Abstract

This thesis presents the architecture and implementation of a high-performance floating-point coprocessor for Atmel's new microcontroller.

The coprocessor architecture is based on a fused multiply-add pipeline developed in the specialization project, TDT4720. This pipeline has been optimized significantly and extended to support negation of all operands and single-precision input and output. New hardware has been designed for the decode/fetch unit, the register file, the compare/convert pipeline and the approximation tables. Division and square root is performed in software using Newton-Raphson iteration.

The Verilog RTL implementation has been synthesized at 167 MHz using a 0.18 μm standard cell library. The total area of the final implementation is 107 225 gates. The coprocessor has also been synthesized with the CPU.

Test-programs have been run to verify that the coprocessor works correctly. A complete verification of the floating-point coprocessor, however, has not been performed due to limitations in time.

Preface

I would like to thank my supervisor Erik Renno and Andreas Engh-Halstvedt at Atmel Norway for giving me helpful advice and active support during the work on this thesis. I would also like to thank professor Lasse Natvig at the Department of Computer and Information Science at NTNU for providing good feedback on the thesis text.

Contents

1 Introduction **1**
1.1 Task Description . . . 1
1.2 A Short Introduction to Floating-point Numbers and Floating-point Hardware . 2
1.3 Background for this Thesis . . . 4
1.4 Goals for this Thesis . . . 4
1.5 Outline . . . 4

2 Background **7**
2.1 IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE 754-1985) . . . 7
2.1.1 IEEE 754 Floating-point Implementation-practice . . . 7
2.1.2 Formats and Encoding . . . 8
2.1.3 Rounding . . . 10
2.1.4 Operations . . . 10
2.1.5 Exceptions . . . 11
2.1.6 Traps . . . 12
2.1.7 IEEE 754 and the ISO C Standard . . . 12
2.1.8 Mapping Floating-point Numbers to the Real Number Line . . . 12
2.2 The IEEE 754 Revision . . . 13
2.2.1 Purpose . . . 14
2.2.2 Current Changes and Additions . . . 15
2.2.3 Quadruple-precision Floating-point . . . 15
2.2.4 Fused Multiply-Add . . . 15
2.2.5 Other Proposed Additions . . . 16
2.3 Embedded FPU Implementations . . . 16
2.3.1 Throughput and Latency . . . 16

2.3.2 ARM VFP9-S 16
2.3.3 MIPS32 24Kf FPU 18
2.3.4 IBM PowerPC 603e FPU 19

3 Architecture **23**
3.1 Background for the Architecture Design 23
3.2 Requirements for the Architecture 23
3.3 Properties of the Architecture 25
3.4 Target Processor 25
3.5 Instruction Set 26
3.5.1 Arithmetic 26
3.5.2 Compare 27
3.5.3 Convert 28
3.5.4 Approximation Instructions 28
3.5.5 Move and Load-Constant 28
3.5.6 Supported Formats 29
3.5.7 Implemented Instruction Set 29
3.5.8 Encoding 29
3.5.9 Load/Store 29
3.6 IEEE 754 Compliance 30
3.7 Datapath and Main Building Blocks 31
3.7.1 Top Level 31
3.7.2 Tightly Coupled Bus 33
3.7.3 Fused Multiply-Add (FMA) 33
3.7.4 Decode/Fetch 36
3.7.5 The Register File 36
3.7.6 Compare/Convert 39
3.7.7 Approximation Table 40
3.8 Instruction Execution 40
3.8.1 Writing To a Coprocessor Register 40
3.8.2 Reading From a Coprocessor Register 41
3.8.3 FMA 41
3.8.4 Compare 41
3.8.5 Conversion Between Integer and Floating-point 41

3.8.6 Approximation 42
3.9 Division and Square Root 42

4 Implementation **47**
4.1 Implementation Requirements 47
4.2 Verilog Code Conventions 48
4.3 DesignWare Building Block IP 48
4.4 The Round-, Guard- and Sticky-bit 49
4.5 Module Descriptions 49
4.5.1 Top Level and Decode/Fetch (fpcp) 50
4.5.2 The Fused Multiply-Add Pipeline (fma) 53
4.5.3 Compare/Convert Pipeline (cpcvt) 56
4.5.4 The Approximation Tables Module (apx) 57
4.5.5 The Register File Module (fpr) 61
4.5.6 Write Unit (wr) 63
4.6 Unimplemented Functionality 63
4.7 Optimizing for Low Power Consumption 64
4.7.1 Power-reduction Strategies 64
4.7.2 Reducing Power in the Design 65
4.7.3 Potential Improvements (not Implemented) 67
4.8 Synthesis 67

5 Verification **69**
5.1 Co-simulation with the CPU 69
5.2 __builtin Macros 70
5.2.1 Moving Data into Coprocessor Registers 71
5.2.2 Moving Data from Coprocessor Registers 71
5.2.3 Sending Commands to the Coprocessor 71
5.3 Test Programs 71
5.3.1 `asm_test` 71
5.3.2 `test` 72
5.3.3 `test_fused` 73
5.3.4 `test_approx` 73
5.3.5 `cpuflushed_test` 74

5.3.6 `test_dotprod` . . . 74
5.4 FMA Pipeline Verification . . . 74
5.5 Floating-point Test Software . . . 74
5.6 Utility Programs . . . 75
5.6.1 `fdata` . . . 75
5.6.2 `fcvt` . . . 75
5.6.3 `arith` . . . 75
5.6.4 `approx` . . . 75

6 Results **77**
6.1 Latency . . . 77
6.2 Throughput . . . 81
6.3 Functionality . . . 83
6.4 Performance of Typical Embedded Applications . . . 83
6.5 Other Criteria for a Good Implementation . . . 84
6.6 IEEE 754 Compliance Summary . . . 85
6.7 Synthesis Results . . . 85
6.7.1 The Effect of Clock Gating . . . 89
6.7.2 Synthesizing CPU and Coprocessor Together . . . 89

7 Future Work **91**

8 Conclusion **93**
8.1 Project Experiences . . . 94

List of Figures

2.1 The IEEE 754 Single Format 8
2.2 The IEEE 754 Double Format 8
2.3 Single-precision Floating-point Exponent Ranges from $[1, 2^1)$ to $+\infty$ 13
2.4 Single-precision Floating-point Negative Exponent Ranges 13
2.5 The Quadruple Floating-point Format 15

3.1 Coprocessor Top Level Architecture 32
3.2 Fused Multiply-Add Datapath 34
3.3 Register File 37
3.4 Compare/Convert Unit 39

4.1 Floating-point Coprocessor Block Diagram 50
4.2 Decode/Fetch Data-flow in fpcp Module 51
4.3 Coprocessor Interfacing Using the TCB 52
4.4 Original vs. New FMA Pipeline 53
4.5 FMA Pipeline 55
4.6 Compare Data-flow 57
4.7 Convert Data-flow 58
4.8 The apx Module 60
4.9 The fpr Module 62
4.10 Read Port Logic 63
4.11 Saving Power by Restricting Inputs to Logic Modules 66
4.12 Bypassing Multiplier to Save Power 67

6.1 Instruction Latency Comparison (single) 78
6.2 Instruction Latency Comparison (double) 79

6.3 Division and Square Root Latency Comparison 80
6.4 Division and Square Root Throughput Comparison 82
6.5 Total Cell Area of Individual Modules 86
6.6 Total Cell Area at Different Clock Periods 88

List of Tables

2.1 The Exponent Range and Bias . . . 9
2.2 IEEE 754 Special Values . . . 9
2.3 Range Size and Gap Size (Single-precision) . . . 13
2.4 Quadruple Exponent Format . . . 15
2.5 ARM VFP9-S Throughput and Latency Cycle Counts . . . 18
2.6 MIPS32 24Kf FPU Throughput and Latency Cycle Counts . . . 20
2.7 PowerPC 603e FPU Throughput and Latency Cycle Counts . . . 21

3.1 Two-operand Arithmetic Instructions . . . 27
3.2 Three-operand Fused Arithmetic Instructions . . . 27
3.3 Move-Type Instructions . . . 29
3.4 Load Constant Instructions . . . 29
3.5 Floating Point instruction frequency . . . 38

Chapter 1

Introduction

1.1 Task Description

The task was given by Atmel Norway. The full text is listed here:

"Some embedded applications are best implemented using floating-point arithmetic. Software floating-point emulation can be used in some situations but for most floating-point intensive applications this is insufficient due to strict performance requirements. Many high-end microcontrollers therefore supports floating-point instructions.

The new family of microcontrollers from Atmel supports coprocessor extensions. Atmel wants to build a *floating-point coprocessor*, based on an existing fused multiply-add unit. The coprocessor must be designed using Verilog, and synthesizable with a clock period of at most 6.0 ns (166.7 MHz) using Atmel's 0.18 μm process technology. The implementation should, if time allows, support all operations specified in the IEEE 754 standard.

During design, trade-offs must be made between area, power consumption and performance, but the coprocessor should provide good performance. The resulting design should also be optimized for low-power and area.

The work should include:

- Defining an instruction set for the coprocessor.
- Implementing a coprocessor with a register file and data dependency detection.
- Implementing hardware for:
  - Conversion between single-precision, double-precision
  - Conversion between floating-point formats and integer
  - Floating-point compare
- An implementation of division and square root, if time allows.
- Verification of the coprocessor implementation."

1.2 A Short Introduction to Floating-point Numbers and Floating-point Hardware

Real numbers are implemented as *floating-point numbers* in digital computers. In contrast to real numbers used in mathematics which represent continuous numbers with unlimited precision and range, floating-point numbers are discrete entities and have both limited precision and range. Arithmetic on floating-point numbers is usually performed by a dedicated hardware module, generally referred to as a *floating-point unit* (FPU), but can also be implemented in software utilizing a regular integer Arithmetic Logic Unit (ALU). There is however a big difference in efficiency between a hardware and a software implementation. A software implementation is much slower than a hardware counterpart. A software floating-point multiply typically takes 80 cycles and a software division takes typically about 130 cycles [5]. A typical hardware implementation, like the ARM VFP9-S [3], needs only 4-5 cycles for multiply and 17-31 cycles for division. Also, multiplications are usually pipelined. The relatively slow software implementations and the opportunity for pipelining in hardware makes floating-point arithmetic well suited for hardware implementation.

Floating-point units are now a built-in component of most modern microprocessors for desktop computers and servers. In these processors the floating-point unit is tightly integrated in the pipeline. The need for efficient floating-point calculation required in 3D graphics applications and simulation has made the floating-point unit an integral part of any competitive microprocessor. In fact, most vendors also provide multiple floating-point units that executes in parallel, in order to further increase throughput and meet the demand for higher performance.

Microcontrollers used in embedded systems have been and are still primarily designed for integer and fixed-point arithmetic. The primary reason for this is that most algorithms in embedded applications do not need floating-point calculations. Fixed-point calculation implemented using integer operations is often sufficient. Also, due to a long history with very limited support for floating-point hardware, there has been an effort throughout the industry to solve many problems which in principle requires floating-point arithmetic using integer-only instructions. The Tremor decoder library [42], for example, is an integer-only Ogg Vorbis compliant audio codec for processors without floating-point unit.

High-performance floating-point arithmetic, if needed, is usually supported by utilizing a coprocessor. A coprocessor architecture is common since very few microcontrollers have been designed with floating-point support originally. The floating-point support has been added to these processors at a later stage in response to market demands. A coprocessor does not require any change to the processor architecture. This makes it a convenient solution. Another reason for using a loosely coupled coprocessor architecture is that die area can be reduced for those parts that are manufactured without the coprocessor. This reduces the cost of these parts.

The ability to provide both large *dynamic* range and precision makes floating-point numbers attractive for many applications. Fixed-point numbers, often used to model real numbers in systems without floating-point support, have a fixed precision and range which limits their flexibility and the accuracy of the results.

Floating-point arithmetic has been standardized in the IEEE-754 Standard for Floating-point

Numbers [18] since 1985, and most implementations today follow this standard. The details of the IEEE 754 Standard are described in the next chapter.

The use of floating-point numbers in embedded applications has been limited to high-end systems for years and many battery-powered systems may not afford a floating-point unit simply due to increased power consumption. The recent announcement of Floating Point Extensions [1] from ARC International and the optional MicroBlaze v4.00 Floating Point Unit [41] from Xilinx Inc., however, clearly shows that floating-point will find its way into future embedded applications to a larger extent than before.

ARC International lists the following applications in the product brief for their new Floating Point extensions (FPX):

- Consumer Products
  - 3D graphics
  - Image Processing
- Imaging
  - Post-script processing
  - Cameras
- Communications
  - GPS computations
  - Impedance matching
- Automotive Control
  - Powertrain control
  - Antilock braking systems
  - Active suspension
- Industrial Control
  - Robotics and motion control

3D graphics and Automotive Control are discussed in more detail in [30, ch. 2.1]. Impedance matching is a method for maximizing the efficiency of power transfers.

Floating-point arithmetic is inherently more complex than integer arithmetic. There are several reasons for this:

- The IEEE 754 Standard for Floating-point Arithmetic is quite detailed and must be studied carefully to ensure compliance.

- Several design alternatives exist. In contrast to most integer arithmetic units, floating-point units are usually pipelined. The number of pipeline stages and the level of resource sharing are only two factors that differentiates design alternatives. The design of floating-point hardware is also an active area of research both at academic institutions and in the industry and new designs are constantly being presented at conferences around the world. Most new advances in computer arithmetic are presented at the yearly IEEE Symposium on Computer Arithmetic.
- Due to the complexity of a floating-point unit, thorough verification is needed to ensure correct operation. The time spent on verification can thus contribute significantly to the total development time. A separate chapter of this thesis describes verification.

Designing floating-point hardware is therefore a complex engineering task that require much effort and includes the evaluation of many trade-offs.

1.3 Background for this Thesis

Fall 2004 I designed and implemented a floating-point arithmetic core for Atmel Norway as part of my 5th year specialization project (TDT 4720) [30]. The core was a double-precision Fused Multiply-Add unit. A natural continuation of this work was to integrate the Fused Multiply-Add unit in a Coprocessor. This thesis therefore builds on previous work for Atmel Norway.

1.4 Goals for this Thesis

The goal for this thesis is to design an architecture for a high-performance floating-point coprocessor and implement a subset of this architecture. The implementation will show whether or not the proposed architecture also meets the timing requirements of the target processor with which the coprocessor is to be integrated. An accurate estimate of area requirements for the architecture will also be possible based on the implemented subset. The performance and area of the implemented architecture should be comparable to and possibly better than relevant implementations of the ARM, MIPS and PowerPC architectures.

1.5 Outline

In chapter two the IEEE 754 standard for Binary Floating-point Numbers is described along with a description of the new revision of the IEEE 754 standard. The new revision is not yet finished, but will be completed in December 2005. In the end of the chapter the floating-point units (FPUs) for three state-of-the-art architectures are described and compared. The third chapter describes the architecture for the floating-point coprocessor. Chapter four then describes the implementation. Low-power techniques and synthesis is also discussed here. Chapter 5 is devoted to verification. This chapter describes how the implementation has been tested and a

description of the test programs used is provided. The results are presented in chapter 6. The throughput and latency for all instructions are compared against the three state-of-the-art architectures presented in chapter 2. Synthesis results are also presented in chapter 6. Possible future work is described in chapter 7. Chapter 8 concludes the thesis.

The appendix contains confidential material and is therefore not included in this document. The appendix will be opened to the public after a 5 year disclosure period. The Department of Computer and Information Science at NTNU could be consulted to get access to the appendix when the disclosure period is over.

Chapter 2

Background

This chapter first presents the IEEE Standard for Binary Floating-Point Arithmetic. Then ISO C standard floating-point to integer casting is described. The chapter then continues with a description of how floating-point numbers are mapped to the real number line. Section 2.2 presents the current status of the new revision of the IEEE 754 standard. The chapter then ends with a description and comparison of three state-of-the-art floating-point units used in microcontrollers from ARM, MIPS and IBM. A reader familiar with floating-point and the IEEE 754 standard may skip section 2.1.

2.1 IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE 754-1985)

All the following subsections, except for the first and the last two, are based on information that can be found in [18].

Floating-point arithmetic can be implemented in several ways, but having many different implementations makes software difficult to port between different architectures. Today most general purpose floating-point units are implemented according to the specifications in the IEEE 754-1985 standard.

2.1.1 IEEE 754 Floating-point Implementation-practice

Even though all floating-point implementations that are said to conform to the IEEE 754 standard must implement all the obligatory features, the number of *recommended* features implemented usually differs. Consequently IEEE conformance alone provides a very limited description of what a floating-point implementation is capable of. The standard also opens up to a great extent for different implementations. One example is the encoding of so-called quiet and signaling NaNs (described below), which is up to the implementer to decide. This freedom of choice with respect to implementation manifests itself several places in the standard. Indeed one of the desiderata that guided the formulation of the standard was to "Enable rather than preclude further refinements and extensions" [18, Foreword].

The standard use the words *shall* and *should* to distinguish between elements that are obligatory and elements that are only recommendations. Trying to extract only the obligatory requirements and making an implementation based on that, is not necessarily a good strategy, even if IEEE-754 conformance is the only requirement from a marketing perspective. Leaving out support for double-precision, for example, might not be a good idea since the C programming language has support for the **double** type and many applications may need the additional precision. It is therefore very important also to consider the current practice and the user of the floating-point implementation. Implementing features that are rarely used in practice will probably not increase the value of the floating-point unit from the user's perspective.

2.1.2 Formats and Encoding

The standard specifies two *basic* formats and two *extended* formats. Single and double are the basic formats. The single and double formats are 32 and 64 bits wide respectively and both formats are encoded in the same way using a sign, exponent and fraction field. The sign bit is always in the MSB and is 0 for positive numbers and 1 for negative numbers. The exponent follows the sign bit and is 8 bits for single and 11 bits for double. The rest of the bits make up the fraction which is 23 bits for single and 52 bits for double. Figures 2.1 and 2.2 illustrates the single and double format respectively.

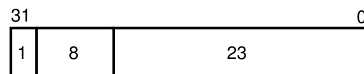

Figure 2.1: The IEEE 754 Single Format

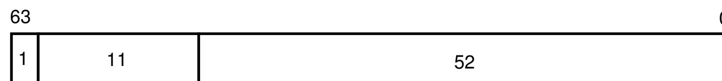

Figure 2.2: The IEEE 754 Double Format

An implementation is only required to support the single format to conform to the standard.

The binary point is always located immediately to the left of the MSB of the fraction. The double and single formats need an implicit 1 or 0 bit to the left of the binary point. This implicit bit is encoded using the exponent and is often called the "hidden" bit. A nonzero exponent indicates that the hidden bit is 1. The resulting floating-point number is then said to be *normalized*. An exponent of 0 implies that the hidden bit is 0. In this case the floating-point number is said to be *denormalized* or represent zero. The hidden bit and the fraction constitutes the significand or mantissa. The term *mantissa* will be used throughout this thesis.

The extended formats are *implementation-dependent* versions of the two basic formats. In contrast to the single and double formats, the standard does not dictate the encoding for the extended formats, but states certain requirements for the significand and exponent. The requirements for a single extended number is that it must be capable of representing a positive or

negative floating-point number with a significand with a precision of at least 32 bits and a maximum exponent greater than or equal to 1023 and a minimum exponent less than or equal to -1022. For double extended the significand must be at least 64 bits wide, the maximum exponent must be greater or equal to 16383 and the minimum exponent must be less than or equal to -16382. The extended formats must also be able to represent $+\infty$ and $-\infty$, +0, -0 and at least one signaling NaN and one quiet NaN. A NaN is returned from invalid operations like divide by zero and $\infty + \infty$. A signaling NaN raises an exception when used, a quiet does not.

Support for the extended formats are not required, but the standard recommends that implementations *should* support the extended version of the widest basic format supported. Consequently, if an implementation has support for the double format it should also have support for the double extended format, but need not have support for the single extended format. A single-only implementation similarly need no support for the extended double format.

For single and double the exponent is not stored using a 2's complement representation, as might seem natural. Instead a *biased* notation is used. The bias is a constant added to the actual exponent to make the exponent *always positive*. The smallest and largest biased exponent then becomes 0 and $2^n - 1$ respectively, where n is the number of bits used for the exponent. The smallest and largest biased exponents are reserved for encoding ± 0, $\pm\infty$, NaN and denormalized numbers. Thus, the minimum exponent is 1 and the maximum exponent is $2^n - 2$. The maximum and minimum values for the unbiased exponents along with the biases are provided in table 2.1. No bias is specified for the extended formats since the details of how the exponent is represented is chosen by the implementer.

	single	**double**
Unbiased Max. Exp.	127	1023
Biased Max. Exp.	254	2046
Unbiased Min. Exp.	-126	-1022
Biased Min. Exp.	1	1
Bias	127	1023

Table 2.1: The Exponent Range and Bias

The unbiased exponent is found by subtracting the bias: *unbiased exponent = biased exp. - bias*.

Table 2.2 shows how special values are encoded.

	Exponent	**Fraction**
± 0	all 0's	all 0's
Denormalized	all 0's	nonzero
$\pm\infty$	all 1's	all 0's
NaN	all 1's	nonzero

Table 2.2: IEEE 754 Special Values

The unbiased exponent is -126 and -1022 for all denormalized single and double numbers respectively. Thus, the smallest exponent (that is not reserved) is used.

Since a NaN requires a nonzero fraction there are several different NaNs. The standard requires

a conforming implementation to distinguish between two types of NaN: Quiet and signaling. At least *one* signaling NaN and at least *one* quiet NaN must be supported. The sign of a NaN is not interpreted. The standard recommends that quiet NaNs should "afford retrospective diagnostic information inherited from invalid or unavailable data and results" [18, 6.2]. How this is to be performed in practice, however, is unclear.

If one or both of the operands to an operation are of type signaling NaN, an invalid operation exception is raised and the result should be a quiet NaN. Using one or more quiet NaNs as operands, none of them signaling, does not raise an invalid operation exception. The result of the operation, if available, *should* be one of the input NaNs. Notice that a different NaN may be output as result instead.

It is common practice to use the MSB of the fraction to distinguish between quiet and signaling NaNs. A 1 in the MSB indicates a quiet NaN and 0 indicates a signaling NaN. The quiet NaN with only the MSB of the fraction set and the sign bit set to 0 is often called the *default NaN*. The Java Virtual Machine only use the default NaN and has no support for signaling NaNs. The default NaN is 0x7FC00000 and 0x7FF8000000000000 for single and double respectively.

2.1.3 Rounding

A conforming implementation must support four rounding modes:

- Round to Nearest (default)
- Round toward $+\infty$
- Round toward $-\infty$
- Round toward 0

Round to Nearest is the default rounding mode. The three others are user-selectable *directed* rounding modes. The *Round to Nearest* mode rounds to the representable value nearest to the infinitely precise result. If the value before rounding lies exactly in the middle of the two possible rounded results, the *even* number is chosen. More details on the Round to Nearest rounding mode will be presented in chapter 4.

2.1.4 Operations

The standard specifies the following operations, all of which must be supported in a conforming implementation:

- Add
- Subtract
- Multiply
- Divide

- Remainder
- Square root
- Round to integer in floating-point format
- Convert between different floating-point formats
- Convert between floating-point and integer formats
- Convert binary $\leftrightarrow$ decimal
- Compare

The standard allows floating-point compare to be implemented in two ways. By using a condition code or by a true-false response to predicates. Four possible condition codes are possible: *less than*, *equal*, *greater than* and *unordered*. The *unordered* condition code is used when any of the operands are NaN. Notice that the sign of zero is ignored ($+0 = -0$). An implementation that uses predicates must provide at least the following predicates $=, \neq, >, \geq, <, \leq$.

It must be possible to round a floating-point number in a given format to an integer in the same format. Conversion between all supported floating-point formats must also be supported. A conforming implementation must also be able to convert between all supported floating-point formats and all supported integer formats. Rounding should be performed according to the selected rounding mode.

For details on the specific operations see [18].

2.1.5 Exceptions

There are 5 types of exceptions, all of which must be supported in a conforming implementation:

- Invalid Operation
- Division by Zero
- Overflow
- Underflow
- Inexact

An implementation must provide a status flag for each exception. When an exception occurs, the default response is to set the corresponding flag. The standard also recommends that the user should be able to associate a trap handler with each exception. If traps are enabled, the response to an exception is to call the trap handler. The status flag for the exception may also be set.

It must be possible to test and alter each flag individually. When a flag is set, it can *only* be reset explicitly by the user. The standard also recommends that it should be possible to save and restore all the flags at one time.

Over- or underflow may occur simultaneously with inexact. All other exceptions can only occur individually.

Details on which conditions that can trigger the different exceptions can be found in [18].

2.1.6 Traps

Traps are not required by the standard, but it is recommended that a conforming implementation has support for it. See [18] for details on traps.

2.1.7 IEEE 754 and the ISO C Standard

The ISO C Standard [20] specifies that *truncation* shall be used when casting a floating-point number to an integer. This is equivalent to applying the *round toward zero* mode. For other operations the rounding mode is not defined but the default rounding mode of the IEEE 754 standard is often used in practice. This often makes it necessary to switch rounding mode before a cast and switch back again afterwards. This switching between modes can be costly on most architectures and may result in performance loss, especially in applications like 3D graphics and sound processing. 3D graphics applications must convert to integer screen coordinates and sound processing must convert to and from integer sound samples. Richard Gerber discusses ways to cope with this problem on the IA-32 architecture in [9, chap. 11].

2.1.8 Mapping Floating-point Numbers to the Real Number Line

The set of floating-point numbers is finite. This finite set of floating-point numbers are not spread out evenly across the real number line. The gap or spacing between large floating-point numbers close to infinity is much greater than the gap between small floating-point numbers. In fact, the magnitude of the gap between floating-point numbers doubles each time the exponent is increased by 1. This happens because the range doubles in size while the number of discrete floating-point numbers to spread out within that range is kept constant (2^{23} for single and 2^{52} for double). Notice that all floating-point numbers mapped to a given exponent range $[2^{n-1}, 2^n)$, are evenly spread out across this range.

Figure 2.3 shows how the ranges of single-precision floating-point numbers with increasing positive exponents grows exponentially in size. For every range, 2^{23} discrete floating-point numbers are mapped to the number line.

Figure 2.4 shows ranges for positive floating-point numbers with negative exponents (numbers between 0 and 1). Denormalized floating-point numbers are mapped to the range $(0, 2^{-126}]$.

Table 2.3 shows range- and gap-size for some exponent ranges of single precision floating-point numbers. The gap size is calculated by dividing the range size by 2^{23}. Notice how the gap size

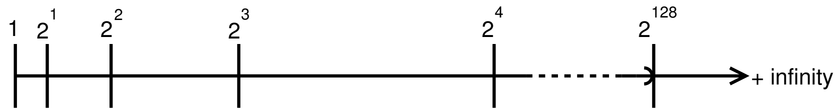

Figure 2.3: Single-precision Floating-point Exponent Ranges from $[1, 2^1)$ to $+\infty$

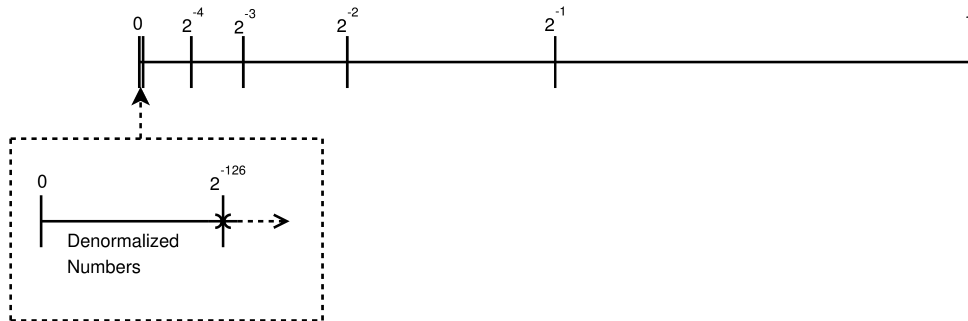

Figure 2.4: Single-precision Floating-point Negative Exponent Ranges

doubles as the exponent is increased by 1. Floating-point numbers in the range $[2^{23}, 2^{24})$ have a gap size of 1.0.

Range	Range Size	Gap Size
$(2^{-126}, 2^{-125}]$	$\approx 1.18 \cdot 10^{-38}$	$\approx 1.40 \cdot 10^{-45}$
$[2^{-2}, 2^{-1})$	0.25	$\approx 2.98 \cdot 10^{-8}$
$[2^{-1}, 2^{0})$	0.5	$\approx 5.96 \cdot 10^{-8}$
$[2^{0}, 2^{1})$	1	$\approx 1.19 \cdot 10^{-7}$
$[2^{1}, 2^{2})$	2	$\approx 2.38 \cdot 10^{-7}$
$[2^{23}, 2^{24})$	8388608	1.0
$[2^{127}, 2^{128})$	$\approx 1.70 \cdot 10^{38}$	$\approx 2.03 \cdot 10^{31}$

Table 2.3: Range Size and Gap Size (Single-precision)

For clarity, only positive single-precision floating-point numbers were considered here. The same type of mapping applies to negative floating-point numbers and double precision numbers as well.

2.2 The IEEE 754 Revision

The information in this section is based on information found in [16], [17], [31] and [39].

The IEEE 754 standard is currently under revision. The revision work started in 2000 and is to be completed in December 2005. Each month several academic- and industry-experts in the field of floating-point arithmetic meet in the San Francisco Bay area to work on the

standardization. The meetings are open to the public and participation is free of charge. In addition to these meetings a mailing list, stds-754@ieee.org, has been created which provides crucial feedback to the revision group [17]. The current draft of the proposed standard can be found at [16].

2.2.1 Purpose

The main purpose of this revision is to make the IEEE 754 standard more precise. The current standard allows too many implementation choices and makes programs less portable since the result of a computation on two implementations may differ. The standard also clarifies terminology (e.g. subnormal is used instead of denormal).

Systems that implement arithmetic according to the IEEE 754 standard can be divided into two groups, *Extended-based systems* and *Single/Double systems* [31]. Extended-based systems support an extended double-precision format that is used for all calculations. Single- and double-precision is supported through load and store operations which rounds to/from the extended double-precision format. Extended-based systems can also be configured to round results to single- or double-precision even though the result is stored in double-extended format, but this mode is not the default. Intel x86 compatible processors are extended-based systems. Single/-Double systems have no support for an extended double-precision format but outputs results directly in single- or double-precision format. The ARM VFP9-S floating-point unit described later in this chapter is one example of a Single/Double system.

The problem is that the results of a computation may differ on extended-based systems and Single/Double systems. The following excerpt, taken from [31], demonstrates this.

```
int main() {
  double  q;

  q = 3.0/7.0;
  if (q == 3.0/7.0) printf("Equal\n");
  else printf("Not Equal\n");
  return 0;
}
```

On Single/Double systems this code will work properly and print out *Equal*, since both the stored double value `q` and the intermediate compare value, `3.0/7.0`, are calculated using double-precision.

On Extended-based systems this code may not always work properly. The `q` value is stored using double-precision, but the intermediate value `3.0/7.0` in the compare will be calculated in double-extended format. Therefore, the comparison will be *not equal*.

The important thing to notice is that the extended-based systems do *conform* to the IEEE 754 standard. The standard requires that the result is rounded correctly to the precision of the destination, *but does not require the precision of the destination to be determined by the user-program*. Thus, intermediate results may differ on extended-based and single/double systems.

More examples of code that will execute differently on extended-based and single/double systems are provided in [31] along with a more thorough elaboration on these issues.

2.2.2 Current Changes and Additions

According to the IEEE 754 revision groups webpage [17], these points have been agreed upon:

- Merging IEEE-854 into IEEE-754
- Reducing implementation choices
- Resolving ambiguities in 754
- Standardizing fused multiply-add
- Including quadruple precision

The IEEE-854 is the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE 854-1987). This standard only generalizes the IEEE-754 Standard for *Binary* Floating-Point Arithmetic to other radices, and provides no important additions to the standard.

2.2.3 Quadruple-precision Floating-point

The quadruple-precision or "quad" format is a third basic binary floating-point format and is represented using 128 bits. The encoding follows the same pattern as single and double, but with wider fields for the exponent and fraction bits. Figure 2.5 shows how the 128 bits are arranged into sign bit, exponent bits and fraction bits.

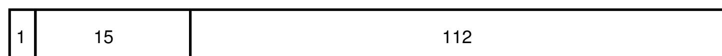

Figure 2.5: The Quadruple Floating-point Format

Table 2.4 summarizes the exponent :

	Encoding (hex)	Value
Minimum Exponent	0x0001	-16382
Maximum Exponent	0x7FFE	16383
Exponent bias	0x3FFF	16383

Table 2.4: Quadruple Exponent Format

2.2.4 Fused Multiply-Add

The Fused Multiply-Add (FMA) operation performs $x \times y + z$ in one atomic operation without an intermediate rounding of the product. This operation is currently supported in several

floating-point architectures, so it is time for a proper standardization of this operation. The Itanium architecture [19] from Intel and PowerPC architecture [15] from IBM both have support for Fused Multiply-Add instructions. The revision also proposes other versions of the FMA operation that negates the product and subtracts the z operand.

2.2.5 Other Proposed Additions

The revision proposes several other additions to the standard:

- A standard for decimal arithmetic (need in financial applications due to problems of representing decimal fractions exactly in radix 2) is proposed.
- The *round-to-nearest, ties away from zero* rounding mode for decimal arithmetic is proposed.
- Min and Max functions.
- Several classification predicates (e.g. isNaN(x)).

2.3 Embedded FPU Implementations

To fully understand the requirements for a competitive floating-point coprocessor, it is important to study other floating-point units on the market. These floating-point units sets the standard for what is to be expected from a new floating-point coprocessor from Atmel. The following sections list important properties of three industry-leading floating-point unit (FPU) architectures from ARM, MIPS and IBM. All implementations are implemented according to the IEEE 754 standard and supports all rounding modes. In the end of each section, latency and throughput numbers for supported instructions are presented. These numbers will also be very helpful for evaluating the coprocessor implementation. A graphical comparison will be presented in chapter 6.

2.3.1 Throughput and Latency

Throughput is defined as the number of cycles after the instruction has been issued when another instruction of the same type can start. *Latency* is the number of cycles after which the result is available for another operation. These definitions for throughput and latency are used throughout the thesis.

2.3.2 ARM VFP9-S

The following description is based on information found in [3] and [2].

The ARM VFP9-S is a synthesizable *Vector* Floating-point Coprocessor that can optionally be used by the ARM9E family of processor cores. These cores support clock frequencies of up to

200 MHz in 0.18 μm process technology. There also exists vector floating-point coprocessors for the ARM10 and ARM11 cores, but these are designed for much higher clock frequencies and therefore does not compare very well to the 32-bit target microcontroller the coporcessor is to be designed for.

- The VFP9-S implements the ARM VFPv2 Instruction Set Architecture. The register file has 32 registers that can be used to store 32 single-precision numbers or 16 double-precision numbers. The single precision registers are named S0-S31 and the double precision registers are named D0-D15.
- Support code is needed for complete IEEE compliance (denormalized numbers and correct handling of NaNs are not supported in hardware).
- A Run-Fast mode is supported. In this mode denormalized numbers are flushed to zero, only the default NaN is used and the hardware never traps to software routines. The motivation for having a near-IEEE compliant Run-Fast mode is that it simplifies the hardware and that some users do not need complete IEEE compliance.
- The VFP9-S coprocessor can also perform operations on vectors. A vector operation is started in only a single instruction and operates on so-called *short vectors*. Single-precision short vectors can hold up to 8 single-precision values. Double-precision short vectors can hold up to 4 double-precision values. The short vectors are composed of registers in the register file.
- There are 3 separate pipelines in the VFP9-S, which operate in parallel.
  - The Floating-point multiply-accumulate (FMAC) pipeline
  - The Divide and square root (DS) pipeline
  - The Load and Store (LS) pipeline

  The FMAC pipeline performs a multiplication, rounds the result and then adds a third operand to the product. The FMAC pipeline has 5 stages (1 Decode stage followed be 4 Execute stages):

  - The Decode stage decodes the operation and detects if any of the inputs are NaN, Infinity or Zero.
  - The first Execute stage performs the multiplication and possibly negates the value that is to be added to the product. The multiplier generates two partial products.
  - In Execute stage 2, the two partial products are added to form the product. Then the product is rounded. The operand to add is aligned to the product in parallel.
  - The third Execute stage adds the aligned addend to the product and normalizes the result.
  - The last stage rounds the final result and performs write-back.

All the arithmetic instructions except for division and square root are executed in the FMAC pipeline. Conversions between floating-point formats and between integer and floating-point formats are also executed in the FMAC pipeline.

The Divide and Square Root pipeline has also 5 stages (decode followed by 4 execute stages) but iterates several times between the first to execute stages. The third execute stage normalizes the results and performs rounding. The last execute stage selects between two results and writes the the selected result back to the register file.

The Load and Store pipeline performs loads and stores and has 5 stages: Fetch, Decode, Execute, Memory and Writeback.

Schematics for all the pipelines can be found in [3].

The area for the VFP9-S is estimated to 100 - 130K gates and have a worst case clock frequency of 180 - 210 MHz in 0.13um. The power consumption will typically be less than 0.4 mW/MHz for 0.13um [2].

The instruction latencies and throughputs for the VFP9-S are provided in table 2.5.

Instructions	Throughput (single)	Latency (single)	Throughput (double)	Latency (double)
add/sub, abs, neg, move single $\leftrightarrow$ double	1	4	1	4
compare	1	4	1	4
integer $\leftrightarrow$ floating-point	1	4	1	4
multiply	1	4	2	5
multiply accumulate	1	4	2	5
division, square root	14	17	28	31
load	1	4	2	5
store	1	3	2	4

Table 2.5: ARM VFP9-S Throughput and Latency Cycle Counts

2.3.3 MIPS32 24Kf FPU

This presentation is based on information that can be found in [27], [26] and [28].

The MIPS32 24Kf is a high-performance synthesizable MIPS core from MIPS Technologies that also implements a floating-point unit. The CPU has a 8-stage pipeline and the maximum clock frequency for the core is 625 MHz.

The properties for the FPU are listed below.

- The FPU implements the floating-point instructions in the MIPS64 Instruction Set Architecture (ISA). In this architecture there are 32 64-bit floating-point registers. Each floating-point register can store a single- or a double-precision number.

- In addition to the 32-bit single and 64-bit double floating-point formats, the FPU also supports a 64-bit *paired single* format. The FPU can also perform operations on fixed point numbers. There are two fixed-point types, a 32-bit and a 64-bit format.
- The FPU is optimized for execution of single-precision operations and most instructions have single cycle throughput and 4-cycle latency. Double precision operations have 2-cycle throughput.
- Multiply-Add instructions are supported. The product is rounded before the addition takes place. Thus, the operation is equivalent to a multiplication followed by an addition.
- The FPU cannot run on the maximum clock frequency of the CPU. Therefore the FPU can be configured to run at half the clock frequency of the CPU. This configuration is set at build time (when the chip is sent to production) and cannot be changed later. A reduced clock frequency for the FPU can also save power for battery-powered applications.
- The FPU is a separate coprocessor and has its own pipeline that executes in parallel with the CPU and is connected to the CPU via a 64-bit coprocessor interface.
- The Pipeline has 7 stages:
  - Decode, register read and unpack (FR stage)
  - Multiply tree (M1)
  - Multiply complete (M2)
  - Addition first step (A1)
  - Addition second step (A2)
  - Packing to IEEE format (FP)
  - Register writeback (FW)

  The M1 stage is executed twice for double precision multiply. Forwarding is implemented from the A2, FP and FW stages.
- The MIPS FPU is sometimes also referred to as “Coprocessor 1”.

The latency and throughput for supported instructions are given in table 2.6.

2.3.4 IBM PowerPC 603e FPU

The information in this section is based on [12], [14], [13] and [22].

The PowerPC is a well-known high-performance RISC architecture with implementations that span a wide range of market segments, from small microprocessors at 100 MHz to desktop- and server-processors operating in the GHz range. PowerPC processors are also embedded into the Virtex 4 Field Programmable Gate Array (FPGA) from Xilinx [40]. The PowerPC 603e processor is a dual-issue processor designed for low-power consumption and optimized for embedded applications. The processor operates at 100-200 MHz, depending on the process technology.

Instructions	Throughput (single)	Latency (single)	Throughput (double)	Latency (double)
add/sub, abs, neg, move single ↔ double	1	4	1	4
compare	1	4	1	4
integer ↔ floating-point	1	4	1	4
multiply	1	4	2	5
multiply accumulate	1	4	2	5
reciprocal	10	13	21	26
reciprocal sqrt.	14	17	31	36
division, square root	14	17	29	32
load	1	4	1	4
store	1	1	1	1

Table 2.6: MIPS32 24Kf FPU Throughput and Latency Cycle Counts

- The architecture has 32 Floating-Point Registers (FPRs). Double precision is used as the internal format for all the registers. When a single-precision floating-point number is loaded from memory it is automatically converted to the internal double-precision format. Three types of store instructions are provided: store double, store single and store integer. The value is also here automatically converted from the internal double-precision format to the requested output format.
- The FPU is optimized for single-precision multiply-add (single-precision multiply has single-cycle throughput, while double-precision has 2-cycle throughput).
- Fused Multiply-Add instructions are supported. The product is calculated to full 106-bits precision and then the addition is performed. The result is then rounded.
- The PowerPC 603e FPU has a 3 stage multiply-add pipeline. The three stages are:
  - Multiply stage
  - Carry Propagate Add (CPA) stage
  - Write Back (WB) stage

  There is also a Bypass Unit that checks for abnormal operands and operations. This unit may send a default result to the Write Back stage.

  A dual-pass multiplier is used. Double-precision operations therefore need two passes through the multiplier. This is why the throughput for single- and double-precision multiply operations differs. The PowerPC 604e has a single-pass multiplier. A comparison between the *604e* and 603e and a discussion of the design complexities around dual-pass multiplier architectures can be found in [22].

  Division is performed in a separate unit.
- Division is supported for both single- and double-precision. There is also support for a single-precision *reciprocal estimate* instruction.

- The PowerPC 603e FPU does not implement a square root instruction, but a fast *square root estimate* instruction is implemented. A software implementation of square root that uses an algorithm based on iterative approximation can use this estimate as an initial seed.
- Denormalized numbers and correct handling of NaNs are supported in hardware, thus eliminating the need for software exception routines.

Latency and throughput numbers for the PowerPC 603e are listed in table 2.7.

Instructions	Throughput (single)	Latency (single)	Throughput (double)	Latency (double)
abs, neg, move	1	3	1	3
add/sub single ↔ double	1	3	1	3
compare, fsel	1	3	1	3
integer ↔ floating-point	1	3	1	3
multiply	1	3	2	4
multiply accumulate (fused)	1	3	2	4
division	*	18	*	33
reciprocal estimate	*	18	**	**
reciprocal sqrt. estimate	**	**	1	3
load	1	2	1	2
store	1	2	1	2

Table 2.7: PowerPC 603e FPU Throughput and Latency Cycle Counts

A * in table 2.7 means that the operation is not pipelined and exact throughput numbers are not available. A ** in table 2.7 means that the instruction is not available.

Chapter 3

Architecture

This chapter describes the architecture of the floating-point coprocessor. First a list of the requirements for the design is presented. The chapter then continues with a description of the instruction set and a discussion of IEEE 754 compliance issues. Then, the main building blocks of the datapath are described. The discussion focuses on design-issues and on describing what each module does. How different instructions are executed in the architecture is then described. The chapter ends with a description of how division and square root is performed.

3.1 Background for the Architecture Design

The coprocessor architecture is based on previous results presented in [30]. During that project an analysis of floating-point usage in the EEMBC Benchmarks [6] was performed. The outcome of this analysis, among other considerations, motivated the design of an arithmetic core based on a Fused Multiply-Add (FMA) structure. The focus in this thesis is to define an efficient floating-point coprocessor architecture and implementing a working, synthesizable design that meets the timing requirements. Thus, a thorough analysis of floating-point instruction frequencies in typical programs is not performed. The ARM VFP9-S, MIPS32 24Kf FPU and the PowerPC 603e FPU provides insight into what is to be expected of a competitive floating-point coprocessor and are used for evaluating the proposed architecture.

3.2 Requirements for the Architecture

Based on the task description and feedback from Atmel, the following list of requirements for the architecture was made:

- In applications with specific floating-point needs, dedicated solutions can often be used. In 3D graphics applications, for example, dedicated chips can be used very effectively. The Mali(tm) Graphics Solution from Falanx [8] is a good example. The coprocessor should therefore be designed for general purposes to cover a wide range of application

areas. This means that it does not have to be optimized for specific embedded applications. The results from the floating-point usage analysis performed in the project work in [30] should be used as guidelines for the design of the architecture. A general purpose floating-point coprocessor will also be easier to program than a vector floating-point coprocessor, for example.

- High performance is important for a competitive design and the coprocessor should have performance compared to floating-point implementations from ARM, MIPS and PowerPC. This means that the latency and throughput for most instructions should be equivalent or better than for these competitors. High performance is most important for the basic arithmetic operations add, subtract and multiply. Multiply-accumulate instructions should also be executed efficiently.

- The implementation does not have to include division and square root operations, but the architecture should define how these operations can be performed. A longer latency for division may be acceptable, since division operations were not found to be be very frequent operations in [30]. Square root should also be considered as a relatively infrequent operation.

- Low power consumption is important for many battery-powered embedded applications. The architecture, however, should primarily be designed to achieve high performance. The *implementation* of the architecture, on the other hand, should try to reduce the power consumption.

- The area should not exceed the area of the VFP9-S Vector Floating-Point Coprocessor from ARM.

- The coprocessor architecture should be designed for clock frequencies in the range 150-200 MHz.

- The coprocessor interface must conform to the specification for the Tightly Coupled Bus (TCB), as specified by Atmel. The TCB bus is described in appendix C.

- The result from floating-point compare operations should be easy to process for the CPU. This will reduce the latency for branches based on the outcome of floating-point compares.

- The design should follow the IEEE 754 Standard to a great extent. Deviation from the standard is possible, if the feature is rarely used in practice or inclusion of the feature has significant implications for performance or area when implemented.

- Practical use of floating-point in the C and Java programming languages must be kept in mind during design. It should be easy to compile C and Java code into an efficient stream of floating-point assembly instructions.

- The existing Fused Multiply-Add unit from [30] should be used as the arithmetic core for the coprocessor to simplify development. The functionality of the Fused Multiply-Add unit is currently very limited and should be further extended.

- Both single- and double-precision formats should be supported, but the architecture should be optimized for double. The motivation for optimizing for double is that the study of the EEMBC benchmarks in [30] showed that the benchmark programs uses only double-precision.

- Instructions for variants of Fused Multiply-Add operations should be implemented. The variants negate the product or the addend, and makes it possible to reduce the number of floating-point negate instructions.

3.3 Properties of the Architecture

The following is a list the most important properties of the architecture.

- Double-precision Fused multiply-add unit implements all arithmetic operations and also executes single-precision operations.

- Single-cycle throughput for both double and single precision. The architecture is designed for efficient execution of double-precision operations. Single-precision operations have the same latency as double- precision operations.

- Low latency compare. This will speed up automotive applications since these applications use compares frequently [5].

- Low latency load/store. This will make it possible to reduce the size of the register file since data can be swapped in and out more quickly.

- High precision fused multiply-add operations for future IEEE 754 revision compliance.

- Divide and square root operations are executed in software. The 4-cycle latency fused multiply-add operations allow for fast software implementations of divide and square root.

- The architecture allows hardware to be shared for compare and integer to floating-point conversion.

- Denormalized numbers are supported in hardware to avoid the need for software trap routines for IEEE compliance.

3.4 Target Processor

The target system for the floating-point coprocessor is a microcontroller implementation of a new 32-bit architecture from Atmel. The clock frequency of this target is in the range 150-200 MHz. Since the microcontroller is not released yet and not open to to the public, the properties of this microcontroller are not discussed here. Further details can be found in appendix E.

3.5 Instruction Set

The instruction set was selected based on several criteria.

- The instruction set should be as complete as possible with respect to the IEEE 754 standard. There should be instructions for performing most of the operations specified in the standard.
- ISO C style conversion (truncation) from floating-point to integer should be supported.
- Both single- and double-precision should be supported. The ARM VFP9-S, MIPS32 24Kf FPU and the PowerPC 603e FPU all support both formats. Consequently, both formats should be supported to be competitive. The floating-point coprocessor should support *many* applications, also those which require double precision. If the floating-point coprocessor was to be designed for specific applications, support for single precision might be sufficient.
- Multiply-accumulate (MAC) instructions should be included. These instructions should be easy to implement using the Fused Multiply Add (FMA) pipeline. They also potentially reduce the number of instructions in programs.
- Fused Multiply-add instructions should also be included. The fused multiply-add instruction differs from MAC in that the destination register need not be the same as the addend. Fused Multiply-Add instructions can also be used to implement division and square root efficiently in software.
- Instructions for supporting multiplicative division and square root should be included. These support instructions should be provided to speed up the division and square root algorithms.

The floating-point instructions can be separated into 5 groups:

- Arithmetic
- Compare
- Convert
- Approximation
- Move and Load-Constant

3.5.1 Arithmetic

The instruction set defines both regular two-operand and *fused* three-operand arithmetic operations. The fused operations perform two operations using only one rounding. Table 3.1 lists the two-operand instructions and table 3.2 lists the three-operand instructions.

There are several motivations for including the three-operand instructions.

Mnemonic	Name
fadd	Add
fsub	Subtract
fmul	Multiply
fnmul	Negated Multiply

Table 3.1: Two-operand Arithmetic Instructions

Mnemonic	Name
fmac	Fused Multiply and Accumulate
fnmac	Fused Negated Multiply and Accumulate
fmsc	Fused Multiply and Subtract
fnmsc	Fused Negated Multiply and Subtract
ffma	Fused Multiply and Add
ffnma	Fused Negated Multiply and Add

Table 3.2: Three-operand Fused Arithmetic Instructions

- The dot product $d = a_1 \cdot b_1 + ... + a_n \cdot b_n$, is implemented effectively using multiply-accumulate. The computation is sequential as follows: $s_1 = a_1 \cdot b_1$, $s_k = s_{k-1} + a_k \cdot b_k$ for $k > 1$.
- Matrix multiplication, which is a sequence of dot products, is also accelerated.
- Another time-consuming computational problem where the multiply-accumulate instruction is useful, is in computing the value of a polynomial $y = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$. By using the Horner's rule this can be rewritten to $a_0 + x \cdot (a_1 + ... + x \cdot (a_{n-1} + a_n \cdot x)...)$. The computation then becomes $s_{n-1} = a_{n-1} + a_n \cdot x$, $s_i = a_i + x \cdot s_{i+1}$ for $i < n - 1$. s_0 is the final result.
- Inspection of the EEMBC Benchmarks for Embedded Microprocessors in [30] revealed that multiply-accumulate instructions can improve the performance of typical embedded applications.
- The Fused Multiply-Add operation will be part of the new IEEE 754 revision.
- The Fused Multiply-Add operation can be efficiently implemented in hardware.
- The regular two-operand instructions are implemented using the Multiply-Add hardware unit.
- By replacing two instructions by one the code size can be reduced. This could be important for cost sensitive applications.

3.5.2 Compare

There is only one instruction for comparing floating-point numbers. The two operands are compared and a condition code is placed in the destination register. The condition code can be

any of the four condition codes specified by the IEEE 754 standard: *greater than, less than, equal or unordered*. By putting the result into a floating-point register, a simple Move From Coprocessor Register to CPU Register instruction can be used to read the result. It is possible to perform floating-point compare using the integer instructions of the CPU. In [5] a software implementation of compare is estimated to take more than 20 cycles and a new non-IEEE floating-point format is proposed to make software compare more efficient. A hardware compare instruction is therefore the only choice for high-performance IEEE-compliant compare.

3.5.3 Convert

Conversion instructions exist for converting between single and double and converting between integer and the two floating-point formats. Both 32-bit integer and 64-bit integer formats are supported and the integers may be signed or unsigned. The wast number of formats are needed to support all the integer formats supported by the CPU. Both a Round-to-Nearest and a Round-to-Zero version of the convert from floating-point number to integer instruction must exist. The Round-to-Zero version is included since the ISO C standard specifies that the Round-to-Zero rounding mode shall be used when casting floating-point numbers to integers.

3.5.4 Approximation Instructions

Two approximation instructions are supported in the instruction set:

- Reciprocal Approximation
- Reciprocal Square Root Approximation

The Reciprocal Approximation instruction returns an approximation to $\frac{1}{x}$ correct to 4 bits of precision. The Reciprocal Square Root Approximation instruction returns an approximation to $\frac{1}{\sqrt{x}}$ correct to 4 bits of precision. This approximation is used as a seed for software implementations of multiplicative division and square root algorithms. Having support for these approximation instructions in hardware is essential for being able to implement efficient software routines for division and square root.

3.5.5 Move and Load-Constant

A few move instructions are supported in the instruction set. Table 3.3 lists the move-type instructions.

Negate and Absolute Value are also considered move-type instructions since they only alter the sign of the operand.

Three instructions for loading constants are also provided in the instruction set. These instructions are listed in table 3.4.

These instructions are included for supporting the software implementations of division and square root.

Mnemonic	Name
fmov	Move
fneg	Negate
fabs	Absolute Value

Table 3.3: Move-Type Instructions

Mnemonic	Name
fldh	Load 0.5
fldtwo	Load 2.0
fldthree	Load 3.0

Table 3.4: Load Constant Instructions

3.5.6 Supported Formats

Both single and double variants exist for all the instructions. Instructions that convert between single and double need no explicit format specifier.

3.5.7 Implemented Instruction Set

A detailed description of the *implemented* instruction set can be found in appendix B.

3.5.8 Encoding

All floating-point instructions are encoded with the generic coprocessor operation instruction, `cop`, which is part of the CPU instruction set. The syntax is as follows:

`cop CP`n`, CR`d`, CR`x`, CR`y`,` *op*

$n \in \{0, 1, ..., 7\}$, denotes the coprocessor number.

$d, x, y \in \{0, 1, ..., 15\}$, denote the destination, first operand and second operand respectively.

op $\in \{0, 1, 2, ...127\}$, is used to encode the coprocessor operation.

3.5.9 Load/Store

Generic *load/store coprocessor* and *load/store coprocessor multiple* instructions are used for loading data into the coprocessor register file and storing register file content back to memory. Instructions for moving data between the register file in a coprocessor and the register file in the CPU also exist. These instructions are part of the CPU instruction set. A description of the generic load/store coprocessor instructions can be found in Appendix D.

3.6 IEEE 754 Compliance

The Floating-point coprocessor architecture is designed to follow the IEEE 754 standard to a great extent, but is not completely IEEE compliant due to the following:

- Division and square root operations are not always correctly rounded.
- Only the the default rounding-mode is supported.
- Signaling NaNs are not supported.
- Status flags are not included in the architecture.
- Rounding to integer in floating-point format is not supported.
- Remainder is not part of the architecture.

If correctly rounded division and square root should be supported, a combined SRT divide / square root unit should be included in the architecture. The SRT method is named after Sweeney, Robertson and Tocher. They independently proposed the algorithm [34], [29]. The SRT unit would be able to calculate an exact reminder, which could be used to obtain a correctly rounded result. A detailed discussion of SRT division is out of the scope of this thesis. An in-depth description of digit-recurrence methods for division and square root can be found in [7, ch. 5,6].

There are many reasons for choosing approximate multiplicative division and square root algorithms instead of an SRT division unit.

- Higher radix SRT dividers are quite complex. Designing an efficient radix-4 or radix-8 SRT divider could easily become a masters thesis in its own right.
- Radix-2 SRT dividers are too slow.
- The throughput of SRT dividers is very low. This conflicts with the goal of building a high-performance floating-point coprocessor.
- Using a software-based multiplicative method makes it possible to pipeline division and square root operations. Up to 5 single precision divide operations or 2 double precision divide operations can be pipelined. This can improve throughput significantly compared to division/square root based on SRT.
- In many applications, especially low-cost consumer applications, a correctly rounded result is not always critical. In 3D graphics applications, for example, it should be possible to trade the reduced precision for higher throughput. Audio and video decoding will also benefit from increased throughput and the user will probably not be able to see or hear any difference in the output.

- Division and square root are generally regarded as not very frequent operations. Thus, mission-critical applications requiring a correctly rounded result might still reach acceptable performance with slow IEEE-compliant software implementations of division and square root.

- The lookup-tables required for the initial approximation instructions will use much less area than a complete SRT divider unit.

Supporting only one rounding mode simplifies the design. The Round-to-Nearest mode is the default and produces the smallest round-off error. Consequently this is the rounding mode used in practice and most applications will never need the other three rounding modes. Conversion from floating-point to integer is an exception. The ISO C standard specifies that truncation shall be used for this operation. The architecture therefore supports Round-to-Zero conversion instructions.

Signaling NaNs are not supported in the architecture. The reason for this is that signaling NaNs are not much used in practice. I base this assumption on the fact that I was not able to find *any* good source of information on the practical use of signaling NaNs. The only source I found, was a short list of proposed usages of signaling NaNs on wikipedia.org.

The architecture does not include the required status flags of the IEEE standard. The motivation for omitting these flags is that most applications do not need access to these flags. Well designed software should never underflow or overflow. If an underflow or overflow condition occurs, it can be hard to recover from the error. In practice you have to check the result of every operation if you want to be on the safe side.

The reminder operation can be implemented in software using similar techniques as for division, but is not considered in this thesis.

3.7 Datapath and Main Building Blocks

This section describes the architecture of the Floating-point Coprocessor. Relevant design issues are discussed in each subsection along with the descriptions.

3.7.1 Top Level

Figure 3.1 shows the top level architectural building blocks of the floating-point coprocessor and how they are interconnected. The wide black arrows also illustrate the data-flow through the functional units. The FMA-, Approximation Table- and Compare/Convert-units are independent and can operate in parallel. The FMA has a dedicated write-port in the register file while all the other modules share a common write-port. The Approximation Table unit is used for accelerating software-based multiplicative methods for division and square root. The following subsections describe each of the functional units and the Coprocessor Bus.

Figure 3.1: Coprocessor Top Level Architecture

3.7.2 Tightly Coupled Bus

The Tightly Coupled Bus (TCB) is the interface between the CPU and the Floating-point Coprocessor. The TCB is a pipelined bus with an address-phase and a data-phase. The CPU is the master and initiates all commands to the coprocessor slaves. If a coprocessor is not ready to reply to the request from the CPU, however, it stalls the CPU until it is ready. The coprocessor does not need to flush commands due to mispredicted branches. If the CPU issues a command to a coprocessor, the command is never aborted by the CPU unless the CPU is flushed due to an exception. A detailed description of the TCB can be found in appendix C.

3.7.3 Fused Multiply-Add (FMA)

The double-precision Fused Multiply-Add unit is the arithmetic core of the design. All arithmetic operations are executed in this unit. The Multiply-Add unit performs the operation $A \times B + C$ as one atomic operation using only one rounding. The product is calculated to full precision before the C operand is added. Figure 3.2 shows the datapath of the Fused Multiply-Add unit. The Multiply-Add unit is based on the basic design in [24] and the implementation is an optimized and extended version of the implementation presented in [30].

The FMA unit is fully pipelined. This allows a new operation to be started every cycle. The number of pipeline stages of an implementation depends on the clock frequency requirements. The original implementation presented in [30] used 3 cycles to complete an operation. The new FMA design expands the number of pipeline stages to 4. This was done to allow an implementation to meet the timing requirements and possibly reduce area.

Since the FMA unit is fully pipelined, the multiplier uses a single pass to calculate the product. Dual-pass architectures use a smaller multiplier and calculates single precision products in one pass and double-precision products in two passes. The smaller multiplier in dual-pass multipliers occupies less area but limits the throughput of double precision multiply operations to 2 cycles. A dual-pass multiplier design also complicates the design, making the verification task harder. A Comparison of single- and dual-pass floating-point units can be found in [22]. The article discusses 7 complications that arise when a dual-pass multiplier is used. The area of the multiplier arrays in the single- and dual-pass floating-point units are $2.5mm^2$ and $5.1mm^2$ respectively.

The FMA unit is also used to perform conversion between the two floating-point formats. Conversion is performed by setting the input-format and output-format appropriately and setting both the A and B operand to 0.

The floating-point operands input to the FMA unit can be normalized or denormalized. Having support for denormalized numbers eliminates the need to trap to software exception handlers when denormalized numbers are used as input. This allows for predictable performance that is independent of the input.

The FMA unit outputs the default NaN if the result is Not a Number and only the round-to-nearest rounding mode is supported.

A fused multiply-add unit has many advantages and some disadvantages compared to sepa-

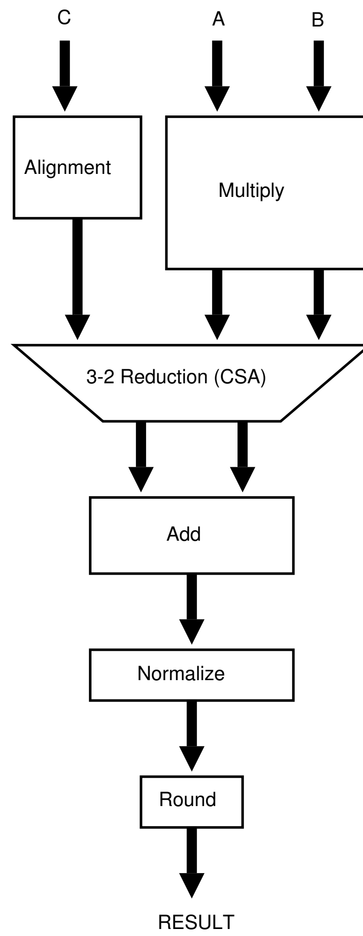

Figure 3.2: Fused Multiply-Add Datapath

rate multiply and add units.

A fused multiply-add unit executes multiply-add sequences very efficiently. Multiplication followed by addition is a very common operation in many algorithms, the most important being calculation of the dot product. The dot product is an essential operation in several linear algebra problems. Matrix multiplication for example is implemented as a series of dot products. The dot product is also very common in physics and geometry calculations based on vector math.

A fused multiply-add unit produces a more accurate result compared to a sequence of a multiplication followed by an addition. The increased accuracy is due to performing only one rounding. For long computations that involve steps where results are accumulated thousands of times this additional accuracy can cause the FMA unit to produce better results since fewer round-off errors are accumulated.

Division and square root algorithms based on multiplication can be implemented efficiently using fused multiply-add operations and may also benefit from the increased accuracy (the software implementations presented in [25] use Fused Multiply-Add instructions). The proposed method for division and square root is based on fused multiply-add instructions.

Addition and multiplication are easily implemented using the Fused Multiply-Add unit. The C operand is set to 0.0 for multiplication and the B operand is set to 1.0 for addition.

Only a few modifications must be applied to improve a basic fused multiply-add unit to also be able to negate the C operand and negate the product. This makes it possible to perform subtraction and the need for a negate instruction is eliminated in practice. The Multiply-Add is capable of negating both the C operand and the product.

A fused multiply-add unit is also efficiently implemented in hardware.

- The alignment of the C operand is performed in parallel with the multiplication.
- A fast Carry Save Adder is used to reduce the number of operands to add from 3 to 2. Using this approach the C operand simply becomes part of the Wallace tree [38] of the multiplier.
- Combining multiply and add into a common pipeline allows for more sharing of hardware resources. In particular the shifter used for normalization and rounding are shared.

The draft for the next revision of the IEEE 754 standard [16] includes the fused multiply-add operation. By choosing a fused multiply-add design it should be easy to adopt future implementations to the new revision of the standard.

The FMA structure is a modern architecture used by many high-performance processors today. The floating-point architecture for Intel Itanium 2 and IBM PowerPC are both based on the fused multiply-add units.

Due to the high precision of the intermediate result (161-bits) a large adder is needed. Fast and area-efficient adder-implementations like the Brent-Kung adder [4] must be used in the implementation to meet the timing requirements.

A possible disadvantage in super-scalar designs is that multiply and add instructions cannot be issued in parallel. The Floating-point Coprocessor is a a single-issue architecture so this apparent disadvantage will not have any impact on performance.

For some applications it may turn out to be a *disadvantage* that the fused multiply-add operation not always produces the exact same result as a multiply followed by an add. If the outputs from a signal-processing application is compared to a Matlab reference model the results may differ slightly. This complicates verification. A compiler-switch should therefore be used to explicitly enable the fused multiply-add instructions. The ARM VFP9-S [2] implements a *chained* multiply-accumulate operation which rounds the intermediate product before adding the third operand. A compiler can therefore safely insert MAC operations.

3.7.4 Decode/Fetch

The Decode/Fetch unit is responsible for decoding incoming commands from the TCB bus, fetching operands from the register file and then issue commands to the FMA unit, Compare/Convert unit or Approximation Table unit. The Decode/Fetch unit uses 1 cycle to decode a command, fetch the operands and issue the command to the appropriate unit. The Decode/Fetch unit also handles read and write operations from the TCB bus, move operations, and loading of constants. Since the Compare/Convert unit, Approximation Table unit and write operations from the TCB bus share a write-port in the register file, the Decode/Fetch unit must be careful not to cause any write-conflicts to occur. It is not possible to stall any of the pipelines, so the Decode/Fetch unit must examine the state of all units that share the write-port before an operation can be issued. If a write-conflict is predicted to occur, the operation must be delayed by signaling back to the CPU that it is not ready.

Data dependency detection is also performed by the Decode/Fetch unit. When an operation is started, the destination register is always locked. A locked register cannot be read or written. Thus, an operation that tries to access currently locked registers will not be started until all register operands have been unlocked. It is the Register File that is responsible for the actual locking mechanism and for keeping track of locked registers, but it is the Decode/Fetch unit that specifies when and which register to lock. Unlocking of registers is performed automatically by the Register File when a result is written back to the Register File. The functionality for checking if register operands are locked is also part of the Register File unit.

3.7.5 The Register File

The register file has 16 32-bit registers, CR0-CR15. Double-precision numbers are stored in pairs of registers with the least significant half stored in the even numbered registers and are addressed using only the even register addresses. Thus, the register file can store up to 16 single- or up to 8 double-precision numbers. It is the compiler's responsibility to keep track of the formats of the floating-point numbers in the registers. Notice that the register file may contain a mix of single- and double-precision numbers.

Figure 3.3 shows the register file interface.

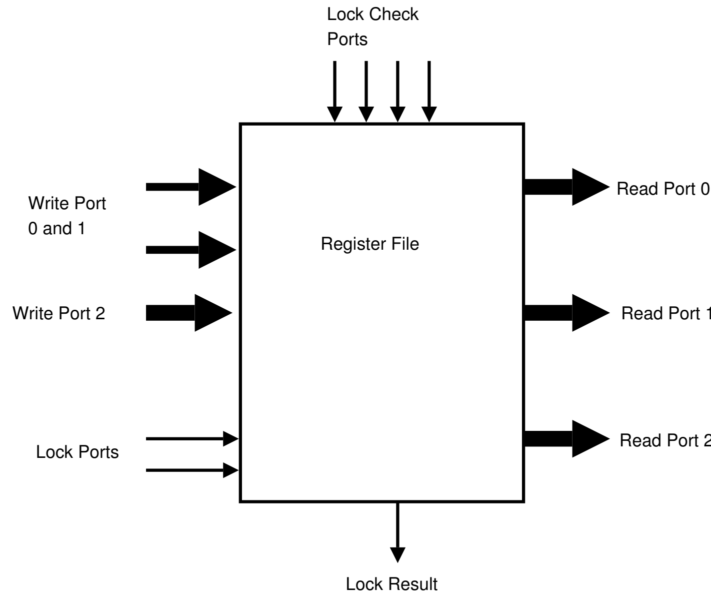

Figure 3.3: Register File

The register file has 3 write-ports, two 32-bit and one combined 32/64-bit. The two 32-bit write-ports can write to any of the 16 32-bit registers. The combined 32/64-bit can write to a single 32-bit register (single) or two consecutive 32-bit registers (double). Two separate 32-bit write-ports are needed since the *load coprocessor multiple word* instruction may need to write two possibly non-consecutive 32-bit words at a time. The combined 32/64-bit write-port is dedicated for write-back from the FMA unit. A dedicated write-port is used to ensure that a steady high throughput through the FMA unit is possible. The total number of write-ports, however, should be kept at a minimum to reduce area and simplify routing. It should be reasonable to let the non-arithmetic operations share a common write-port.

- Only conversion between integer and floating-point numbers causes other non-arithmetic operation to be suspended one cycle since writes, compares and approximation lookups all complete in one clock cycle. The results from the study of floating-point instruction frequencies performed by ARM Ltd. in [11], show that such conversion instructions only account for 2% of all executed floating-point instructions. The impact on performance due to write-conflicts should therefore be negligible.
- Write operations from the TCB bus are not suspended due to write-conflicts as long as only the FMA unit is used.

3 combined 32/64-bit read-ports are needed to fetch all the operands for the multiply-add instructions.

The Register File has the ability to forward input values that are to be written on the next clock edge. Forwarding improves the latency of instructions and is essential if maximum perfor-

mance is to be achieved. Centralizing all forwarding logic in the Register File simplifies the forwarding of results and input data from write operations on the TCB bus.

Keeping track of locked registers is also the responsibility of the Register File. Putting the lock/unlock mechanism into the Register File simplifies the unlocking of registers by allowing an automatic unlocking scheme to be used. A register is simply unlocked whenever a value is written to the register. This has the implication that all registers must always be explicitly locked before they are written.

The register file should be capable of holding enough floating-point numbers to keep a steady high throughput through the pipeline. A longer pipeline will, in general, require more registers if the same level of throughput is to be achieved. If the register file has too few registers compared to the length of the pipeline, the coprocessor may not be able to start incoming operations due to unfinished operations in the pipeline with locked destination registers. With a single-cycle throughput pipeline, the number of registers should be no less than *the maximum number of operands for an instruction* $\times$ *pipeline length* to ensure maximum flexibility and throughput. The Fused Multiply and Add instructions that calculate $d = x \times y + z$ require 4 operands and the number of pipeline stages for the FMA unit is 4. The ideal number of registers would therefore be 16. The number of double-precision registers, however, is limited to 8 in the architecture. Limiting the number of double-precision registers to 8, reduces the size of the register file.

It is essential that the architecture provides fast load/store operations since the size of the register file is relatively small. The compiler will have to swap data in and out of the register file very frequently. The architecture therefore uses highly efficient load/store instructions for both single- and double-precision. It has been shown that load and store operations are in fact very frequent operations. As a part of the design of the FPA10 floating-point coprocessor in 1993, ARM Ltd. performed an analysis of floating-point instruction frequencies for typical programs running on ARM processors. The results, taken from [11], are presented in table 3.5.

Instruction	**Percentage of Exec. Instr.**
Loads	44 %
Stores	23 %
Add	13 %
Multiply	10.5 %
Compare	3 %
Int/fp and fp/int conversion	2 %
Divide	1.5 %
Others	3 %

Table 3.5: Floating Point instruction frequency

The results in table 3.5 show that loads and stores contribute to 67 % of the executed instructions. This also clearly motivates the design of efficient load/store instructions. Execution of load/store instructions is described in sections 3.8.1 and 3.8.2 later in this chapter.

One alternative is to store all floating-point numbers in one format. Most processors for desktop and server computers use this approach, usually by converting to double extended format

or even higher precision. Intel Itanium, for example, uses 82-bit format (64-bit significand, 17-bit exponent and 1 sign bit) internally [19]. PowerPC converts single precision numbers to double precision during load and converts back to single precision during store. This scheme clearly simplifies the hardware since single precision floating-point numbers does not need any special treatment in the arithmetic datapath.

Even though the IEEE 754 standard dictates a specific format for single- and double-precision floating-point numbers, the register file, may use a different storage format internally. The load/store instructions could then convert to and from this internal format respectively. One interesting option would be to store the significand as a two's complement number and the exponent as two's complement unbiased number. This would require additional bits to represent -0, ±infinity and NaN. Another disadvantage with this representation is that compare operations cannot be performed directly. To keep compare and load/store operations simple the IEEE 754 format is used also internally in the register file. Several changes to the original FMA pipeline might also be necessary if the internal format was changed.

3.7.6 Compare/Convert

The compare/convert unit executes floating-point compare instructions and instructions that convert between integer and floating-point formats. Both 32-bit and 64-bit integers are supported and the integer may be signed of unsigned. Figure 3.4 shows the pipeline stages in the Compare/Convert pipeline.

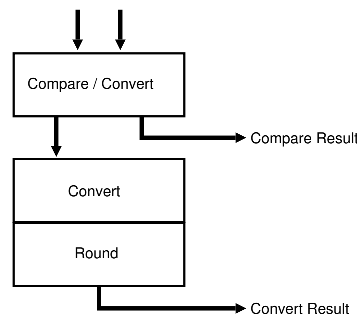

Figure 3.4: Compare/Convert Unit

Compare instructions complete during the first clock cycle and the condition code is written back to the register file. The adder used for comparison is also used for converting a negative integer to a positive integer when executing an integer to floating-point conversion. The zero-check performed during comparison is also reused for conversion.

An examination of a series of *automotive* control programs in [5] revealed that floating-point compare instructions dominate floating-point computation. 44 % of the executed floating-point instructions were compare instructions. This motivates the design of a very efficient floating-point compare operation. The outcome of a compare operation is often used as a branch condition right after the compare. Therefore the latency of the compare instruction should also be

as low as possible.

The IEEE 754 floating-point format was specifically designed to simplify comparison of floating-point numbers. By using a biased exponent and a one's complement significand and assigning the exponent to the most significant bits the comparator for floating-point simplifies to a comparator similar to an integer comparator.

Conversion between integer and floating-point is performed in the next two stages. The integer is converted from signed to unsigned in the first stage. That is why it is a combined compare/convert stage.

3.7.7 Approximation Table

The Approximation Table unit is responsible for looking up approximations to reciprocal and reciprocal square root. The approximation is used in software implementations of multiplicative division and square root based on Newton-Raphson iteration. The approximation is guaranteed to be correct to 4 digits after the binary point. How division and square root is computed is discussed in more detail in the next section. The Approximation Table unit is purely combinational and writes the requested approximation back to the register file on the next clock edge.

3.8 Instruction Execution

It is not possible to stop an operation when it has been started. Since compare, convert, approximation and load (write) instructions all share the same write-ports of the Register File, the Decode/Fetch unit must take care not to cause any write-conflicts to occur. The Decode/Fetch unit inspects the registers in the Compare/Convert pipeline to predict write-conflicts before they occur. If a write-conflict is predicted the instruction is not allowed to start. The Decode/Fetch unit must also check that none of the operand registers or the destination register are locked. If any of the operands or the destination register is locked, the coprocessor signals back to the CPU that it is not ready to perform the operation. The CPU will then stall until the coprocessor becomes ready.

3.8.1 Writing To a Coprocessor Register

A WRITE operation (see appendix C) is performed as follows: First, the Decode/Fetch unit must ensure that it will be safe to write to the specific register in the next clock cycle. It will not be safe to write to the register if the register is locked or there is a write-conflict with the Compare/Convert pipeline. If the Decode/Fetch unit finds it safe to perform the WRITE operation it saves the register number in a register. In the next clock cycle the coprocessor normally expects the value to write to be present on the TCB and the value is written to the Register File. If the CPU was flushed in the previous clock cycle, however, the value on the bus will be invalid and the Write Unit must abort the WRITE operation. WRITE operation(s) are produced

on the TCB bus when a *load coprocessor* instruction is executed. The load instructions are listed in appendix D.

3.8.2 Reading From a Coprocessor Register

When the Decode/Fetch unit detects a READ operation (see appendix C) on the TCB it tries to fetch the value of the requested coprocessor register from the Register File. If the register is not locked the value is fetched and stored in a temporary register. In the next clock cycle the stored value is written to the CPU over the TCB. If the register to read is currently locked, the coprocessor signals back to the CPU that it is not ready. It is not necessary to abort the READ operation if the CPU is flushed. The value written to the TCB will simply be ignored by the CPU. The CPU generates READ operation(s) on the TCB bus when a *store coprocessor* instruction is executed. The store instructions are listed in appendix D.

3.8.3 FMA

Several instructions are executed by the Fused Multiply-Add pipeline. When the Decode/Fetch unit decodes one of these instructions it tries to fetch up to 3 operands from the Register File depending on the instruction. If none of the operands or the destination register are locked, the operands are written to registers in the Decode/Fetch unit. The destination register is locked to ensure that no other instructions can write to the destination register before the instruction completes. All other relevant inputs to the FMA pipeline are also written to registers. If any of the operands or the destination register are locked the coprocessor signals that it is not ready, and will not start the FMA-operation. The result of the FMA operation is written back into the Register File in the last pipeline stage using a dedicated write-port.

3.8.4 Compare

A compare instruction will cause the Decode/Fetch unit to fetch and store operands similar to instructions that execute in the FMA pipeline. The compare instruction, however, will always need to fetch exactly two operands and will always write the result to a 32-bit register. The destination register is locked. The Decode/Fetch unit will not start the compare instruction if this results in a write conflict with an already started convert instruction. The result of the compare is written to the register file in the end of the clock cycle after decode. The compare result is placed in the 4 LSBs of the destination register. The result can be less, greater, equal or unordered.

3.8.5 Conversion Between Integer and Floating-point

Convert instructions only need to fetch one operand. The operand is sent to the Compare/-Convert pipeline in the next clock cycle. The conversion then takes 2 clock cycles. In the third cycle the result is rounded and written back to the register file.

3.8.6 Approximation

The approximation instructions also only need to fetch one operand. The operand is sent to the Approximation Table unit in the cycle after decode and the result is written back to the register file in the end of this cycle, as for compare operations.

3.9 Division and Square Root

Division and square root is executed in software by utilizing the multiply and multiply-add instructions of the coprocessor. These multiplicative algorithms computes division and square root by iterative approximation. Both algorithms start with an estimate of the result and come closer to the correct result for each iteration. The total execution time for the multiplicative methods is determined by two factors:

- The number of iterations
- The execution time of one iteration

The algorithm used for computing division and square root is based on Newton-Raphson iteration. A division is performed by first computing the reciprocal, $\frac{1}{d}$, and then multiplying by the dividend to get the quotient. The recurrence used to calculate the reciprocal is:

$$R_{j+1} = R_j \times (2 - R_j \times d)$$

where the index j is the iteration number, d is the divisor and R_j is the approximation to $\frac{1}{d}$ in iteration j.

The Newton-Raphson method is a general method for finding the zero of a function, $f(R)$. The general Newton-Raphson iteration formula is as follows:

$$R_{j+1} = R_j - \frac{f(R_j)}{f'(R_j)}$$

By using the function $f(R) = \frac{1}{R} - d$, whose zero is $\frac{1}{d}$, the recurrence formula for the reciprocal can be derived.

To start the first iteration, an initial approximation, R_0, to the final result is needed. The accuracy of this approximation defines the number of iterations that must be executed in order to achieve a desired accuracy of the final result. The reason for this is due to the following: Unlike digit recurrence methods like SRT, that retires a constant number of correct bits of accuracy in each iteration, multiplicative methods like Newton-Raphson have a *quadratic convergence rate*. This means that the number of bits of accuracy doubles after each iteration. A simple proof for the quadratic convergence rate property can be found in [7, ch. 7].

Square root is also computed based on the Newton-Raphson iteration method. First, the reciprocal square root, $\frac{1}{\sqrt{x}}$, is calculated using the following recurrence:

$$R_{j+1} = 0.5 \times R_j \times (3 - x \times R_j^2)$$

Then, the result is multiplied by x to obtain the square root ($\frac{1}{\sqrt{x}} \times x = \sqrt{x}$).

The function used in the generic Newton-Raphson formula to derive the above recurrence formula is $f(R) = \frac{1}{R^2} - x$. This function is zero when $R = x^{-1/2}$.

Since the Newton-Raphson method is based on a recurrence, it is self-correcting. This means that errors caused by rounding of intermediate results is corrected in the next iteration. In other words, the error due to rounding is not allowed to build up during the computation.

For division, each iteration involves two dependent operations, both executed by the Fused Multiply-Add unit: A Negated Multiply-Add instruction and a multiplication. The advantage of being able to perform multiply and add in a single fast instruction is clear.

For square root, each iteration is more costly. 3 Multiply instructions and 1 Negated Multiply-Add instruction are required. The instructions can be pipelined to a certain degree.

Another multiplicative method for division and square root is the Goldschmidt's algorithm [10]. This algorithm is based on series expansion. The advantage of this method is that the individual operations in each iteration can execute in parallel. A disadvantage of this method is that it does not have the self-correcting property. This makes the accuracy of the result less predictable. A good solution would be to use Goldschmidt's algorithm in the first iterations and then switch to Newton-Raphson iteration in the last iterations. The Newton-Raphson iterations would then correct the rounding errors. The implemented division algorithm is a based entirely on Newton-Raphson to provide a simple and reliable solution. The Newton-Raphson iterations are also easily pipelined. The number of registers needed for the Newton-Raphson method is 3. This allows up to 5 single-precision divisions or 2 double-precision divisions to be pipelined. The registers are mapped to the 5 pipelined divisions as follows during execution (single precision divides):

- CR0, CR1, CR2 - Used for division operation 1
- CR3, CR4, CR5 - Used for division operation 2
- CR6, CR7, CR8 - Used for division operation 3
- CR9, CR10, CR11 - Used for division operation 4
- CR12, CR13, CR14 - Used for division operation 5
- CR15 - The constant 2.0

For double precision divides the registers are used as follows:

- CR1:CR0, CR3:CR2, CR5:CR4 - Used for division operation 1
- CR7:CR6, CR9:CR8, CR11:CR10 - Used for division operation 2
- CR15:CR14 - The constant 2.0

Wave-forms from simulation of the Newton-Raphson division and square root functions, both pipelined and non-pipelined, can be found in appendix L.

The Goldschmidt algorithm requires more registers and therefore does not provide the same level of pipelining. Square root operations are pipelined in a similar fashion, but maximum 4 single-precision square root operations can be pipelined since the Newton-Raphson iteration requires *two* constants in the register file (0.5 and 3.0). A good description of Goldschmidt's algorithm along with a discussion of how this algorithm can be used effectively in software implementations can be found in [25].

The number of iterations needed to achieve enough correct bits in the final result can be reduced by increasing the number of correct bits in the initial approximation, R_0. A trade-off, however, must be made between the accuracy of the initial approximation and the number of iterations, since the size of the lookup table more than doubles each time the number of correct bits in the approximation is increased by 1. The simplest solution would be to always start the first iteration with an approximation with only one correct bit. This would remove the need for a lookup table. Since the number of correct bits grows very slowly in the first iterations, this would not be a good solution (only 8 correct bits are produced after 3 Newton-Raphson iterations). In order to get 53 bits of precision, 6 iterations are needed. $(1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64)$. The coprocessor architecture uses 4 bits of precision for the initial approximation. This leads to a small table with 16 entries of 4 bits each for reciprocal approximation and a slightly bigger one for reciprocal square root (32 entries of 5 bits each). The number of iterations is 3 for single and 4 for double. The number of correct bits develops as follows:

- $4(initial) \rightarrow 8 \rightarrow 16 \rightarrow 32$ (single)
- $4(initial) \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$ (double)

In the last iteration the single-precision result is correct to 32 bits which is sufficient to get the 23 bits. With 64 correct bits after 4 iterations the double result is also correct to 52 bits.

Instructions for loading the constants used in the Newton-Raphson iteration (0.5, 2.0 and 3.0) are included in the instruction set. This speeds up the software implementations since loading of constants from memory may potentially involve a costly cache miss.

There are many advantages of using a multiplicative software-based method for computing division and square root. These advantages were outlined previously when IEEE 754 compliance was discussed and will not be repeated here. A few disadvantages with the software implementation of division and square root, however, must also be mentioned:

- Getting a correctly rounded result according to the Round-to-Nearest scheme dictated by the IEEE 754 standard is not trivial. If a correctly rounded result *must* be obtained, it does not help if the least significant bit is sometimes incorrect. Careful analysis and mathematical proofs are needed to ensure that the result is correctly rounded for all inputs. Thus, a trade-off between accuracy of results, performance and analysis-complexity was made. Reduced precision was prioritized over IEEE compliance. This keeps the design simple and made it possible to complete an implementation of division and square root within the time bounds for the project.

- Since the CPU is kept busy during the whole division operation, it consequently cannot do any other useful work in parallel. This may lower the overall performance of the application.
- The increased activity on the TCB bus increases the power consumption.

Chapter 4

Implementation

This chapter describes the implementation of the architecture. First a list of requirements to the implementation is presented. The next sections then describe the code conventions used and mentions which IP modules have been used. Section 4.4 then describes the extra bits needed for correctly rounded addition and subtraction of floating-point numbers. In section 4.5 the modules of the implementation are described. This is the largest and most important part of this chapter. A short list of unimplemented functionality is then presented. The chapter ends with a description of how power consumption was reduced in the implementation and a description of synthesis constraints.

4.1 Implementation Requirements

The implementation should follow certain requirements. Below is a list of these requirements.

- The coprocessor must be implemented in synthesizable Verilog.
- A clock period of 6.0 ns or less must be used for the coprocessor clock. It is desirable that the minimum clock period is slightly less than 6.0 ns in order to simplify layout and routing.
- The implementation should be as complete as possible with respect to the IEEE 754 standard, but due to the extent and complexity of the task it is reasonable that certain limitations must be made.
- The Fused Multiply-Add unit currently (at project start) has a minimum clock period of 12 ns and must be optimized significantly to meet the new timing requirements.
- The coprocessor implementation should be optimized for low power consumption where this is appropriate.

4.2 Verilog Code Conventions

To make the implementation more readable and easier to debug, Atmel's code conventions for Verilog have been used. The most important guidelines are described below.

- All signals that are driven directly by registers ends with `_r`. This makes it easy to distinguish combinational signals from registered ones.
- Combinational signals that are going to be written to registers in the next clock cycle have the same name as the register but ends with `_r_nxt`. This makes it easier to identify the combinational logic that produces the registered value.
- Signals that are active low ends with a `_n`.
- Asynchronous signals ends with a `_a`. The only asynchronous signal is the reset signal. It is an asynchronous active low signal and is named *rst_n_a*.
- All signals are prefixed with the name of the module that drives the signal. This makes it easier to follow signals through the sub-modules.
- Constants used by a module are put in a file with the same name as the module with "_params" appended (ie. the constants used for the module `fma` are declared in a file called `fma_params.v`).
- Functions reused several times are also placed in files similar to constants. The file-name is appended with "_funcs" in this case.
- The macros provided for Emacs in Verilog mode are used where this is possible.
- Blocking assignments are used for all combinational logic and non-blocking statements are only used for clocking of registers and for reset. This makes it much easier to understand what the code does and simplifies debugging. The synthesis tool will be able to infer what operations can go in parallel so the use of blocking assignments will not have any consequences for synthesis.
- Clocking of registers is performed in the end of each module.

4.3 DesignWare Building Block IP

Intellectual Property (IP) modules from the Synopsys DesignWare Library [32] have been used throughout the implementation. The use of IP modules has several advantages:

- It simplifies development since fewer modules need to be designed.
- Another important advantage with using the IP modules from Synopsys is that Design Compiler (DC) [33] can select among several different implementations during optimization (e.g. if a *fast* adder is needed, a Brent-Kung adder implementation may be selected and if a *small and slow* adder is needed a simple ripple-carry adder may be selected).

- The verification task is also simplified since fewer modules will need verification.

4.4 The Round-, Guard- and Sticky-bit

The default rounding mode in the IEEE 754 standard is used for the implementation. In this rounding mode the result is rounded to the number that is closest to the infinitely correct result. Since we are working with radix 2 numbers there are only two rounding options: Cut off the low-order bits and round up by adding 1 ulp (unit in last position) or just cut off the low-order bits. If the result before rounding lies equally close to two possible rounded results, the number is rounded up if this makes the rounded result even. This is why the default rounding mode is sometimes also called Round to Nearest Even. Rounding is easy to understand by looking at a few examples. Assume all mantissas are 9 bits and are going to be rounded to 4 bits (For double and single the number of bits to round to would be 53 and 24 respectively).

1. 1.10001001 will round to 1.100.

   In this case the value is closest to 1.100.

2. 1.10010011 will round to 1.101.

   Here, the value is closer to 1.101 than 1.100, so we round up.

3. 1.10010000 will round to 1.100.

   Now the value is equally close to 1.100 and 1.101. Rounding up, however, would make the result odd so we do not round up.

4. 1.10110000 will round to 1.110.

   In this situation the value is equally close to 1.101 and 1.110. Rounding up produces an even result, so we round up this time.

To get correctly rounded results according to the Round to Nearest rounding mode for addition/subtraction, two extra bits must be appended to the mantissas before performing the addition/subtraction. These bits are called *Guard (G)* and *Round (R)*. The G bit is the bit left to the LSB of the mantissa. The R bit is positioned to the right of the G bit. Why these extra bits must be appended to the operands is described in [7, ch. 8.4.3] and will not be elaborated further here. In addition to appending these bits an implementation must also know when all the "shifted-out" bits are 0. The "shifted-out" bits are the bits that are discarded in the addition/subtraction due to alignment (shifting) of the mantissa. The bit-wise OR of all the shifted out bits is called the *Sticky Bit (S)*.

4.5 Module Descriptions

Figure 4.1 is a block diagram of the implementation. The names of the Verilog modules are written in parentheses. Modules or parts of modules that only contain combinational logic are dotted.

The complete Verilog RTL code can be found in appendix J.

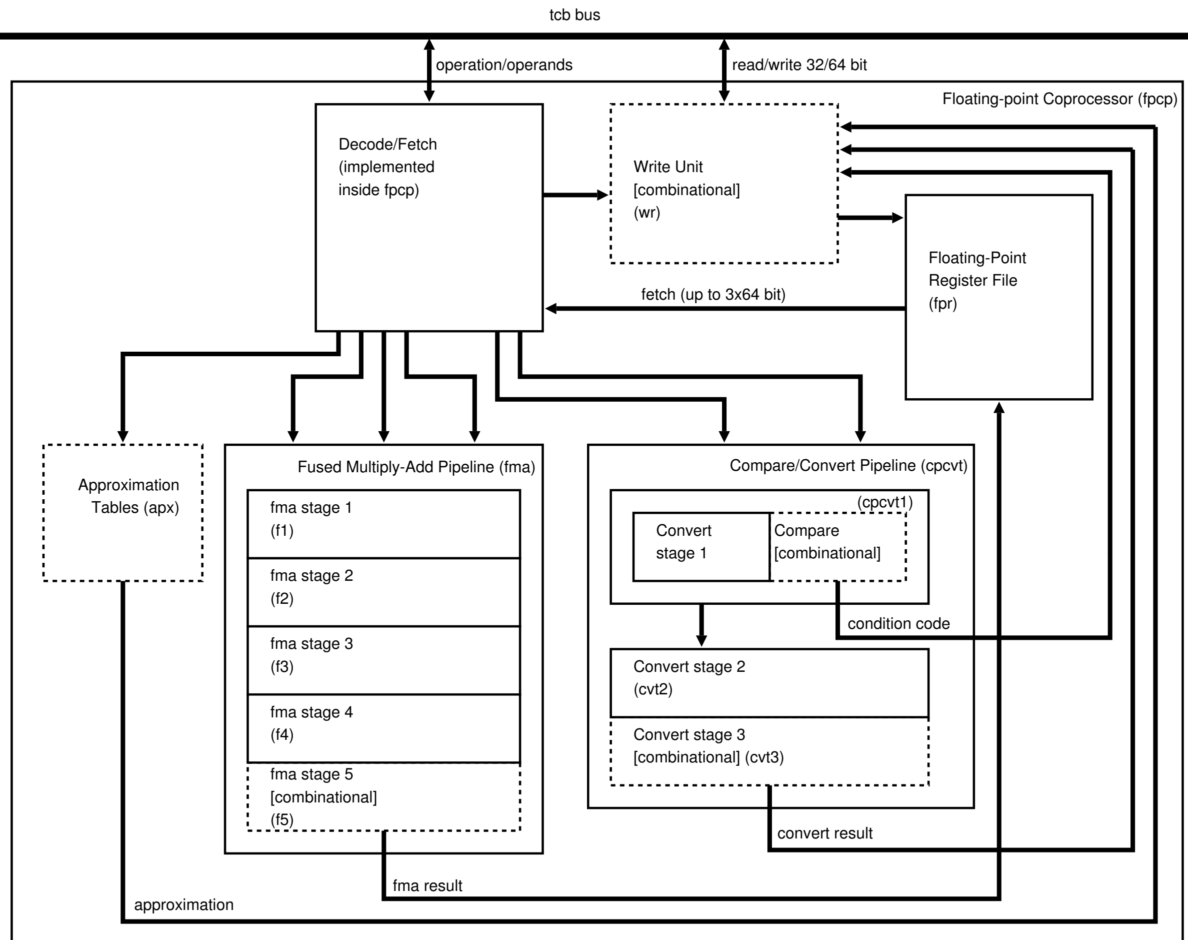

Figure 4.1: Floating-point Coprocessor Block Diagram

In the following subsections the different parts of the implementation are described.

4.5.1 Top Level and Decode/Fetch (fpcp)

The `fpcp` module is the top level Verilog module for the implementation. The `fpcp` module instantiates the `fma`, `fpr`, `cpcvt`, `wr` and `apx` modules and implements the decode/fetch logic. The decode/fetch logic is the control center for the coprocessor. Figure 4.2 shows the data-flow for the decode/fetch logic in the `fpcp` module. When the `tcb_cpno` signal on the TCB bus equals 0 (the number for the floating-point coprocessor) the command on the TCB bus is decoded and control signals are set to read data from the register file and check if any of the operands are locked. If some of the operands are locked or the operation could not start due to a predicted write-conflict the `tcb_ready` signal on the TCB bus is pulled low. The CPU will then stall and thus try the command again and again until the coprocessor sets the `tcb_ready`

high again. If the coprocessor is ready inputs are set to the FMA pipeline, the Compare/Convert Pipeline or the Write Unit/Approximation Tables. The Write Unit and the Approximation Tables share the data input registers. The FMA pipeline and the Compare/Convert pipeline does not share data input registers to save power. If they had shared data input registers this would have caused switching activity in both units even if only one is used.

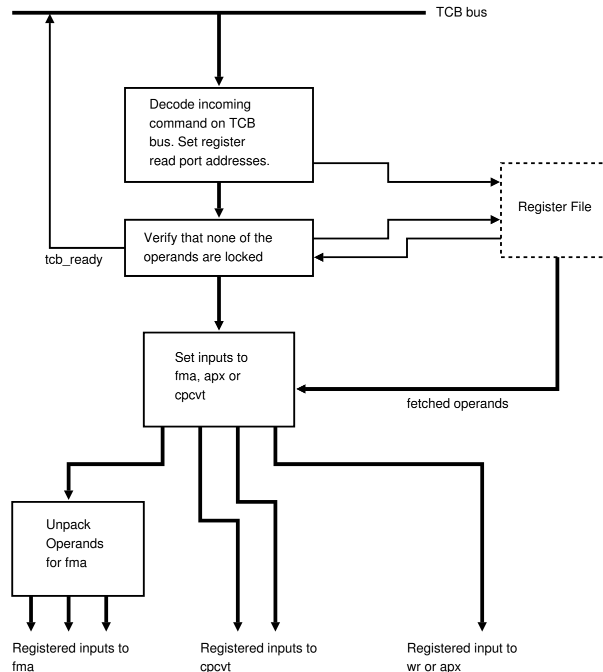

Figure 4.2: Decode/Fetch Data-flow in fpcp Module

The `fpcp` (and `wr`) together implement the interface to the Tightly Coupled Bus (TCB). Up to 8 coprocessors can be attached to the TCB. The CPU is the only master and initiates all operations for the coprocessor slaves. A coprocessor can however indirectly stall the CPU if it is not ready to reply to the request from the CPU. The TCB is also used to read and write the processor system registers. A detailed description of the TCB bus is provided in appendix C. Figure 4.3 shows how two coprocessors can be connected to the CPU with the TCB bus.

The commands to the coprocessor are decoded by first inspecting the MSB of `tcb_cmd` on the TCB bus. If this bit is 0, the rest of the bits (bit [6:0]) may specify any of 5 commands (described in appendix C):

- IDLE
- READ WORD
- READ DOUBLE WORD
- WRITE WORD

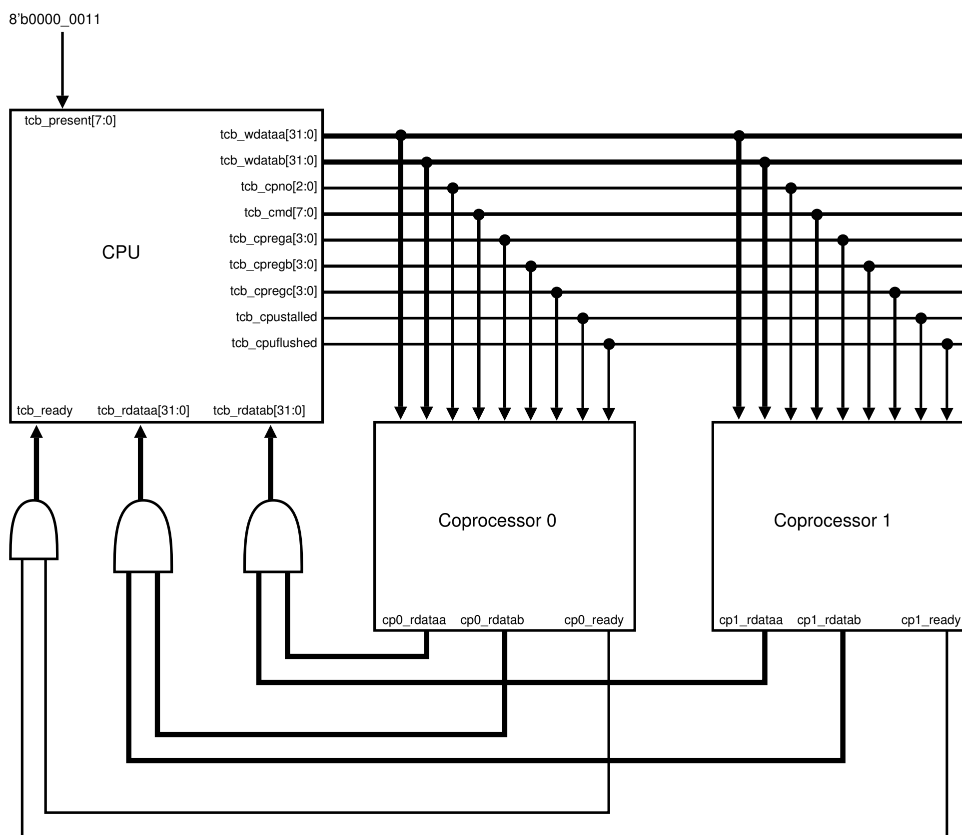

Figure 4.3: Coprocessor Interfacing Using the TCB

- WRITE DOUBLE WORD

All other encodings for the low order 7 bits are discarded.

If the MSB of `tcb_cmd` if 1, the low order 7 bits encode the coprocessor operation. For the convert instructions the LSB of the `tcb_cpregc` is used in addition to specify the rounding mode. The 7 command bits are used as follows:

Bit 6 is used to distinguish between single and double for instructions where this is needed. This bit is 0 for single and 1 for double. Bits 5 and 4 define the instruction *group* and the low order 4 bits are used to encode the specific instruction within that group. All instructions for the FMA pipeline have these bits set to "00". Move, absolute value, negate and load constant instructions are also in this group. When the group bits are "01", the instruction is a compare, convert or approximation instruction. When the group bits are "10" the instruction is **ffma** (Fused Multiply-Add) and the 4 low order bits are used to encode the destination register. When the group bits are "11" the instruction is **ffnma** (Fused Negated Multiply-Add) and the low order bits are used to encode the destination register. The specific encoding of each coprocessor operation/instruction is described in appendix B.

4.5.2 The Fused Multiply-Add Pipeline (fma)

The original Fused Multiply-Add pipeline presented in [30] used 3 stages and had a maximum clock frequency of about 83 MHz. Significant optimization effort has been devoted to make the the new pipeline synthesizable at 167 MHz (6.0 ns clock period). To be able to reach this goal without increasing the area, the pipeline length had to be extended. The new FMA pipeline therefore has 4 pipeline stages. The rounding step is moved down below the 4th pipeline stage and merged into the write-back stage. The original pipeline was a stand-alone core and did not have a write-back stage. Figure 4.4 shows the original vs. the new pipeline.

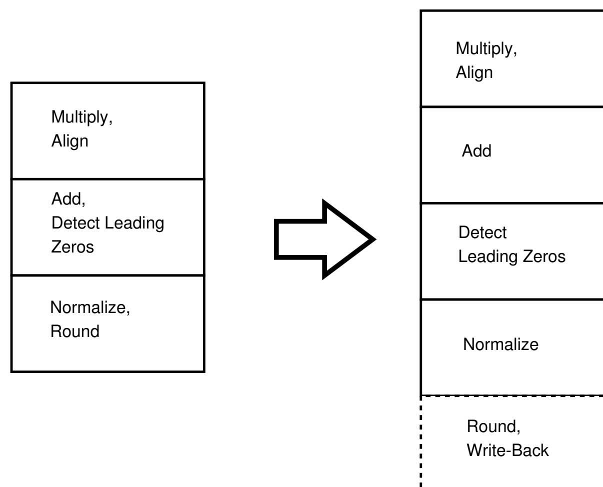

Figure 4.4: Original vs. New FMA Pipeline

As can be seen from the figure, the leading zeros detector and the normalization shifter have been put into separate stages. The leading zeros detection in the new pipeline also involves computation of the normalization shift amount for denormalized numbers. For denormalized numbers the shift amount is *not* equal to the number of leading zeros. In addition to the changes to the pipeline and the optimizations performed to meet the new timing requirements, the functionality of the new pipeline has also been extended.

- Is it now possible to negate both the product and the addend. This makes it possible to perform several new operations, for example *negated multiply*.

- The input- and output-format can be set to single or double. When the input-format is single the input is transformed to double internally in the first pipeline stage. When the output format is single, the output is rounded to single-precision. This makes it possible to perform single-precision operations in the new pipeline. It also makes it easy to perform conversion between single and double by simply setting the input and output format appropriately.

To save power, all registers now have enable-signals that allows them to be clock-gated. This reduces the power consumption significantly when the pipeline is not in use. Reset has also been removed for the pipeline registers. Only the flip-flop in each pipeline stage that stores the enable signal for the next pipeline stage need reset. The enable flip-flop is set to 0 when reset.

The dataflow for the new implementation of the FMA pipeline is shown in figure 4.5. The dotted lines represent pipeline registers. A detailed description of the FMA operation will not be given in this section, since the purpose of this chapter is to describe the implementation of the coprocessor as a whole. Instead the major steps in the computation will be described to give an overview. The implementation is based on the basic scheme presented in section 2 in [24] and the interested reader should consult this article to gain a better understanding of how the FMA pipeline works. More details can also be found in [30].

f1

The `f1` module performs multiplication of the two 53-bit mantissas for the two operands `op2` and `op3`. The mantissas are sent directly into the multiplier from the register outputs in the top level decode/fetch logic so the *hidden bit* is appended in the cycle before `f1` starts. The multiplication must be started immediately for the two partial products to arrive at the pipeline registers between `f1` and `f2` in time. The delay of the 53×53 bit partial product multiplier is 5 - 5.5 ns. The operand to add, *the addend* `op1`, is aligned to the 106-bit product from the multiplier in parallel with the multiplication. The total width of the aligned `op1` mantissa is 161 bits (53 + 2 + 106). The two extra bits that must be inserted between the `op1` mantissa and the product are the necessary Guard (G) and Round (R) bits described in section 4.4. The exponent for the product is also computed by adding the exponents of `op2` and `op3`.

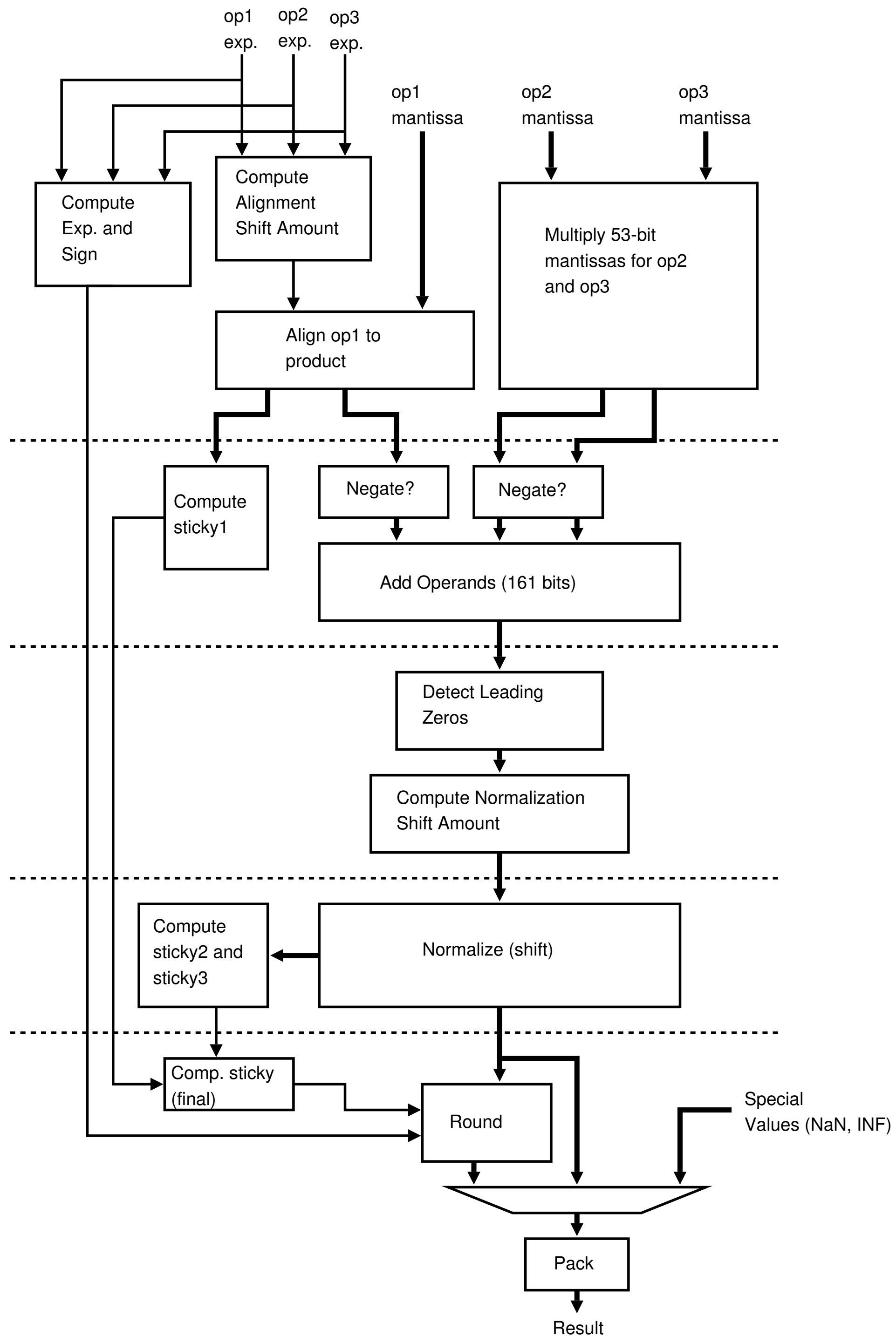

Figure 4.5: FMA Pipeline

f2

In the `f2` module, the operands may be negated before the addition takes place. Notice that all computations are performed in the one's complement system, so negation is performed by just inverting the number. Adding 1, as is done in the 2's complement system, is not needed. The possibly negated operands are then reduced to two by using a 106-bit Carry Save Adder (CSA). Notice that the upper 55 bits are not included in the CSA since these bits are all either 0 or 1 for the product. The addition is performed using two parallel 106-bit adders, a 56-bit incrementer and a 56-bit decrementer (a 106-bit carry chain is also needed). The two adders are needed since the one's complement addition requires the *carry out* to be added to get the result. Instead of waiting for the carry to propagate and then adding the *carry out*, the two adders compute both possibilities by setting the *carry in* to 0 for one adder and 1 for the other. Only one of the results is used based on the carry out from the adder with *carry in = 0*. After the operands have been added, the sum is negated (by inverting it) if it is negative before it is written to the pipeline register between `f2` and `f3`. The 53 shifted-out bits from the alignment shifter in f1 are also OR-ed together in parallel to get the partial sticky bit `sticky1`.

f3

In `f3` the number of leading zeros in the sum is detected. If a left shift will not cause the exponent to underflow, the normalization shift amount will be equal to the number of leading zeros. If the exponent underflows, a denormalized result will be produced and the sum might need to be shifted *right*. The output from this pipeline stage is the shift amount for the normalization shifter.

f4

In this stage the sum is normalized by a shifter. If the result is not a denormalized number, the normalization shift will place the first 1 in the MSB of the shifter output. Parts of the sticky bit computation is also performed in this stage.

f5

In this stage the final sticky bit is computed and the result is rounded according to the Round to Nearest rounding mode. A NaN or Infinity result may be selected if the operation is invalid or overflows respectively. The result is then packed to a single or double-precision number according to the IEEE standard.

4.5.3 Compare/Convert Pipeline (cpcvt)

The Compare/Convert pipeline performs compare and conversion from integer to floating-point. The Compare instruction is performed by the `cpcvt1` module and the major steps of the operation are depicted in figure 4.6.

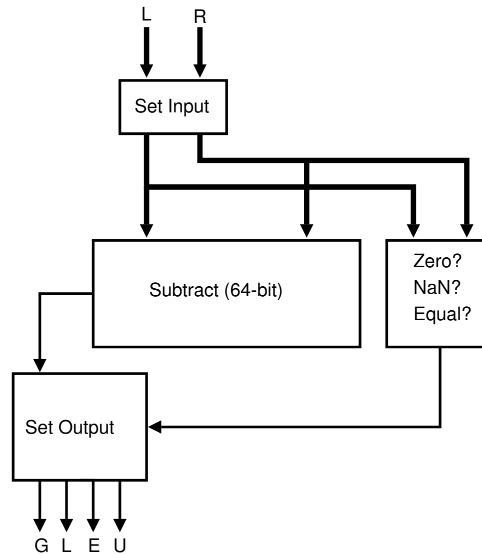

Figure 4.6: Compare Data-flow

The IEEE single and double formats were specificaly designed to simplify comparison. Using a bias that always makes the exponent positive and placing the exponent in front of the fraction allows floating-point numbers to be compared as signed-magnitude integers. The comparison can therefore be performed by a subtractor that computes the difference. The carry-out is used to decide if the result is greater or less. A NaN-, zero- and equal-check is also performed in parallel with the subtraction. The result, which can be any of Greater(G), Less(L), Equal(E) and Unorderd(U), is then selected based on the carry out from the subtractor and the check for special input values. The check for equality is performed in parallel with the subtraction to reduce the length of the critical path.

The conversion from integer to floating-point operation is described in figure 4.7.

The integer may be signed or unsigned and can be 32 or 64 bits wide. The first step is to negate the integer if it is negative. The negation is performed by the adder/subtractor in cpcvt1. This allows compare and convert instructions to share the adder/subtractor. Then the first level of a leading zeros detection is performed. In this step leading zero detection is performed in parallel on 8 groups of 8-bits each. Zero detection is also performed on the 8-bit groups. In the next pipeline stage (cvt2), the second part of the leading zeros detection is performed and the integer is shifted so that the first 1 is placed in the MSB of the output from the shifter. The exponent is then computed and the the result is packed to IEEE format. If the result needs to be rounded up this is performed in the next stage which also performs write-back.

4.5.4 The Approximation Tables Module (apx)

The reciprocal of a *normalized* floating-point number, D, can be written as follows:

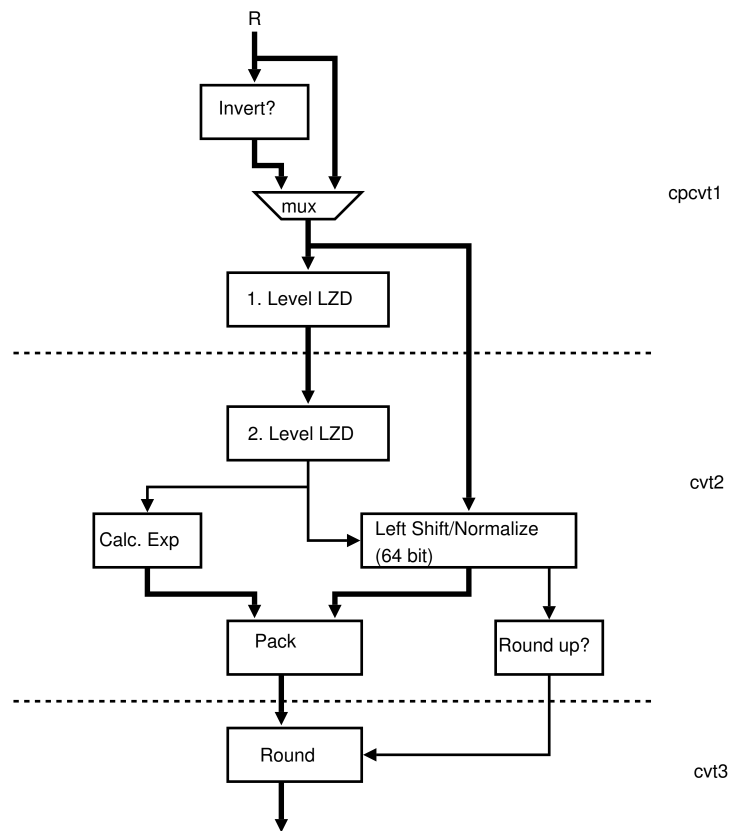

Figure 4.7: Convert Data-flow

$$\frac{1}{X} = \frac{1}{1.xxx..x \times 2^e} = \frac{1}{1.xxx..x} \times 2^{-e}$$

where e is the unbiased exponent and $xxx..x$ represents the fraction bits.

Since $0.5 < \frac{1}{1.xxx..x} \leq 1$, it will consequently have the form 0.1yyy..y for all fractions, $xxx..x$, except when the fraction is all zero bits. The result must therefore be shifted left one place to make the result normalized (1 in the MSB). Thus, the exponent e, must also be decreased by 1 if the input fraction is not equal to 000..0. The lookup table for reciprocal approximation takes the m MSBs of the fraction bits, $xxx..x$, as input and looks up the t MSBs of the result fraction $yyy..y$. If $t = m$ and the best value for the sub-interval is chosen, the error will be less than 0.5×2^{-m} [21]. There are 2^m sub-intervals. If we set $m = 4$, as is done in the implementation, the first sub-interval is $[1.000000...0, 1.000100...0)$, the next is $[1.000100...0, 1.001000...0)$ and the last is $[1.111100...0, 10.000000...0)$. The best value for a given sub-interval $[p, p + 2^{-m})$ is calculated by taking the average of the reciprocal at both endpoints as follows [21]:

$$0.5 \times \left(\frac{1}{p} + \frac{1}{p+2^{-m}}\right)$$

This value is put into the reciprocal lookup table.

For reciprocal square root the procedure is similar. The lookup table, however, must be twice as large. Half of the lookup table is used when the exponent of the input is even and the other half is used when the exponent of the input is odd. This is shown below:

$$\frac{1}{\sqrt{X}} = \frac{1}{\sqrt{1.xxx..x \times 2^e}} = \frac{1}{\sqrt{1.xxx..x}} \times 2^{-e/2}$$

When e is even it is divisible by 2, and $\frac{1}{\sqrt{1.xxx..x}}$ can be used. When e is odd, however, $\frac{1}{\sqrt{2 \times 1.xxx..x}}$ must be used instead. The LSB of the exponent is therefore also used to index the table. The lookup table entries are computed in a similar way as for the reciprocal, taking the average of the two endpoints. Notice that the reciprocal square root approximation will be in the range $(\frac{1}{\sqrt{2}}, 1]$, so the result will have to be normalized as for reciprocal.

The `apx` hardware module takes a divisor or radicand as input and produces an approximation to the reciprocal or reciprocal square root respectively. The dataflow is shown in figure 4.8.

First the single/double input is split into sign, exponent and fraction. The 4 most significant bits of the fraction is then used to index the reciprocal and reciprocal square root tables. The reciprocal table is indexed directly only using the 4 bits and returns the 4 most significant bits of the fraction for the reciprocal approximation. Thus, the reciprocal table has 16 entries of 4 bits each. The most significant bits of the reciprocal square root approximation also depend on whether the exponent is odd or even. Therefore the lookup table also needs the least significant bit of the exponent. The index used to index the reciprocal table is then a concatenation of the LSB of the exponent and the 4 MSBs of the fraction. The reciprocal square root table has 32 entries of 5 bits each. Only 4 bits are actually required, but adding a 5th bit makes the approximation a bit more accurate. The computation of the exponent for the approximation is carried out in parallel with the table lookup. For the reciprocal, the exponent is simply negated. For square root, the exponent must be negated and divided by 2. In both cases the exponents

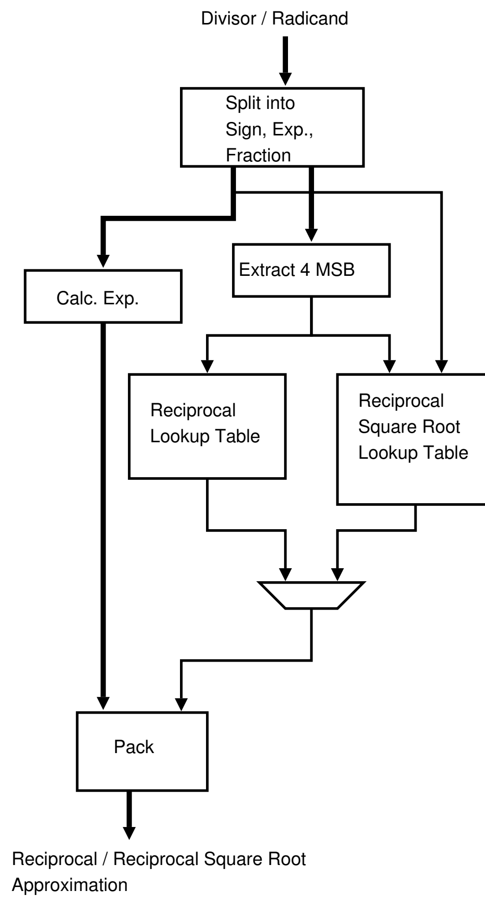

Figure 4.8: The apx Module

must then be decreased by 1 to adjust for the implicit re-normalization. In the end, the result is packed to the single/double IEEE format. The MSBs of the fraction is set to the bits returned by the appropriate lookup table and the rest of the fraction is filled with zeros. The `apx` module contains only combinational logic. Thus, the result is written directly to the register file on the next clock edge. The result may also be forwarded by the register file and used as an operand for next issued instruction. Notice that the current implementation sets certain limitations for the input to simplify the hardware.

- The input must be a *valid normalized number*.
- A zero input will produce wrong results, not infinity as expected.
- The reciprocal square root does not currently check that the sign of the number is positive. A negative number is therefore treated like a positive one.

The user must ensure that these conditions are met. The purpose of this implementation is to demonstrate how reciprocal and reciprocal square root approximation can be implemented. In an improved, future version of this module, all inputs should return correct results.

To build the lookup tables a utility program was written. The source code for this program can be found in the `appendixH/approx` folder on the appendix CD.

4.5.5 The Register File Module (fpr)

The register file serves several purposes:

- Store Floating-point numbers in registers.
- Automatically forward data to the read ports. The data is to be written to the registers in the next clock cycle.
- Keep track of locked registers. A locked register should not be read or written.
- Provide 2 lock ports. Each lock port can lock one 32-bit register in a given clock cycle.
- Automatically unlock registers when data is written to them.
- Provide 4 lock-check ports that can be used to check if any of the register operands to an instruction are locked.

The functionality of the `fpr` module is illustrated in figure 4.9.

There are 3 write ports, two 32-bit ports that can write to any of the 16 registers and 1 32/64-bit port that can write a 32-bit data value to one of the 16 registers or a 64-bit data value to a pair of 32-bit registers. In the latter case, the LSBs of the 64-bit value is written to the even register of the pair. Three combined 32/64-bit read ports are provided. Each read port has a lo and a hi 32-bit part. The hi part always outputs data from registers with odd address, while the lo part

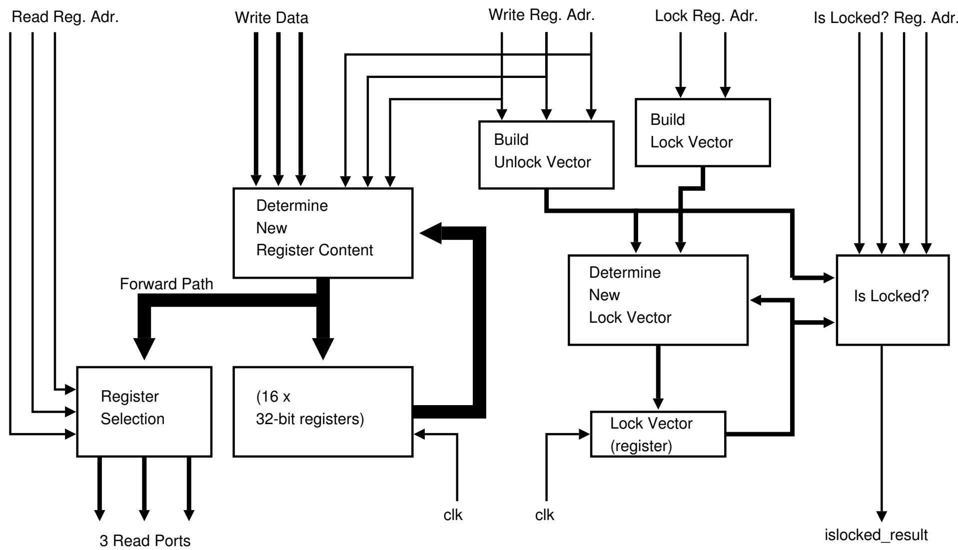

Figure 4.9: The fpr Module

may output data from even and odd registers. When the LSB of a read port address (used to index the register file) is 1 the value of the 32-bit register corresponding to this address is placed on the *lo* 32-bit port. The *hi* 32-bit port will then contain the same value, since the hi port always reads odd registers. This is OK since it is known that the *hi* part will not be used when the LSB of the address is 1 (only even addresses are allowed for double). When the LSB of the register address is 0 a pair of registers is always output on the *hi* and *lo* ports. To summarize, when reading single-precision 32-bit registers the *hi* part of the read port will not be zero in general and may contain garbage. By using this method a control signals for specifying single/double reads are not needed. The read port logic is shown in figure 4.10. The numbered boxes at the top of the figure represents the new register values (a combination of old and new forwarded values) that are going to be written into the register on the next clock edge.

A 16-bit Lock Vector Register is used to keep track of locked registers. When a bit 1 in this vector, the register corresponding to its index is locked. For example if the lock vector is 1000001000110010, it means that register 1, 4, 5, 9, and 15 are locked. Each cycle an unlock- and a lock-vector is constructed. Based on write-port addresses, unlock addresses and the current lock vector a new lock vector is then determined. Up to 4 lock-check ports are needed to check if any of the 4 registers used in the **ffma** and **ffnma** instructions are locked. If any of the registers specified by the lock-check ports are locked, the `islocked_result` will go high (and the operation will not start).

To simplify routing, the registers in the register file have no reset. The contents of the registers is therefore undefined when the microcontroller starts and the operating system should initialize all coprocessor registers to a known value. The Lock Vector, however, must be reset to 0. The registers in the register file are also automatically clock gated by the synthesis tool.

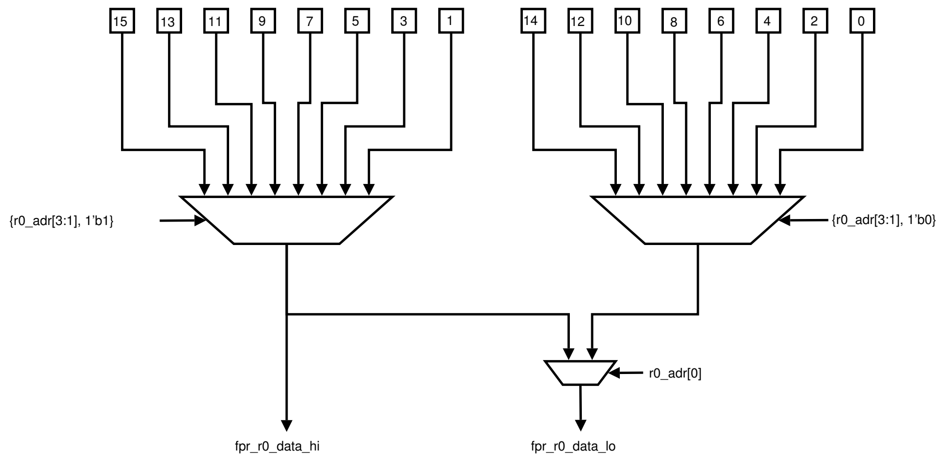

Figure 4.10: Read Port Logic

4.5.6 Write Unit (wr)

This is a very simple combinational module that writes data to the TCB bus in the data-phase of a READ operation and writes data to the register file from one of 4 sources:

- The TCB bus (data from the CPU after the address phase of a WRITE operation on the TCB bus).
- The apx module (the approximation of the reciprocal or reciprocal square root).
- The cpcvt1 module (the result from a compare).
- The cvt3 module (the result from an integer to floating-point conversion).

The data from each source is simply bitwise OR-ed together. The Decode/Fetch logic in the `fpcp` module must ensure that only one of the 4 sources writes data. All other three sources must write 0.

4.6 Unimplemented Functionality

- Conversion from floating-point to integer has not been implemented. This functionality should be added to the Compare/Convert pipeline. Round to Zero rounding mode should be supported for this operation to be compliant with the ISO C standard [20].
- The approximation instructions can produce wrong results for some inputs as was described in the section on `apx`.

4.7 Optimizing for Low Power Consumption

The coprocessor has not primarily been *designed* for low power consumption, but a few power-saving techniques have been applied to reduce the power consumption in parts of the design. Due to limitations in time I was not able to complete the low power optimization work. If low power-consumption had been a more important design-goal than performance, other trade-offs might have been made in several parts of the design. A lower precision may have been found sufficient, a separate multiply and add pipeline may have been used instead of a fused multiply-add operation and hardware resource sharing may have been utilized to a great extent. Often high-performance and low power-consumption are conflicting goals, so in most cases you will have to trade power-consumption for speed. Finding a perfect trade-off between power-consumption and performance can be a very complex engineering task and often requires careful analysis of different design-alternatives. This process therefore involves a lot of trial-and-error and can be quite time-consuming. In this section I will not go into detail on how the design could have been *designed* for low-power, but instead show how power-consumption is reduced in the design. The following discussion assumes the reader is familiar with basic theory on power consumption in CMOS technology. Information about power consumption can be found in [38].

4.7.1 Power-reduction Strategies

There are several design-techniques that can be applied to reduce power consumption:

- Reduce area.
- Reduce toggle-rate of nets.
- Reduce the critical path between pipeline registers.
- Design for clock-gating.
- Set synthesis constraints for low power-consumption.

By reducing the area a reduction in both static power and dynamic power can be achieved. Fewer transistors leads to less total leakage current and dynamic power is also reduced if the switching activity is not increased as a consequence of the area-reduction.

Reducing the toggle-rate of signals is a very important technique for reducing power consumption and be can applied using Verilog. The toggle-rate is the number of times a signal switches between 0 and 1 per unit time. As described in the previous section current flows from Vcc to Ground through the CMOS transistors for a short period of time during a switch, and power is consumed. By altering the contents of registers that drives logic only when necessary and multiplexing valid signals into logic blocks only when the output of the logic block is to be used, the total number of switches can be reduced.

Reducing the critical paths may also reduce the power consumption. The synthesis tool will try to optimize critical paths by doing more logic in parallel and inserting speed-optimized gates.

When more gates are inserted, more switching occurs. Since the speed-optimized gates use more transistors than regular ones this also increases the switching activity.

Clock-gating should be used for as many registers as possible to reduce power consumption. A clock-gated register only changes value when the clock into the register is enabled. For the synthesis-tool to always be able to insert clock-gating the clocking of the registers should have an explicit `enable` signal. Clock gating also reduces the fan-out of the clock network as described in the previous section.

The synthesis tool can be set up with low-power constraints. This is a quick and simple way to reduce power, but the quality of the result is of course limited by the given design and the power-reduction algorithms used by the tool.

4.7.2 Reducing Power in the Design

When trying to reduce power-consumption it is important to first identify the most power-hungry parts of the design. Reducing the effect of these modules will gain the largest over-all power-reduction.

The 53×53 partial product multiplier is the largest combinational unit in the coprocessor. To reduce switching activity, the inputs to the multiplier should therefore only change when the partial product results are needed in later cycles. To achieve this, *dedicated* registers for the operands are needed to drive the inputs. If the same register was used for driving both multiplier inputs and compare/convert unit inputs, both the multiplier and the compare/convert unit would consume power, even though only one of them is used.

The decode/fetch unit constantly listens to the TCB-bus for commands addressed to the floating-point coprocessor. The decoder will only perform decoding when the coprocessor number on the bus matches the floating-point coprocessor number. Most of the control signals from the decoder are only changed when necessary and are set to a constant value when unused. The synthesis tool will also be able to reduce power here by analyzing the RTL code.

Each pipeline stage in the FMA unit has an explicit enable signal. The pipeline registers will only be clocked when the enable signal is 1. This enable signal will make the synthesis tool able to append clock gating to these registers too.

Clock gating is also used extensively in all other modules to reduce power. The registers that drives inputs for the FMA- and Compare/Convert-unit for example are only clocked when a FMA- and Compare/Convert-operation respectively is to be started.

In the compare/convert unit the combinational inputs to some logic modules have been restricted to save power. The technique used is basically to stop the toggling signals before they enter a logic module when the output is known not to be used. This is done by multiplexing constants into the logic module or AND-ing the signal with a bit-mask. The latter method is illustrated in figure 4.11.

For this method to be most effective it is important that the enable-signal arrives no later than the data signals at the inputs of the AND gates. Otherwise, switching activity will spread down to the logic unit when the enable-signal switches from 1 to 0. Below follows a description of

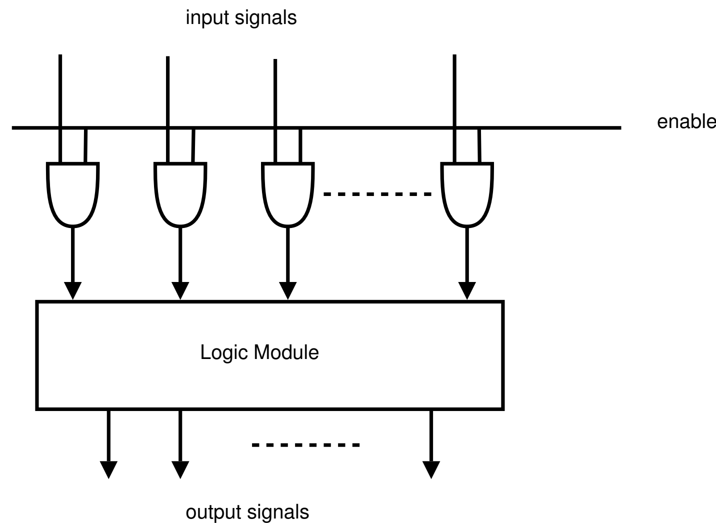

Figure 4.11: Saving Power by Restricting Inputs to Logic Modules

the places in the design where this technique has been applied.

- In the `cpcvt1` module the first level of a leading zeros detection on the `cpcvt1_ - unsigend_r_nxt` 64-bit value must be carried out when the operation is a conversion from integer to floating-point number. When the operation is not such an operation the `cpcvt1_ unsigned_r_nxt` signal is set to constant 0. Consequently when compare operations are executed only the multiplexer will consume power.

- The `cvt3` module rounds the floating-point number resulting from the integer to floating-point conversion. The rounding is done by incrementing the unrounded fraction by 1. If the incrementer overflows the exponent must also be incremented by 1. In the Round to Nearest scheme rounding is performed roughly 50% of the time. The situation when the incrementer overflows is very rare and happens only when all the fraction bits are 1 and the result should be rounded. The input signals to the incrementer is ANDed with `cvt2_rnd_up_r` signal. When `cvt2_rnd_up_r` is 0 all inputs signals to the incrementer will be constant 0 and the incrementer will not consume much power. When the `cvt2_rnd_up_r` signal is 1 the fraction bits are let through the AND gates and into the incrementer. The same technique is used for the exponent incrementer. All input signals to the exponent incrementer is set to 0 when the exponent need not be incremented.

It is important to notice that the additional AND gates or multiplexer in front of the logic modules will contribute to *increase* power consumption in the situations when the logic module *is* used (ie. are not fed with constant input). This additional power, however, is much less than the power saved by not switching any signals in the logic module when the module is not used.

4.7.3 Potential Improvements (not Implemented)

It is not necessary to perform the multiplication for addition and subtraction, since the product is known to be equal to the second operand. Further reductions in power consumption could potentially be achieved by allowing the second operand to bypass the multiplier when executing additions and subtractions. This requires a multiplexer to be inserted after the multiplier. This multiplexer selects between the partial products from the multiplier or the second operand. It is also possible to add a new 64-bit input operand that is only used in addition/-subtraction. This, however, is a very costly approach since it requires a new 64-bit register to hold the input signal in the Decode/Fetch unit. The method is illustrated in figure 4.12.

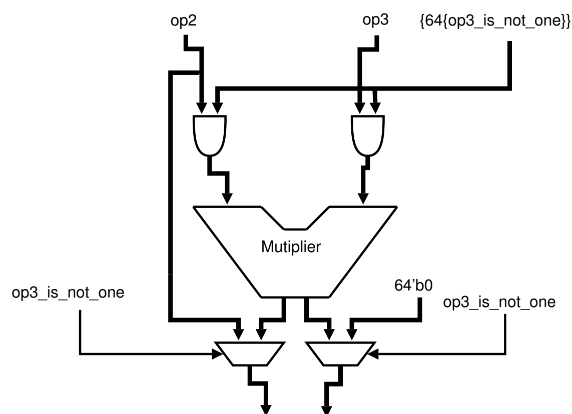

Figure 4.12: Bypassing Multiplier to Save Power

This was not implemented since the effective power reduction is very dependent on the mix of instructions. The savings achieved by disabling the multiplier for addition/subtraction can easily be canceled out by the additional power consumed by the AND-gates and multiplexers when the multiplier *is* used. Also the AND gates and multiplexer adds to the critical path in `f1`. A longer critical path will cause the synthesis tool to apply more aggressive optimization by inserting costly speed-optimized gates and perform more logic functions in parallel. This will *increase* the power consumption and could make it impossible to meet the timing requirements.

4.8 Synthesis

The implementation has been synthesized using Atmel's standard cell library for 0.18 μm process technology. The synthesis tool used was Design Compiler (DC) [33] from Synopsys.

A clock period of 6.0 ns (167 MHz) was used for synthesis. This is the same clock period as the CPU currently uses.

The implementation uses the same wireload model as the CPU. It is reasonable to assume that this will be OK since the coprocessor has a relatively regular structure. The wireload model enables the synthesis tool to estimate the effects of routing, without having to perform

a complete layout. The name of the wire load model used is “30KG”. If no wire-load model is specified in the synthesis script a pessimistic wire-load model is selected based on the gate-count of the design.

Input delays for all input signals to the top level coprocessor module are set to 1.0 ns. The output delay for `tcb_rdataa` and `tcb_rdatab` are both set to 3.0 ns and the output delay for `tcb_ready` is set to 2.0 ns. These input and output timing constraints affect the area of the resulting design since they define how quickly the coprocessor must be able to decode the incoming command and respond back to the CPU. The input delays are relative to the previous rising clock edge, while the output delays are relative to the next rising clock edge. To check that these input and output delays were appropriate the coprocessor was synthesized together with the CPU.

Clock gating is used for all modules and is inserted automatically by running a command in the synthesis script. Clock gating reduces the power consumption and fan-out for the clock tree. All registers that can be clock gated have explicit enable signals that allow them to be clock gated.

Scan flip-flops are used in the design, since a scan-chain will have to be inserted in a silicon implementation for testing purposes.

To gain the best possible results the following steps were followed:

- All compilation is performed using *DC ultra optimization mode*.
- Two improve synthesis results, two *incremental* compilations are performed after the first compilation.
- The map effort is set to *high*.

The synthesis script, synt.tcl, can be found in appendix F.

Chapter 5

Verification

This chapter describes the methods used to verify the coprocessor implementation. First the co-simulation method is described. The chapter then continues with a description of the test programs used. A short section on the verification of the FMA pipeline then follows. A description of an IEEE compliance checker that could be used during future verification work is also included. The chapter ends with a description of the utility programs developed during the project.

5.1 Co-simulation with the CPU

I have used a co-simulation methodology for verification, utilizing the pre-built test-suite available for testing other modules that are part of the new microcontroller. This test-suite makes it possible to run programs written in C or assembler in the RTL Simulator. Each test has it's own folder in the `<project number>/source/verification/` folder. The source code for the test program is located here. The test-folder also contains a small text file called `options`. This file specifies several options for the test (only the most important ones are described here):

- Which C compiler to use.
- What compiler and linker flags to use.
- Where to find include-files and libraries.
- Which libraries should be linked in.
- Whether the R12 register in the CPU should be checked against a value in the end of the test and what value R12 should have to signal that the test returned successfully. In a program written in C, the R12 register will contain the return-value when the program exits. This return-value is set by calling the `exit()` function from within the C program. This function takes the program return-value as argument. In an assembly program the value of R12 must be set explicitly.

- A watchdog timeout value can also be specified in the `options` file. The watchdog limit is specified in nano-seconds and causes the simulator to stop when simulation time exceeds this limit. This is a very important option since errors in the RTL code may cause the CPU to stall indefinitely or an error in the test program may cause the program to enter a never-ending loop.
- It is also possible to set an option that the contents of all registers should be compared against a cycle-correct C++ reference implementation once every clock cycle. Building a cycle-correct C++ implementation of the floating-point coprocessor was not a prioritized task in this project, so this option was not used.

The test-folder may also contain a Makefile for building libraries not placed in the test-folder that the test-program depends on.

To run a test, the RTL code for the CPU must be checked out from the CVS repository. The RTL code for the module and the test-folder for the test to run are then imported. The test is run be executing a script called `testone.sh` in the `/source/verification` sub-folder of the CPU RTL code. The script takes the test-folder as an argument and performs the following steps:

1. Builds the executable spec. This is the cycle-correct C++ model of the CPU used if the option for comparing contents of registers every cycle is enabled.
2. Use the Synopsys RTL simulator to build an executable simulator that also includes the RTL code for the module to test.
3. Compile and link the test program.
4. Run the executable simulator.
5. Check the value of R12 against the correct value specified in the `options` file (if this option is enabled).

It is possible to skip the first two steps if only the test program has been changed and no changes have been done to the RTL code. This is done by using the `-f` option when running the `testone.sh` script.

The `testone.sh` script may also be run with or without logging. Logging causes the simulation to take considerably more time but makes it possible to view the waveforms after simulation has completed. Easy access to all signals and information about how they change value over time is essential when tracking down bugs in a complex design like a floating-point coprocessor. Running without logging is used for long test-runs, verifying correct operation. Logging is specified with the `-l` option.

5.2 __builtin Macros

To send commands to the coprocessor from within programs written in C a few *builtin* macros have been used. The coprocessor number, `cpno`, used in these macros is always set to 0, since

the floating-point coprocessor is defined as coprocessor number 0. These macros are part of the ported C compiler and have not been implemented by me.

5.2.1 Moving Data into Coprocessor Registers

The `__builtin_mvrc_w(cpno, cpreg, value32)` makes sure that `value32` is kept in a 32-bit (word) register and generates a Move Register to Coprocessor Register instruction that moves the contents of this register into coprocessor register `cpreg`. The `__builtin_mvrc_d(cpno, cpreg, value64)` does the same with 64-bit integers (double words). `cpreg` must be even for this macro.

5.2.2 Moving Data from Coprocessor Registers

The `__builtin_mvcr_w(cpno, cpreg)` returns the 32-bit value of coprocessor register number `cpreg`. The `__builtin_mvcr_d(cpno, cpreg)` returns the 64-bit value of coprocessor register number `cpreg`. `cpreg` must in this case be even.

5.2.3 Sending Commands to the Coprocessor

To send commands to the coprocessor the `__builtin_cop(cpno, cprega, cpregb, cpregc, op)` is used. This macro inserts a `cop` instruction encoded with the parameters specified.

5.3 Test Programs

Several test programs have been written for verification. All the test programs can be found in the `appendixG/` directory on the appendix CD. Each test program has its own sub-directory. The floating-point coprocessor must be enabled before instruction can be issued to it. This is performed in all the test programs by a short sequence written in assembly.

5.3.1 `asm_test`

This was the first test program that was written for testing the coprocessor. The program tests several instructions by loading the register file with inputs and comparing the result to known correct results. Writing test programs in assembly takes time and makes the test programs less trustworthy. This is because the probability for introducing bugs in the test programs becomes higher when the complexity of the programming language used to write the test increases. The `asm_test` program is included in appendix G to demonstrate what assembly code for the coprocessor looks like.

5.3.2 `test`

The most important test-program is called `test`. This program was used extensively for finding bugs throughout the last stages of the project. The `test` program is based on random stimuli generation.

This test program uses the SoftFloat library [23]. SoftFloat is a program library written in C that implements the IEEE 754 standard. The package defines two types *float32* and *float64* that encodes single and double floating-point numbers respectively. The float32 is defined as an **unsigned int** and the float64 is defined as **unsigned long long**. The SoftFloat library is written by John R. Hauser and is freely available from the authors webpage [23].

At the heart of the `test` program is the `cop_test()` function. It has the following signature:

```
int cop_test(int op, int precision, int iter)
```

`op` specifies the operation/instruction to perform. `precision` is 0 for single and 1 for double. `iter` is the number of test iterations to perform.

During each iteration the following is performed:

1. Two floating-point numbers are selected at random.
2. These two numbers are loaded into the coprocessor register file.
3. The requested operation, `op`, is performed in software by calling the appropriate function in the SoftFloat library.
4. The requested operation, `op`, is executed in hardware (actually in RTL simulator) using the coprocessor.
5. The result from the coprocessor operation is then compared to the IEEE-correct result returned by the software implementation. If the results differ, the function returns 1. If both results are NaN and the encoding differs, the function will *not* return 1. This will happen occasionally, since the coprocessor only uses one encoding for all NaNs, while the SoftFloat library encodes NaNs in several different ways.

If all iterations complete successfully, the function returns 0.

The `cop_test()` function is run for the two-operand instructions ADD, SUB and MUL for both single and double. The one-operand instructions are also run through the `cop_test()` function:

- Convert Single to Double
- Convert Double to Single
- Compare (single/double)
- 32/64-bit *Signed* Integer to Single/Double

The SoftFloat library does not support the unsigned integer types so unsigned conversion was tested manually (not in the `test` program).

The `test` program has been run successfully with 200 iterations for each instruction.

The `test` program indirectly returns the value returned by `cop_test()` so the `options` file specifies that the R12 register should be equal to 0 for the test to succeed.

5.3.3 `test_fused`

The `test_fused` test program is used to test the *fused* three operand instructions:

- Multiply-Accumulate
- Multiply-Negate-Accumulate
- Multiply-Subtract
- Multiply-Negate-Subtract
- Multiply-Add
- Negated Multiply-Add

The program generates random stimuli similar to the `test` program, but is restricted to generating positive and negative integers between -1000 and 1000. The correct result is then computed based on the integers. Then the integer result along with the input operands are converted to floating-point representation using the SoftFloat library. The requested operation is then executed in the coprocessor and the result is checked against the correct converted integer result.

The `test_fused` program is very basic and only verifies that the three-operand instructions work and does not produce wrong results for floating-point integers. The reason that random integers are used and not random floating-point numbers is that the operations will return a slightly different result than a sequence with the two sub-operations would. The SoftFloat library can therefore not be used for verification. Further verification of the Fused Multiply-Add unit is described later in this chapter.

5.3.4 `test_approx`

The `test_approx` test-program implements the software division and square root functions based on the Newton-Raphson method. Only a few floating-point numbers for which the results were known in advance were used to test these functions. The test program therefore *only* serves to demonstrate that it is possible to use the Newton-Raphson method to obtain division and square root results which only differs from the correct result in the LSB. There was unfortunately not enough time to verify the correctness of division and square root within the time bounds for the project, but the results should be correct (except for the LSB) as long as the

initial approximation is correct to 4 bits. The functions in this program were used to measure latency and throughput for division and square root operations. Wave-forms from simulation of the functions in this test can be found in appendix L.

5.3.5 `cpuflushed_test`

The `cpuflushed_test` test-program forces the CPU-pipeline to be flushed due to a ldc.w (Load Word into Coprocessor Register) instruction. The pipeline is flushed since the memory access is to the unaligned memory address 0x00000001. This causes the CPU to raise an exception and the tcb_cpuflushed signal on the TCB bus to go high. The coprocessor should then abort the write operation on the TCB bus. This test was run successfully. The execution of the program must be examined in a wave-form viewer to verify correct operation.

5.3.6 `test_dotprod`

This program tests computation of the dot-product of two vectors of size 11 for both single- and double-precision. The `test_dotprod` program is used for demonstrating that the coprocessor can successfully perform a practical computation with no errors.

5.4 FMA Pipeline Verification

The arithmetic performed by the FMA pipeline has been verified by using the same test-vectors that was used for the tests described in [30, ch. 6.5]. These test-vectors do not cover all operations that can be performed by the new and improved FMA pipeline but running the test-vectors increases the probability for correct FMA operation. The test-vectors were applied to the FMA unit by a testbench that reads the input/output test-vectors from a file. Due to limited time for verification I was not able to write a program that produces new test-vectors for the FMA unit.

5.5 Floating-point Test Software

Verification of floating-point units has been an active area of research during past years due to the growing number of IEEE compliant floating-point implementations developed in the industry and the need for methods for verifying these inherently complex designs. Several articles on the subject have been published and some software packages for testing have been developed. These software packages are often freely available for use in academic as well as commercial projects. Below is a description of some of these test-software packages.

The IeeeCC754 software package can be used to check if an implementation is compliant to the IEEE-754 standard. It uses a set of predefined correct (input, output) pairs and checks these numbers against the implementation that is being tested. Unfortunately I was not able to run this test due to lack of time and the effort required to port the compliance checker. The

background for the testing philosophy in the IeeeCC754 Compliance Checker can be found in [37] and [36]. The Compliance Checker was developed at the University of Antwerp and the source code is available for free from [35].

5.6 Utility Programs

Several utility programs have been written to simplify debugging and verification. These programs are written in C and C++ and can be found in the `appendixH/` directory on the appendix CD. The following sections describe these programs.

5.6.1 `fdata`

The `fdata` program is a utility for converting between decimal floating-point number and the single and double formats represented with hexadecimal digits.

5.6.2 `fcvt`

This program can be used to convert between the single and double formats. It was used extensively when adding support for conversion between single and double.

5.6.3 `arith`

The `arith` utility program is used to perform various arithmetic operations. It was used extensively during testing of division and square root.

5.6.4 `approx`

The `approx` program produced the lookup tables for the `apx` module. How the values in the lookup tables are generated was described in chapter 4.

Chapter 6

Results

This chapter elaborates on the results. First, the performance of the architecture in terms of instruction latency and throughput will be discussed. The performance is evaluated by comparing the performance of the floating-point coprocessor implementation with the state-of-the-art FPUs described in chapter 2. The chapter then discusses functionality, other criteria for a good implementation and summarizes IEEE 754 compliance. The rest of the chapter presents the synthesis results.

6.1 Latency

The instruction latencies for the coprocessor instructions are compared to the ARM VFP9-S, MIPS32 24Kf FPU and PowerPC 603e FPU in figures 6.1 and 6.2. Figure 6.1 shows latencies for single precision instructions and figure 6.2 shows the latencies for double precision instructions. Latency numbers for division and square root are presented separately in figure 6.3. For all instructions it is assumed that the operands have been loaded into the register file. For division and square root it is also assumed that the constants used (0.5, 2.0, 3.0) have been loaded before the operation is started and that there is no function call overhead (ie. the function is inlined).

Low latency is important in instruction sequences with many data dependencies. A data dependency occurs when an instruction needs the result produced by another instruction. The coprocessor is designed to support low-latency loads, stores and compares. Load/store instructions were found to be executed very frequently [11].

Having only 1 cycle latency for loads allows the CPU to issue a coprocessor operation that uses the newly loaded value immediately during the next clock cycle. A low-latency load/store architecture for double precision is essential since the number of double-precision registers is limited to 8. In many programs, the CPU will have to swap data in and out of the register file very frequently. It is therefore important that the overhead associated with load/store operations is reduced to a minimum.

Store instructions should respond as quickly as possible, but the total latency of a store instruction depends very much on the cache architecture.

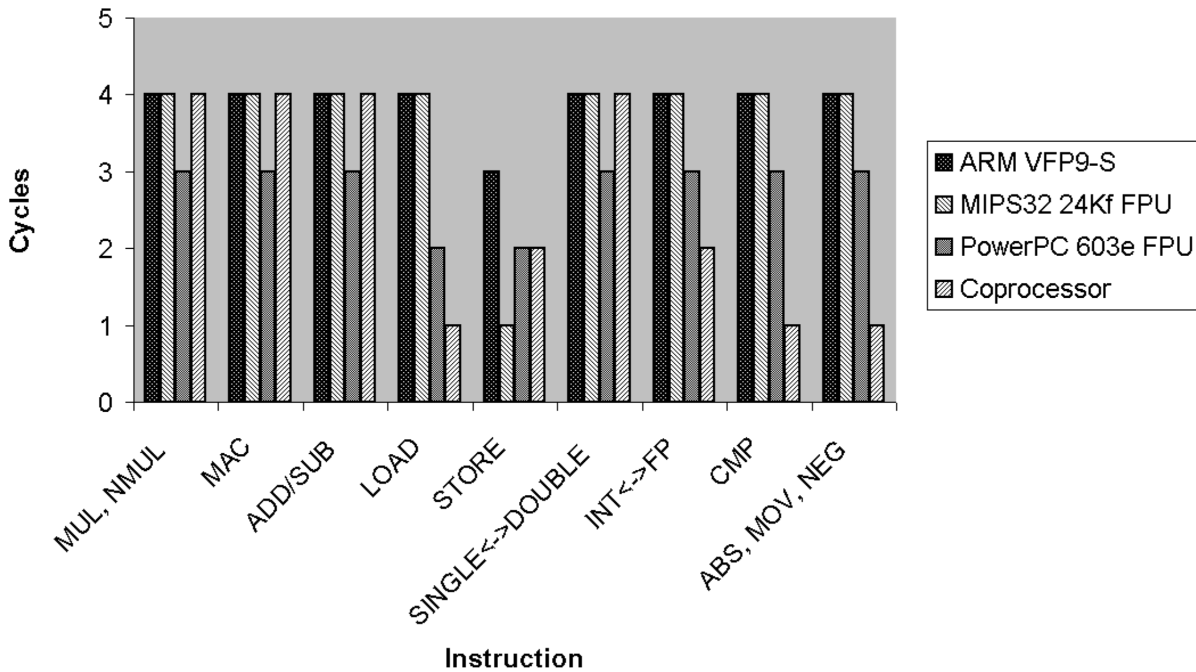

Figure 6.1: Instruction Latency Comparison (single)

Branches often depend on the outcome of a compare. The latency of compare is significantly lower compared to the other implementations. This will help to improve the performance of programs that performs many branches based on floating-point compare results. Computations in automotive control programs uses compare instructions extensively [5].

Absolute Value, Move and Negate operations are very simple operations and are executed efficiently in the coprocessor. The Negate instruction should not be needed very often since the instruction set supports negated versions of multiply and multiply-accumulate.

Conversion between integer and floating-point is also faster compared to the 3 competitive architectures.

For arithmetic operations and conversion between floating-point formats the latency is very similar for all the architectures. The PowerPC 603e is faster in general due to a shorter pipeline. Notice that double-precision multiply and multiply-accumulate take one more cycle on the ARM and MIPS architectures.

Figure 6.3 compares single- and double-precision division and square root latencies for ARM VFP9-S, MIPS32 24Kf FPU and the floating-point coprocessor implementation. The PowerPC 603e FPU is not compared due to its limited support for division and square root. The latency numbers for the coprocessor are measured by inspecting the execution of software implementations based on Newton-Raphson in the RTL Simulator and must be treated as rough estimates for a realistic implementation. The reasons for this is outlined below:

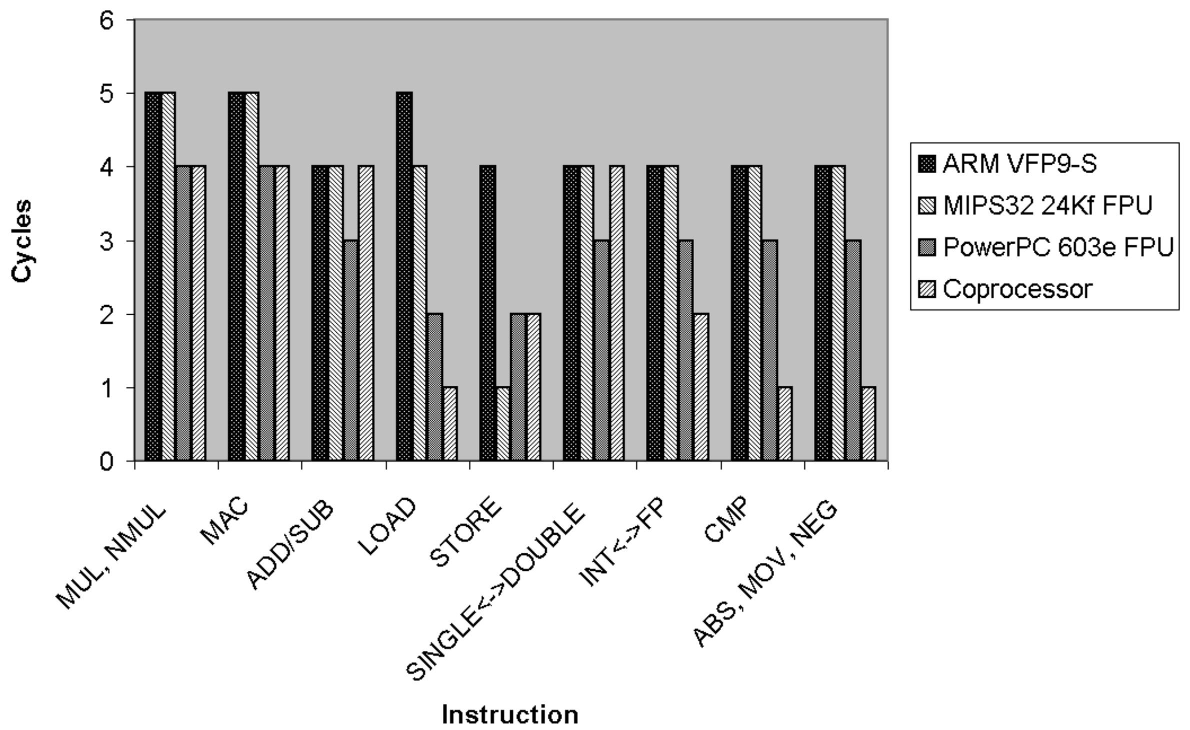

Figure 6.2: Instruction Latency Comparison (double)

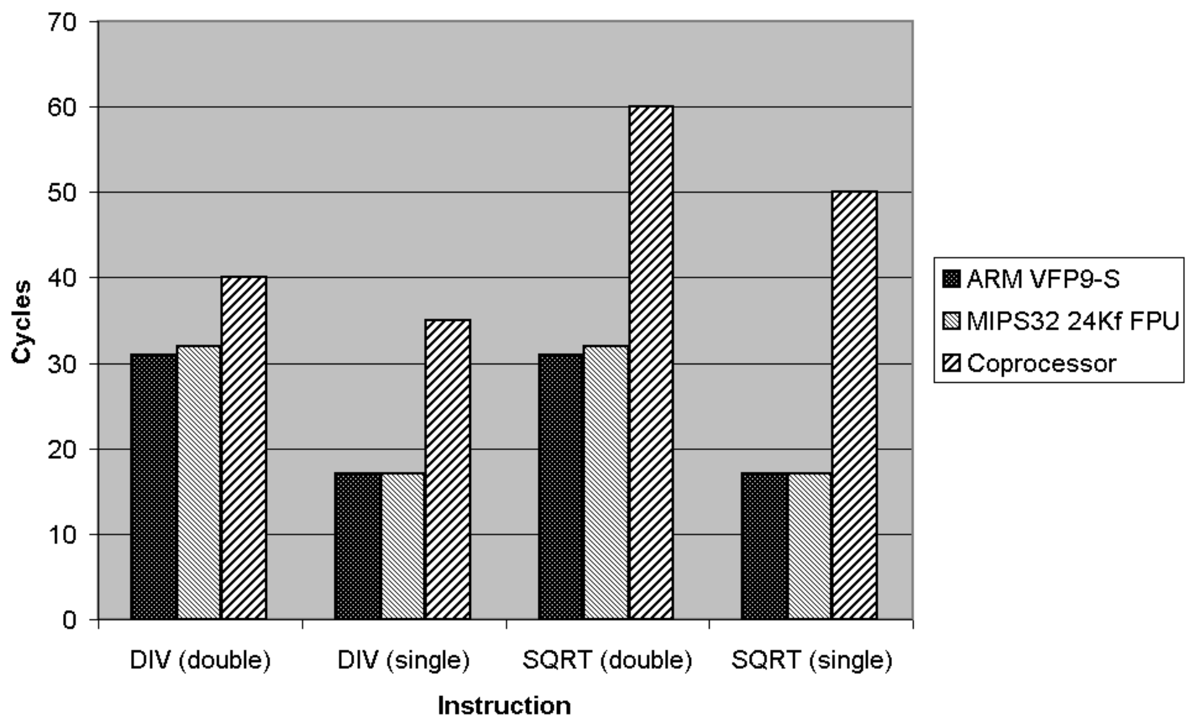

Figure 6.3: Division and Square Root Latency Comparison

- The software implementation is not optimized for low latency. An implementation that uses Goldschmidt's algorithm would have a lower latency.
- It is also important to keep in mind that the division and square root instructions for ARM and MIPS always return a correctly rounded result according to the Round to Nearest rounding mode, while the coprocessor may produce a result that is incorrect in the LSB. The results are therefore only valid for applications that does not require the result always to be correctly rounded.
- The latency of the software implementation will be longer when the instructions have not been loaded into the instruction cache. This will happen the first time the division/square root routine is executed and may also occur later if the routine is not used for a longer period. The other two architectures have a constant latency.

The software implementations of division and square root for the coprocessor generally have a longer latency compared to ARM and MIPS. This is to be expected since the algorithms are implemented in software and uses the longer-latency Newton-Raphson method. Double precision division achieves the best results, taking only about 10 cycles more than ARM and MIPS. This is because the advantage of the quadratic convergence rate is combined with a relatively short iteration time (compared to square root). Notice that the double precision software division algorithm has a latency that is only about 5 cycles more than for single. Thus, the execution time is increased by only 15 %. The other two architectures, on the other hand, use a digit recurrence method with a linear convergence rate. The execution time therefore almost doubles when going from single- to double-precision.

6.2 Throughput

Programs with a low degree of data dependence will benefit from the ability to pipeline instructions. Applications what processes blocks of data sequentially are often of this type. It is very simple to compare throughput numbers: ARM VFP9-S, MIPS32 24Kf and PowerPC 603e all have single-cycle throughput for all single-precision operations except for division and square root. For double-precision multiply and multiply-accumulate operations, however, an operation can only be started every second clock cycle. The other double-precision operations all have single-cycle throughput. Division and square root have very low throughput. *The coprocessor presented is a fully pipelined architecture with single cycle throughput for all arithmetic operations*. Throughput numbers for division and square root are presented in figure 6.4. Having single-cycle throughput also for double-precision operations makes it possible to achieve higher performance for double-precision multiply operations. There is one exception to the single-cycle throughput result. If a write-conflict occurs, single cycle throughput cannot be achieved. Write-conflicts only apply to instructions that do not execute in the FMA pipeline, and can only occur after an integer to floating-point conversion instruction.

The throughput numbers for the software implementations of division and square root must be considered "best case" since the time to load data into the register file is not included and

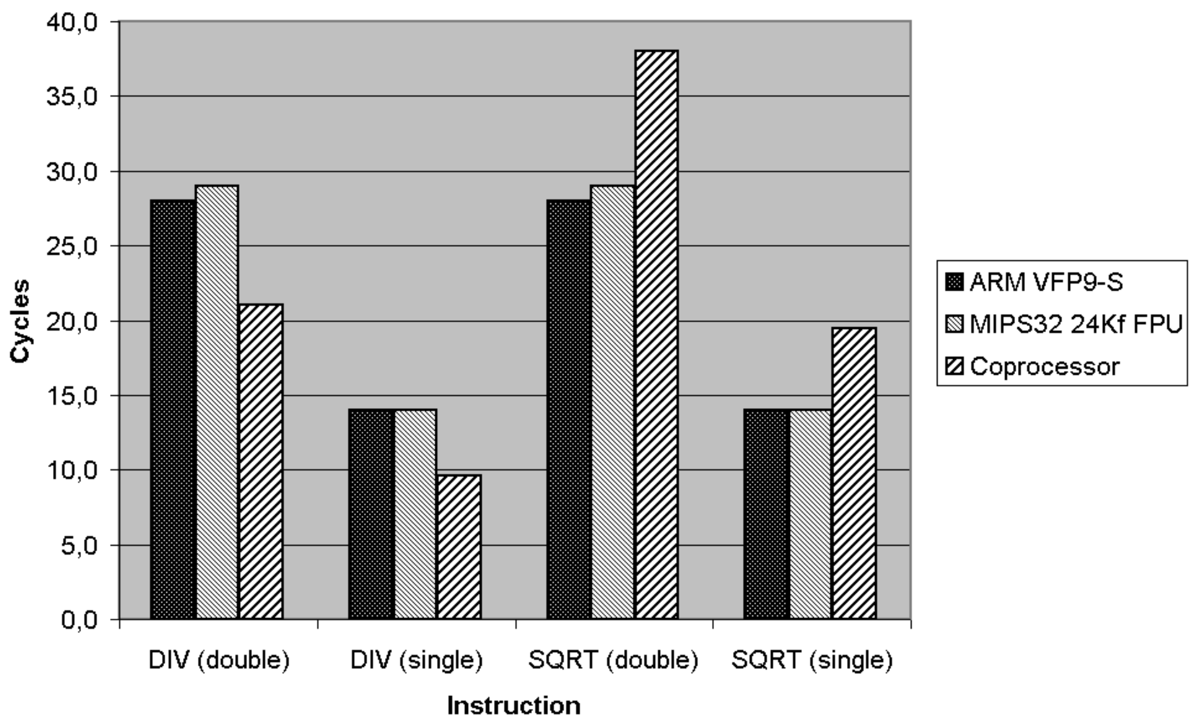

Figure 6.4: Division and Square Root Throughput Comparison

it is assumed that a maximum number of operations are always available for piplining. The throughput numbers were calculated as follows:

- Division (single): The number of cycles needed to compute 5 pipelined operations was measured. Then number was then divided by 5.
- Division (double): The number of cycles needed to compute 2 pipelined operations was measured. Then this number is divided by 2.
- Square Root (single): The number of cycles needed to compute 4 pipelined operations was measured. Then this number was divided by 4.
- Square Root (double): Similar to double precision division.

The throughput numbers for square root are about 30 % higher for the software implementations for the coprocessor.

To summarize of division and square root: The performance of the current implementations of division and square root for the proposed architecture are not better in general than the hardware versions provided by ARM and MIPS. There is, however, potential for improvements. By switching to the Goldschmidt algorithm, better results should be achieved. It is also possible to improve performance by increasing the accuracy of the initial approximation of the reciprocal and reciprocal square root at the expense of increased gate-count. The latter optimization will have a large impact on performance since the number of iterations can be reduced.

6.3 Functionality

The instruction set for the coprocessor is very similar to the instruction sets for the other architectures. This is reasonable, since all architectures implement the IEEE-754 standard. There are, however, a few differences:

- The multiply-add instructions in the MIPS and ARM instruction sets use an intermediate rounding, whereas the multiply-add instructions in the coprocessor architecture does not round the intermediate result.
- The instruction set has support for both 32- and 64-bit integer to floating-point conversion.

6.4 Performance of Typical Embedded Applications

The best way to evaluate the performance of the coprocessor implementation is to run typical embedded applications and measure the execution time. By also running the same applications on other competitive implementations, ie. ARM, MIPS and IBM PowerPC, the execution times can be compared. Benchmarks can also be used to evaluate performance and provides a very

simple way to compare the coprocessor against other implementations. It was unfortunately not enough time to perform this kind of evaluation at the end of the project. To be able to run a realistic benchmark the C compiler must be able to produce floating-point instructions for the coprocessor. This work would take several weeks to complete and would be outside the scope of this thesis. This is why only latency and throughput numbers are used to evaluate performance.

6.5 Other Criteria for a Good Implementation

There are also several criteria other than performance that can be used to evaluate the quality of the implementation:

- Can the implementation or parts of the implementation be reused in other designs?
- Is it possible to change the implementation easily by changing the values of parameters? It would, for example, be very nice to have a floating-point unit where the precision and number of pipeline stages could be changed by simply altering the values of parameters.
- How much work is needed to verify the implementation?
- It it easy to extend the design with new modules?
- Is is possible to synthesize the implementation? Does it require much effort/time to perform synthesis?

The FMA pipeline and compare/convert pipeline both have a simple interface and should be easy to reuse in other designs. The decoder is very specific to the floating-point coprocessor but the main structure of the code should be possible to extract and use in other coprocessors.

The implementation has not been designed for easy parametrization. This is because there are just two specific floating point formats. It should, however, be possible to generalize the design to increase the level of possible parametrization. It would be difficult to make the number of pipeline stages possible to change.

The FMA pipeline is inherently difficult to verify. There are 3 64-bit floating-point input numbers and 4 different operations. Choosing simpler two-operand units would make verification easier, especially if the inputs are restricted to single-precision only. Performance and support for double-precision, however, was considered more important.

Inserting new modules in the floating-point coprocessor is possible, but the number of write-ports in the register file may have to be increased.

It is possible to synthesize the implementation, but it currently requires significant effort from the synthesis tool to meet the timing at 6.0 ns clock period.

6.6 IEEE 754 Compliance Summary

The implementation can be described as *near-IEEE compliant*. All one- and two-operand operations except for division and square root will produce correct results according to the default rounding mode. The three-operand multiply-add operations are not yet part of the IEEE 754 standard so these operations should not be considered for IEEE 754 compliance. All the special values specified by the standard, +0, -0, +infinity, -infinity, NaN and denormalized numbers, are supported. Signaling NaNs, however, are not included. At least *one* signaling NaN must be supported for an implementation to be compliant. The implementations does not currently have instructions for converting floating-point numbers to integers. The standard also dictates that there shall be instructions for rounding a floating-point number to an integer (in floating-point format). These instructions are not supported, but should be easy to add if they are needed. Only the default rounding mode is supported in general. Integer to floating-point conversion instructions, however, may also round to zero. The exception flags are not included, but should not be very hard to add support for. Instructions for reading and clearing the exception flags must then be added. The remainder operation is also not included. The list of unsupported features may seen long, but I have focused on adding support for the most useful operations and features. Some features of the standard were also not found very important and thus excluded from the architecture to simplify the design.

6.7 Synthesis Results

This section presents the synthesis results for the implementation. A complete synthesis report from the synthesis at 6.0 ns clock period can be found in appendix K. The maximum clock frequency is 6.0 ns.

The area of the coprocessor top level module (fpcp) is 107 225 gates when it is synthesized with the required clock frequency of 6.0 ns. This is approximately one third of the area of the CPU.

The area of each individual module is shown in figure 6.5. The lower part of each column in the figure represent combinational area and the dark dotted areas in the upper parts represent non-combinational area (registers). The combinational modules are synthesized with the maximum delay from all inputs to all outputs set as small as possible. For all clocked modules a clock frequency of 6.0 ns are used. This gives a reasonable number for the area each module contributes to the total area. The area is measured in NAND gates (the smallest gate in the cell library). Since the Decode/Fetch logic is contained within the fpcp top module, the area of this logic had to be estimated. This logic should be (roughly) equal to the total area minus the area of all other modules.

The figure clearly shows that the 53-bits wide partial product multiplier, and shifter in f1 is costly in terms of combinational area. The large registers needed to store the two 106-bit partial products, the 161-bit aligned addend and the 53-bit shifted-out bits (for sticky bit calculation) also require a significant number of flip-flops. The f1 module alone contributes to 33 % of the total cell area.

The f2 module is the second largest sub-module in the FMA pipeline. One might initially

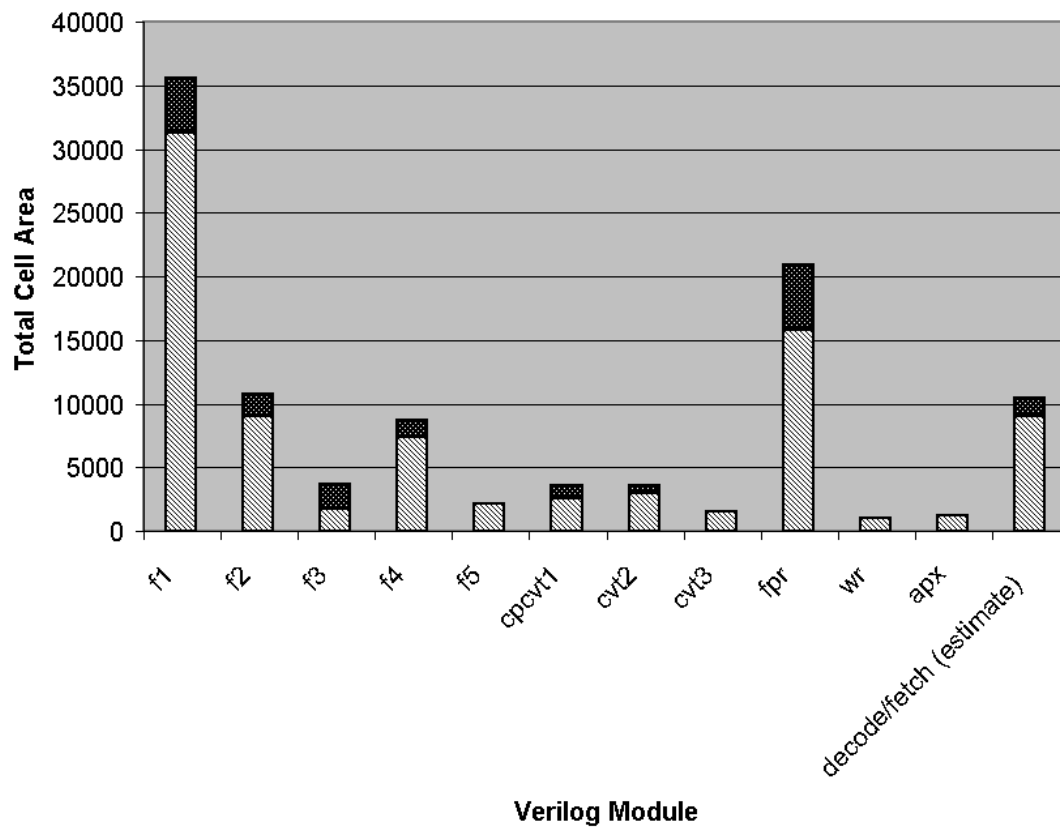

Figure 6.5: Total Cell Area of Individual Modules

think that a 106-bit CSA, two 106-bit adders, a 56-bit incrementer, a 56-bit decrementer and an additional 106-bit carry chain would contribute to most of the total cell area. The synthesis results, however, show that all this combinational logic takes less than 10 000 gates, only one third of the combinational area of f1.

f3 is quite small. This is reasonable, since the simple (but quite time-consuming) 161-bit leading zeros detector and the normalization shift-amount computation do not require much logic resources.

In f4, the wide two-way 214-bit normalization shifter contributes to almost all the area. It is interesting to notice that the area of this shifter is almost the same as the area of all the adders in the f2 stage.

The rounding in f5 requires only a few relatively small incrementers, so the area of this module is small compared to f1, f2 and f4.

The two first stages of the Compare/Convert pipeline, cpcvt1 and cvt2, are both roughly the same size. The rounding performed in cvt3 is simpler than that performed in f5 so the area is slightly smaller.

The register file, fpr, is the second largest module in the implementation. It is interesting to notice that the combinational selection logic required by the 3 read and write ports and the lock/unlock functionality contributes almost 74 % to the total cell area of the fpr module.

Notice also the very small area of the approximation tables in apx. This was reasonable to assume since the tables are rather small. This shows that the area-cost of adding support for division and square root is not very large.

Also notice that the non-combinational area is smaller in the end of the FMA pipeline compared to the first stages. This is because the number of bits for the intermediate results is being reduced. In f1, all 3 operands must be stored in registers. Since there is no time to calculate the sticky bit, the 53 shifted-out bits must also be stored in registers in f1. In f2, the operands are reduced to one 161-bit intermediate result so the non-combinational area is also reduced. In f3, nothing is done to the intermediate result, only the normalization shift amount is computed. The non-combinational area therefore does not change. In f4, however, the number of bits needed to compute the sticky bit in f5 are reduced by OR-ing groups of bits. This reduces the size of the register that stores the intermediate normalized result.

It is impossible to compare the area of the coprocessor to the MIPS32 24Kf and the PowerPC 603e FPU since the area information for these implementations is not available. ARM, however, has published an approximate gate count for the VFP9-S vector floating-point coprocessor. The gate count is 100 - 130K gates [2]. If one assumes a typical implementation of VFP9-S with a clock period of 6.0 ns (167 MHz) is 115K gates, the coprocessor uses 6 % fewer gates.

It is not completely fair to directly compare the VFP-S to the coprocessor implementation since the VFP9-S has twice as many registers and also includes an IEEE compliant divider. Taking into account that the coprocessor implementation uses a much larger single-pass multiplier, computes multiply-accumulate with better accuracy and provides better performance for load-/store operations, it can be expected that the area of the coprocessor will be large. It is also important to have in mind that the implementation is not completely optimized. During implementation, I discovered that reducing the delay of paths with only one or two logic levels

can have large implications for the total area of the implementation. In some situations the area was reduced by 2-3000 gates by just removing *one* logic level. The reason for these reductions is that other logic along the reduced path get a total increased slack in the timing requirements. It is also very important to keep in mind that the implementation is pushed to the limits with respect to timing when synthesized at 6.0 ns. The diagram in figure 6.6 shows how the area of the coprocessor implementation is reduced when synthesized at longer clock periods. Notice that the Total Cell Area axis begins at 80 000 gates.

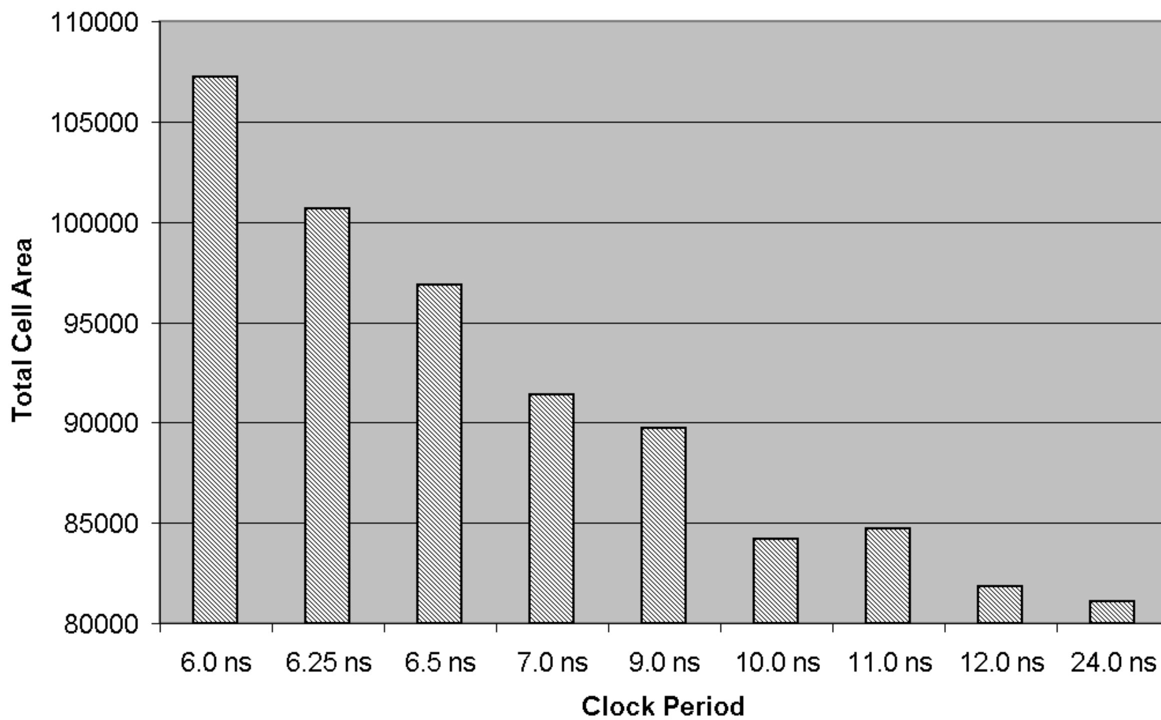

Figure 6.6: Total Cell Area at Different Clock Periods

As can be seen from the figure, the area grows quickly when the clock period approaches the maximum clock period of 6.0 ns. By just setting the clock period up to 6.5 ns (154 MHz) the area is reduced to 96 862 gates, a 10% reduction compared to the 6.0 ns clock period synthesis! It is only the *combinational area* that is reduced when the clock period is increased. The diagram also gives an indication for the maximum area-reduction potential. The area approaches about 81 000 gates when the clock period is increased beyond 12 ns. Thus, the optimizations DC performs to the basic logic implementation to meet the 6.0 ns timing constraint increases the area by approximately 25 500 gates (24 %).

6.7.1 The Effect of Clock Gating

The synthesis report shows that most registers are successfully clock-gated. When a register is clock gated, a multiplexer in front of the register is removed. This reduces the total area. Another nice consequence of this is that the timing requirements for all paths in the design are reduced since the signals do not need to propagate through the multiplexers. Then fan-out for the clock signal is also reduced since the clock inputs to the flip-flops are not driven directly by the clock. When the top level coprocessor module (fpcp) is synthesized without clock gating the timing requirements are *not* met. A slack violation of -0.02 ns is reported. When the clock period is increased to 6.1 ns, the timing is met but the area is about *2000 gates larger* than for the 6.0 ns clock gated version.

6.7.2 Synthesizing CPU and Coprocessor Together

The Coprocessor has also been synthesized together with the CPU. The primary motivation for doing this was to check that the timing constraints for the input and output signals for the TCB bus were met. It is very important that the coprocessor can be integrated along with the CPU and not only works stand-alone! The results from this synthesis were good. The critical path was shown not to be in the coprocessor.

Chapter 7

Future Work

Even though the implementation is almost complete and runs all the test-programs with no errors, there are many aspects of the implementation that could be improved. To be able to put the floating-point coprocessor on silicon several other tasks must also be performed. Future work for the floating-point coprocessor project is outlined below. Future work that is very important and should be performed first is marked with (1). Secondary future work that *could* be done is marked with (2).

- Instructions for converting floating-point numbers to integers should implemented. These instructions were not implemented simply due to limitations in time. It should be easy to incorporate this functionality into the Compare/Convert pipeline. If the shifter is changed to a two-way shifter it should be possible to share the shifter with integer-to-floating-point conversion. (1)
- A disadvantage of performing division and square root in software is that the CPU is kept busy during the division. This causes the coprocessor to operate less independently. An important extension of the coprocessor would therefore include the implementation of vector instructions that perform up to 5 single precision or up to 2 double precision divisions. The implementation will basically consist of a state-machine that implements the pipelined Newton-Raphson division algorithm. The reduced activity on the TCB bus due to performing the Newton-Raphson iterations in hardware independently inside the coprocessor will also save power. The additional hardware required (a state machine) will probably not increase the total area of the implementation much. Vector versions of square root operations should also be implemented in a similar manner as for divides. (2)
- The reason for the long latency of division and square root is very much due to the high degree of dependency between operations in each Newton-Raphson iteration. This makes it impossible to pipeline a single division/square root computation. Instead of using the Newton-Raphson method, Goldschmidt's algorithm could be used instead. This will reduce the latency since this method allows a higher degree of pipelining. (1)
- The floating-point instructions (in appendix B) should be included in the assembler. This should be a very simple task and will make it much easier to write programs in assembly.

(1)

- Add support for the floating-point instructions in the GNU GCC compiler. A careful analysis of how the GNU GCC compiler schedules floating-point instructions should then be performed. Optimization should then be applied until satisfactory code-quality is achieved. (1)

- Further verification is needed to make sure that the implementation always produces correct numerical results and that the coprocessor always responds correctly. It is particularly important to verify that the coprocessor cannot enter a state that causes the CPU to stall indefinitely. The verification work should not be done by me. Another person will most likely discover errors in the implementation that I have overlooked. An IEEE compliance checker should also be run to verify that all operations are IEEE compliant. The test programs that performs testing based on random stimuli for the operands could be run millions of iterations when executed in real-time on the FPGA. The already present UART could be used to signal when the result returned by the coprocessor and the SoftFloat library function does not match. The operands, the operation and the results must then be sent to the PC using the UART. (1)

- Benchmarks that measures floating-point performance should be run to ensure that the performance of the architecture is good enough in practice and achieves competitive results. (1)

- To increase the throughput of double-precision instructions it might be necessary to increase the number of double-precision registers to 16. Having 16 double-precision register will allow the compiler to produce code that achieves a higher degree of pipelining for double. It also makes it possible to fully pipeline 4 double-precision software division operations, which will improve throughput significantly. (1)

- A formal verification of the arithmetic, conversion and compare modules could be done to ensure they always produce correct results. Such an analysis should be performed if the floating-point coprocessor is going to be used in mission-critical applications. (2)

- Formally verify that the coprocessor always will perform correctly and never enter an incorrect state. For example that locked registers are never unlocked. (2)

- It should be possible to obtain a correctly rounded division and square root result using the multiplicative approach. This might require changes to the hardware. It might also be necessary to perform mathematical proofs to ensure a correctly rounded result is always returned. (2)

- A further analysis of the power-consumption should be carried out and the design should be optimized for low-power where such techniques have not already been applied. (1)

- Further optimization is also needed to reduce the area. I think it should be possible to reduce the area by 10-15% by carefully optimizing critical paths. By reducing critical paths the power consumption will also be improved since simpler gates can be used. (1)

Chapter 8

Conclusion

In this thesis a near-IEEE 754 compliant floating-point coprocessor has been implemented. The coprocessor is designed to be part of a new 32-bit microcontroller from Atmel and implements the generic coprocessor interface for this new microcontroller.

The architecture for the coprocessor was designed for high performance and provides single-cycle throughput for all operations except for division and square root. An optimized and extended version of the fused multiply-add pipeline that was developed in the specialization project fall 2004 performs the arithmetic operations. A new combined compare/convert pipeline has been developed to provide support for compare and conversion from integer to floating-point formats.

Software implemented algorithms based on Newton-Raphson approximation have been used for division and square root. These algorithms take advantage of the fused multiply-add instruction to reduce the latency. Reciprocal and reciprocal square root approximation instructions have been included in the architecture to support the software implementations of division and square root. Unfortunately, the division and square root operations does not always produce a correctly rounded result. A register file with 16 32-bit registers is used. The register file can store up to 16 single-precision numbers or 8 double-precision numbers.

Some parts of the IEEE 754 standard has not been implemented. Only the default NaN is used and signaling NaNs are not supported. The exception flags have also not been implemented and only the default rounding mode is supported.

The Verilog implementation of the architecture has been synthesized at a clock frequency of 167 MHz using the standard cell library for Atmel's 0.18 μm process technology. This is the maximum clock frequency and equivalent to the current clock frequency the microcontroller uses. The total area of the floating-point coprocessor is 107 225 gates when synthesized at maximum clock frequency. This is about the same as for the VFP9-S vector floating-point coprocessor from ARM and about one third of the area of the CPU pipeline.

The coprocessor has not been designed for low power consumption, but the implementation tries to reduce the power consumption as much as possible.

The only operation specified in the architecture that has not been implemented is conversion

from floating-point number to integer. This operation was excluded from the implementation only due to limitations in time and should be easy to add to the implementation.

Verification has primarily been performed by co-simulation of the CPU and the coprocessor. To verify that the implementation functions correctly, test- programs were developed and run on the CPU with the coprocessor. All the tests have been completed successfully. Unfortunately, there was not enough time to perform a thorough verification of the coprocessor.

The latency and throughput for most operations except division and square root are similar to the ARM, MIPS and PowerPC architectures. The load instructions for the coprocessor, however, have only 1 cycle latency. This is 3-4 cycles less than ARM and MIPS. The compare instruction also has lower latency. The resulting floating-point coprocessor also provides single-cycle throughput for all *double-precision multiply* instructions. This is twice as good as the floating-point units from ARM, MIPS and IBM (PowerPC).

The latency for division and square root is not good enough, but could be optimized by using Goldschmidt's algorithms instead of Newton-Raphson. The throughput, however, is very good to be a software-only implementation. A test-program shows that up to 5 single-precision division operations can be pipelined for maximum throughput.

Personally, I am quite satisfied with the resulting coprocessor implementation. The coprocessor performs most of the operations specified in the IEEE 754 standard very efficiently and all instructions work well. There is, of course, potential for improvement, but I feel that a good solution that fulfills the requirements to a large extent has been obtained.

8.1 Project Experiences

During the project I learned that designing and implementing a floating-point coprocessor is a very large and complex task. Parts of the implementation is therefore simplified and not all aspects of the IEEE 754 standard are implemented. I have also learned that verification of hardware takes very much time and requires significant effort. During optimization I also learned how to reduce critical paths. I feel that I have gained important experience that will be very useful in future projects.

Bibliography

[1] ARC International, http://www.arc.com/configurablecores/fpx/. *ARC^{TM}FPX Floating Point Extensions*.

[2] ARM, http://www.arm.com/products/CPUs/VFP9-S.html. *VFP9-S Product Website*.

[3] ARM, http://www.arm.com/pdfs/VFP-S_Vector_Floating_Point_Tech_Manual.pdf. *VFP9-S Vector Floating-point Coprocessor Technical Reference Manual*.

[4] R. Brent and H. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, C-31(3):260–264, 1982.

[5] Daniel Connors A., Yoji Yamada, and Wen-mei Hwu W. A Software-Oriented Floating-Point Format for Enhancing Automotive Control Systems. In *Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES98)*, December 1998.

[6] The Embedded Microprocessor Benchmark Consortium, http://www.eembc.com/. *EEMBC Benchmarks*.

[7] Milos D. Ercegovac and Tomas Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2003.

[8] Falanx, http://www.falanx.com/product.html. *Mali(tm) Graphics Solution*.

[9] Richard Gerber. *The Software Optimization Cookbook*. Intel Press, 2002.

[10] Robert E. Goldschmidt. Applications of Division by Convergence. *MSc. dissertation, M.I.T.*, 1964.

[11] P.L. Harrod, A.J. Baum, J.P. Biggs, D.W. Howard, A.J. Merritt, H.E. Oldham, D.J. Seal, and H.L. Watters. FPA10-A 4 MFLOP floating point coprocessor for ARM. In *Proceedings of the IEEE 1993 Custom Integrated Circuits Conference*, May 1993.

[12] IBM, http://www-306.ibm.com/chips/techlib/techlib.nsf/ products/PowerPC_603e_Microprocessor. *PowerPC 603e and EM603e RISC Microprocessor Family User's Manual*.

[13] IBM, http://www-306.ibm.com/chips/techlib/techlib.nsf/ products/PowerPC_603e_Microprocessor. *PowerPC 603e RISC Microprocessor Technical Summary*.

[14] IBM, http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_603e_Microprocessor. *PowerPC EM603e and 603e Product Brief*.

[15] IBM/Apple, http://www.apple.com/g5processor/. *PowerPC G5 White Paper*.

[16] IEEE Computer Society, http://754r.ucbtest.org. *DRAFT Standard for Floating-Point Arithmetic P754*.

[17] IEEE Computer Society, http://grouper.ieee.org/groups/754/revision.html. *IEEE 754 Revision Group Webpage*.

[18] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[19] Intel, http://www.intel.com/design/itanium/manuals/iiasdmanual.htm#application. *Intel Itanium Architecture Software Developer's Manual*.

[20] International Organization for Standardization, http://www.iso.org. *ISO C Standard*.

[21] Masayuki Ito, Naofumi Takagi, and Shuzo Yajima. Efficient Initial Approximation and Fast Converging Methods for Disision and Square Root. In *12th IEEE Symposium on Computer Arithmetic*, July 1995.

[22] Romesch Jessani M. and Michael Putrino. Comparison of Single- and Dual-Pass Multiply-Add Fused Floating Point Units. *IEEE Transactions on Computers*, 47, September 1998.

[23] John R. Hauser, http://www.jhauser.us/arithmetic/SoftFloat.html. *SoftFloat Library*.

[24] Tomas Lang and Javier D. Bruguera. Floating-point multiply-add-fused with reduced latency. *IEEE Transactions on Computers*, 53, August 2004.

[25] Peter Markstein. Software Division and Square Root Using Goldschmidt's Algorithms. In *6th. International Conferance on Real Numbers and Computing*, November 2004.

[26] MIPS Technologies, http://www.mips.com. *MIPS Website*.

[27] MIPS Technologies, http://www.mips.com (requires registration). *MIPS32 24Kf Processor Core Datasheet*.

[28] MIPS Technologies, http://www.mips.com (requires registration). *MIPS64 Architecture For Programmers. Volume I.*

[29] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC(7):218–222, 1958.

[30] Kristian Skogstrøm. *TDT4720 Project Report: Design and Implementation of Arithmetic Core for Floating-point Coprocessor*. Department of Computer and Information Science, NTNU, November 2004.

[31] Sun Microsystems, http://docs.sun.com/source/817-5073/index.html. *Numerical Computation Guide, Appendix D, Differences Among IEEE 754 Implementations*.

[32] Synopsys, http://www.synopsys.com/products/designware/buildingblock.html. *DesignWare Building Block IP*.

[33] Synopsys, http://www.synopsys.com/products/logic/design_compiler.html. *Synopsys Design Compiler*.

[34] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quarterly J. Mech. Appl. Math.*, 11(3):364–384, 1958.

[35] University of Antwerp, http://www.win.ua.ac.be/ cant/ieeecc754.html. *IEEE 754 Compliance Checker*.

[36] B. Verdonk, A. Cuyt, , and D. Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic ii: conversions. *ACM TOMS*, 27(1):119–140, 2001.

[37] B. Verdonk, A. Cuyt, and D. Verschaeren. A precision- and range-independent tool for testing floating-point arithmetic i: basic operations, square root and remainder. *ACM TOMS*, 27(1):92–118, 2001.

[38] Neil H.E. Weste and David Harris. *CMOS VLSI Design - A Circuits and Systems Perspective*. Addison Wesley, third, international edition, 2005.

[39] www.validlab.com, http://www.validlab.com/754R. *Some Proposals for Revising ANSI/IEEE Std 754-1985 http://754r.ucbtest.org*.

[40] Xilinx, http://www.xilinx.com/products/virtex4/capabilities/powerpc.htm. *PowerPC RISC Processor*.

[41] Xilinx Inx., http://www.xilinx.com/ipcenter/processor_central/microblaze/microblaze_fpu.htm. *MicroBlaze v4.00 Floating Point Unit*.

[42] The Xiph.org Foundation, http://www.xiph.org/ogg/vorbis. *The Ogg Vorbis CODEC project*.