NTNU

Norwegian University of
Science and Technology

Low Power Floating-Point Unit for RISC-V

Torbjørn Viem Ness

Master of Science in Electronics

Submission date:	July 2018
Supervisor:	Per Gunnar Kjeldsberg, IES
Co-supervisor:	Vemund Bakken, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronic Systems

Low Power Floating-Point Unit for RISC-V

RISC-V is an open source microprocessor platform that currently gains popularity in many domains. Included among its modules a floating-point unit (FPU) can be found, intended for use in high-end microprocessors.

Nordic Semiconductor has a RISC-V implementation based on an existing public domain platform. The platform consists of both a high-level Python model and a hardware description language (HDL) implementation. The overall goal of this assignment is to investigate how an FPU can be integrated in the platform to achieve an energy efficient solution, followed by a Register Transfer Level (RTL) implementation.

During a project in the fall of 2017, the Python model has been extended to include an FPU, and initial experiments were performed to evaluate energy efficient solutions. In this master thesis project, these evaluations shall be continued and suggestions for improvement in the existing FPU related to energy efficiency shall be presented. A complete RTL implementation shall be performed according to the findings. A mechanism for evaluation of the FPU with the complete platform shall be developed. This will include simulation, verification, and further refinement of the FPU and platform integration.
References:

- RISCV architecture : https://riscv.org/specifications/
- FPU from rocket (existing RISCV CPU) can be reused as starting point https://github.com/ucb-bar/rocket

Supervisor:
Vemund Bakken, Nordic Semiconductor

Professor:
Per Gunnar Kjeldsberg, NTNU

Abstract

When working on a limited energy budget, wireless and battery powered devices that monitor sensors need to do some degree of local computation to save power on transmission. Therefore there is a need for floating-point capabilities in a lower power segment than what has traditionally been the case. In modern technologies leakage power plays an increasingly important role, and as such reducing the area can also have a positive impact on the total energy consumption, with the added benefit of lower manufacturing cost.

This Thesis proposes modifications to the *floating-point unit* from the *PULP platform* that yield area reductions of approximately 13% by reusing the *fused multiply-add* unit for regular add/sub and multiply operations. Due to poor optimization the initial results are worse than expected, with more than twice the energy/op for multiplication and addition compared to the standard FMA-enabled PULP FPU, but with the appropriate power optimizations this could still prove to be a reasonable compromise for area-constrained implementations that still need high throughput and low power. For operations other than add, subtract and multiply, the modified FPU yields up to 7% lower energy/op.

Preface

First I would like to thank my mentors and all the great people in the System Architects Group at Nordic Semiconductor for their assistance, helpful advice and interesting coffee-machine conversations. It has been a truly inspiring experience to work alongside such talented people and learn from the best!

Secondly I would like to thank my girlfriend who is patient and understanding when I have to spend long evenings at the office to make tools compile and simulations run, and helps me relax and unwind when I finally go home for the day.

And last but not least, big thanks to my parents and family who have always supported and encouraged me to follow my dreams and pursue an engineering education. The inspiration from my engineer uncle and godfather was what first made me interested in technology and taking things apart to look inside, I would probably not be where I am today if it wasn't for that!

Initially the FPU implementation was going to be based on the *Berkeley Hardware Floating-Point Units* module from the *Rocket Chip Generator*[12], available on GitHub[19]. However that code is written in relatively advanced CHISEL (*Constructing Hardware In Scala Embedded Language*), making it somewhat difficult to understand and modify without prior experience. Therefore the much more accessible SystemVerilog FPU implementation[35] from the *PULP platform* was selected instead. This also fit easily into the existing tool flow for synthesis and power estimation at Nordic, lowering the entry barrier significantly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In a modern society, monitoring and controlling the environment is becoming increasingly important. The trend seems to be more and more sensors just about everywhere; everything should be measured and reported, and with the advent of the so-called *Internet of Things* this seems to have exploded in popularity - now *everything* should have a sensor.

These sensors should also be independent of wires, so running on batteries or energy harvested from the environment is a given. This creates a huge demand for devices capable of reading and transmitting the sensor readings while using as little power as possible.

Analog signals like audio and other sensor inputs typically vary in intensity, and it is usually desirable to be able to capture and process the data in a way that preserves the most information detail possible.For this purpose floating-point numbers (as described in Section 2.1) are suitable, as they can represent a very wide range of values in a very high precision.

These numbers are somewhat more complex to work with than fixed point, and therefore take longer to compute in software. *Floating-point units* contain dedicated hardware modules for performing operations on such numbers, and will therefore dramatically speed up floating-point computations when used[34].

RISC-V is a relatively new CPU architecture which started as a research project in the Computer Science Division of the *Electrical Engineering and Computer Sciences* Department at UC Berkeley. It began in 2010 as a Master's thesis and was later completed in 2016 as a PhD project[45] by *Andrew Waterman*[1], and its *Instruction Set Architecture* (ISA) is now maintained by the RISC-V Foundation[15].

The architecture is designed to be a modern, open source alternative

[1]With major contributions from Yunsup Lee

to commercial architectures like MIPS and the ones from ARM. This was done in order to have *one* easily accessible and flexible platform that allows developers to focus their efforts on developing good software and tools, while providing educators with an architecture that is suitable for use in teaching[45].

There are existing floating-point units for RISC-V today, but these are mostly limited to the more powerful 64-bit solutions. This project aims to investigate whether it is possible and worthwhile to include hardware floating-point support for smaller 32-bit microcontroller-scale systems, and what can be done to make it more energy-friendly.

This thesis is a continuation of the work done in [30], and investigates various techniques for reducing the area and power overhead of adding hardware floating-point support to a low-power system. The strategy has been to take an existing design as a starting point, identify hotspots and bottlenecks in typical applications, and explore various possible optimizations that may impact performance and power consumption in an embedded environment.

Details of the optimizations explored can be found in Chapter 4. These implementations consist mainly of various techniques discussed in [30], but also include new ideas discovered while exploring the design space.

As discussed in Chapter 3 having a dedicated FPU in microcontrollers that are on the lower end of the power spectrum is not very common. One question this project aims to address is how much energy can be saved by including a dedicated floating-point unit in small, ultra-low-power chips.

Chapter 2 will introduce some of the related background theory used later in the report. In Chapter 3 some previous solutions will be briefly discussed before describing this implementation in Chapter 4. Results are then presented in Chapter 5 followed by a brief discussion and conclusion in Chapters 6 and 7.

1.1 Objectives and limitations

The main target for this implementation is a single issue in-order RISC-V CPU with a shallow pipeline (typically 2-3 stages) and primary focus on low power. A typical use case will be 32-bit microcontrollers in the performance range between the *Zero-riscy*[36] and *RI5CY*[37]. Due to this focus the described optimizations and resulting implementation may not be suited for use in high-performance processors. Some of the outlined optimizations also assume that all resources will be available at any time, limiting the viability for out of order implementations.

1.2 Main contributions

The main contributions in this thesis are related to bugfixes and optimizations on the PULP FPU, in addition to several contributions to the PULP and other projects:

- Implemented power optimizations to the FMA of the PULP FPU, including multiply-stage pipeline bypass.
- Estimated power consumption and area impact of reusing the FMA pipeline for multiplication and addition.
- Improved tool compatibility for the PULP FPU, now works with QuestaSim and Synopsys tools. My contribution was merged into the central PULP project repository[32].
- Fixed a bug in the PULP FPU and got my code merged into the central project repository[31] - see Appendix B.1

Chapter 2

Background theory

This chapter will give an introduction to the topics discussed later, and explain some of the terms used. First about floating-point number representations and arithmetics, before going into basics about energy consumption, various architectural optimizations and verification topics.

2.1 Floating-point arithmetics

Floating-point is the name of a formulaic representation for real numbers that can be used to achieve a good trade-off between dynamic range and precision. Working with floating-point numbers is somewhat different than working with integers, and the following sections will try to explain why.

An alternative to using floating-point when working with fractional numbers is to define a set precision and use integers. This is called *fixed-point arithmetics*[42], and although much simpler and faster to work with it is a lot more limited than floating-point in several ways. Much like one uses *cents* to describe fractions of a larger monetary value, one can define a scope where integer numbers represent smaller fractions - for example 0.01. One example of a limitation is dynamic range - while a (signed) 32-bit integer can represent numbers between -2147483648 and $+2147483647$, the binary 32-bit *IEEE-754* floating point format can represent the entire range between $\pm 340282346638528859811704183484516925440$ (or $3.4028235 * 10^{38}$), with a decreasing degree of precision as the magnitude increases[1].

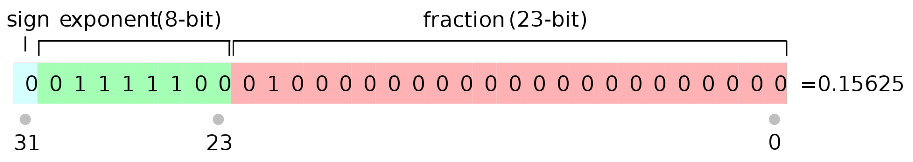

Figure 2.1: The binary representation of a IEEE 754 single precision floating-point number

By Codekaizen (Own work) [CC BY 3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons

	Single	Double
Sign bits	1	1
Exponent bits	8	11
Fraction bits[3]	23	52
Exp. bias	127	1023
Total	32	64

Table 2.1: Parameters for single and double precision formats

2.1.1 Number representation

According to the *IEEE-754* standard[1], floating-point numbers in compliant implementations have a binary representation as seen in Figure 2.1, which consists of a *sign bit*, an *exponent* and a *fraction*. The value of a floating-point number can be calculated as described in Equation (2.1).

$$B_{float} = (-1)^{sign} * radix^{(exponent-bias)} * fraction \tag{2.1}$$

The radix is normally 2 (*binary*) or 10 (*decimal*), with the value of *exponent* determining the magnitude of the number and the *fraction* representing the numerical value. For decimal (base-10) floating-point representation, 3 decimal digits are usually packed into 10 bits using the so-called *densely-packed decimal* format (for more information, see [1, Section 3.5]). However this format is not commonly used[2], so in this project the focus will be on *binary* floating-point arithmetic with single (32-bit) and double (64-bit) precision. For an overview of these two binary formats and their differences, see Table 2.1.

Not included in the fraction is an implicit leading bit, which is used in

[1]Highest precision around zero: $1.401298*10^{-45}$ and lowest near infinity: $2.028241*10^{31}$

[2]Although the RISC-V ISA intends to supports it in the future, with the *L* standard extension[46, Ch. 11]

[3]The effective precision is increased by one bit due to the implicit leading bit

order to get one more bit of precision while keeping the data width identical. This extra bit is set to '1' for every exponent except the minimum value (in which case the number is *subnormal* and the leading bit is '0'). This is valid because floating-point numbers are normalized (aligned such that it always starts with a '1', as long as the exponent is greater than zero) before being stored[1, 3.4].

The *bias* is used for representing a wide dynamic range without needing to use 2's complement to represent negative exponent values - the bias for a given precision can be calculated using the formula $2^{k-1} - 1$, where k is the number of bits in the exponent.

The example shown in Figure 2.1 is computed like this, using Equation 2.1 with exponent $01111100_2 = 124_{10}$ and bias from Table 2.1:

$$B = (-1)^0 * 2^{124-127} * (1 + 2^{-2}) = 0.15625 \tag{2.2}$$

The result of computations will in many cases end up being a value that can not be exactly represented in the binary format[4] due to only being able to combine powers of two. When this happens the number must be *rounded* to a value that can be represented in the given format and precision before it can be stored. How rounding should be applied is given as a configuration parameter to each operation; for details about the different modes available, see [1, 4.3].

In addition to representing regular positive and negative numbers, the standard has special representations for $\pm\infty$ and *NaN* (*Not a Number*, used to represent the result of various illegal operations, such as $\frac{0}{0}$ and $\frac{\infty}{\infty}$). Such results can be especially useful when troubleshooting, as the fraction field of a NaN may contain more information about what led to the error.

When it comes to NaNs there are two different types - *signaling* and *quiet*, each with specific rules for handling in different operations[1, 6.2]. *Quiet* NaNs (*qNaN*) are generally used when computations yield invalid results (such as the square root of a negative number), and it should be the result whenever an *invalid operation* exception is signaled (for more details about invalid operations see [1, 7.2]). Operations on qNaNs should preserve as much diagnostic information as possible, in order to help developers find out what went wrong. *Signaling* NaNs (*sNaN*) are reserved operands that when used trigger the *invalid operation* exception for every general computational instruction. This is typically used to represent uninitialized variables, for example when debugging, to see whether a variable has been set before use.

[4]For example 1.1 will be rounded to 1.10000002384185791015625 in single precision

2.1.2 Arithmetic operations

The most important operations that a floating-point unit accelerates are multiplication and addition. Therefore it also becomes an important target for optimization, both for area and performance.

For many applications there are both multiplications and additions to be performed that involve the same operands. In terms of implementation there are currently two main approaches commonly used - *Fused Multiply-and-Add (FMA)* and *Cascaded Multiply-and-Add (CMA)*. While the cascaded approach is simple and consists of performing multiplication and addition in sequence, the fused approach yields higher throughput but at the cost of slightly higher latency. Each approach has its own set of drawbacks and advantages, so the choice of which one to use really depends on the estimated workload. Therefore it has become increasingly common to choose the FMA approach, as long as there is room for it in a design. However the cascaded approach is more common for simpler and smaller designs, which can be beneficial when area is at a premium and performance isn't paramount. In the case of RISC-V the FMA operations are a mandatory part of the floating-point extensions[46, Ext. 'F', 'D'], and for implementations without hardware FMA these instructions would have to be implemented in *microcode* (as described in Section 2.3.3) or a similar way.

2.1.3 Fused Multiply-Add

This method allows great throughput by combining two operations that would otherwise be separate and put them into the same pipeline, producing one result every clock cycle instead of having to store the final multiplication result before using it in an addition. The computation performed can be as described by Equation (2.3).

$$Result = A * B + C \tag{2.3}$$

Although a bit more complicated than a setup of simple multipliers and adders, these fused units can perform all operations involving addition and multiplication by simply setting the correct operand to 1 or 0 - addition is done by setting A or B to 1, and multiplication by setting C to 0.

Aside from achieving higher throughput in suited applications, combining the two operations will usually also reduce the rounding error. This is a result of the multiplication product being fed directly into the adder without rounding the intermediate result first.

One obvious cost of the FMA compared to using the simpler CMA architecture is increased complexity and area; the data path needs to be significantly wider for FMA - 72 bits for single-precision compared to 48 for a CMA implementation[16]. For double-precision it is even more - 161 bits according to Scwarz[41]. An FMA needs to align the multiplication product before performing the addition, and this could in the worst case scenario require shifting it all the way from one end to the other. A more thorough introduction to FMAs including some hints on how to optimize the design can be found in a paper by Eric M. Schwarz[41].

2.2 Energy consumption basics

In order to fully understand low power optimizations, some background knowledge is required about what consumes power in a digital CMOS circuit.

2.2.1 Energy and power

Power is the rate of energy consumption *at a given time*, while *energy* is the power consumption accumulated over a time period[39] (see Equation 2.4).

$$E = \int p(t)dt \tag{2.4}$$

If the power is constant or one uses the average power, it can be simplified to Equation 2.5:

$$E = P_{avg} * t \tag{2.5}$$

Due to this the answer may not always be obvious when it comes to deciding between very low power or a faster circuit with a higher power draw. If the processor can complete the calculations quickly before returning to a low-power sleep state (commonly referred to as *Race to Halt*), then the ever-improving sleep modes and low power states of modern microprocessors enable significant system power savings[10].

2.2.2 Static and dynamic power

In CMOS circuits, power consumption is mainly divided in two categories: *static* and *dynamic* power[7].

Static power is the consumption that is constant regardless of what the circuit is doing, which mainly consists of *leakage currents* from supply to ground via the transistors. This phenomenon is due to the fact that a transistor is not ideal and in the real world it will not act entirely like the on/off switch that it is often simplified to. Instead there will always be some

Tech.	V_{DD}	Static [mW]	Dyn. [mW]	Static [%]	Dyn. [%]	Tot. [mW]
45nm	1.1	0.2250	1.8230	≈ 11	≈ 89	2.0480
15nm	0.8	0.1134	0.5206	≈ 18	≈ 82	0.6340
Diff.	0.727x	0.5040x	0.2856x	(relative difference)		0.3096

Table 2.2: Example of effects on static and dynamic power due to technology scaling[40]

current running in the channel (*subthreshold leakage*) or through the gate insulation layer[7].

Some decades ago it would not be very prominent compared to the active power consumption, but this effect scales adversely with the ever-decreasing feature sizes and threshold voltages that comes along with new manufacturing techniques[7]. In modern technologies such as 22, 14 and even 10 nm, leakage can in many cases becomes the major contributing factor to device power consumption. See table 2.2 for an example of this scaling. Due to this it becomes increasingly important to optimize technology libraries and designs for low leakage, for example by utilizing techniques such as *power gating* and selecting low-leakage transistors[5].

A comparison between 45 and 15nm was done in [40], where the same single-precision FPU was synthesized in the two different process technologies (keeping architecture and synthesis parameters constant) using *Synopsys Design Compiler*[21]. In that study it was found that the cell leakage power roughly halved[6], while the total power was reduced by almost a factor of 4. This means that the ratio of static/dynamic power nearly doubled, as shown in Table 2.2. If one takes into account the fact that this was simulated as running without any special power saving features, it becomes clear that switching off power to unused parts of the circuit (*power gating*) is a technique that will become increasingly important; as sub-threshold leakage can already constitute about 20% of the power consumption when actively computing, one can only imagine how much of the total system power will be leakage once low-power sleep modes and clock gating is included.

In terms of mitigating this effect there are a few options we can utilize (e.g. adjusting transistor threshold voltages or power gating parts of the design when not in use), but the most effective way is probably to keep the area to a minimum. If we have fewer transistors leaking power the total leakage will naturally also be lower, and then there will be the added benefit

[5]One way to lower the leakage in a transistor is to increase the threshold voltage, others may include using different gate insulation materials[33]

[6]Although area was reduced by approximately a factor of 3 due to scaling

of lower manufacturing cost.

Dynamic power is the power consumed by a transistor when it is switching. This power consumption mainly comes from charging the capacitive loads connected to the transistor (i.e. gates of other transistors, parasitic capacitances due to wires etc.). There is also some dynamic power consumed in the brief period of time when both the high and low side transistors are (partially) open, which leads to a direct path from supply to ground[7]. However due to the fast switching times in modern CMOS, this consumption is relatively small compared to the capacitance charging so it is often neglected.

Based on the above, one can express the total power consumption of a circuit as:

$$P_{tot} = P_{static} + P_{dynamic} = (V_{supp} * I_{leakage}) + (\alpha * C * V_{supp}^2 * f) \qquad (2.6)$$

where V_{supp} is the supply voltage, $I_{leakage}$ is the total leakage current, α is the activity factor[7], C is the driven capacitance and f is the operating frequency of the circuit[9].

From this we can see that the most effective ways of keeping the power consumption down by design are:

- Lower the operating voltage (if possible)
- Eliminate unnecessary toggling (especially for larger nets)
- Keep wires short (i.e. group related functions close together)
- Enable power gating of entire modules when not in use

2.3 Architectural options

This section gives a brief overview of some implementation options available to a system architect and how they can affect performance and energy consumption.

2.3.1 Pipelining

When optimizing for energy consumption, a pipelined architecture can be beneficial as it can help reduce the size because of less complex and duplicated logic required to meet a certain performance target.This in turn

[7]A number between 0 and 1 describing on average how large a portion of the transistors or logic gates that toggle from '0' to '1' on every clock period

reduces the leakage power[16, p.916], which is one of the major sources of energy consumption in today's modern CMOS processes and low power architectures, as discussed in Section 2.2.2 It is however important to find a balance, as more stages incur extra latency and require more power and area for the pipeline registers.

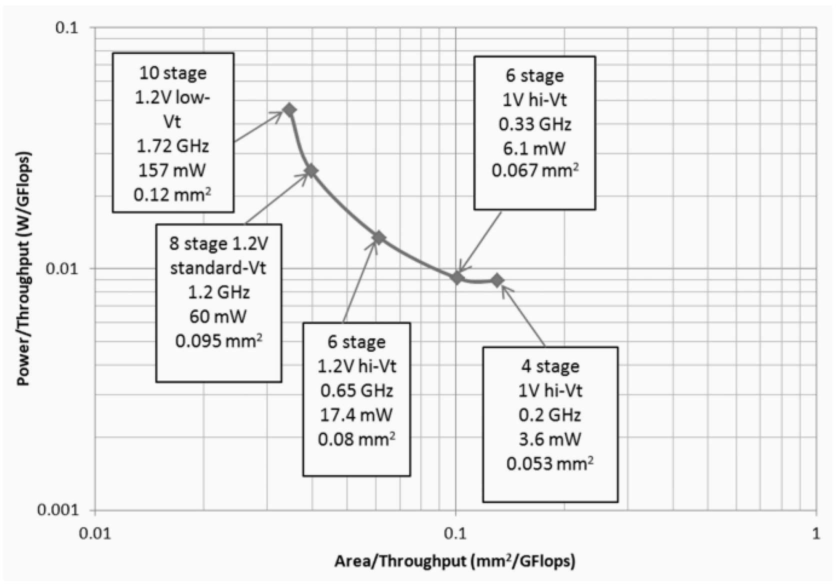

Figure 2.2: Power- versus area efficiency a 90 nm single-precision FMA. Using supply voltage, threshold voltage, area and pipeline depth as parameters for optimization[16]

In Figure 2.2 this relationship is shown for a particular fused multiply-add (FMA) implementation, and in terms of performance per watt the sweet spot appears to be a 6-stage pipeline. It is important to note that this result may not hold true for all other implementations, as the target process will be different and other factors like leakage come into play.

2.3.2 Timing

One important parameter that affects how fast a design can run is the *critical path* - the longest path a signal must travel between two endpoints before the deadline (typically the next clock edge). The maximum speed is determined by how much *slack* the system has:

$$t_{slack} = t_{required} - t_{arrived} \tag{2.7}$$

If the *slack* is *positive* that means all signals will arrive on time, and if necessary the clock frequency can be improved. However if the slack is *negative* the circuit has a timing issue and will not function properly at the given clock frequency. In that case the designer must either find ways to reduce the critical path, or accept that the circuit becomes slower than desired.

There are several ways to reduce the critical path and improve max speed. One solution that most tools can apply automatically is *register retiming*; that involves moving internal registers along the data path in either direction to shorten the critical path. This is described more detailed in this paper[28].

Another commonly used technique is *pipelining* (as described in Section 2.3.1), but that requires some manual work from the designer..

2.3.3 Microcoding

Microcode is a type of very low-level software that can be used to implement sequences of operations as if they were one instruction. It is typically stored in a read-only memory space that is not accessible from programs[48], often hardwired in a ROM *(Read-Only Memory)*. By using microcode, instructions that are not directly supported in hardware can still be executed as if they were - transparently to the user program For example, fused mutliply-add instructions can be coded as a regular multiplication followed by an addition and executed without the hardware overhead of a FMA unit.

In typical desktop processors it is usually possible to load new microcode upon boot, but these changes are not persistent and will be lost at the next power cycle[26].

2.4 Synthesis and verification

Synthesis is the process of converting a functional system description into a lower-level implementation. It is performed by specialized tools; both commercial ones from vendors like *Synopsys*, *Cadence* and *Mentor* but some free alternatives also exist - most notably *Yosys*[49] and *ABC*[11].

2.4.1 RTL simulation

Simulation on the *Register Transfer Level (RTL)* is useful because it models the system behavior with a fairly high degree of accuracy, and can be used to spot bugs in functionality before running more time-consuming *netlist* simulations or sending the design to manufacturing.With suitable tools like *Spyglass Power* from Synopsys[21] or *Mentor PowerPro*[8] it is even possible to get an estimate on how much power a design will consume just from running simulations on the RTL code.

2.4.2 High-level model

It is common practice when designing digital systems and circuits to first create a simplified model that implements the desired behavior of the system. When deciding on architectural issues, such a high-level model is very useful as it allows the designer to explore several different options without having to wait for hours or even days for a cycle-accurate simulation to complete. It is also (typically) much easier to modify than a full RTL implementation.

The model is especially useful for functional validation, as one can run the same test on both the model and the RTL implementation and compare output. Debugging high-level code is generally much easier than finding functional errors in a hardware description, and if the results are identical then the functionality should also be the same.

There are several options for implementing such a model - for instance high-level languages like *Python* and *Matlab* are commonly used as they are both relatively simple to use for getting a quick data flow/algorithmic model of the problem, and they complete relatively quickly on modern hardware when compared to cycle-accurate RTL or netlist simulations. Some drawbacks of making models in those languages are that they place the designer relatively far away from how it will actually work on the silicon level, and "bare-metal" operations on raw bytes and bits become unnecessarily complicated and difficult.

In contrast, something like SystemC[22] (based on C++) would be a bit closer to reality, as C/C++ allows direct access to the memory via pointers and gives the programmer full freedom to manipulate it without having to go through many layers of wrapping and abstraction. As an alternative to SystemC there is also SpecC[18] (based on C), but it is not in very active development at the moment; the latest release is currently 10 years old.

2.4.3 Scoreboard

The term *scoreboard* comes from the board commonly used in sports stadiums to display information like the time left and number of points each team has. In a verification context this term has been adopted to describe a system that keeps track of the tests that have been run on a design, and how many of them passed or failed. The scoreboard also an important part of the popular *Universal Verification Methodology (UVM)*[2], where it collects various stimuli and compares output from the *Design Under Test (DUT)* with the output of a high-level model.

Chapter 3

Previous work

A floating-point unit is not by any means something new - already in 1980 Intel released their *8087 math coprocessor*, but when it comes to low power embedded systems, hardware floating-point support is usually reserved for models in the higher performance end of the spectrum like tablet and smartphone SoCs. For example *SiFive* (the company driving the commercial part of RISC-V) only include a FPU in their top-tier 64-bit quad-core SoC solution (the *U54-MC* application processor), while the single-core and 32-bit alternatives have no such support. One notable example of a small, low-cost FPU is the *MIPS R4200*[50], which achieved exceptionally low hardware overhead by sharing the ALU logic for addition and multiplication with the integer part of the CPU. *ARM* on the other hand only offers FPUs as an option for their *Cortex-M33* processor and above. The goal of this project is to find an efficient way to implement a power-optimized FPU that can be used to accelerate tasks such as signal processing and precise sensor readings for low-power IoT devices.

This section will briefly discuss some previous work done in the field.

3.1 Rocket Chip Generator

The *Rocket Chip Generator*[13] is an open-sourced System-on-Chip (SoC) design generator for RISC-V. It is created by the same team that developed the ISA specification[46], and allows the designer to specify a high-level configuration for a SoC using CHISEL[1]. All the code involved in the original Rocket Chip Generator from UC Berkeley is freely available on the project's GitHub page[12].

[1]Constructing Hardware In Scala Embedded Language[4]

A library of various *tiles* containing logic modules and interconnect is also published there, and custom logic modules may be written in the same specification language to add new functionality. Among the available tiles is a hardware floating-point unit[19] library, with both single and double precision support.

After describing the desired module configuration the SoC design specification can be compiled into various target languages, including C for high-speed simulation and synthesizable RTL for implementation in FPGA[2] or ASIC[3] designs. As of April 2016, cores built using Rocket Chip had been taped out eleven times, and yielded functional silicon prototypes fully capable of running Linux.[13].

3.2 NanoRV32

The *NanoRV32*[5] is a small 32-bit implementation of the *RISC-V* architecture, and a research project by Ronan Barzic. Among other things it includes a Python high-level model which can be used for functional verification of HDL implementations by comparing register contents.

The Python simulator emulates data flow in the system and stores the computed results after each instruction in a Python array representing the CPU *register file*. Code to be executed is first converted from RISC-V compiled binary code into a text-based hexadecimal byte representation. This hex bytecode is then interpreted by an instruction decoder and dispatched to the correct Python function for computation.

Currently this implementation supports the *RV32IM* ISA, which includes basic integer arithmetics and a hardware multiply and divide unit. It also supports interrupts and a compressed instruction format - *RVC*, which allows static and dynamic code size reductions[4] by representing the most commonly used instructions with a shortened 16-bit format instead of the full 32-bit word.

3.3 Project thesis

The *project report* that was the precursor to this thesis work[30] started out with the *NanoRV32* (Section 3.2) and extended its high-level model to

[2]Field-Programmable Gate Array

[3]Application Specific Integrated Circuit

[4]On average, the use of RVC is estimated to replace somewhere around 50-60 % of the instructions in a program, leading to a 25-30 % reduction in code size[46, Ch.12]

support single-precision IEEE-754 floating-point arithmetics. This model can be used to verify the functionality of a HDL implementation with floating-point support similarly to how the NanoRV32 model verifies implementations without floating-point support.

3.4 Berkeley Hardfloat

The *Berkeley Hardfloat* is a library of components to use for constructing floating-point units. It is the one used by the Rocket Chip Generator (Section 3.1) and it is written in CHISEL. The components from this library have been proven to work in silicon[27] and as such it could be used as a reference implementation. All source code is freely available on GitHub[19].

3.5 PULP platform

The *Parallel Ultra Low Power* (PULP) platform is a collection of RISC-V based processor implementations, with the goal of running several ultra low power cores in a cluster to achieve high energy efficiency and performance for parallelized jobs.

In the process several individual chips have been made, like the *PULPino*[17] (taped out in February 2016) and the *PULPissimo* (among others taped out as part of a chip in 2017[38]).

The PULP platform also includes a collection of modules for a low power FPU, all written in SystemVerilog and used several times as part of the *RI5CY core*[37] which is the current processor in PULP and PULPino. According to the *ETH Zürich wiki*[14] these FPU modules have been taped out in at least 6 different chips at the time of writing. These floating-point modules are also open source and available on GitHub[35], and will be used as a basis for the RTL implementation in this project. A block diagram of the RI5CY core can be seen in Figure 3.1.

In addition to supporting the standard RISC-V ISA the PULP project has also made some custom extensions in order to accelerate certain tasks; e.g. a hardware looping instruction and various DSP instructions, including support for non-standard floating-point precisions like 8-bit for increased performance when lower accuracy is required - see [43] and [29].

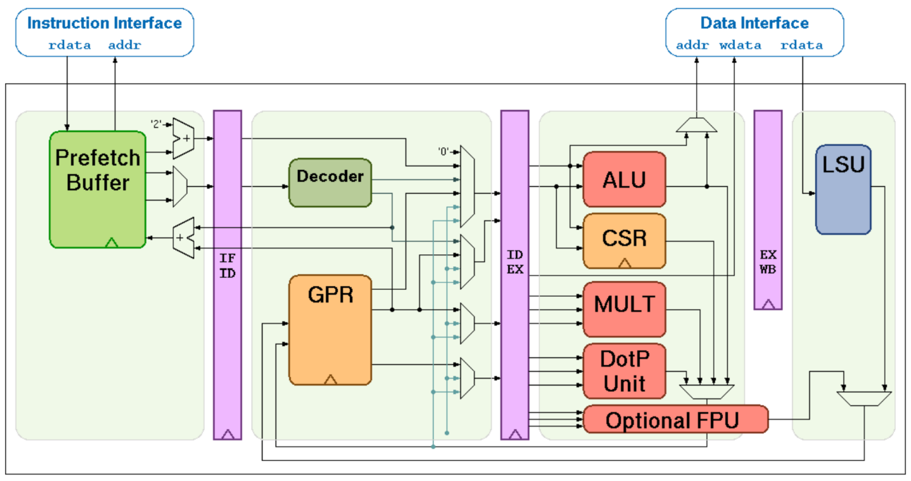

Figure 3.1: Block diagram of the RI5CY core

3.6 Reduced latency for addition with a FMA

In a paper by Bruguera and Lang[6] an implementation was proposed that allows bypassing the multiplication stage in a fused multiply-add unit (FMA) in order to reduce the latency to be comparable with a separate adder when using the FMA only for add operations. This allows for area reductions by eliminating the need for separate add and multiply units alongside the FMA, while avoiding the latency penalty of the FMA unit compared to a simpler adder module. In the case of a 3-stage FMA pipeline, skipping the multiplication stage could result in only a 2 cycle latency, and for a 5-stage it could be reduced to 3 cycles by following this approach.

These improvements are, according to the paper, achieved by moving the alignment shift operation to happen after the multiplication instead of in parallel, and combining addition with rounding into a single operation. Additionally in order to avoid increasing the critical path, normalization is performed before addition and in a separate datapath from the alignment operation.

The idea of reusing the FMA for regular add and multiply operations will also be investigated in this project.

Chapter 4

Implementation

In order to save time and reuse previous work done by others this implementation is mostly based on an FPU from the *PULP platform* (Section 3.5), along with custom low-power optimizations which are detailed in the following sections.

This chapter starts with an overview of the modules necessary to make a floating-point unit in Section 4.1, before describing the overall verification strategy in Section 4.2. Then a description of the FPU register file and main module interface follows in Sections 4.3 and 4.4, before introducing the power optimization strategy followed in Section 4.5 and detailing the chosen *Fused Multiply-Adder* (FMA) implementation in Section 4.6.

4.1 What to implement

As the standard[46] is relatively modular in its design, a compliant implementation is not strictly required to implement all functions in hardware; specifically the toolchain[47] allows configuring parameters as to enable/disable things like hardware division and sqrt.

Therefore, here is a list of things that need to be implemented for a functional RISC-V FPU (in order of importance):

1. Load/store unit, this is necessary for moving data between the FPU and memory without having to go through the main CPU registers, wasting both bandwidth and time. It can be implemented by extending the existing integer load/store unit to allow using the FPU registers as source and destination. However it is an implementation-specific detail that does not affect the optimizations outlined here, and for that reason has been placed outside of the scope for this thesis.

2. FP computational pipeline, used for performing multiplication and addition. This can either be implemented as separate multiplier and adder units, an integrated *FMA pipeline* or both. However it must be capable of performing *FMADD/FMSUB/FNMADD/FNMSUB* operations, either natively or as microcoded operation sequences, as those operations are a mandatory part of the ISA specification[46].

3. Comparison unit, performs *less-than (FLT), less-or-equal (FLE) and equal (FEQ)* operations, could possibly be combined with a *FMIN/FMAX* unit with a mode selection signal

4. Sign manipulation unit, for the *FSGN.** instructions

5. Floating-point number classifier, for determining which class of numbers the input belongs to[1]

6. *FSQRT unit*, performs division and square root operations[2]

4.2 Verification plan

The verification strategy is to do a bottom-up approach; first validate the smallest components before moving up to subsystems and then simulating the entire system. This is a common industry practice, as it is much easier to validate and debug a full system when knowing that the individual parts are working as intended.

For the actual verification there are several options, but the two most likely candidates in this case are to use *SystemVerilog* and *Coroutine Co-simulation Test Bench* (CoCoTB)[44] One advantage of the latter alternative is that writing testbenches is just a matter of creating a Python script and giving it the HDL files, while SystemVerilog is a more industry-tested language with broad support and integration with formal verification tools like OneSpin 360. However when it comes to larger and more complex circuits, the cost of formal verification in terms of time and computing resources grows exponentially and becomes impractical to use.

For this project test benches for the various modules have been implemented using CoCoTB, and once they have been found to work as intended the entire FPU is tested as a module with all available instructions.

[1]Normal positive/negative number, subnormal positive/negative number (Section 2.1.1), positive/negative ∞ or *NaN*

[2]This is the lowest priority because this operation takes a long time to perform even with acceleration, and single-precision *FSQRT* instructions are not even emitted by the *GNU toolchain*[47] with default settings[30, 5.2.1]

One part of the functional verification test bench is a high-level model (as described in Section 2.4.2) written in Python (see Appendix A.2). This is a simplified version of the model created in [30] which returns computation results that can be used to verify that a module computes the correct result. Results computed by the high-level model are then compared with the results from simulation and statistics like number of tests passed and failed are displayed in a *scoreboard* (Section 2.4.3) once the testing is complete, along with state information about the cases that failed.

For ease of reuse across various test cases and alternative implementations, the test bench code (Appendix A.3, A.5 and A.4) is written in a modular fashion; one file contains the simplified FPU model that performs the calculations (Appendix A.2), another contains common functions for clock and reset generation, logging and data conversion (Appendix A.1) while another set of files (Appendix A.6) contain the stimuli for different test cases that should be run. In that way, bugs only have to be fixed once and it is easy to run various tests on many different configurations.

4.3 FPU register file

The register file for the FPU (Appendix A.13) is a fairly standard implementation with 1 write port and 3 read ports. This was done because it is not targeting a high-performance out-of-order architecture, but rather a low-power in-order model that should be as simple as possible.Therefore the register file does not need any more ports than what's required to serve one operation at a time; the most variable-intensive instruction supported by the RISC-V FPU is *FMADD*[3], which require 3 input operands and results in one output.

In order to avoid duplicating logic the FPU registers can be added to the existing register file of the main CPU, with one important exception; only the FPU-specific registers are connected to the third read port as it is only used by the FMA.

4.4 FPU interface

The FPU interface is implemented as a set of control signals along with a data bus between the FPU, CPU and memory.Having only one 32-bit data bus to the FPU was selected because in this low-power implementation there

[3]Floating-point Multiply-and-ADD

should be only one data movement transaction taking place at any given time:

- FPU read from memory
- FPU write to memory
- FPU to CPU register transfer
- CPU to FPU register transfer

Because of this it's not strictly necessary with separate data buses for all different transaction types, and we reduce the number of data lines required by 75%; with a 32-bit wide architecture this translates into 32 lines instead of 128, and if the bus lines are long then the capacitance they incur can be a significant driver of active power consumption.

The control signals will mainly come from the existing instruction decoder in the NanoRV32[5] CPU, while some signals (like the *rounding mode*) will be extracted from the instruction word or the FPU CSR[4] (as determined by the instruction).

4.5 Power optimizations

The power optimization base strategy is to

1. Avoid unnecessary toggling (active power)
2. Minimize the consumed area (static power)

and therefore the optimization will largely consist of methods to approach those two targets.

One specific strategy that has been investigated is to implement a *Fused Multiply-Adder* with the option to bypass any one of the two stages using control signals. Something similar has already been done and demonstrated in [6] where the multiplication stage could be bypassed to perform only additions, yielding approximately 40% reduced delay compared to a similar multiply-add implementation without this bypass option. In a practical pipelined approach, this translates into skipping one stage and thereby reducing the latency.

The FMA approach in itself is mainly for increasing performance, but when using sleep modes and a Race to Halt strategy the increased

[4]Control and Status Register

performance can also translate into total energy savings.Bypassing a stage in the FMA pipeline will effectively make the latency similar to that of separate multiply and add units while maintaining the improved throughput and precision of the FMA in workloads that utilize it.

To avoid unnecessary toggling, inputs to all combinational circuits could also be latched with an enable signal for that specific circuit; this would for example prevent toggling the logic for computing the product of two numbers when all one wanted to do was find the largest one.

4.6 FMA implementation

The FMA implementation presented here is based on the one from *PULP*[35], with some modifications to include optimizations like the multiplier-bypass for reduced latency. Based on the work done by Bruguera and Lang[6] (see Section 3.6), the FMA from PULP was modified to allow bypassing the multiply stage when only an addition is required. See Figure 4.2, and code in Appenices A.8 and A.9. This reduces the latency from 3 to 2 cycles when only addition is performed, and makes it feasible to have only the FMA and save area compared to having both regular adders and multipliers alongside the FMA.

These modifications required some changes to the existing FMA architecture, and can be seen in Appendix A.8.

4.6.1 PULP modifications

In order to support the FMA multiplier-bypass optimization for reduced latency, some modifications had to be made. First the addend, which is operand B in an add/sub operation, needs to be redirected into input C of the FMA[5]. Then it must be aligned with the leftmost bit such that it becomes equal to the result of a multiplication by 1, before being routed into the next stage of the pipeline for adding.See Figure 4.4 for the modifications compared to the original (Figure 4.3).

In order to investigate the preliminary multiplier output and verify that the bypass was implemented correctly, a small test bench for the Booth encoding and Wallace tree was created (see Appendix A.14). Then this test bench was driven with random data on one input and a bit pattern corresponding to that of a IEEE-754 1.0 on the other to verify that these modules indeed were the ones performing multiplication. This helped make

[5]The operation performed is A * B + C

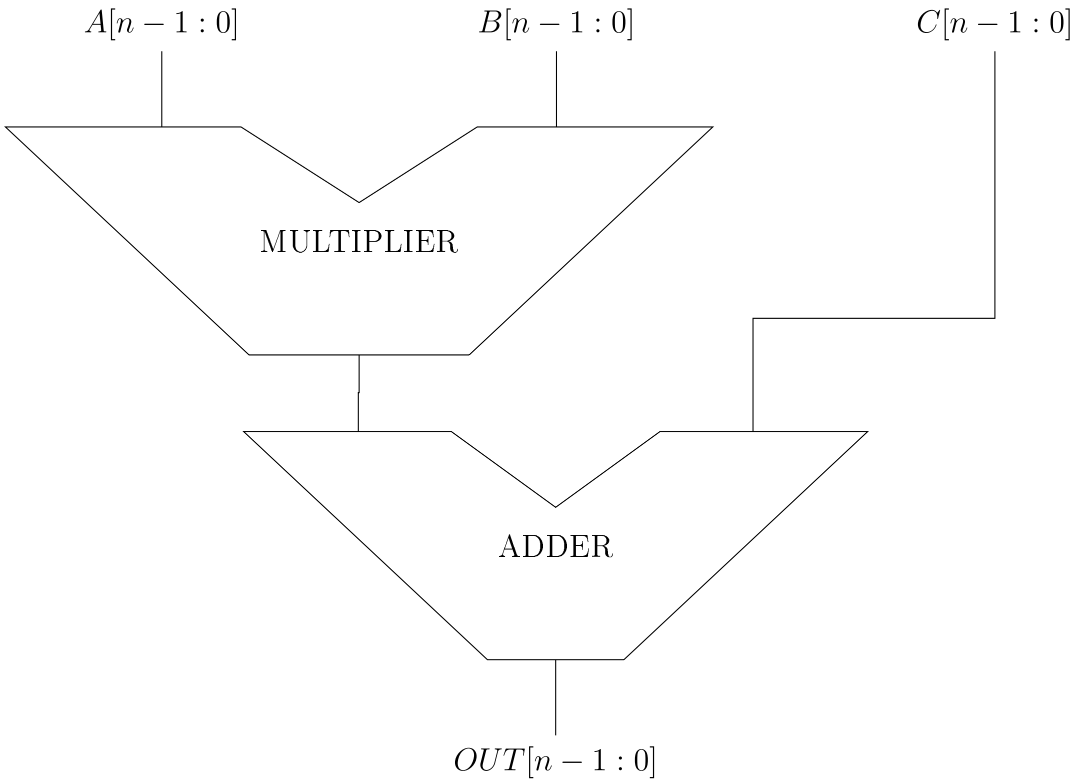

Figure 4.1: The principle of an FMA

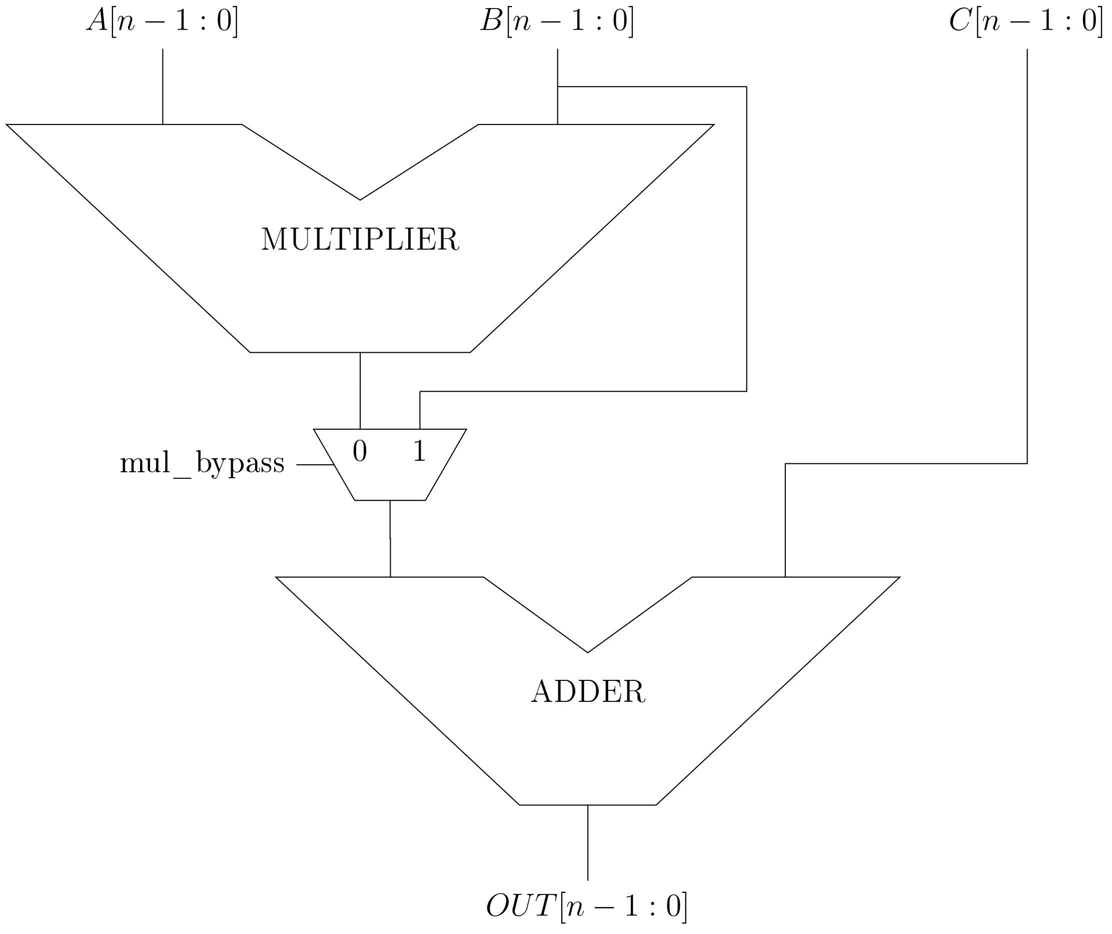

Figure 4.2: FMA modifications for allowing multiplier bypass

sure that the addend was directed to the correct part of the pipeline and that the preprocessing was right.

The existing FMA pipeline

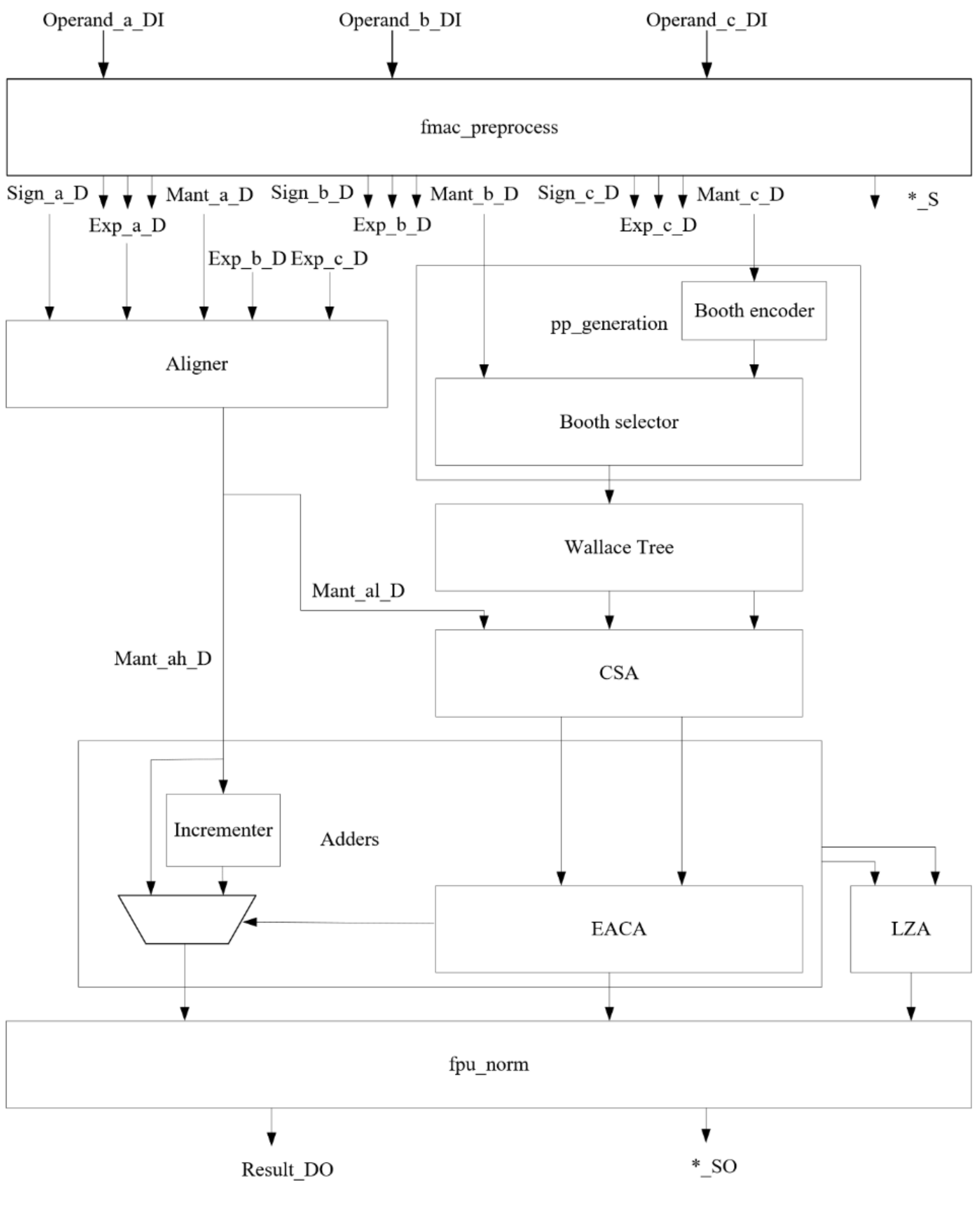

Figure 4.3: Simplified block diagram of the PULP FMAC pipeline, taken from [35]

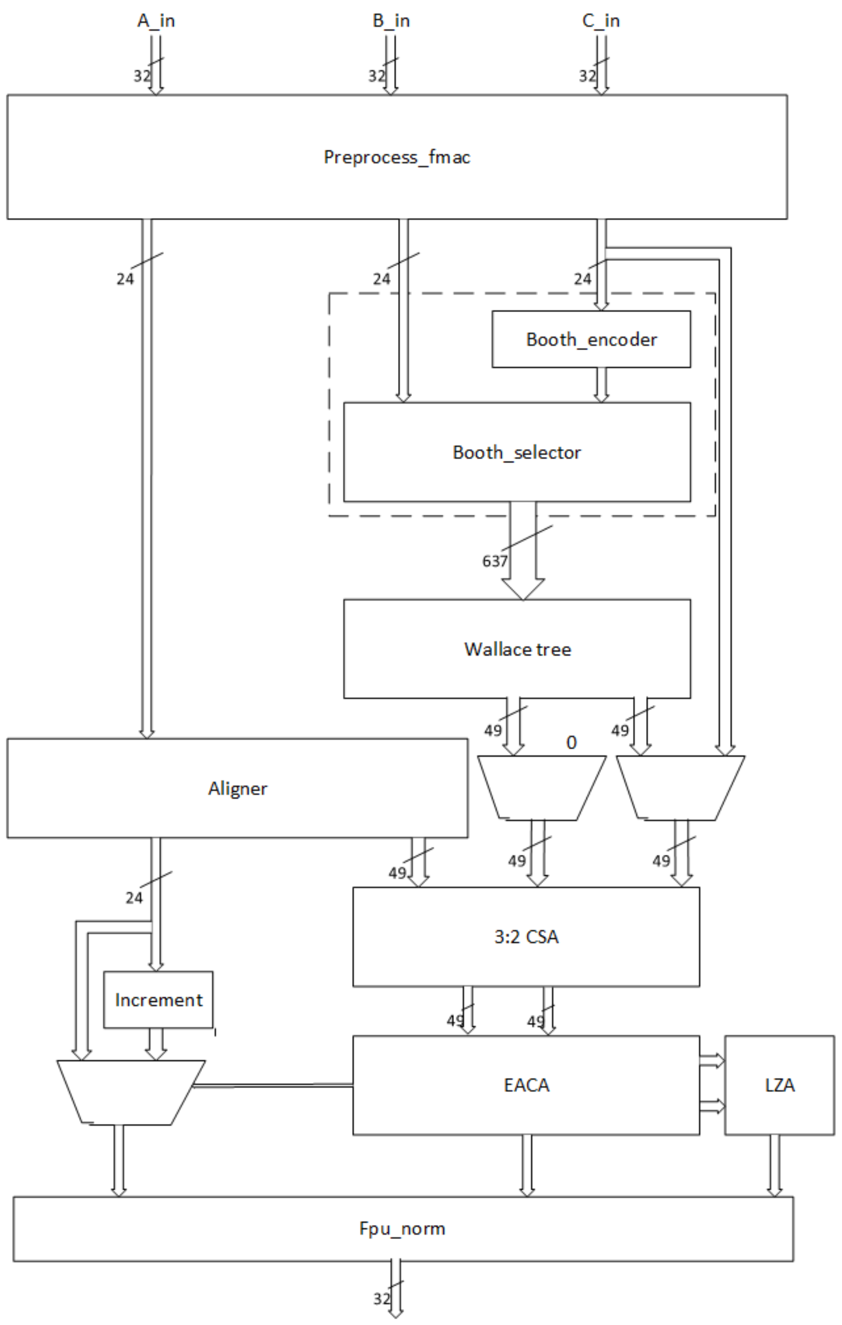

Figure 4.4: Simplified block diagram of the modified PULP FMAC pipeline, see Appendix A.8

4.7 Test and measurement

As a base comparison between the original FPU and the one presented here, each available operation was executed and measured in terms of power consumption and time. This then yielded results that could be extrapolated into approximations of real-world application benchmarks, as the FPU was not yet integrated with a CPU capable of running user code. However when looking at the sequence of instructions executed, using the full CPU Python model implemented in [30], it is possible to make a fairly good estimate which combined with the average power and energy per operation is good enough as a basis for comparison.

4.7.1 Stimuli generation

For each operation, a loop of 1000 iterations with random input data was executed in order to minimize the effect of unrelated operations like reset and initialization on the results. To keep consistency across designs, the random number generator was initialized with the same seed for all runs. Average power was then estimated using *Spyglass Power* from Synopsys[21], and a rough estimate of energy per operation was calculated by multiplying average power with the time required for one iteration.

Chapter 5

Results

5.1 Area comparison

To get a rough estimate of how much area a FPU would consume on chip, the Rocket Chip Generator[12] (see Section 3.1) was used to generate a full Verilog implementation of the default configuration[1]. This implementation was then synthesized in a 55nm-process from TSMC using Synopsys Design Compiler 2016[21], and a breakdown of the results can be seen in Tables 5.1 and 5.2. The target voltage used for synthesis is $1V$ and the frequency is *100MHz* unless specified otherwise.

5.1.1 Rocket chip

Here are the synthesized area results from various configurations of the Rocket Chip Generator.

Module	Area [GE][2]	Area [%]
Integer core	43729	2.0
FPU core	**73582**	**3.4**
Caches (I&D)	1972361	92.4
Other	44750	2.2

Table 5.1: Area consumption of the default Rocket SoC configuration

[1]64-bit single-core SoC with a double-precision hardware floating-point unit
[2]*Gate Equivalents* - the area of a 2-input NAND[23]
[3]Relative to the total FPU area only

Module	Area [GE][2]	Area [%][3]
MulAdd	31410	42.69
DivSqrt	5178	7.04
Other[4]	36994	50.27

Table 5.2: Area consumption breakdown for the double-precision Rocket FPU

For comparison a smaller 32-bit version with a single-precision FPU was also synthesized, the results can be seen in Tables 5.3 and 5.4.

Module	Area [GE][2]	Area [%]
Integer core	24632	2.0
FPU core	**26568**	**2.2**
Caches (I&D)	1156981	95.2
Other	7774	0.6

Table 5.3: Area consumption of the small 32-bit SoC configuration

Module	Area [GE][2]	Area [%][3]
MulAdd	7823	29.45
DivSqrt	1783	6.71
Other[4]	16962	63.84

Table 5.4: Area consumption breakdown for the single-precision Rocket FPU

5.1.2 PULP FPU

These are the area results of the synthesized FPU from the PULP platform (see Section 3.5)

5.1.3 PULP area vs. target frequency

Shown here is how the different PULP designs scale in terms of area as the target frequency increases. Figure 5.1 shows how the FPU area scales without enabling *register retiming*. The maximum frequencies are relatively similar, at $110MHz$ for the stock PULP FPU and $109MHz$ for the customized version.

[4]Conversion-, comparison-, decoding- and clock gating logic
[5]Conversion, normalization, exception detection and clock gating logic

Module	Area [GE][2]	Area [%][3]
MulAdd	15843	56.6
DivSqrt	4988	17.8
Adder	943	3.4
Multiplier	3012	10.8
Other[5]	3198	11.4
Total	27984	

Table 5.5: Area consumption breakdown for the single-precision PULP FPU

Module	Area [GE][2]	Area [%][3]
MulAdd	16723	67.8
DivSqrt	4984	20.2
Other[5]	2964	12.0
Total	24671	

Table 5.6: Area consumption breakdown for the customized single-precision PULP FPU

Figure 5.2 shows the results with retiming enabled, and here the maximum frequencies have increased to $140MHz$ for the standard PULP FPU and 119 for the customized version.

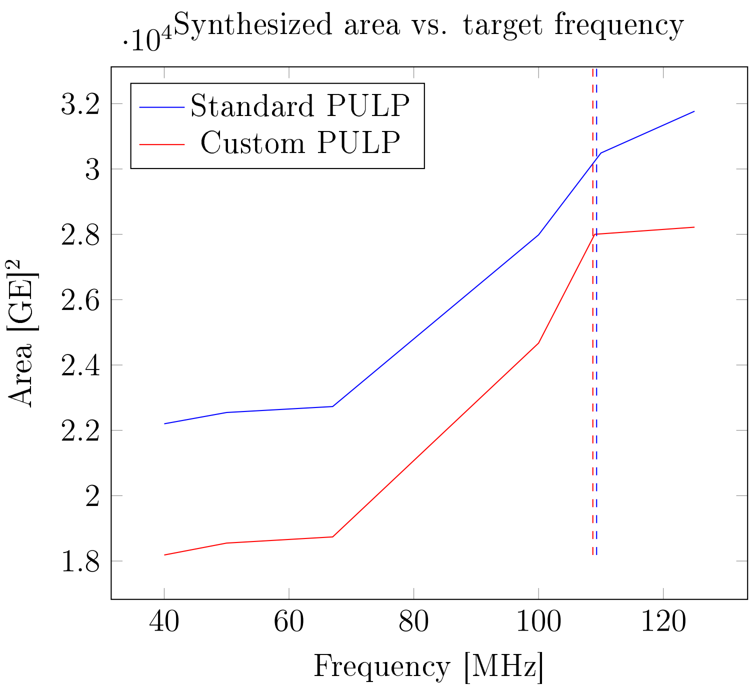

Figure 5.1: Synthesized area vs. frequency plot, no retiming

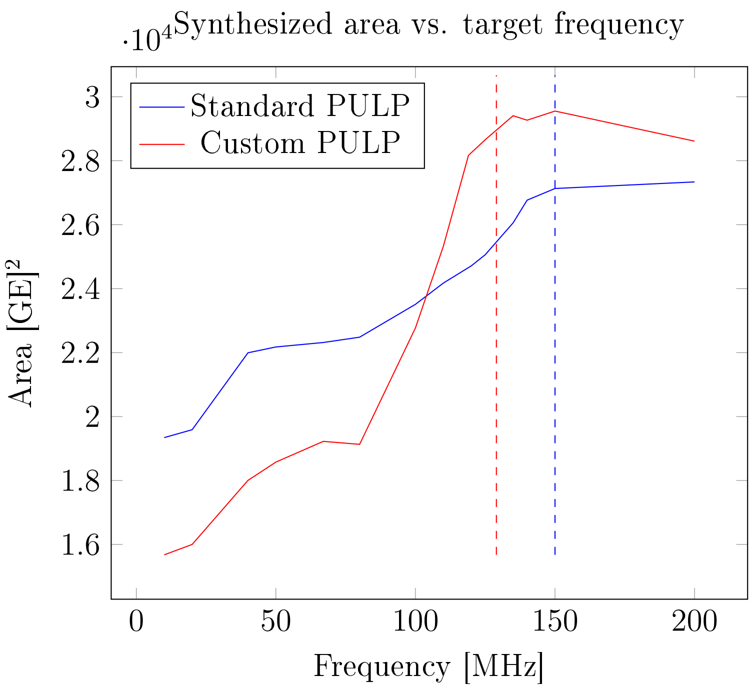

Figure 5.2: Synthesized area vs. frequency plot, retiming enabled

5.2 Timing

When synthesizing for target freq. 100MHz, the difference in slack between standard and modified PULP was approximately 5ps, so the modifications affected the critical path somewhat. However, 5ps is less than 0.05% of the clock period so it should not make any practical difference in this frequency range.

5.3 Power

This Section contains power estimations based on RTL simulation using the *Spyglass Power* suite from Synopsys[21]. It compares the power consumption of the different implementations using activity data from various benchmarks - both single instructions and more varied workloads. Here as in the previous sections, the results are taken from running the at a target frequency of 100MHz unless specified otherwise. Note that these results do not include production calibration data, and as such may not necessarily be absolutely correct. However they do give a relatively accurate comparison.

5.3.1 Single instructions

In order to have a baseline and be able to give rough estimates on the energy consumption of different tasks, each FPU configuration has been measured (using *Synopsys Spyglass*) while performing all supported instructions. The results can be seen in Tables 5.7, 5.8 and 5.9

[6]From the input is applied until the result is ready at the output
[7]Includes a mix of all 4 FMAC ops: *FMADD, FMSUB, FNMADD, FNMSUB*

Operation	Avg. power[μW]	Energy/op[pJ]	Cycles[6]
Add/sub	497.195	4.972	1
Mul	834.259	8.343	1
Div	545.621	43.650	8
Sqrt	534.034	42.723	8
FMAC[7]	822.807	16.456	2
F2I	387.524	3.875	1
I2F	424.180	4.242	1
Idle/Noop	333.976	3.340	1

Table 5.7: Power and energy consumed per FPU op for default PULP config

Operation	Avg. power[μW]	Energy/op[pJ]	Cycles[6]
Add/sub	718.445	14.369	2
Mul	782.559	15.651	2
Div	507.574	40.606	8
Sqrt	500.093	40.007	8
FMAC[7]	825.959	16.519	2
F2I	384.179	3.842	1
I2F	417.666	4.177	1
Idle/Noop	312.249	3.122	1

Table 5.8: Power and energy consumed per FPU op for customized PULP config

Operation	Avg. power[μW]	Energy/op[pJ]	Cycles[6]
Add/sub	157.754	3.155	2
Mul	524.081	10.482	2
Div	N/A	N/A	N/A
Sqrt	N/A	N/A	N/A
FMAC[7]	N/A	N/A	N/A
F2I	98.337	1.967	2
I2F	118.331	2.367	2
Idle/Noop	84.416	0.844	1

Table 5.9: Power and energy consumed per FPU op for small (no FMA) PULP config

Operation	Avg. power[μW]	Energy/op[pJ]	Cycles[6]
Add/sub	157.754	1.578	1
Mul	524.081	5.241	1
Div	N/A	N/A	N/A
Sqrt	N/A	N/A	N/A
FMAC[7]	N/A	N/A	N/A
F2I	98.337	0.983	1
I2F	118.331	1.183	1
Idle/Noop	84.416	0.844	1

Table 5.10: Power and energy consumed per FPU op for improved small (no FMA) PULP config

Note that the small FPU configuration, *riscv_fpu* from the PULP FPU GitHub repository[35] uses two cycles for each operation because it is configured to do so, therefore its performance and energy/op figures can be improved by using the same ready-signal as in the other PULP FPU configurations. See Table 5.10 for improved results.

5.3.2 OPUS codec

Using the power estimated for each instruction in the previous Section, a rough estimate of the energy benefits on the OPUS benchmark is given below for each FPU configuration. Figure 5.3 shows the breakdown in number of instructions required to complete the benchmark, and Figure 5.4 show how it would have been if the compiler was better at converting operation sequences to FMA operations where possible.

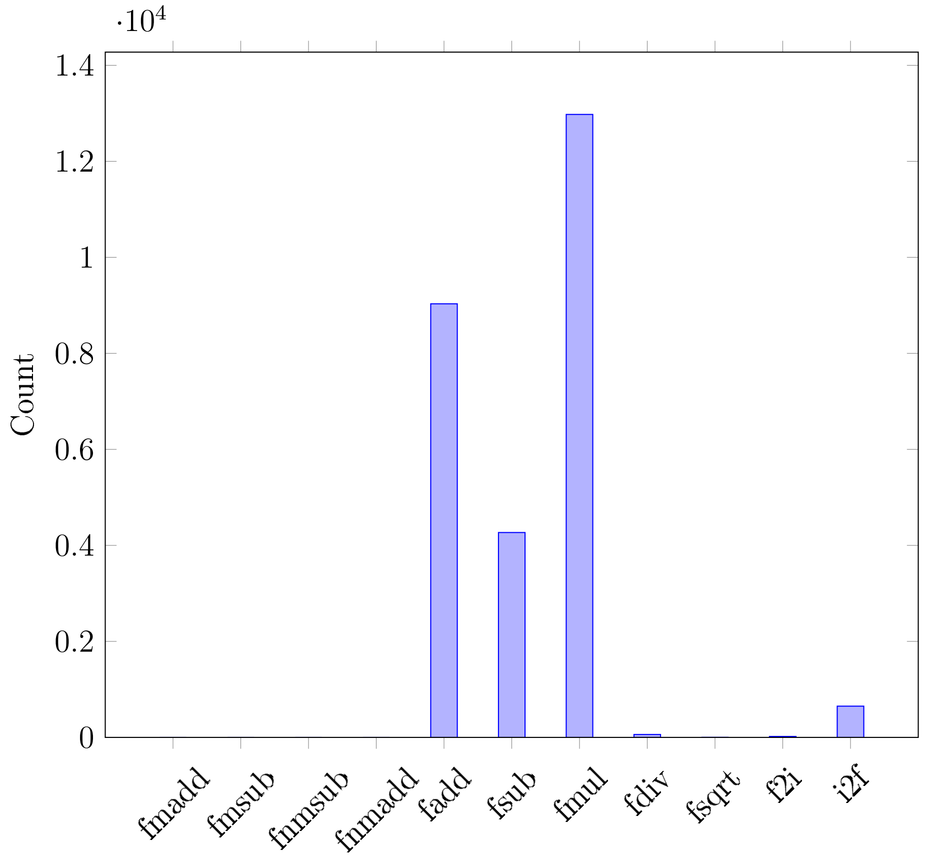

Figure 5.3: Distribution of FPU instructions in OPUS benchmark run

Operation	Standard[pJ]	Custom[pJ]
Add/sub	66126.94	191106.37
Mul	108261.79	203105.36
Div	2662.63	2476.96
Sqrt	0.00	0.00
FMAC[7]	0.00	0.00
F2I	77.50	76.84
I2F	2761.41	2719.01
Idle/Noop	1534596.34	1434762.30
Total	*1714486.61*	*1834246.83*

Table 5.11: Power and energy consumed for OPUS benchmark, grouped by FPU op

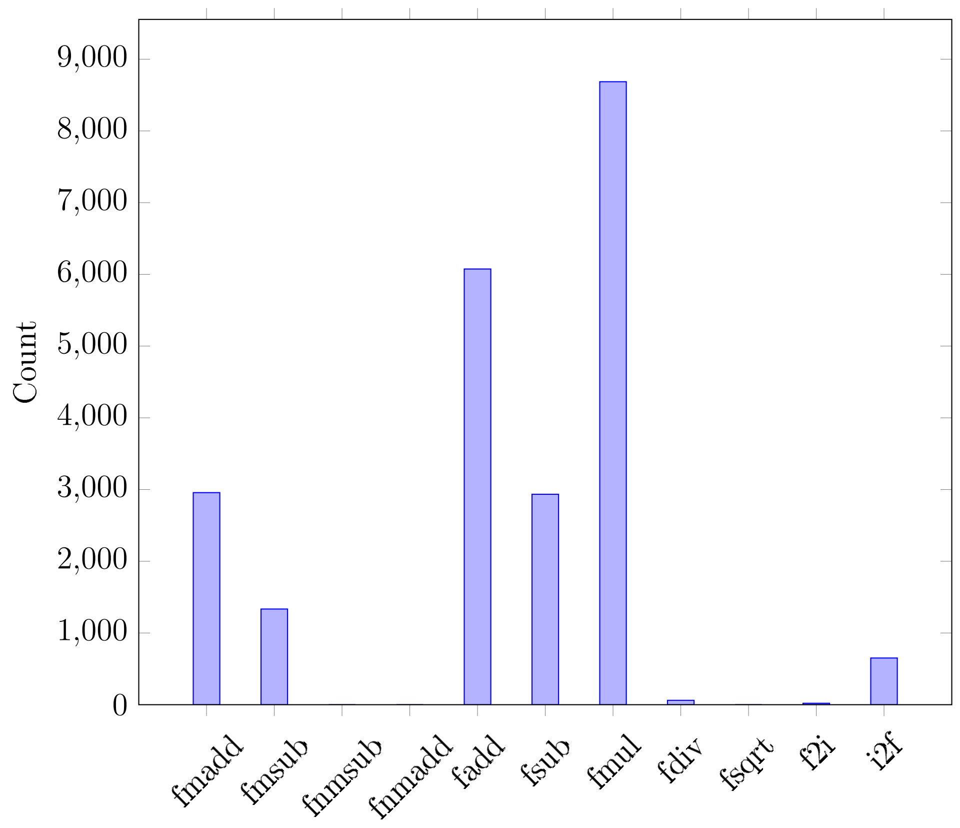

Figure 5.4: Distribution of FPU instructions in OPUS benchmark run, with better compiler optimization

Operation	Standard[pJ]	Custom[pJ]
Add/sub	44792.30	129449.42
Mul	72463.74	135946.15
Div	2662.63	2476.96
Sqrt	0.00	0.00
FMAC[7]	70613.30	70883.80
F2I	77.50	76.84
I2F	2761.41	2719.01
Idle/Noop	1534596.34	1434762.30
Total	*1727967.22*	*1776314.47*

Table 5.12: Power and energy consumed for OPUS benchmark with improved FMADD detection and optimization, grouped by FPU op

Chapter 6

Discussion and future work

The energy consumption results in Section 5.3.2 showed some surprising findings; if the compiler had optimized sequences of multiplications and additions into *fused-multiply-and-add* operations, the actual energy consumed for the Opus benchmark would have *increased* even though it would have completed marginally faster.

When looking at the average power and energy for each operation in Section 5.3.1, the most notable difference is the *add/sub*, with an increase in average power of almost 45%. This is due to lack of optimization; according to the simulation reports, approximately $157\mu W$ were consumed by toggling of the unused multiplier logic in the FMA pipeline. With input latches in place the resulting power consumption would have been closer to $560\mu W$ - approximately 13% higher than the original PULP FPU. The issue with *add/sub* taking 2 clock cycles is simply due to poor pipelining, and should be easily fixable with some manual register placement. With that resolved the energy/op would come closer to the same figures as the original PULP.

The results in Figure 5.2 clearly shows that there is much optimization left to be done on the customized FMA with regards to pipelining and timing. With a massive drop in maximum frequency from $140MHz$ to $119MHz$ we can see evidence that the changes have restricted the options available to the retiming algorithm.

6.1 Unum and posits

Unum and posits[3] are part of a new proposal for floating-point number representation proposed in 2013 by John L. Gustafson, and elaborated in his book *The End of Error*[20]. It is not very different from the IEEE 754 format, apart from simplifying things like removing infinity and instead capping values at the min-/maximum representable value. Therefore, it should be feasible to modify an IEEE 754-compliant FPU to implement support for it.

6.2 Transprecision

As the complexity, and therefore also the area, power and time increases with the number of bits needed to compute a result, reducing the precision can be a great way of saving power when the application allows[29]. There is currently ongoing research in the topic of *transprecision floating-point* at *ETH Zürich* and *University of Bologna*, where they have proposed two additional floating-point formats at 8 and 16 bits[43], along with hardware to work with the formats.

There has also been research into variable-precision FMA units that track the need for accuracy and compute results at reduced power consumption when the full precision is not required[24].

6.3 64-bit precision on 32-bit hardware

For small, low power designs a double-precision 64-bit FPU is usually considered too costly to implement in an otherwise 32-bit system. In one paper[25] from 2002 a design was proposed that enabled execution of double-precision operations on a single-precision FPU with very low hardware overhead. This was achieved by splitting the 64-bit operands into two 32-bit and computing the exponent and alignment first, before performing computation on the 53-bit fraction in multiple following cycles.

6.4 Hardware sharing and reuse

Because of the limitation specified in Section 1.1 that the scope is a single-issue in-order pipeline, it should be feasible to avoid any concurrent activity in the integer and floating-point ALUs. That should in turn allow a 32-bit integer ALU to be reused for computing the fraction in single-precision floating-point operations.By doing this it could be possible to greatly reduce the overhead of adding floating-point support to an otherwise small system.

This method has previously been employed for commercial processors like the *MIPS R4200*[50].

Chapter 7

Conclusion

Surprisingly enough, it would seem like the optimizations performed on the PULP FPU to reuse the fused multiply-adder for regular multiplication and addition operations ended up making the overall energy consumption *significantly worse.* This may however be due to lack of optimization, and the results could have been better if more time had been invested in improvements.

However the area savings were not insignificant at around 12%, and when scaling down to newer process nodes where leakage becomes increasingly dominant, this difference could prove beneficial. In conclusion this does not seem to be a tradeoff that is worth it for now, at least not in *55nm*, unless area is at an extreme premium or the applications make extensive use of *fused multiply-add* operations. With better optimization it could have some potential, and coupled with some of the improvements suggested in Section 6.4 it could become worthwile.

It seems that for small, low-power implementations there would in most cases be more efficient to choose a small FPU without FMA and then implement the *FMADD* instructions in microcode (as described in Section 2.3.3).

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] Jacob Andersen, Kevin Seffensen, and Peter Jensen. A Generic UVM Scoreboard. `https://verificationacademy.com/verification-horizons/november-2015-volume-11-issue-3/a-generic-uvm-scoreboard`, November 2015. Online, accessed 2018-07-04.

[3] Next Generation Arithmetic. Unum and Posit - about. `https://posithub.org/about`. Online, accessed 2018-06-15.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[5] Ronan Barzic and Jean-Baptiste Brelot. NanoRV32 - GitHub repository. `https://github.com/rbarzic/nanorv32`, 2016. Online, accessed 2017-11-15.

[6] J. D. Bruguera and T. Lang. Floating-point fused multiply-add: reduced latency for floating-point addition. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 42–51, June 2005.

[7] Anantha P Chandrakasan, Samuel Sheng, and Robert W Brodersen. Low-power cmos digital design. *IEICE Transactions on Electronics*, 75(4):371–382, 1992.

[8] Mentor Graphics Corporation. PowerPro Power Estimation. `https://www.mentor.com/hls-lp/powerpro-rtl-low-power/power-estimation`. Online, accessed 2018-07-04.

[9] B. Davari, R. H. Dennard, and G. G. Shahidi. Cmos scaling for high performance and low power-the next ten years. *Proceedings of the IEEE*, 83(4):595–606, Apr 1995.

[10] Gaurav Dhiman, Kishore Kumar Pusukuri, and Tajana Rosing. Analysis of dynamic voltage scaling for system level energy management. *USENIX HotPower*, 8, 2008.

[11] Alan Mischenko et al. ABC: A System for Sequential Synthesis and Verification. `https://people.eecs.berkeley.edu/~alanmi/abc/`. Online, accessed 2018-06-15.

[12] Krste Asanović et al. Rocket Chip Generator - GitHub repository. `https://github.com/freechipsproject/rocket-chip`, 2016. Online, accessed 2017-09-01.

[13] Krste Asanović et al. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[14] Michael Schaffner et. al. PULP - iis-projects wiki. `http://iis-projects.ee.ethz.ch/index.php/PULP`. Online, accessed 2018-07-04.

[15] RISC-V Foundation. RISC-V Foundation - about. `http://riscv.org/riscv-foundation`. Online, accessed 2017-11-24.

[16] S. Galal and M. Horowitz. Energy-Efficient Floating-Point Unit Design. *IEEE Transactions on Computers*, 60(7):913–922, July 2011.

[17] Michael Gautschi, Andreas Traber, and Florian Zaruba. Imperio, the first PULPino tapeout. `http://asic.ethz.ch/2015/Imperio.html`, 2015.

[18] Andreas Gerstlauer. SpecC homepage. `http://www.cecs.uci.edu/~specc/`. Online, accessed 2017-11-17.

[19] UC Berkeley Architecture Research group. Berkeley Hardware Floating-Point Units - GitHub repository. `https://github.com/ucb-bar/berkeley-hardfloat`, 2017. Online, accessed 2017-12-11.

[20] John L Gustafson. *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.

[21] Synopsys Inc. Synopsys home page. `http://www.synopsys.com/`. Online, accessed 2017-11-15.

[22] Accelera Systems Initiative. SystemC homepage. `http://www.accellera.org/downloads/standards/systemc`. Online, accessed 2017-11-17.

[23] JEDEC. Gate Equivalents - dictionary definition. `https://www.jedec.org/standards-documents/dictionary/terms/gate-equivalent-1-cmos`. Online, accessed 2018-07-04.

[24] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. In *2012 IEEE International Solid-State Circuits Conference*, pages 182–184, Feb 2012.

[25] Seungchul Kim, Yongjoo Lee, Wookyeong Jeong, and Yongsurk Lee. Low cost floating point arithmetic unit design. In *Proceedings. IEEE Asia-Pacific Conference on ASIC,*, pages 217–220, 2002.

[26] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse engineering x86 processor microcode.

[27] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pages 199–202. IEEE, 2014.

[28] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, Jun 1991.

[29] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn. The transprecision computing paradigm: Concept, design, and applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1105–1110, March 2018.

[30] Torbjørn Viem Ness. Low Power Floating-Point Unit for RISC-V - project report, 12 2017.

[31] Torbjørn Viem Ness. Contribution to GitHub repository pulp-platform/fpu - multiplication bug. https://github.com/pulp-platform/fpu/pull/7, May 2018.

[32] Torbjørn Viem Ness. Contribution to GitHub repository pulp-platform/fpu - tool compatibility. https://github.com/pulp-platform/fpu/pull/9, May 2018.

[33] H. J. Osten, J. P. Liu, P. Gaworzewski, E. Bugiel, and P. Zaumseil. High-k gate dielectrics with ultra-low leakage current based on praseodymium oxide. In *International Electron Devices Meeting 2000. Technical Digest. IEDM (Cat. No.00CH37138)*, pages 653–656, Dec 2000.

[34] R. Peesapati, K. K. Anumandla, and S. L. Sabat. Performance evaluation of floating point differential evolution hardware accelerator on fpga. In *2016 IEEE Region 10 Conference (TENCON)*, pages 3173–3178, Nov 2016.

[35] PULP project. PULP platform FPU - GitHub repository. https://github.com/pulp-platform/fpu, 2018. Online, accessed 2018-02-26.

[36] PULP project. RI5CY: RISC-V Core - GitHub repository. https://github.com/pulp-platform/zero-riscy, 2018. Online, accessed 2018-07-04.

[37] PULP project. RI5CY: RISC-V Core - GitHub repository. https://github.com/pulp-platform/riscv, 2018. Online, accessed 2018-07-04.

[38] Antonio Pullini and Davide Rossi. Imperio, the first PULPino tapeout. http://asic.ethz.ch/2017/Mr.Wolf.html, 2017.

[39] Robert Resnick, Jearl Walker, and D Halliday. *Fundamentals of physics*, volume 1. John Wiley, 1988.

[40] Soheil Salehi and Ronald F DeMara. Energy and area analysis of a floating-point unit in 15nm CMOS process technology. In *SoutheastCon 2015*, pages 1–5. IEEE, 2015.

[41] Eric M. Schwarz. *Binary Floating-Point Unit Design*, pages 189–208. Springer US, Boston, MA, 2006.

[42] Hayden So. Introduction to Fixed Point Number Representation. `http://www-inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html`, 2006. Online, accessed 2018-06-15.

[43] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1051–1056, March 2018.

[44] Potential Ventures. CoCoTB - GitHub repository. `https://github.com/potentialventures/cocotb`, 2018. Online, accessed 2018-02-26.

[45] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture.* PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

[46] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, May 2017.

[47] Andrew Waterman and contributors. RISCV GNU Compiler Toolchain - GitHub repository. `https://github.com/riscv/riscv-gnu-toolchain`, 2017. Online, accessed 2017-10-11.

[48] Martin H. Weik. *microcode*, pages 1012–1012. Springer US, Boston, MA, 2001.

[49] Clifford Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`. Online, accessed 2018-06-15.

[50] B. Zivkov, B. Ferguson, and M. Gupta. R4200: a high-performance mips microprocessor for portables. In *Compcon Spring '94, Digest of Papers.*, pages 18–25, Feb 1994.

Appendices

Appendix A

High-level Python model

The following appendices contain the Python files used in the simulation test bench.

A.1 common.py

This file contains the common functions and definitions used across the Python testbench infrastructure.

```python
import cocotb
from cocotb.decorators import coroutine
from cocotb.triggers import Timer

from simple_model import *

# Set the default clock period, 10_000ps = 100MHz
DEFAULT_CKPER = 10000

@cocotb.coroutine
def reset_gen(reset_n, duration=20000):
    """Generator for resetting at the beginning of the test"""
    reset_n <= 0
    yield Timer(duration)
    reset_n <= 1
    reset_n._log.info("Reset done")

def data_gen():
    """Generator for random 32-bit input"""
    while True:
        yield random.randint(0,0xFFFFFFFF)

@cocotb.coroutine
def clock_gen(signal, period=DEFAULT_CKPER):
    """Generate the clock signal"""
    while True:
        signal <= 0
        yield Timer(period>>1)
        signal <= 1
        yield Timer(period>>1)
```

A.2 simple_model.py

This is the simplified high-level model used for verifying correctness of the FPU simulation results.

```
import ctypes as ct, math, random

INT32_MAXVAL  =  0x7FFFFFFF
INT32_MINVAL  =  0x80000000
UINT32_MAXVAL =  0xFFFFFFFF

C_FLEN = 32
RV32F_NAN_CANONICAL      = 0x7fc00000 # The default (quiet) NaN
↪  result to return unless something else is specified
RV32F_NAN_INF_MASK       = 0x7f800000 # The mask of a signalling
↪  NaN or inf - NaN or inf is determined by whether or not the
↪  entire significand is 0
RV32F_SIGNIFICAND_MASK   = 0x007fffff
RV32D_NAN_CANONICAL      = 0x7ff8000000000000 # The default
↪  (quiet) NaN result to return unless something else is
↪  specified
RV32D_NAN_INF_MASK       = 0x7ff0000000000000 # The mask of a
↪  signalling NaN or inf - NaN or inf is determined by whether
↪  or not the entire significand is 0
RV32D_SIGNIFICAND_MASK   = 0x000fffffffffffff
RV32D_SINGLE_MASK        = 0xFFFFFFFF00000000 # The upper 32
↪  bits of narrower (single-precision) numbers should be set
↪  to '1' (secion 9.2 in RISCV-spec v2.2)

C_FPU_ADD_CMD       = 0x0;
C_FPU_SUB_CMD       = 0x1;
C_FPU_MUL_CMD       = 0x2;
C_FPU_DIV_CMD       = 0x3;
C_FPU_I2F_CMD       = 0x4;
C_FPU_F2I_CMD       = 0x5;
C_FPU_SQRT_CMD      = 0x6;
C_FPU_NOP_CMD       = 0x7;
C_FPU_FMADD_CMD     = 0x8;
C_FPU_FMSUB_CMD     = 0x9;
C_FPU_FNMADD_CMD    = 0xA;
C_FPU_FNMSUB_CMD    = 0xB;
```

```
C_RM_NEAREST    = 0x0;
C_RM_TRUNC      = 0x1;
C_RM_PLUSINF    = 0x3;
C_RM_MINUSINF   = 0x2;
C_RM_NEAREST_MAX = 0x4;

# For prettier logging
op_names = ["add", "sub", "mul", "div", "i2f", "f2i", "sqrt",
 ↪  "nop", "fmadd", "fmsub", "fnmadd", "fnmsub"]

# Interprets the submitted data as raw bytes and returns a
 ↪  floating-point number
def int2float(i):
    return ct.c_float.from_buffer(ct.c_uint32(i)).value

# Returns the data of the supplied float untouched but as an
 ↪  int, to allow use of python's number printing functions
def float2int(f):
    return ct.c_uint.from_buffer(ct.c_float(f)).value

# Check that an alleged single-precision number is legally
 ↪  NaN-boxed (see section 9.2 of RISC-V ISA spec)
def isNanBoxed(num):
    #print("isNanBoxed called with num=0x{:08x} ".format(num))
    if C_FLEN == 32 or ((num & RV32D_SINGLE_MASK) ==
    ↪  RV32D_SINGLE_MASK):
        # If double precision is not used, just return True to
        ↪  signal everything is OK
        #print("\tTrue")
        return True
    #print("\tFalse")
    return False

def isNaN(num, precision='s'):
    if type(num) is float:
        if precision is 'd':
            tmp = double2long(num)
        else:
            tmp = float2int(num)
    else:
        tmp = num
    if precision is 'd':
        if (tmp & RV32D_NAN_CANONICAL) == RV32D_NAN_CANONICAL
         ↪  or isSigNaN(num, precision):
            return True
    else:
        if (not isNanBoxed(tmp)) or (tmp & RV32F_NAN_CANONICAL)
         ↪  == RV32F_NAN_CANONICAL or isSigNaN(num, precision):
            return True
    return False

# NOTE: When using python float, signalling NaN is NOT
↪   supported - as soon as a number is cast or converted by
↪   python, it is automatically forced into quiet NaN -
↪   0x7fcPPPPP ('P'=payload bytes)
def isSigNaN(num, precision='s'):
    if type(num) is float:
        if precision is 'd':
            tmp = double2long(num)
        else:
            tmp = float2int(num)
    else:
        tmp = num
    if precision is 'd':
        if (tmp & RV32D_NAN_CANONICAL) == RV32D_NAN_INF_MASK
         ↪  and (tmp & RV32D_SIGNIFICAND_MASK) != 0:
            return True
    else:
        if (tmp & RV32F_NAN_CANONICAL) == RV32F_NAN_INF_MASK
         ↪  and (tmp & RV32F_SIGNIFICAND_MASK) != 0:
            return True
    return False

def simple_model(operation, operand_a, operand_b, operand_c,
↪   rounding_mode=0):
    if operand_a is not None:
        f_operand_a = int2float(operand_a)
    if operand_b is not None:
        f_operand_b = int2float(operand_b)
    if operand_c is not None:
        f_operand_c = int2float(operand_c)
```

```
# The sign of the result is important in some cases as
↪  IEEE-754 has two representations of 0
res_sign = 0

if operation == C_FPU_ADD_CMD:
    f_res = f_operand_a + f_operand_b
elif operation == C_FPU_SUB_CMD:
    f_res = f_operand_a - f_operand_b
elif operation == C_FPU_MUL_CMD:
    f_res = f_operand_a * f_operand_b
    res_sign = (operand_a & (1 << 31)) ^ (operand_b & (1 <<
    ↪  31))
elif operation == C_FPU_DIV_CMD:
    if (f_operand_b == 0):
        print("Invalid operation: divide by zero")
        res_sign = (operand_a & (1 << 31)) ^ (operand_b &
        ↪  (1 << 31))
        # Return an inf value with the correct sign
        return (RV32F_NAN_INF_MASK | res_sign)
    f_res = f_operand_a / f_operand_b
elif operation == C_FPU_I2F_CMD:
    f_res = ct.c_float(ct.c_int32(operand_a).value).value
elif operation == C_FPU_F2I_CMD:
    if isNaN(operand_a):
        return RV32F_NAN_CANONICAL
    if (f_operand_a > INT32_MAXVAL):
        return INT32_MAXVAL
    elif (f_operand_a < -INT32_MINVAL):
        return INT32_MINVAL
    # TODO: handle rounding mode
    return ct.c_uint(int(f_operand_a)).value
elif operation == C_FPU_SQRT_CMD:
    if (f_operand_a < 0):
        print("Invalid operation: sqrt of a negative
        ↪  number: {0:f}".format(f_operand_a))
        return RV32F_NAN_CANONICAL
    f_res = math.sqrt(f_operand_a)
elif operation == C_FPU_NOP_CMD:
    return 0 # TODO: is there a better value to return
    ↪  here? Don't really care
    elif operation == C_FPU_FMADD_CMD:
        f_res = (f_operand_a * f_operand_b) + f_operand_c
        res_sign = (operand_a & (1 << 31)) ^ (operand_b & (1 <<
         ↪  31))
        print("res_sign: {0:08x}".format(res_sign))
    elif operation == C_FPU_FMSUB_CMD:
        f_res = (f_operand_a * f_operand_b) - f_operand_c
        res_sign = (operand_a & (1 << 31)) ^ (operand_b & (1 <<
         ↪  31))
        print("res_sign: {0:08x}".format(res_sign))
    elif operation == C_FPU_FNMADD_CMD:
        f_res = -((f_operand_a * f_operand_b) + f_operand_c)
        res_sign = ~((operand_a & (1 << 31)) ^ (operand_b & (1
         ↪  << 31))) & (1 << 31)
        print("res_sign: {0:08x}".format(res_sign))
    elif operation == C_FPU_FNMSUB_CMD:
        f_res = -((f_operand_a * f_operand_b) - f_operand_c)
        res_sign = ~((operand_a & (1 << 31)) ^ (operand_b & (1
         ↪  << 31))) & (1 << 31)
        print("res_sign: {0:08x}".format(res_sign))
    else:
        print("Illegal operation: " + operation)

    # Handle NaN
    if ( # 2-operand computation
        (operation == C_FPU_ADD_CMD
        or operation == C_FPU_SUB_CMD
        or operation == C_FPU_MUL_CMD
        or operation == C_FPU_DIV_CMD
        ) and (
            (isNaN(operand_a) or isNaN(operand_b))
        )
    ) or ( # 3-operand computation
        (operation == C_FPU_FMADD_CMD
        or operation == C_FPU_FMSUB_CMD
        or operation == C_FPU_FNMADD_CMD
        or operation == C_FPU_FNMSUB_CMD
        ) and (
            (isNaN(operand_a) or isNaN(operand_b) or
             ↪  isNaN(operand_c))
        )
```

```
) or ( # 1-operand operations
    (operation == C_FPU_F2I_CMD
    or operation == C_FPU_SQRT_CMD
    ) and (
        isNaN(operand_a)
    )
):
    return RV32F_NAN_CANONICAL

# TODO: care about rounding mode
rounded_res = ct.c_float(f_res).value
if (rounded_res != f_res):
    print("Inexact result")
i_res = float2int(rounded_res)
if (i_res & 0x7FFFFFFF == 0):
    # Make sure the sign bit is correct if the result is
     ↪  -0.0, as Python doesn't care about such trivial but
     ↪  important details
    i_res = (i_res & 0x7FFFFFFF) | res_sign
return abs(i_res)
```

A.3 fpu_private_framework.py

This is a part of the Python test harness specific to the standard PULP FPU design.

```
import random, ctypes as ct, math, os

import cocotb, logging
from cocotb.decorators import coroutine
from cocotb.triggers import Timer, RisingEdge, FallingEdge,
↪  ReadOnly
from cocotb.result import TestFailure, TestSuccess, TestError

from cocotb.regression import TestFactory

from common import *

@cocotb.coroutine
def run_test(dut, operation=0, operand_a=0, operand_b=0,
↪  operand_c=0, rounding_mode=0, repetitions=1):
    """Setup testbench and run a test
    module fpu_private
      (
       //Clock and reset
       input logic                 clk_i,
       input logic                 rst_ni,
       // enable
       input logic                 fpu_en_i,
       // inputs
       input logic [C_OP-1:0]      operand_a_i,
       input logic [C_OP-1:0]      operand_b_i,
       input logic [C_OP-1:0]      operand_c_i,
       input logic [C_RM-1:0]      rm_i,
       input logic [C_CMD-1:0]     fpu_op_i,
       input logic [C_PC-1:0]      prec_i,

       // outputs
       output logic [C_OP-1:0]     result_o,
       output logic                valid_o,
       output logic [C_FFLAG-1:0] flags_o,
       output logic                divsqrt_busy_o
    );
"""
dut.log.setLevel(logging.DEBUG)

if (repetitions < 1):
    raise TestFailure("Repetitions must be a positive
     ↪  number")

try:
    ckper = int(os.environ.get('CKPER', DEFAULT_CKPER))
except:
    ckper = DEFAULT_CKPER

if operand_a is None:
    random_a = True
else:
    random_a = False
if operand_b is None:
    random_b = True
else:
    random_b = False
if operand_c is None:
    random_c = True
else:
    random_c = False

dut._log.info("Testing op:
 ↪  {0:s}".format(op_names[operation]))
dut.fpu_en_i <= 0
cocotb.fork(clock_gen(dut.clk_i, ckper))
yield reset_gen(dut.rst_ni)
clkedge = RisingEdge(dut.clk_i)
negedge = FallingEdge(dut.clk_i)

dut.prec_i <= 0 # Number of bits precision in the division
 ↪  result
yield clkedge

for i in range(repetitions):
    if random_a:
        operand_a = data_gen().next()
        if random_b:
            operand_b = data_gen().next()
        if random_c:
            operand_c = data_gen().next()

        dut._log.info("\nTesting round {3} of {4}: a={0:f},
        ↪   b={1:f}, c={2:f}".format(int2float(operand_a),

                                                    ↪   int2float(operand_b),
                                                    ↪   int2float(operand_c),
                                                    ↪   i+1,
                                                    ↪   repetitions))

        dut.fpu_en_i <= 1
        dut.operand_a_i <= operand_a
        dut.operand_b_i <= operand_b
        dut.operand_c_i <= operand_c
        dut.rm_i <= rounding_mode
        dut.fpu_op_i <= operation

        yield clkedge
        dut._log.debug("valid_o = {0}, divsqrt_busy_o = {1},
        ↪   result_o = {2}, flags_o = {3}"
                                .format(dut.valid_o.value,
                                ↪   dut.divsqrt_busy_o.value,
                                        dut.result_o.value,
                                        ↪   dut.flags_o.value))
        dut.fpu_en_i <= 0

        cycles_spent = 0
        yield ReadOnly()
        while operation != C_FPU_NOP_CMD and
        ↪   dut.valid_o.value.integer == 0:
            #print("valid_o = {0}, divsqrt_busy_o = {1},
            ↪   result_o = {2}, flags_o = {3}"
            #                  .format(dut.valid_o.value,
            ↪   dut.divsqrt_busy_o.value,
            #                          dut.result_o.value,
            ↪   dut.flags_o.value))
            if cycles_spent >= 10:
                #raise TestError("Timed out waiting for
                ↪  result")
                dut._log.error("Timed out waiting for result")
                break
            yield clkedge
            yield ReadOnly()
            cycles_spent = cycles_spent + 1

        dut.result_o._log.info(dut.result_o.value)
        dut.valid_o._log.info(dut.valid_o.value)
        dut.flags_o._log.info(dut.flags_o.value)

        dut.result_o._log.info("Result:
        ↪   {0:f}".format(int2float(int(dut.result_o.value))))
        computed = int(dut.result_o.value)
        expected = simple_model(operation, operand_a,
        ↪   operand_b, operand_c, rounding_mode)
        if computed != expected:
            #raise TestError("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))
            dut._log.error("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))

        # End simulation at a negative edge to get some time to
        ↪   see the result on waveforms
        yield negedge
```

Makefile

```
# Makefile for use with CoCoTB, based on DFF example Makefile
# - Torbjørn Viem Ness (NTNU), spring 2018

CWD=$(shell pwd | sed 's/:/\\:/')
COCOTB?=/home/tone/cocotb

# TODO: specifying TOPLEVEL_LANG may not be needed
TOPLEVEL_LANG ?=verilog
```

```
VERILOG_SOURCES
↪   =$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_defs.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/defines_fpu.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_fmac/fpu_defs_fmac.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_div_sqrt_tp_nlp/fpu_defs_div_sqrt_tp.s
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_private.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_mult.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_add.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_itof.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_ftoi.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_norm.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_core.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fp_fma_wrapper.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpexc.sv
VERILOG_SOURCES +=$(wildcard
↪   $(CWD)/../../../../../pulp-fpu/hdl/fpu_fmac/*.sv)
VERILOG_SOURCES +=$(wildcard
↪   $(CWD)/../../../../../pulp-fpu/hdl/fpu_div_sqrt_tp_nlp/*.sv)

#EXTRA_ARGS=-s fpu_private
ifeq ($(SIM),questa)
    EXTRA_ARGS=-sfcu
else
    # Default is Icarus
    EXTRA_ARGS=-g2012
endif

TOPLEVEL=fpu_private
MODULE=$(TOPLEVEL)_cocotb
```

```
CUSTOM_SIM_DEPS=$(CWD)/Makefile

VSIM_ARGS=-t 1ps
ifeq ($(WAVES),1)
    VSIM_ARGS +=-pli novas_fli.so
endif

include $(COCOTB)/makefiles/Makefile.inc
include $(COCOTB)/makefiles/Makefile.sim

# list all required Python files here
sim: $(MODULE).py
```

A.4 fpu_tiny_framework.py

This is a part of the Python test harness specific to the small PULP FPU design.

```
import random, ctypes as ct, math, os

import cocotb, logging
from cocotb.decorators import coroutine
from cocotb.triggers import Timer, RisingEdge, FallingEdge,
↪  ReadOnly
from cocotb.result import TestFailure, TestSuccess, TestError

from cocotb.regression import TestFactory

from common import *

@cocotb.coroutine
def run_test(dut, operation=0, operand_a=0, operand_b=0,
↪  operand_c=0, rounding_mode=0, repetitions=1):
    """Setup testbench and run a test
    module riscv_fpu
      (
       //Clock and reset
       input logic              clk,
       input logic              rst_n,

       //Input Operands
       input logic [C_OP-1:0]   operand_a_i,
       input logic [C_OP-1:0]   operand_b_i,
       input logic [C_RM-1:0]   rounding_mode_i,    //Rounding
↪  Mode
       input logic [C_CMD-1:0] operator_i,
       input logic              enable_i,

       input logic              stall_i,

       output logic [C_OP-1:0] result_o,
       //Output-Flags
       output logic             fpu_ready_o,   // high if fpu is
↪  ready
```

```
        output logic             result_valid_o // result is
↪   valid
        );
    """
    dut.log.setLevel(logging.DEBUG)

    if (repetitions < 1):
        raise TestFailure("Repetitions must be a positive
         ↪   number")

    try:
        ckper = int(os.environ.get('CKPER', DEFAULT_CKPER))
    except:
        ckper = DEFAULT_CKPER

    if operand_a is None:
        random_a = True
    else:
        random_a = False
    if operand_b is None:
        random_b = True
    else:
        random_b = False
    if operand_c is None:
        random_c = True
    else:
        random_c = False

    dut._log.info("Testing op:
     ↪   {0:s}".format(op_names[operation]))
    dut.enable_i <= 0
    dut.stall_i <= 0
    cocotb.fork(clock_gen(dut.clk, ckper))
    yield reset_gen(dut.rst_n)
    clkedge = RisingEdge(dut.clk)
    negedge = FallingEdge(dut.clk)

    yield clkedge

    for i in range(repetitions):
        if random_a:
```

```
        operand_a = data_gen().next()
    if random_b:
        operand_b = data_gen().next()
    if random_c:
        operand_c = data_gen().next()

    dut._log.info("\nTesting round {3} of {4}: a={0:f},
    ↪   b={1:f}, c={2:f}".format(int2float(operand_a),

                                                    ↪   int2float(operand_b),
                                                    ↪   int2float(operand_c),
                                                    ↪   i+1,
                                                    ↪   repetitions))

    dut.enable_i <= 1
    dut.operand_a_i <= operand_a
    dut.operand_b_i <= operand_b
    dut.rounding_mode_i <= rounding_mode
    dut.operator_i <= operation

    yield clkedge
    exceptions = "{0}0{1}{2}{3}".format(dut.fpcore.IV_SO,
    ↪   dut.fpcore.OF_SO.value, dut.fpcore.UF_SO,
    ↪   dut.fpcore.IX_SO)
    dut._log.debug("result_valid_o = {0}, result_o = {1},
    ↪   flags_o = {2}"
                         .format(dut.result_valid_o.value,
                         ↪   dut.result_o.value,
                         ↪   exceptions))

    cycles_spent = 0
    yield ReadOnly()
    while operation != C_FPU_NOP_CMD and
    ↪   dut.result_valid_o.value.integer == 0:
        #print("result_valid_o = {0}, divsqrt_busy_o = {1},
        ↪   result_o = {2}, flags_o = {3}"
        #                 .format(dut.result_valid_o.value,
        ↪   dut.divsqrt_busy_o.value,
        #                         dut.result_o.value,
        ↪   dut.flags_o.value))
        if cycles_spent >= 10:
                #raise TestError("Timed out waiting for
                ↪  result")
                dut._log.error("Timed out waiting for result")
                break
            yield clkedge
            yield ReadOnly()
            cycles_spent = cycles_spent + 1

        exceptions = "{0}0{1}{2}{3}".format(dut.fpcore.IV_SO,
        ↪   dut.fpcore.OF_SO.value, dut.fpcore.UF_SO,
        ↪   dut.fpcore.IX_SO)
        dut.result_o._log.info(dut.result_o.value)
        dut.result_valid_o._log.info(dut.result_valid_o.value)
        dut.fpcore._log.info(exceptions)

        dut.result_o._log.info("Result:
        ↪   {0:f}".format(int2float(int(dut.result_o.value))))
        computed = int(dut.result_o.value)
        expected = simple_model(operation, operand_a,
        ↪   operand_b, operand_c, rounding_mode)
        if computed != expected:
            #raise TestError("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))
            dut._log.error("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))

        # End simulation at a negative edge to get some time to
        ↪   see the result on waveforms
        yield negedge
```

Makefile

```
# Makefile for use with CoCoTB, based on DFF example Makefile
# - Torbjørn Viem Ness (NTNU), spring 2018

CWD=$(shell pwd | sed 's/:/\\:/')
COCOTB?=/home/tone/cocotb

# TODO: specifying TOPLEVEL_LANG may not be needed
```

```
TOPLEVEL_LANG ?=verilog

VERILOG_SOURCES
↪  =$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_defs.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/defines_fpu.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_utils/fpu_ff.sv
VERILOG_SOURCES +=$(CWD)/../../../../../riscv_fpu.sv
#VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/riscv_fpu.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_mult.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_add.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_itof.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_ftoi.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_norm.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_core.sv
VERILOG_SOURCES
↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpexc.sv

#EXTRA_ARGS=-s fpu_private
ifeq ($(SIM),questa)
    EXTRA_ARGS=-sfcu
else
    # Default is Icarus
    EXTRA_ARGS=-g2012
endif

TOPLEVEL=riscv_fpu
MODULE=$(TOPLEVEL)_cocotb

CUSTOM_SIM_DEPS=$(CWD)/Makefile

VSIM_ARGS=-t 1ps
ifeq ($(WAVES),1)
```

```
    VSIM_ARGS +=-pli novas_fli.so
endif

include $(COCOTB)/makefiles/Makefile.inc
include $(COCOTB)/makefiles/Makefile.sim

# list all required Python files here
sim: $(MODULE).py
```

A.5 fpu_custom_framework.py

This is a part of the Python test harness specific to the customized PULP FPU design.

```
import random, ctypes as ct, math, os

import cocotb, logging
from cocotb.decorators import coroutine
from cocotb.triggers import Timer, RisingEdge, FallingEdge,
↪ ReadOnly
from cocotb.result import TestFailure, TestSuccess, TestError

from cocotb.regression import TestFactory

from common import *

@cocotb.coroutine
def run_test(dut, operation=0, operand_a=0, operand_b=0,
↪ operand_c=0, rounding_mode=0, repetitions=1):
    """Setup testbench and run a test
    module pulp_custom_fpu #( parameter FLEN = 32 )
    (
            input                               clk,
            input                               reset_n,
            // Control input signals
            input                               enable,
            input [2:0]                      rm,
            input [3:0]                      operation,
            // Data input signals
            input [FLEN-1:0]        operand_a,
            input [FLEN-1:0]        operand_b,
            input [FLEN-1:0]        operand_c,
            // Output signals
            output [FLEN-1:0]        result,

↪           output                              result_valid,
            output [4:0]                 exceptions,
            output                              divsqrt_busy
    );
    """
    dut.log.setLevel(logging.DEBUG)

    if (repetitions < 1):
        raise TestFailure("Repetitions must be a positive
         ↪  number")

    try:
        ckper = int(os.environ.get('CKPER', DEFAULT_CKPER))
    except:
        ckper = DEFAULT_CKPER

    if operand_a is None:
        random_a = True
    else:
        random_a = False
    if operand_b is None:
        random_b = True
    else:
        random_b = False
    if operand_c is None:
        random_c = True
    else:
        random_c = False

    dut._log.info("Testing op:
     ↪  {0:s}".format(op_names[operation]))
    dut.enable <= 0
    cocotb.fork(clock_gen(dut.clk, ckper))
    yield reset_gen(dut.reset_n)
    clkedge = RisingEdge(dut.clk)
    negedge = FallingEdge(dut.clk)
    yield clkedge

    for i in range(repetitions):
        if random_a:
            operand_a = data_gen().next()
        if random_b:
            operand_b = data_gen().next()
        if random_c:
            operand_c = data_gen().next()
        dut._log.info("\nTesting round {3} of {4}: a={0:f},
         ↪  b={1:f}, c={2:f}".format(int2float(operand_a),

                                                    ↪  int2float(operand_b),
                                                    ↪  int2float(operand_c),
                                                    ↪  i+1,
                                                    ↪  repetitions))

        dut.enable <= 1
        dut.operand_a <= operand_a
        dut.operand_b <= operand_b
        dut.operand_c <= operand_c
        dut.rm <= rounding_mode
        dut.operation <= operation

        yield clkedge
        dut._log.debug("result_valid = {0}, divsqrt_busy = {1},
         ↪  result = {2}, exceptions = {3}"
                             .format(dut.result_valid.value,
                              ↪  dut.divsqrt_busy.value,
                                  dut.result.value,
                                   ↪  dut.exceptions.value))
        dut.enable <= 0

        cycles_spent = 0
        yield ReadOnly()
        while operation != C_FPU_NOP_CMD and
         ↪  dut.result_valid.value.integer == 0:
            #print("result_valid = {0}, divsqrt_busy = {1},
             ↪  result = {2}, exceptions = {3}"
            #                  .format(dut.result_valid.value,
             ↪  dut.divsqrt_busy.value,
            #                          dut.result.value,
             ↪  dut.exceptions.value))
            if cycles_spent >= 10:
                #raise TestError("Timed out waiting for
                 ↪  result")
                dut._log.error("Timed out waiting for result")
                break
            yield clkedge
            yield ReadOnly()
            cycles_spent = cycles_spent + 1

        dut.result._log.info(dut.result.value)
        dut.result_valid._log.info(dut.result_valid.value)
        dut.exceptions._log.info(dut.exceptions.value)

        dut.result._log.info("Result:
        ↪   {0:f}".format(int2float(int(dut.result.value)))
        computed = int(dut.result.value)
        expected = simple_model(operation, operand_a,
        ↪   operand_b, operand_c, rounding_mode)
        if computed != expected:
            #raise TestError("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))
            dut._log.error("Computed result 0x{0:08x} differs
            ↪   from expected: 0x{1:08x}".format(computed,
            ↪   expected))

        # End simulation at a negative edge to get some time to
        ↪   see the result on waveforms
        yield negedge
```

Makefile

```
# Makefile for use with CoCoTB, based on DFF example Makefile
# - Torbjørn Viem Ness (NTNU), spring 2018

CWD=$(shell pwd | sed 's/:/\\:/')
COCOTB?=/home/tone/cocotb

# TODO: specifying TOPLEVEL_LANG may not be needed
TOPLEVEL_LANG ?=verilog

VERILOG_SOURCES
↪   =$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_defs.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/defines_fpu.sv
VERILOG_SOURCES
↪   +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_fmac/fpu_defs_fmac.sv
```

```
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_div_sqrt_tp_nlp/fpu_defs_div_sqrt_tp.s
VERILOG_SOURCES +=$(CWD)/../../../../../pulp_customized.v
VERILOG_SOURCES +=$(CWD)/../../../../../conversion_core.v
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_mult.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_add.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_itof.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_ftoi.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_norm.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpu_core.sv
VERILOG_SOURCES +=$(CWD)/../../../../../fma_wrapper_custom.sv
VERILOG_SOURCES +=$(CWD)/../../../../../fmac_customized.sv
VERILOG_SOURCES
 ↪  +=$(CWD)/../../../../../pulp-fpu/hdl/fpu_v0.1/fpexc.sv
VERILOG_SOURCES +=$(wildcard
 ↪  $(CWD)/../../../../../../pulp-fpu/hdl/fpu_fmac/*.sv)
VERILOG_SOURCES +=$(wildcard
 ↪  $(CWD)/../../../../../../pulp-fpu/hdl/fpu_div_sqrt_tp_nlp/*.sv)

#EXTRA_ARGS=-s fpu_private
ifeq ($(SIM),questa)
    EXTRA_ARGS=-sfcu
else
    # Default is Icarus
    EXTRA_ARGS=-g2012 -u
     ↪  -I$(CWD)/../../../../../../pulp-fpu/hdl/fpu_v0.1/
     ↪  -I$(CWD)/../../../../../../pulp-fpu/hdl/fpu_fmac/
     ↪  -I$(CWD)/../../../../../../pulp-fpu/hdl/fpu_div_sqrt_tp_nlp/
endif

TOPLEVEL=pulp_custom_fpu
MODULE=$(TOPLEVEL)_cocotb

CUSTOM_SIM_DEPS=$(CWD)/Makefile
```

```
VSIM_ARGS=-t 1ps
ifeq ($(WAVES),1)
    VSIM_ARGS +=-pli novas_fli.so
endif

include $(COCOTB)/makefiles/Makefile.inc
include $(COCOTB)/makefiles/Makefile.sim

# list all required Python files here
sim: $(MODULE).py
```

A.6 CoCoTB stimuli

The following Python files are used together with the testbench framework (comprising of *common.py, simple_model.py* and the setup files for each design variant) for specifying input stimuli and running the tests:

all.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_ADD_CMD, C_FPU_SUB_CMD,
↪  C_FPU_MUL_CMD, C_FPU_DIV_CMD,
                                 C_FPU_I2F_CMD, C_FPU_F2I_CMD,
                                 ↪  C_FPU_SQRT_CMD,
                                 ↪  C_FPU_NOP_CMD,
                                 C_FPU_FMADD_CMD,
                                 ↪  C_FPU_FMSUB_CMD,
                                 ↪  C_FPU_FNMADD_CMD,
                                 ↪  C_FPU_FNMSUB_CMD])
factory.add_option("operand_a", [0, 0x3f800000, 0xbf800000,
↪  0x40800000]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [0x3fc00000, 0x00000000]) #
↪  1.5, 0.0
factory.add_option("operand_c", [0x40000000, 0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.generate_tests()
```

fadd.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_ADD_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [None]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

fmul.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_MUL_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [None]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

fdiv.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_DIV_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [None]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

fsqrt.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_SQRT_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [0]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

fmadd.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_FMADD_CMD,
↪  C_FPU_FMSUB_CMD, C_FPU_FNMADD_CMD, C_FPU_FNMSUB_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [None]) # 1.5, 0.0
factory.add_option("operand_c", [None]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [250])
factory.generate_tests()
```

ftoi.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_F2I_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [0]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

itof.py

```
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_I2F_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [0]) # 1.5, 0.0
factory.add_option("operand_c", [0]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

nop.py

```python
from fpu_test_framework import *

factory = TestFactory(run_test)
factory.add_option("operation", [C_FPU_NOP_CMD])
factory.add_option("operand_a", [None]) # 1.0, -1.0, 4.0
factory.add_option("operand_b", [None]) # 1.5, 0.0
factory.add_option("operand_c", [None]) # 2.0
factory.add_option("rounding_mode", [0])#[0,1,2,3])
factory.add_option("repetitions", [1000])
factory.generate_tests()
```

A.7 FPU_regfile_cocotb_simple.py

This is the testbench used for verifying correctness of the simple FPU regfile implementation.

```
import random

import cocotb
from cocotb.decorators import coroutine
from cocotb.triggers import Timer, RisingEdge, FallingEdge,
 ↪  ReadOnly
#from cocotb.monitors import Monitor
#from cocotb.drivers import BitDriver
#from cocotb.binary import BinaryValue
from cocotb.regression import TestFactory
#from cocotb.scoreboard import Scoreboard
from cocotb.result import TestFailure, TestError, TestSuccess

# ==============================================================
@cocotb.coroutine
def reset_gen(reset_n, duration=10000):
    """Generator for resetting the register file at the
     ↪  beginning of the test"""
    reset_n <= 0
    yield Timer(duration)
    reset_n <= 1
    print("Reset done")
    reset_n._log.info("Reset done")

# ==============================================================
def seq_addr_gen():
    """Generator for generating 5-bit addresses for the
     ↪  register testing, wraps to 0"""
    addr = 0
    while True:
        yield addr
        addr = addr + 1
        if addr > 31:
            addr = 0

def rand_addr_gen():
    """Generator for generating random 5-bit addresses"""
    while True:
        yield random.randint(0,0x1F)

# ==============================================================
def data_gen():
    """Generator for making random input data, returns a random
    ↪  32-bit number"""
    while True:
        yield random.randint(0,0xFFFFFFFF)

# ==============================================================
@cocotb.coroutine
def clock_gen(signal, period=5000):
    """Generate the clock signal."""
    while True:
        signal <= 0
        yield Timer(period) # ps
        signal <= 1
        yield Timer(period) # ps

# ==============================================================
# NOTE: both these annotations can be used, however the
# @cocotb.coroutine allows for initializing the test with
# different parameters
#@cocotb.test()
@cocotb.coroutine
def run_test(dut):
    """Setup testbench and run a test.
    dut interface:
        input                   clk,
        input                  reset_n,
        input                  in_data_valid,
        input [4:0]              wb_port_addr,
        input [FLEN-1:0]      wb_port_data,
        input [4:0]              read_port_a_addr,
        output [FLEN-1:0]      read_port_a_data,
        input [4:0]              read_port_b_addr,
        output [FLEN-1:0]      read_port_b_data,
        input [4:0]              read_port_c_addr,
        output [FLEN-1:0]      read_port_c_data
    """
    cocotb.fork(clock_gen(dut.clk))
    yield reset_gen(dut.reset_n)
    clkedge = RisingEdge(dut.clk)
    # Data to be clocked on the next cycle is set on the
     ↪  falling edge to avoid delta-cycle issues in the TB
    negedge = FallingEdge(dut.clk)

    write_addr_gen = seq_addr_gen()
    write_data_gen = data_gen()
    read_addr_gen = rand_addr_gen()

    ref_data = [0] * 32
    yield clkedge;

    dut.in_data_valid <= 1
    # TODO: also run some tests with in_data_valid deasserted
     ↪  to make sure it doesn't catch data it isn't supposed
     ↪  to
    for i in range(32):
#        write_addr = write_addr_gen.next()
        write_data = write_data_gen.next()
#        dut.wb_port_addr <= write_addr
        dut.wb_port_addr <= i
        dut.read_port_a_addr <= i
        dut.wb_port_data <= write_data

        # Store data for comparison
        ref_data[i] = write_data
        if not dut.reset_n:
            ref_data[i] = 0

        dut.wb_port_addr._log.info("Write 0x{:08x} to reg.
         ↪  {:d}".format(int(write_data), i))
        # Wait for clock
        yield clkedge
        yield ReadOnly()

        read_data = int(dut.read_port_a_data)
        if read_data != ref_data[i]:
            raise TestError("Content of regfile address {}
            ↪  (0x{:08x}) differs from expected value:
            ↪  0x{:08x}".format(i, read_data, ref_data[i]))
            #print("Content of regfile address {} (0x{:08x})
            ↪  differs from expected value:
            ↪  0x{:08x}".format(i, read_data, ref_data[i]))

        # TODO: CoCoTB should be able to wait for delta cycles
        ↪  (propagation delay), this is a workaround in the
        ↪  meantime
        yield negedge;

    dut.in_data_valid <= 0

    # TODO: implement test for trying to write without
    ↪  in_data_valid asserted

    # Test random reads for 100 clock cycle.
    for i in range(100):
        read_addr_a = read_addr_gen.next()
        read_addr_b = read_addr_gen.next()
        read_addr_c = read_addr_gen.next()
        dut.read_port_a_addr <= read_addr_a
        dut.read_port_b_addr <= read_addr_b
        dut.read_port_c_addr <= read_addr_c

        # Wait for clock
        yield clkedge
        dut.read_port_a_data._log.info("Read data 0x{:08x} from
        ↪  reg. {:d}".format(int(dut.read_port_a_data),
        ↪  read_addr_a))
        dut.read_port_b_data._log.info("Read data 0x{:08x} from
        ↪  reg. {:d}".format(int(dut.read_port_b_data),
        ↪  read_addr_b))
        dut.read_port_b_data._log.info("Read data 0x{:08x} from
        ↪  reg. {:d}".format(int(dut.read_port_c_data),
        ↪  read_addr_c))

    dut._log.info("Testing reset clear")
    yield reset_gen(dut.reset_n);
    yield negedge;
    for i in range(32):
        dut.read_port_a_addr <= i
        yield clkedge
        read_data = int(dut.read_port_a_data)
        dut.read_port_a_data._log.info("Read data 0x{:08x} from
         ↪  reg. {:d}".format(int(dut.read_port_a_data), i))
        if read_data != 0:
            raise TestError("Content of regfile address {}
             ↪  (0x{:08x}) is not 0 after reset".format(i,
             ↪  read_data))

    yield clkedge

# NOTE: this is not needed when annotating the main test
 ↪  function as @cocotb.test() instead of @cocotb.coroutine
factory = TestFactory(run_test)
factory.generate_tests()
```

A.8 Modified FMAC pipeline

This is the modified FMAC pipeline that was implemented for this thesis.

```
// Copyright 2017, 2018 ETH Zurich and University of Bologna.
// Copyright and related rights are licensed under the
↪  Solderpad Hardware
// License, Version 0.51 (the “License”); you may not use this
↪  file except in
// compliance with the License.  You may obtain a copy of the
↪  License at
// http://solderpad.org/licenses/SHL-0.51. Unless required by
↪  applicable law
// or agreed to in writing, software, hardware and materials
↪  distributed under
// this License is distributed on an “AS IS” BASIS, WITHOUT
↪  WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See the
↪  License for the
// specific language governing permissions and limitations
↪  under the License.
/////////////////////////////////////////////////////////////////////
// Company:          IIS @ ETHZ - Federal Institute of Technology
↪  //
//
↪  //
// Engineers:        Lei Li -- lile@iis.ee.ethz.ch
↪  //
//
↪  //
// Additional contributions by:
↪  //
//                   Torbjørn Viem Ness, NTNU --
↪  torbjovn@stud.ntnu.no          //
//
↪  //
//
↪  //
// Create Date:      01/06/2017
↪  //
// Design Name:      fmac
↪  //
```

```
// Module Name:    fmac.sv
↪  //
// Project Name:   Private FPU
↪  //
// Language:       SystemVerilog
↪  //
//
↪  //
// Description:    The top module of fmac
↪  //
// function:
↪  Result_DO=Operand_a_DI+Operand_b_DI*Operand_c_DI
↪  //
//
↪  //
// Revision:       07/07/2017
↪  //
// Revision:       04/09/2017
↪  //
//                 No_one_S was added  by Lei Li
↪  //
// Revision:       03/04/2018
↪  //
//                 Fixed Torbjørn Viem Ness bugs  and Sticky
↪  bit                //
// Revision:       NEXT
↪  //
//                 Modified FMAC to allow multiplier bypass to
↪  cut latency    //
////////////////////////////////////////////////////////////////////////////////

import fpu_defs_fmac::*;

module fmac_customized
(
   //Inputs
   input logic [C_OP-1:0]   Operand_a_DI,
   input logic [C_OP-1:0]   Operand_b_DI,
   input logic [C_OP-1:0]   Operand_c_DI,
   input logic [C_RM-1:0]   RM_SI,    //Rounding Mode
   input logic              mul_bypass,
   //Output-result
   output logic [31:0]      Result_DO,
   //Output-Flags
   output logic             Exp_OF_SO,
   output logic             Exp_UF_SO,
   output logic             Flag_NX_SO,
   output logic             Flag_IV_SO
 );

  logic [C_MANT-1:0]         Mant_res_DO;
  logic [C_EXP-1:0]          Exp_res_DO;
  logic                      Sign_res_DO;
  logic                      DeN_a_S, Sub_S, Sign_postalig_D,
   ↪  Sign_amt_D, Sft_stop_S, Sign_out_D;


  assign Result_DO =   {Sign_res_DO,Exp_res_DO, Mant_res_DO};

   logic                    Sign_a_D;
   logic                    Sign_b_D;
   logic                    Sign_c_D;
   logic [C_EXP-1:0]        Exp_a_D;
   logic [C_EXP-1:0]        Exp_b_D;
   logic [C_EXP-1:0]        Exp_c_D;
   logic [C_MANT:0]         Mant_a_D;
   logic [C_MANT:0]         Mant_b_D;
   logic [C_MANT:0]         Mant_c_D;
   logic                    Inf_a_S;
   logic                    Inf_b_S;
   logic                    Inf_c_S;
   logic                    NaN_a_S;
   logic                    NaN_b_S;
   logic                    NaN_c_S;

preprocess_fmac  precess_U0
 (
   .Operand_a_DI          (Operand_a_DI       ),
   .Operand_b_DI          (Operand_b_DI       ),
   .Operand_c_DI          (Operand_c_DI       ),
   .Exp_a_DO              (Exp_a_D            ),
   .Mant_a_DO             (Mant_a_D           ),
   .Sign_a_DO             (Sign_a_D           ),
   .Exp_b_DO              (Exp_b_D            ),
   .Mant_b_DO             (Mant_b_D           ),
   .Sign_b_DO             (Sign_b_D           ),
   .Exp_c_DO              (Exp_c_D            ),
   .Mant_c_DO             (Mant_c_D           ),
   .Sign_c_DO             (Sign_c_D           ),
   .DeN_a_SO              (DeN_a_S            ),
   .Inf_a_SO              (Inf_a_S            ),
   .Inf_b_SO              (Inf_b_S            ),
   .Inf_c_SO              (Inf_c_S            ),
   .Zero_a_SO             (Zero_a_S           ),
   .Zero_b_SO             (Zero_b_S           ),
   .Zero_c_SO             (Zero_c_S           ),
   .NaN_a_SO              (NaN_a_S            ),
   .NaN_b_SO              (NaN_b_S            ),
   .NaN_c_SO              (NaN_c_S            )
   );

// Generate partial products for multiplication
 logic [12:0] [2*C_MANT+2:0]                       Pp_index_D;
 pp_generation  pp_gneration_U0
 (
   .Mant_a_DI             (Mant_b_D          ),
   .Mant_b_DI             (Mant_c_D          ),
   .Pp_index_DO           (Pp_index_D        )
 );

   logic [2*C_MANT+2:0]                            Pp_sum_D;
   logic [2*C_MANT+2:0]                            Pp_carry_D;
   logic                                           MSB_cor_D;
 wallace   wallace_U0
 (
   .Pp_index_DI           (Pp_index_D        ),
   .Pp_sum_DO             (Pp_sum_D          ),
   .Pp_carry_DO           (Pp_carry_D        ),
   .MSB_cor_DO            (MSB_cor_D         )
 );
// "Multiplication section" ends here?

// Operand alignment for the mantissa of the addend
```

```
logic [74:0]                            Mant_postalig_a_D;
logic signed [C_EXP+1:0]                Exp_postalig_D;
logic [2*C_MANT+2:0]                    Pp_sum_postcal_D;
logic [2*C_MANT+2:0]                    Pp_carry_postcal_D;
logic [C_EXP+1:0]                       Minus_sft_amt_D;
logic
 ↪  Mant_sticky_sft_out_S;
logic                                   Sign_change_S;
aligner  aligner_U0
(
  .Exp_a_DI               (Exp_a_D          ),
  .Exp_b_DI               (Exp_b_D          ),
  .Mant_a_DI              (Mant_a_D         ),
  .Exp_c_DI               (Exp_c_D          ),

  .Sign_a_DI              (Sign_a_D         ),
  .Sign_b_DI              (Sign_b_D         ),
  .Sign_c_DI              (Sign_c_D         ),
  .Pp_sum_DI              (Pp_sum_D         ),
  .Pp_carry_DI            (Pp_carry_D       ),
  .Sign_change_SI         (Sign_change_S    ),
  .Sub_SO                 (Sub_S            ),
  .Mant_postalig_a_DO     (Mant_postalig_a_D),
  .Exp_postalig_DO        (Exp_postalig_D   ),
  .Sign_postalig_DO       (Sign_postalig_D  ),
  .Sign_amt_DO            (Sign_amt_D       ),
  .Sft_stop_SO            (Sft_stop_S       ),
  .Pp_sum_postcal_DO      (Pp_sum_postcal_D ),
  .Pp_carry_postcal_DO    (Pp_carry_postcal_D),
  .Minus_sft_amt_DO       (Minus_sft_amt_D   ),
  .Mant_sticky_sft_out_SO (Mant_sticky_sft_out_S)
);

// 48-bit CSA. In fact the bit width is 49 bits including sign
↪  bit. After this CSA, EDCA is used to produce the correct
↪  sum and the carry out bit.
  logic [2*C_MANT+1:0]                   Csa_sum_D;
  logic [2*C_MANT+1:0]                   Csa_carry_D;
  // EXPERIMENTAL: multiplier bypass
  logic [2*C_MANT+1:0]                   Csa_in_B;
  logic [2*C_MANT+1:0]                   Csa_in_C;
```

```
   assign Csa_in_B = (mul_bypass ? {'0} :
   ↪  {Pp_sum_postcal_D[2*C_MANT+1:0]});
   assign Csa_in_C = (mul_bypass ? {Mant_c_D, {(C_MANT){1'b0}}}
   ↪  : {Pp_carry_postcal_D[2*C_MANT:0],1'b0});
   // end of EXPERIMENTAL
CSA   #(2*C_MANT+2)  CSA_U0
 (
   .A_DI                  (Mant_postalig_a_D[2*C_MANT+1:0]),
   .B_DI                  (Csa_in_B  ), //
   ↪  ({Pp_sum_postcal_D[2*C_MANT+1:0]}),
   .C_DI                  (Csa_in_C  ), //
   ↪  ({Pp_carry_postcal_D[2*C_MANT:0],1'b0}),
   .Sum_DO                (Csa_sum_D                       ),
   .Carry_DO              (Csa_carry_D                     )
 );

// The correction based sign extension is included in adders.
 logic [73:0]                                  Sum_pos_D;
 logic [3*C_MANT+4:0]                          A_LZA_D;
 logic [3*C_MANT+4:0]                          B_LZA_D;
 logic                                         Minus_sticky_bit_S;
adders adders_U0
 (
  .AL_DI                  (Csa_sum_D        ),
  .BL_DI                  (Csa_carry_D      ),
  .Sub_SI                 (Sub_S            ),
  .Sign_cor_SI            ({MSB_cor_D,
  ↪  Pp_carry_postcal_D[2*C_MANT+2],{Pp_sum_postcal_D[2*C_MANT+2]
  ↪  && Pp_carry_postcal_D[2*C_MANT+1]}}),
  .Sign_amt_DI            (Sign_amt_D       ),
  .Sft_stop_SI            (Sft_stop_S       ),
  .BH_DI
  ↪  (Mant_postalig_a_D[3*C_MANT+5:2*C_MANT+2]),
  .Sign_postalig_DI       (Sign_postalig_D  ),
  .Inf_b_SI               (Inf_b_S          ),
  .Inf_c_SI               (Inf_c_S          ),
  .Zero_b_SI              (Zero_b_S         ),
  .Zero_c_SI              (Zero_c_S         ),
  .NaN_b_SI               (NaN_b_S          ),
  .NaN_c_SI               (NaN_c_S          ),
  .Minus_sft_amt_DI         (Minus_sft_amt_D  ),
  .Sum_pos_DO               (Sum_pos_D        ),
  .Sign_out_DO              (Sign_out_D       ),
  .A_LZA_DO                 (A_LZA_D          ),
  .B_LZA_DO                 (B_LZA_D          ),
  .Minus_sticky_bit_SO      (Minus_sticky_bit_S),
  .Sign_change_SO           (Sign_change_S    )
 );


 logic [C_LEADONE_WIDTH-1:0]                Leading_one_D;
 logic                                      No_one_S;

LZA #(3*C_MANT+5) LZA_U0
  (
   .A_DI                    (A_LZA_D       ),
   .B_DI                    (B_LZA_D       ),
   .Leading_one_DO          (Leading_one_D ),
   .No_one_SO               (No_one_S      )
   );


 fpu_norm_fmac  fpu_norm_U0
  (
   .Mant_in_DI             (Sum_pos_D         ),
   .Exp_in_DI              (Exp_postalig_D    ),
   .Sign_in_DI             (Sign_out_D        ),
   .Leading_one_DI         (Leading_one_D     ),
   .No_one_SI              (No_one_S          ),
   .Sign_amt_DI            (Sign_amt_D        ),
   .Sub_SI                 (Sub_S             ),
   .Exp_a_DI               (Operand_a_DI[C_OP-2:C_MANT]),
   ↪  //exponent
   .Mant_a_DI              (Mant_a_D          ),
   .Sign_a_DI              (Sign_a_D          ),
   .DeN_a_SI               (DeN_a_S           ),
   .RM_SI                  (RM_SI             ),     //Rounding
   ↪  Mode
   .Inf_a_SI               (Inf_a_S           ),
   .Inf_b_SI               (Inf_b_S           ),
   .Inf_c_SI               (Inf_c_S           ),
   .Zero_a_SI             (Zero_a_S           ),
   .Zero_b_SI             (Zero_b_S           ),
   .Zero_c_SI             (Zero_c_S           ),
   .NaN_a_SI              (NaN_a_S            ),
   .NaN_b_SI              (NaN_b_S            ),
   .NaN_c_SI              (NaN_c_S            ),
   .Mant_sticky_sft_out_SI (Mant_sticky_sft_out_S),
   .Minus_sticky_bit_SI   (Minus_sticky_bit_S ),
   .Mant_res_DO           (Mant_res_DO        ),
   .Exp_res_DO            (Exp_res_DO         ),
   .Sign_res_DO           (Sign_res_DO        ),
   .Exp_OF_SO             (Exp_OF_SO          ),
   .Exp_UF_SO             (Exp_UF_SO          ),
   .Flag_Inexact_SO       (Flag_NX_SO         ),
   .Flag_Invalid_SO       (Flag_IV_SO         )
   );

endmodule
```

A.9 Modified FMA wrapper

This is the modified FMAC wrapper that was implemented to support the FMAC listed in Appendix A.8.

```
// Copyright 2017, 2018 ETH Zurich and University of Bologna.
// Copyright and related rights are licensed under the
↪  Solderpad Hardware
// License, Version 0.51 (the "License"); you may not use this
↪  file except in
// compliance with the License.  You may obtain a copy of the
↪  License at
// http://solderpad.org/licenses/SHL-0.51. Unless required by
↪  applicable law
// or agreed to in writing, software, hardware and materials
↪  distributed under
// this License is distributed on an "AS IS" BASIS, WITHOUT
↪  WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See the
↪  License for the
// specific language governing permissions and limitations
↪  under the License.
/////////////////////////////////////////////////////////////////
// Copyright (C) 2017 ETH Zurich, University of Bologna
↪  //
// All rights reserved.
↪  //
//
↪  //
//
↪  //
// Engineer:       Michael Gautschi - gautschi@iis.ee.ethz.ch
↪  //
// Create Date:    20/06/2017
↪  //
// Design Name:    fp_mac_wrapper
↪  //
// Module Name:    fpu_fma_wrapper.sv
↪  //
// Project Name:   Shared APU
↪  //
```

```
// Language:         SystemVerilog
↪  //
//
↪  //
// Description:      Wraps the fp-mac unit
↪  //
//
↪  //
//
↪  //
// Revision:
↪  //
////////////////////////////////////////////////////////////////////////////////

`ifndef SYNTHESIS
//`define FP_SIM_MODELS;
`endif

module fma_wrapper_custom
#(
  parameter C_MAC_PIPE_REGS = 2,
  parameter RND_WIDTH       = 2,
  parameter STAT_WIDTH      = 5
)
 (
  // Clock and Reset
  input  logic                  clk_i,
  input  logic                  rst_ni,

  input  logic                  En_i,
  input  logic                  mul_bypass,

  input logic [31:0]            OpA_i,
  input logic [31:0]            OpB_i,
  input logic [31:0]            OpC_i,
  input logic [1:0]             Op_i,

  input logic [RND_WIDTH-1:0]   Rnd_i,
  output logic [STAT_WIDTH-1:0] Status_o,

  output logic [31:0]           Res_o,
  output logic                  Valid_o,
  output logic                  Ready_o,
  input  logic                  Ack_i
);

   // DISTRIBUTE PIPE REGS
   parameter C_PRE_PIPE_REGS   = C_MAC_PIPE_REGS - 1;
   parameter C_POST_PIPE_REGS  = 1;

   // PRE PIPE REG SIGNALS
   logic [31:0]          OpA_DP     [C_PRE_PIPE_REGS+1];
   logic [31:0]          OpB_DP     [C_PRE_PIPE_REGS+1];
   logic [31:0]          OpC_DP     [C_PRE_PIPE_REGS+1];
   logic                 En_SP      [C_PRE_PIPE_REGS+1];
   logic                 mul_bypass_SP  [C_PRE_PIPE_REGS+1];
   logic [RND_WIDTH-1:0] Rnd_DP     [C_PRE_PIPE_REGS+1];

   // POST PIPE REG SIGNALS
   logic                  EnPost_SP      [C_POST_PIPE_REGS+1];
   logic [31:0]           Res_DP         [C_POST_PIPE_REGS+1];
   logic [STAT_WIDTH-1:0] Status_DP      [C_POST_PIPE_REGS+1];
   // Fflags of DW        {PA/DV, HugeInt, NX, Huge, Tiny, IV,
   ↪  Inf, Zero}
   logic [7:0]            status;

   // assign input. note: index [0] is not a register here!
   assign OpA_DP[0]     = En_i ? OpA_i :'0;
   assign OpB_DP[0]     = En_i ? {OpB_i[31] ^
   ↪  Op_i[1],OpB_i[30:0]} :'0;
   assign OpC_DP[0]     = En_i ? {OpC_i[31] ^
   ↪  Op_i[0],OpC_i[30:0]} :'0;
   assign En_SP[0]      = En_i;
   assign mul_bypass_SP[0] = mul_bypass;
   assign Rnd_DP[0]     = Rnd_i;

   // propagate states
   assign EnPost_SP[0]       = En_SP[C_PRE_PIPE_REGS];

   // assign output
   assign Res_o              = Res_DP[C_POST_PIPE_REGS];
   assign Valid_o            = EnPost_SP[C_POST_PIPE_REGS];
   assign Status_o          = Status_DP[C_POST_PIPE_REGS];
   assign Ready_o           = 1'b1;

`ifndef VERILATOR
`ifdef FP_SIM_MODELS
   shortreal              a, b, c, res;

   assign a = $bitstoshortreal(OpA_DP[C_PRE_PIPE_REGS]);
   assign b = $bitstoshortreal(OpB_DP[C_PRE_PIPE_REGS]);
   assign c = $bitstoshortreal(OpC_DP[C_PRE_PIPE_REGS]);

   // rounding mode is ignored here
   assign res = (a*b) + c;

   // convert to logic again
   assign Res_DP[0] = $shortrealtobits(res);

   // not used in simulation model
   assign Status_DP[0] = '0;
`else
   //Fflags of RISC-V    {NV,        DZ,    OF,         UF,
   ↪  NX        }
   assign Status_DP[0] = {status[2], 1'b0, status[4],
   ↪  status[3], status[5]};

fmac_customized
fp_fma_i
  //fmac assumes a + b*c, the fpu core assumes a*b + c
  (
   .Operand_a_DI          ( OpC_DP[C_PRE_PIPE_REGS]        ),
   .Operand_b_DI          ( OpB_DP[C_PRE_PIPE_REGS]        ),
   .Operand_c_DI          ( OpA_DP[C_PRE_PIPE_REGS]        ),
   .RM_SI                 ( Rnd_DP[C_PRE_PIPE_REGS]        ),
//   .mul_bypass            ( mul_bypass                       ),
↪  // TODO: support pipelining the bypass signal as well
   .mul_bypass            ( mul_bypass_SP[C_PRE_PIPE_REGS] ),
   .Result_DO             ( Res_DP[0]                      ),
   .Exp_OF_SO             ( status[4]                      ),
   .Exp_UF_SO             ( status[3]                      ),
   .Flag_NX_SO            ( status[5]                      ),
   .Flag_IV_SO            ( status[2]                      )
   );

`endif
`endif
   // PRE_PIPE_REGS
   generate
    genvar i;
      for (i=1; i <= C_PRE_PIPE_REGS; i++)  begin: g_pre_regs

         always_ff @(posedge clk_i or negedge rst_ni) begin :
          ↪  p_pre_regs
            if(~rst_ni) begin
               En_SP[i]          <= '0;
               mul_bypass_SP[i]  <= '0;
               OpA_DP[i]         <= '0;
               OpB_DP[i]         <= '0;
               OpC_DP[i]         <= '0;
               Rnd_DP[i]         <= '0;
            end
            else begin
               // this one has to be always enabled...
               En_SP[i]        <= En_SP[i-1];
               mul_bypass_SP[i] <= mul_bypass_SP[i-1]; // TODO:
                ↪  stall pipeline if bypass is switched from
                ↪  off to on?
               // enabled regs
               if(En_SP[i-1]) begin
                  OpA_DP[i]       <= OpA_DP[i-1];
                  OpB_DP[i]       <= OpB_DP[i-1];
                  OpC_DP[i]       <= OpC_DP[i-1];
                  Rnd_DP[i]       <= Rnd_DP[i-1];
               end
            end
         end
      end
   endgenerate


   // POST_PIPE_REGS
   generate
    genvar j;
    for (j=1; j <= C_POST_PIPE_REGS; j++)  begin: g_post_regs

      always_ff @(posedge clk_i or negedge rst_ni) begin :
       ↪  p_post_regs
        if(~rst_ni) begin
          EnPost_SP[j]      <= '0;
          Res_DP[j]         <= '0;
          Status_DP[j]      <= '0;
        end
        else begin
          // this one has to be always enabled...
          EnPost_SP[j]        <= EnPost_SP[j-1];

          // enabled regs
          if(EnPost_SP[j-1]) begin
            Res_DP[j]         <= Res_DP[j-1];
            Status_DP[j]      <= Status_DP[j-1];
          end
        end
      end
    end
  endgenerate

endmodule
```

A.10 Simplified normalization and rounding unit

This is a simplified version of the fpu normalizer core from PULP, stripped down to only normalize results of float-to-integer conversions.

```
// Copyright 2017, 2018 ETH Zurich and University of Bologna.
// Copyright and related rights are licensed under the
↪  Solderpad Hardware
// License, Version 0.51 (the "License"); you may not use this
↪  file except in
// compliance with the License.  You may obtain a copy of the
↪  License at
// http://solderpad.org/licenses/SHL-0.51. Unless required by
↪  applicable law
// or agreed to in writing, software, hardware and materials
↪  distributed under
// this License is distributed on an "AS IS" BASIS, WITHOUT
↪  WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See the
↪  License for the
// specific language governing permissions and limitations
↪  under the License.
////////////////////////////////////////////////////////////////////////////////
// Company:          IIS @ ETHZ - Federal Institute of Technology
↪  //
//
↪  //
// Engineers:        Lukas Mueller -- lukasmue@student.ethz.ch
↪  //
//
↪  //
// Additional contributions by:
↪  //
//                   Torbjørn Viem Ness -- torbjovn@stud.ntnu.no
↪  //
//
↪  //
//
↪  //
```

```
// Create Date:    06/10/2014
↪  //
// Design Name:    FPU
↪  //
// Module Name:    fpu_norm.sv
↪  //
// Project Name:   Private FPU
↪  //
// Language:       SystemVerilog
↪  //
//
↪  //
// Description:    Floating point Normalizer/Rounding unit
↪  //
//
↪  //
//
↪  //
//
↪  //
// Revision:
↪  //
//                15/05/2018
↪  //
//                Pass package parameters as default args
↪  instead of using    //
//                them directly, improves compatibility with
↪  tools like       //
//                Synopsys Spyglass and DC (GitHub #7) -
↪  Torbjørn Viem Ness   //
//
↪  //
////////////////////////////////////////////////////////////////////////////////

import fpu_defs::*;

module fpu_itof_norm
#(
   parameter C_MANT_PRENORM     = fpu_defs::C_MANT_PRENORM,
   parameter C_EXP_PRENORM      = fpu_defs::C_EXP_PRENORM,
   parameter C_MANT_PRENORM_IND = fpu_defs::C_MANT_PRENORM_IND,
   parameter C_EXP_ZERO         = fpu_defs::C_EXP_ZERO,
   parameter C_EXP_INF          = fpu_defs::C_EXP_INF,

   parameter C_RM               = fpu_defs::C_RM,
   parameter C_CMD              = fpu_defs::C_CMD,
   parameter C_MANT             = fpu_defs::C_MANT,
   parameter C_EXP              = fpu_defs::C_EXP,

   parameter C_RM_NEAREST       = fpu_defs::C_RM_NEAREST,
   parameter C_RM_TRUNC         = fpu_defs::C_RM_TRUNC,
   parameter C_RM_PLUSINF       = fpu_defs::C_RM_PLUSINF,
   parameter C_RM_MINUSINF      = fpu_defs::C_RM_MINUSINF
)
  (
   //Input Operands
   input logic        [C_MANT_PRENORM-1:0] Mant_in_DI,
   input logic signed [C_EXP_PRENORM-1:0]  Exp_in_DI,
   input logic                             Sign_in_DI,

   //Rounding Mode
   input logic [C_RM-1:0]                  RM_SI,
   input logic [C_CMD-1:0]                 OP_SI,

   output logic [C_MANT:0]                 Mant_res_DO,
   output logic [C_EXP-1:0]                Exp_res_DO,

   output logic                            Rounded_SO,
   output logic                            Exp_OF_SO,
   output logic                            Exp_UF_SO
   );

    ↪ ///////////////////////////////////////////////////////////////////
   // Normalization
    ↪ //

    ↪ ///////////////////////////////////////////////////////////////////

   logic [C_MANT_PRENORM_IND-1:0]        Mant_leadingOne_D;
   logic                                 Mant_zero_S;
   logic [C_MANT+4:0]                    Mant_norm_D;
logic signed [C_EXP_PRENORM-1:0]       Exp_norm_D;

//trying out stuff for denormals
logic signed [C_EXP_PRENORM-1:0]       Mant_shAmt_D;
logic signed [C_EXP_PRENORM:0]         Mant_shAmt2_D;

logic [C_EXP-1:0]                      Exp_final_D;
logic signed [C_EXP_PRENORM-1:0]       Exp_rounded_D;

//sticky bit
logic                                  Mant_sticky_D;

logic                                  Denormal_S;
logic                                  Mant_renorm_S;

//Detect leading one
fpu_ff
#(
  .LEN(C_MANT_PRENORM))
LOD
(
  .in_i        ( Mant_in_DI         ),
  .first_one_o ( Mant_leadingOne_D ),
  .no_ones_o   ( Mant_zero_S        )
);


logic                                 Denormals_shift_add_D;
assign Denormals_shift_add_D = ~Mant_zero_S & (Exp_in_DI ==
↪  C_EXP_ZERO);

assign Denormal_S    =
↪  ((C_EXP_PRENORM)'(signed'(Mant_leadingOne_D)) >=
↪  Exp_in_DI) || Mant_zero_S;
assign Mant_shAmt_D  = Denormal_S ? Exp_in_DI +
↪  Denormals_shift_add_D : Mant_leadingOne_D;
assign Mant_shAmt2_D = {Mant_shAmt_D[$high(Mant_shAmt_D)],
↪  Mant_shAmt_D} + (C_MANT+4+1);

//Shift mantissa
always_comb
```

```
    begin
       logic [C_MANT_PRENORM+C_MANT+4:0] temp;
       temp = ((C_MANT_PRENORM+C_MANT+4+1)'(Mant_in_DI) <<
        ↪  (Mant_shAmt2_D) );
       Mant_norm_D =
        ↪  temp[C_MANT_PRENORM+C_MANT+4:C_MANT_PRENORM];
    end

  always_comb
    begin
       Mant_sticky_D = 1'b0;
       if (Mant_shAmt2_D <= 0)
         Mant_sticky_D = | Mant_in_DI;
       else if (Mant_shAmt2_D <= C_MANT_PRENORM)
         Mant_sticky_D = | (Mant_in_DI << (Mant_shAmt2_D));
    end

  //adjust exponent
  assign Exp_norm_D = Exp_in_DI -
   ↪  (C_EXP_PRENORM)'(signed'(Mant_leadingOne_D)) + 1;
  //Explanation of the +1 since I'll probably forget:
  //we get numbers in the format xx.x...
  //but to make things easier we interpret them as
  //x.xx... and adjust the exponent accordingly

  assign Exp_rounded_D = Exp_norm_D + Mant_renorm_S;
  assign Exp_final_D   = Exp_rounded_D[C_EXP-1:0];


  always_comb //detect exponent over/underflow
    begin
       Exp_OF_SO = 1'b0;
       Exp_UF_SO = 1'b0;
       if (Exp_rounded_D >= signed'({2'b0,C_EXP_INF}))
        ↪  //overflow
         begin
            Exp_OF_SO = 1'b1;
         end
       else if (Exp_rounded_D <= signed'({2'b0,C_EXP_ZERO}))
        ↪  //underflow
         begin
```

```
          Exp_UF_SO = 1'b1;
        end
    end


  ↪  ///////////////////////////////////////////////////////////////////////////
  // Rounding
  ↪  //

  ↪  ///////////////////////////////////////////////////////////////////////////

  logic [C_MANT:0]   Mant_upper_D;
  logic [3:0]        Mant_lower_D;
  logic [C_MANT+1:0] Mant_upperRounded_D;

  logic              Mant_roundUp_S;
  logic              Mant_rounded_S;

  assign Mant_lower_D = Mant_norm_D[3:0];
  assign Mant_upper_D = Mant_norm_D[C_MANT+4:4];


  assign Mant_rounded_S = (|(Mant_lower_D)) | Mant_sticky_D;

  always_comb //determine whether to round up or not
    begin
       Mant_roundUp_S = 1'b0;
       case (RM_SI)
         C_RM_NEAREST :
           Mant_roundUp_S = Mant_lower_D[3] && (((|
           ↪  Mant_lower_D[2:0]) | Mant_sticky_D) ||
           ↪  Mant_upper_D[0]);
         C_RM_TRUNC   :
           Mant_roundUp_S = 0;
         C_RM_PLUSINF :
           Mant_roundUp_S = Mant_rounded_S & ~Sign_in_DI;
         C_RM_MINUSINF:
           Mant_roundUp_S = Mant_rounded_S & Sign_in_DI;
         default      :
           Mant_roundUp_S = 0;
       endcase // case (RM_DI)
```

```
     end // always_comb begin

   assign Mant_upperRounded_D = Mant_upper_D + Mant_roundUp_S;
   assign Mant_renorm_S       = Mant_upperRounded_D[C_MANT+1];


   ↪  ///////////////////////////////////////////////////////////////////////////
   // Output Assignments
   ↪  //

   ↪  ///////////////////////////////////////////////////////////////////////////

   assign Mant_res_DO = Mant_upperRounded_D >> (Mant_renorm_S &
   ↪  ~Denormal_S);
   assign Exp_res_DO  = Exp_final_D;
   assign Rounded_SO  = Mant_rounded_S;

endmodule // fpu_norm
```

A.11 Stripped conversion core

This is the minimal FPU core used to implement float-to-integer and integer-to-float conversions.

```
/**
 * TODO: license/copyright header
 *
 * Module description: Stripped down FPU core for conversion
↪  between IEEE-754
 *                     binary floating-point numbers and
↪  integers.
 *
 * Author:             Torbjørn Viem Ness, NTNU --
↪  torbjovn@stud.ntnu.no
 * Date:               2018-04-23
 *
 * Acknowledgments:    This module uses modules from the PULP
↪  FPU project
 *                     (https://github.com/pulp-platform/fpu)
 */

// NOTE: this can be used to disable latching of inputs in
↪  order to compare power consumption
`ifndef NO_LATCH_INPUTS
`define LATCH_INPUTS
`endif

module conversion_core #( parameter FLEN = 32 ) // TODO: use
↪  the FLEN parameter to support double precision in the
↪  future
        (
                // Clock and reset
                input clk,
                input reset_n,
                input enable_in,

                // Operands
                input [FLEN-1:0] data_in,
                input [2:0]      rm_in,
                input [1:0]      operation_in, // Mirrors the
                ↪  RVF encoding - one bit for direction (F2I
                ↪  vs I2F) and one for un-/signed

                output [FLEN-1:0] result,
                output            result_valid,

                output [4:0]      flags
        );

        parameter C_MANT_WIDTH = 23;
        parameter C_EXP_WIDTH = 8;
        parameter C_EXP_PRENORM = C_EXP_WIDTH + 2;
        parameter C_MANT_PRENORM = C_MANT_WIDTH*2+2;

    reg enable;
    reg [FLEN-1:0] operand;
    reg [2:0] rm;
    reg [1:0] operation;

    wire unsigned_c = operation[1];
    wire op_fromint = operation[0];

    always @(posedge clk, negedge reset_n) begin
                if (~reset_n) begin
                        enable = 'b0;
                        operand = 'b0;
                        rm = 'b0;
                        operation = 'b0;
                end
                else begin
                        enable = enable_in;
                        if (enable) begin
                                operand = data_in;
                                rm = rm_in;
                                operation = operation_in;
                        end
                end
        end
    end
    // Early detection whether the number is zero, in which
     ↪  case the result is also zero
    // TODO: do we need to do this at the rising clock in order
     ↪  to avoid glitches before setting the number_is_zero
     ↪  signal?
    reg number_is_zero;
    always @(operand) begin
            // Ignore the sign bit here if the input is a
             ↪  floating-point number (because -0 also exists)
            if (op_fromint)
                    number_is_zero = ~(|operand[FLEN-1:0]);
            else
                    number_is_zero = ~(|{1'b0,
                     ↪  operand[FLEN-2:0]});
        end

        ////////////////////////////////////////////////////////////////////////
   // Integer to floating point conversion

    ↪  ////////////////////////////////////////////////////////////////////////
   wire                            Sign_prenorm_itof_D;
   reg signed [C_EXP_PRENORM-1:0] Exp_prenorm_itof_D;
   wire [C_MANT_PRENORM-1:0]       Mant_prenorm_itof_D;
   wire                            EnableITOF_S;
   wire [FLEN-1:0]                 res_itof_D;


`ifdef LATCH_INPUTS
        reg
         ↪  [FLEN-1:0]                                      operand_itof_D;

        assign EnableITOF_S = enable & op_fromint &
         ↪  (~number_is_zero);
        // Latch input to prevent unnecessary toggles
        always @(EnableITOF_S, operand, reset_n) begin
                if (~reset_n)
                        operand_itof_D = 'b0;
                else if (EnableITOF_S)
                        operand_itof_D = operand;
        end
`else
```

```
        wire
         ↪   [FLEN-1:0]                                    operand_itof_D;

        assign EnableITOF_S = enable & op_fromint;
        assign operand_itof_D = (EnableITOF_S ? operand : 'b0);
`endif

   fpu_itof int2fp
     (
      .Operand_a_DI    ( operand_itof_D      ),

      .Sign_prenorm_DO ( Sign_prenorm_itof_D ),
      .Exp_prenorm_DO  ( Exp_prenorm_itof_D  ),
      .Mant_prenorm_DO ( Mant_prenorm_itof_D )
      );


   wire [C_MANT_WIDTH:0] Mant_norm_D;
   wire [C_EXP_WIDTH-1:0]  Exp_norm_D;
   wire Exp_OF_S, Exp_UF_S;

   fpu_norm normalizer // TODO: implement a simplified
    ↪   normalizer that only handles the cases required for i2f
    ↪   conversion?
     (
      .Mant_in_DI   ( Mant_prenorm_itof_D ),
      .Exp_in_DI    ( Exp_prenorm_itof_D  ),
      .Sign_in_DI   ( Sign_prenorm_itof_D    ),

      .RM_SI        ( rm            ),
      .OP_SI        ( 4'h4            ), // = C_FPU_I2F_CMD

      .Mant_res_DO ( Mant_norm_D    ),
      .Exp_res_DO   ( Exp_norm_D     ),

      .Rounded_SO   ( Mant_rounded_S ),
      .Exp_OF_SO    ( Exp_OF_S       ),
      .Exp_UF_SO    ( Exp_UF_S       )
      );

   wire [C_MANT_WIDTH-1:0] Mant_res;
```

```
          // NOTE: overflow cannot possibly occur with 32-bit
           ↪  integer to single-precision float conversion,
          // as the max value (~3.4*10^34) is way beyond what is
           ↪  possible to represent with 32 bits integers
          // TODO: revisit this if half-precision becomes
           ↪  relevant to implement, as the max value is only
           ↪  65504.
//          reg [C_EXP_WIDTH-1:0] Exp_res;
//          always begin
//                  Exp_res = Exp_norm_D;
//                  if (Exp_toZero_S)
//                          Exp_res = {C_EXP_WIDTH{1'b0}};
//          end

   assign Mant_res = Mant_norm_D[C_MANT_WIDTH-1:0];
   assign res_itof_D = {(unsigned_c ? 1'b0 :
    ↪  Sign_prenorm_itof_D), Exp_norm_D, Mant_res};


    ↪  ////////////////////////////////////////////////////////////////////////
   // Floating point to integer conversion

    ↪  ////////////////////////////////////////////////////////////////////////
   wire [FLEN-1:0]    Result_ftoi_D;
   wire               UF_ftoi_S;
   wire               OF_ftoi_S;
   wire               Zero_ftoi_S;
   wire               IX_ftoi_S;
   wire               IV_ftoi_S;
   wire               Inf_ftoi_S;
   wire               EnableFTOI_S;

`ifdef LATCH_INPUTS
   reg                    input_sign;
   reg [C_EXP_WIDTH-1:0]  input_exp;
   reg [C_MANT_WIDTH:0] input_mant;
   reg hidden_bit;

        assign EnableFTOI_S = enable & (~op_fromint) &
         ↪  (~number_is_zero);
    // Latch input data here as well
    always @(EnableFTOI_S, operand, reset_n) begin
            if (~reset_n) begin
                    input_exp = 0;
                    input_mant = 0;
                    hidden_bit = 0;
                    input_sign = 0;
            end
            else if (EnableFTOI_S) begin
                    input_exp =
                    ↪   operand[C_MANT_WIDTH+C_EXP_WIDTH-1:C_MANT_WIDTH];
                    input_sign = (unsigned_c ? 1'b0 :
                    ↪   operand[FLEN-1]);
                    hidden_bit = | input_exp;
                    input_mant = {hidden_bit,
                    ↪   operand[C_MANT_WIDTH-1:0]};
            end
    end
`else
   wire                   input_sign;
   wire [C_EXP_WIDTH-1:0]  input_exp;
   wire [C_MANT_WIDTH:0] input_mant;
   wire hidden_bit;

        assign EnableFTOI_S = enable & (~op_fromint);
        assign input_sign = (unsigned_c ? 1'b0 :
        ↪   operand[FLEN-1]);
        assign input_exp =
        ↪   operand[C_MANT_WIDTH+C_EXP_WIDTH-1:C_MANT_WIDTH];
        assign hidden_bit = | input_exp;
        assign input_mant = {hidden_bit,
        ↪   operand[C_MANT_WIDTH-1:0]};
`endif

   fpu_ftoi fp2int
     (
      .Sign_a_DI ( input_sign ),
      .Exp_a_DI  ( input_exp  ),
      .Mant_a_DI ( input_mant ),

      .Result_DO ( Result_ftoi_D ),
        .UF_SO      ( UF_ftoi_S     ),
        .OF_SO      ( OF_ftoi_S     ),
        .Zero_SO    ( Zero_ftoi_S   ),
        .IX_SO      ( IX_ftoi_S     ),
        .IV_SO      ( IV_ftoi_S     ),
        .Inf_SO     ( Inf_ftoi_S    )
        );

    assign result = (number_is_zero ? {FLEN{1'b0}} :
    ↪   (op_fromint ? res_itof_D : Result_ftoi_D));
    assign flags  = (op_fromint ? {IV_ftoi_S, 1'b0, OF_ftoi_S,
    ↪   UF_ftoi_S, IX_ftoi_S} : {2'b0, Exp_OF_S, Exp_UF_S,
    ↪   1'b0});
    assign result_valid = enable;

endmodule
```

A.12 Modified top-level FPU

This is the modified FPU top level that implements the changes outlined in this thesis.

```
// TODO: add header with info and copylefts etc..

// TODO: import operation defs from instruction decoder?

`define FPU_OP_ADD_CMD    4'h0
`define FPU_OP_SUB_CMD    4'h1
`define FPU_OP_MUL_CMD    4'h2
`define FPU_OP_DIV_CMD    4'h3
`define FPU_OP_I2F_CMD    4'h4
`define FPU_OP_F2I_CMD    4'h5
`define FPU_OP_SQRT_CMD   4'h6
`define FPU_OP_NOP_CMD    4'h7
`define FPU_OP_FMADD_CMD  4'h8
`define FPU_OP_FMSUB_CMD  4'h9
`define FPU_OP_FNMADD_CMD 4'hA
`define FPU_OP_FNMSUB_CMD 4'hB

`ifdef        DOUBLE
        `define IEEE_VAL_1        64'h3ff0000000000000
`else
        `define IEEE_VAL_1        32'h3f800000
`endif


module pulp_custom_fpu #( parameter FLEN = 32 )
(
        input                                clk,
        input                                reset_n,
        // Control input signals
        input                                enable,
        input [2:0]                       rm,
        input [3:0]                       operation,
        // Data input signals
        input [FLEN-1:0]        operand_a,
        input [FLEN-1:0]        operand_b,
        input [FLEN-1:0]        operand_c,
        // Output signals
        output [FLEN-1:0]        result,
        output                                 result_valid,
        output [4:0]                   exceptions,
        output                                 divsqrt_busy
);

        wire divsqrt_enable, fpu_enable, fma_enable,
         ↪  mul_bypass, mul_only;

        assign divsqrt_enable        = enable &
         ↪  ((operation==`FPU_OP_DIV_CMD) |
         ↪  (operation==`FPU_OP_SQRT_CMD));
        assign fpu_enable                = enable &
         ↪  ((operation==`FPU_OP_I2F_CMD) |
         ↪  (operation==`FPU_OP_F2I_CMD));
        assign fma_enable                = enable &
         ↪  ((operation==`FPU_OP_FMADD_CMD) |
         ↪  (operation==`FPU_OP_FMSUB_CMD) |
         ↪  (operation==`FPU_OP_FNMADD_CMD) |
         ↪  (operation==`FPU_OP_FNMSUB_CMD) |
         ↪  (operation==`FPU_OP_ADD_CMD) |
         ↪  (operation==`FPU_OP_SUB_CMD) |
         ↪  (operation==`FPU_OP_MUL_CMD));

        assign mul_bypass                = (reset_n &&
         ↪  ((operation == `FPU_OP_ADD_CMD) | (operation ==
         ↪  `FPU_OP_SUB_CMD)));
        assign mul_only                        = (operation ==
         ↪  `FPU_OP_MUL_CMD);

        reg [FLEN-1:0] fma_operand_a, fma_operand_b,
         ↪  fma_operand_c, fma_result;
        reg [1:0] fma_op;

        wire                         fma_valid;
        wire [4:0]           fma_flags;

        wire                         divsqrt_valid;
        wire [4:0]           divsqrt_flags;
        wire [FLEN-1:0]        divsqrt_result;
wire                         fpu_valid;
wire [4:0]           fpu_flags;
wire [FLEN-1:0]        fpu_result;

// Handle input routing
always @(operand_a, operand_b, operand_c, fma_enable,
 ↪  mul_bypass, mul_only, reset_n) begin
        // Latch the various values with their
         ↪  respective enable signals to prevent
         ↪  unintended toggling
        if (~reset_n) begin
                fma_operand_a = 'b0;
                fma_operand_b = 'b0;
                fma_operand_c = 'b0;
        end
        else if (fma_enable) begin
                fma_operand_a = operand_a;
                fma_operand_b = operand_b;
                fma_operand_c = operand_c;
                if (mul_bypass) begin
                        // For add/sub the second
                         ↪  operand appears on input b,
                         ↪  but FMA input c is the one
                         ↪  fed directly to the adder
                        fma_operand_b = `IEEE_VAL_1;
                        fma_operand_c = operand_b;
                end
                else begin
                        if (mul_only) begin
                                fma_operand_c = 'b0;
                        end
                end
        end
end // always block

always @(operation) begin
        fma_op = 2'b00;
        case (operation)
                // FMA operations
                `FPU_OP_FMADD_CMD:
```

```
                                        fma_op = 2'b00;
                                `FPU_OP_FMSUB_CMD:
                                        fma_op = 2'b01;
                                `FPU_OP_FNMADD_CMD:
                                        fma_op = 2'b11;
                                `FPU_OP_FNMSUB_CMD:
                                        fma_op = 2'b10;
                                // Regular operations performed by FMA
                                ↪   - TODO: verify that these are the
                                ↪   correct opcodes
                                `FPU_OP_ADD_CMD: // = A*1+B
                                        fma_op = 2'b00;
                                `FPU_OP_SUB_CMD: // = A*1-B
                                        fma_op = 2'b01;
                                `FPU_OP_MUL_CMD: // = A*B+0
                                        fma_op = 2'b00;
                                default:
                                        fma_op = 2'b00;
                        endcase
                end
                // Instantiate FMA wrapper - TODO: replace this wrapper
                ↪   altogether eventually?
   fma_wrapper_custom
     #(
       .C_MAC_PIPE_REGS(2),
       .RND_WIDTH(2),
       .STAT_WIDTH(5)
       )
   fp_fma_wrap_i
     (
      .clk_i              ( clk         ),
      .rst_ni             ( reset_n        ),
      .En_i               ( fma_enable    ),
      .mul_bypass         ( mul_bypass    ),
      .OpA_i              ( fma_operand_a   ),
      .OpB_i              ( fma_operand_b   ),
      .OpC_i              ( fma_operand_c   ),
      .Op_i               ( fma_op          ),
      .Rnd_i              ( rm[1:0]     ),
      .Status_o           ( fma_flags     ),
      .Res_o              ( fma_result    ),
      .Valid_o          ( fma_valid     ),
      .Ready_o          (               ),
      .Ack_i            ( 1'b1          )
      );

        // Instantiate divsqrt unit
   ///////////////////////////////////////////////
   // Iterative DIV-Sqrt Unit                   //
   ///////////////////////////////////////////////

   // generate inputs for div/sqrt unit - TODO: support double
   ↪  precision
   wire                                div_start, sqrt_start;
   reg [FLEN-1:0]        divsqrt_operand_a;
   reg [FLEN-1:0]        divsqrt_operand_b;
   wire                                divsqrt_nv;
   wire                                divsqrt_ix;
   wire divsqrt_zero, divsqrt_of, divsqrt_uf; // TODO: why were
   ↪  these not declared in the original PULP fpu_private?

   assign sqrt_start = divsqrt_enable & (operation ==
   ↪  `FPU_OP_SQRT_CMD);
   assign div_start  = divsqrt_enable & (operation ==
   ↪  `FPU_OP_DIV_CMD);

   //assign divsqrt_operand_a = (div_start | sqrt_start) ?
   ↪  operand_a : 0;
   //assign divsqrt_operand_b = (div_start)              ?
   ↪  operand_b : 0;

        // Latch the inputs to prevent useless toggling here as
        ↪  well
    always @(div_start, sqrt_start, divsqrt_operand_a,
    ↪  divsqrt_operand_b, reset_n) begin
                if (~reset_n) begin
                        divsqrt_operand_a = 'b0;
                        divsqrt_operand_b = 'b0;
                end
                else begin
                        if (sqrt_start) begin
```

```
                                        divsqrt_operand_a =
                                         ↪  operand_a;
                                end
                                if (div_start) begin
                                        divsqrt_operand_a =
                                         ↪  operand_a;
                                        divsqrt_operand_b =
                                         ↪  operand_b;
                                end
                        end
                end


   div_sqrt_top_tp fpu_divsqrt_tp
     (
      .Clk_CI             ( clk               ),
      .Rst_RBI            ( reset_n              ),
      .Div_start_SI       ( div_start           ),
      .Sqrt_start_SI      ( sqrt_start          ),
      .Operand_a_DI       ( divsqrt_operand_a ),
      .Operand_b_DI       ( divsqrt_operand_b ),
      .RM_SI              ( rm[1:0]           ),
      .Precision_ctl_SI   ( 5'b0              ),
      .Result_DO          ( divsqrt_result      ),
      .Exp_OF_SO          ( divsqrt_of          ),
      .Exp_UF_SO          ( divsqrt_uf          ),
      .Div_zero_SO        ( divsqrt_zero        ),
      .Ready_SO           ( divsqrt_busy      ),
      .Done_SO            ( divsqrt_valid       )
      );

   assign divsqrt_nv = 1'b0;
   assign divsqrt_ix = 1'b0;
   assign divsqrt_flags = {divsqrt_nv, divsqrt_zero,
    ↪  divsqrt_of, divsqrt_uf, divsqrt_ix};

   // TODO: what happens when XLEN!=FLEN?
   reg [FLEN-1:0] conversion_operand;

        always @(fpu_enable, reset_n, operand_a) begin
                if (~reset_n) begin
                conversion_operand = 'b0;
        end
        else begin
                if (fpu_enable) begin
                        conversion_operand = operand_a;
                end
        end
end

reg [1:0] conversion_operation;
// Operation sel for conversion core
always @(operation) begin
        case (operation) // TODO: this core also
         ↪  supports unsigned, this is what bit 1 is
         ↪  for
                `FPU_OP_F2I_CMD:
                        conversion_operation = 2'b00;
                `FPU_OP_I2F_CMD:
                        conversion_operation = 2'b01;
                default:
                        conversion_operation = 2'b00;
        endcase
end

// FP<->Int conversion unit
conversion_core fpu_conv_core
        (
                .clk ( clk                          ),
                .reset_n ( reset_n         ),
                .enable_in ( fpu_enable),
                .data_in ( conversion_operand ),
                .rm_in ( rm ),
                .operation_in ( conversion_operation ),
                .result ( fpu_result ),
                .result_valid ( fpu_valid ),
                .flags ( fpu_flags )
        );

assign result_valid        = divsqrt_valid | fpu_valid
 ↪  | fma_valid;
```

```
        assign result                  = divsqrt_valid ?
        ↪   divsqrt_result : fpu_valid ? fpu_result : fma_valid
        ↪   ? fma_result : 0;
        assign exceptions          = divsqrt_valid ?
        ↪   divsqrt_flags  : fpu_valid ? fpu_flags  : fma_valid
        ↪   ? fma_flags  : 0;


endmodule
```

A.13 Simple FPU regfile

This is a simple register file with 3 read ports and one write port, with parameterizable data widths.

```
`ifdef DOUBLE
        `define FLEN 64
        `define BIT_SIGN 63:63
        `define BITS_EXP 62:52
        `define BITS_FRC 51:0
        `define PRODUCT_W 106
`else
        `define FLEN 32
        `define BIT_SIGN 31:31
        `define BITS_EXP 30:23
        `define BITS_FRC 22:0
        `define PRODUCT_W 48
`endif

`define XLEN 32
`define NUM_FPU_REGS 32



module FPU_regfile #(
    parameter NUM_REGS = `NUM_FPU_REGS,
    parameter WIDTH = `FLEN
) (
        input                                           clk,
        input                                          reset_n,
        input                                          in_data_valid,
        input
         ↪                  [4:0]                 wb_port_addr,
        input                 [WIDTH-1:0]         wb_port_data,
    // TODO: parameterize the number of read ports?
        input
         ↪                  [4:0]                 read_port_a_addr,
        output reg          [WIDTH-1:0]         read_port_a_data,
        input
         ↪                  [4:0]                 read_port_b_addr,
        output reg         [WIDTH-1:0]         read_port_b_data,
        input
        ↪                    [4:0]                   read_port_c_addr,
        output reg          [WIDTH-1:0]          read_port_c_data
        ↪  // NOTE: this port can be skipped if opting for CMA
        ↪  instead of FMA
);

        reg [WIDTH-1:0] fpu_regfile[NUM_REGS-1:0];

/*
        // TODO: this could be a possible alternative; latching
↪   outputs to only drive connected nets when requested
        wire [WIDTH-1:0] read_port_a_data_next;
        wire [WIDTH-1:0] read_port_b_data_next;
        wire [WIDTH-1:0] read_port_c_data_next;
        assign read_port_a_data_next =
↪   fpu_regfile[read_port_a_addr];
        assign read_port_b_data_next =
↪   fpu_regfile[read_port_b_addr];
        assign read_port_c_data_next =
↪   fpu_regfile[read_port_c_addr];
*/
/* // First version, not sure if works
        always @(posedge clk) begin
                if (in_data_valid) begin
                        // Load the input data to the
↪   destination register
                        fpu_regfile[wb_port_addr] <=
↪   wb_port_data;
                end // in_data_valid
        end // @posedge clk
*/
        generate
                genvar i;
                for (i = 0; i < NUM_REGS; i = i + 1) begin
                        always @(posedge clk or negedge
                        ↪  reset_n) begin
                                if (reset_n == 0) begin
                                        fpu_regfile[i] <=
                                        ↪  {WIDTH{1'b0}};
                                end
                                else if (in_data_valid &&
                                 ↪  wb_port_addr == i) begin
                                        fpu_regfile[i] <=
                                         ↪  wb_port_data;
                                end // if
                        end // @negedge reset
                end // for
        endgenerate

        always @(read_port_a_addr or
         ↪  fpu_regfile[read_port_a_addr]) begin
                read_port_a_data <=
                 ↪  fpu_regfile[read_port_a_addr];
        end // @(read_port_a_addr)

        // TODO: latch the read port outputs so they don't
         ↪  change when their values are not going to be used?
        // Most relevant for read port C, but depending on the
         ↪  workload it could also be useful for port B
        always @(read_port_b_addr or
         ↪  fpu_regfile[read_port_b_addr]) begin
                read_port_b_data <=
                 ↪  fpu_regfile[read_port_b_addr];
        end // @(read_port_b_addr)

        always @(read_port_c_addr or
         ↪  fpu_regfile[read_port_c_addr]) begin
                read_port_c_data <=
                 ↪  fpu_regfile[read_port_c_addr];
        end // @(read_port_c_addr)

endmodule // FPU_regfile
```

A.14 Mutliplication test

This is a minimal multiplier implemented to verify where the multiplication product could be found in the FMA pipeline.

```
import fpu_defs_fmac::*;

module mult_tb(
        input logic [C_MANT:0]      Mant_b_D,
        input logic [C_MANT:0]      Mant_c_D,
        output logic [2*C_MANT+2:0] sum_D,
    output logic [2*C_MANT+2:0] carry_D,
    output logic                MSB_cor_D
);
        // Generate partial products for multiplication
 logic [12:0] [2*C_MANT+2:0]                 Pp_index_D;
 pp_generation  pp_gneration_U0
 (
   .Mant_a_DI             (Mant_b_D         ),
   .Mant_b_DI             (Mant_c_D         ),
   .Pp_index_DO           (Pp_index_D       )
 );
        logic [2*C_MANT+2:0] Pp_sum_D;
    logic [2*C_MANT+2:0] Pp_carry_D;

 wallace   wallace_U0
 (
   .Pp_index_DI           (Pp_index_D       ),
   .Pp_sum_DO             (Pp_sum_D         ),
   .Pp_carry_DO           (Pp_carry_D       ),
   .MSB_cor_DO            (MSB_cor_D        )
 );
// "Multiplication section" ends here?

// 48-bit CSA. In fact the bit width is 49 bits including sign
↪  bit. After this CSA, EDCA is used to produce the correct
↪  sum and the carry out bit.
   logic [2*C_MANT+1:0]                     Csa_sum_D;
   logic [2*C_MANT+1:0]                     Csa_carry_D;
CSA   #(2*C_MANT+2)  CSA_U0
 (
   .A_DI                    ('0),
   .B_DI                    ({Pp_sum_D[2*C_MANT+1:0]}        ),
   .C_DI                    ({Pp_carry_D[2*C_MANT:0],1'b0}  ),
   .Sum_DO                  (Csa_sum_D                       ),
   .Carry_DO                (Csa_carry_D                     )
 );

 assign sum_D = Csa_sum_D;
 assign carry_D = Csa_carry_D;

endmodule
```

Appendix B

Other contributions

This Appendix lists other contributions to open source projects made while working with this Thesis.

B.1 PULP FPU bugfix

```
From: =?UTF-8?q?Torbj=C3=B8rn=20Viem=20Ness?= <tbness@gmail.com>
Date: Tue, 15 May 2018 10:22:26 +0200
Subject: [PATCH] Fixed bug with the sign being ignored in multiplications
when the magnitude of the result is zero
(https://github.com/pulp-platform/fpu/issues/5)

---
hdl/fpu_v0.1/fpu_core.sv | 10 +++++++---
1 file changed, 7 insertions(+), 3 deletions(-)

diff --git a/hdl/fpu_v0.1/fpu_core.sv b/hdl/fpu_v0.1/fpu_core.sv
index 37302d1..73161a6 100644
--- a/hdl/fpu_v0.1/fpu_core.sv
+++ b/hdl/fpu_v0.1/fpu_core.sv
@@ -12,10 +12,10 @@
//                                                                    //
// Engineers:  Lukas Mueller -- lukasmue@student.ethz.ch              //
//             Thomas Gautschi -- gauthoma@student.ethz.ch            //
-//                                                                    //
+//                                                                    //
// Additional contributions by:                                       //
//              lile  -- lile@iis.ee.ethz.ch                          //
-//                                                                    //
+//               Torbjørn Viem Ness -- torbjovn@stud.ntnu.no          //
//                                                                    //
// Create Date:    26/10/2014                                         //
// Design Name:    FPU                                                //
```

```
@@ -30,6 +30,10 @@
// Revision:                                                          //
//          12/09/2017                                                //
//          Updated the special cases   by Lei Li                     //
+// Revision:                                                         //
+//          15/05/2018                                               //
+//          Fixed bug with the sign being ignored in multiplications  //
+//          where the result is zero (GitHub #5) - Torbjørn Viem Ness //
////////////////////////////////////////////////////////////////////////

import fpu_defs::*;
@@ -338,7 +342,7 @@ module fpu_core
// Output Assignments
/////////////////////////////////////////////////////////////////////////////

-   assign Sign_res_D = Zero_S ? 1'b0 : Sign_norm_D;
+   assign Sign_res_D = (Zero_S && (OP_SP != C_FPU_MUL_CMD)) ? 1'b0 :
↪   Sign_norm_D;
always_comb
begin
Exp_res_D = Exp_norm_D;
```