View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Trepo - Institutional Repository of Tampere University

TAMPEREEN TEKNILLINEN YLIOPISTO

TIMO VIITANEN

FLOATING-POINT ARITHMETIC IN TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Jarmo Takala and Pekka Jääskeläinen, M.Sc.
Examiners and topic approved in the Computing and Electrical Engineering Faculty Council meeting 7.11.2012

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TIMO VIITANEN: Floating-Point Arithmetic in Transport Triggered Architectures
Master of Science Thesis, 48 pages, 2 Appendix pages
December 2012
Major: Embedded Systems
Examiner: Prof. Jarmo Takala and Pekka Jääskeläinen, M.Sc.
Keywords: floating-point unit, application-specific instruction set processor, FPGA

Many computational applications have high performance and energy-efficiency requirements which "off-the-shelf" general-purpose processors cannot meet. On the other hand, designing special-purpose hardware accelerators can be prohibitively expensive in terms of development time. One approach to the problem is to design an Application-Specific Instruction set Processor (ASIP), which is programmable, but tailored for the task at hand. The process of customizing an ASIP requires heavy automation to be cost-effective.

The TTA-based Codesign Environment (TCE) is an ASIP design toolset based on the highly flexible Transport Triggered Architecture (TTA) processor model, which scales from simple low-power cores up to high performance Very Long Instruction Word (VLIW) processors. Hardware accelerated support for floating-point arithmetic is necessary for many applications in the fields of scientific computation and digital signal processing, which would especially benefit from the scalability and instruction-level parallelism of TTA.

This thesis introduces a comprehensive suite of Register Transfer Level (RTL) implementations of floating-point units designed and implemented for the TCE project. The main design requirements were portability and performance on Field-Programmable Gate Array (FPGA) platforms even at the cost of reduced standards compliance. The suite includes an option for half-precision arithmetic. In addition, this thesis proposes fast software floating-point division and square root algorithms based on special instructions.

The implemented units were verified on the register transfer level using an automated test bench. When benchmarked on an Altera Stratix-II FPGA, the units exhibited performance close to the highly optimized units supplied by Altera, while retaining platform independence. On more recent FPGAs such as the Xilinx Virtex-6, finer-grained pipelining is required for maximum performance.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TIMO VIITANEN: Floating-Point Arithmetic in Transport Triggered Architectures
Diplomityö, 48 sivua, 2 liitesivua
Joulukuu 2012
Pääaine: Sulautetut järjestelmät
Tarkastajat: prof. Jarmo Takala, DI Pekka Jääskeläinen
Avainsanat: liukulukuyksikkö, sovelluskohtainen käskykantaprosessori, FPGA

Laskentajärjestelmiin kohdistuu usein suorituskyky- ja virrankulutusvaatimuksia, joita ei pystytä saavuttamaan yleiskäyttöisellä prosessorilla. Toistaalta laitteistokiihdyttimien suunnittelu voi vaatia kohtuuttoman paljon työaikaa. Ongelmaa voidaan lähestyä käyttämällä sovellusta varten räätälöityä sovelluskohtaista käskykantaprosessoria (Application-Specific Instruction set Processor, ASIP), joka on kuitenkin ohjelmoitava. Prosessorin räätälöinnin täytyy olla pitkälle automatisoitua säästääkseen kustannuksia.

TTA-based Codesign Environment (TCE) on siirtoliipaistuun prosessoriarkkitehtuuriin (Transport Triggered Architecture, TTA) perustuva ASIP-kehitysympäristö. TTA on arkkitehtuurina helposti räätälöitävä ja joustaa pienistä ytimistä suuritehoisiin pitkän käskysanan suorittimiin. Useat tieteellisen laskennan ja signaalinkäsittelyn sovellukset, joissa TTA:n skaalautuvuudesta ja käskytason rinnakkaisuudesta olisi erityistä hyötyä, vaativat tuen laitteistokiihdytetylle liukulukulaskennalle.

Tässä diplomityössä suunniteltiin ja toteutettiin TCE-projektia varten sarja liukulukuyksiköitä. Yksiköiden suunnittelussa pyrittiin alustariippumattomuuteen sekä korkeaan suorituskykyyn Field Programmable Gate Array-alustoilla (FPGA) jopa tinkimällä tuetusta liukulukustandardista. Yksiköt sisältävät työkalut puolen tarkkuuden liukulukulaskentaan. Lisäksi työssä esitetään erikoiskäskyihin perustuvat nopeat algoritmit liukulukujakolaskun ja -neliöjuuren laskentaan.

Yksiköiden toiminta varmistettiin automaattisella rekisterisiirtotason (Register Transfer Level, RTL) testipenkillä. Vertailussa Altera Stratix-II-FPGA:lla yksiköt pääsivät lähelle Alteran omien liukulukuyksiköiden suorituskykyä. Uudemmalla Xilinx Virtex-6-FPGA:lla korkein mahdollinen suorituskyky vaatisi tiheämpää liukuhihnoitusta.

PREFACE

The work for this M. Sc. thesis was carried out at the Department of Computer Systems at Tampere University of Technology as part of the Scalable Parallel Energy Efficient Exposed Datapath Accelerators (SPEEED) project.

I would like to thank Professor Jarmo Takala for letting me work on this interesting and challenging project, and for his ideas on how to improve my work. I am most grateful for Pekka Jääskeläinen for constant feedback and guidance throughout the project. I would also like to thank my colleagues in the TCE project for creating a fun and relaxed atmosphere at work, and giving me a helping hand whenever it was needed. Finally, I would like to thank my friends and family for supporting me all the way.

Tampere, October 15, 2012

Timo Viitanen

CONTENTS

1. Introduction 1
2. Application-Specific Processors 3
2.1 Transport Triggered Architectures 4
2.2 TTA-Based Codesign Environment 6
2.2.1 TCE Function Unit Interface 8
2.2.2 OpenCL Support in TCE 9
3. Floating-Point Arithmetic 10
3.1 Floating-Point Number Representation 10
3.2 IEEE Standard for Floating-Point Arithmetic 11
3.2.1 Subnormal Numbers 12
3.2.2 Rounding Modes 12
3.2.3 Exception Handling 13
3.2.4 Half-Precision 14
3.3 OpenCL Embedded Profile 14
3.4 Fused Multiply-Adder Unit 16
4. Floating-Point Unit Implementations 17
4.1 Floating-Point Units in Commercial Processors 17
4.1.1 Intel Itanium 17
4.1.2 IBM Cell Broadband Engine 18
4.1.3 AMD Bulldozer 19
4.1.4 Intel Ivy Bridge 19
4.2 Open Source Floating-Point Units 19
4.2.1 FloPoCo 20
4.2.2 VHDL-2008 Support Library 20
4.2.3 Sabrewing 21
4.2.4 OpenCores 21
4.3 Floating-Point Libraries for FPGAs 23
4.3.1 Xilinx 23
4.3.2 Altera 23
5. Design and Implementation 25
5.1 Requirements 25
5.2 Single-Precision Components 26
5.2.1 Adder-Subtractor 27
5.2.2 Multiplier 27
5.2.3 Divider 27
5.2.4 Square Root 27
5.2.5 Comparator 28

5.2.6 Integer-Float Converter . . . 29
5.2.7 Fused Multiply-Adder . . . 29
5.3 Sabrewing Wrapper . . . 30
5.4 Half-Precision Components . . . 30
5.4.1 Software Integration . . . 30
5.4.2 Miscellaneous Units . . . 31
5.4.3 Float-Half Converter . . . 31
5.4.4 Comparator . . . 32
5.4.5 Fused Multiply-Adder . . . 32
6. FMA Accelerated Software Operations . . . 35
6.1 Division . . . 36
6.2 Square Root . . . 39
6.3 Accelerator Component . . . 39
7. Verification and Benchmarking . . . 41
7.1 Instruction-Level Simulator Test . . . 41
7.2 VHDL Simulator Tests . . . 41
7.3 Hardware Synthesis Benchmark . . . 42
7.3.1 Altera Stratix-II FPGA . . . 42
7.3.2 Xilinx Virtex-6 FPGA . . . 43
8. Conclusions . . . 47
Bibliography . . . 49
A. Floating-Point Unit Specifications . . . 54

LIST OF ABBREVIATIONS

ADF Architecture Definition File

ALU Arithmetic-Logical Unit

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction Set Processor

CPU Central Processing Unit

FPGA Field-Programmable Gate Array

FPU Floating-Point Unit

FU Function Unit

GPP General-Purpose Processor

GPU Graphical Processing Unit

HDL Hardware Description Language

HPC High Performance Computing

IDF Implementation Definition File

ILP Instruction-Level Parallelism

LSU Load-Store Unit

LUT Look-Up Table

NaN Not a Number

OSAL Operation Set Abstraction Layer

RF Register File

TCE TTA-based Codesign Environment

TTA Transport Triggered Architectures

VLIW Very Long Instruction Word

1. INTRODUCTION

Many computational systems are intended for a specific application, and have strict requirements on cost, power economy or performance. *General-Purpose Processors* (GPP) which can be programmed to run many different applications with reasonable efficiency, may not be able to meet these requirements. A common approach is to augment a GPP with digital signal processors, application-specific fixed function units, etc. A system designed in this way may be difficult to program and inefficient, since the off-the-shelf components have unnecessary functionality. Moreover, if there are no directly applicable off-the-shelf components, designing a suitable fixed-function unit may be prohibitively expensive.

One approach to the problem is replace the GPP with an *Application-Specific Instruction set Processor* (ASIP) whose architecture is tailored for the specific application at hand. Fixed function units may be integrated at the instruction set level, reducing the programming effort necessary to interact with them. In order to be useful, the process of codesigning ASIPs and the corresponding software has to be highly automatized. The *TTA-based Codesign Environment* (TCE), developed in the Tampere University of Technology since 2002, is an ASIP design toolset that attempts to provide such automatization. TCE is mature enough to be used in real-world applications. For example, in [1], TCE was used to build an application-specific processor to accelerate video decoding. In terms of area, power consumption and performance, the ASIP lost to dedicated accelerator blocks, but significantly outperformed a general-purpose ARM processor on all counts.

Several potential applications of TCE would benefit from hardware-accelerated floating-point arithmetic. For instance, there is growing interest in *High Performance Computing* (HPC) applications. Emphasis on power consumption is turning FPGAs into an attractive alternative to *Graphical Processing Units* (GPUs) [2]. TCE might be a good fit for HPC due to its scalable processor template and its support for the OpenCL language. Many HPC applications, e.g., physics modeling, require large amounts of floating-point operations. On a similar note, a potential future application for TCE is GPU implementation. Simulation results show that TTA processors might scale better than the existing GPU architectures [3]. This *TTAGPU* concept would also require massively parallel floating-point computation.

A common belief is that low-power applications should use fixed-point rather than

floating-point arithmetic for signal processing. However, studies suggest that this is not always the case. For instance in [4, 5], TCE was used to design a FPGA-based wireless sensor platform, an application which calls for very high power efficiency. In a signal processing task within the low-power environment, reduced-precision 12-bit floating-point operations were found to save area and power consumption compared to 16-bit fixed-point arithmetic without increasing the signal-to-noise ratio.

Previously, TCE lacked hardware accelerators for floating-point operations, which are necessary in these application domains. Floating-point processors could be simulated at the instruction set level, but could not be synthesized for the lack of said accelerators. In this thesis, this problem was addressed by designing and implementing a set of *Floating-Point Units* (FPU) for TCE, based on the open source VHDL-2008 Support Library [6]. Particular effort was taken to improve performance on FPGA platforms, even by relaxing standard compliance. In addition, the FPUs are customizable for nonstandard floating-point formats such as the aforementioned 12-bit format. In particular, instruction-level support and customized FPUs were implemented for a 16-bit floating-point representation.

This thesis is structured as follows. Chapter 2 describes the TTA processor architecture and the TCE toolset. Chapter 3 discusses common floating-point standards. Chapter 4 is a review of existing floating-point unit implementations, both to evaluate candidates for inclusion into TCE, and to provide points of comparison. Chapter 5 describes the implementation process and the produced FPUs. Chapter 6 discusses a software-leaning approach for implementing complex floating-point operations. In chapter 7, the FPUs are verified and benchmarked. Finally, conclusions and future work are presented in Chapter 8.

2. APPLICATION-SPECIFIC PROCESSORS

It is well known that ASIPs tailored for the application at hand can produce major performance, area and power improvements over GPPs. Usage of ASIPs has been limited by a long and expensive design cycle, which may even deliver a processor based on outdated technology, as the performance of commodity hardware will have improved at a rapid pace while the ASIP was being developed. Therefore, there is continuing interest in tools that automate the design process [7].

The main technologies for realizing ASIPs are the *Application-Specific Integrated Circuit* (ASIC) and the *Field-Programmble Gate Array* (FPGA). An FPGA is an integrated circuit which can be reconfigured in the field, by developers or end users, to emulate different arrangements of logic gates. It consists of many small, programmable logic cells, typically built around *Look-Up Tables* (LUT) which are rewritten during the reconfiguration process. Newer FPGAs include fast accelerator components for common operations, such as barrel shifters and multipliers. The major FPGA vendors at the moment are Xilinx and Altera.

In an ASIC, the logic gates are fixed on the silicon and cannot be changed. An ASIC outperforms an FPGA implementation of the same logic by a wide margin in terms of performance, area and power efficiency, when both circuits are produced using the same process technology. A 2006 study found that over several test cases, the FPGA implementation using the same process technology is on average three times slower, twenty times larger, and consumes nine times as much dynamic power [8]. On the other hand, ASICs are characterized by very high up-front and low unit costs. That is, FPGAs are cheaper for prototyping and small production runs, but ASICs break even when the application justifies mass-production on a sufficient scale.

Altera claims that recent developments in *Complementary Metal Oxide Semiconductor* (CMOS) technology have been advantageous for the FPGA [9]. The up-front costs of high-end process technology are growing extremely high, limiting its use to a handful of large vendors such as Intel and AMD, and to markets measured in billions of dollars, such as commodity *Central Processing Units* (CPU) and *Graphical Processing Units* (GPU). For many applications, an affordable ASIC realization will be based on much older technology than the corresponding FPGA, which bridges some of the aforementioned performance gap. Consequently, the number of FPGA-based

projects is increasing rapidly.

2.1 Transport Triggered Architectures

Transport Triggered Architectures (TTA) are a class of processor architectures first proposed by Lipovski in [10] as an efficient microcontroller design, and elaborated by Corporaal et al for the purpose of ASIP design [7]. Since the long design cycle is an obstacle for the use of ASIPs, the architectures should be suitable for automatic processor generation, and cover a wide range of possible applications with differing functional and performance requirements [7].

A TTA processor is made up of a set of *Function Units* (FU) joined by an interconnection network, which is made up of several Buses. Typical FUs ínclude *Arithmetic-Logical Units* (ALU), *Load-Store Units* (LSU) and *Register Files* (RF). An sample TTA processor is shown in Figure 2.1. Corporaal argues that TTA is highly flexible since an ASIP designer can customize the FUs and the interconnection network independently of each other: performance can be improved either by inserting more function units, pipelining the existing FUs to increase clock rate, or adding more data transfer capacity [7]. An early demonstration of the TTA architecture is the MOVE32INT processor [11], which showed a remarkably high performance compared to a simple RISC processor synthesized with the same process.

Most processors can be described as *Operation Triggered Architecture* processors, where instructions specify operations. Data transfers are implied that gather together the necessary inputs for the operation, and dispose of the output. By contrast, in TTA processors the interconnections are visible to the programmer at the *Instruction Set Architecture* (ISA) level, and instructions define data transfers along those interconnections. Operations are implicitly triggered by transferring data to the special *triggering port* of a function unit. One advantage to this approach is that the output of an operation can be used directly as an input to another, without having to access a register file. According to a benchmark, commonly 50% of register file accesses can be eliminated with this approach [7].

TTA is similar to traditional *Very Long Instruction Word* (VLIW) processors in that the burden of scheduling operations to exploit instruction-level parallelism (ILP) rests on the compiler. A drawback of TTA is that it is expensive to make the processor interruptable, since in addition to the register files, the hidden internal state of each pipelined function unit must be saved. In addition, instruction words in TTA processors tend to grow very long with large interconnection networks, even compared to VLIW processors. In practice, variable-length coding or lossy instruction compression may be necessary. [12]

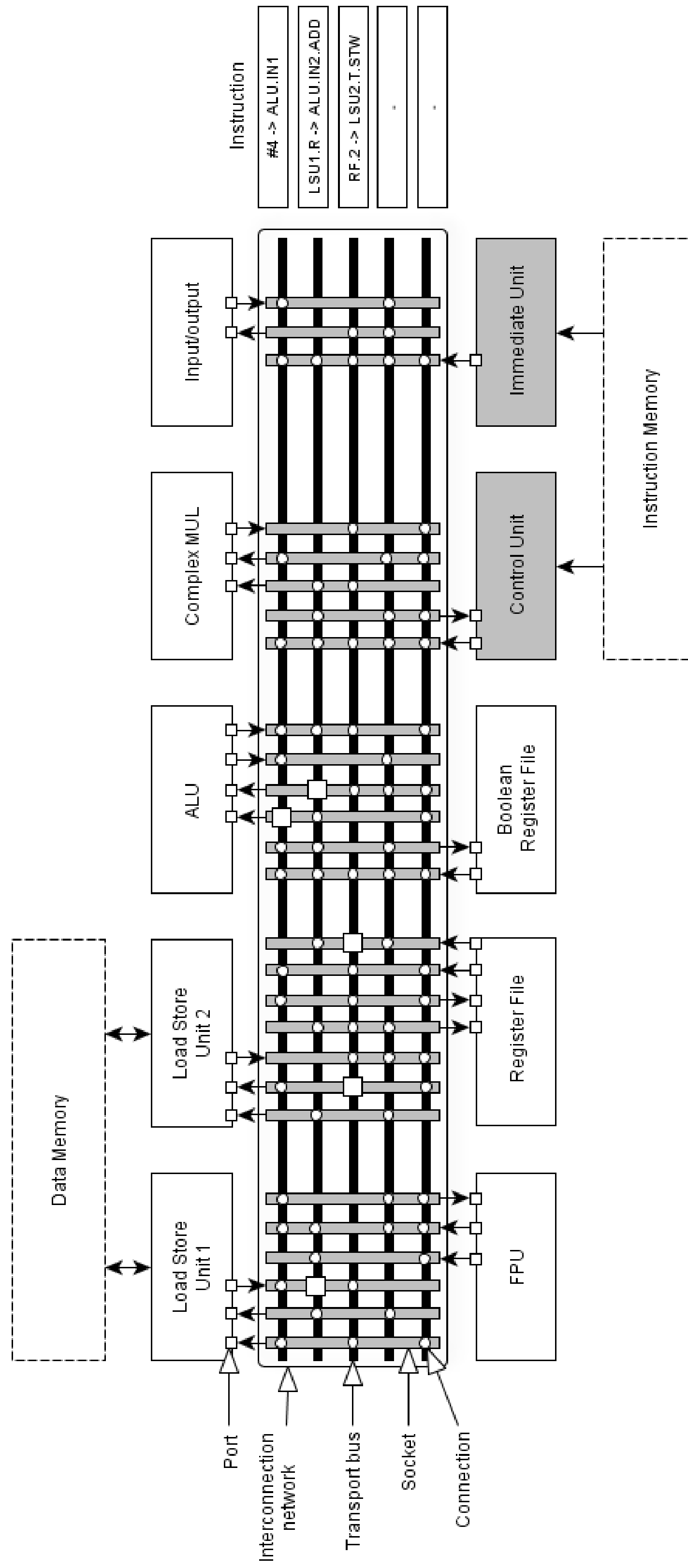


Figure 2.1: An example of a TTA processor with several function units, register files and a customized interconnection network. An example instruction shown on the right defines data transports or *moves* for three buses out of five, performing an integer summation of a value loaded from memory and a constant. The third move stores a register to memory in parallel. The remaining buses are idle. The connections enabled by the moves are highlighted with squares.

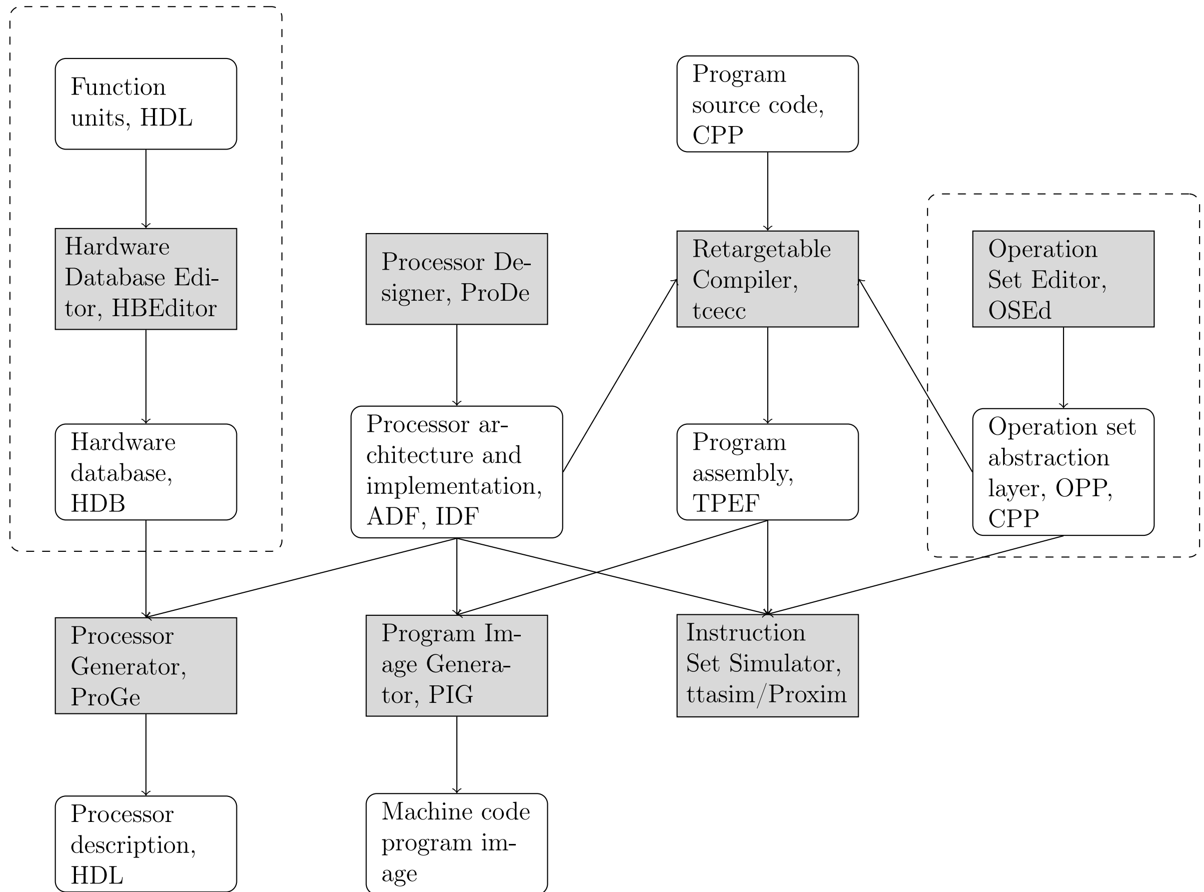


Figure 2.2: Software architecture of the TCE toolset. Sharp grey rectangles indicate utilities. Rounded white rectangles indicate data formats. The main work in this thesis relates to the dashed regions.

2.2 TTA-Based Codesign Environment

The TTA-based Codesign Environment (TCE) is a toolset for developing TTA-based ASIPs, developed in the Tampere University of Technology and released as an open-source project [12]. A coarse software architecture of TCE is shown in Figure 2.2. It is divided into a number of interconnected command line and graphical utility programs. A developer using TCE to create a custom processor from existing FUs will mainly use the utilities *ProDe*, *ProGe*, *tcecc*, *PIG* and *ttasim*.

The *Processor Designer* (ProDe) is a graphical processor design tool, where the developer builds an *Architecture Definition File* (ADF) which specifies the interconnections and function units of his ASIP, and an *Implementation Definition File* (IDF), which references implementations for each function unit in the *Hardware Database* (HDB). The HDB may contain several function units that satisfy the same architectural interface, e.g., low-power units or ones tailored for a specific synthesis

target. The processor architecture so described is turned into a HDL description using the *Processor Generator* (ProGe) [13]. *ProGe* may be configured either to integrate the processor to a target FPGA, or to prepare scripts for simulating the processor using the open-source VHDL simulator GHDL [14], the commercial Modelsim simulator, etc.

Programs in the C and C++ languages can be compiled for the designed architecture with the retargetable LLVM-based compiler *tcecc* [15], which gives an assembly language program object in the *TTA Program Exchange Format* (TPEF) as output. The *Program Image Generator* (PIG) converts such an object into an instruction memory image ready for deployment. In addition, it can simulated using the instruction set simulator *ttasim* [16], or its graphical wrapper *proxim*. The simulator is exact on the instruction cycle level. To be more precise, a TTA processor usually performs an instruction cycle, i.e. the process of fetching and executing an instruction, during a single clock cycle but, e.g., a cache miss may cause the processor to stall, which the simulator does not take into account.

In addition these utilities, a developer who implements his own custom operations and hardware accelerators will require the *OSEd* and *HDBEditor* programs. The *Operation Set Editor* (OSEd) is used to make changes to the *Operation Set Abstraction Layer* (OSAL), which stores the operation set used in TTA processors. The OSAL lists, e.g., the name and operand count of each operation, and can be used to define *trigger semantics*, i.e. legal replacements of an operation with a combination of other operations. For example, a *less-than-or-equal* operation could be computed as *not greater-than*, or $a + b$ as $b + a$ when it is convenient. Furthermore, the simulator behavior of each operation is defined by writing a C++ function that performs the operation.

The *Hardware Database Editor* (HDBEditor) is used to modify SQL-based *Hardware Databases* (HDB), which represent the function unit implementations available for the processor architecture. Each entry in the HDB refers to a *Hardware Description Language* (HDL) description which implements the corresponding unit. Information about the HDL entity needs to be filled in, such as the entity name and the names of each input, output and clock signal, so as to allow *ProGe* to automatically generate a HDL description of the interconnection network which incorporates the unit.

This thesis is mainly concerned with developing new function units and inserting them into a hardware database. This involved writing a HDL descriptions of each function unit, ensuring that they meet the TCE function unit semantics, and writing corresponding hardware database entries with *HDBEditor*. The default OSAL supplied with TCE already included floating-point operations, but some new operations were added during the course of this work, as well as an operand datatype

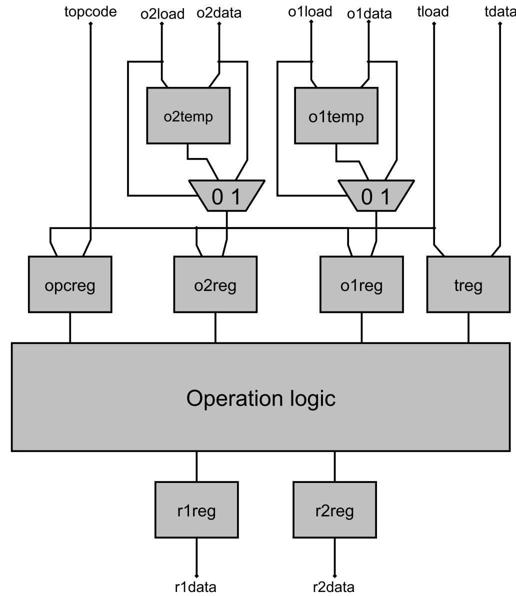


Figure 2.3: An example implementation of the TCE function unit interface with three inputs and two outputs.

for a 16-bit floating-point format.

2.2.1 TCE Function Unit Interface

Since the input and output registers of each FU are visible to the programmer at every cycle, it is important to specify an interface that defines, e.g., what values will appear in the output register of a pipelined function unit at each cycle, with the given instructions. Corporaal discusses several alternative function unit semantics with varying benefits and drawbacks [7]. For instance *hybrid latching FUs*, used in MOVE32INT, stall the internal pipeline until the previous result has been read from the register.

Function units in TCE are expected to start operating only when data is transferred to a *triggering port*, and then deposit the results of the operation to an output register, where the program can access it until rewritten by a later operation. Conversely, the unit should remain idle as long as it is not triggered. Data transfers are signaled to the target FU through by raising the *load* signal of the target port. The *data* signals carrying the actual operands may be shared with other users of the bus and thus should be ignored when the *load* signal is inactive. [17]

Figure 2.3 shows a possible way to implement the TCE function unit interface. Note the *shadow registers* which ensure that the operation logic input only changes when triggered by *tload*, the load signal of the triggering port. The output registers *r1reg* and *r2reg* could be removed without violating the interface, but are often

included in order to reduce the critical path of the processor. Alternatively, the function unit interface could be implemented without the shadow registers by propagating the trigger signal through a pipeline to the output register. The arrangement shown here is larger, but exhibits a lower dynamic power consumption.

2.2.2 OpenCL Support in TCE

In addition to C and C++, processors developed with TCE can be programmed with the *Open Computing Language* (OpenCL) [18]. OpenCL is a product of earlier *General-Purpose GPU* (GPGPU) activity, where programmable GPUs were used for non-graphics computation. It aims to provide a framework for parallel computation where programs can be run both on CPUs and GPUs, and benefit from the parallel processing capabilities of both, instead of the using *shader languages* which are highly graphics-specific and limited to GPUs. Due to the emphasis on portability and concurrency, OpenCL is a useful language for exploiting the instruction-level parallelism of TTA processors. Recent research proposes TTA-based ASIPs used as OpenCL-controlled accelerators. [19]

3. FLOATING-POINT ARITHMETIC

This chapter discusses some theoretical aspects of floating-point arithmetic that are involved in the design of a floating-point unit. The first section introduces the concept of floating-point representation. The following section discusses the ubiquitous IEEE-754 standard for floating-point computation, and its design tradeoffs and special features from a hardware implementation perspective. The next section introduces the OpenCL Embedded Profile, which provides a floating-point standard for embedded devices which is less rigorous and simpler to implement. The final section discusses the fused multiply-add operation, a development in floating-point unit architecture which is recently becoming prevalent in high-end computation hardware.

3.1 Floating-Point Number Representation

The most widely used representation for real numbers is the floating-point representation [20]. In general, floating-point numbers, often referred to as *floats*, are of the form

$$x = s \times m \times b^e \tag{3.1}$$

where s is the sign, m is called the *mantissa*, *fraction* or *significand*, and e is the *exponent*, which causes a binary point to move, or float, relative to the significand. The variable b is the *base* of the floating-point system and in digital systems it is usually two. [21]

An alternative to the floating-point representation often seen in embedded digital signal processors is the fixed-point representation, where a binary point is implied in a fixed position. A hardware implementation of fixed-point arithmetic is inherently cheaper and faster than floating-point arithmetic of equal bit width. However, it is often implemented in software without a native fixed-point datatype, and the required shifts may reverse the speed advantage. Moreover, the much greater *dynamic range* of the floating-point representation is necessary for many algorithms. [22]

s	exponent e	significand m
	30..23	22..0

Figure 3.1: The IEEE-754 single-precision representation.

3.2 IEEE Standard for Floating-Point Arithmetic

In early stages of microcomputing, each computer vendor typically had their own implementation of floating-point arithmetic, which sometimes changed between successive computer models by the same vendor. This made it difficult to write numerical software that was portable between vendors [20]. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) [23] defines rules for floating-point computation in order to address this issue. Since its publication in 1985, it has become prevalent, and most of modern computation hardware complies to the standard.

IEEE 754 defines:

- *computational types* such as single-, double- and quadruple-precision numbers, including special values such as INF (infinity) and NaN (not-a-number,
- *representations* that encode floating-point data in a compact form,
- *arithmetic operations* between floating-point numbers, and
- *exception handling* for e.g division by zero.

The latest revision to the standard, IEEE 754-2008, adds a fused multiply-add operation, and half-precision and decimal computational types.

An IEEE-compliant single-precision number is made up of a sign bit, a 8-bit exponent, and a 23-bit significand, as shown in Figure 3.1. The exponent is a number between -127 and 128, with the two extreme values reserved for special cases. Instead of using two's complement numbers, the exponent is encoded by adding a *bias* of 127, e.g., an exponent of 0 would be encoded as 127, -1 as 126, and so on. This has the advantage that the floating-point zero resembles the integer zero, and simplifies magnitude comparison between floats.

The significand is a fixed-point number in the range $1 < m \leq 2$. Since the most significant digit of such a number is always 1, it need not be encoded, saving one bit of space in the representation. Numbers encoded in this way are called *normal*. Since intermediate results of floating-point arithmetic may be outside this range, they need to be *normalized*. In a hardware context, normalization involves a leading-zero counter and a left-shifter.

Special values are encoded using the two reserved exponent encodings 00 and ff, which correspond to -127 and 128, as shown in Table 3.1. Zero cannot be represented

as a normalized number, therefore, it is represented with a special value. In many early floating-point systems, overflow and division by zero were considered errors that should immediately abort the program. The IEEE standard attempts to be more robust by producing a positive or negative infinity, which behave as very large numbers, thereby allowing some algorithms to succesfully complete even in the presence of overflows. [24]

Table 3.1: Special cases in the IEEE-754 single-precision format

Exponent	Zero Significand	Nonzero Significand
255	± Infinity	Not a Number (NaN)
1 ... 254	Normalized value	
0	Zero	Subnormal number

Computations that have no clearly defined result even when extended to infinity, such as $\sqrt{-1}$, $0/0$ and $\infty - \infty$, produce a *Not a Number* (NaN) special value. If any arithmetic operand is NaN, the result is also NaN. This allows an error condition to be detected at the end of a long series of computations without compromising performance. [24]

3.2.1 Subnormal Numbers

Numbers with a 0 exponent and a nonzero significand are called *subnormals*. The significand in such numbers is not assumed to have an implicit high bit, so they can represent progressively smaller numbers with shrinking accuracy, in a process called *gradual underflow*. Subnormals are expensive to implement in hardware, as they need to be checked for in the unpacking stage, and then normalized using a leading-zero counter and a right-shifter.

Gradual underflow was a source of controversy during the standardization process. Counterproposals to the standard argued that it was unintuitive and difficult to implement, and called for a more traditional truncation to zero in the event of underflow [25]. One argument for subnormals is that they ensure that the identity $a - b = 0 \Leftrightarrow a = b$ holds. Otherwise, subtracting nearby but inequal numbers close to the underflow boundary would result in 0 [21]. Several numerical algorithms can be found that rely on gradual underflow. [24]

3.2.2 Rounding Modes

The IEEE-754 specifies five rounding modes, of which a compliant implementation must support at least four:

Round ties to Even (RtE): Returns the floating-point number closest to the infinitely precise result. In the case of a tie, round to the even number.

Round ties to Away (RtA): Same as RtE, except in the case of a tie, round to the number with the greater magnitude. Support for RtA is optional.

Round toward Positive (RtP): Round to the closest floating-point number that is no smaller than the exact result.

Round toward Negative (RtN): Round to the closest floating-point that is no greater than the exact result.

Round toward Zero (RtZ): Round to the closest floating-point with magnitude no greater to the exact result.

The final three rounding modes are called directed roundings. To comply with the standard, an implementation should use RtE by default, and allow the programmer to select any of the directed roundings at will. A rationale for including them in the standard is that they enable estimation of roundoff error with *interval arithmetic*, where each computation is performed twice with different rounding modes, to obtain an upper and lower bound for the exact value. [21]

Since the absolute value of a floating-point error can vary considerably between different exponents, errors are measured using *units in the last place* (ulp). The difference in ulp between floats with equal exponents is obtained by interpreting their significands as integers and subtracting them. An important distinction between rounding modes is that RtE and RtA are guaranteed to incur rounding errors of at most ± 0.5ulp, while the directed roundings are simpler to implement, but have errors of up to ± 1ulp. [21]

3.2.3 Exception Handling

In the IEEE-754 standard, typical behavior in the presence of error conditions is to produce a result and continue. In the event that this does not suffice, IEEE-754 provides two forms of exception handling, *flags* and *traps*. By default the five exceptional situations defined in the standard raise status flags which may be inspected later in the program. In addition, the programmer should be able to specify a custom trap handler for each exception. One application for such trap handlers is to provide backward compatibility for code designed for archaic computers that interrupted on errors [21]. The five exceptions are [23]:

Invalid operation: Coincides with the operations that produce a NaN result, such as $0/0$ and $\infty - \infty$.

Division by zero: Raised when the divisor in a division operation is zero.

s	exponent e					significand m									
	14..10					9..0									

Figure 3.2: The IEEE single-precision representation.

Overflow: Raised when the exact result of a computation would have been finite, but out of the dynamic range of the floating-point format at hand, and so overflowed to infinity. That is, infinities produced by division by zero, or arithmetic with infinite inputs, do not signal this exception.

Underflow: Raised when a result is very small. The implementation may decide if this means that

Inexact: Raised when an operation incurs a roundoff error, that is, the result is not exact.

3.2.4 Half-Precision

Reduced-precision floating-point formats are useful in some low-power signal-processing applications [4]. The most well-known reduced-precision format is the *half-precision* format defined in the IEEE-754 standard, which is composed of a sign bit, a 5-bit biased exponent, and a 10-bit significand, as shown in Figure 3.2.

Half-precision floating-point numbers or *half-floats* are widely used in *high dynamic range photography*, which stores and displays images using half-floats for color components in place of the traditional 8-bit integers. This allows the computer to display scenes that have detail both in very bright and very dim areas. [26]

IEEE 754-describes half-floats as a storage format, which has to be converted to, e.g., single-precision before arithmetic operations can be performed [23]. The latest 1.2 revision of OpenCL defines an optional extension for half aritmetic, but hardware support is rare as of this writing. Recent commodity CPUs and GPUs often provide accelerated conversion from half-precision to single-precision and vice versa. On these platforms, some applications, for instance [27] derive significant performance gains by storing data in memory in half-precision, but performing arithmetic in single-precision, thereby reducing the usage of memory bandwith, which is often the bottleneck in high performance computing.

3.3 OpenCL Embedded Profile

Some features of the IEEE standard require complex hardware and produce little practical benefit in many applications. In particular, subnormal numbers and the

various rounding modes require costly hardware to implement. The impact is especially large in the FPGA environment, where accelerator blocks are provided for, e.g., significand multiplication which would dominate the hardware cost in ASIC, making denormalization and rounding logic more expensive by comparison. Furthermore, the standard exception handling system is nontrivial to implement on a TTA architecture. The custom trap handlers required by the standard would require interrupt support, which is expensive to implement on a TTA processor [12].

Developers using TCE for, e.g., high performance computing or low-power systems often prefer faster, more power-efficient hardware at the cost of some simplification, and to work around the corner cases where these features become significant. When designing application-specific hardware, often enough is known of the application domain for numerical error analysis that ensures that computation proceeds correctly on the given hardware. In fact, similar analysis is routinely performed in fixed-point DSP, and automated tools exist for this purpose [28]. It can be argued that the IEEE standard is better suited for general-purpose hardware whose specific application is not know at design time. It would, therefore, be desirable to have a looser standard which still describes floating-point behavior, but imposes less hardware cost. One such standard is the OpenCL *Embedded Profile* (EP).

OpenCL requires full IEEE 754 compliance by default but includes an optional "embedded profile" with the following relaxed rules for floating-point accuracy [18]:

- Round to Zero may be used as the default rounding mode instead of Round to Nearest. The other three rounding modes need not be supported.
- Subnormal inputs may be treated as zero, and subnormal outputs may be forced to zero.
- Special case handling, i.e. correct treatment of INFs and NaNs, is optional. If the inputs or the correct output of an arithmetic operation would be a special case, the result is implementation-defined.
- Addition and multiplication must be correctly rounded. Complex operations such as division, square root and transcendental functions have error bounds instead, for instance division must be within ± 3ulp of the exact result, and square root within ± 3.5ulp.

The OpenCL specification characterizes these relaxations of the standard as undesirable. However, fairly recent desktop GPUs like the NVIDIA GTX 280 disregarded IEEE requirements, and were regardless used for, e.g., high performance computing. ATI released an IEEE-compliant GPU only in late 2009, and NVIDIA in 2010. [29]

3.4 Fused Multiply-Adder Unit

A fused multiply-add (FMA) instruction performs the operation $a + bc$ without rounding the intermediate result bc. Many applications involve accumulation of products and therefore benefit from a hardware accelerated FMA instruction. Examples include Newton's iteration, matrix multiplication, polynomial evaluation, the dot product, and the Fast Fourier Transform. The fused operation can be faster than the separate operations because the intermediate result need not be rounded. [30]

An FMA unit can also compute addition and multiplication, for example by adding zero (in case of multiplication), or multiplying by one (in case of addition). Therefore, it can replace separate adder and multiplier components. In e.g. [30] it is proposed that the entire floating-point unit should be built around a fused multiply-adder, listing several advantages:

- Rounding and normalization logic is shared between addition and multiplication, resulting in an efficient use of area.
- As discussed above, an FMA instruction speeds up a variety of computations. It also improves their accuracy by introducing only one roundoff error, where separate operations would suffer from two.
- The FMA instruction enables fast software algorithms for complex operations such as division and square root, which may perform well enough to replace separate hardware accelerators for these instructions. These algorithms are discussed in detail in chapter 6.

As a disadvantage, separate addition and multiplication may suffer from a slightly higher latency and power consumption. Furthermore, indiscriminate use of an FMA operation may damage the accuracy of some rare algorithms. For instance, an algorithm may rely on the fact that $a \times a - a \times a$ is not negative; however, if an FMA operation is used, the unrounded intermediate $a \times a$ may be greater than the rounded-off $a \times a$ on the left side, producing a negative result [31]. Still, several notable processors have FPUs built around this concept, including the Intel Itanium [32], the IBM Cell Broadband Engine [33], and the recent AMD Bulldozer [34]. The 2008 revision of the IEEE-754 standard defines an FMA operation [23].

4. FLOATING-POINT UNIT IMPLEMENTATIONS

This chapter is a review of various existing floating-point unit implementations. The following section describes the floating-point capabilities of modern off-the-self computing hardware. Points of particular interest are standard relaxations and the prevalence of the FMA unit, as well as engineering solutions related to it. The second section reviews floating-point units that are freely available as *Hardware Description Language* (HDL) descriptions, the main goal being to find suitable FPUs to form the basis of floating-point support in TCE, as developing them from scratch would be needlessly time-consuming and error-prone. Finally, the third section describes the platform-specific floating-point solutions supplied by the major FPGA vendor, which will be used as points of comparison.

4.1 Floating-Point Units in Commercial Processors

This section is an overview of the floating-point capabilities of some commodity computation devices. Such devices can reach very high performance due to the fine-grained process technology they are synthesized with, made possible by the scale of their market. A particularly interesting device is the IBM Cell processor, which employs an FMA unit and reduced standard compliance. The findings are summarized in Table 4.1.

4.1.1 Intel Itanium

The Itanium is a processor family by Intel targeted for the server and HPC markets. It breaks instruction-set compatibility with the x86 ISA to achieve this goal. The Itanium is interesting as an early example of the FMA unit. Its FPU is built around two parallel FMA units, and later dual-core Itaniums have a total of four. [32]

The newest Itanium processor for which detailed information is available is the Montecito, released in 2005. At 1.66 GHz, and considering FMA to count as two floating-point operations, the theoretical peak performance of a Montecito is 13.28 *billions of floating-point operations per second* (GFLOPS).

The Itanium does not have a built-in divider. Instead, the Itanium *Instruction-Set Architecture* (ISA) provides instructions for fast approximation of e.g. reciprocals

Table 4.1: Comparison of off-the-shelf computation hardware. The reported floating-point performance is a theoretical maximum.

	Intel Itanium	IBM Cell	AMD Bulldozer	Intel Ivy Bridge
FMA based	yes	yes	yes	no
Process	90nm	65nm	32nm	22nm
Clock frequency	1.66GHz	3.20GHz	3.60GHz	3.50GHz
Cores	2	6	8	4
FP perf. per core (GFLOPS)	6.64	25.6	33.6	56.0
FP perf. total (GFLOPS)	13.28	153.6	268.8	224.0

and square roots by table lookup, which can be then be refined into correctly rounded results using the FMA instruction. Intel provides fast algorithms for this purpose, along with mathematical proofs of their correctness [32]. The Itanium was widely used in supercomputing, appearing in 84 machines in the Top500 supercomputer list of 2004 [35].

4.1.2 IBM Cell Broadband Engine

The IBM Cell Broadband Engine processor [33] is well known for its application in the Playstation 3 video game console. The Cell is made up of eight Synergistic Processing Elements (SPE) coordinated by a single PowerPC core. One SPE is disabled to increase chip yield, and one reserved for the operating system, so six SPEs are available for the developer.

Each SPE has a fully pipelined vector floating-point unit capable of completing four single-precision operations on every cycle. The vector FPU is based on an FMA datapath, but engineered so that separate addition and multiplication operations do not suffer a latency penalty. When adding, the multiplication pipeline stage is skipped, and vice versa for multiplication, which works out to a latency of six cycles for the FMA operation, and five cycles for each separate operation. The FPU hardware can also be used for 16-bit integer arithmetic.

Counting every FMA operation as two floating-point operations, a single SPE has a theoretical maximum performance of 25.6 GFLOPS, and the entire Broadband Engine reaches 153.6 GFLOPS. The single-precision unit in the Cell FPU makes heavy use of relaxed standard compliance in order to "place emphasis on real-time graphics requirements that are typical of multimedia processing". The FPU only supports the Round toward Zero rounding mode, subnormal operands are treated as zero and subnormal results forced to zero, and the 'all-ones' exponent is treated as a normal exponent instead of a special case. Overflows are saturated to the maximum representable value. These relaxations coincide closely with the OpenCL

EP; therefore the OpenCL EP may have been designed in part to accommodate the Cell.

As with the Itanium, complex operations are performed in software, with the help of special instructions that provide reciprocal and square root estimates. The complex operations fall short of correct IEEE rounding, likely due to the standard relaxations above [36].

The double-precision unit in the original Cell is IEEE-compliant but is engineered to minimize chip area at the cost of performance. IBM has since released a HPC-oriented revision of Cell named the PowerXCell 8i, with a total of eight developer-visible SPEs and improved double-precision performance, only slower than single-precision by a factor of two. The PowerXCell was used in IBM's Roadrunner supercomputer, which was the first computer to reach a sustained floating-point performance of one petaflop. [35]

4.1.3 AMD Bulldozer

The Bulldozer is AMD's latest desktop processor microarchitecture, released in 2011. The most high-end Bulldozer model in the market has a theoretical peak floating-point performance of 268.8 GFLOPS. The Bulldozer is divided into 'modules' of two cores. Each module has a single FPU shared between its constituent cores. The FPU is built around two 128-bit FMA units, each of which is capable of completing either four single-precision operations or two double-precision operations every cycle. Instead of emulating complex operations such as division and square root on the software level, they are performed by an internal state machine within the FPU. [34]

4.1.4 Intel Ivy Bridge

Intel's latest desktop processor microarchitecture is codenamed Ivy Bridge. The first Ivy Bridge processors entered production in 2011. The Ivy Bridge does not use an FMA-based datapath, but instead reaches similar performance as the Bulldozer by employing both a 256-bit vector adder and a 256-bit vector multiplier for each core. Ivy Bridge's successor Haswell is slated to replace these with FMA units [37]. Consequently, unlike the aforementioned devices, the Ivy Bridge has a built-in divider, shared between the integer and floating-point datapaths. The microarchitecture also includes an on-chip GPU which is OpenCL programmable, which could add significant floating-point performance if properly used. [38]

4.2 Open Source Floating-Point Units

This section is a review of FPUs for which HDL descriptions have been made available without charge. Such FPUs are interesting as points of comparison and also as

candidates for integration into TCE, if they have a sufficiently liberal license to be compatible with the MIT license used by TCE.

4.2.1 FloPoCo

FloPoCo is a FPGA-based floating-point arithmetic system developed in the French National Institute for Research in Computer Science and Control (INRIA), applicable also for fixed-point arithmetic and ASIC platforms. Its basic idea is to generate fused complex operation units which are faster, more accurate and require less hardware than if they were implemented with elementary operations. One example given is a custom datapath for $x^2 + y^2 + z^2$. Compared to an implementation pieced together from vendor-provided elementary operations, a fused datapath generated by FloPoCo requires one-third of the cycles and chip area. [39]

Though its focus is on complex custom operations, FloPoCo includes a large variety of elementary floating-point operations which can be used by themselves or as building blocks for the aforementioned exotic operations. In addition to those shown in Table 4.2, the operations include three-input addition, squaring, multiplication by constant, division by constant, exponential, natural logarithm, and raising to a power. The operations are reported competitive with FPUs supplied by FPGA vendors. Multiple architectures are provided for, e.g., addition and square root with varying tradeoffs between area, latency and clock rate. There is emphasis on pipelining for performance: the developer can specify a target frequency, and the system attempts to meet it by automatically inserting pipeline stages. [40]

Instead of using the IEEE-754 representation, FloPoCo stores information on special cases in three additional bits, which simplifies decoding logic. While this would be impractical with standard memory built around 32-bit words, the developers reason that due to the flexibility of FPGAs it is feasible to support internal use of 35-bit memory. In addition, support for subnormal numbers was left out of FloPoCo, arguing that they can be replaced by adding one bit of precision to the exponent field. [40]

There is also a commercial fork of FloPoCo's predecessor FPlibrary, named libHDLfltp. Unfortunately, FPlibrary and libHDLfltp each fall under a GPL-based license, which is incompatible with integration into TCE, and since the copyright owners of FloPoCo itself have not yet decided on terms of distribution, it is distributed "all rights reserved".

4.2.2 VHDL-2008 Support Library

The VHDL-2008 revision of the VHDL language specifies a mathematics library with native floating-point types and arithmetic operations. At this time neither Altera

nor Xilinx synthesis tools implement this part of the standard, but the public-domain *VHDL-2008 Support Library* [6] implements synthesizable versions of these types using VHDL-1993.

The Support Library is flexible, with options for different significand and exponent widths, rounding modes, as well as parameters to turn off special case and denormal number support. As a downside, all the functions are combinatorial, and need to be pipelined by hand.

4.2.3 Sabrewing

The Sabrewing is an FPU aimed for embedded applications. It uses the FMA datapath to minimize area, and also implements hardware pipeline resource sharing between integer and floating-point operations, possibly eliminating the need for a separate integer ALU. To achieve this, the Sabrewing internally uses a 32-bit significand suitable for integer arithmetic, and allows computation both with single-precision numbers and extended 46-bit floats. The FPU supports floating-point and integer multiply-addition, multiplication, addition and comparison, as well as integer left-shift and right-shift operations. [30]

The Sabrewing performs well in a 65 nm technology ASIC prototype, reaching a clock speed of 1.35 GHz with a latency of three cycles. It outperforms the VHDL-2008 Support Library in terms of maximum frequency, power consumption and area, while producing more standard-compliant results. However, being optimized for ASIC synthesis, it may be suboptimal in FPGA use.

4.2.4 OpenCores

The OpenCores project is a repository for open-source hardware IP cores. It hosts many arithmetic cores, a small fraction of which are characterized as floating-point.

The *fpu100* unit developed in the Vienna University of Technology appears to be the most mature floating-point IP in OpenCores. The monolithic FPU is capable of addition, multiplication, division and square root. Since square root and division are rare operations, they were implemented digit-serially in order to save area, with a latency of 35 cycles. Addition and multiplication are fully pipelined at 7 and 12 cycles, respectively. The entire unit reaches a clock rate of 100 MHz on a Cyclone EP1C6 FPGA. It is tested with a comprehensive test suite and verified on hardware.

The *fpu* unit was used as a point of comparison in the *fpu100* documentation. Similarly to *fpu100*, it is a monolithic FPU comprehensively tested to be IEEE-754 compliant, but less effort has gone to performance optimization. Each operation has a latency of 4 cycles, including division, resulting in a clock rate of only 6.17 MHz.

The *fpuvhdl* project contains an adder and a multiplier. The units are fully

Table 4.2: Comparison of available FPUs. The FloPoCo operations marked as (yes) are not supplied as elementary operators, but could be trivially synthesized using the FloPoCo system. The OpenCores column includes the *fpu100* and *fp_ log* units. *free* licenses permit all use and redistribution, but may require the reproduction of a copyright notice when redistributing.

FPU	Bishop	Sabre-wing	Xilinx	Altera	Open-Cores	FloPoCo
Operations:						
add, sub	yes	yes	yes	yes	yes	yes
multiply	yes	yes	yes	yes	yes	yes
FMA	yes	yes				(yes)
divide	yes		yes	yes	yes	yes
compare	yes	yes	yes	yes		yes
sqrt(x)	yes		yes	yes	yes	yes
$1/x$	yes		yes			yes
1/sqrt(x)			yes	yes		(yes)
log(x)			yes	yes	yes	yes
Features:						
subnormals	optional	yes	no	no	yes	no
rounding modes	all	all	nearest	nearest	all	nearest
INF, NaN	optional	optional	yes	yes	yes	yes
precision	all	single, 46b	all	single, double	single	all
pipelined	no	yes	yes	yes	yes	yes
param. latency	no	no	yes	yes	no	yes
integer operations	no	yes	no	no	no	no
portable	yes	yes	no	no	yes	yes
license	free	free	comm.	comm.	free	no

pipelined at latencies of 6 and 4 cycles, respectively, and reach clock rates of approximately 90 MHz on a Xilinx Virtex-II XC2V3000 FPGA. Judging by the operation latencies, the *fpu100* multiplier was designed for an old FPGA without embedded multiplier blocks, while the *fpuvhdl* multiplier may be better suited for modern FPGAs which include such blocks.

The remaining single-precision units in OpenCores are the multiplier *cf_fp_ mul* which is based on the discontinued high-level HDL Confluence, and the logarithm-taking unit *fp_ log* which implements the ICSILog algorithm. In addition, OpenCores contains three double-precision units named *fpu_ double*, *double_fpu* and *openfpu64*. These are of less interest since the main focus of this thesis is on single- and reduced-precision arithmetic.

4.3 Floating-Point Libraries for FPGAs

This section is a review of the floating-point units supplied by major FPGA vendors for use on their FPGA platforms. Due to their high level of platform-specific optimization, they can be considered as representative of the maximum floating-point performance reachable on a given platform. These units are summarized in Table 4.2 together with the open-source FPUs from the previous section.

4.3.1 Xilinx

Xilinx provides a floating-point IP with its ISE design environment, called the *LogicCore IP Floating-Point Operator*. The component may be configured to perform the following operations: basic arithmetic (addition/subtraction, multiplication, division, square root) and other functions (conversions, comparison). [41]

The IP is highly generic. The user can specify the significand and exponent widths, specify a latency between zero and an operation-specific maximum value, and specify whether special DSP blocks are used in synthesis.

As a downside, only one of the operations can be selected. Therefore, the IP cannot synthesize, e.g., an FPU that shares rounding and normalization logic across operations. The IP also does not include a fused multiply-adder. It complies with IEEE-754 partially, but for instance supports only the default rounding mode, round-to-nearest-even. Xilinx also does not support subnormals, arguing that they do not contribute to the result in most practical calculations, since they are very small. [41]

Also, Xilinx provides a larger monolithic floating-point unit, *LogicCore IP Virtex-5 APU Floating-Point Unit*, intended for use in its *Embedded Development Kit* (EDK) system-on-chip design flow as a coprocessor to a PowerPC CPU. As such, it decodes standard PowerPC floating-point instructions. The APU has several attractive qualities, such as full IEEE-754 support and an FMA operation. However, the unit is closely linked to the EDK system, and may be difficult to extract for general use. As a point of technical interest, the unit is designed to run at either one-half or one-third the clock rate of the host PowerPC core. [42]

4.3.2 Altera

The synthesis toolset for Altera FPGAs includes a large palette of *megafunctions* for floating-point computation, divided into: basic arithmetic (addition/subtraction, multiplication, division, square root), algebraic functions (exponential, inverse, inverse square root, natural logarithm), trigonometric functions (sine, cosine, arctangent), other functions (absolute value, conversion, comparison), and complex functions (matrix inverse, matrix multiplication). [43]

Most functions may be configured to use single-precision, double-precision, or

"single extended precision" which can vary between 43 and 64 bits. Furthermore, most functions have two latency options. The basic arithmetic functions also have a low-latency option with more granularity. Some complex functions offer less choice, for instance the sine-cosine function has no parameters except that either a sine or a cosine unit can be synthesized.

As with Xilinx, the Altera components flush denormal inputs to zero, and support only the default round-to-nearest-even rounding mode. In summary, Altera supplies a larger arsenal of FPUs than Xilinx, but they support a lesser degree of customization.

5. DESIGN AND IMPLEMENTATION

This chapter describes the implemented floating-point units. The following section describes the design requirements, and the process of selecting a freely available FPU library as a basis for implementation. Subsequently, each of the implemented function units is described, beginning with the single-precision units and followed by the half-precision units. The descriptions are not intended to be exhaustive, but instead to lay out the interface of each FU necessary for their use, including supported operations, generic parameters and latencies, and to elucidate points of technical interest in their implementation.

5.1 Requirements

Several considerations had to be taken into account when designing the FPUs:

- The FPUs should provide the elementary operations necessary for writing floating-point programs. These include basic arithmetic, comparison, conversion between floats and integers, and the square root operation.
- The FPUs should reach high clock speeds on multiple FPGAs. Additional pipeline stages should be inserted as necessary to reach this goal.
- Standard compliance can be sacrificed for performance if possible. To keep some measure of well-defined behavior, the OpenCL EP is complied to instead of the full IEEE 745.
- Even though the EP does not require special case handling, all operations should properly generate and preserve INFs and NaNs, so that error conditions can be detected at the end of a long series of calculations.
- The FPUs should be customizable for various floating-point formats through the use of parameters. In particular, half-precision floats should be supported.
- The FPUs should be synthesizable on both Altera and Xilinx FPGAs, and simulable with GHDL [14]. That is, they should not use vendor-specific IP cores.
- The FPUs should use no third-party code that cannot be integrated into the TCE project and distributed under TCE's MIT license.

As shown in the previous chapter, there is an abundance of freely available floating-point units. Consequently, it would make little sense to write the FPUs from scratch. None of the discussed FPUs fit all of the requirements as they are, so some modification is necessary. The VHDL-2008 Support Library was chosen as the basis for FPU implementation since it is written in easily modifiable high-level VHDL, well tested, used in several scientific publications, and flexible in terms of parameters. Due to the parametrization, it is simple to convert a Support Library-based FPU to use e.g. reduced-precision formats, to select the rounding mode, and to remove subnormal number support.

Furthermore, the floating-point operations are divided into separate functions. In TTA, it is desirable to have separate function units for different operations, so that the processor's hardware operations can be customized more easily for the application at hand. If the implementation were based on a monolithic FPU, it would take significant effort to produce these separate units.

A major disadvantage of the Support Library is that the units need to be pipelined by hand. As seen later in their corresponding sections, the division and square root functions use algorithms unsuitable for synthesis, and need to be partially rewritten. Furthermore, the high-level VHDL code is not highly optimized for any single platform, and cannot be expected to attain maximal performance.

In addition to the main Support Library-based FPUs, the Sabrewing FPU was provided as an alternative optimized for ASIC performance. The FloPoCo system has attractive properties, such as a high performance competitive with FPGA vendor FPUs, and support for exotic operations such as logarithms and exponentials. Its idea of automatically generating custom operations would be a good match to the ASIP concept. Unfortunately it could not be integrated into TCE due to licensing issues. Another issue is its exotic 36-bit wide single-precision format which would need to be converted at some point into the conventional 32-bit format in order to reconcile it with commodity memory. However this would involve less effort than the manual pipelining required by the Support Library. In addition, since FloPoCo is designed ground-up without subnormal support, it would be difficult to make IEEE-compliant on demand, as opposed to the Support Library where a parameter change suffices.

5.2 Single-Precision Components

The components were implemented by taking a VHDL-2008 Support Library function, parametrizing it to comply to OpenCL EP, inserting sufficient pipeline stages to achieve the required clock speed, and wrapping the result in a TCE Function Unit. The square root and divider components required algorithm-level performance optimization. Technical specifications of each unit are shown in Appendix A, including

parameters and latencies.

5.2.1 Adder-Subtractor

The implemented adder block is shown in Figure 5.1a. Dashed lines indicate pipelining. Some adder designs, e.g. the Sabrewing [30] handle negative results by negating them in two's complement. In contrast, the Support Library adder ensures that the lower significand is always subtracted from the higher and the result is guaranteed to be positive. This involves a full floating-point comparison, and swapping the significands if necessary.

5.2.2 Multiplier

The implemented multiplier block is shown in Figure 5.1b. The 'truncation' stage of the multiplier appears superfluous, especially when using the RtZ rounding mode. It is required in order to give the synthesis tools room for register retiming around the expensive integer multiplication. Removing the stage reduces performance.

5.2.3 Divider

The Support Library divider uses the simple base-2 shift-and-subtract algorithm for integer division. In order to compute each bit of the quotient, the divisor is shifted and subtracted from the dividend, and the result is sign-checked. The algorithm produced a reasonable clock rate, but required one pipeline stage per significand bit; packing two iterations to one pipeline stage damaged the clock rate.

For the purpose of improving latency, the divider was reworked to divide base-4 instead of base-2. The unit initally precomputes multiplication of the divisor by two and three. Subsequently in each iteration, the unit generates two bits of the quotient on every iteration by attempting to subtract three multiples of the divisor in parallel. This redesign retained a high clock rate while almost halving latency. A block diagram of the redesigned divider is shown in Figure 5.2a.

5.2.4 Square Root

The Support Library includes a square root function, however, it is implemented with a simple Newton's iteration:

$$f(x) = x^2 - n \tag{5.1}$$

$$f'(x) = 2x \tag{5.2}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - n}{2x_i} = \frac{1}{2}\left(x_i + \frac{n}{x_i}\right) \tag{5.3}$$

where x_i approaches the square root of n as $f(x_i)$ approaches 0. When synthesized on hardware and fully pipelined, the function requires an integer divider for each iteration step. It is therefore useful mainly for simulation, and unreasonably large and slow for synthesis.

To produce a more useful component, the core integer square rooter was rewritten using Hain's algorithm [44], keeping the exponent computation, unpacking, packing and normalization logic of the original Support Library function. Hain's algorithm determines the output bit by bit, starting from the MSB. In each iteration, the algorithm adds a trial offset b to the intermediate result a. If then $(a+b)^2 < n$, where n is the input, the corresponding output bit must be 1. After further optimization by keeping track of two auxiliary variables, the critical path of an iteration step involves only a subtractor, a multiplexer and an incrementor. The optimized Hain's algorithm is shown in the block diagram of the unit in Figure 5.2b.

Even so improved, the unit is large and has a long latency of 26. The Newton's iteration requires only $O(\log n)$ iterations as opposed to the $O(n)$ of Hain's algorithm, but this is more than offset by the cost of each iteration.

5.2.5 Comparator

The Support Library provides separate function calls for various comparisons. Since the comparator should support all six comparison operations (equal, not-equal, less-than, less-than-or-equal, greater-than, greater-than-or-equal), it was implemented by hand without referring to the Support Library. An alternative would be to remove, e.g., the greater-than and greater-than-or-equal operations since they can be restated using other operations by swapping the parameters. This would save one instruction word bit at the cost of a minor performance hit when the interconnection network is better suited for one parameter ordering.

As discussed in Section 3.2, the IEEE Single-Precision format is designed such that the low 31 bits can be treated as integers for the purpose of comparing magnitudes. After comparing magnitudes, the results of each comparison operation can be generated with cheap boolean logic. The very cheap negation and absolute value operations were added since they did not appear to justify an FU of their own. Negation is implemented simply by flipping the sign bit of the input operand, and

absolute value by setting it to zero.

5.2.6 Integer-Float Converter

Conversion between integers and floats was a surprisingly complex operation, requiring four clock cycles to complete. The bottleneck is the signed integer-float conversion operation, CIF, which is similar to the normalization stages of the previous units. Adding to the complexity, the number being normalized is 10 bits longer than usual, and needs to be sign-checked and negated in two's complement if negative, which involves a full carry-propagation. The OpenCL EP specifies that all conversions should comply exactly to IEEE-754, but since the RTZ rounding mode is still allowed, and denormal numbers never affect the result, there is no performance impact.

5.2.7 Fused Multiply-Adder

As described in Section 3.4, a fused multiply-adder computes the operation $a + bc$, which speeds up a variety of computations that require accumulation of products, such as matrix multiplication. The FMA component implemented in this thesis provides multiplication, addition and subtraction instructions using the simple technique discussed in Section 3.4.

Moreover, a multiply-subtract operation was added, which computes $a - bc$. The operation has a minimal hardware cost, as it is sufficient to flip the sign bit of b or c. Multiply-subtract improves the performance of, e.g., the fast Fourier transform and any other computations that involve complex multiplication, which is defined as $(a, b) \times (c, d) = (ac - bd, ad + bc)$.

A block diagram of the implemented FMA is shown in Figure 5.1c. The FMA has six pipeline stages, four of which can be bypassed using generic parameter flags, named *bypass_2*, *bypass_3*, etc. Each flags has the effect of turning the related D-flipflops into wires. The latency of the entire unit is therefore parametric and can be set between 2 and 6. These parameters were used to specify a palette of FMA units with different latencies into the hardware database. Since a given latency is reached with multiple combinations of flags, optimal combinations were determined by synthesizing each possible unit.

Relaxed standard compliance was very effective when implementing the FMA component. The original implementation requires an expensive inital normalization stage to handle denormal numbers. When both inputs to the multiplier stage are normal, the result can have at most one leading zero, making normalization of the intermediate result trivial. Initially, a component was experimented with that was otherwise identical, but did not implement the subtraction or addition operations.

Benchmarking showed no meaningful difference in area or frequency.

5.3 Sabrewing Wrapper

The Sabrewing FPU [30] described in section 4.2.3 was thought useful to include as a TCE function unit, since it has no license restrictions and performs well on 90nm ASIC. It is also highly IEEE 754-compliant. As a drawback, having been designed for ASIC synthesis it may not perform well on FPGA.

The original unit has a latency of three cycles. Pipeline registers were added to the beginning and end of the unit. In addition, an optional pipeline register bank was inserted after the logic that translates TCE input data and operation codes into the format expected by the Sabrewing. Enabling this optional register improved performance significanly on the Xilinx Virtex-6. The total latency of the unit can, therefore, be set to five or six cycles.

Some of the Sabrewing's functionality had to be ignored in writing the function unit wrapper. In particular, neither rounding mode selection nor the extended 41-bit floating-point format appeared important enough to warrant addition to the default TCE instruction set. Therefore the function unit only supports single-precision numbers and the Round to Zero rounding mode, so as to make it a fair point of comparison. Changing the default rounding mode to e.g. Round to Nearest would be trivial.

5.4 Half-Precision Components

This section first discusses the software issues related to integrating half-float arithmetic into TCE, and subsequently describes each implemented half-precision floating-point unit. Most of these are slightly modified single-precision FPUs with different parameters.

5.4.1 Software Integration

Single-precision floats were already well integrated into the software side of TCE at the outset of this thesis work, so that language constructs from all high-level languages using floats were properly converted into operations by the compiler, and even replaced with emulation code if hardware acceleration was unavailable. Only the actual function unit implementations were absent. Half-precision floats did not yet have equivalent support.

In order to integrate halfs into TCE, a *HALF_ WORD* datatype was added as a possible argument type for operations in the *Operation Set Abstraction Layer* (OSAL), and corresponding half-precision operations were added to the default operation set for each existing single-precision operation. The simulation behavior of

the operations is to perform computation in single-precision and truncate the results to half-precision.

A major remaining obstacle is that TCE relies on LLVM code-generation to convert floating-point calculations in high-level language into machine code instructions. At this time, LLVM only supports half-floats as a storage format without arithmetic operations, as defined by IEEE-754. Code-generation support for half-float arithmetic is planned, but not yet operational as of this writing. Consequently, accelerated half-float operations need to be invoked as custom operations.

So as to facilitate easier use of halfs, a C++ utility class named *half* was included in the test suite. The class has overloaded arithmetic operators that invoke the half-float operations. However, this method is inferior to full LLVM-level support. For instance, the compiler cannot replace custom operations called in this way with equivalent operations. If a program uses halfs, it therefore cannot be compiled on a processor without hardware acceleration.

5.4.2 Miscellaneous Units

Some half-precision units have emerged from earlier research in collaboration with researchers from the University of Oulu. As part of this thesis, they were integrated into the TCE codebase. These include an adder, a multiplier and an inverse square root unit.

The adder and multiplier components are based on a light version of the VHDL-2008 Support Library with unused library infrastructure removed. The inverse square root unit computes an approximation of the inverse square root with a single Newton's iteration.

The components are designed for low-power applications with a low clock frequency. Though half-precision arithmetic is much more simple than single-precision, the latencies are insufficient for e.g. $\approx$ 200MHz on Stratix II. For high performance arithmetic, either the multiply-adder should be used, or the single-precision adder and multiplier should be parametrized for half-precision.

5.4.3 Float-Half Converter

This unit implements conversion operations between singles and halfs. Computation of exponents required special care, since the special case exponents remain all-ones or all-zeros after conversion, but all other exponents need to be rebiased. Significands can simply be truncated when converting to half-precision, and zero-padded when converting to single-precision.

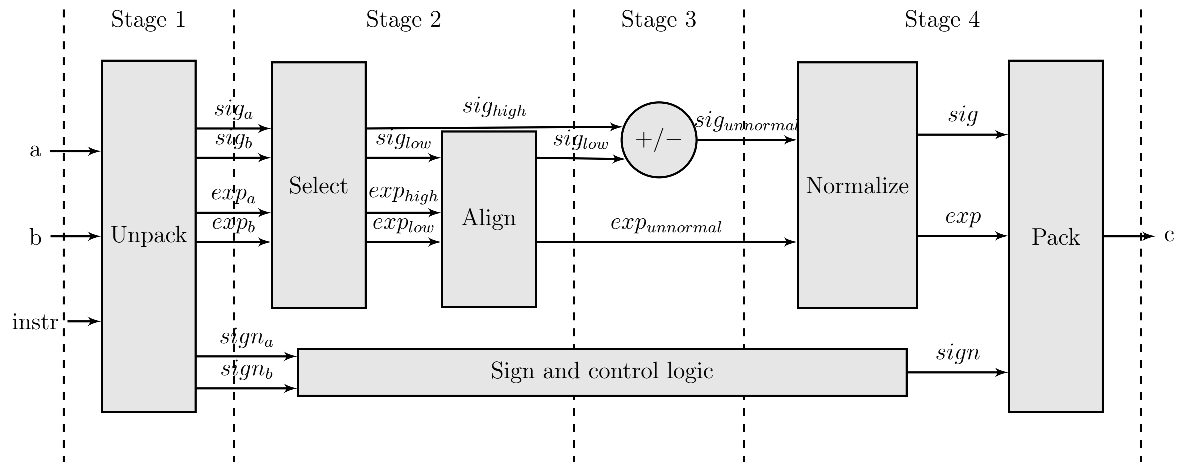
5.4.4 Comparator

The half-precision comparator is identical to the single-precision comparator, except that operation codes had to be reordered. TCE specifies that the operation codes must be in alphabetic order. Usually this does not require changes between single- and half-precision units, however in this unit the negation and not-equal operations (NEF and NEGF in the single-precision case, NEH and NEGH in the half-precision case) had to be swapped.

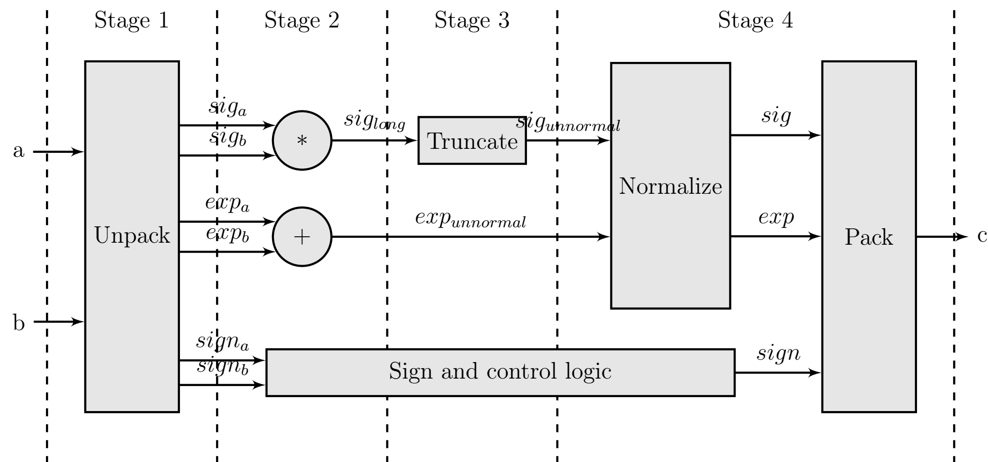
5.4.5 Fused Multiply-Adder

The half-precision fused multiply-adder is identical to the single-precision component described above, except for the operation codes. The exponent width and mantissa width were specified using generic parameters.

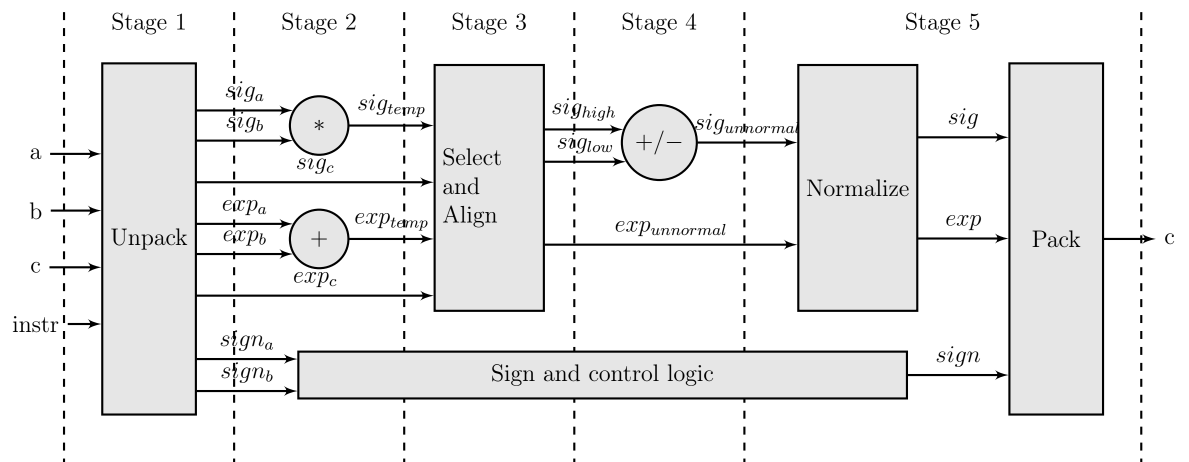
Of course, given the lower hardware complexity of half-precision operations, the unit is smaller than its single-precision cousin, and may achieve the same clock speed with fewer pipeline stages.



(a) Block diagram of the implemented single-precision adder.

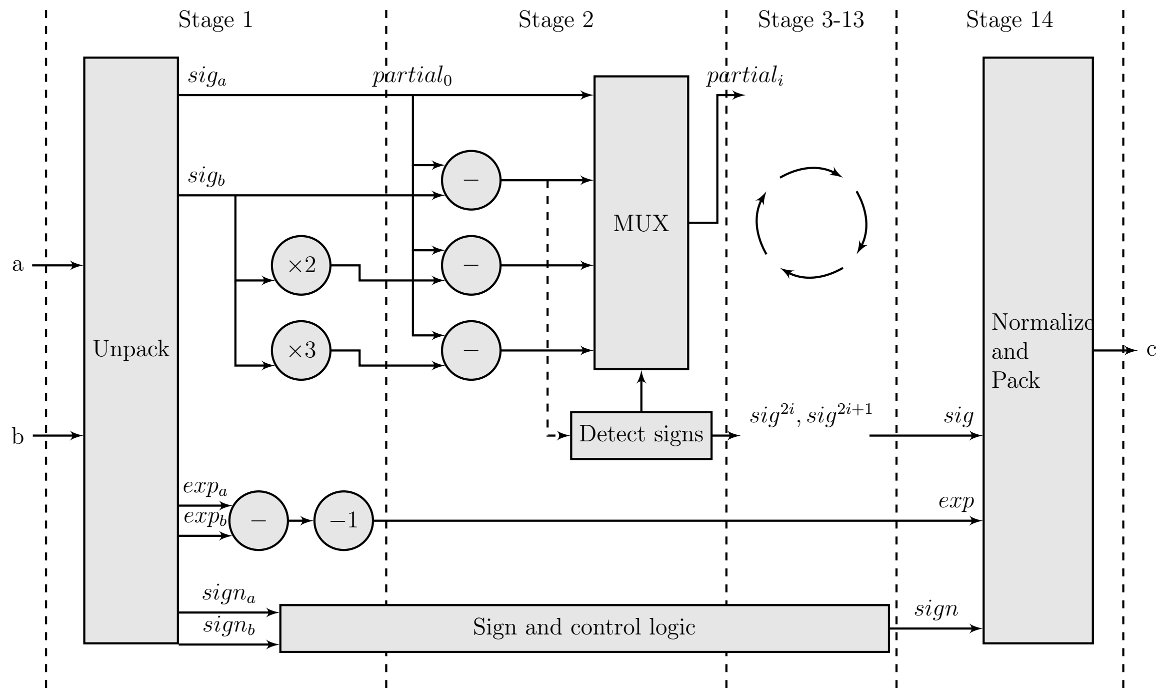


(b) Block diagram of the multiplier component.

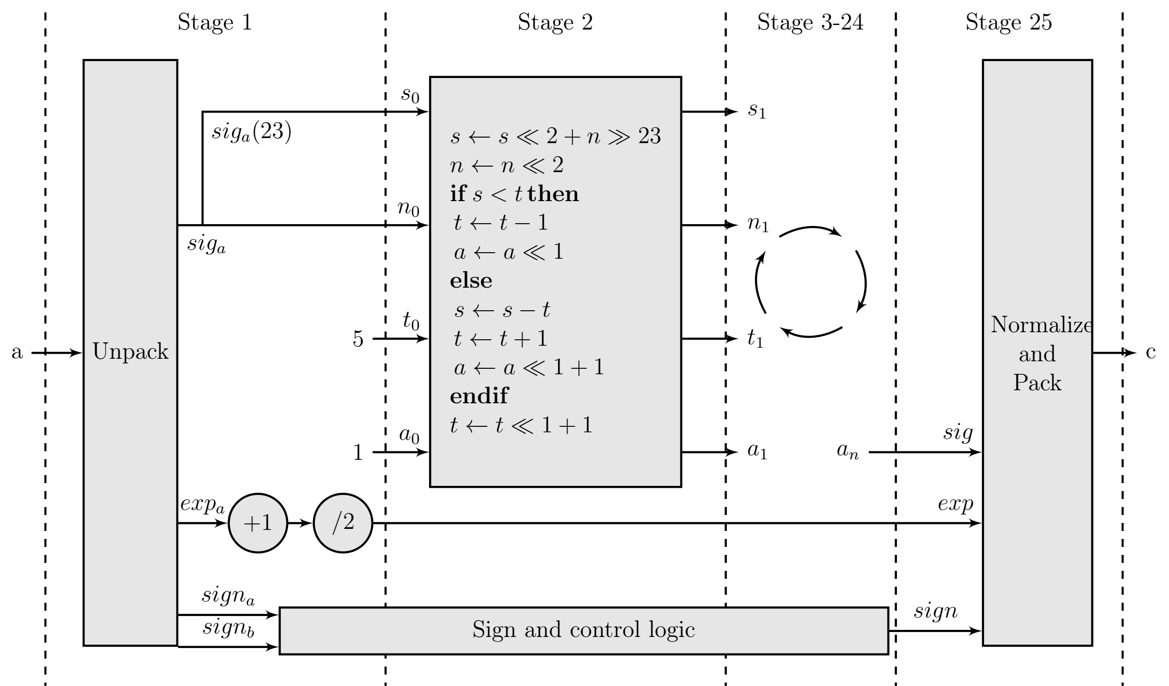


(c) Block diagram of the implemented floating-point fused multiply-adder.

Figure 5.1: Block diagrams of the implemented floating-point accelerators for basic arithmetic. Dashed lines indicate pipeline stages.



(a) Block diagram of the floating-point divider. The second stage is repeated 11 times for a single-precision float; each iteration computes two bits of the significand.



(b) Block diagram of the implemented square root unit. The second stage is repeated 23 times for a single precision float; each iteration computes one bit of the significand.

Figure 5.2: Block diagrams of the implemented floating-point accelerators for complex operations. Dashed lines indicate pipeline stages.

6. FMA ACCELERATED SOFTWARE OPERATIONS

As discussed in section 3.4, an IEEE-compliant FMA unit allows fast software algorithms for correctly rounded division and square root. Therefore, an FPU built around a FMA unit often does not include separate dividers or square root accelerators. For instance, the Intel Itanium processor family takes this approach. Itanium software developers are expected to implement division and square root using the algorithms described in [45] and [32].

It would be desirable to have the same option in TCE, given that dedicated divider and square root units are expensive. In most applications, division and square root are relatively rare operations and may not warrant dedicated hardware. The default behavior in TCE of integer-based emulation with *softfloat* [46], a floating-point emulation library, is extremely slow. In contrast to dedicated units which are inactive 'dark silicon' when not in use, performance of special operations could be increased by adding fused multiply-adders which are useful for many types of computation.

However, the algorithms are derived with an IEEE-compliant, round-to-nearest FMA operation in mind, and it is questionable whether they work at all on our relaxed FMA unit. It is at least difficult to attain correctly rounded results. This chapter investigates the matter and proposes reasonably fast software algorithms which, with the help of some cheap special instructions, stay within the OpenCL EP accuracy bounds: ± 2.5ulp for division and ± 3ulp for square root [18]. In the future, the same approach may be extended to computing, e.g., trigonometric or exponential functions that can be approximated with power series. The algorithms in this chapter are designed for single-precision floats, but may be extended for half-precision in the future.

The IBM Cell processor uses similar reduced standard compliance, and therefore has to deal with the same accuracy issues. The SIMDmath arithmetic library supplied with the Cell computes a less accurate square root than the algorithm proposed below. Its division appears to be more accurate, though it was tested with fewer numbers over a more limited range. [36]

6.1 Division

The division algorithms in [45] are derived for double-precision arithmetic. Some paraphrasing produces three promising single-precision algorithms. The first one, labeled DivideFast, starts with a fast approximation y of the reciprocal of b; computes a rough estimate of the quotient by multiplying $q_0 = ay$; and then refines the estimate with two Goldschmidt's iterations, a variation on Newton-Raphson iteration. Since Goldschmidt's method does not refer back to the original input data, rounding errors accumulate with each intermediate step. Correctly rounded results are attained in [45] only by performing the last iteration with double accuracy. The initial approximation is computed with a special hardware instruction, which can be implemented using a small lookup table [47].

Algorithm 1: DivideFast

Data: Single-precision float dividend a and divisor b
Result: The quotient a/b
begin
- $a', b', c \longleftarrow$ InitializeDivision(a, b)
- $y \longleftarrow$ ReciprocalApproximation(b)
- $q_0 \longleftarrow a' \cdot y; \quad e \longleftarrow 1 - b' \cdot y$
- $q_1 \longleftarrow q_0 \cdot e + q_0; \quad e_1 \longleftarrow e \cdot e$
- $q_2 \longleftarrow q_1 \cdot e_1 + q_1$
- $q_2' \longleftarrow$ MultiplyByPowerOfTwo(q_2, c)
- **return** q_2'

The second algorithm, labeled DivideMedium is identical except that the final Goldschmidt iteration is replaced with a Newton-Raphson iteration meant to eliminate the accumulated rounding errors. The drawback is that the Newton-Raphson does not parallelize as well, growing the critical path by one FMA latency.

Algorithm 2: DivideMedium

Data: Single-precision float dividend a and divisor b
Result: The quotient a/b
begin
- $a', b', c \longleftarrow$ InitializeDivision(a, b)
- $y \longleftarrow$ ReciprocalApproximation(b)
- $q_0 \longleftarrow a' \cdot y; \quad e \longleftarrow 1 - b' \cdot y$
- $q_1 \longleftarrow q_0 \cdot e + q_0$
- $r \longleftarrow a' - b' \cdot q_1$
- $Q \longleftarrow r \cdot y + q_1$
- $Q' \longleftarrow$ MultiplyByPowerOfTwo(Q, c)
- **return** Q'

The last algorithm, labeled DivideSlow, takes a different approach of first estimating an accurate reciprocal of b with Goldschmidt iterations and then using it to estimate $\frac{a}{b}$ through a Newton-Raphson iteration. An interesting feature is that the algorithm could be used to accelerate repeated divisions with the same divisor, by performing only the Newton-Raphson iteration for each successive division. In addition, the first half of the algorithm could be used when computing reciprocals instead of full division, saving one operation compared to DivideFast.

Algorithm 3: DivideSlow

Data: Single-precision float dividend a and divisor b
Result: The quotient a/b
begin
- $a', b', c \longleftarrow$ InitializeDivision(a, b)
- $y_0 \longleftarrow$ ReciprocalApproximation(b)
- $e \longleftarrow 1 - b' \cdot y_0$
- $y_1 \longleftarrow y_0 \cdot e + y_0; \quad e_1 \longleftarrow e \cdot e$
- $y_2 \longleftarrow y_1 \cdot e_1 + y_1$
- $q \longleftarrow a' \cdot y_2$
- $r \longleftarrow b' \cdot q + a'$
- $Q \longleftarrow r \cdot y_2 + q$
- $Q' \longleftarrow$ MultiplyByPowerOfTwo(Q, c)
- **return** Q'

Each algorithm often produced large errors when confronted with very large or very small inputs, due to over- and underflows. In particular, the quotient of two large numbers was often flushed to zero. The Itanium documentation advises developers to use extended-precision 'register floats', so this may be a feature of the algorithms, but the lack of subnormal number support probably contributes to the problem.

The problem was worked around by introducing a preprocessing step before the computation which sets the exponents of each input to zero, and a postprocessing step which restores the correct scale. This is possible because the significand in division can be computed independently of the exponent. Both operations are simple to implement in hardware as one-cycle special instructions, since they mostly involve exponent manipulation. The postprocesssing step can be formulated as a multiply-by-power-of-two instruction which may have more general use. The preprocessing step could in principle be fused with the reciprocal-approximation operation, possibly saving one cycle, but for the sake of clarity they are assumed to be separate instructions.

The accuracy of each algorithm was investigated using a simulated FMA that has the same relaxations as the hardware FMA, that is, results are rounded to zero, and

Table 6.1: Accuracy and performance characteristics of each division algorithm, compared to a hardware divider. Accuracy is measured with 10,000,000 random float pairs except for the HW divider. D_{FMA} is assumed to be 6 cycles.

Procedure	DivideFast	DivideMedium	DivideSlow	HW Divider
Avg. Error (ulp)	0.58	0.21	0.22	-
Max. Error (ulp)	4	2	2	1
OpenCL EP compliant	no	yes	yes	yes
FMA count	5	5	7	-
FMA latency	3	4	6	-
Latency	21	27	39	16
Throughput w/ 2 FMA	0.4	0.4	0.29	1
Min. LUT size	6x6	8x8	6x6	-

subnormal inputs and outputs are flushed to zero. 10,000,000 randomly generated pairs of floats were divided, and the results compared to native IEEE-compliant division. Since the native division is within ±0.5ulp of the correct result, the goal is a maximum observed error of 2ulp. The results are shown in Table 6.1. As expected, DivideFast is not OpenCL EP compliant even with preprocessing. However, the preprocessing step eliminates a class of potentially harmful errors, where the quotient of two large numbers is flushed to zero. DivideMedium and DivideSlow both appear EP compliant. Surprisingly DivideMedium requires a larger, 8x8 lookup table to produce sane results, while a 6x6 table suffices for DivideSlow and DivideFast.

Since exhaustive search is impossible, an airtight verification of DivideMedium and DivideSlow would require involved theoretical work similar to [45], which is outside the scope of this thesis. However, the algorithms are more exhaustively tested than the hardware components described in this chapter, owing to the slow speed of RTL simulation. In terms of performance, the operations are slower than the hardware divider, but in the same order of magnitude. Therefore, it appears reasonable to use either DivideSlow or DivideMedium as the default TCE division algorithm when no hardware divider is available, and to provide DivideFast as a lossy optimization, for instance with a compiler flag or as a custom operation. Notably, the IEEE-compliant division procedure in [32] requires ten FMA operations, twice as many as DivideMedium.

Moreover, OpenCL provides an interface for even broader approximation with the half_div, half_sqrt, etc. functions, which have very loose error bounds of ±8192ulp. These functions might be implemented with just one Goldschmidt iteration of two or three FMA operations, respectively.

6.2 Square Root

For square root, [45] provides a double-precision square root algorithm, which was modified into a single-precision operation by reducing the amount of iterations:

Algorithm 4: SquareRoot

Data: A single-precision float a
Result: The square root of a
begin
$b', c \longleftarrow \text{InitializeSquareRoot}(b)$
$y_0 \longleftarrow \text{InverseSquareRootApproximation}(b)$
$g \longleftarrow b \cdot y; \quad h \longleftarrow 1/2 \cdot y$
$r \longleftarrow 1/2 - h \cdot g$
$g_1 \longleftarrow g \cdot r + g; \quad h_1 \longleftarrow h \cdot r + h$
$d \longleftarrow g_1 \cdot g_1 + b$
$g_2 \longleftarrow h_1 \cdot d + g_1$
$g_2' \longleftarrow \text{MultiplyByPowerOfTwo}(g_2, c)$
return g_2'

As with division, the square root procedure often fails when given inputs near the edges of the single-precision dynamic range. Again, a cheap workaround is possible. A crucial difference is that while significand computation is completely independent of the exponent when dividing, in the square root operation it depends on the LSB of the exponent. That is, multiplying the input by a power of four preserves the output significand, but multiplying by a power of two may change it. For this reason the inverse-square-root-approximation operation also requires separate tables for even and odd exponents.

For the correction steps, the input is multiplied by a suitable power of four 2^{-2n} that sets the exponent to 0 or 1, the square root is taken, and finally the result is multiplied by 2^n. The postprocessing step can be implemented with the same "multiply by power of two" function that was used for division.

Since the square root operation has only one input, it was easily verified by iterating through all single-precision floats. As shown in Table 6.2, the algorithm has a maximum error of 1ULP compared to an IEEE-compliant native square root, which is well within the OpenCL EP accuracy bounds. The performance is similar to DivideSlow in the previous chapter. Given that the dedicated square root unit has a latency of 26, the software implementation appears very competitive.

6.3 Accelerator Component

The division and square root functions presented in this chapter depend on five custom operations, which are:

Table 6.2: Accuracy and performance information on the Square Root algorithm, based on exhaustive search of all single-precision floats.

Procedure	Software	Hardware
Avg. Error (ulp)	0.26	-
Max. Error (ulp)	1	1
OpenCL EP compliant	yes	yes
FMA count	7	-
FMA latency	5	-
Latency	33	26
Throughput w/ 2 FMA	0.29	1
Min. LUT size	7x7	-

initialize-division: Performs the preprocessing step used in the division algorithms.

initialize-square-root: Performs the preprocessing step of the square root algorithm.

reciprocal-approximation: Computes a fast LUT-based approximation of the reciprocal of the input.

inverse-square-root-approximation: Computes a fast LUT-based approximation of the inverse square root of the input.

multiply-by-power-of-two: Takes two floating-point inputs, a and b, and performs a correct multiplication with the assumption that b is either a power of two, 0, ∞ or NaN. That is, the significand of b is only used to differentiate between ∞ and NaN, and otherwise ignored. The operation is used for the postprocessing steps described above, and may have general use in other applications.

Ideally, these operations would be provided in a function unit which would add support for both software operations in a single convenient package. Since each individual operation is simple to implement, the FU could be expected to perform well with a latency of one cycle.

7. VERIFICATION AND BENCHMARKING

This chapter describes how the implemented function units were evaluated. The next section lays out the process by which the correct operation of the function units was verified. The following sections investigate the hardware characteristics of the function units as synthesized on two FPGA platforms.

TCE includes an automated system-level test suite called *systemtest*. It is intended to act as a "smoke test" for regressions by stressing various aspects of TCE functionality. As a guideline, before any changes are committed to the project's version control repository, systemtest should be run without errors. It therefore is a natural site for a set of automated tests that verify the correct operation of the functionality implemented in this thesis.

7.1 Instruction-Level Simulator Test

As discussed in Chapter 2, TCE includes an instruction-level simulator named *ttasim* [16] and the TCE OSAL contains C++ simulation models for each operation. A test in the systemtest suite called *BaseOperations* verifies that the simulation models of the operations in the base operation set supplied with TCE function correctly with a number of test inputs. Since this thesis made additions into the base operation set, including a fused multiply-add operation and half-precision arithmetic operations, corresponding trials had to be inserted into BaseOperations. This test offers no insight into the correct function of the implemented hardware units, only the corresponding simulation models.

7.2 VHDL Simulator Tests

The TCE Processor Generator can automatically generate shell scripts for simulating the processor on the open source VHDL simulator, *GHDL* [14]. Some test cases verify correct behavior by printing a log of bus values at each timestep and checking that the result is equal to *ttasim* simulation. Since the *ttasim* simulation models have IEEE compliant floating-points, results of various FP operations are instead checked in the program and printed to a text file.

The final single-precision and half-precision tests include separate test cases for the FMA components, since if they were to be tested in the same processor with the adder and the multiplier, any given test could be scheduled in either component.

The Sabrewing wrapper was tested separately for the same reason. In total, five test cases were implemented: three single-precision tests (separate accelerators, FMA unit, and Sabrewing) and two half-precision tests (separate accelerators and FMA unit).

Each test case is a shell script that generates a processor with floating-point units, compiles a test program for the processor, and simulates the processor the processor at RTL level, running the test program, using GHDL. Different areas of functionality are tested with trials that involve both typical inputs and corner cases. The output of the program has one character for each trial, 'O' for successes and 'N' for failures. More verbose error messages would be expensive to simulate; therefore any errors are better debugged by e.g. by examining wire activity logged by GHDL. The output is divided into lines by test caregory, which are as follows:

- robust tests which should pass even with nonstandard floating-point precisions,
- addition and subtraction,
- multiplication,
- division (skipped for half-precision tests),
- square root (skipped for half-precision tests),
- conversion, and
- pipeline behavior.

To ensure that the test checks for correct pipeline behavior, the test processor is configured with a four-bus interconnection network that is sufficient to feed an FPU with its full set of inputs and read its output on every cycle.

7.3 Hardware Synthesis Benchmark

In this section, the floating-point units are benchmarked based on their area and performance when synthesized on two FPGA platforms. Floating-point units supplied by the vendors of each FPGA are shown as references.

7.3.1 Altera Stratix-II FPGA

The Stratix-II EP2S180F1020C3 is included in the benchmark as an example of an older high performance FPGA by the major vendor Altera. TCE also has a Platform Integrator to facilitate processor integration to it [17]. Synthesis flags were selected to favor clock speed.

Synthesis results on the Stratix-II are shown in Table 7.1. All 32-bit operations achieve a clock rate of around 180MHz, which is close to the clock rate of a minimal FPU-less processor and was, therefore, considered a success.

Since the latency of the FMA component is parametrized with flags that separately disable any of four pipeline stages, the best possible flags for each latency had to be determined by synthesizing the component with each possible combination. The reported values are those with the highest attained clock rate. The FMA component is clearly smaller than separate adders and multipliers in terms of LUTs, but requires one more pipeline stage to reach equivalent clock rate. It is larger in terms of registers, but they appear to be a less scarce resource. Consequently, lower power consumption could be expected. The half-precision units require less than half the area of their single-precision counterparts, and achieve the same clock speed at lower latencies.

Altera's adder and multiplier megafunctions [43] are shown as points of comparison. The implemented FPUs come reasonably close to Altera's units in terms of performance, given that Altera's units are carefully optimized for the specific platform. Notably, the FMA unit can complete a multiply-add operation approximately 40% faster than a combination of Altera's units using roughly the same area. Surprisingly, the divider unit shows a clear improvement over Altera's unit, producing results at a halved latency without sacrificing nearly any clock rate or area economy.

The divider and square root units are very large compared to the other components, which lends credibility to the software-based approach described in the previous chapter. Together the components displace as many logic cells as five fused multiply-accumulators, which could in principle match the division throughput of the dedicated divider, and nearly match the square root throughput of the dedicated unit, while greatly increasing the processing power available for floating-point arithmetic in general. Of course, this analysis disregards the cost of interconnecting the FMA units so that they actually reach the theoretical throughput. As expected, the FPUs implemented in this thesis significantly outperform the wrapped Sabrewing FPU, which has been developed for ASIC technologies.

7.3.2 Xilinx Virtex-6 FPGA

The Xilinx Virtex-6 XC6VSX315T-2FF1759 is included as an example of a more modern, high-end FPGA and to represent both major FPGA vendors. It is also a platform of interest for prototyping future TCE applications.

As with Stratix-2, reasonably simple synthesis flags that favor clock speed were selected. These included a high speed rating of "-3", setting the "Design Goal" to "Timing Performance", and selecting a "Performance without IOB Packing" strategy, which assumes that the synthesized component is for on-chip use, and will not

Unit	Latency	Frequency(MHz)	Registers	ALUTs	DSP
General information					
Total resources	-	-	143,520	143,520	768
Minimal	-	196.31	883	1040	0
Altera floating-point units					
add [48]	7	260.00	347	613	0
mul [49]	5	228.00	148	126	?
div [50]	33	231.00	1854	1442	0
32-bit units					
add	5	180.73	419	652	0
mul	5	180.08	324	339	8
div	15	190.99	1303	1526	0
sqrt	26	194.74	1585	3201	0
conv	4	185.39	81	460	0
cmp	1	187.65	188	169	0
fma	4	127.63	452	868	8
fma	5	157.33	618	879	8
fma	6	179.21	727	854	8
sabrewing	4	70.55	937	2724	16
sabrewing	5	72.25	1015	2672	16
16-bit units					
add	2	71.06	82	267	0
mul	2	71.60	122	183	2
fma	3	131.41	204	321	2
fma	4	182.92	209	307	2

Table 7.1: Synthesis results on an Altera Stratix-2 FPGA. LUT and register counts were obtained by synthesizing each FPU connected to a minimal processor, and subtracting the size of that processor, which is shown as "Minimal". Note that the selected model of Stratix-II has very large resources; most practical applications would opt for a smaller, less-expensive Stratix-II. This overprovisioning has no effect on the figures shown here.

be connected to IO pins.

Synthesis results are shown in Table 7.2. The results reflect that the newer Xilinx has otherwise much faster logic than the Stratix-2 but operations that involve multiplication are surprisingly slow. While all other operations reach clock speeds of 240 MHz or higher, the multiplier and FMA units cut off at 180 MHz. Moreover, the FMA unit reaches this clock speed with a latency of 4, and does not improve with the addition of two more pipeline stages. This suggests that the automatically instantaniated hardware multiplier is the bottleneck. The synthesis tool fails to either pipeline the operation or break it into smaller multiplications for register retiming. Platform-specific optimization may be necessary in order to attain a higher clock rate.

The half-precision units are once more smaller and faster than the corresponding single-precision units. They also are less affected by the multiplier bottleneck, as they require only a 22-bit integer multiplication compared to 48 bits for single-precision. Interestingly, the low-power half-precision adder and multiplier units, which had severely limited clock rates on the Altera, are here faster than the corresponding single-precision operations.

Xilinx's own vendor-supplied floating-point IP is shown as a reference. It reaches a much higher clock frequency, by more than doubling the number of pipeline stages. Fine-grained pipelining appears to be necessary in order to attain maximum throughput on this platform. Considering the difference in latency, all the FPUs implemented in this thesis appear to perform well, except for the square root unit, which Xilinx manages to implement in a much smaller area. The Sabrewing has a more competitive performance on the Xilinx than on the Altera, but is still very large in terms of area.

Unit	Latency	Frequency(MHz)	Registers	ALUTs	DSP
General information					
Total resources	-	-	393,600	196,800	1,344
Xilinx floating-point units					
add [41]	12	476.00	?	498	0
mul [41]	11	408.00	?	160	2
div [41]	28	429.00	?	929	0
sqrt [41]	28	384.00	?	645	0
32-bit units					
add	5	244.14	527	634	0
mul	5	187.62	286	247	2
div	15	239.46	1339	1280	0
sqrt	26	245.04	993	2723	0
conv	4	298.77	371	513	0
cmp	1	260.01	132	94	0
fma	3	155.11	481	711	2
fma	4	188.61	671	671	2
fma	5	186.36	688	688	2
fma	6	187.51	682	682	2
sabrewing	4	123.42	1023	2751	4
sabrewing	5	151.37	1221	2662	4
16-bit units					
add	2	252.21	124	344	0
mul	2	227.69	88	113	1
fma	3	200.72	301	392	1
fma	4	231.11	230	304	1

Table 7.2: Synthesis results on a Virtex-6 FPGA. Note that the selected model of Virtex-6 has very large resources; most practical applications would opt for a smaller, cheaper Virtex-6. This overprovisioning has no effect on the figures shown here.

8. CONCLUSIONS

A set of floating-point function units was designed and implemented for the TCE toolset, which cover the basic operations needed for floating-point computation. They are based on the VHDL-2008 Support Library and described in high-level VHDL that is readily extensible to e.g. IEEE-compliant or non-standard precision arithmetic. The FUs are optimized for performance by complying to the looser OpenCL EP standard instead of the ubiquitous IEEE 754. The freely available Sabrewing FPU [30] was included as an alternative geared for ASIC performance.

An automated test suite was written and integrated into TCE's *systemtest* facility. The suite verifies the correct behavior of the simulation models for each floating-point operation using instruction-level simulation and the hardware units themselves using register transfer level simulation.

The implemented FUs can be divided into a set of separate accelerators for each arithmetic operation and a fused multiply-add unit, which shares hardware between addition and multiplication. The common algorithms for correctly-rounded FMA-accelerated division and square root assume the default IEEE-754 rounding and consequently malfunction on the Round-to-Zero FPUs described in this thesis. Modified procedures were proposed that are not quite correctly rounded, but at least OpenCL EP-compliant, with the help of cheap special instructions. Dedicated divider and square root units were found to be large enough to justify their replacement with software emulation in most cases. An interesting result is that the proposed OpenCL EP-compliant division requires only five multiply-add operations, as opposed to ten for IEEE-compliant division [32]. Hence, the benefits of relaxed standard compliance appear to be more pronounced in software operations than in hardware units.

The FUs were found to attain reasonably high performance when synthesized on an Altera Stratix-2 FPGA, being not much larger and slower than Altera's own platform-specific floating-point units. On a Xilinx Virtex-6 FPGA, the vendor-provided FPUs reached a much higher clock rate using fine-grained pipelining. However, the new FPUs may be worth considering even on the Virtex-6 in applications that benefit from an FMA instruction, if the clock rate is limited by other considerations.

Separate function units and instruction-level support was included for IEEE half-precision arithmetic. However, the half operations currently lack high-level language

support pending additions to LLVM, and can only be used as assembly instructions. Half-precision units were found to require less than half the area of single-precision units, and obtain the same clock speed at a more favorable latency. A part of the floating-point functionality described in this thesis was included in the 1.6 release of TCE. The rest, including half-precision components, can be found in the version control repository at tce.cs.tut.fi, and will be included in TCE 1.7.

In conclusion, this thesis successfully added hardware accelerated floating-point support to the TCE toolset. It is questionable whether the main design requirement of FPGA performance was met even by relaxing standards compliance. Especially the Virtex-6, and by extension other recent FPGAs require an amount of pipelining that was not anticipated. Inserting further pipeline stages would be a difficult engineering task. The original choice to base the units on the Support Library may have been suspect, since the library is not designed for high performance. In retrospect pipelining and optimizing the units required a design effort comparable to designing new FPUs from the ground up.

However, the existing implementations reviewed in Chapter 4 did not include many feasible alternatives, and it is uncertain whether a more thorough review would have helped. Many FPU architectures have been proposed in the literature, but few have public HDL implementations. The OpenCores single-precision units are optimized for old hardware, and likely have similar performance characteristics. The FloPoCo system is reported to be clearly superior in terms of performance and flexibility but due to licensing issues, it cannot address the main issue in this thesis, namely the lack of floating-point units in TCE.

One alternative would have been to wrap the vendor-supplied FPUs and include, e.g., the *fpu100* unit from OpenCores as a slow, platform-independent alternative. This approach is attractive in terms of performance and design effort, but the lack of an FMA unit would remove the option of using the software-based divider and square root operations described in Chapter 6. In addition, nonstandard floating-point formats would have been difficult to support across the board. For these reasons, this thesis seems to be have addressed the issue of floating-point support in TCE in an effective way.

Short-term future work will include further evaluation of accelerated software operations and implementation of the accelerator block proposed in Chapter 6. A point of interest is whether even faster algorithms are possible for reduced-precision arithmetic. Another interesting direction of study would be to automatically wrap custom operators generated with the FloPoCo system into TCE function units, even though processors so produced would be restricted by licensing issues. In the future, the implemented FPUs will be used to investigate the feasibility of using FPGA-based TTA processors for low power high performance floating-point computing.

BIBLIOGRAPHY

[1] L. Nurmi, P. Salmela, P. Kellomäki, P. Jääskelainen, and J. Takala, "Reconfigurable video decoder with transform acceleration," in *Proc. IEEE Signal Processing Systems Workshop*, Tampere, Finland, Oct. 7–9 2009, pp. 081–086.

[2] P. Pereira and K. Savio, "Characterization of FPGA-based high performance computers," Master's thesis, Virginia Tech, USA, 2011.

[3] C. de La Lama, P. Jääskeläinen, and J. Takala, "Programmable and scalable architecture for graphics processing units," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, vol. 5657, pp. 2–11.

[4] J. Janhunen, P. Salmela, O. Silvén, and M. Juntti, "Fixed- versus floating-point implementation of MIMO-OFDM detector," in *Proc. IEEE Int. Conf. Acoustics, Speech and Signal Processing*, Prague, Czech Republic, May 22–27 2011, pp. 3276–3279.

[5] T. Nyländen, J. Janhunen, J. Hannuksela, and O. Silvén, "FPGA based application specific processing for sensor nodes," in *Proc. Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 18–21 2011, pp. 118–123.

[6] D. W. Bishop, "VHDL-2008 support library," 2011. [Online]. Available: http://www.eda.org/fphdl/

[7] H. Corporaal, "Transport triggered architectures; design and evaluation," Doctoral dissertation, Delft Univ. of Technology, Netherlands, 1993.

[8] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[9] V. Betz, "FPGA challenges and opportunities at 40nm and beyond," in *Proc. Int. Conf. Field Programmable Logic and Applications*, Prague, Czech Republic, Aug. 31 – Sept. 2 2009, p. 4.

[10] G. J. Lipovski, "The architecture of a simple, effective, control processor," in *Second Annual Euromicro Symposium*, 1976.

[11] H. Corporaal and P. Arend, "Move32int, a sea of gates realization of a high performance transport triggered architecture," *Microprocessing and Microprogramming*, vol. 38, pp. 53–60, 1993.

[12] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, San Jose, CA, USA, Jan. 29–30 2007, pp. 65 070X–1 – 65 070X–11.

[13] L. Laasonen, "Program image and processor generator for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, 2007.

[14] T. Gringold, "GHDL," 2011. [Online]. Available: ghdl.free.fr

[15] V.-P. Jääskeläinen, "Retargetable compiler backend for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, 2011.

[16] P. Jääskeläinen, "Instruction set simulator for transport triggered architectures," Master's thesis, Tampere University of Technology, Finland, 2005.

[17] O. Esko, "ASIP integration and verification flow," Master's thesis, Tampere University of Technology, Finland, 2011.

[18] A. Munshi, "The OpenCL specification version: 1.2 document revision: 15," Khronos, 2011.

[19] P. Jääskeläinen, C. S. de La Lama, P. Huerta, and J. Takala, "OpenCL-based design methodology for application-specific processors." in *Proc. Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 19–22, pp. 223–230.

[20] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*, 1st ed. New York, NY, USA: Birkhäuser Boston, 2009.

[21] D. Goldberg, "What every computer scientist should know about floating point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[22] C. Inacio and D. Ombres, "The DSP decision: Fixed point or floating?" *Spectrum, IEEE*, vol. 33, no. 9, pp. 72–74, Sept. 1996.

[23] *Standard for Floating-Point Arithmetic*, IEEE Std. 754, 2008.

[24] W. Kahan and J. Palmer, "On a proposed floating-point standard," *SIGNUM Newsl.*, vol. 14, no. si-2, pp. 13–21, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/1057520.1057522

[25] W. Cody, "Analysis of proposals for the floating-point standard," *Computer*, vol. 14, no. 3, pp. 63–68, 1981.

[26] H. Seetzen, W. Heidrich, W. Stuerzlinger, G. Ward, L. Whitehead, M. Trentacoste, A. Ghosh, and A. Vorozcovs, "High dynamic range display systems," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 760–768, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015797

[27] C. Maass, M. Baer, and M. Kachelriess, "CT image reconstruction with half precision floating-point values," *Medical Physics*, vol. 38, no. S1, pp. S95–S105, 2011.

[28] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," in *Proc. IEEE/ACM Int. Conf. Computer-aided design.* San Jose, CA, USA: IEEE Computer Society, Nov. 9–13 2003, pp. 275–.

[29] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs," Technical report, 2011. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

[30] T. M. Bruintjes, K. H. G. Walters, S. H. Gerez, B. Molenkamp, and G. J. M. Smit, "Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 41:1–41:22, Jan. 2012.

[31] W. Kahan, "Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic," World-Wide Web document, p. 30, Oct. 1997. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

[32] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium-Based Systems.* Hillsboro, OR, USA: Intel Press, 2002.

[33] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong, "The vector floating-point unit in a synergistic processor element of a CELL processor," in *Proc. IEE Int. Symp. Comput. Arithmetic*, Cape Cod, MA, USA, June 27–29 2005, pp. 59–67.

[34] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *Micro, IEEE*, vol. 31, no. 2, pp. 6–15, March-April 2011.

[35] E. Strohmaier and J. D. Meuer, "Top500 supercomputer sites," University of Tennessee, Knoxville, TN, USA, Tech. Rep., 1997.

[36] "Accuracy information for the MASS libraries for Cell/B.E. SPU," Tech. Rep., 2009. [Online]. Available: http://www-01.ibm.com/support/docview.wss?uid=swg27009549

[37] J. Jean and S. Graillat, "A parallel algorithm for dot product over word-size finite field using floating-point arithmetic," in *Int. Symp. Symbolic and Numeric Algorithms for Scientific Computing*, Timisoara, Romania, Sept. 23–26 2010, pp. 80–87.

[38] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah, "A 22nm IA multi-CPU and GPU system-on-chip," in *IEEE Int. Solid-State Circ. Conf. Digest of Technical Papers*, San Francisco, CA, USA, Feb. 19–23 2012, pp. 56–57.

[39] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, July-Aug. 2011.

[40] *FloPoCo User Manual*, INRIA. [Online]. Available: http://flopoco.gforge.inria.fr/flopoco_user_manual.html

[41] *LogiCORE IP Floating-Point Operator v6.0 Product Specification*, Xilinx, 2012.

[42] *LogiCORE IP Virtex-5 APU Floating-Point Unit v1.01a Product Specification*, Xilinx, 2011.

[43] *Floating Point Megafunctions User Guide*, Altera, 2011.

[44] T. Hain and D. Mercer, "Fast floating point square root," in *Proc. Int. Conf. Algorithmic Mathematics and Computer Science*, Las Vegas, Nevada, USA, June 20–23.

[45] P. Markstein, "Software division and square root using Goldschmidt's algorithms," in *Conf. Real Numbers and Computers*, Schloß Dagstuhl, Germany, Nov. 15–17 2004, pp. 146–157.

[46] J. Hauser, "Softfloat release 2b," 2002. [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[47] E. Schwarz and M. Flynn, "Hardware starting approximation for the square root operation," in *Proc. Int. Symp. Comput. Arithmetic*, Windsor, Canada, June 29 – July 2 1993.

[48] *Floating Point Adder/Subtractor (ALTFP_ADD_SUB) Megafunction User Guide*, Altera, 2007.

[49] *Floating Point Multiplier (ALTFP_MULT) Megafunction User Guide*, Altera, 2008.

[50] *Floating Point Divider (ALTFP_DIV) Megafunction User Guide*, Altera, 2008.

A. FLOATING-POINT UNIT SPECIFICATIONS

Table A.1: List of floating-point units. The latencies of fpu_sp_div and fpu_sp_sqrt depend on the parameters *mw* and *ew* which are defined in Table A.2. The fpu_sp_sabrewing supports the listed operations on both floats and integers.

Entity name	Latency	Supported operations
Single-precision units, fpu_embedded.hdb:		
fpu_sp_add_sub	5	Add, Subtract
fpu_sp_mul	5	Multiply
fpu_sp_div	$\frac{mw}{2} + 3 = 15$	Multiply
fpu_sp_sqrt	$mw + 3 = 26$	Square Root
fpu_sp_compare	1	Absolute Value, Negation, Comparisons
fpu_sp_convert	4	Convert Float $\leftrightarrow$ Signed/Unsigned Int
fpu_sp_mac_v2	$2 - 6$	Multiply-Add, Multiply-Subtract, Add, Subtract, Multiply
fpu_sp_sabrewing	$5 - 6$	Multiply-Add, Add, Multiply, Comparisons, Shift Left, Shift Right
Half-precision units, fpu_half.hdb:		
fpadd_fpsub	2	Add, Subtract
fpmul	2	Multiply
invsqrth	5	Inverse Square Root
fpu_chf_cfh	1	Convert Single $\leftrightarrow$ Half
fpu_hp_compare	1	Absolute Value, Negation, Comparisons
fpmac_v2	$2 - 6$	Multiply-Add, Multiply-Subtract, Add, Subtract, Multiply

Table A.2: List of generic parameters used to customize floating-point units. In addition, the single-half converter unit has parameters *smw*, *sew*, *hmw* and *hew* which replace *mw* and *ew* for single- and half-precision floats, respectively.

Parameter	Type	Description	Appears in
mw	Integer	Significand width	All units
ew	Integer	Exponent width	All units
dataw	Integer	Input data signal width, at least $mw + ew1$.	All units
busw	Integer	Output data signal width, at least $mw + ew1$.	All units
bypass_2	Boolean	Disables pipeline stage 2, used to customize latency.	FMA, Sabrewing
bypass_3	Boolean	Disables pipeline stage 3.	FMA
bypass_4	Boolean	Disables pipeline stage 4.	FMA
bypass_5	Boolean	Disables pipeline stage 5.	FMA