected. This optimization decreases the size of the alignment shifter, comparator, and mantissa adder/subtractor significantly, and leads to lower resource utilization and better performance efficiency.

3.3.3 Floating-Point Divider

Typically, without considering exception handling, floating-point division consists of the following steps:

1. Pre-normalize RS_1 and RS_2 if they are denormal.
2. Calculate sign $Sign_{result} = XOR(Sign_1, Sign_2)$
3. Calculate exponent: $Exp_{result} = Exp_1 - Exp_2 + BIAS$.
4. Divide mantissas.
5. Normalize/denormalize.
6. Round.

The mantissa division module is the most complex step of the floating-point divide algorithm. We discussed in Section 2.4.2 that existing multiplier-based floating-point divider implementations are prohibitively large for integrated FPU designs. Therefore, we elect to implement the mantissa division core using a radix-2 integer divider.

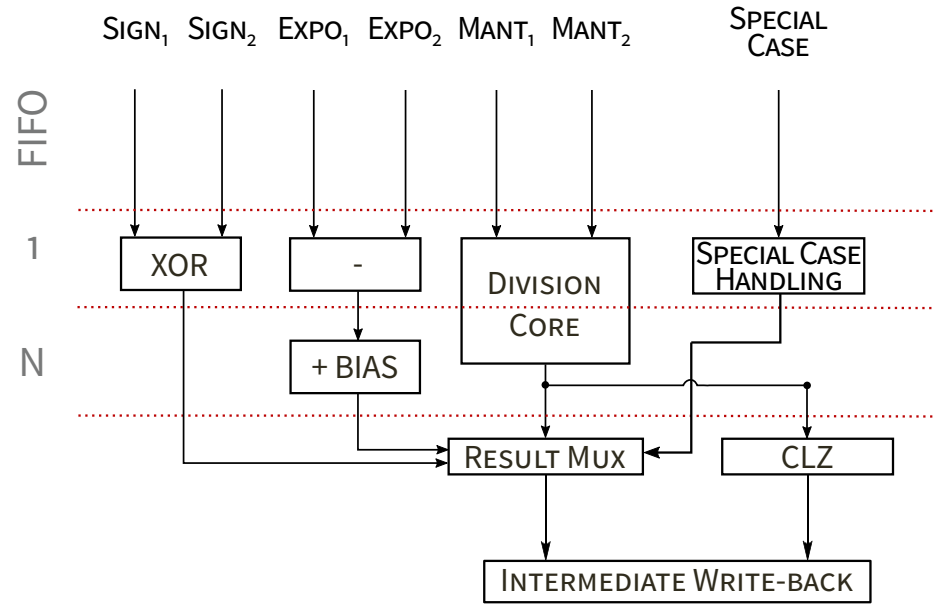

Figure 3.16: Floating-point divide uses a fixed-latency mantissa divider.

Given the mantissa divider's multi-cycle nature, a FIFO is used to store inputs, as shown in Figure 3.2.3. Figure 3.16 presents the rest of the pipeline. With the two inputs pre-normalized in the pre-processing stage, we calculate the result sign and exponent fields as outlined in step 2 and 3. The division core computes $Mant_1/Mant_2$ in fixed latency of N, where $N = MANT\_WIDTH + 2$. Two extra bits are calculated for rounding, and the sticky bit is produced by OR-ing the remainder field. We then compute the count-leading-zero of the divided mantissa. The results are multiplexed with special case output, and packed for post-normalization and rounding.

3.3.4 Floating-Point Square Root

Floating-point square-root consists of the following steps:

1. Pre-normalize RS_1 if it is denormal.
2. Result sign is always 0 (positive).
3. Calculate exponent: $Exp_{result} = Exp_1 \div 2$
4. Square-root mantissa.
5. Normalize/denormalize.
6. Round.

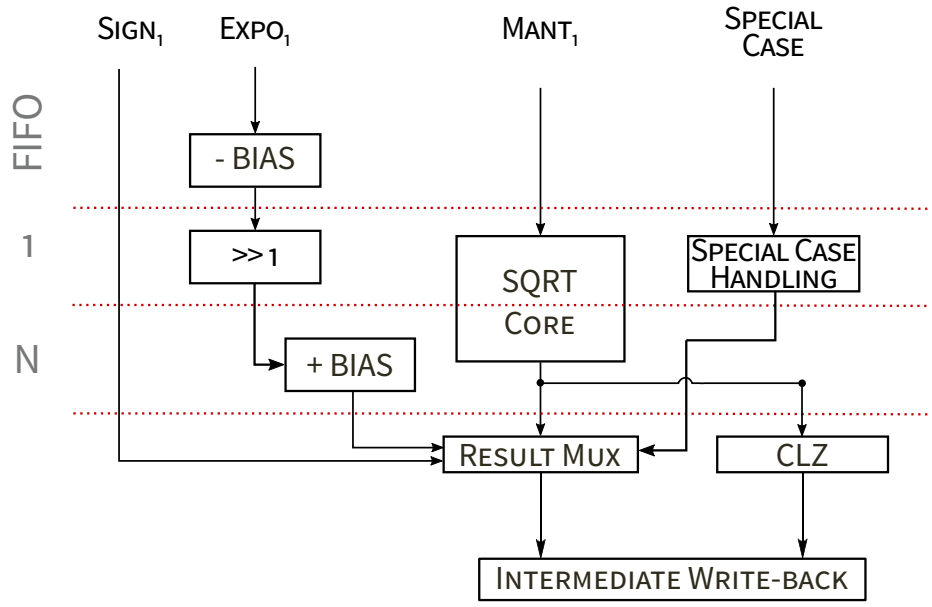

Figure 3.17: Floating-point square-root uses a fixed-latency mantissa square-root core.

Figure 3.17 presents the floating-point square-root pipeline. We note that the inputs are pre-normalized in the pre-processing stage before being stored in the FIFO. As negative inputs are invalid, the result sign is always logic low. The final exponent should be half of the input exponent, since

$$\sqrt{M * 2^E} = \sqrt{M} * 2^{\frac{E}{2}}$$

Therefore, we first un-bias the input exponent, right-shift it by 1 to compute its half, and re-bias it back to the floating-point format.

```
Iteration=Overflow=Subtractend=Quotient=0, Rad=Radicand

while(Iteration<Mant_Width+2)
   Subtractend={Subtractend[WIDTH-3:0], rad[WIDTH-1:WIDTH-2]}
   {Overflow, Sub}=Subtractend - {Q[WIDTH-3:0], 2'b01}
   Q=Q<<1
   Q[0]=Overflow ? 0 : 1
   Subtractend=Sub
   Iteration++
Remainder=Subtractend
```

Figure 3.18: Pseudo-code for the mantissa square-root algorithm

The mantissa square-root algorithm is shown in Figure 3.18. The algorithm has fixed latency of N, where $N = MANT\_WIDTH + 2$. Two extra iterations are needed to facilitate rounding. During each cycle, two most significant bits of the radicand is shifted into the subtrahend. The subtractor is computed by left-shifting the current root by two and setting the least significant two bits to $2'b01$. We perform the subtraction, and set the new root bit to 1 if the subtraction result if positive, and vice versa. After calculating the root and two rounding

bits, we generate the sticky bit by OR-ing the final subtractend. We then compute the count-leading-zero of the root mantissa. The results are multiplexed with special case output, and packed for post-normalization and rounding.

3.3.5 Floating-Point Units that Write-back to Floating-Point Register (WB2FP)

This section describes the pipelines of the instructions supported by the WB2FP unit:

- Integer to floating-point conversion.
- Floating-point minimum/maximum.
- Floating-point sign injection.

Integer to Floating-Point Conversion

To minimize resource utilization, we implement the integer to floating-point conversion using the post-normalization shifter. We first initialize the mantissa by placing the 32-bit integer in the least significant bits, and leaving the most significant bits as zeros. This is equivalent to right-shifting the the input integer by MANT_WIDTH bits. Therefore, we add MANT_WIDTH to the integer's initial biased exponent (1023 for DP floating-point).

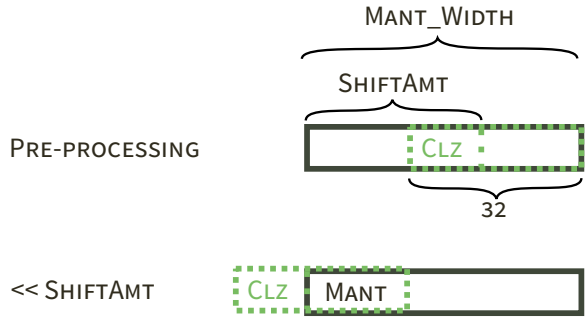

Figure 3.19: Integer to floating-point conversion uses the post-normalization shifter.

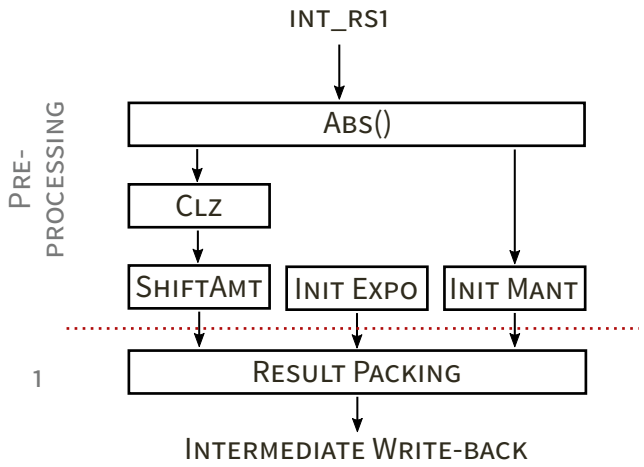

Figure 3.20: Integer to floating-point conversion pipeline.

To utilize the post-normalization shifter, we need to calculate the shift amount. Figure 3.19 visualizes the algorithm. We first compute the count-leading-zero of the magnitude of the input integer. We then generate the shift amount using

$$\textit{Shift Amount} = \textit{MANT\_WIDTH} - 32 + \textit{CLZ}_{\textit{Integer}}$$

, which gives us the count-leading-zero of the intermediate mantissa. The results are packed and propagated to the post-normalization unit where the intermediate mantissa is normalized to floating-point format. The bulk of the processing happens in the pre-processing stage, as shown in Figure 3.20.

Floating-Point Minimum and Maximum

The floating-point min/max instructions compare two floating-point numbers and return the smaller/larger one respectively. Fortunately, the sorting module already put the two input floating-point numbers in descending order by magnitude. Therefore, we only need to add little logic to determine the right output for the FMIN/FMAX instructions.

Table 3.5: Floating-point min/max logic table (assuming inputs are sorted in descending order by magnitude). The first row is the instruction type, and the first column is the RS_1's sign.

	FMAX	FMIN
+	RS_1	RS_2
-	RS_2	RS_1

Table 3.5 summarizes the FMIN/FMAX logic. Assuming $|RS_1| >= |RS_2|$:

- Instruction is FMAX: output RS_2 if RS_1 is negative, output RS_1 otherwise.
- Instruction is FMIN output RS_1 if RS_1 is negative, output RS_2 otherwise.

We note one exception to the logic table: RISC-V [1] mandates $+0 > -0$, therefore a special case is generated when both inputs are zeros.

Floating-Point Sign Injection

Table 3.6: Floating-point sign injection implementation.

Instruction	Implementation
FSGNJ	$\{Sign_2, Expo_1, Mant_1\}$
FSGNJN	$\{\sim Sign_2, Expo_1, Mant_1\}$
FSGNJX	$\{Sign_1 \hat{} Sign_2, Expo_1, Mant_1\}$

Floating-point sign injection instructions takes two inputs RS1 and RS2, and modifies RS1's sign bit using RS2's sign bit, as shown in Table 3.6:

- *FSGNJ* returns RS1's exponent and mantissa fields with RS2's sign bit.

- *FSGNJN* is similar, but result sign bit is the inverse of RS2's sign bit.
- *FSGNJX*'s result sign bit is equal to RS1 sign bit XOR-ed with RS2's sign bit.

3.3.6 Floating-Point Units that Write-back to Integer Register (WB2INT)

This section describes the pipelines of the instructions supported by the WB2FP unit:

- Floating-point to integer conversion.
- Floating-point comparison.
- Floating-point classification.

Floating-Point to Integer Conversion

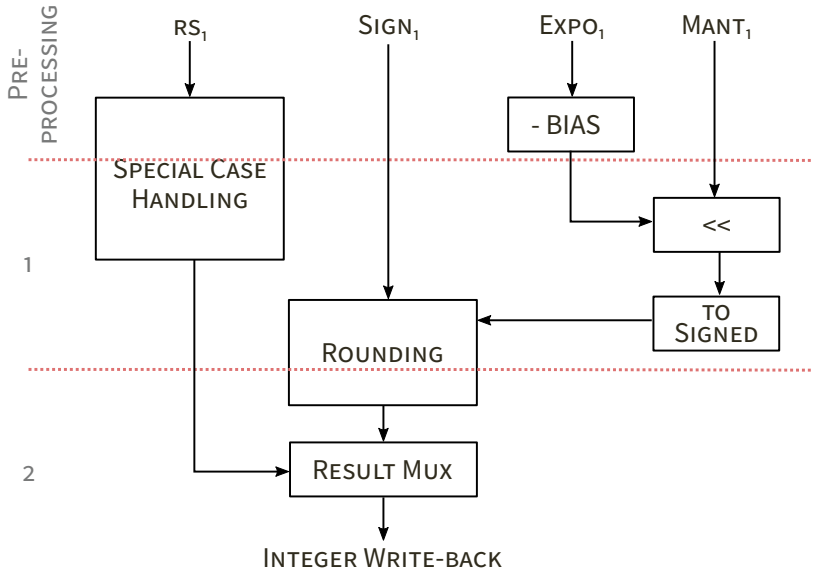

Figure 3.21: Floating-point to integer conversion pipeline.

Figure 3.21 provides an overview of the floating-point to integer conversion instructions' pipeline. We first obtain the unbiased exponent by subtracting the bias from the input exponent. We then left-shift the mantissa field by the maximum between the unbiased exponent and 31 to obtain the absolute value of the integer. The lower mantissa bits that are not part of the integer are used as rounding bits. We then convert the absolute value to signed integer if necessary, and round the result integer according to IEEE 754 rounding modes outlined in Section 2.2.

Floating-Point Comparison

```
DifferentSign
    CompareSign
SameSign
    PositiveInputs
        CompareMagnitude
    NegativeInputs
        CompareMagnitude
```

Figure 3.22: Floating-point comparison pseudocode.

Figure 3.22 provides the pseudocode for processing floating-point compare instructions, Floating-Point Less than (FLT), Floating-Point Less than or Equal to (FLE), and Floating-Point Equal to (FEQ). If the input operands have different signs, then the positive number is larger. Otherwise, the number with the greater magnitude is larger for positive inputs, and vice versa. Similar to FMIN/FMAX, we reuse the pre-processing stage sorting module to compare the magnitudes. We note that floating-point compare instructions must assert $+0 == -0$. This is different from FMIN/FMAX handling of ± 0 Section 3.3.5.

Floating-Point Classification

Floating-point classification is a simple instruction that returns the type of floating-point number to the integer register file. Its output format is shown in Table 3.7 [1]. We note that the classification result is delayed by one cycle to synchronize with the floating-point to integer pipeline, as discussed in Section 3.2.3.

Table 3.7: Floating-point classify instruction output format.

result bit	Meaning
0	input is -∞.
1	input is a negative normal number.
2	input is a negative subnormal number.
3	input is -0.
4	input is +0.
5	input is a positive subnormal number.
6	input is a positive normal number.
7	input is +∞.
8	input is a signalling NaN.
9	input is a quiet NaN.

3.3.7 Post-Normalization and Rounding Integration Scheme

We wish to provide more details regarding the shared post-normalization-and-rounding module in this section. Figure 3.23 visualizes:

- (a): a single write-back stage for all floating-point results.
- (b): separate write-back stages for floating-point load and arithmetic results.

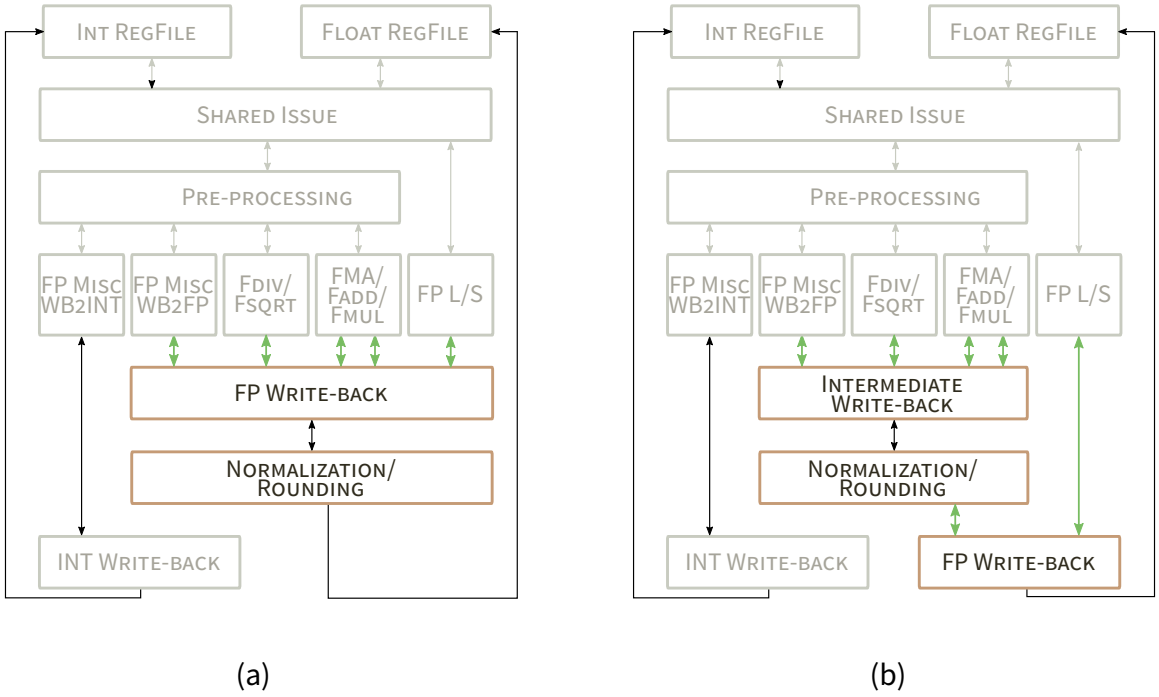

Figure 3.23: Post-Normalization and Rounding Integration: (a) single write-back interface. (b) separate write-back interfaces for arithmetic and memory instructions.

We integrated the post-normalization and rounding using scheme (a) in the early FPU designs. While scheme (a) enables the sharing of the post-normalization and rounding hardware, it inadvertently adds three additional cycles of latency to floating-point load instructions. The added latency can result in non-negligible performance degradation, as floating-point workloads can contain up to 84% memory operations [55]. As a result, we proposed design (b) which allows the load results to bypass the post-normalization-and-rounding module. Under (b), the intermediate write-back stage selects arithmetic result for post-normalization and rounding, while the final write-back stage commits floating-point numbers to the floating-point register file.

Design (b) uses 2% more LUTs, as it instantiates one extra write-back interface, as illustrated by the green arrows in Figure 3.23. Moreover, additional pipeline control logic is needed in the post-normalization-and-rounding module to ensure the correct intermediate result propagation. Furthermore, design (b) achieves 2% higher clock frequency. In terms of runtime performance, we find that bypassing load results can increase memory intensive workloads' IPC by 7.9%. We use scheme (b) in our final design, as it has similar resource utilization and clock frequency as (a) while improving IPC by a non-negligible amount.

3.3.8 Post-Normalization

Post-normalization is the process of converting floating-point results to the final format. For a normal floating-point number, this entails adjusting the exponent and shifting the mantissa fields so that the implicit leading mantissa bit is 1. On the other hand, negative exponent indicates that the intermediate result is denormal. This can occur during multiplication/division, and is handled by setting the normalized exponent to zero, and right shifting mantissa accordingly.

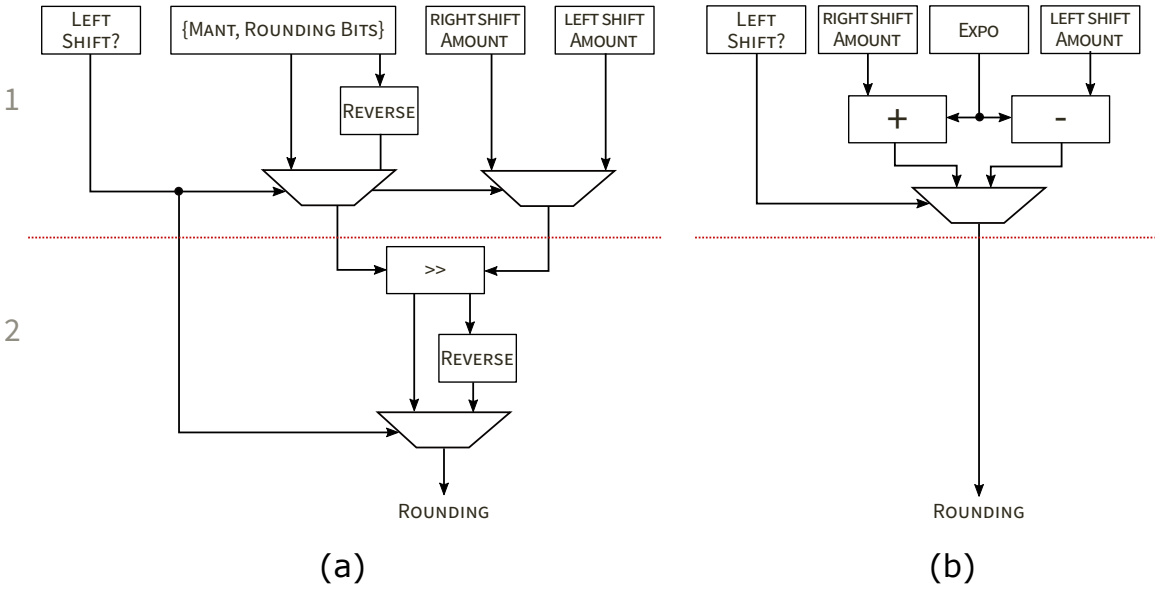

Figure 3.24: Floating-point normalization data-paths: (a) mantissa left and right shifting are implemented using a combined right-shifter. (b) the exponent decreases when left-shifting, and increases when right-shifting.

Figure 3.24 presents the normalization unit pipeline. The data-path is partitioned into two graphs for clarity: (a) mantissa shifting and (b) exponent normalization. In the mantissa data-path, we first obtain the shift amount by multiplexing between *right shift amount* and *left shit amount*. We then determine the shifter input by multiplexing between the concatenation of the un-normalized mantissa and rounding bits and its bit-reversal. The reversed bit field allows us to implement the left-shifting using the right-shifter, although the shifter output must be reversed back for left-shifting. Moreover, the exponent must be adjusted accordingly as the mantissa is shifted. In left-shifting, we compute the normalized exponent by subtracting shift amount from the input exponent. On the other hand, right-shifting increases the input exponent by shift amount. The normalized mantissa and exponent fields are then packed and propagated to the rounding stage.

3.3.9 Rounding Unit

The rounding unit implements the five rounding modes discussed in Section 2.2.4. Depending on the rounding mode and rounding bits, we generate a ”roundup” signal which is added to the mantissa to generate the final floating-point number. We calculate the *roundup* signal for each mode in parallel, and select the appropriate one using a multiplexer. Additionally, each mode handles overflow exception differently as we will

outline in this section. For the optional configuration where only the default rounding mode is supported, we assign the ”roundup” and overflow output to one generated by roundTiesToEven.

Round Ties to Even

RoundTiesToEven delivers the floating-point number closest to the infinitely precise result. We have defined how ”closeness” is measured in Section 2.2.4.

```
if(G&(R|S))
  roundup=1
else if(G&!R&!S)
  roundup=LSB
else
  roundup=0
```

Figure 3.25: Pseudo Code for round-ties-to-even.

Figure 3.25 provides the pseudocode for the rounding attribute roundTiesToEven. The intermediate result is rounded up if G is asserted, and (R | Sticky Bit (S)) evaluates to true. When there's a tie, i.e. GRS=3'b100, the floating-point number with an even (1'b0) LSB is delivered.

Round Ties to Away

RoundTiesToAway is similar to RoundTiesToEven, except it outputs the number with the larger magnitude when two numbers are equally near to the infinitely precise result. Three scenarios are considered:

- GRS < 3'b100: the unrounded result is nearest.
- GRS = 3'b100: there exists two nearest numbers, rounding up produces the one with the larger magnitude.
- GRS > 3'b100: the rounded-up result is nearest.

It can be seen that the intermediate result is rounded up if the guard bit is asserted. Thus:

$$roundup_{TiesToAway} = G \tag{3.1}$$

Round toward Positive

In roundTowardPositive mode, the intermediate result if rounded to the closest value no less than the infinitely precise result. This implies that negative values are not rounded, and positive values are always rounded up if any of the GRS bits are asserted. Therefore:

$$roundup_{towardPositive} = \sim sign \ \& \ |(\{G, R, S\}) \tag{3.2}$$

Round toward Negative

RoundTowardNegative is the opposite of roundTowardPositive. Positive values are not rounded, and negative numbers are rounded if G|R|S==1, as shown below:

$$roundup_{towardNegative} = sign \ \& \ |(\{G, R, S\}) \tag{3.3}$$

Round toward Zero

RoundTowardZero is effectively truncation, where all bits beyond the destination format's precision are discarded.

$$roundup_{towardZero} = 0 \tag{3.4}$$

Overflow Handling

Table 3.8 outlines the default overflow handling for each rounding mode.

Table 3.8: Default overflow exception handling for each rounding modes

Rounding Mode	Positive Overflow Value	Negative Overflow Value
roundTiesToEven/roundTiesAway	+∞	-∞
roundTowardPositive	+∞	format's most negative finite number
roundTowardNegative	format's most positive finite number	-∞
roundTowardZero	format's most positive finite number	format's most negative number

3.4 Summary

In this chapter, we described the FPU's integration to Taiga. We presented the four top-level stages in the FPU: issue, pre-processing, execution and write-back stages. We then provided implementation details of all floating-point execution units, including the post-normalization and rounding units. We also briefly discussed a few design choices, such as the instruction write-back priority, the FMA micro-architecture design, and the post-normalization-rounding integration scheme.

Table 3.9: Existing standalone floating-point data-paths: clock frequency, resource usage and latency.

Instruction	Freq (MHz)	LUTs	FFs	DSPs	BRAMs	Latency
FMA [15]	204	1700[1]	1915[1]	9	0	13
FADD, FSUB [14]	200	800[1]	607[1]	0	0	9
FMUL [14]	206	1221[1]	1000[1]	9	0	11
FDIV [2]	177	2540	1833	0	8	10
FSQRT [2]	179	6285	4025	0	58	10

Table 3.10: Our FPU: floating-point instructions' clock frequency, resource usage and latency.

Instruction	Freq (MHz)	LUTs	FFs	DSPs	BRAMs	Latency
FMA	132	1853	1314	9[2]	0	8
FADD, FSUB	170	1279	498	0		6
FMUL	132	900	772	9[2]		6
FDIV	217	279	422	0		60
FSQRT	185	275	342			60
FLD	116	NA[3]				4
FMIN, FMAX	489[4]	91[4]	79[4]			5
FSGNJ(N)(X)						
FCVT.D.(U)W						
FCVT.(U)W.D	306[5]	515[5]	67[5]			4
FLE, FLT, FEQ						
FCLASS						

Table 3.9 presents the clock frequency, resource usage, and latency for existing standalone arithmetic floating-point data-paths. Table 3.10 presents the same data for our implementation, along with the miscellaneous floating-point instructions. It can be seen from the tables that our FMA and FADD implementations achieve lower operating frequency and require higher resource utilization compared to the custom data-paths. This is expected as our FMA design supports denormal processing, whereas existing work [15] does not. More importantly, our FMA unit implements three instructions: FMA, FADD and FMUL. Therefore, the resource overhead can be attributed to the extra control logic as discussed in Section 3.3.2 and visual-

[1] Estimated from 4-LUTs.

[2] Shared.

[3] Cannot separate floating-point load-store data from that of the integer load-store.

[4] Reported WB2FP data as these instructions share most of the logic.

[5] Reported WB2INT data as these instructions share most of the logic.

ized in Figure 3.26 — the glue logic and FADD input multiplexers. Similarly, since the floating-point adder is shared by the FMA and FADD instructions, it must support the wider input of the FMA instructions. As such, Figure 3.26 visualizes the FADD's input width of 2*MANT_WIDTH, while a standalone floating-point adder processes MANT_WIDTH-bit inputs. In both cases, our designs have significantly lower latency, which is crucial for runtime performance for general software as discussed in 2.4.2. Specifically, our FMA implementation, the most frequently used instruction in floating-point applications, completes 5 cycles quicker than the existing solution [15]. Furthermore, our design focuses on minimizing resource usage for the less frequently used instructions floating-point divide and square-root. Compare to existing implementations, our FDIV and FSQRT designs require 9x and 22x fewer LUTs, although the lower area comes at a cost of higher latency. We discuss this trade-off further in Section 5.1.2.

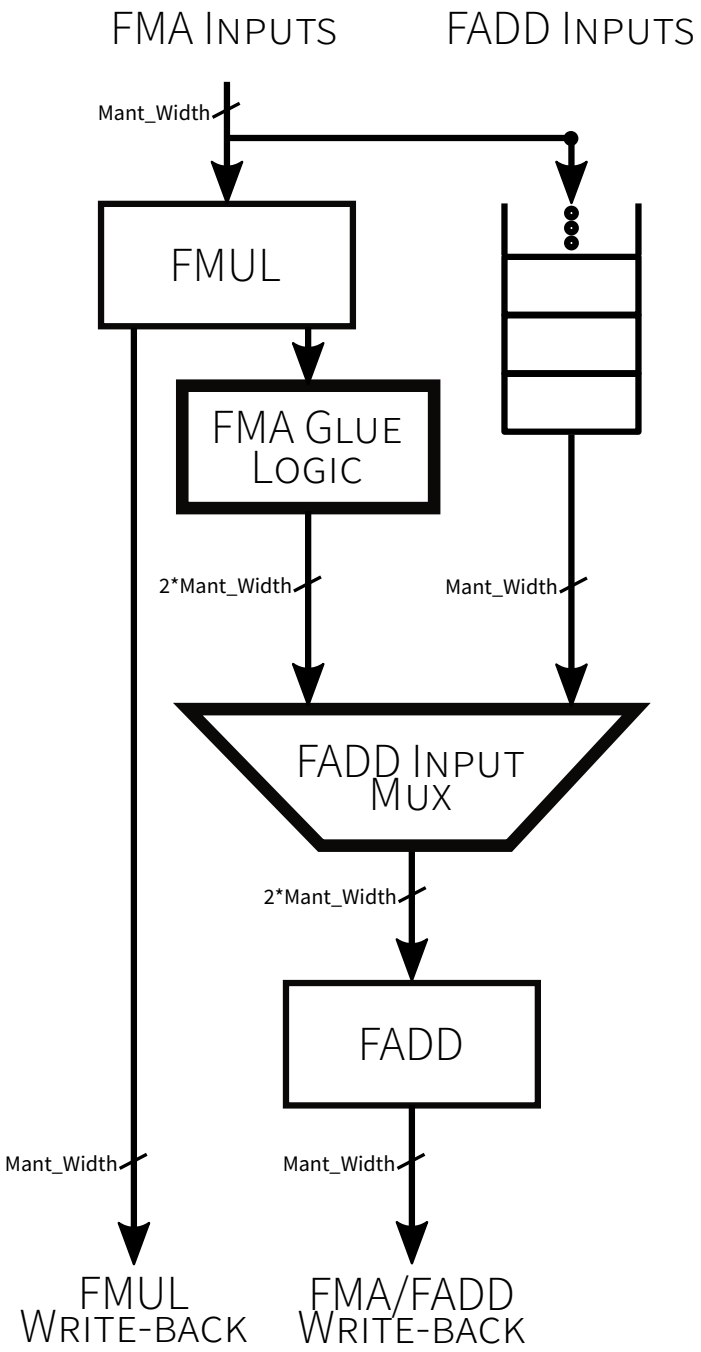

Figure 3.26: The FMA unit is larger due to the larger shared adder and control logic overhead.

4 Experimental Framework

This chapter provides an overview of the evaluation methods and experiments used to characterize the FPU implementation. First, we present the base Taiga configuration — into which we integrated the FPU, and the FPU design configurations to be evaluated. We then introduce the benchmarking methodology used to measure the FPU's run-time performance and performance efficiency, including a brief introduction of the workloads used, an overview of the evaluation platform, and our metrics for comparing among different implementation variations.

4.1 Evaluation Configurations

In this section, we first provide an overview of the base processor configuration that the FPU is integrated in. We then present a set of FPU variants to be tested. These variants include FPUs configured with different compliance levels and reduced precision. We also discuss a few architectural features that did not make into the final FPU.

4.1.1 Base Taiga

To minimize the base processor's impact on the FPU's performance, we configure Taiga with a high performance configuration as outlined by the creators [56] :

- 2-way set-associative, 512-entry branch predictor with an 8-entry Return Address Stack (RAS).
- Dedicated integer multiplier and divider.
- Maximum of 8 in-flight instruction IDs.
- 64 KB shared local memory.

We note that 64 KB of local memory is used for collecting Vivado place-and-route data. However, we increase the simulation local memory size to 16 MB to accommodate for the large benchmark applications in order to collect runtime performance data.

4.1.2 FPU Configurations

Due to the high configurability of the FPU, only a selected number of common options are tested. As such, this section presents the configurations we have chosen to evaluate in this work.

Reduced Compliance

Table 4.1: Reduced Compliance Feature List

	Full FMA Intermediate Representation	Denormal Support	All Rounding Modes
full-compliance [27]	✓	✓	✓
default-rounding-only [26] [57]	✓	✓	
no-denormal [26] [57] [52]	✓		✓
no-intermediate-fma [28]		✓	✓

We mentioned in Section 2.4.3, that many existing floating-point operators, commercial/open-source floating-point cores are only partially compliant. We have identified the following common FPU configurations outlined in Table 4.1, along with existing FPUs that resemble the respective options. *Full-compliance* supports all compliance features mandated by IEEE 754, and is used as the baseline FPU. The *default-rounding-only* FPU only supports the roundTiesToEven mode, which we presented in Section 3.3.9. *No-denormal* removes denormal number processing. This entails the removal of the pre-normalization shifters discussed in Section 3.2.2. Finally, *no-intermediate-fma* decreases the number of bits preserved in the FMA instructions' multiplication stage from 2*MANT_WIDTH to MANT_WIDTH+4 bits.

Reduced Precision

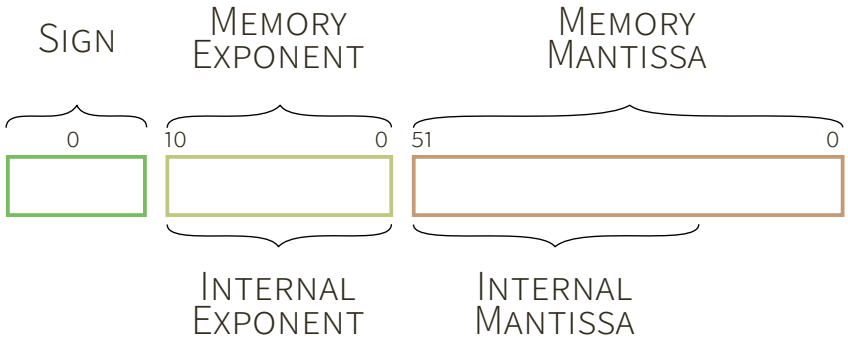

Figure 4.1: Reduced Floating-Point Format

We are interested in investigating the effect of running applications in lower precision, and this is realized by adjusting the mantissa width. Figure 4.1 visualizes the reduced precision format used for this work. Internally, the exponent width is fixed since its processing is significantly less intensive than that of the mantissa. Instead, the internal mantissa width is reduced, therefore decreasing the data-path, control logic, and reg-

ister file sizes. As such, we maintain the same dynamic range, while reducing the computation precision. Externally, the memory interface retains the same format as specified by IEEE 754. This is done so that the familiar software interface is supported, and existing binaries can be executed without re-compilation. We collect runtime and Vivado place-and-route data for mantissa widths in the range of $[10, 52]$.

Merged FMA Unit

Given the prevalence of FMA, FADD and FMUL instructions — up to 42%, 31%, and 24% in our benchmark workloads respectively, their designs have a significant impact on the FPU's performance. As such, we have chosen to investigate two different architectures for the FMA unit which implements the three critical instructions.

We discussed in Section 3.3.2 that the floating-point adder is shared by the FMA and FADD instructions, and that a FIFO is needed to mitigate the structural hazard due to simultaneous requests from FMA and FADD instructions. We can eliminate the FIFO by issuing FMA, FMUL and FADD instructions as the following:

- FMA: $\pm(rs1 * rs2) \pm rs3$ (unchanged)
- FADD: $1.0 * rs1 \pm rs2$
- FMUL: $rs1 * rs2 + 0$

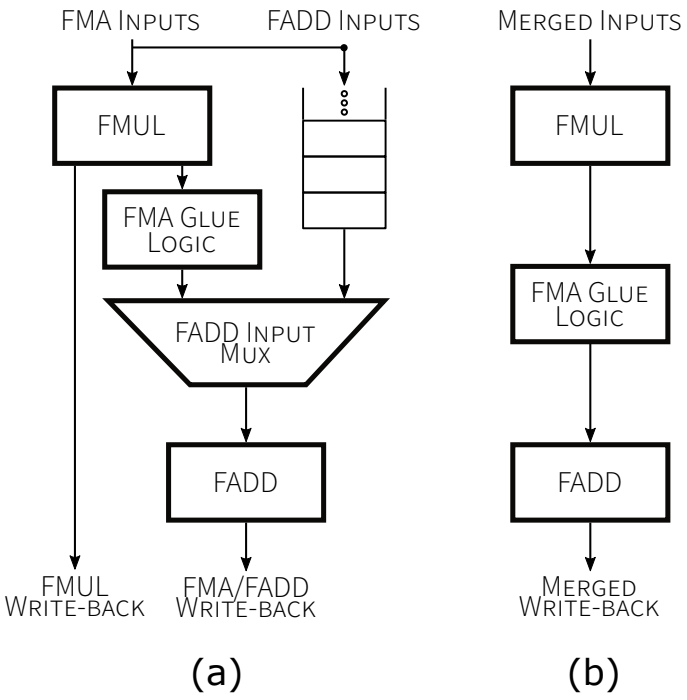

Figure 4.2: Compared to (a), design (b) removes the FADD FIFO, FMA glue logic, and the independent write-back interface.

Figure 4.2 visualizes the design differences. While (a) allows the floating-point adder to be shared by FMA and FADD instructions with the resource overhead FADD's input buffer. The single-pipeline design in (b) re-

quires less hardware, as it removes the FIFO and FMUL's write-back interface. However, design (b) adds additional latency to the FMUL and FADD instructions, as they now must propagate through the entire FMA pipeline.

4.2 Benchmarking Applications

In this section, we provide an overview of the applications used to benchmark the FPU, as well as the compliance test suite used to verify the floating-point implementation.

4.2.1 FPMark

Table 4.2: FPMark workloads.

Description	Name	Data Size	Iterations
Arctangent	atan	1k	10k
		64k	10k
Black Scholes	blacks	500x20	10
Horner's Method	horner	10k	1k
		1k	10k
Fast Fourier Transform	radix2	8k	10k
		2k	100k
Linear Algebra	linear_alg	100x100	50
		50x50	500
Enhanced Livermore Loops	loops	32	500
	inner-product	10k	400
		1k	1k
LU Decomposition	lu	200	1k
		20	10k
Neural Net	nnet	35	1k
Fourier Coefficients	xp1px	100	1k

We chose FPMark [55] as the benchmarking applications used to collect runtime data on the FPU. Table 4.2 presents an overview of the 16 workloads selected for this work. FPMark features a suite of 10 different kernels that can be configured to 53 workloads of varying memory/compute demands. To accommodate our soft-processor environment and accelerate data collection, we selected a subset of the workloads with moderate runtime (<30 minutes).

The FPMark workloads are compiled with self-verification mode turned on. We modified the verification/reporting infrastructures so as to exclude the verification function calls from the performance calculations. Otherwise the performance data will be inflated due to verification functions comprising of mostly integer instructions. Moreover, we extend the verification infrastructure to calculate the minimum, maximum, average, and standard deviation of the number of accurate bits. These additional data points enable us to perform statistical analysis, and quantify the quality of result produced by the various FPU variants. Furthermore, FPMark's provided reference data is generated without the FMA instructions. Since RISCV-GCC does not allow

the omission of FMA instructions, we had to re-generate the reference data as the use of FMA instructions can drastically affect the results produced. This is necessary since, unlike an FMUL followed by FADD, FMA instructions only round once, and therefore generate different rounding error for certain inputs. The rounding error then propagates in the algorithms, and can have a significant impact on computation result. As such, we re-compile and re-run the FPMark on a Linux machine (Ubuntu 18.04, Intel Xeon 4214 CPU) using GCC with FMA instructions enabled (gcc -mfma). For benchmarking our FPU, we compile the workloads with GCC 11.1.1 (-O3) to singled-threaded binaries, and then simulate execution using Verilator [58].

4.2.2 Imperas Compliance Tests

Table 4.3: Imperas RV32D compliance test suite data.

Category	**Instruction**	**# of Tests**
DP floating-point Computational	FADD	248
	FDIV	248
	FMADD	248
	FMIN	31
	FMSUB	248
	FMAX	31
	FMUL	248
	FNMADD	248
	FNMSUB	248
	FSQRT	248
	FSUB	248
DP floating-point Classify	FCLASS	31
DP floating-point conversion and move	FCVT.D.W	31
	FCVT.D.WU	31
	FCVT.W.D	248
	FCVT.WU.D	248
	FSGNJ	31
	FSGNJN	31
	FSGNJX	31
DP floating-point compare	FEQ	31
	FLE	31
	FLT	31
DP floating-point load and store	FSD	3317
	FLD	31

```
OK: 144/144
RISCV_TARGET=taiga RISCV_DEVICE=rv32d RISCV_ISA=rv32d
```

Figure 4.3: Imperas compliance test passed.

Full floating-point formal verification is an ongoing research topic in of itself, and is beyond the scope of this work due to the vast complexity inherent to floating-point operations. However, there are existing software compliance tests that perform some checks on the implementation. *RiscvOVPsimPlus*[59] contains a collection of compliance tests for the RISC-V ISA and its extensions. Table 4.3 presents the RV32D test details. The test suite drives various normal and edge case inputs to the FPU, and verifies standard conformance by comparing the output with reference signatures — both result and exception flags are verified. Figure 4.3 shows that our FPU passes all 144 compliance tests for the RV32D instruction set. We note that Imperas reports the RV32D test suite has a basic functional coverage of 91.02% [59].

4.3 Instrumentation

In this section, we provide details on how we quantify the FPUs' performance. We first introduce the firmware used to collect run-time metrics, including IPC, cycle count, number of instructions executed, and three selected trace signals used to analyze the FPU's performance. We then briefly discuss the environment used to collect timing and resource utilization data.

4.3.1 Runtime Performance

To compare the run-time performance of different FPU configurations, we need an accurate way to record application runtime. Since the applications are run bare-metal, we resort to RISC-V's performance counter Control and Status Registers (CSRs) to measure the number of instructions executed, and the number of clock cycles used by the benchmark applications. Taiga provides wrapper functions *start_profiling()* and *end_profiling()* to abstract the performance counter access to *instret* and *cycle* CSRs. They are 64-bit user-level, read-only counters specified in RV32I, which can be accessed using *CSRRS* instructions. The benchmark applications are modified to call the *start_profiling()* and *end_profiling()* routines at the beginning and end of executions respectively. The number of instruction executed and processor cycle counts are determined by computing the difference between ending and starting instruction/cycle counts.

Since we collect performance data using simulation, we implement a set of simulation trace signals to record the FPU's microarchitecture details. These signals enable us to analyze the performance differences among various FPU configurations. We collect a range of internal states, such as:

- The number cycles of *operand-stall*: captures pipeline stalls due to data dependency.
- The number cycles of *unit-stall*: captures pipeline stalls due to busy execution units.
- The number cycles of *write-back-stall*: captures floating-point arithmetic pipeline stalls due to prioritized memory operation in the write-back stage.

- Instruction mix.

4.3.2 Hardware Data

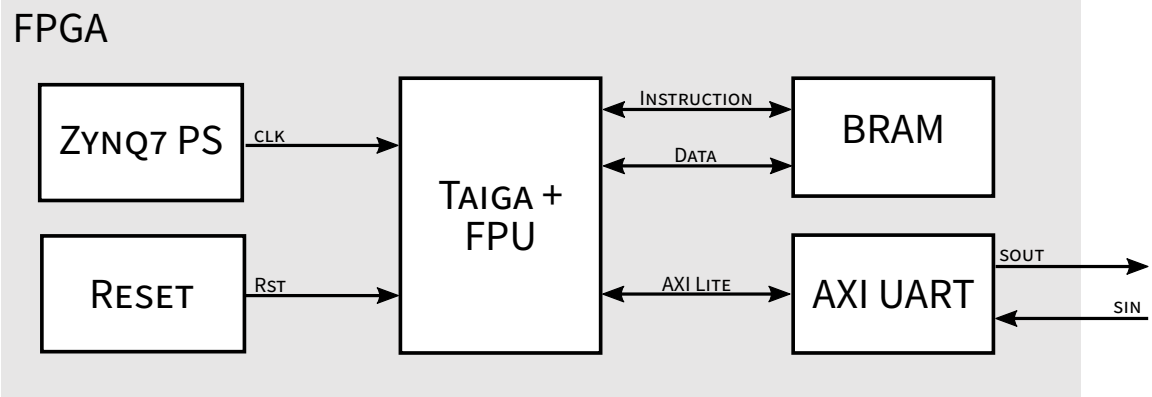

Figure 4.4: Taiga and FPU Block Design in Vivado

Figure 4.4 presents an overview of the overall system. The Zynq7 Processing System (Zynq7 PS) and Processor System Reset (Reset) provide clock and reset sources for the design. The FPU is integrated with Taiga, and has two memory interfaces with local instruction and data memory implemented using BRAMs. The processor communicates with external world through an AXI Universal Asynchronous Receiver/Transmitter (UART) IP core by Xilinx [60].

We use Vivado 2020.1 with default settings and run place-and-route for all FPU configurations on a Zedboard and U200. Collecting hardware data on the two boards allows us to learn about the FPUs's performance on both old and modern FPGAs. We note that the reported resource usage data includes the arithmetic hardware, as well as necessary instruction decoding, floating-point register file, and instruction management logic.

4.3.3 Metrics

Since this work targets FPGAs, we use the number of LUTs used by each FPU variant as an indicator of resource utilization, as LUTs are proportionally used (relative to device capacity) more than BRAMs and DSPs. Processors performance is often measured by throughput. Taiga's firmware reports the number of user instructions, and user cycle count, which can be used to calculate IPC:

$$\textit{Instruction per Cycle} = \frac{\textit{instruction count}}{\textit{cycle count}}$$

For configurations with differing clock frequencies, we can use Millions of Instructions Per Second (MIPS) to quantify the throughput difference:

$$\textit{Million Instructions per Second} = 10^6 * \frac{\textit{Instruction}}{\textit{Cycle}} * \frac{\textit{Cycle}}{\textit{Second}}$$

and finally, we use MIPS/LUT as a measurement of performance efficiency.

5 Experimental Results

In this chapter, we present the experimental results for the configurations mentioned in the previous chapter:

- *full-compliance*: baseline fully-compliant FPU.
- *default-rounding-only*: only default rounding mode supported.
- *no-denormal*: no denormal processing supported.
- *no-intermediate-fma*: FMA instructions do not preserve the full 2*mant_width-bit intermediate multiplication results.

Our evaluation of the different FPU variants includes their operating frequency, resource utilization, accuracy and throughput.

5.1 Compliance Variations

In Section 2, we discussed that few FPGA-based, RISC-V capable integrated FPUs are fully compliant. Most existing designs omit compliance features in order to achieve better performance and performance efficiency. Since our FPU allows users to optionally enable/disable certain compliance features, we provide a trade-off analysis of supporting these features.

5.1.1 Resource Usage and Clock Frequency

Table 5.1: Resource utilization and clock frequency of compliant/non-compliant FPUs on Zedboard.

Configurations	**Frequency (MHz)**	LUTs	FFs	DSPs	BRAMs
Taiga [56]	122.0	2503	1129	4	2
full-compliance	105.7	5423	3527	9	0
default-rounding-only	107.7 (+1.89%)	5427 (+0.03%)	3527	9	0
no-denormal	108.3 (+2.46%)	4873 (-10.14%)	3351 (-4.99%)	9	0
no-intermediate-fma	114.5 (+8.33%)	4048 (-25.35%)	2862 (-18.85%)	9	0

Table 5.2: Resource utilization and clock frequency of compliant/non-compliant FPUs on U200.

Configurations	**Frequency (MHz)**	LUTs	FFs	DSPs	BRAMs
Taiga [56]	369.1	2785	1711	4	2
full-compliance	323.6	5735	3575	9	0
default-rounding-only	341.5 (+5.53%)	5658 (-1.34%)	3557 (-0.5%)	9	0
no-denormal	343.4 (+6.09%)	5144 (-10.31%)	3381 (-5.43%)	9	0
no-intermediate-fma	340.0 (+5.07%)	4197 (-25.60%)	2894 (-19.05%)	9	0

Table 5.1 shows that *no-intermediate-fma* has the largest reduction in resource: 25.35% and 18.85% in LUTs and FFs on the Zedboard, respectively. We discussed in Section 3.3.2 that the fully compliant FMA addition stage must account for the (2*MANT_WIDTH)-bit mantissa generated by the floating-point multiplier. By decreasing the number of multiplication bits preserved, we significantly reduce the widths of the shifters, adders, and other logic needed to implement the compliant FMA instructions. Specifically, the floating-point alignment shifter can be reduced from 3*MANT_WIDTH to MANT_WIDTH+6 bits. Moreover, we can shrink the (3*MANT_WIDTH)-bit mantissa adder to MANT_WIDTH+3 bits. Additionally, the widths of intermediate write-back multiplexers and the post-normalization shifter also decrease as fewer rounding bits are propagated and processed.

Disabling denormal support sees a moderate improvement in resource utilization of 10.14% and 4.99% in LUTs and FFs, respectively. The reduction in resource usage can largely be attributed to the removal of the (MANT_WIDTH)-bit pre-normalization shifters. Moreover, while the *full-compliance* data-paths carry out exponent related computation using (EXPO_WIDTH+2)-bit vectors in order to support the negative denormal exponents, (EXPO_WIDTH)-bit exponents suffice for the *no-denormal* configuration. Configuration *default-rounding-only* has almost identical usage as *full-compliance*. This is expected since disabling non-default rounding modes removes little logic and a few multiplexers used to select the ”roundup” signals and the overflow results.

We observe similar resource usage trend on the U200, as shown in Table 5.2. The *no-intermediate-fma* design sees the most area reduction of 25.6% and 19.05% in LUT and FF requirements, respectively. Furthermore, *no-denormal* configurations uses 10.31% fewer LUTs and 19.05% fewer *FF*, compared to the baseline. Additionally, the *default-rounding-only* design is again almost identical to *full-compliance*, and has 1.34% and 0.5% reduction in LUT and FF usage, respectively.

With regard to operating frequency, Table 5.1 shows that *full-compliance*, *default-rounding-only*, and *no-denormal* configurations achieve almost identical frequencies on the Zedboard — 105.7 Mhz, 107.7 Mhz, and 108.3 Mhz, respectively. The rounding path is the critical path for both *Full-compliance* and *default-rounding-only* designs, although both configurations suffer from significant routing congestion — more than 81% of critical path delay. Moreover, as the data-paths shrink, the decode stage becomes the critical path for *no-denormal* with a routing delay of 77%. Due to Taiga’s extensive use of LUTRAMs for instruction and register management, the decode stage has often been the critical path throughout our design process. LUTRAMs have longer delays than FFs, and impose additional placement constraints on Vivado as not all LUTs can be

used as LUTRAM. Only *no-intermediate-fma* sees a non-trivial frequency increase of 8.33% compared to the baseline. Similar to *no-denormal*, the LUTRAM-heavy decode stage is in the critical path where routing delay consists of more than 81% of the total delay.

On the U200, the baseline *full-compliance* achieves 323.6 Mhz, representing a 206% increase over the frequency achieved on the Zedboard. Despite being almost identical, configuration *default-rounding-only* has a 5.53% higher operating frequency than the baseline. We believe this is due to Vivado's variability, as evidenced by the fact that the two designs have completely different critical paths:

- *full-compliance*: the single-cycle integer Arithmetic Logic Unit (ALU).
- *default-rounding-only*: floating-point intermediate write-back.

The random nature of the synthesis tool is furthered demonstrated by *no-denormal* configuration's 6.09% higher frequency compared to the baseline. While the achieved frequency is similar to *default-rounding-only*, *no-denormal*'s critical path is in the store-forwarding logic in the load-store unit. Furthermore, we expected *no-intermediate-fma* to have the best operating frequency due to its lower resource usage. However, it has the lowest frequency among the reduced compliance design as shown in Table 5.2. Examining the timing report, we notice that the rounding unit is back in the critical path. Although the three reduced compliance configurations have similar clock frequencies — within 1.5 MHz of each other, their critical paths are drastically different as a result of the tool's randomness. However, the commonality among these FPUs is high routing congestion.

Table 5.3: Resource utilization and clock frequency: *full-compliance*/*no-intermediate-fma* vs. NaxRiscv/VexRiscv targeting Zedboard.

Configurations	Frequency (MHz)	LUTs	FFs	DSPs	BRAMs
full-compliance	105.7	5423	3527	9	0
NaxRiscv [27]	98 (-7.28%)	5591 (+3.1%)	3194 (-9.44%)	9	0
no-intermediate-fma	114.5	4048	2862	9	
VexRiscv [28]	114 (-0.44%)	3779 (-6.65%)	3035 (6.04%)	9	0

Table 5.4: Resource utilization and clock frequency: *full-compliance*/*no-intermediate-fma* vs. NaxRiscv/VexRiscv targeting U200.

Configurations	Frequency (MHz)	LUTs	FFs	DSPs	BRAMs
full-compliance	323.6	5735	3575	9	0
NaxRiscv [27]	305.7 (-5.53%)	5977 (+4.22%)	3196 (-10.6%)	9	0
no-intermediate-fma	340.0	4267	2894	9	0
VexRiscv [28]	346.7 (+1.97%)	3714 (-12.96%)	3236 (+11.82%)	9	0

As discussed in Section 4.1.2, the NaxRiscv [27] and VexRiscv [28] FPUs resemble our *full-compliance* and *no-intermediate-fma*, respectively. Table 5.3 and Table 5.4 outline the resource utilization and clock frequency

of the four FPUs targeting the Zedboard and U200, respectively. Since the FPUs have similar relative frequency and resource utilization on the two FPGAs, we use the data generated on U200 for the following discussion.

It can be seen that NaxRiscv achieves a 5.53% lower clock frequency than our *full-compliance*, whereas VexRiscv observes a 1.97% higher frequency than *no-intermediate-fma*. We note that we chose the *FMax* configurations for both NaxRiscv and VexRiscv. As for utilization, we could not calculate the resource required by NaxRiscv and VexRiscv's floating-point instruction decode and issue logic, as the logic is embedded with the integer dispatch module. NaxRiscv uses 4.22% more LUTs and 10.6% fewer FFs than *full-compliance*, despite excluding decode-and-issue logic. Moreover, VexRiscv sees 12.96% decrease in LUTs usage but 11.82% higher FFs utilization than *no-intermediate-fma*. In addition to the decode and issue logic, *no-intermediate-fma*'s larger LUT usage can be attributed to different floating-point register file implementations. We use a 64-entry physical floating-point register pool to support register renaming, while VexRiscv has a 32-entry register file. Furthermore, our register files are mapped to LUTRAMs, whereas VexRiscv uses FFs. The different technology mapping thus explain the resource utilization variations.

5.1.2 Benchmarking Results

In this section, we present an accuracy and performance comparison for the four configurations for FPMark [55].

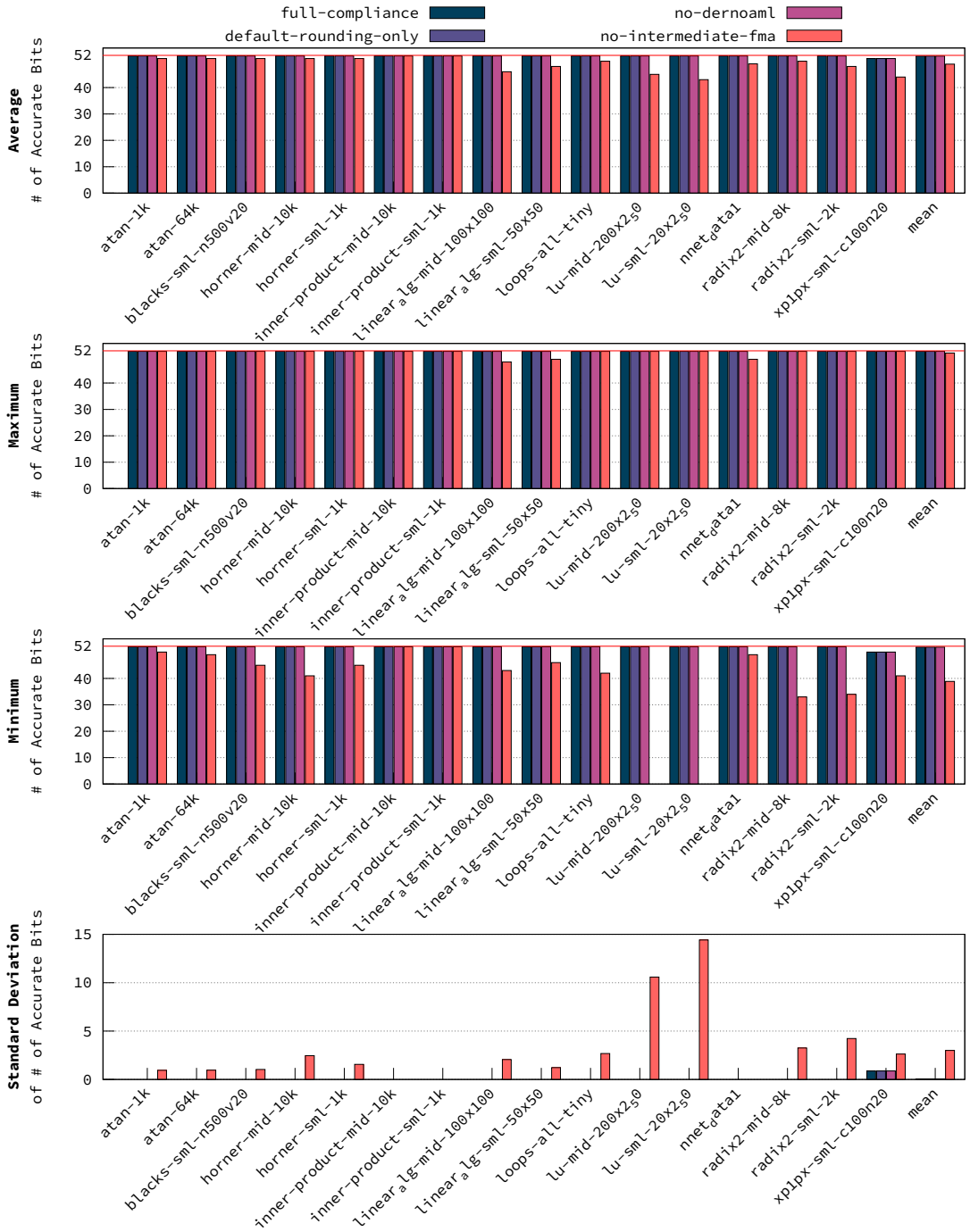

Figure 5.1: The FPMark's self-verification passes when average, maximum, and minimum number of accurate bits are 52 (DP).

Initial Verification

The Imperas compliance tests do not test denormal inputs, therefore both *full-compliance* and *no-denormal* pass all test cases. However, both *default-rounding-only* and *no-intermediate-fma* designs fail the compliance test, as all rounding modes and corner case FMA inputs are tested. In addition to the compliance tests, FPMark

performs self-verification that checks the accuracy of computed results against reference datasets. Figure 5.1 visualizes the average number of accurate bits generated by the four configurations for each workload. We notice that *no-denormal* and *default-rounding-only* see no impact on the average number of accurate bits generated by the FPUs for our workloads. Through our tracing infrastructure, we find that FPMark's workload utilizes the default roundTiesToEven rounding mode only. Additionally, our profiling shows that none of the workloads has either denormal input operands, nor produces denormal results. As removing these two compliance features have no impact on the computation accuracy, we omit them from the rest of the discussion.

In addition to presenting the average accuracy, we are interested in whether *no-intermediate-fma* decreases the accuracy uniformly across all computations. As such, we also present the minimum, maximum and standard deviation of the number of accurate bits. We observe that for most workloads, reducing FMA's intermediate representation results in a reduction in the average number of accurate bits. Moreover, the accuracy degradation affects computation unevenly, as evidenced by the increase in standard deviation of the number of accurate bits, as shown in Figure 5.1. We note that *nnet_data1* is an outlier: its accurate bit standard deviation remains zero, while *no-intermediate-fma* clearly decreased the number of accurate bits. This is because all computation, therefore all comparison points, for this workload sees a uniform reduction in the number of accurate bits.

Only workload *xp1px-sml-c100n20* fails the FPMark's verification using the *full-compliance* configuration. To better understand the accuracy loss, we perform additional profiling for this workload, and found that the algorithm diverges at *sin()* calls compared to running the application on a standard x86 server. Therefore, we locate the specific *sin()* function input with which our FPU disagrees with the server. We then simulate the *sin()* subroutine floating-point instructions individually, and compare those results with the ones computed by our server and find that our FPU and server generate the same intermediate results. This implies that our FPU delivers perfectly rounded results for any single basic operations per IEEE 754's mandate, as evidenced by the matching intermediate instruction results. However, the difference in the micro-architectures between our FPU and x87 can cause inconsistency in complex computations. We believe this specific instance is the result of x87's 80-bit floating-point registers, which allow floating-point numbers to accumulate in higher precision before they are stored in memory. Additionally, as VexRiscv resembles the *no-intermediate-fma* design, its computed results likely suffer similar accuracy degradation. We conclude that while we allow users to select configurations that are not fully compliant for improved performance, the decision to disable compliance features should be application specific as the impact on accuracy is highly dependent on the algorithm and input data.

Runtime Performance

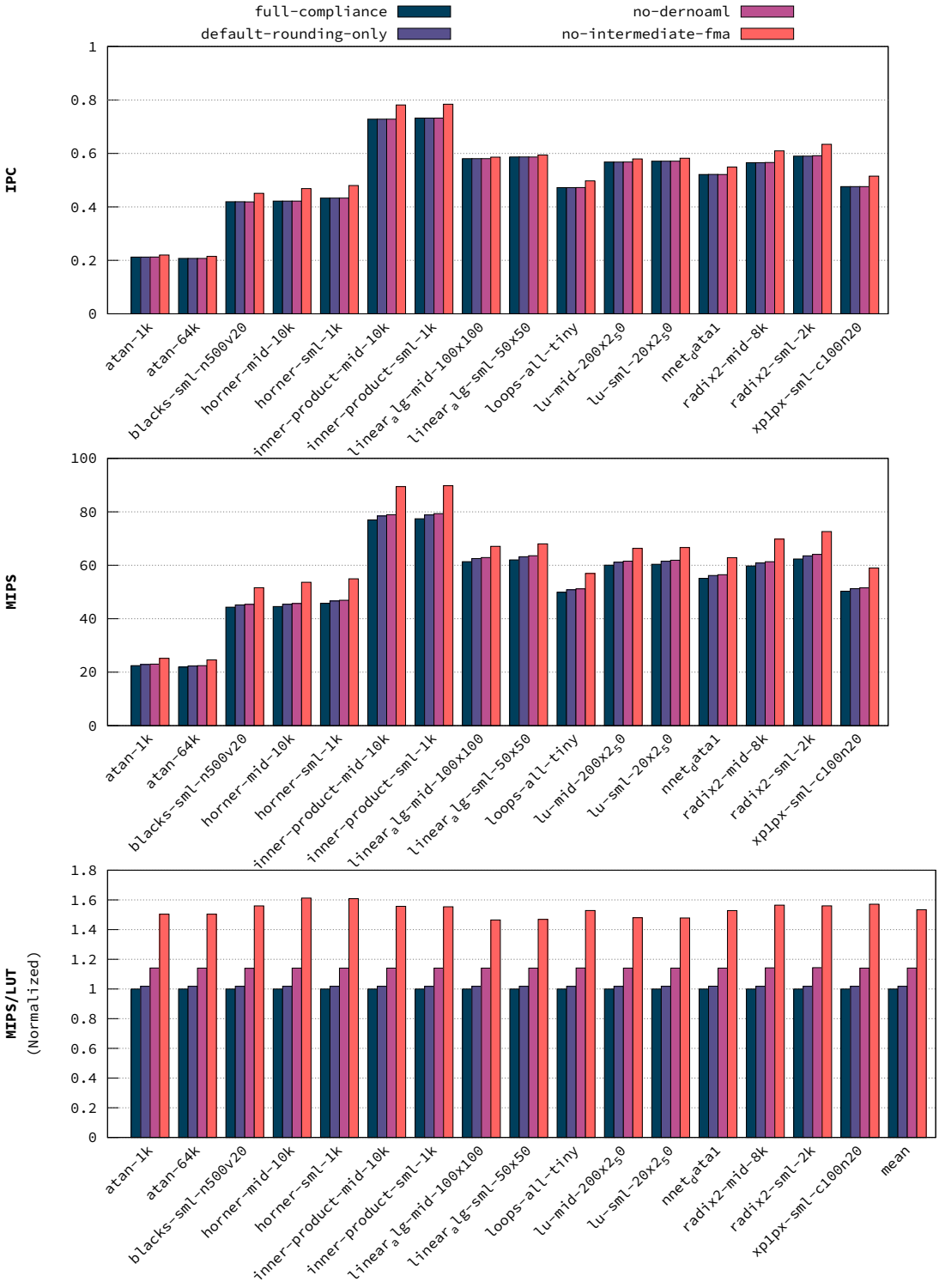

Figure 5.2: FPMark: IPC, MIPS, and MIPS/LUT (normalized to *full-compliance*)

Table 5.5: Reduced compliance runtime performance changes compared to base line *full-compliance* on Zedboard.

Configuration	**IPC**			**MIPS**			**MIPS/LUT**		
	max	min	avg	max	min	avg	max	min	avg
default-rounding-only	0.1%	0.0%	0.0%	1.9%	1.9%	1.9%	1.9%	1.8%	1.8%
no-denormal	0.3%	0.0%	0.0%	2.8%	2.4%	2.5%	14.4%	13.9%	14.1%
no-intermediate-fma	11.1%	1.0%	5.7%	20.4%	9.3%	14.5%	61.2%	46.5%	53.4%

We visualize the IPC, MIPS, and MIPS/LUT data for the four compliance configurations in Figure 5.2. Table 5.5 also presents each reduced compliance design's runtime performance changes compared to the base line *full-compliance* configuration.

It can be seen that workloads executed using *default-rounding-only* and *no-denormal* all have the same IPC as those executed using *full-compliance*. This is expected since all three designs have the same latency. Furthermore, workloads executed using *no-intermediate-fma* configuration see up to 11.1% increase in IPC (horner-mid-10k). On average, *no-intermediate-fma* results in a 5.7% IPC improvement. This is because reducing the number of intermediate bits in the FMA data-path allows us to convert the 2-stage pipelined post-normalization shifters to combinational ones. As a result, floating-point arithmetic instructions have one cycle lower latency in *no-intermediate-fma*.

Since *full-compliance*, *default-rounding-only*, and *no-denormal* configurations achieve largely identical IPC as *full-compliance*, we expect their throughout to increase the same amount as their respective operating frequency gain. Table 5.5 shows that *default-rounding-only* and *no-denormal* see an average MIPS increase of 1.9% and 2.5%, respectively. These throughput increases thus are attributed to their clock frequency improvement of 1.9% and 2.5% respectively, as presented in Table 5.2. Moreover, we observe a substantial MIPS increase in *no-intermediate-fma*, as a result of its improved IPC and higher operating frequency compared to its peers. Workload *horner-mid-10k* sees the highest increase in throughput of 20.4% over the baseline, and the *no-intermediate-fma* design results in a 14.1% MIPS increase on average.

Figure 5.2 also presents the normalized MIPS per LUT. It's clear that *default-rounding-only* has almost the identical performance efficiency as the baseline, while *no-denormal* and *no-intermediate-fma* have significantly better performance per resource improvements. On average, disabling denormal support increases MIPS/LUT by a maximum of 14.4%, and on average of 14.1%. Furthermore, *no-intermediate-fma* has a maximum performance efficiency improvement of 61.2% (*horner-mid-10k* and *horner-sml-1k*), and an average improvement of 46.5%.. The increased efficiency can be attributed to the resource reduction discussed in the previous section.

We were not able to benchmark the VexRiscv and NaxRiscv FPUs, as neither project provided up-to-date/functional documentation on running custom applications either in simulation or in hardware with FPU enabled. However, we can draw some general performance estimations based on VexRiscv's instruction latency and clock frequency.

Table 5.6: Our *no-intermediate-fma* design has lower latency for the most frequently used floating-point instructions.

Instruction	Our FPU	VexRiscv	Max %	Min %	Avg %
FLD	4	9	66.4%	25.0%	45.0%
FMA	8	15	41.8%	2.9%	23.0%
FADD, FSUB	6	9	33.0%	0.0%	9.9%
FMUL	6	10	24.3%	0.0%	8.0%
FDIV	60	34	3.7%	0.0%	0.6%
FSQRT	60	62	0.2%	0.0%	0.0%
FMIN, FMAX	5	6	0.0%	0.0%	0.0%
FSGNJ, FSGJN, FSGNJX	5	6	3.4%	0.0%	1.0%
FCVT.D.(U)W	5	9	1.4	0.0	0.3
FCVT.(U)W.D	3	5			
FLE, FLT, FEQ	3	5	4.0%	0.0%	1.5%
FCLASS	3	NA	0.0%	0.0%	0.0%

Table 5.6 compares our *no-intermediate-fma* FPU's latency to that of VexRiscv, and outlines each instruction as a percentage of the total number of floating-point instructions. We note that the latency is calculated from the instruction issue stage to write-back. It can be seen that our FPU processes floating-point instructions in fewer clock cycles for all but floating-point divide. Notably, our FPU has lower latency for the most frequently used instructions:

- Load: five cycles faster.
- FMA: seven cycles faster.
- FADD/FSUB: three cycles faster.
- FMUL: four cycles faster.

With the four instructions make up to 66%, 41%, 33%, and 24% of FPMark [55] applications respectively, we can reasonably conclude that our low-latency design has materially better IPC. While VexRiscv's floating-point divider completes in half as many clock cycles as our FPU, floating-point divide consists at most 3.7% of FPMark's workloads. Therefore, the low-latency divider offers little runtime performance benefit. Moreover, although we could not find NaxRiscv's microarchitecture details, its FPU is built upon VexRiscv's FPU. Therefore, NaxRiscv's FPU has at best the same instruction latency as VexRiscv. Moreover, as we have mentioned in the previous section, NaxRiscv is frequency-optimized. This design methodology likely introduced deeper pipelines which would decrease its IPC and MIPS.

5.1.3 Summary

In this section, we explored four FPU designs with four different levels of compliance. Using the fully compliant FPU as the base line, we analyzed the impact of each reduced compliance configuration. Additionally, we compared two of our designs against two existing open-source FPUs, NaxRiscv [27] and VexRiscv [28].

Compared to the base processor Taiga [3], the *full-compliance* design saw a 13% reduction in frequency, and 116.7% and 212.4% increase in LUT and FF usage respectively on a Zedboard. When comparing against the *full-compliance* design, we found that the *default-rounding-only* configuration is virtually identical to the base line. Moreover, disabling denormal processing saw minimal operating frequency gain, while the design required 10% less LUTs, which resulted in an average MIPS and MIPS/LUT improvement of 2.5% and 13.9%. Furthermore, our more performant *no-intermediate-fma* had 8.33% higher frequency and 25.4% fewer LUTs. These improvements meant that the design had an average increase of 11.1%, 9.4%, and 46.5% in IPC, MIPS, and MIPS/LUT, respectively. Additionally, our *full-compliance* saw a 7.28% higher frequency, and uses 3% less LUTs, compared to NaxRiscv. Since NaxRiscv's resource utilization data did not include floating-point instruction decode and issue, their resource overhead would be higher in practice. On the other hand, although our *no-intermediate-fma* required 13% more LUTs than VexRiscv, the extra LUTs can be attributed to:

- VexRiscv's data did not include decode and issue either.
- VexRiscv implemented a 32-entry floating-point register file, whereas ours used a 64-entry one for register renaming.
- VexRiscv's register file were mapped to FFs, whereas ours mapped to LUTRAMs.

Moreover, our designs had substantially lower latency for all floating-point instruction, except floating-point divide, as shown in Table 5.6. This allowed us to estimate that our designs would have significantly better runtime performance, since the performance critical instructions — FLD, FMA, FADD/FSUB, and FMUL all completed in fewer cycles.

In terms of computation accuracy, since FPMark [55] does not utilize non-default rounding modes or denormal processing, only *no-intermediate-fma*'s results saw degraded accuracy/precision. We noticed that while most workloads still produce meaningful results despite the lack of FMA intermediate representation, workloads *lu-mid-200x2_50* and *lu-mid-20x2_50* observed complete loss of precision.

5.2 Reduced Precision Variations

5.2.1 Resource Usage and Clock Frequency

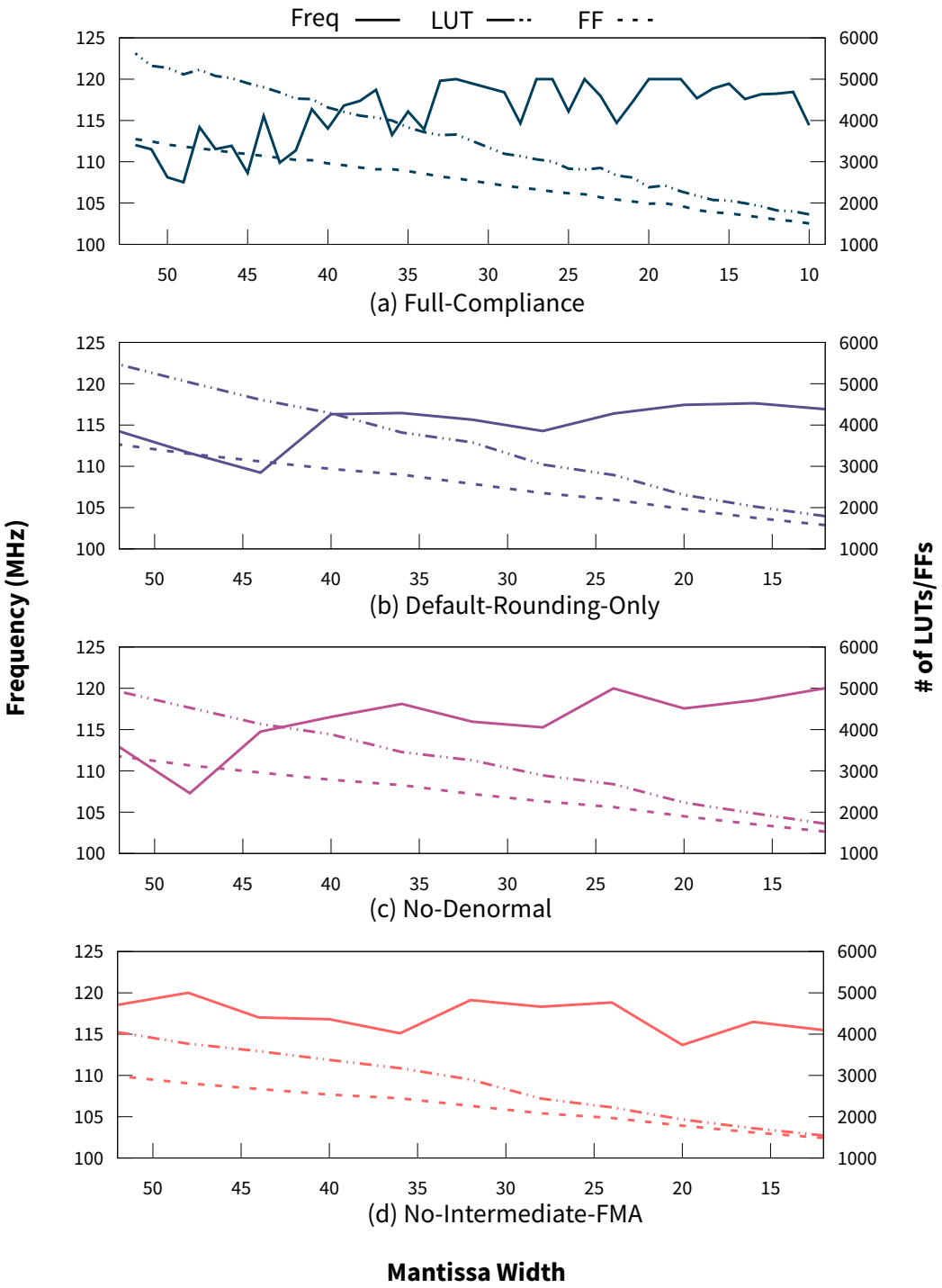

Figure 5.3: Reduced Precision: Clock Frequency and Resource Utilization on Zedboard.

Table 5.7: DSP usage decreases slowly as mantissa width decreases on Zedboard.

Mantissa Range	DSP Usage
[52:42]	9
41	8
[40:37]	5
[36:25]	4
24	3
[23:20]	2
[19:10]	1

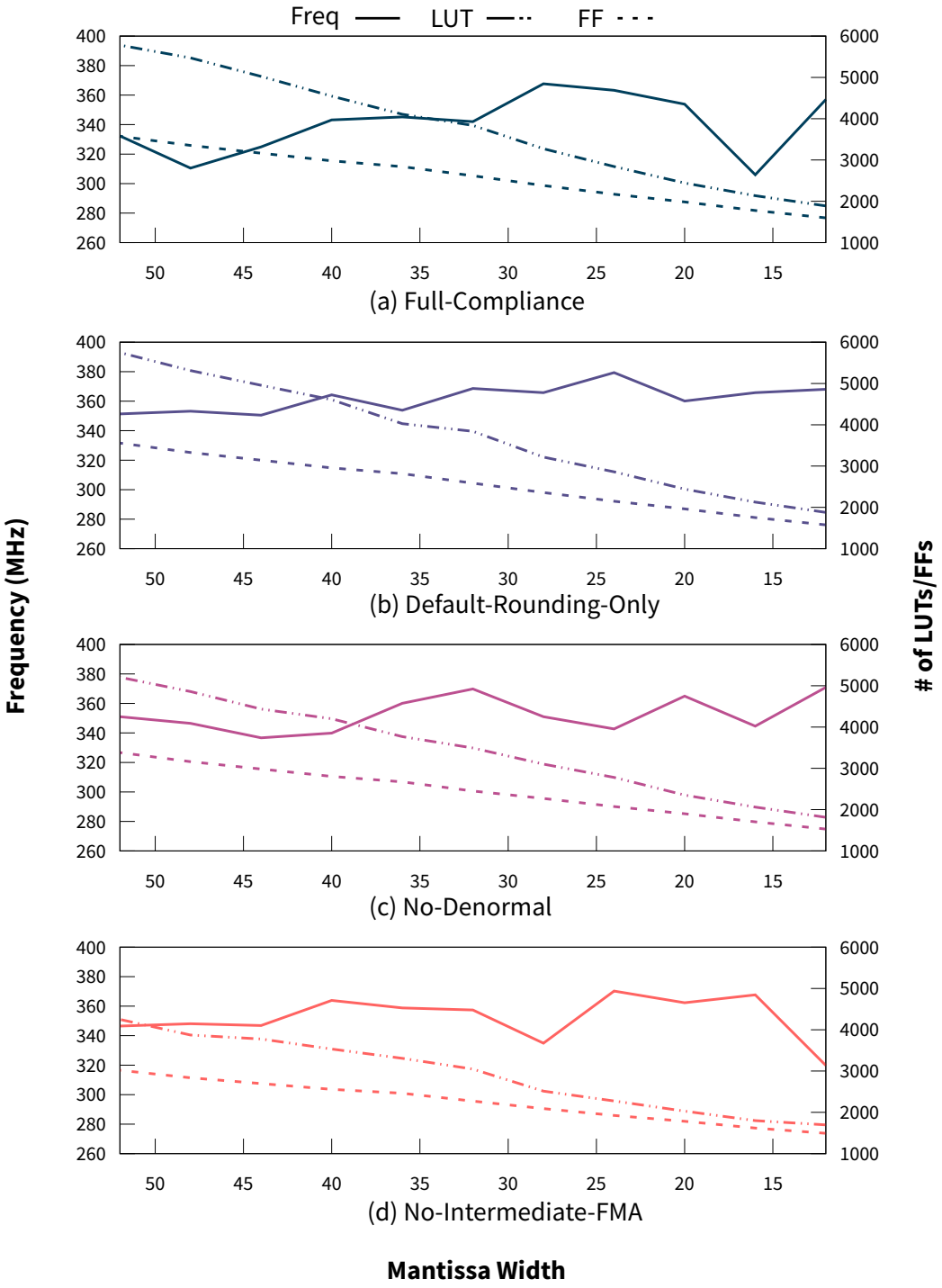

Figure 5.4: Reduced Precision: Clock Frequency and Resource Utilization on U200.

Table 5.8: DSP usage decreases slowly as mantissa width decreases on U200.

Mantissa Range	DSP Usage
[52:43]	9
[42:37]	5
[36:26]	4
[25:20]	2
[19:10]	1

It is common to trade-off precision and accuracy to performance and performance efficiency. In this section, we present our experimental data of running floating-point workloads using reduced precision FPUs. We note that in order to automate the data collection process, each reduced precision configuration targets 120 MHz on the Zedboard and 400 Mhz on the U200 during place-and-route. While the higher target frequency slightly inflates the resource usage as Vivado compensates for the tighter timing constraint, the trend should still reflect how FPU's hardware performance scales with mantissa width.

Figure 5.3 and Table 5.7 visualize the scaling of resource usage and operating frequency of FPUs on Zedboard, as mantissa decreases from 52 bit to 10 bits. Since the data trends are similar, our discussion focuses on the *full-compliance* FPU and its reduced precision configurations. In order to expedite data collection for the other configurations, we present their data in the mantissa width range of [52:12] in steps of 4 bits. It can be seen that both LUT and FF usage scales linearly with the mantissa width, while DSP usage decreases slowly. The 52-bit configuration uses 5626 LUTs, whereas the 10-bit mantissa variant uses 1726 LUTs, representing a significant 69.3% decrease in resource utilization. The reduction in resource usage is the result of the smaller adders, shifters, register file, and control logic widths as mantissa width decreases. We note that the relatively smooth LUT and FF utilization curves indicate that Vivado always maps the mantissa multiplier to DSPs, instead of the fabric, even though many bits may be left unused. This is due to DSPs having routing resources independent to the fabric routing, and their usage helps alleviate Vivado's timing constraints. Moreover, the multiplier output registers can be absorbed by the DSPs, further reducing the resource usage. Additionally, Figure 5.4 and Table 5.8 present the same data for the U200. The resource usage on the more modern FPGA has the same scaling with a peak-to-trough LUT utilization reduction of 69.4%.

Conversely, the operating frequency graphs show substantially more variations. Notably, the 49-bit-mantissa FPU has the lowest frequency of 107 MHz and 283 MHz on the Zedboard and U200 respectively, representing a 4.4% and 18% reduction on the respective FPGAs. Moreover, the configurations with the 10-bit data-paths achieve only a 1.7% and 8.1% higher frequency over the Double-Precision (DP) design on Zedboard and U200 respectively, despite using 69% less resources. We believe the unexpected timing trend can be attributed to two factors:

- Vivado's variability.
- Taiga's heavy use of LUTRAMs.

FPU configurations with similar mantissa widths, i.e. the 52-bit and the 49-bit mantissa FPUs, have similar resource requirements. The non-trivial frequency difference between two otherwise nearly identical FPUs can therefore be explained by the placer's variability, which affects the downstream routing performance. Additionally, contrary to our expectation, FPUs with significantly smaller resource footprint does not translate to meaningfully better operating frequency. Further examining the timing reports, we see that routing consists of 80% and 65% of the critical path delay among virtually all reduced precision configurations on the Zedboard and U200, respectively. Moreover, we note that the decode-and-issue stage is often in the critical path. As we discussed in Section 3.2.3, Taiga utilizes LUTRAMs to manage instruction IDs and register files. However, LUTRAMs have worse timing performance than FFs, and their usage poses additional placement constraints on Vivado, as not all LUTs can be mapped to LUTRAMs. Given the additional LUTRAMs needed to manage the floating-point instructions, Vivado struggles to find an optimal design, which limits the clock frequency of the overall system.

5.2.2 Benchmarking Results

Accuracy

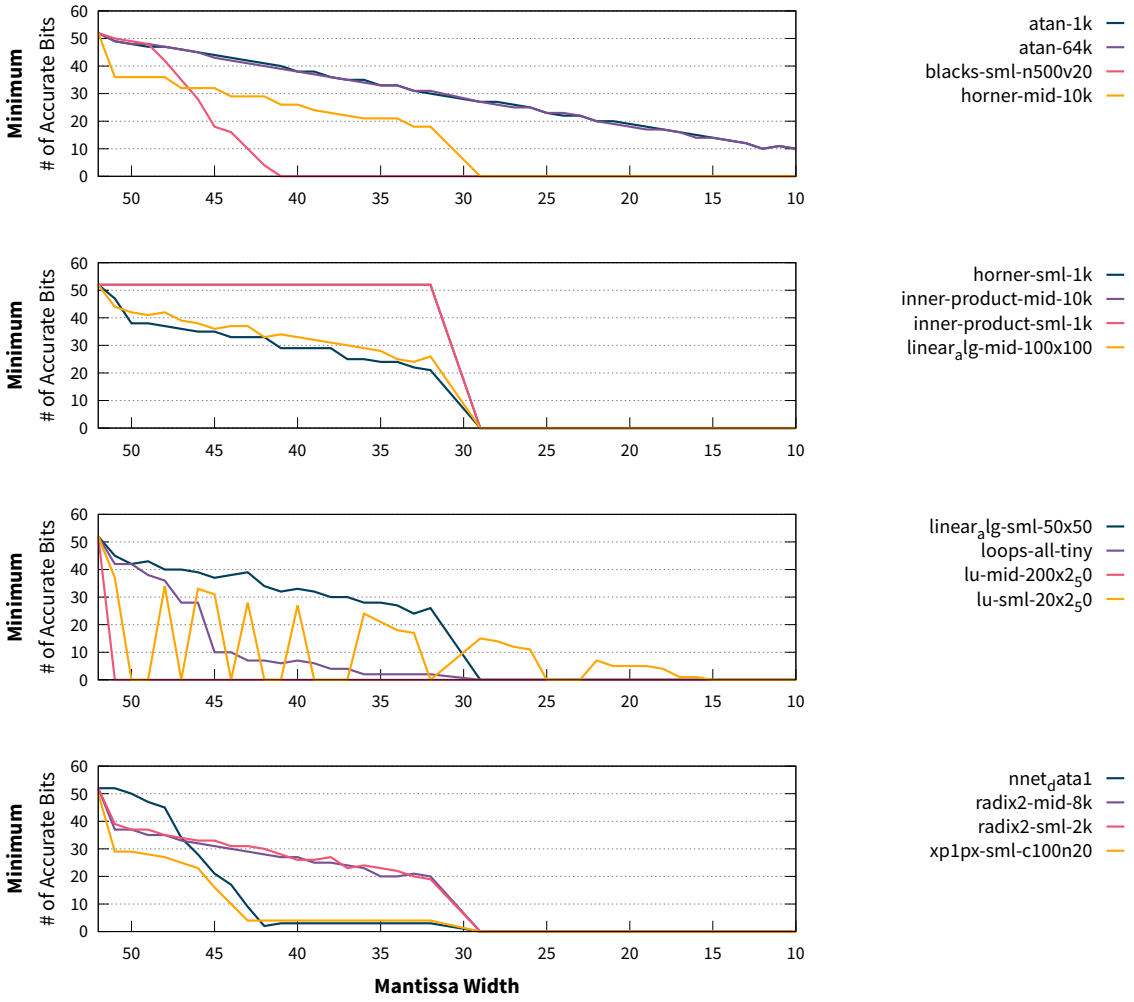

Figure 5.5: Minimum number of accurate bits generated by reduced precision configurations of the *full-compliance* FPU.

FPMark [55] contains scientific computing workloads, therefore, it is expected that executing these application with reduced precision will result in degradation of accuracy and precision. We present the minimum numbers of accurate bits produced by our *full-compliance* FPU as mantissa width decreases in Figure 5.5, since computation results are as accurate as their least accurate data point. Although all workloads generally see positive correlation between accuracy and mantissa width, the exact scaling heavily depends on the algorithms. It can be seen from Figure 5.5 (a) that workloads *atan-1k* and *atan-64k* experience linear scaling between mantissa width and accuracy, whereas *blacks-sml-n500v20* scales poorly and produces completely meaningless results when mantissa width is ≤ 42. Examining the source code, *atan-1k* and *atan-64k* com-

pute the arctan function using a telescoping series:

$$arctan(x) = x * \frac{P(x^2)}{Q(x^2)}$$

, where P and Q are polynomials. On the other hand, *blacks-sml-n500v20* computes the equations:

$$C = SN(d_1) - Ke^{-rt}N(d_2)$$
$$d_1 = \frac{\ln\frac{S}{K} + (r + \frac{\sigma^2}{2}t)}{\sigma\sqrt{t}}$$
$$d_2 = d_1 - \sigma\sqrt{t}$$

It is obvious that *blacks-sml-n500v20* computes significantly more intermediate results, which are reused throughout the algorithm. Any relatively small initial error, therefore, accumulates and causes drastic degradation to accuracy to the overall application.

Furthermore, input data also plays a role in computation accuracy. Workload *inner-product-sml-1k* computes the inner product of two vectors. Due to its highly sparse input data, the output contains only two non-zero floating-point numbers. This causes the unusually high average number of accurate bits. However, the computation becomes meaningless when the mantissa width is ≤ 32, as shown in Figure 5.5 (b). Moreover, despite being the same algorithm, workload *lu-mid-200x2_50* sees a rapid drop in the minimum number of accurate bits, while *lu-sml-20x2_50* has extreme fluctuation as the mantissa width decreases. This indicates that the algorithm is extremely sensitive to input data and reduced precision.

Runtime Performance

Two factors can affect the workloads' IPC as the mantissa width decreases:

- FDIV, FSQRT latency.
- Program execution path.

As discussed in Section 3.3.3, the floating-point divide and square-root are multi-cycle, and their latency decreases with mantissa width. Moreover, execution using reduced precision may change floating-point workloads' control flow, as intermediate values change due to the error introduced by smaller mantissa widths.

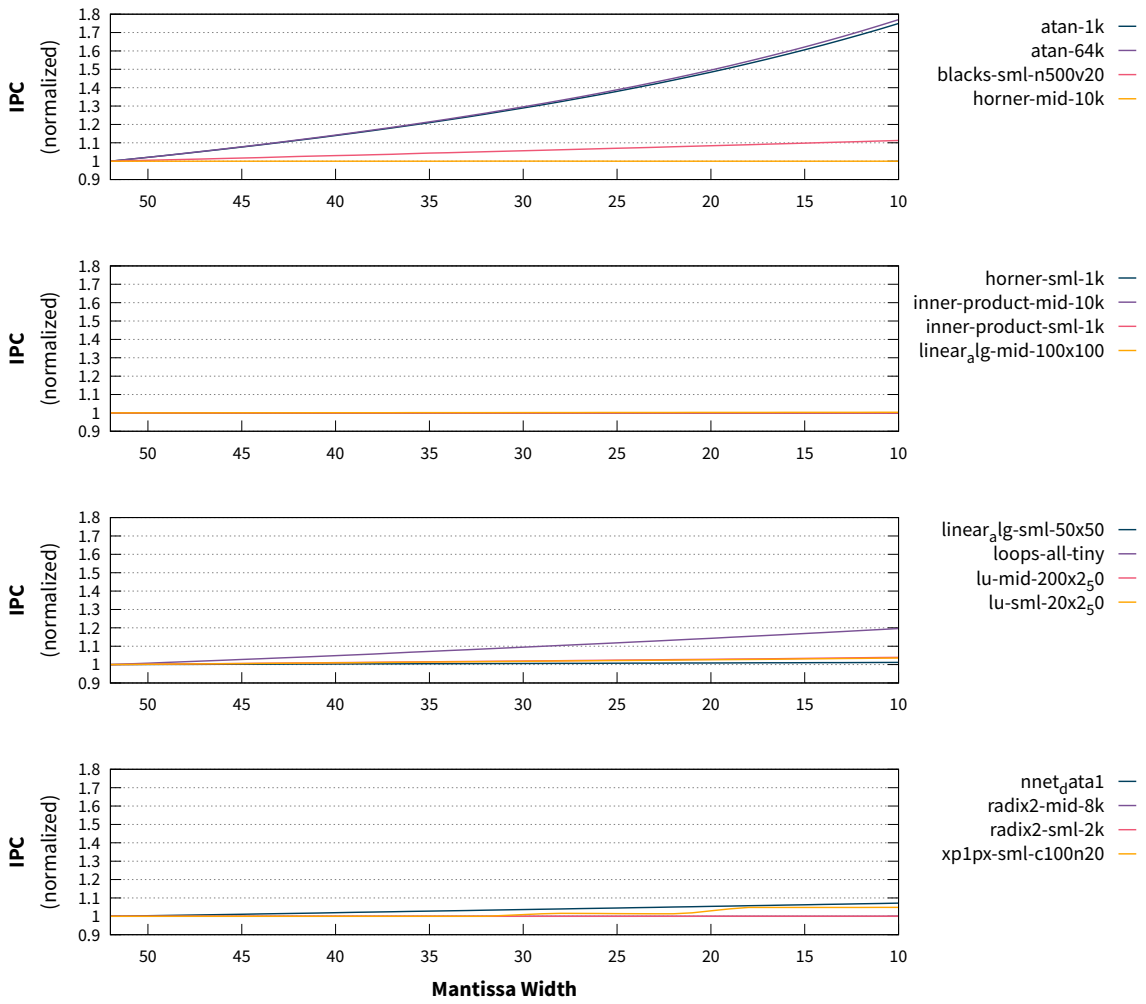

Figure 5.6: IPC (normalized to 52-bit): workloads with meaningful number of floating-point divide/square-root instructions benefit from the reduced latency. The reduced precision changes workload *xp1px-sml-c100n20*'s execution paths, resulting in increased IPC.

Figure 5.6 presents the *full-compliance*'s reduced precision IPC data, normalized to the 52-bit data-paths' result. We omit the graph for the other reduced compliance configurations as the data trends are similar since the designs have the same latency. We see that workloads *atan-1k*, *atan-64k*, *blacks-sml-n500v20* and *loops-all-tiny* have 75%, 75%, 10%, and 17% increased IPC, respectively. As the four workloads contain the most floating-point divide and square-root instructions, their IPCs benefit the most from the reduced latency. Additionally, the lower latency also mitigates other processor bottlenecks, e.g. *atan-64k* sees a decrease of 46% and 78% in the number of *operand-stalls* and *no-id-stalls* respectively.

Furthermore, workload *xp1px-sml-c100n20* sees a 9% increase in IPC despite having virtually zero floating-point divide or square-root instructions. It is likely that the error introduced by reduced precision causes the algorithm to take different execution paths. This is evidenced by the 36% increase in the number of branch instructions issued when mantissa decreases from 52 to 10. The change in execution sequence results in a 14%

increase in the number of integer instructions issued, which biases the IPC higher, since integer instructions have lower latency.

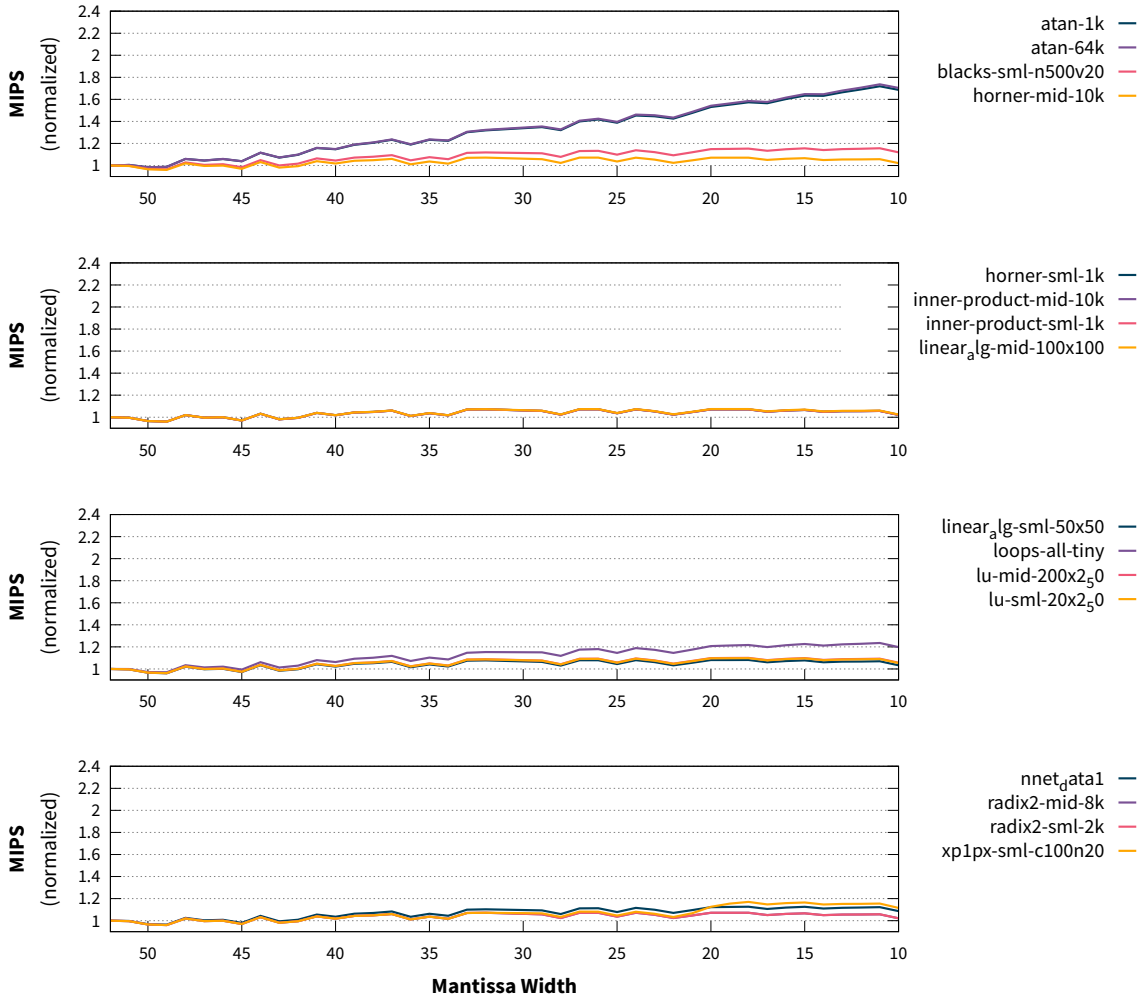

Figure 5.7: MIPS (normalized to 52-bit): MIPS mostly scale with IPC since operating frequency has little fluctuation among reduced precision FPUs.

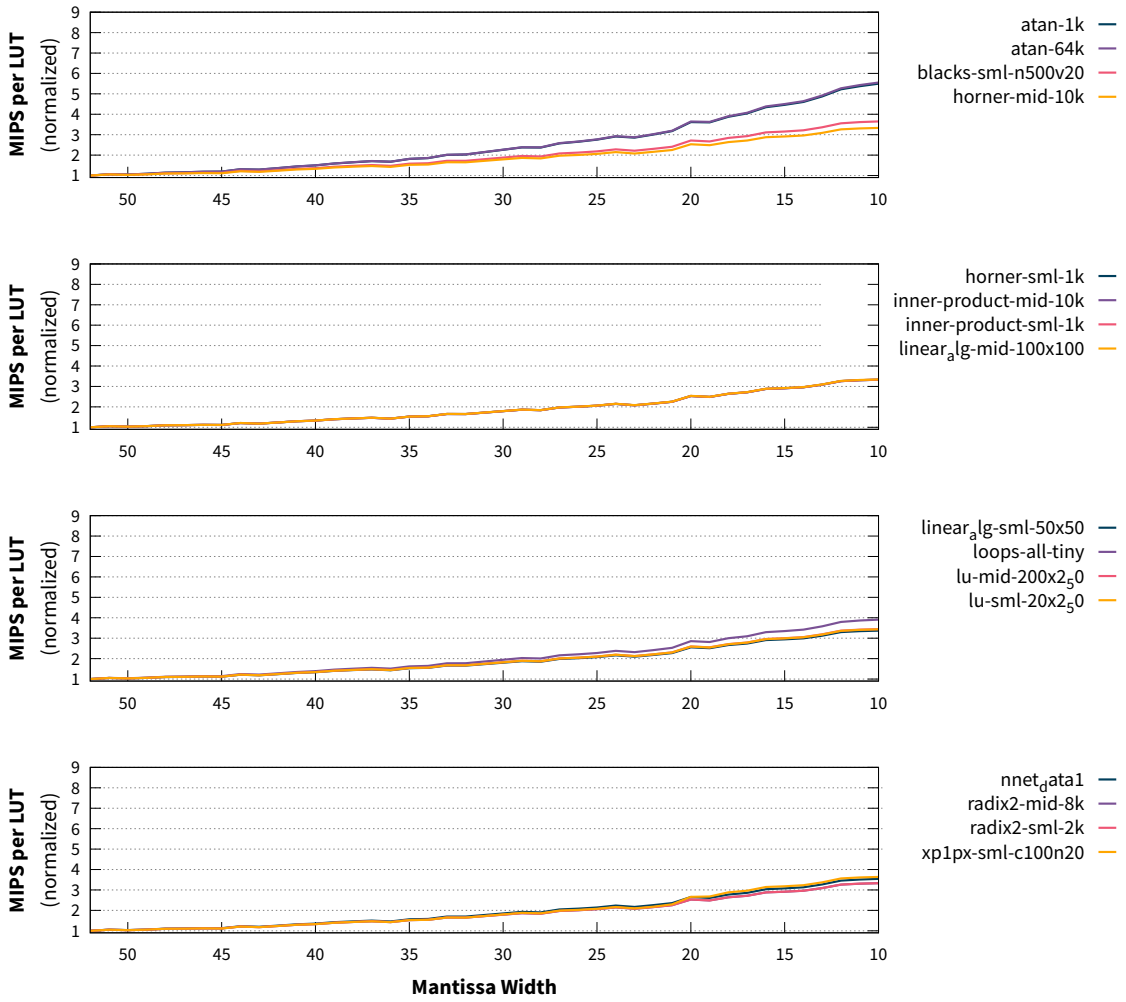

Figure 5.8: MIPS/LUT (normalized to 52-bit).

Full-compliance's reduced precision FPUs have trivial improvements in operating frequency as discussed in Section 5.2.1. As such, the MIPS largely scales with IPC, which is dependent on algorithms and input data, as visualized in Figure 5.7. It can be seen that only workloads *atan-1k*, *atan-64k*, *blacks-sml-n500v20* and *loops-all-tiny* have non-trivial increase in throughput, while others stay mostly flat. Furthermore, as the resource utilization decreases with mantissa width, the performance efficiency improves significantly, as shown in Figure 5.8. Workloads *atan-1k* and *atan-64k* have the most MIPS/LUT improvement of 5.5x, while those workloads whose IPCs do not improve as a result of lower mantissa width see a 3.3x increase in their performance per LUT.

5.2.3 Summary

In our reduced precision work, we evaluated the low-precision designs in terms of operating frequency, resource usage, runtime performance, and accuracy. While we found that our floating-point data-paths' resource utilization scaled linearly with the mantissa width, the clock frequency had significant fluctuation

on both Zedboard and U200. Specifically, the smallest 10-bit-mantissa FPU configuration used 69% less resources compared to the 52-bit configuration, but only achieved a 1.7% and 8% frequency improvement on the respective FPGAs. Additionally, our examination of the timing reports revealed that routing congestion dominated the critical paths of virtually all reduced precision FPUs, and decode-and-issue logic was often in the critical paths. The decode-and-issue logic heavily leveraged LUTRAMs, which had worse timing than FFs. Moreover, the use of LUTRAMs constrained the placer and router further, as not all LUTs can be used as LUTRAMs. Future work could explore different storage mechanisms for the floating-point decode-and-issue logic.

In terms of runtime performance, only workloads with a meaningful number of floating-point divide instructions (3.7%) saw IPC improvements of up to 75%, while others' IPCs stayed mostly flat. This is due to the multi-cycle nature of our division algorithm, whose latency decreased as the mantissa width decreased. The unaffecting IPC scaling meant that MIPS and MIPS/LUT were solely dependent on the resource utilization reduction, and operating frequency improvements — which were erratic and unremarkable on the Zedboard. Future work will explore lower latency floating-point data-paths with low mantissa widths on modern FPGAs in order to extract more IPC. Furthermore, our accuracy analysis showed that only *atan-64k* and *atan-1k* could take advantage of the reduced precision FPUs, as others quickly generated meaningless results as the mantissa width decreased. In the future, we hope to benchmark the reduced precision FPUs with applications that can withstand the reduced mantissa widths, such as machine learning workloads.

5.3 Merged FMA

In this section, we evaluate the impact of merging the FMUL unit's write-back port with that of the FMA unit. We use the *full-compliance* FPU as the baseline configuration. The *merged-fma* and *full-compliance* produce the same floating-point results. Therefore, we focus our analysis on hardware performance, runtime performance and performance efficiency, and omit the discussion on result accuracy.

5.3.1 Clock Frequency and Resource Usage

Table 5.9: Clock Frequency and Resource Utilization of *full-compliance* and *merged-fma* FPUs on Zedboard.

Configurations	Frequency (MHz)	LUTs	FFs	DSPs	BRAMs
full-compliance	105.7	5423	3527	9	0
merged-fma	109.8 (+3.9%)	5169 (-4.7%)	3233 (-8.4%)	9	0

Table 5.10: Clock Frequency and Resource Utilization of *full-compliance* and *merged-fma* FPUs on U200.

Configurations	Frequency (MHz)	LUTs	FFs	DSPs	BRAMs
full-compliance	323.6	5735	3575	9	0
merged-fma	341.2 (+5.4%)	5546 (-3.3%)	3264 (-8.7%)	9	0

Table 5.9 presents the clock frequency and resource utilization of *full-compliance* and *merged-fma* FPU on the Zedboard. Compared to *full-compliance*, *merged-fma* has 4.7% and 8.4% reduction in LUT and FF usage respectively. Examining the two pipelines shown in Figure 4.2, we see that *merged-fma* removes the FIFO used to arbitrate the floating-point adder. Additionally, merging the FMA, FMUl and FADD instructions allows us to remove the dedicated write-back port to FMUL instructions. Similarly, *merged-fma*'s LUT and FF usage decreases by 3.3% and 8.7% on the U200.

In terms of operating frequency, we see a 3.9% increase in *merged-fma* on the Zedboard, and a 5.4% increase on the U200. On Zedboard, the two FPUs have the same critical path in the rounding unit, and both are dominated by routing delay (+80%). Similarly on the U200, *full-compliance* and *merged-fma* have 71% and 72% routing delay in their critical paths, respectively. This small increase in frequency for both platforms can be attributed to the non-trivial decrease in resource utilization which relaxes Vivado's placement and routing constraints.

5.3.2 Benchmarking Result

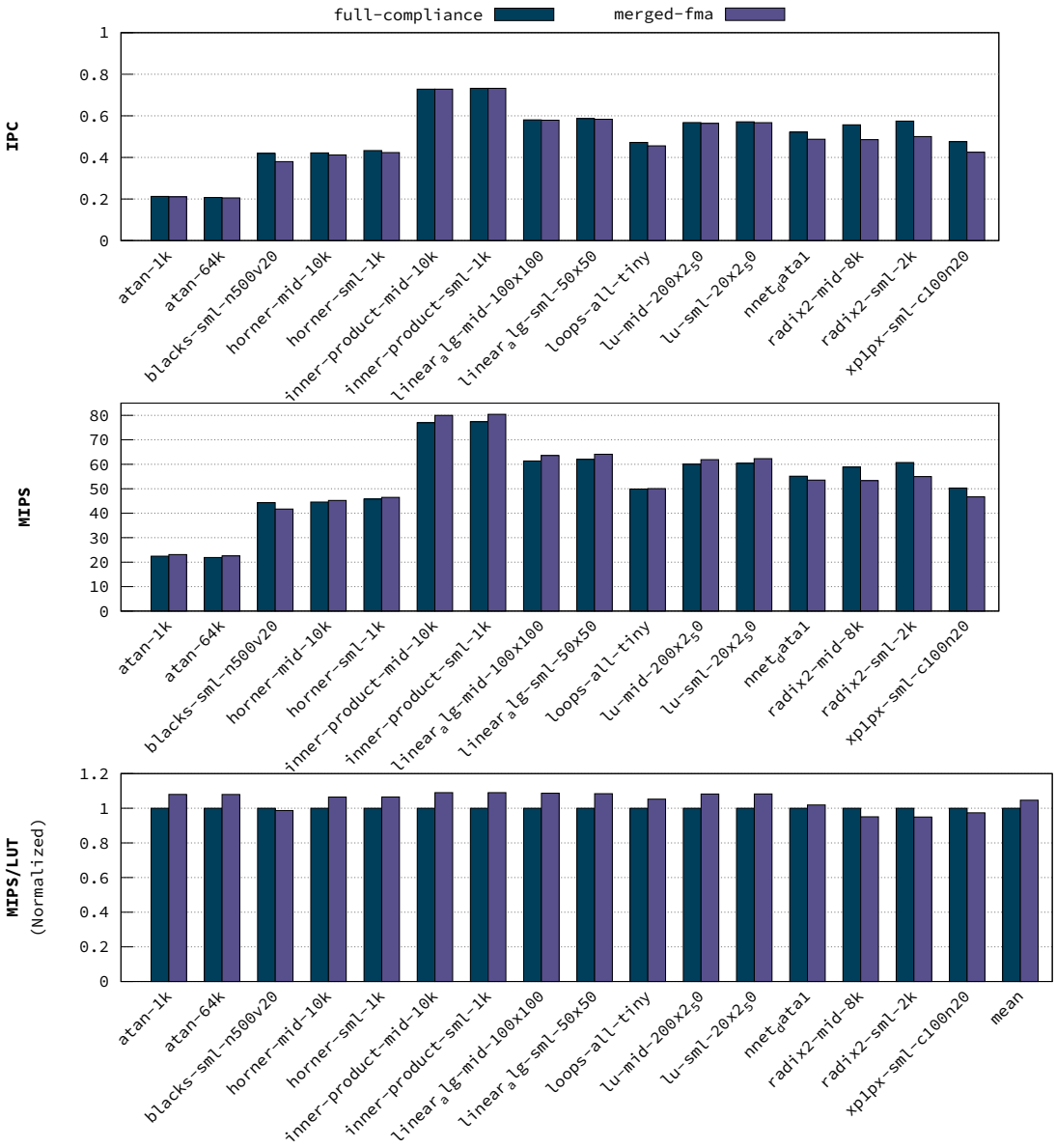

Figure 5.9: IPC, MIPS and MIPS/LUT (normalized to *full-compliance*) comparison between *full-compliance* and *merged-fma*

Table 5.11: Columns are: FPMark workloads' FADD and FMUL instruction count as a percentage of total floating-point instructions; *merged-fma* IPC *no-id-stall*, and *operands-stall* changes vs. *full-compliance*.

Workloads	FADD + FMUL	IPC Change	No ID Stall Change	FP Operand Stall Change
atan-1k	26.1%	-0.9%	1.6%	3.6%
atan-64k	26.2%	-0.9%	1.8%	3.4%
blacks-sml-n500v20	54.4%	-9.5%	33.7%	30.0%
horner-mid-10k	7.0%	-2.3%	12.2%	3.6%
horner-sml-1k	6.9%	-2.3%	12.4%	3.6%
inner-product-mid-10k	0.0%	0.0%	200.0%	0.0%
inner-product-sml-1k	0.0%	0.0%	200.0%	0.0%
linear_alg-mid-100x100	1.0%	-0.3%	1.5%	1.0%
linear_alg-sml-50x50	1.8%	-0.5%	2.9%	1.0%
loops-all-tiny	14.7%	-3.4%	6.0%	13.7%
lu-mid-200x2_50	2.0%	-0.7%	0.5%	6.1%
lu-sml-20x2_50	1.9%	-0.7%	0.5%	6.0%
nnet_data1	26.2%	-6.5%	16.4%	18.0%
radix2-mid-8k	30.5%	-12.8%	1595.9%	14.0%
radix2-sml-2k	30.0%	-12.9%	1920.0%	14.3%
xp1px-sml-c100n20	57.2%	-10.7%	70.9%	43.0%

Figure 5.9 visualizes the IPC, MIPS, and normalized MIPS/LUT data. Additionally, Table 5.11 presents, for each workload:

- the FADD and FMUL instruction mix as a percentage of the total number of floating-point instructions.
- *merged-fma* design's IPC change vs. *full-compliance*.
- *merged-fma* design's number of *no-id-stalls* change vs. *full-compliance*.
- *merged-fma* design's number of *operand-stalls* change vs. *full-compliance*.

In Figure 5.9, we observe that workloads with a relatively low number of FADD and FMUL instructions (<20%) do not see materially reduction in their IPCs. However, workloads *atan-1k* and *atan-64k* are the outliers here, as their IPCs decrease by only 0.9%, despite having 26% of FADD and FMUL instructions. Examining the stall data, we notice that the two workloads have a trivial 3.6% increase in the number of operands stalls with *merged-fma*. This indicates that the FADD and FMUL results produced by *atan-1k* and *atan-64k* are not frequently immediately reused, resulting in a negligible impact on their IPCs.

Moreover, the *merged-fma* design causes a non-negligible (>5%) IPC decrease for workloads *blacks-sml-n500v20*, *nnet_data1*, *radix2-mid-8k*, *radix2-sml-2k*, and *xp1px-sml-c100n20*, which are highlighted in Table 5.11. Specifically, workloads *radix2-mid-8k* and *radix2-sml-2k* have the most reduction in IPC of 12.8% and 12.9% respectively, as a result of the 15x and 19x higher number of *no-id-stalls* respectively, and 14% and 14.3% increased number of *operand-stalls* respectively. It is evident that for these workloads, the *merged-fma* design causes more *operand-stalls* as FADD and FMUL instructions take longer to commit. Moreover, it forces instructions to stay in the execution pipeline longer, which prevents new instructions from entering the execution stage due to the depleted instructoin ID pool.

In terms of throughput, those workloads highlighted in Table 5.11 have reduced MIPS, as a result of their degraded IPC. Workloads *radix2-mid-8k* and *radix2-sml-2k* have the most MIPS reduction of 9.4% and 9.5%, respectively. Similarly, despite the clock frequency and resource usage improvements, the highlighted workloads all have lower MIPS/LUT using *merged-fma* — except *nnet_data1*. Moreover, workload *inner-product-sml-1k* has the most MIPS/LUT improvement of 9.0%. On average, the *merged-fma* design has a 4.7% higher performance per LUT than *full-compliance*.

5.3.3 Summary

In this section, we studied the alternative merged FMA unit microarchitecture, and evaluated its trade-off with that of *full-compliance* in terms of resource, clock frequency, and runtime performance. On average, the *merged-fma* design had 4.7% higher clock frequency and used 4% fewer LUTs on the Zedboard and U200. However, the merged FMA microarchitecture increased FADD and FMUL instruction's latency, which led to more *operands-stalls* and *no-id-stalls*, and lower IPC. In general, workloads with a meaningful number of FADD and FMUL instructions had substantial decrease in IPCs. The outliers here were the *atan-1k* and *atan-64k* workloads, which did not immediately reuse their FADD or FMUL results, leading to a low *operand-stalls* and *no-id-stalls* increases of 3.6% and 1.6%. In terms of throughput and efficiency, since the *merged-fma* configuration did not have significant operating frequency and resource utilization improvements over *full-compliance*, IPC dominated the MIPS and MIPS-per-LUT calculations. The worst performing workloads, *radix2-mid-8k* and *radix2-sml-2k*, saw the most MIPS reduction of 9.5% and MIPS/LUT decrease of 5% when using *merged-fma*. On average, the *merged-fma* caused a 0.3% reduction in MIPS and 9.0% increase in performance-per-LUT. Based on our findings, users may choose the *merged-fma* design if their applications do not contain many FADD or FMUL instructions, or if the results produced by these instructions are not frequently reused.

6 Conclusions and Future Work

Throughout this work, we have explored designing IEEE 754 compliant FPU targeting FPGA-based soft-processors. We reemphasized the speedup opportunity of hardware floating-point implementations, and that modern FPGAs are well-suited for such designs. We demonstrated that many existing floating-point designs do not map well to FPGAs. ASIC-based optimizations are resource intensive and can incur over 88% utilization overhead [40] . The majority of FPGA-based floating-point research focuses on standalone data-paths. These frequency-optimized designs often have deep pipelines, which are not appropriate for soft-processors where instruction-level parallelism is not guaranteed. Existing FPGA-based integrated FPUs have scant compliance support, omitting denormal processing, compliant FMA instructions implementation, and full rounding modes and exception support.

In our exploration of reduced compliance FPUs, we showed that the *no-denormal* configuration provided minimal throughput increase over the baseline, while it saw a substantial 14% performance-efficiency increase as the result of lower LUT requirements. Moreover, we found that removing the compliant FMA implementation provided over 25% reduction in LUTs over the compliant FPU. Furthermore, the smaller *reduced-fma* design allowed us to decrease unit latency, which translated to up to 7.3% increase in IPC, 16% in throughput, and 61% in MIPS/LUT compared to the baseline. Despite the performance and efficiency gain, the lack of compliant FMA instructions caused accuracy degradation to all but two of our benchmark applications. For some corner-case inputs, *reduced-fma*'s computed results became completely meaningless, as we have discussed in Section 2.4.4. Compared to the fully compliant integrated RISC-V FPUs NaxRiscv, our *full-compliance* design had a 7% better clock frequency and 3% fewer LUT usage. On the other hand, our *reduced-fma* design achieved the same frequency as VexRiscv, but required 13% more LUTs. In our analysis, we attributed our resource overhead to our LUTRAM-based register file implementation, whereas VexRiscv used FFs. Additionally, our FPUs's register renaming scheme meant that we had double the register count. Neither NaxRiscv or VexRiscv's resource data included floating-point instruction decode and issue logic, due to the flattened Verilog files generated by SpinalHDL [17]. Although we could not directly analyze the performance difference between our FPUs against NaxRiscv and VexRiscv, Table 5.6 showed that our *reduced-fma* would outperform VexRiscv, as our design had smaller unit latency for all but floating-point divide instructions — which accounted for at most 3.7% of floating-point instructions in our benchmark. NaxRiscv was built upon VexRiscv, therefore its FPU had at best the same latency. Coupled with our higher frequency and lower LUT requirement, the *full-compliance* design likely had better throughput and efficiency.

In our study of reduced precision FPUs, we showed that as precision decreased, resource usage scaled linearly while frequency fluctuated erratically. Furthermore, we found that Taiga's front end dominated the critical paths with more than 80% routing delay on the Zedboard, as a result of its heavy use of LUTRAMs. This indicated that changes to the base processor was needed, as integrating even the smallest FPU resulted in massive routing congestion.

Finally, we explored two FMA unit microarchitectures as FMA, FADD and FMUL instructions dominated floating-point applications. We showed that workloads with many FADD and FMUL instructions saw up to 13% decreased IPC with our *merged-fma* design. However, the reduction in IPC was partially alleviated by *merged-fma* design's 4.7% higher clock frequency and 4.0% lower resource usage. As a result, using *merged-fma* resulted in a 9% increase in performance efficiency.

6.0.1 Future Work

This work has studied the implementation of FPGA-based integrated FPUs and some design trade-offs. As the current Taiga is limited on the dated Zedboard, we would like to continue the trade-off exploration on modern FPGAs. Specifically, the U200 saw significant timing improvement over the Zedboard. The better board will allow us to investigate more aggressive latency-optimized designs. Moreover, we would like to explore the more performant multiplication-based floating-point divider and square-root implementations. The current radix-2 design uses a substantial number of LUTs, whereas a multiplicative algorithm may allow us to share the multiplier with that of the FMUL implementation and enable low-latency FDIV and FSQRT. Furthermore, the current CVA5 [61], formerly known as Taiga [3], retires two instructions per cycle when only one instruction writes back to the register file. Future FPU integration will support multi-retire when the retiring instructions write back to both integer and register files.

Bibliography

[1] A. Waterman and A. Krste, "The risc-v instruction set manual, volume i: User-level isa, document version 20191214-draft." [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/draft-20220301-9ec8c01/riscv-spec.pdf

[2] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2851507

[3] E. Matthews and L. Shannon, "Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[4] P. Stewart and V. Kadirkamanathan, "Dynamic control of permanent magnet synchronous motors for automotive drive applications," in *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*, vol. 3, 1999, pp. 1677–1681 vol.3.

[5] S. Z. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient floating-point arithmetic for software-defined radio architectures," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2011, pp. 122–129.

[6] "Floating point and ieee 754 compliance for nvidia gpus." [Online]. Available: https://docs.nvidia.com/cuda/floating-point/index.html#cuda-and-floating-point

[7] M. 14 and P. Kharya, "Nvidia blogs: Tensorfloat-32 accelerates ai training," May 2020. [Online]. Available: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/

[8] "Understanding exr data compression." [Online]. Available: https://rainboxlab.org/downloads/documents/EXR_Data_Compression.pdf

[9] "The bfloat16 numerical format." [Online]. Available: https://cloud.google.com/tpu/docs/bfloat16

[10] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[11] "Berkeley softfloat." [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[12] "The gnu mpfr library." [Online]. Available: https://www.mpfr.org/

[13] "The gnu mp bignum library." [Online]. Available: https://gmplib.org/

[14] K. S. Hemmert and K. D. Underwood, "Fast, efficient floating-point adders and multipliers for fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, pp. 11:1–11:30, 2010.

[15] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 86–95. [Online]. Available: https://doi-org.proxy.lib.sfu.ca/10.1145/1046192.1046204

[16] M. K. Jaiswal and H. K.-H. So, “Dsp48e efficient floating point multiplier architectures on fpga,” in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, 2017, pp. 1–6.

[17] SpinalHDL, “Spinalhdl.” [Online]. Available: https://github.com/SpinalHDL/SpinalHDL

[18] Cliffordwolf, “cliffordwolf/picorv32: Picorv32 - a size-optimized risc-v cpu.” [Online]. Available: https://github.com/cliffordwolf/picorv32

[19] Riscveval, “riscveval/orca-1: Risc-v by vectorblox.” [Online]. Available: https://github.com/riscveval/orca-1

[20] “Floating-point operator.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/floating_pt.html

[21] “Nios® ii processors for fpgas - intel® fpga.” [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html

[22] L. Bertaccini, M. Perotti, S. Mach, P. D. Schiavone, F. Zaruba, and L. Benini, “Tiny-fpu: Low-cost floating-point support for small risc-v mcu cores,” in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[23] N. Hockert and K. Compton, “Ffpu: Fractured floating point unit for fpga soft processors,” in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 143–150.

[24] “Armhardfloatportvfpcomparison.” [Online]. Available: https://wiki.debian.org/ArmHardFloatPort/VfpComparison

[25] X. Wang and M. Leeser, “Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware,” vol. 3, no. 3, Sep. 2010. [Online]. Available: https://doi-org.proxy.lib.sfu.ca/10.1145/1839480.1839486

[26] “Microblaze soft processor core.” [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html

[27] SpinalHDL, “Spinalhdl/naxriscv.” [Online]. Available: https://github.com/SpinalHDL/NaxRiscv

[28] ——, “Spinalhdl/vexriscv: A fpga friendly 32 bit risc-v cpu implementation.” [Online]. Available: https://github.com/SpinalHDL/VexRiscv

[29] M. Schmookler, M. Putrino, C. Roth, M. Sharma, A. Mather, J. Tyler, H. Van Nguyen, M. Pham, and J. Lent, “A low-power, high-speed implementation of a powerpc/sup tm/ microprocessor vector extension,” in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*, 1999, pp. 12–19.

[30] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, “A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors,” in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–8.

[31] J. Bruguera and T. Lang, “Leading-one prediction scheme for latency improvement in single datapath floating-point adders,” in *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273)*, 1998, pp. 298–305.

[32] ——, “Leading-one prediction with concurrent position correction,” *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1083–1097, 1999.

[33] ——, “Rounding in floating-point addition using a compound adder,” 12 2000.

[34] P. M. Farmwald, “On the design of high performance digital arithmetic units,” Ph.D. dissertation, Stanford, CA, USA, 1981, aAI8201985.

[35] S. Oberman, H. Al-Twaijry, and M. Flynn, “The snap project: design of floating point arithmetic units,” in *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*, 1997, pp. 156–165.

[36] P.-M. Seidel and G. Even, “Delay-optimized implementation of ieee floating-point addition,” *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 97–113, 2004.

[37] T.-J. Kwon, J. Sondeen, and J. Draper, “Floating-point division and square root using a taylor-series expansion algorithm,” in *2007 50th Midwest Symposium on Circuits and Systems*, 2007, pp. 305–308.

[38] J. D. Bruguera, “Low latency floating-point division and square root unit,” *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 274–287, 2020.

[39] ——, “Radix-64 floating-point divider,” in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, pp. 84–91.

[40] A. Malik and S.-b. Ko, “A study on the floating-point adder in fpgas,” in *2006 Canadian Conference on Electrical and Computer Engineering*, 2006, pp. 86–89.

[41] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, “Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 774–787, 2021.

[42] Administrator. [Online]. Available: https://www.gaisler.com/index.php/products/processors/leon3

[43] C. Renard, “Fpga implementation of an ieee standard floating-point unit,” Tech. Rep, Tech. Rep.

[44] B. Fagin and C. Renard, “Field programmable gate arrays and floating point arithmetic,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 3, pp. 365–367, 1994.

[45] J. Allan and W. Luk, “Parameterised floating-point arithmetic on fpgas,” in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*, vol. 2, 2001, pp. 897–900 vol.2.

[46] G. Govindu, R. Scrofano, and V. K. Prasanna, “A library of parameterizable floating-point cores for fpgas and their application to scientific computing,” in *In Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, 2005, pp. 137–148.

[47] “About floating-point ip cores.” [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683750/20-1/about-floating-point-ip-cores.html

[48] F. de Dinechin and B. Pasca, “Designing custom arithmetic data paths with flopoco,” *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[49] Xilinx, “Xilinx/hls: Vitis hls llvm source code and examples.” [Online]. Available: https://github.com/Xilinx/HLS

[50] M. Langhammer and T. VanCourt, “Fpga floating point datapath compiler,” in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 259–262.

[51] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, “Legup: High-level synthesis for fpga-based processor/accelerator systems,” ser. FPGA ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1950413.1950423

[52] Administrator. [Online]. Available: https://www.gaisler.com/index.php/information/ordering/orderip-cores/ordergrfpu

[53] “Technical documents.” [Online]. Available: https://sparc.org/technical-documents/

[54] E. Matthews, Y. Gao, and L. Shannon, “Exploring writeback designs for efficiently leveraging parallel-execution units in fpga-based soft-processors,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 120–128.

[55] “About the eembc fpmark™ floating-point benchmark suite.” [Online]. Available: https://www.eembc.org/fpmark/

[56] E. Matthews, “Modernizing soft-processor architecture for today’s sram-based fpgas,” Aug 2021. [Online]. Available: https://summit.sfu.ca/item/35160

[57] “5. introduction to nios® ii floating point custom instructions.” [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683242/current/introduction-to-floating-point-custom.html

[58] Verilator, “Verilator/verilator: Verilator open-source systemverilog simulator and lint system.” [Online]. Available: https://github.com/verilator/verilator

[59] Riscv-Ovpsim, “Riscv-ovpsim/imperas-riscv-tests.” [Online]. Available: https://github.com/riscv-ovpsim/imperas-riscv-tests

[60] “Axi uart16550.” [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_uart16550.html

[61] Openhwgroup, “Openhwgroup/cva5: The core-v cva5 is an application class 5-stage risc-v cpu specifically targetting fpga implementations.” [Online]. Available: https://github.com/openhwgroup/cva5