NTNU – Trondheim
Norwegian University of
Science and Technology

Configurable Floating-Point Unit for the SHMAC Platform

Audun Lie Indergaard

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Per Gunnar Kjeldsberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area at NTNU. SHMAC is planned to run in an FPGA and be an evaluation platform for research on heterogeneous multi-core systems. Due to battery limitations and the so-called Dark silicon effect, future computing systems in all performance ranges are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous systems.

The standard SHMAC processing tile only supports fixed-point calculations. For efficient programming, floating-point is preferred. In the general case this normally comes with a performance, energy and area overhead. For many applications it is not necessary to have a floating-point unit (FPU) that follows the complete IEEE standard. However, the overhead can then be reduced significantly. A possible trade-off would be a configurable FPU, e.g., with respect to bit width and exception handling.

The main parts of this assignment are as follows:

- Study the IEEE floating-point standard and its implementation.
- Study application specific FPU implementations and in particular any configurable FPU implementations found in the literature.
- Implement a simple FPU for use on the SHMAC platform and test this for selected software applications.
- Study the requirements of the selected software applications and look for FPU optimization possibilities.
- Implement one or more application specific and/or configurable FPUs.
- Evaluate performance and energy gains achieved as well as area results. If time allows, also compare with fixed-point implementations of the software applications.

Assignment given: January 15th 2014
Supervisor: Per Gunnar Kjeldsberg

Abstract

The use of floating-point hardware in FPGAs has long been considered infeasible or related to use in expensive devices and platforms. However, floating-point operations are crucial for many scientific computations and for efficient programming, floating-point is preferred. The IEEE Standard 754 for floating-point arithmetic provides a method that will yield the same results whether the processing is done in hardware, software or the combination of the two. However, the scope of this standard is much more comprehensive than what is needed for many systems and can cause a lot of overhead. This thesis presents ways to lower the power consumption, area usage and latency by using a configurable floating-point unit (FPU) with variable bit-width.

There is a linear relation between the bit-width of floating-point numbers and the dynamic power consumption, while there is an exponential relation between the bit-width and area consumption. If only a limited range and precision are needed, using a tailored FPU design can reduce the area and dynamic power consumption by up to 96%. Choosing the right FPU can also reduce the number of clock cycles per operation with up to 98%. For the applications analyzed, a maximum of 33% of the bit-width in floating-point numbers are unnecessary, and removing these leads to great performance and area gains. By analyzing the frequency the different operations are used in applications, some floating-point operations can be emulated in software and greater area and power savings can be accomplished.

Sammendrag

Flyttallsenheter i FPGAer har lenge vært lite hensiktsmessig og relatert til bruk i kostbare enheter og plattformer. Likevel er flyttallskalkulasjoner nødvendig for mange vitenskapelige beregninger, samt det blir lettere å programmere software for enheten. IEEE Standard 754 for flyttallsaritmetikk viser til metoder som vil gi riktige resultater uansett om det er designet for hardware, software eller en kombinasjon av de to. Derimot resulterer omfanget av denne mye ekstra kombinatorikk som er unødvendig for mange systemer. Denne rapporten presenterer måter å minke effektforbruket, arealet og forsinkelsen i systemet ved å bruke en konfigurerbar flyttallsenhet med variabel bitbredde.

Det er et lineært forhold mellom det totale antall bit brukt på flyttallet og det resulterende dynamiske effektforbruket, mens det er et eksponentielt forhold mellom bitbredden og arealet. Hvis kun en begrenset bitbredde og presisjon er nødvendig, kan en tilpasset flyttallsenhet redusere arealet og det dynamiske effektforbruket med opptil 96%. Ved riktig valg av FPU kan så mye som 98% av klokkesyklene for aritmetiske operasjoner bli redusert. For de analyserte applikasjonene, maksimum 33% av bitbredden til flyttallene er unødvendig og ved å fjerne disse kan ytelsen og arealet bedres. Ved å analysere frekvensen for forskjellige operasoner i applikasjonene, kan deler av flyttallsenheten bli emulert i software og mindre areal og effektforbruk kan oppnås.

Preface

This thesis was written at the Norwegian University of Science and Technology and is a part of the SHMAC research project initiated by EECS, which aims to investigate the challenges posed by heterogeneous computing system. It was written during the spring of 2014 and was chosen because optimization problems are challenging and a lot of research is done on the subject.

To approach this thesis, the *IEEE 754 Standard* was studied and I soon figured out that this standard is much more comprehensive than what is needed for many systems. After studying articles on the subject, I discovered that the bit-width of floating-point numbers have a big influence on the area and power consumption. This became my primary focus. By analyzing software, I found out that there is a significant difference between the rates of usage for the different arithmetic operations. Therefore, I explored the options of having some floating-point arithmetic in software.

Next, different floating-point units were explored. The floating-point unit from Xilinx and the floating-point library support variable bit-width. However, since the floating-point library is big and complex, I decided to design my own floating-point unit. I quickly discovered that the workload was bigger than expected, and only the adder, subtractor and multiplier were prioritized.

My next challenge was to implement an FPU on the SHMAC platform. However, the Amber core, which is implemented on the SHMAC platform, does not have hardware floating-point support. Another student is working on this task, but as of today, this is not yet implemented. To be independent of this problem, I decided to use the OpenRISC core, which does have hardware floating-point support. My next challenge was to find a Xilinx FPGA to implement the processing core. However, none of the institutes on the IME faculty did have available Xilinx FPGAs that was big enough to handle the OpenRISC. The only option was the ZedBoard using the Xilinx Zynq. I found an implementation of the OpenRISC for this system, but unfortunately many of the vital functions were removed. Other open source processing cores were considered, but none of these included the functionality I was looking for. As a result, I decided to abandon the implementation of the FPU on a processing core and focus more on testing and analyzing.

I would like to thank Per Gunnar Kjeldsberg and others working with the SHMAC project for help, feedback and guidance on this project.

Audun Lie Indergaard
June 11th 2014

Contents

List of Figures

List of Tables

List of Acronyms

SHMAC Single-ISA Heterogeneous MAny-core Computer

ISA Instruction Set Architecure

APB Advanced Peripherals Bus

FPGA Field Programable Gate Array

IP Intellectual Property

FPU Floating-Point Unit

FP Floating-Point

ASIC Application-Specific Integrated Circuit

NaN Not a Number

IP Intellectual Property

RISC Reduced Instruction Set Computing

DDR Double Data Rate

MMU Memory Management Unit

RTL Register-Transfer Level

DSP Digital Signal Processor

XST Xilinx Synthesis Technology

LUT LookUp Table

SoC System on Chip

SPEC Standard Performance Evaluation Corporation

CMU Communication Management Unit

NIST National Institute of Standards and Technology

HDL Hardware Description Language

VHDL Very High Speed Integrated Circuit HDL

DPC Dynamic Power Consumption

IO Input/Output

Chapter 1

Introduction

In the roughly 65 years since the first general-purpose electronic computer was created, computer technology has made incredible progress. According to Moore's law, the number of transistors on an integrated circuit doubles approximately every two years and this will, according to Pollack's rule, enable a new microarchitecture that delivers a 40% performance increase [1]. The rapid growth in microprocessor performance has been enabled by three key technology drivers; transistor-speed scaling, core microarchitecture techniques and cache memories [2]. In a Dennard scaling process the dimension of transistors are reduced, while the electric fields are held constant to maintain reliability. As transistors scales down, the supply voltage and threshold voltage scales to keep the electrical field constant. Since transistors are not a perfect switch, the current leakage when the transistor is off, increase exponentially with reduction in the threshold voltage. This results in transistor leakage being a substantial portion of the power consumption and transistors can no longer be scaled to increase performance. This, among other aspects, causes the Dark Silicon effect, which is that only a portion of the die can be used simultaneously to sustain the power budget [3].

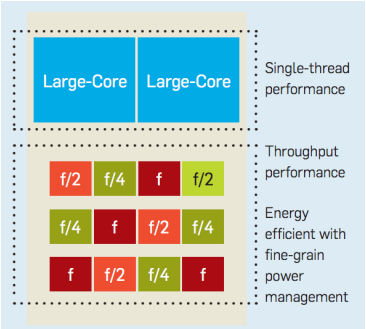

Figure 1.1: A Heterogeneous Many-Core System [2].

Borkar *et al.* [2] predict that heterogeneous processors, consisting of a number of large cores for single-thread performance and many small cores for throughput performance, will better utilize the power budget. Figure 1.1 shows a heterogeneous many-core system. Many small cores operating at a low frequency and voltages near threshold will consume less power then large single-threaded cores, therefore it is important to schedule the tasks to the most suitable processor.

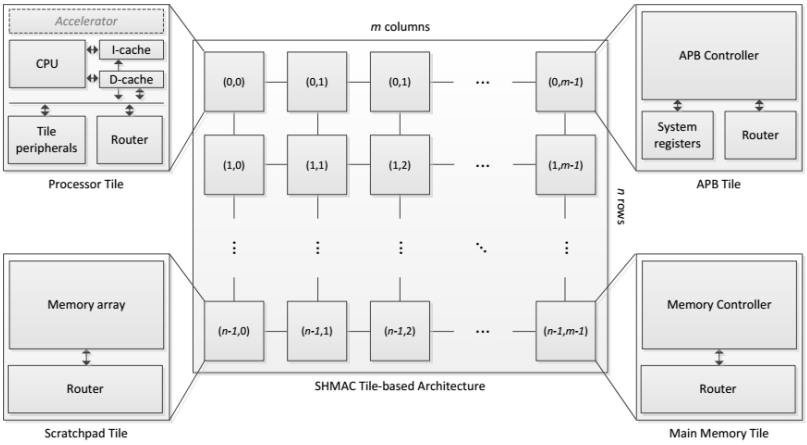

Figure 1.2: High-Level Architecture of ARM-Based Single-ISA Heterogeneous MAny-Core Computer (SHMAC) [4].

To investigate the different issues with heterogeneous systems, the Single-ISA Heterogeneous MAny-core Computer (SHMAC) platform is proposed. This system, as shown in Figure 1.2, is a tile-based architecture that supports the ARM Instruction Set Architecture (ISA) [4]. Each tile can either contain a processor, advanced peripherals bus (APB), scratchpad, main memory or

a dummy. The dummy tile only contains a router and is used to fill the remaining unused tiles.

The use of floating-point (FP) hardware in FPGAs has long been considered infeasible or related to use in expensive devices and platforms [5]. Fixed-point is an alternative to FP and is frequently used in many smaller hardware systems where decimal numbers are needed. However, the complexity of fixed-point operations demands much more preparations and analysis to make sure that the precision and range of the calculations are sustained. Implementing a floating-point unit (FPU) on an FPGA consumes a large amount of resources and is power hungry. However, FP operations are crucial for many scientific applications such as graphics processing, physical simulation, mathematical computations, multimedia application, etc. and for efficient programming, FP is preferred. In applications where FP is not frequently used, emulated FP operations are common. However, in FP intensive applications, emulated FP operation can consume over 90% of the application's total clock cycles, which is unacceptable in most situations [6].

Most FPU design supports the *IEEE Standard 754* for floating-point arithmetic, which among others includes formats, operations, conversions and exceptions. In embedded systems the precision and range of the FP numbers and the operations that are required, are often known. Using a full size FPU supporting the IEEE standard may result in greater power consumption and a bigger part of the FPGA being used. However, using the standard makes it easier to adapt the FPU to different systems.

The SHMAC platform contains many different cores and only the cores handling much arithmetic with decimal numbers needs an FPU. However, the need for range and precision may differ, so implementing the same FPU on every core may result in a lack of utilization and larger power consumption. To overcome this problem a configurable FPU will be implemented. The advantages and disadvantages of this unit will be discussed along with ways to analyse software to find the needed range and precision.

This report will first, in Chapter 2, discuss the background information needed to understand the decisions, implementations and evaluations done. Chapter 3 contains some related research in this area. In Chapter 4 the optimizations of the FPUs and how to test the system are determined. Chapter 5 contains the design of the systems and how well they perform. It also includes an analysis of a selection of benchmarks, and how to find the required bit-width for these. Chapter 6 discusses the results and Chapter 7 concludes this project and suggests future work.

Main Contributions:

- A configurable floating-point unit for Xilinx FPGAs.
- A configurable floating-point unit for ASIC and FPGAs from other vendors.
- Power, area and latency analysis of different floating-point units with different bit-widths.
- Example of software analysis to optimize floating-point units

Chapter 2

Background

This chapter includes the information needed to understand design choices and analysis done later in this thesis. It includes theory about the IEEE 754 Standard for floating-point arithmetic, and the definition of fixed-point numbers. Later, asynchronous design and different floating-point and fixed-point designs are explored. Finally a selection of FPGAs, processing cores and testbenches will be presented.

2.1 IEEE Standard 754 for Floating-Point Arithmetic

This standard specifies formats and methods for floating-point arithmetic in computer systems [7]. The purpose of this standard is to provide a method that will yield the same results whether the processing is done in hardware, software, or a combination of the two. The standard includes the following specifications:

- Formats for binary and decimal floating-point (FP) data, for computation and data interchange.
- Addition, subtraction, multiplication, division, fused multiply add, square root, compare and other operations.
- Conversions between integer and FP formats.
- Conversions between different FP formats.
- Conversions between FP formats and external representations as character sequences.

- FP exceptions and their handling, including data that are not numbers (NaN).

This section will discuss how a binary FP format is represented, short how to convert a decimal number to a binary FP format and how to represent exceptions. The next section will discuss four FP arithmetic operations; adding, subtraction, multiplication and division. Other operations as square root and comparisons will not be discussed because of the limited time aspect on this thesis.

2.1.1 Formats

The standard specifies five basic floating-point (FP) formats. There are three binary formats with encoding lengths 32, 64 and 128 bit, and two decimal formats, with encoding lengths 64 and 128 bit. In this thesis only the binary formats will be considered since binary numbers are natural represented in hardware. The representations of FP data in a format consists of a *sign*, an *exponent*, a *mantissa*, and a *radix* b. An FP number is represented in Equation 2.1 and a figure of a 32 bit floating-point number with eight exponent bit and 23 mantissa is shown in Figure 2.1. Later in this thesis, a 32 bit floating-point number may be referred to as single precision, while a 64 bit floating-point number is called double precision.

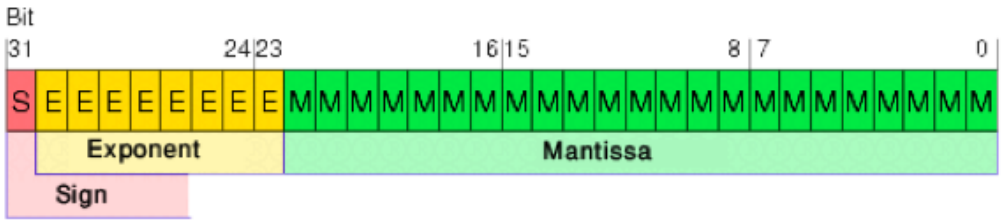

Figure 2.1: Single Precision Floating-Point Number Representation [8].

$$X = (-1)^{sign} * b^{exponent} * mantissa \Leftrightarrow X = (-1)^{S} * b^{E} * M \tag{2.1}$$

The S value can either be 0 or 1, which decides if the number is positive or negative. The b value is the radix and is 2 for binary formats. The E is an integer limited by E_{min} and E_{max} and is represented as e bit. The E_{max} values for three different binary formats are listed in Table 2.1 and can be

Table 2.1: Parameters for Binary Floating-Point Formats [7].

	Binary format *(b=2)*		
parameter	**binary32**	**binary64**	**binary128**
e	8	11	15
E_{max}	+128	+1024	+16384
E_{min}	-127	-1023	-16383
$bias$	127	1023	16383
m	24	53	113

calculated using the following equation: $E_{max} = 2^{e-1}$. The E_{min} values for some binary formats are also listed in Table 2.1 and is calculated as follow: $E_{min} = 1 - E_{max}$. The formula for calculating the exponent value is shown in Equation 2.2 and the *bias* value can be calculated using Equation 2.3. The bias value is used to represent both positive and negative exponent values. As an example, if the exponent is the following sequence of bit; 01111110, then $E = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 - (2^{8-1} - 1) = -1$.

$$E = (\sum_{n=0}^{e-1} b_n \times 2^n) - bias \tag{2.2}$$

$$bias = 2^{e-1} - 1 \tag{2.3}$$

The M value is a string on the form $b_0 2^0 \cdot b_1 2^{-1} b_2 2^{-2} ... b_{m-1} 2^{m-1}$ where b_i is a binary number and m is the number of mantissa bit. The number of mantissa bit for three formats are shown in Table 2.1. The mantissa value can be calculated in to ways, depending on if the number is *normalized* or *denormalized*. A FP number is *denormalized* if the exponent value is equal to the *bias* value, in other words, all exponent bit are zero. The way to calculate the mantissa value is shown in Equation 2.4. This results in the mantissa being a decimal number between one and two if *normalized* and between zero and two if *denormalized*.

$$M = \begin{cases} 1 + \sum_{n=1}^{m} (b_{n-1} \times 2^{-n}) & \text{if normalized} \\ \sum_{n=1}^{m} (b_{n-1} \times 2^{1-n}) & \text{if denormalized} \end{cases} \tag{2.4}$$

2.1.2 Conversion to Binary Format

This thesis will not consider the conversion of decimal numbers to a binary format since this is often taken care of by the compiler [9]. However, a basic understanding can make it easier to understand how floating-point numbers behave. The conversion will be explained by the following example.

4.875 is the decimal number to convert to a 32 bit binary format with 8 exponent bit and 23 mantissa bit. First, the fraction 0.875 is multiplied with two until the remainder is zero. If the result of the multiplication is greater or equal to one the bit on that spot is one, otherwise zero.

$$0.875 \times 2 = 1.750 \Rightarrow b_{-1} = 1,$$
$$0.750 \times 2 = 1.500 \Rightarrow b_{-2} = 1,$$
$$0.500 \times 2 = 1.000 \Rightarrow b_{-3} = 1$$

This results in $(0.875)_{10}$ being represented in binary as $(0.111)_2$. Second, $(4)_{10}$ is converted to binary. It results in the binary representation $(100)_2$, so the entire decimal number is written as $(100.111)_2$. According to the *IEEE 754 Standard* real numbers have to be represented in a $(1.x_1x_2...x_n)_2 \times 2^y$ format. That results in the following conversion: $(100.111)_2 = (1.00111)_2 \times 2^2$. To convert this number to a 32 bit number the exponent has to be biased. According to Table 2.1 the bias for a 32 bit floating-point number is 127. The exponent value can be calculated as $x - 127 = 2 \Rightarrow x = 129 = 2^7 + 2^0$. Since the mantissa is represented as a number between one and two the binary representation of 4.875 is 0 10000001 0011100..... The integer part of the mantissa is "hidden" when the number is *normalized*.

2.1.3 Exceptions

Some of the floating-point bit orders represent special numbers. These are listed below.

+Infinity	All exponent bit are one and others are zero
-Infinity	All bit are one, except mantissa bit
NaN	All exponent bit are one, one/several of mantissa bit are one
+Zero	All bit are 0
-Zero	All bit except sign is zero

In addition does many floating-point units have exceptions for overflow, underflow, invalid operations and divide by zero.

2.2 Arithmetic on Floating-Point Numbers

This section will discuss how to perform floating-point arithmetic "on paper", and give a basic understanding of why the algorithms implemented later in this paper, are implemented the way they are. The arithmetics discussed in

this section are addition, subtraction, multiplication and division [10]. The algorithms are explained by examples.

2.2.1 Addition and Subtraction

The numbers to add and subtract, 100.25 and 0.5, are represented with eight bit exponent and eight bit mantissa. In binary notation these operands are written as:

$$100.25 = 1.0025 \times 10^2 = 0\ 10000101\ 10010001$$
$$0.5000 = 5.0000 \times 10^{-1} = 0\ 01111110\ 00000000$$

The first step is to align the radix point, in other words, make sure that both operands are represented with the same exponent. This can be done by right-shifting the mantissa of the smallest operand. The number of times it has to be shifted is set by the difference between the exponents, which in this example is seven. The resulting binary representation is:

$$0\ 01111110\ 00000000\ \textit{(original value)}$$
$$0\ 01111111\ 10000000\ \textit{(shifted 1 place)}$$
$$0\ 10000101\ 00000010\ \textit{(shifted 7 places)}$$

Notice the "hidden" bit is shifted into the mantissa. It means the new representation of the number is denormalized. Also notice that the exponents of the operands are equal, so the mantissas can easily be added together. The "hidden" bit of the number that is still normalized has to be added.

$$\begin{aligned} &0\ 10000101\ 1.10010001\ (100.25)\\ +&0\ 10000101\ 0.00000010\ (0.5)\\ =&0\ 10000101\ 1.10010011\\ \rightarrow&0\ 10000101\ 10010011\ (100.75) \end{aligned}$$

The next step is to normalize the result. For this example the result is already normalized. However, to normalize a number the "hidden" bit has to be one so the mantissa value is greater than one and smaller than two. This can be done by left or right-shifting the resulting mantissa and subtracting or adding the exponent with the number of shifted places. The final step is to round the results. This is only necessary when the precision of the result exceeds the numbers of mantissa bit. According to the *IEEE 754 Standard*, the number can either be rounded to nearest even, to zero, up or down depending on what the programmer wants [7].

For subtraction the same procedure is followed except instead of adding, the mantissas are subtracted. Note that the smallest number always is subtracted from the biggest, otherwise underflow occurs. An example is $0.5-100.25$. To avoid underflow this have to be rearranged to $-(100.25-0.5)$. The calculation is done below.

$$\begin{aligned} 0\ 10000101\ 1.10010001\ (100.25)\\ -1\ 10000101\ 0.00000010\ (0.5)\\ =1\ 10000101\ 1.10001111\\ \rightarrow 1\ 10000101\ 10010011\ (-99.75) \end{aligned}$$

2.2.2 Multiplication

The bit-width for this example is eight exponent bit and five mantissa bit. The operands are 2.5 and -3.5 and are represented in binary as

$$\begin{aligned} 2.5 &= 0\ 10000000\ 01000\\ -3.5 &= 1\ 10000000\ 11000 \end{aligned}$$

The first step is to multiply the mantissa, note to include the "hidden" bit. This is done as follow:

$$\begin{aligned} &1.01000\\ \times &1.11000\\ = &\ 10.00110000 \end{aligned}$$

Next the exponents are added together and the bias value is subtracted. Since each exponent contains an exponent value and a bias value, the bias value has to be subtracted so it is not added twice. As discussed in the previous section the bias for an eight bit exponent is 127, which results in

$$\begin{aligned} &10000000\\ -&01111111\\ +&10000000\\ =&10000001 \end{aligned}$$

To find the resulting sign, the sign of both operands are XORed and in this example the resulting sign bit is 1. The result is then 1 10000001 10.00110000. The mantissa of this number is not normalized. To normalize the number the mantissa has to be right-shifted one place and the exponent must be added with one. This finally results in 1 10000010 1.000110000 $\rightarrow$ 1 10000010 00011(-8.75).

2.2.3 Division

The algorithm for division is quite similar to the one for multiplication. The only difference is the mantissas are divided instead of multiplied and the exponent is subtracted instead of added. This example uses eight exponent bit and five mantissa bit. The operands are 10.0 and 2.5 and are represented in binary as:

$$10.0 = 0\ 10000010\ 01000$$
$$2.5 = 0\ 10000000\ 01000$$

The first step is to subtract the exponents. The bias has to be added so the bias value is not subtracted twice.

$$\begin{aligned} &10000010 \\ -&10000000 \\ +&01111111 \\ =&10000001 \end{aligned}$$

The next step is to divide the mantissas.

$$\begin{aligned} &1.01000 \\ /&1.01000 \\ =&1.00000 \end{aligned}$$

This mantissa is already normalized so the final result is 0 10000001 00000.

2.3 Fixed-Point

Fixed-point data do not have a clear specification as floating-point data. However, it is defined as either fractional data values or data values with an integer part and a fractional part [11]. Fixed-point data can typically be used when the dynamic range and precision is less important then the size and speed of the system.

The binary interchange fixed-point formats are defined in Figure 2.2. Fixed-point values are represented using a two's complement number that is weighted by a fixed power of two [12]. The bit position is labelled with an index i. The value of a fixed-point number is given by Equation 2.5.

$$v = (-1)^{b_{w-1}} 2^{w-1-w_f} + \sum_{i=0}^{w-2} 2^{i-w_f} b_i \tag{2.5}$$

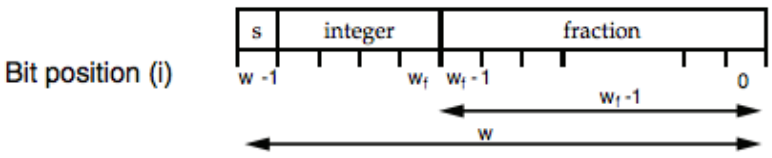

Figure 2.2: Bit Order with Fixed-Point Representation [12].

2.4 Asynchronous Design

Asynchronous design is independent of the clock signal, which can potentially lead to performance advantages[13]. However, asynchronous design requires extra logic to detect the completion of a step, and this may lead to the advantage levelling out. Various from synchronous designs that are either on if the clock is on or visa versa, asynchronous design only consume power when active. This result, in most cases, that asynchronous design is less power hungry then synchronous. The latency in an asynchronous design depends on the longest path through the design. This may vary in how it is designed and which platform it is implemented on. This makes it harder to predict the latency for asynchronous design and more difficult to adapt the design to different systems.

2.5 Floating-Point and Fixed-Point Unit Design

This section will explore some of the available floating-point (FP) and fixed-point implementations. Since the SHMAC platform is using a Xilinx FPGA, FP operators from Xilinx are explored. The other floating-point implementations discussed are open source.

2.5.1 LogiCORE IP Floating-Point Operator

The Xilinx floating-point core provides a range of floating-point arithmetic with a high level of user specification [12]. The interface is shown in Figure 2.3. *A* and *B* are the operands and *OPERATION* specifies the operation when the core is configured for multiple. *OPERATION_ND* is set high to indicate that the operands and the operation are valid, *OPERATION_RFD* is set by the core to indicate that it is ready for new operands. *SCLR* is a synchronous reset, *CE* is clock enable and *CLK* is the clock. *RESULT* is the result of the operation, *UNDERFLOW* is set high when underflow occurs

and *OVERFLOW* is set high when overflow occurs. *INVALID_OP* is set high by core when operands cause an invalid operation, *DIVIDE_BY_ZERO* is set high if a division by zero was performed and *RDY* is set high by the core to indicate that the *RESULT* is valid. Many of the inputs and outputs can be removed by the designer.

The IP supports several fraction and exponent bit-width. The minimum mantissa bit-width is 4 bit and the maximum is 64. The minimum exponent bit-width is 4 bit and the maximum is 16. The minimum exponent width is also limited by Equation 2.6. As an example, if the fraction width is 23, the minimum exponent bit-width is five. This is also controlled when implementing the IP. It is possible to use an asynchronous version of the IP, which does not need any clock input.

$$Minimum\ Exponent\ Width\ = \lceil \log_2(Fraction\ Width\ + 3) \rceil + 1 \quad (2.6)$$

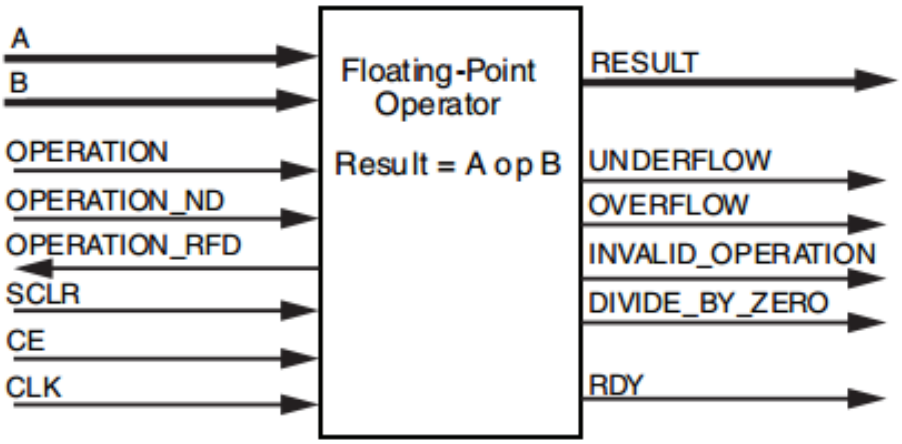

Figure 2.3: Block Diagram of Generic Floating-Point Binary Operator Core from Xilinx[12].

2.5.2 OpenCores Single Precision Floating-Point Unit

The floating-point unit (FPU), provided by OpenCores, is a 32-bit open source processing unit [8]. It fully complies with the *IEEE 754 Standard* for single precision floating-point arithmetic and includes, among others, addition, subtraction, multiplication and division. Unlike the Xilinx Core presented in subsection 2.5.1, this is an open source design. As a result, it is possible to explore the algorithms and functionality of the design. However,

the design is quite complicated and it does not support generics to set the functionality in the FPU. This result in doing changes, e.g. to the bit-width, big parts of the design have to be rewritten.

2.5.3 OpenCores Double Precision Floating-Point Unit

The double precision floating-point core published by OpenCores are designed to meet the *IEEE 754 Standard* for double precision floating point arithmetic [14]. This FPU is the same unit implemented on the Amber core on the SHMAC platform. The Amber core will be discussed later in this chapter. As shown in Figure 2.4, it includes addition, subtraction, multiplication and division, a rounding unit and an exception handler. Like the single precision FPU, it does not support variable bit-widths.

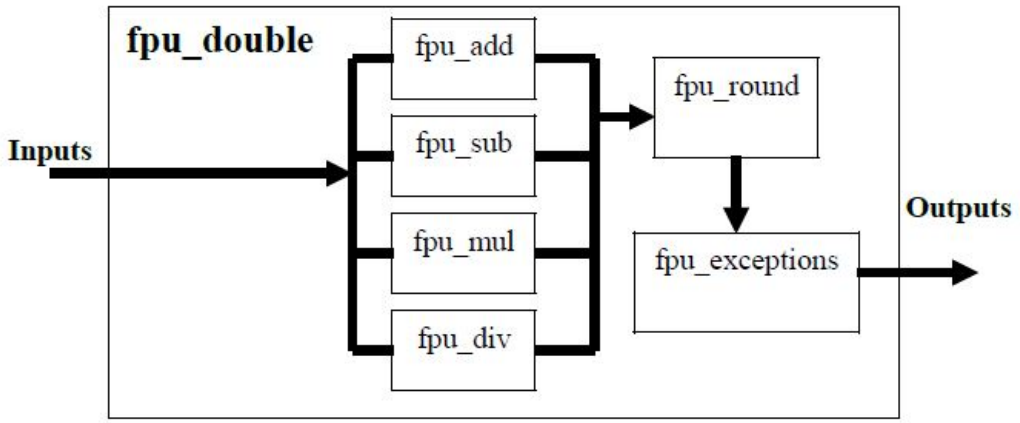

Figure 2.4: Hierarchy of Double Precision Floating-Point Core [14].

2.5.4 Floating-Point Library

This floating-point library complies with the *IEEE 754 Standard* [15]. In addition can the bit-width, rounding style and exception handling be configured and denormalized numbers can be excluded. It is also possible to change the number of guard bit, which is used in arithmetic operations to maintain precision. These constants are shown in Listing 2.1. The library introduces the types float, float32, float64 and float124, which allows the designer do declare signals and variables with different bit-widths.

Listing 2.1: Constants for float type in the IEEE Floating-Point Library [15]

```
package float_pkg is
  constant float_exponent_width : NATURAL    := 11;
  constant float_fraction_width : NATURAL    := 52;
  constant float_denormalize    : BOOLEAN    := false;
  constant float_check_error    : BOOLEAN    := false;
  constant float_guard_bits     : NATURAL    := 0;
  constant no_warning : BOOLEAN := (false);
```

2.5.5 Fixed-Point Library

The fixed-point library is defined as set of types and functions to include in the design [16]. It introduces the types $sfixed$ and $ufixed$ for signed and unsigned fixed-point numbers, with a user specified integer and fraction length. The library contains, among others, operations like addition, subtraction, multiplication and division. Unlike floating-point numbers the results of an operation does not have the same bit-width as the operands. The bit-widths for the results are listed in Table 3.2.

Table 2.2: Bit-Width of Result for Different Operations [16].

Operation	**Result Bit-Width**
A+B	Max(A'int, B'int)+1 downto Min(A'frac, B'frac)
A-B	Max(A'int, B'int)+1 downto Min(A'frac, B'frac)
A*B	A'int+B'int+1 downto A'frac+B'frac
Signed /	A'int-B'frac+1 downto A'frac-B'int
Unsigned /	A'int-B'frac downto A'frac-B'int-1

2.5.6 SoftFloat

The software floating-point library is a part of the gcc library and is regularly used in systems that do not have a hardware floating-point unit [9]. In Table 2.3 the latency and area usage of SoftFloat for single precision floating-point arithmetic on the SHMAC platform are listed. This table will be used for analysis later in this thesis.

2.6 Processing Cores

This section discusses two different processing cores. By having a processing core, compiled high level applications and assembly code can be executed

Table 2.3: Latency and Area Usage for Single Precision Floating-Point in SoftFloat on the SHMAC Platform [17].

	Add/Subtract	Multiply	Divide
Latency without instruction and data cache (Cycles)	1018/1034	3324	2494
Latency with instruction and data cache (Cycles)	59	193	145
Area usage (LUTs)	0	0	0

on the system. It also provides support for accelerators, e.g. a hardware floating-point unit. The first processing core is the Amber 2, which is the same core implemented on the SHMAC platform. Unfortunately this core does not have hardware floating-point support in its compiler. As a result, a second processing core, OpenRISC, which do have hardware floating-point support, is explored.

2.6.1 Amber 2 Core

The Amber processor core is an ARM-compatible 32-bit RISC processor. It is fully compatible with the ARM v2a instruction set architecture (ISA) and provides a complete embedded system with a number of peripherals like UARTs, timers and a double data rate (DDR3) memory controller [18, 19]. As shown in Figure 2.5, the system contains a Wishbone bus. This bus is an open source hardware computer bus, intended to let the parts of an integrated circuit communicate with each other [20]. It is a logic bus, which means that it does not specify the electrical characteristics or the bus topology. It is synchronous and is defined to have 8, 16, 32 or 64-bit buses. The Amber core has a 32-bit Wishbone system bus, a 5-stage pipeline and separate instruction and data caches. It has multiple and multiply-accumulate operations with 32-bit inputs and 32-bit output in 32 clock cycles, using the Booth algorithm.

The Amber 2 Core contains a co-processor, which includes a floating-point unit (FPU). Currently the 64 bit FPU by OpenCores is implemented, but not supported in the compiler. It requires a total of eight clock cycles for loading data and four for storing the data to the co-processor [21].

2.6.2 OpenRISC

The OpenRISC 1200 Core is a 32-bit scalar RISC with Harvard microarchitecture [23]. As shown in Figure 2.6, contains the core, amongst other modules, a debug unit, interrupt controller, direct-mapped instruction and

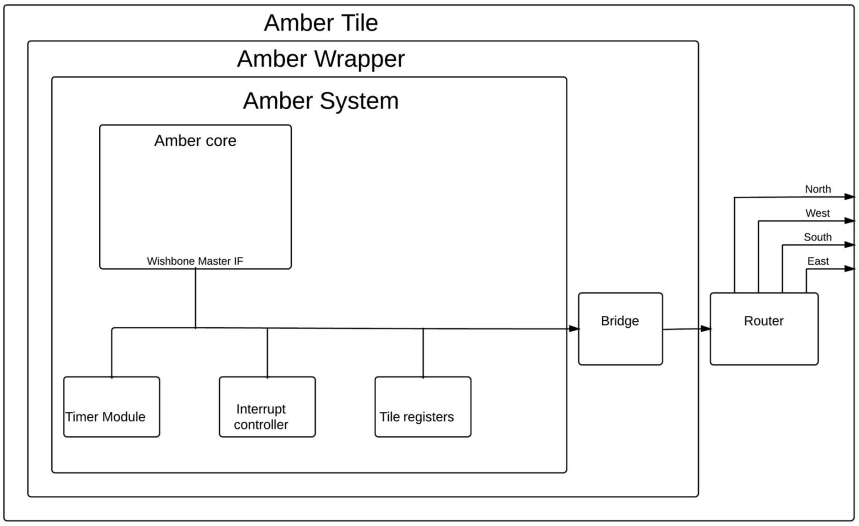

Figure 2.5: Amber Tile on SHMAC Platform[22].

data cache. Two Wishbone interfaces connect the core to external peripherals and external memory systems. The CPU has a 5-stage pipeline and handles the ORBIS32 instruction set architecture (ISA). It contains everything needed in a CPU including a floating-point unit.

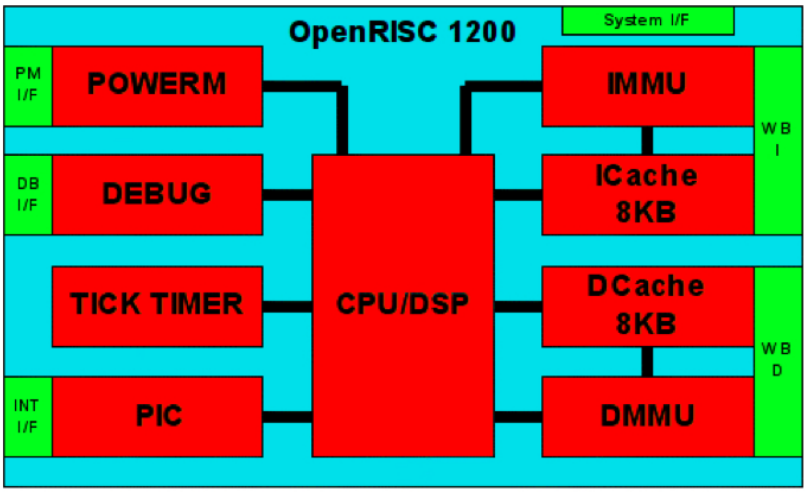

Figure 2.6: OpenRISC 1200 Core Architecture[23].

A reference design for the OpenRISC is the ORPSoC (OpenRISC Reference Platform System on Chip)[24]. It is a development platform targeted

at specific hardware. The project is organized in such ways that register transfer level (RTL) and software can be added or changed by the user. It also contains a GNU compiler so software can be compiled and run on the desired hardware. The design can be simulated using standard event-driven simulators such as Icarus Verilog and Mentor Graphics' Modelsim or it involves creating a cycle accurate model in C or SystemC using the Verialtor tool [25].

2.7 FPGAs and Tools

In this section a list of Xilinx FPGAs and available tools are presented. Since the SHMAC platform is implemented on a Xilinx FPGA, it is desirable to test the system with the same type of FPGA and tools.

2.7.1 Xilinx Virtex-5 XC5VLX330 FPGA

The Virtex-5 LX platform from Xilinx is a high-performance general logic applications FPGA [26]. It contains of 51,840 Virtex-5 slices and 3,420 Kb maximum distributed RAM. Each Virtex-5 slice contains four LUTs and four flip-flops. It also contains 192 DSP48E Slices, which allow the designers to implement multiple slower operations using time-multiplexing methods. They provide, among other, better flexibility and utilization and reduced power consumption. It contains 288 36Kb block RAM blocks, for a total of 10,368Kb and has a total of 33 I/O banks with a maximum of 1,200 user I/Os.

2.7.2 Xilinx Spartan -6 LX16

This evaluation kit from Avnet contains , among other components, a Xilinx Spartan-6 XC6SLX16-2CSG324C FPGA, a Cypress PSoC 3 CY8C3866AXI-40 Programmable System-On-Chip, 32 Mb $\times$ 16 Micron LPDDR Mobile SDRAM and a 128 Mb Numonyx Multi-I/O SPI Flash [27]. A block diagram of the board is shown in Figure 2.7. The FPGA contains 2,278 slices and 14,579 logic cells. Each slice contains four LUTs and eight flip-flops. To access and utilize the various features on the board the software AvProg is used. The software also has the ability to measure the power consumption in real-time.

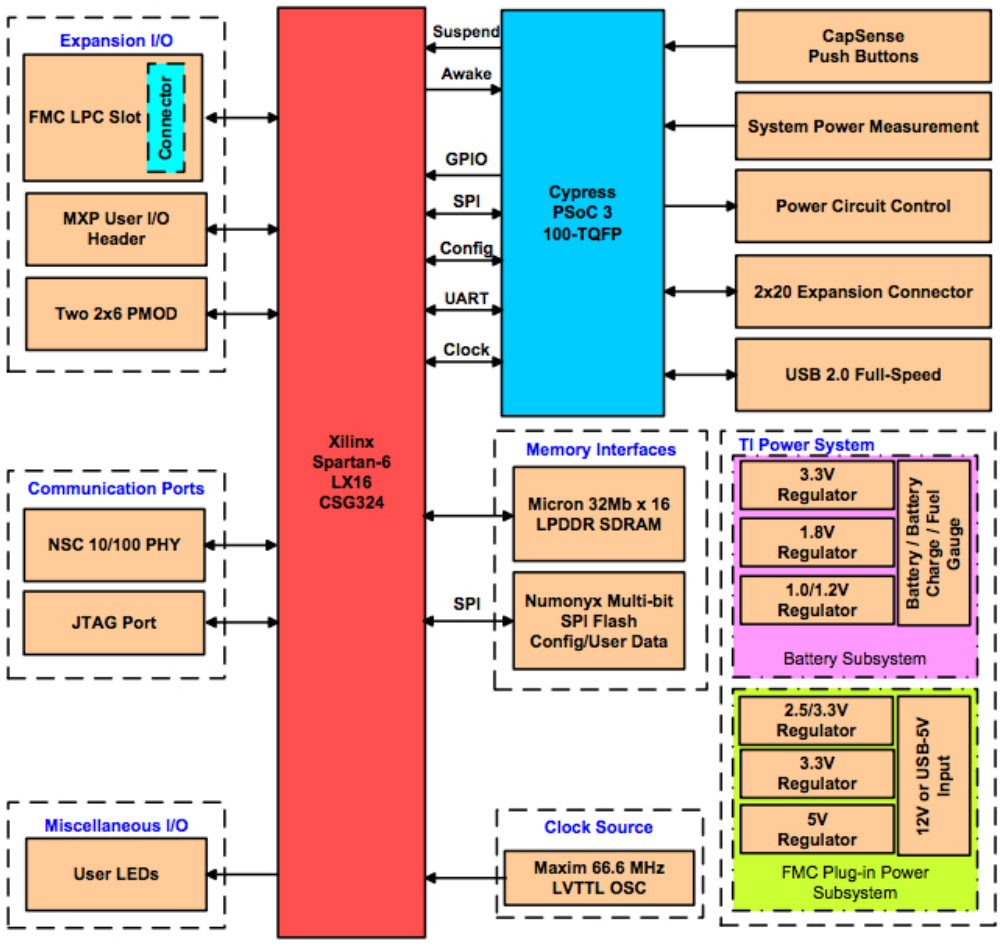

Figure 2.7: Spartan-6 LX16 Evaluation Board Block Diagram [27].

2.7.3 Xilinx Zynq^{TM}-7000

The ZedBoard from Digilent and Avnet is based on the Xilinx Zynq All Programmable SoC (AP SoC) and combines a dual Corex-A9 Processing System with a Artix-7 FPGA [28]. The FPGA contains 53,200 LUTs, 560 KB extensible block RAM and 220 programmable DSP slices. The ZedBoard also contains 512 MB DDR3 memory and USB-JTAG for easy programming and debugging. A block diagram of the complete system is shown in Figure 2.8.

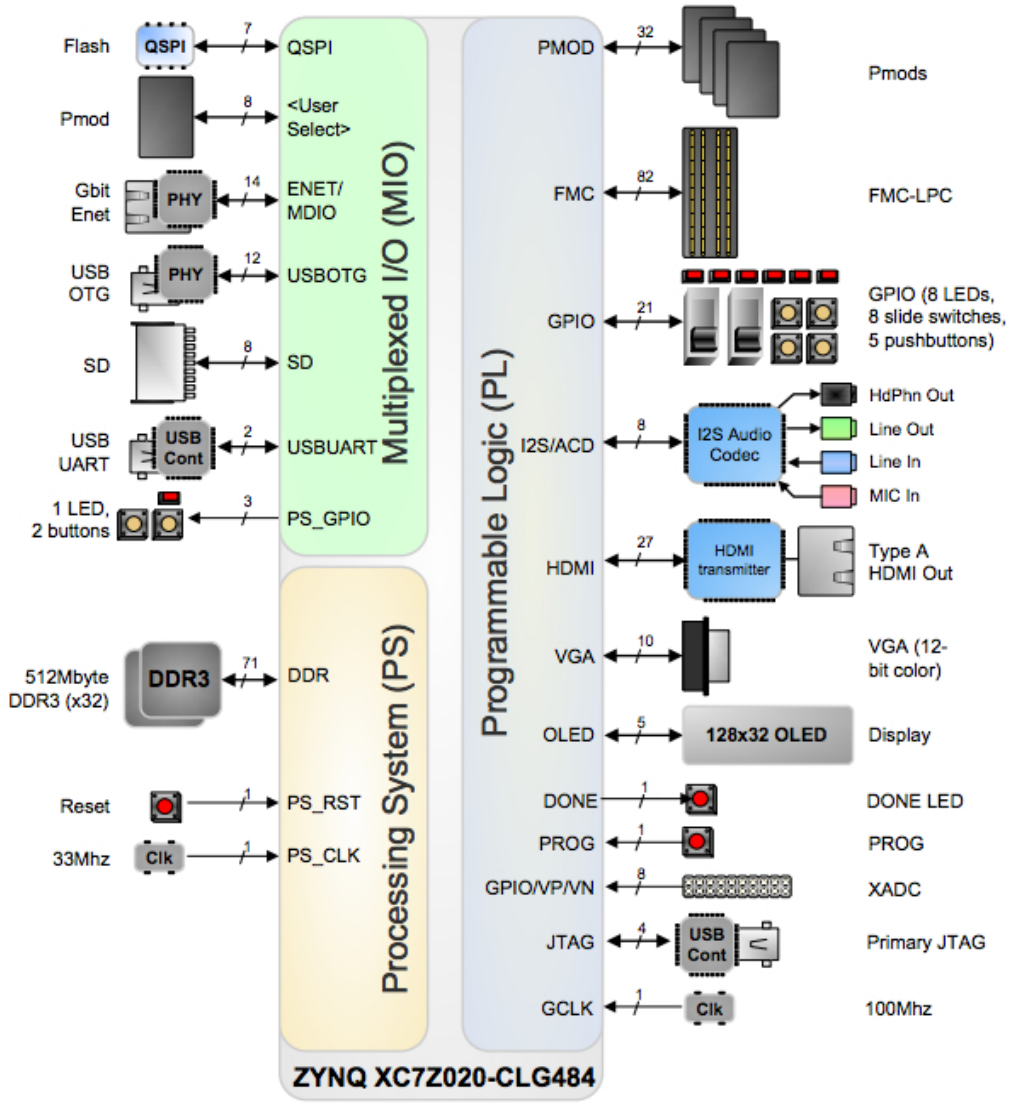

Figure 2.8: ZedBoard Block Diagram [27].

2.7.4 Tools

Xilinx provides a number of tools to analyze the behavior and measure the size, speed and power of the design. Below is a list containing these tools.

- ISE Design Suite 14.7 [29] is a development environment for all Xilinx devices. The tool allows for synthesizing, implementing and pro-

gramming on the FPGA. It also makes it easier to use other analysis programs on the design.

- XST [30] is the Xilinx Synthesis Technology, which synthesize hardware description language for Xilinx devices. It also optimizes the design for the FPGA.

- ISim [31] is a hardware description language simulator that performs behavioral and timing simulations. It supports power analysis and optimization using SAIF, which contains the toggle counts on the signals of the design.

- XPower [32] analyses the power consumption on the designed data. To get a reliable report with power consumption for each component in the design, a "place & route" file, a physical constraint file and a SAIF file have to be presented.

2.8 Testbenches

The Standard Performance Evaluation Corporation (SPEC) was formed to establish, maintain and endorse a standardized set of benchmarks to test the performance of a system [33]. The most resent benchmark is the SPEC CPU2006, which includes CINT2006 for measuring and comparing system with integer computation and CFP2006 for measuring and comparing floating point performance.

The retired benchmark CFP2000 contains a total of 14 floating point components. Four of these are written in C, while the other ones are written in Fortran [34]. Information about the testbenches written in C is listed below.

- 177.mesa is a free OpenGL work-alike library written by Brian E. Paul. The input data is a two dimensional scalar field which is mapped to height creating a three dimensional object with explicit vertex normals. The contour lines are mapped onto the object as a one dimensional texture.

- 179.art (adaptive resonance theory) is used to recognize objects in a thermal image. The input consists of a thermal image of a helicopter or an airplane and a scan file, which contains other thermal views of the helicopter and airplane. The output data consists a report of a match between the learned image and the windowed field of view.

- 183.equake simulates the propagation of elastic waves in large, highly heterogeneous valleys. The goal is to recover the time history of the ground motion everywhere within the valley. The input data contains the grid topology and the seismic event characteristics and it outputs a case summary with seismic source data and a characteristic of the motion at both the hypocenter and epicenter.

- 188.ammp runs molecular dynamics on a protein-inhibitor complex which is embedded in water. The benchmark is derived from publishing work on understanding drug resistance in HIV of Weber and Harrison in 1999. The input is the initial coordinates and velocities of the atoms. The output is the energy of the final configuration of atoms.

In these testbenches the dominating floating-point operations are addition, subtraction, multiplication and division. The number of times these are used in each testbench are listed in Table 2.4 [35].

Table 2.4: Number of Floating-Point Operations in Different Testbenches [35].

Name	Description	+	-	×	/
177.mesa	Graphics Library	347	102	586	27
179.art	Image Recognition/ Neutral Networks	253	14	247	12
183.equake	Seismic Wave Propagation Simulation	127	58	236	18
188.ammp	Computational Chemistry	479	330	930	42

Chapter 3

Related Work

A lot of research has investigated optimization of floating-point units for area, delay and power consumption in hardware. Some of the articles suggest changing the design of the units. This is not directly relevant to this thesis, however it will better explain the complete research done in the field of floating-point units. Only the directly relevant research articles will be explained in detail, while the others will be described shortly.

Chong *et al.* [36] propose a flexible multimode embedded floating-point unit (FPU) for FPGAs to better utilize the die. They suggest duplicating the data path for single precision arithmetic, and linking duplicated functional blocks together to accommodate double precision. This leads to a greater area utilization and delay improvement because of parallelizing. This approach complies with the *IEEE 754 Standard* and is easy to test and validate. However, with this approach it is required to do changes to the hardware design and this may cause a longer time-to-market.

Another work by Chong *et al.* [6] propose custom FPUs for embedded systems to utilize area and performance. A rapid design space exploration was explored to balance between hardware-implemented and the software emulated instructions. Data path merging was also proposed to utilize the area. It means that the same components (for instance adders and multipliers) can be used with different word lengths. The article shows that adding more floating-point hardware does not necessarily result in a lower runtime, and the delay associated with the additional hardware being greater than the cycle count reduction. The advantages with these approaches are that it complies with the *IEEE 754 Standard*. This makes it easy to validate and test. The design space exploration can be used as a front-end to explore the best solution for the system. The downside of the approach is that it requires complicated changes to the floating-point unit design. A bit-alignment algorithm is necessary to design a well working data path merging algorithm and

this may cause a very complicated design.

Liang *et al.* [37] have outlined a floating-point unit generation approach, which allows for the creation of a vast collection of floating point units with differing throughput, latency and area characteristics. Given the constraints, the algorithm chooses the proper implementation and architecture to create the compliant floating point unit.

Galal *et al.* [38] present a method for creating a trade-off curve that can be used to estimate the maximum floating-point performance given a set of area and power constraints.

3.1 Bit-Width Optimisation for Fixed-and Floating-Point

This section presents a method to optimize the bit-width of both fixed-point and floating-point designs [39]. If U_i represent a floating-point number, $(-1)^S \times M \times 2^E$, where S is the sign bit, M is the mantissa and E is the exponent. The precision in a floating-point number depends on the mantissa bit-width (m) and the range depends on the exponent bit-width (e). The error for both fixed-point and floating point is given in Equation 3.1. The l represents the fraction length for fixed-point numbers. The calculated error for floating-point numbers is represented in Equation 3.2. For further analysis the truncation rounding model is chosen. Round-to-nearest will give a better error bound then truncation, but require additional hardware.

$$\Delta U_i = \begin{cases} Err_{flt}(m) & \text{if Type = Float} \\ Err_{fix}(l) & \text{if Type = Fixed} \end{cases} \tag{3.1}$$

$$Err_{flt}(m) = \begin{cases} 2^{-m} \times 2^E & \text{if round-to-nearest} \\ 2^{-(m-1)} \times 2^E & \text{if truncation} \end{cases} \tag{3.2}$$

The equation for calculating the mantissa bit-width m is represented in Equation 3.3. E_{U_i} can be found by solving $E_{U_i} = \lceil \log_2(|U_i|) \rceil$.

$$m \geq E_{U_i} - \lceil \log_2(|\ \Delta U_i\ |) \rceil + 1 \tag{3.3}$$

The dynamic range of the operation is given by $|max(U_i)/min(U_i)|$, so the exponent bit-width of U_i can be calculated with Equation 3.4. To make it easier to understand this equation, a table containing the range with different exponent bit-width is presented in Table 3.1. If the floating-point unit is not supporting denormalized floating-point numbers the minimum exponent value will increase with one.

$$e \geq \lceil \log_2(|max(E_{U_i})/min(E_{U_i})|) \rceil \tag{3.4}$$

Table 3.1: Calculated Range for Floating-Point Numbers with Different Exponent Bit-Widths(e)

e	$bias$	E_{max}	E_{min}	$2^{E_{max}}$	$2^{E_{min}}$
2	1	2	-1	2	0.5
3	3	4	-3	16	0.125
4	7	8	-7	256	1/128
5	15	16	-15	65536	1/32768
6	31	32	-31	$4.29E9$	$4.66E-10$
7	63	64	-63	$1.84E19$	$1.08E-19$
8	127	128	-127	$3.40E38$	$5.88E-39$
9	255	256	-255	$1.16E77$	$1.73E-77$
10	511	512	-511	$1.34E154$	$1.49E-154$
11	1023	1024	-1023	$1.80E308$	$1.11E-308$
12	2047	2048	-2047	$1.62E616$	$3.09E-617$

In the case of fixed-point, the range depends on the integer bit-width, while the precision depends on the fraction bit-width. Consider the case where U_i represents a fixed-point number, k is the number of integer bits and l is the number of fraction bits. The integer bit-width is calculated according to Equation 3.5 and the first twelve values of k is found in Table 3.2.

$$k \geq \lceil \log_2(|max(U_i)/min(U_i)|) \rceil \tag{3.5}$$

Table 3.2: Calculated Range for Fixed-Point Numbers with Different Integer Bit-Width(k)

k	1	2	3	4	5	6	7	8	9	10	11	12
Max integer value	1	3	7	15	31	63	127	255	511	1023	2047	4095

The precision of a fixed-point number depends on the fraction bit-width and the error depending on the fraction bit-width is calculated with Equation 3.6. For further calculation the error using the truncation rounding model is used.

$$Err_{fix}(l) = \begin{cases} 2^{-l} & \text{if round-to-nearest} \\ 2^{-(l-1)} & \text{if truncation} \end{cases} \tag{3.6}$$

From Equation 3.6 and $\mid \Delta U_i \mid$ expressed in Equation 3.1, the bit-width of the fraction part is expressed with Equation 3.7.

$$l \geq \lceil \log_2(\mid \Delta U_i \mid) \rceil + 1 \tag{3.7}$$

To better understand this optimization process for floating-point and fixed-point numbers, an example is given below.

$$Max(U_i) = 200, \Delta U_i = 0.00005, Min(U_i) = 0.0001$$
$$e \geq \lceil \log_2(|max(E_{U_i})/min(E_{U_i})|) \rceil \geq 5$$
$$m \geq E_{U_i} - \lceil \log_2(\mid \Delta U_i \mid) \rceil + 1 \geq 8 - \lceil \log_2(\mid 0.000005 \mid) \rceil + 1 \geq 8 + 14 + 1 \geq 23$$
$$Total\ bit = 29$$

$$k \geq \lceil \log_2(|max(U_i)/min(U_i)|) \rceil \geq 8$$
$$l \geq \lceil \log_2(\mid \Delta U_i \mid) \rceil + 1 \geq 15$$
$$Total\ bit = 23$$

In this example the total bit-width for floating-point numbers are greater then fixed-point. However floating-point numbers has a bigger range for the same amount of bit.

3.2 Minimizing Floating-Point Power Dissipation via Bit-Width Reduction

Tong *et al.* [40] proposes four different ways to reduce power consumption. By reducing the mantissa and exponent bit-widths the precision and range is lowered, but the switching activity and the necessary normalizing shifting will reduce. By changing the implied radix, e.g. from 2 to 4, a greater dynamic range is provided, but this leads to a lower density. This may result in the normalization shifts being reduced. Finally the article suggests a simplification of rounding modes. Full support of rounding modes is very expensive and some programs may achieve acceptable accuracy with a simple rounding algorithm. This article only explores the reduced power consumption differing the exponent and mantissa bit-width.

The article uses four workloads to proof it's results. Sphinx III is the first workload. It is a CMU's (Communication Management Unit) speech recognition program based on fully continuous hidden Markov models. The accuracy is estimated by dividing the number of words recognized correctly

over the total numbers of words in the input set. Second is ALVINN. This workload takes input from a video camera and a laser range finder to guide a vehicle on the road. The accuracy is measured as a number of correct travel directions. Third is the PCASYS, which is a pattern-level finger print classification program developed at NIST (National Institute of Standards and Technology). The accuracy is measured as percentage error in putting the image in the wrong class. The final workload is Bench22. This is a benchmark which wraps a random image and measures the accuracy by comparing the wrapped image with the original.

In Figure 3.1 the accuracy for the different workloads varying the exponent and mantissa bit-width are showed. For this set of workload the accuracy does not drop before the exponent bit-width is lower then seven and mantissa bit-width is lower then 11.

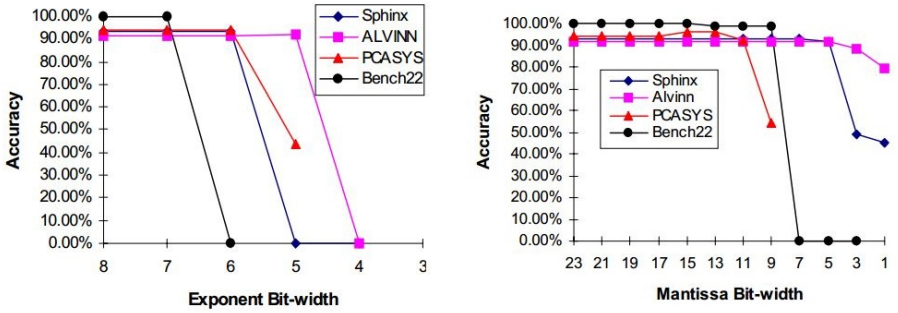

Figure 3.1: Accuracy Compared with Various Exponent and Mantissa Bit-Widths [40].

According to Figure 3.2 the latency and energy consumption per operation drops linear decreasing the operand bit-width.

This paper proposes four important ways to reduce the power consumption in floating-point units and also concludes that the power consumption in the unit highly depends on the operand bit-width. However, the area is not considered in this article.

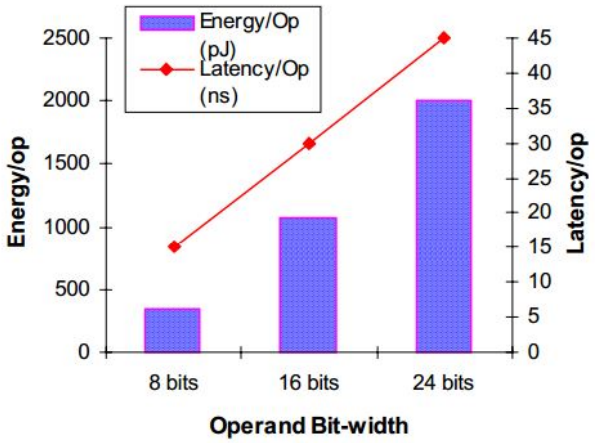

Figure 3.2: Energy and Latency per Operation for Different Operand Bit-Widths [40].

Chapter 4

Design Space Exploration

This chapter will discuss the optimization options for floating-point arithmetic and choose which to implement and test. Then the necessary tools and hardware, to get reliable results, will be analyzed.

4.1 Floating-Point Standard

The suggested optimization options for floating-point arithmetic in the article about minimizing floating-point power dissipation in Subsection 3.2 are reducing the bit-width, changing the radix and simplifying the rounding modes. All of these options violates the IEEE standard discussed in Section 2.1, however great energy reductions is demonstrated by lowering the bit-width. This thesis will expand the results by also analyzing the area when varying the bit-width. The consequence of violating the IEEE standard is lack of portability, however this thesis will consider configurable configurable floating-point units that are tailored for a specific set of tasks. Therefore, a violation of the IEEE standard will have a minor influence on the portability.

A set of exceptions, defined in the IEEE Standard 754, is specified in Subsection 2.1.3. Implementing these will have an influence on the area and most likely the power consumption. The exceptions representing zero and infinity is needed to have a functional FPU, however the other can be avoided by analyzing the applications before executing them. Only zero and infinity will be be implemented, if optional.

The IEEE Standard 754 states a set of arithmetic operations an FPU should support. According to the analysis of the testbenches in Section 2.8, addition, subtraction, multiplication and division are the most frequently used arithmetic floating-point operations. It is reasonable to believe that this may apply for many other applications too. As a result will only these

arithmetic floating-point operations be implemented and tested.

The IEEE standard specifies conversion methods, e.g. the conversion between integers and floating-point formats. These methods will not be included in this thesis because of the limited time aspect on this thesis.

Included in the standard is also a set of rounding rules. To make design less complex, the truncation rounding will be used, if optional.

4.2 Floating-Point Implementation

In Section 2.5 a selection of FPU implementations is discussed. The LogiCORE IP Floating-Point Operator discussed in Subsection 2.5.1 is optimized for Xilinx FPGAs and since the SHMAC platform currently is implemented on a Xilinx FPGA, will this design be implemented and tested. The bit-width of the mantissa and exponent is user editable, and will produce results to support the thesis. The FPU supports all exceptions specified in the IEEE Standard 754, however only the necessary exceptions will be implemented while testing. Both the synchronous and asynchronous designs will be tested.

The single precision FPU by OpenCores discussed in Subsection 2.5.2 and the double precision FPU in Subsection 2.5.3 complies with the *IEEE 754 Standard*, and do not support variable bit-widths. These units will be implemented, and hopefully strengthen the assumption that big gains can be accomplished by varying the bit-widths.

If the SHMAC platform is going to be adapted for ASIC or an FPGA from another vendor, it is important to have alternatives to the Xilinx FPU. Therefore the floating-point library discussed in Subsection 2.5.4 will be implemented and tested for different bit-widths. The library will be set up according to Listing 2.1, only varying the exponent and mantissa bit-width. The library supports guard bit and denormalized number. Guard bit can be used to maintain precision in arithmetic, however when using this library, guard bit and denormalized numbers will not be implemented to achieve lower area and power consumption. The consequence of not supporting denormalized numbers is that the minimum exponent value is one less then usual.

To compare the FPU designs with fixed-point the fixed-point library discussed in Subsection 2.5.5 will be implemented. The unit will be implemented with 32 bit, 16 bit integer and 16 bit fraction, and the signed fixed-point format will be used. The software floating-point library in Subsection 2.5.6 will also be used to compare with the hardware FPU designs.

A customizable floating-point unit will be designed and tested. It is

designed to be easy to expand and customize for your system regardless of platform. The unit will only support addition, subtraction and multiplication with normalized numbers. The divider will not be implemented because of a limited time aspect on the thesis.

4.3 Processing Core

In Section 2.6 both the Amber 2 core and the OpenRISC are discussed. The Amber 2 is the same core as implemented in the SHMAC platform. However, this core does not have a compiler that supports hardware floating-point operations, while the OpenRISC does.

To measure the speed-up, testbenches can be compiled and executed on the processing cores with different FPUs and bit-widths. Another option is to calculate the run time for each operation, including the read and write time for the architecture. The number of operations performed in each testbench is estimated and the performance gain can be calculated. In this thesis the second option is chosen to have more time focusing on design and optimizing of FPUs, instead of implementing a processing core.

4.4 FPGA

In Section 2.7 three different FPGAs are discussed. The Xilinx Virtex-5 FPGA in Subsection 2.7.1 is the same FPGA used on the SHMAC platform. Implementing directly onto the SHMAC platform would be ideal, but in addition to not supporting hardware FPU, many members of the project are using it and the availability is low.

The two evaluation boards discussed in Subsection 2.7.2 and 2.7.3, were available. The Spartan-6 LX evaluation board is able to measure the power consumption in real-time, however the FPGA does not contain enough LUTs to implement any of the discussed processing cores. The Xilinx Zynq-7000 SoC contains an Artix-7 FPGA. This FPGA has about the same properties as the Virtex-5 and enough LUTs and memory to handle both cores. However, since no processing core is implemented, there is a lack of motivation to perform analysis using a physical FPGA. As a result, will all analysis be done in software, using the tools discussed in Subsection 2.7.4, targeting the Virtex-5 FPGA.

4.5 Testbenches

The benchmark discussed in Section 2.8 includes a good selection of testbenches. The floating-point data available will be analyzed and the equations in Section 3.1 will be used to find the optimal bit-width. In addition, will the option of emulating parts of the resulting FPU in software be explored.

The data analyzed will be taken from the input and output files for each testbench. However, arithmetic operations executed in the applications may use numbers with higher range and accuracy than what is presented on input and output. To compensate for this, an additional bit will be added to the exponent and mantissa when calculating the bit-width.

Chapter 5

Implementation and Results

This chapter contains two sections. The first section explains how testing is performed to make sure that all units are functional and tested with the same parameters and variable. This will ensure that applicable results are generated. The second section describes individually how each floating-point unit (FPU) is designed and how well they perform on latency, area and power consumption.

5.1 Test Plan

5.1.1 Functionality

To test if an FPU has the correct behavior, a Matlab function, in Appendix A.1, is written. It generates a user specified number of random operands with user specified exponent and mantissa bit-widths. The operands are saved to file and executed in a simulator with the Xilinx FPU to generate the correct results. An example of this testbench is listed in Appendix B.2. This approach assumes that the Xilinx FPU has the correct behavior. Then the same testbench is run with a different FPU and another file containing it's results are generated. Finally the two generated data files, containing the results, are compared in a Matlab function, Appendix A.2. The functions described above can also be used for fixed-point. To make sure the floating-point operations are performed correctly a third Matlab function is created, Appendix A.3. This function calculates the decimal value of a floating-point number, so the user can check the operands and the result of each calculation.

5.1.2 Performance

The synthesizing tool used is the XST by Xilinx and the optimization goal is set for area. The input and output pins are placed randomly on the FPGA. To generate a SAIF file, which describes the switching activity of the design, all designs are simulated with 100 arithmetic operations with 50 μs to calculate these and a $100MHz$ clock is used. For all FPUs the same testbench, only varying the bit-widths, is used. Each arithmetic unit, addition, subtraction, multiplication and division, has the same work load.

According to Section 3.2 the accuracy of floating-point numbers drops dramatic when exponent bit-width is less then seven and mantissa bit-width is less then 11. As a result will the FPUs with configurable bit-width be tested with exponent bit-width of eight and eleven and mantissa bit-width of eleven, 23 and 52.

To find the best design the power consumption, area usage and speed has to be considered. By mapping the design in Xilinx's tool ISE, the power analysis tool, XPower, can be used to simulate the expected power consumption. XPower also measures the static power consumption. Since the design is tested for FPGA the static power consumption will be the same for all designs. However, when designing for ASIC, parts of the circuit may be turn off and static power consumption is saved.

The speed of the systems is evaluated by analyzing the run time of each arithmetic operation individually. The latency is measured by counting the number of clock cycles from the unit is enabled and operands are presented, until the result and ready signal are presented on the output. If some operations use different time with different operands, worst case will be applied. The run times for asynchronous designs are dependent on the longest path through the design. The longest path for all asynchronous designs tested is shorter than the clock period used. This results in a total latency of one clock cycle. However, the latency may vary dependent on the total system size and the platform it is designed for.

5.2 Design and Performance

The Amber 2 core uses the double precision floating-point unit (FPU) discussed in Subsection 2.5.3. The port map for this unit, in the amber co-processor design, is described in Listing 5.1. The port map should be kept the same to more easily adapt to the co-processor. The bit-width can be adjusted by only handling parts of the already implemented 64 bit registers.

Listing 5.1: Port Map of Double Precision FPU in Amber Co-Processor.

```
// ============================================================
// FPU - Double
// ============================================================
a25_fpu_double u_fpu (
  .clk      (i_clk),
  .rst      (i_rst),
  .enable    (fpu_double_enable),
  .rmode     (fpu_rmode),
  .fpu_op    (fpu_opcode),
  .opa      (fpu_double_data_in_a),
  .opb      (fpu_double_data_in_b),
  .out      (fpu_double_data_out),
  .ready     (fpu_double_ready),
  .underflow  (fpu_double_underflow),
  .overflow  (fpu_double_overflow),
  .inexact   (fpu_double_inexact),
  .exception  (fpu_double_exception),
  .invalid   (fpu_double_invalid)
);
```

5.2.1 LogiCORE IP Floating-Point Operator

The LogiCORE IP Floating-Point discussed in Subsection 2.5.1 was first implemented. The top-level design using the Xilinx FPU is described in Figure 5.1 and a bigger image can be found in Appendix C, while the VHDL code can be found in Appendix B.1.1.

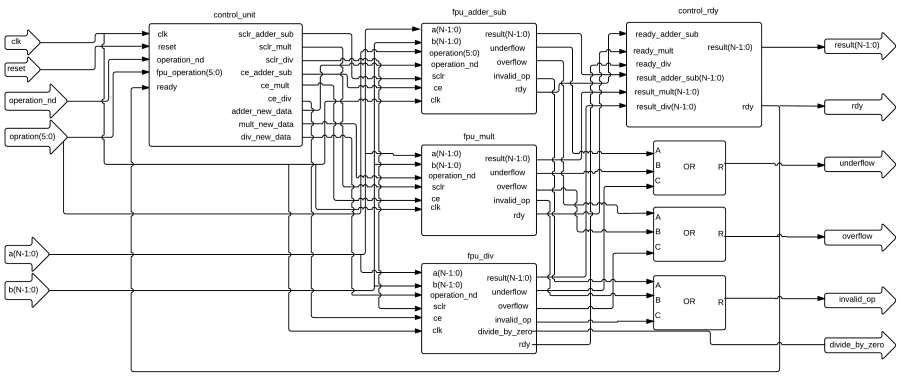

Figure 5.1: Diagram of Top-Level Design for Floating-Point Implementation.

The top-level design has the following input: a and b is the operands and can contain from eight to 80 bit. Next is the clock and reset which resets the system and sets the speed. The clock is not connected to the floating-point arithmetic units when an asynchronous design is used. The new data signal is set high when new operands are present. This signal has to be pulled low before the next operation is to be performed. The operation signal tells the unit what arithmetic operation to perform. Zero is for addition, one is for subtraction, two is for multiplication and three is for division. It is possible for the user to specify if underflow, overflow, invalid operation and divided by zero is present, however this will not be implemented while testing.

The signals, except the operands, are routed to a control unit that makes sure the arithmetic units have the correct input. It also makes sure that when one operation is performed, the other units are disabled using clock gating and the output is set to zero to lower the power consumption. The clock enable signal is not connected using asynchronous designs. The control unit is also controlled by a ready signal, which signals to turn of the arithmetic units when it is done. A flow diagram of this system is presented in Figure 5.2.

The final unit in the FPU design controls the ready signal and makes sure that the correct output is presented. This unit works as a multiplexer and only arithmetic units which presents a ready signal are allowed to send output data.

The resulting size, latency and power consumption for the Xilinx FPUs are listed in Table 5.1. The FPU has been tested with an asynchronous design with and without DSP slices, and a synchronous design with DSP slices. The difference in size between the asynchronous and synchronous design with DSP is small. However, the difference between the asynchronous designs is noticeable.

The power consumption for the different implementations is listed in the same order as for size. The power consumption for the clock in the asynchronous designs is having a minor influence on the total dynamic power consumption, compared with the synchronous. This results in the dynamic power consumption for asynchronous designs being lower than synchronous. The difference between the dynamic power consumptions for asynchronous designs with and without DSPs is quite small. This indicates that DSP slices have a bigger influence on the size than power consumption.

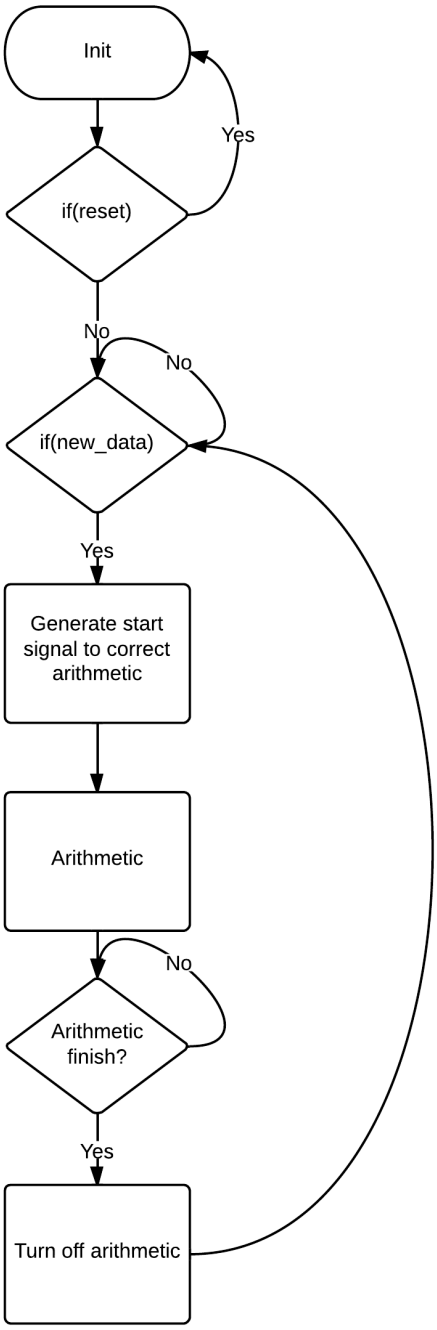

Figure 5.2: Flow Diagram from Input is Present to Output is Produced.

Table 5.1: Latency, Size and Power Consumption for Xilinx IP

e	m	Operation	Latency	Size				Comments
				DSPs	LUTs	Total DSPs	Total LUTs	
8	11	add/sub	1	0	212	2	562	Async DSP (1)
		mult	1	2	63			
		div	1	0	256			
		add/sub	1	0	212	0	757	Async No DSP (2)
		mult	1	0	266			
		div	1	0	256			
		add/sub	8	0	211	2	584	Sync DSP (3)
		mult	7	2	45			
		div	16	0	238			
8	23	add/sub	1	2	245	5	1111	Async DSP (4)
		mult	1	3	73			
		div	1	0	755			
		add/sub	1	0	395	0	1877	Async No DSP (5)
		mult	1	0	834			
		div	1	0	733			
		add/sub	11	2	257	5	1234	Sync DSP (6)
		mult	8	3	91			
		div	28	0	740			
11	52	add/sub	1	3	705	14	4082	Async DSP (7)
		mult	1	11	157			
		div	1	0	3150			
		add/sub	1	0	715	0	6633	Async No DSP (8)
		mult	1	0	3569			
		div	1	0	3150			
		add/sub	14	3	716	14	4314	Sync DSP (9)
		mult	15	11	114			
		div	57	0	3090			

e	m	Power consumption (mW)							Comments
		Clocks	Logic	Signals	IOs	DSPs	Static	Dynamic	
8	11	4.47	0.21	0.59	2.19	0.03	3,294.10	7.76	(1)
		6.29	0.29	0.49	2.17	0	3,294.10	9.23	(2)
		46.42	0.42	0.71	3.78	0.04	3,294.10	51.36	(3)
8	23	7.27	0.49	1.49	3.19	0.12	3,294.10	12.55	(4)
		7.88	0.86	1.41	3.16	0	3294.10	13.31	(5)
		90.48	0.75	2.29	7.28	0.09	3294.10	100.89	(6)
11	52	7.76	2.07	4.74	5.92	0.32	3294.10	20.81	(7)
		12.77	2.71	4.97	7.02	0	3294.10	27.48	(8)
		144.25	3.59	4.40	5.03	0.28	3294.10	157.55	(9)

5.2.2 OpenCores Single Precision Floating-Point Unit

The entity for the single precision FPU by OpenCores, listed in Listing 5.2, is about the same as the double precision. As a result, it is easy to adapt this unit to the coprocessor.

In Table 5.2 the size, latency and power consumption are presented. It is worth noticing that the size of the multiplication unit is much higher then the other arithmetic units.

Listing 5.2: Entity for Single Precision Floating-Point Unit by OpenCores

```
entity fpu is
    port (
        clk_i       : in std_logic;
        opa_i         : in std_logic_vector(FP_WIDTH-1 downto 0)
            ;
        opb_i           : in std_logic_vector(FP_WIDTH-1 downto
            0);
        fpu_op_i    : in std_logic_vector(2 downto 0);
        rmode_i     : in std_logic_vector(1 downto 0);
        output_o        : out std_logic_vector(FP_WIDTH-1 downto
            0);
        start_i     : in std_logic;
        ready_o     : out std_logic;
        ine_o       : out std_logic;
        overflow_o    : out std_logic;
        underflow_o   : out std_logic;
        div_zero_o    : out std_logic;
        inf_o     : out std_logic;
        zero_o      : out std_logic;
        qnan_o      : out std_logic;
        snan_o      : out std_logic
  );
end fpu;
```

5.2.3 OpenCores Double Precision Floating-Point Unit

This FPU is already implemented on the Amber 2 core. However, when simulating and implementing the unit, the software finds numeric operations that are not supported in Xilinx. In addition to that, when placing and routing the design, Xilinx software finds that the design is unroutable. As a result, there is no power analysis for this FPU. The latency and size is described in Table 5.3.

Table 5.2: Latency, Size and Power Consumption for OpenCores Single Precision Floating-Point Unit

e	m	Operation	Latency	Size			
				DSPs	LUTs	Total DSPs	Total LUTs
8	23	add/sub	8	0	1174	0	5892
		mult	13	0	3138		
		div	35	0	1424		

Power consumption (mW)						
Clocks	Logic	Signals	IOs	DSPs	Static	Total DPC
84.61	23.02	82.43	21.18	0	3,294.10	211.25

Table 5.3: Latency and Size for OpenCores Double Precision Floating-Point Unit

e	m	Operation	Latency	Size	
				DSPs	LUTs
11	52	add/sub	21/26	10	10457
		mult	29		
		div	71		

5.2.4 Floating-Point Library

This library is used as described in Subsection 2.5.4. A functional FPU design using this library is shown in Appendix B.1.4. This design has the same input and outputs as the Xilinx design in Figure 5.1, without the extra exceptions. Since the library is asynchronous, no advanced state machine is needed. The only state machine implemented is to set the ready signal one clock cycle after the *operation_nd* is presented. The numbers of clock cycles the operations takes may vary for different platforms, so this state machine only works for simulation.

The latency, size and power consumption is presented in Table 5.4. When testing the library, it is difficult to isolate each arithmetic operation. As a result, only the total size is presented.

5.2.5 Fixed-Point Library

This library is designed according to descriptions in Subsection 2.5.5. A fixed-point design using the library is presented in Appendix B.1.5. The design is quite similar to the top-level design of the floating-point library. However, the Xilinx synthesizer did not allow the division operand. As a result is division not implemented in the design. The test results for latency,

Table 5.4: Latency, Size and Power Consumption for Floating-Point Library

e	m	Operation	Latency	Size	
				Total DSPs	Total LUTs
8	11	add/sub mult div	1	1	1319
8	23	add/sub mult div	1	2	4159
11	52	add/sub mult div	1	15	15898

e	m	Power consumption (mW)						
		Clocks	Logic	Signals	IOs	DSPs	Static	Total DPC
8	11	6.25	1.73	4.49	17.96	0.02	3,294.10	30.45
8	23	12.44	5.81	10.54	30.24	0.02	3,294.10	59.04
11	52	14.49	39.27	69.22	48.72	0.28	3294.10	171.97

size and power consumption are presented in Table 5.5.

Table 5.5: Latency, Size and Power Consumption for Fixed-Point Library

k	l	Operation	Latency	Size	
				DSPs	LUTs
16	16	add/sub mult	1	4	37
		div	-		

Power consumption (mW)						
Clocks	Logic	Signals	IOs	DSPs	Static	Total DPC
14.79	0.21	2.18	20.49	0.05	3,294.10	37.72

5.2.6 Configurable Floating-Point Arithmetic Design

This unit is designed to better understand how floating-point arithmetic works in hardware and have an additional FPU that is more configurable for the user. It contains an adder, a subtractor and a multiplier. The algorithms are based on the algorithms in Section 2.2 and Subsection 2.5.2, which describes the single precision FPU from OpenCores. In later occasions will this unit be referred to as the "configurable design".

The configurable design is using a truncation rounding mode. This results in a minor error when comparing the results with the solution. The multiplication mode is having a bug that causes the wrong result to be generated when very big numbers are multiplied with very small. The floating-point unit do not handle exceptions. Otherwise the FPU is functional. The bugs and errors will be commented in the future work section later in this thesis.

Adder and Subtracter

When adding or subtracting two operands it is important to always know which operand that is greatest. Otherwise you are risking underflow when subtracting. One other aspect is that the arithmetic unit has to handle both positive and negative values. This leads to the unit handling the following situations: $a+b, -a+b, a+(-b), -a+(-b), a-b, -a-b, a-(-b), -a-(-b)$. This is described in the first two steps in the flow chart in Figure 5.3. Next, the size of each operand need to be measured to know which sign the result will have. Then the difference between the exponents have to be calculated and the smallest mantissa right shifted the same number as this difference. Now the mantissas can be added or subtracted. Next step is to round the resulting mantissa. Before the result is presented, any exceptions occurring have to be signaled, and the result has to be normalized. One way to normalize the result is to find the leading zero and then left shift the mantissa so it is represented as a decimal number between one and two, and then subtracting the exponent with the number of places shifted. This is typically an area consuming task, which expands when the mantissa bit-width is bigger. The VHDL code for this algorithm is presented in Appendix B.1.6.

Multiplier

The algorithm for multiplying two floating-point numbers are easier then the algorithm for addition and subtraction. The flow chart for this algorithm is presented in Figure 5.4 and the VHDL code is presented in Appendix B.1.7. The first step of the algorithm is to multiply the mantissas together and add the exponents. To prevent the bias from being added twice, it has to be subtracted. The next step is to find the resulting sign and normalize the result. A big difference between this algorithm and the adder and subtractor is that the normalization of the result is an easier task. If both operands are normalized, which means that their mantissa value is between one and two, the resulting mantissa value is between one and four. If the operands are denormalized, a more advanced algorithm is needed to normalize the result.

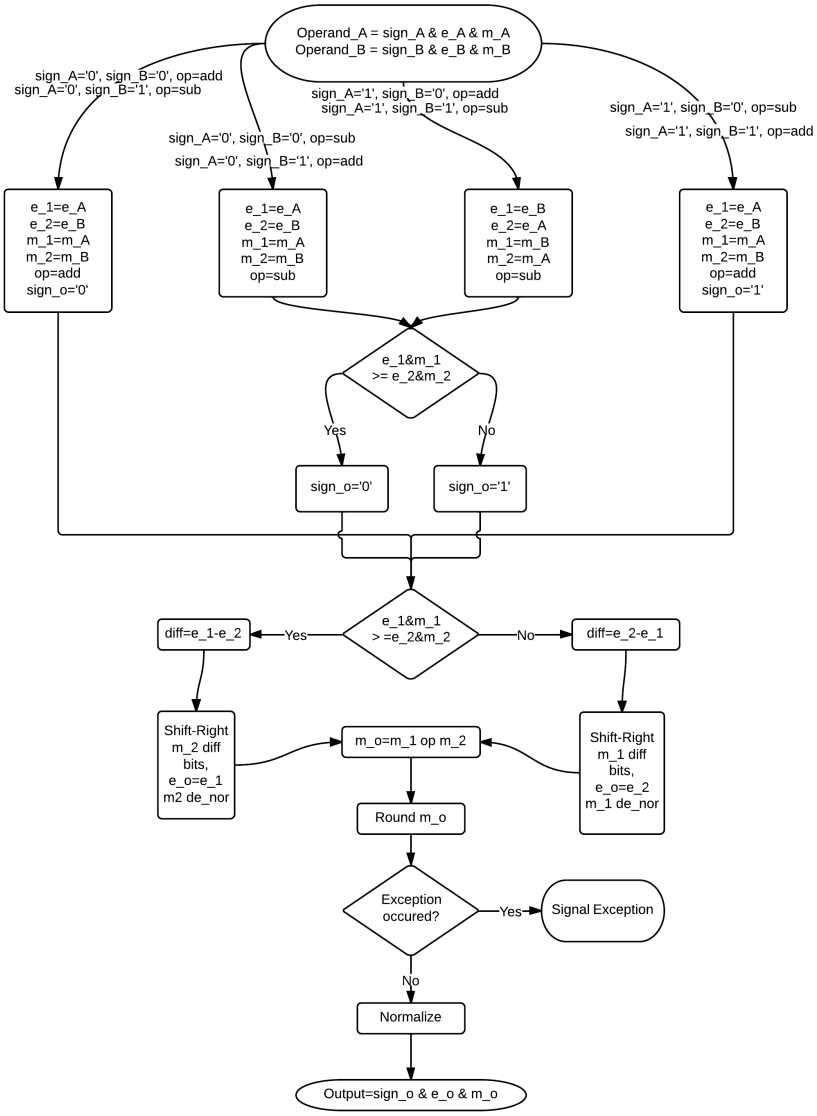

Figure 5.3: Flow Chart of Addition and Subtraction with Floating-Point Numbers.

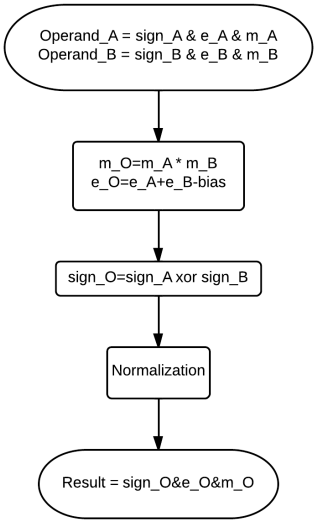

Figure 5.4: Flow Chart of Multiplication with Floating-Point Numbers.

Results

The resulting latency, area and power consumption is shown in Table 5.6. Notice that no divider is implementing and this will affect the total number of LUTs and DSPs.

Table 5.6: Latency, Size and Power Consumption for Configurable Floating-Point Unit

e	m	Operation	Latency	Size			
				DSPs	LUTs	Total DSPs	Total LUTs
8	11	add/sub	8	0	296	1	622
		mult	5	1	101		
		div	-	-	-		
8	23	add/sub	8	0	1155	2	1353
		mult	5	2	161		
		div	-	-	-		
11	52	add/sub	8	0	3814	15	4413
		mult	5	15	530		
		div	-	-	-		

e	m	Power consumption (mW)						
		Clocks	Logic	Signals	IOs	DSPs	Static	Total DPC
8	11	12.12	0.06	0.35	2.20	0	3,294.10	14.72
8	23	29.82	0.08	0.71	3.28	0	3,294.10	33.89
11	52	49.12	0.20	2.89	3.89	0	3,294.10	56.09

5.3 Precision and Range in Testbenches

To evaluate the testbenches and find the required range and precision, all input and output files were analyzed with Matlab scripts, and the largest and smallest number along with the highest precision is found. These are presented in Table 5.7 along with the calculated values for exponent (e) and mantissa (m) bit-width. Also represented are the bit-widths for fixed-point integer (k) and fraction (l). These numbers have not been added with one for compensating. The exponent values and integer values have been found by using Table 3.1 and 3.2, while the other calculations can be found in Appendix D.

Table 5.7: Range and Precision Analysis of Benchmarks

Benchmark	Largest	Smallest	Highest precision	e	m	k	l
177.mesa	9.8658	$3.21E-4$	6	5	18	4	20
179.art	99.2831228	28.3296161	7	4	31	7	24
183.equake	32.6156	$9.0400E-35$	37	8	20	6	123
188.ammp	20421.656321	0.2290	6	5	35	15	20

Chapter 6

Discussion

This chapter will discuss the results presented in Chapter 5 and analyze the gains of consistently choosing a floating-point unit (FPU) that suits your applications. The configurable design will be mentioned, but not compared with the others, since it do not include a floating-point divider. The double precision FPU by OpenCores will be compared for size and latency, but not for power consumption since this data is not available. The chapter is separated in three sections. Section 6.1 will compare the size and latency for different FPUs and analyze the gains of varying the bit-width. Section 6.2 compares the power consumption for different FPUs and Section 6.3 describes how to adapt an FPU to software.

6.1 Size and Latency Analysis

In the previous chapter, the resulting size for both Xilinx FPUs using asynchronous and synchronous design where presented. As discussed in Section 2.4 is asynchronous design independent of the clock signal, and can have lower power consumption. However, the extra "handshaking" signals, which replace the clock, use extra logic. This is not applicable for the Xilinx FPUs. According to Figure 6.1 the asynchronous design is smaller then the synchronous design using the same amount of DSPs. When increasing the bit-width the size increases exponentially and the usage of DSPs also increases. Comparing the asynchronous design with DSP and without DSP, the importance of exploiting the DSP slices are easily visualized. For a total bit-width of 64, the asynchronous FPU with DSP is using approximately 38% less LUTs. For further analysis with the other FPUs, the asynchronous FPU with DSP will be used.

In Table 6.1 the size and latency for different FPUs are presented. For

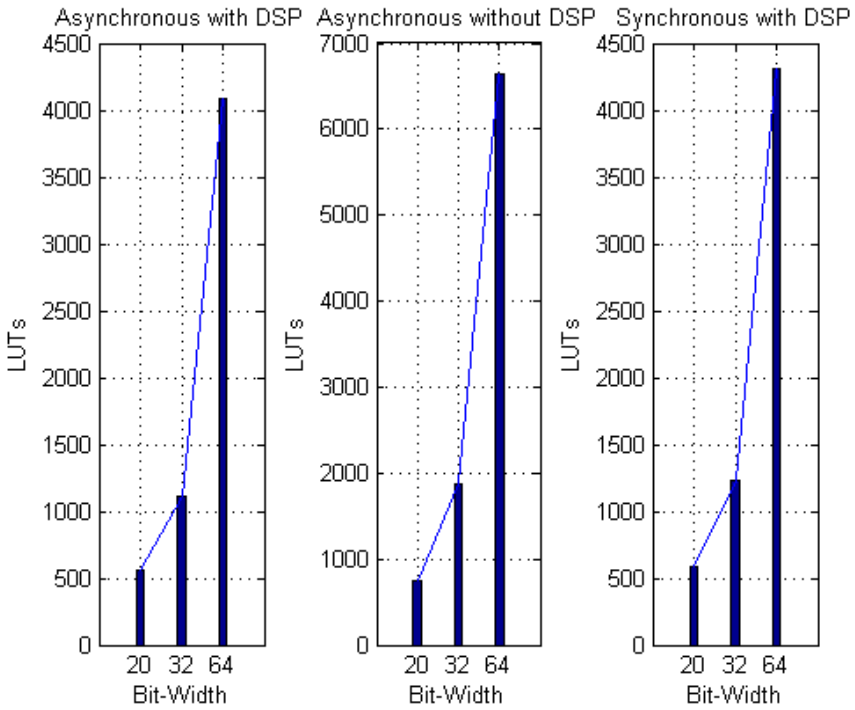

Figure 6.1: Number of LUTs for Different Xilinx Floating-Point Unit Designs.

all exponent and mantissa bit-widths the Xilinx FPU has a much lower area usage then the others. Comparing the Xilinx FPU with the smallest alternative, using a total bit-width of 20, approximately 57% of the LUTs is saved. For a total bit-width of 32, the Xilinx FPU uses approximately 73% less LUTs, and for 64 bit the Xilinx FPU uses 61% less LUTs. By comparing the smallest FPU with the largest overall, a total of approximately 96% of the LUTs can be saved.

The latency for asynchronous design is less then synchronous. By using a Xilinx FPU or the floating-point library instead of the OpenCores FPU for single precision, 87% less time is used when adding and subtracting, while for multiplication and division 92% and 97% are saved. For double precision, compared with the OpenCores FPU, 95% less time is used for addition and 96% less time for subtraction. For multiplication and division the time saved is 98%.

Implementing an FPU on an ASIC or an FPGA from another FPGA vendor, a combination of the floating-point library and the configurable design can be a good choice. If there is no need for configurability in the system, latency is no big concern and high precision and range is required, the double

precision FPU from OpenCores is a good alternative.

If an FPU is not suited for the system, a fixed-point unit may be. The latency is about the same as for the asynchronous FPUs and the numbers of LUTs for the fixed-point unit is only approximately 7% of the smallest FPU.

Table 6.1: Size and Latency for Different Floating-Point Units

FPU	e	m	Operation	Latency	Size			
					DSPs	LUTs	Sum DSPs	Sum LUTs
Xilinx	8	11	add/sub	1	0	212	2	562
			mult	1	2	63		
			div	1	0	256		
Floating-Point Library			add/sub mult div	1	-	-	1	1319
Xilinx	8	23	add/sub	1	2	245	5	1111
			mult	1	3	73		
			div	1	0	755		
Floating-Point Library			add/sub mult div	1	-	-	2	4159
OpenCores			add/sub	8	0	1174	0	5892
			mult	13	0	3138		
			div	35	0	1424		
Xilinx	11	52	add/sub	1	3	705	14	4082
			mult	1	11	157		
			div	1	0	3150		
Floating-Point Library			add/sub mult div	1	-	-	15	15898
OpenCores			add/sub	21/26	-	-	10	10457
			mult	29				
			div	71				

6.2 Power Analysis

In Figure 6.2 the dynamic power consumption for Xilinx FPUs designed asynchronous with and without DSP, and synchronous with DSP, all varying the bit-width, is shown. The asynchronous designs are using less power then synchronous, and the reason is that no clock is present in the asynchronous designs. As discussed in Section 2.7 can DSP slices result in lower power consumption. This results in the asynchronous design using DSPs having the lowest power consumption. Also, the increase in power is more linear

using DSPs then without. For further analysis, comparing with the other FPUs, the Xilinx FPU with DSPs will be used.

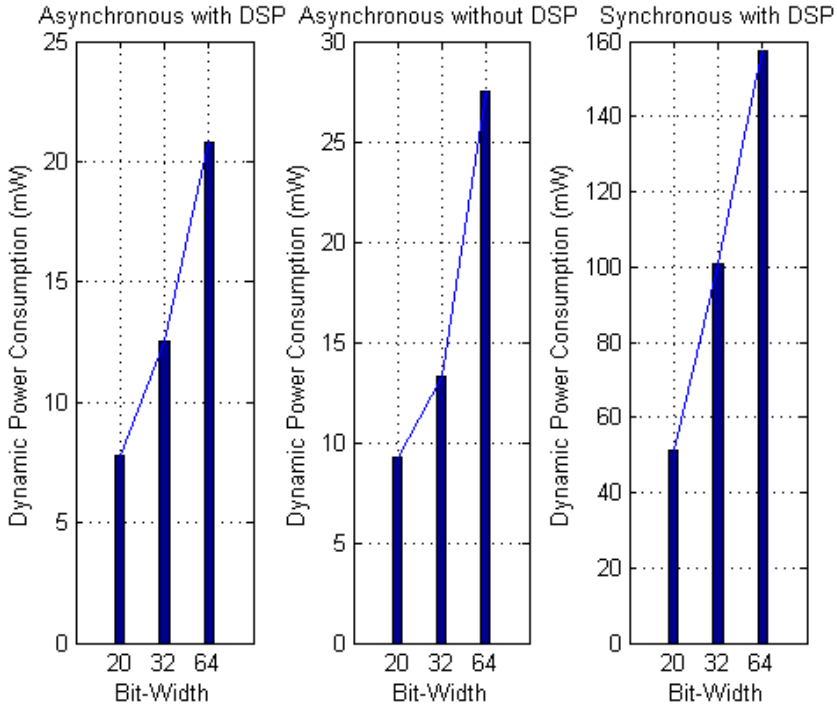

Figure 6.2: Dynamic Power Consumption with Different Xilinx Floating-Point Unit Designs.

Table 6.2: Power Consumption for Different Floating-Point Units

FPU	e	m	Dynamic Power Consumption (mW)
Xilinx	8	11	7.76
FP Library			30.45
Xilinx	8	23	12.55
FP Library			59.04
OpenCores			211.25
Xilinx	11	52	20.81
FP Library			171.97

In Table 6.2 the dynamic power consumption for different FPUs is described. For all bit-widths the dynamic power consumption is less for the Xil-

inx FPU, than the others. Using a total bit-width of 20, the Xilinx FPU is using approximately 75% less dynamic power then the other alternative. Using single precision, a total of 78% can be saved compared with the floating-point library and 94% compared with the FPU by OpenCores. For double precision a total of 88% dynamic power consumption can be saved. For single precision the floating-point library is using 72% less dynamic power consumption than the FPU by OpenCores. By comparing the least power hungry FPU with the most, a total of 96% dynamic power consumption can be saved. The floating-point library is a good alternative when designing for an FPGA from another vendor or an ASIC. Comparing the floating-point library with total bit-width of 20 and the single precision FPU by OpenCores, 86% dynamic power can be saved. The configurable design is also having good results for power consumption. Since the unit is not having a divider, a combination with the floating-point library may be a good alternative designing for an ASIC or another FPGA.

6.3 Optimization for Software

In the two previous sections the area and power consumption for different FPUs are described. Optimizing the FPU according to the needed precision and range can give a much better dynamic power consumption and size. In Section 5.3 the range and precision needed for the four benchmarks were evaluated. It is reasonable to believe that the benchmarks 177.mesa and 183.equake are using single precision floating-point numbers, while the 179.art and 188.ammp are using double precision. After compensating with the extra exponent and mantissa bit, a total of seven bit can be saved for the 177.mesa testbench, 26 bit can be saved for the 179.art testbench, one bit for the 183.equake and 21 bit for the 188.ammp. The fixed-point bit-width is less than the floating-point bit-width for the 179.art testbench and the 188.ammp testbench.

According to Table 2.4 in Section 2.8 there is a big difference in the frequency for the different operations. Common for all benchmarks are that dividing is the least used operation. In Figure 6.3 the percentage of the total size of each asynchronous arithmetic unit without DSP slices is presented. This figure shows that the total area occupied by the multiplier and divider grows when the bit-width increases. As a result, it is worth analyzing the gains of emulating division in software. According to Table 2.3 in Subsection 2.5.6 the latency for emulating division in software is 2494 clock cycles without any caches and 145 clock cycles with caches. These latencies are for single precision floating-point arithmetic. The extra overhead of using a

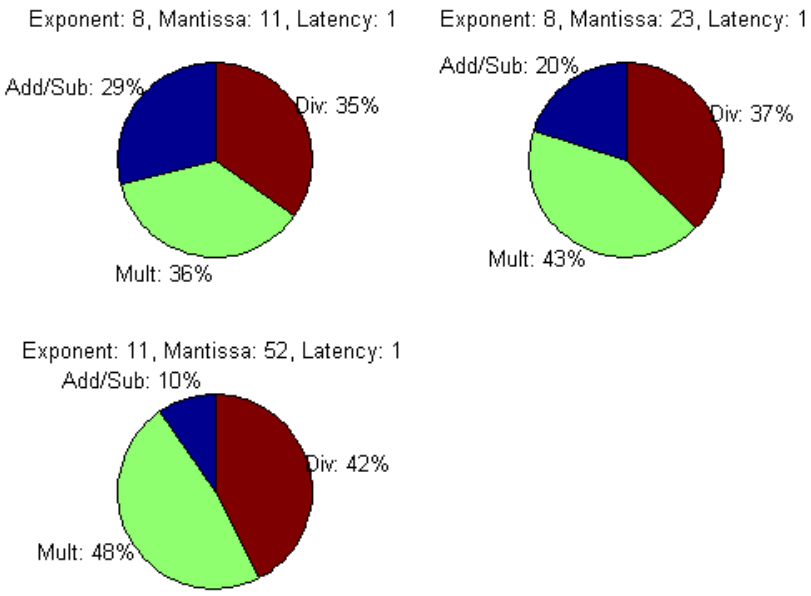

Figure 6.3: Cake Diagram of the Differences Between the Xilinx IP Arithmetic for Different Bit-Width with no DSP Usage.

single precision FPU in the Amber 2 co-processor, described in Subsection 2.6.1, is eight clock cycles for loading data and four clock cycles for storing data. The latency and size for the asynchronous Xilinx FPU will be used. This gives the following calculations:

Emulating division:

$$Latency/operation = \begin{cases} 2494 & \text{if no caches} \\ 145 & \text{if caches} \end{cases}$$

$$Area = 0$$

Hardware division:

$$Latency/operations = 1 + 8 + 4 = 13$$

$$Area = 733$$

If the processing core is designed without caches, the floating-point division is 99% faster then the software emulated division. Otherwise it is 91%

faster. If the system design is not time critical and a small hardware design is preferred, emulating floating-point division may be an option. Another option is to use the fixed-point library. The total size of the fixed-point library is 95% less then only the single precision divider.

Chapter 7

Conclusion

This thesis implements and tests configurable floating-point unit (FPU) for the SHMAC platform. These FPUs are useful for applications with a limited range and precision and when only parts of the comprehensive IEEE Standard 754 are needed. Software is analyzed and the bit-width is optimized to reduce area and power consumption.

An accurate area, latency and power analysis is done for different FPUs with different bit-width to enlighten the user of the gains that can be accomplished. The analysis shows that for Xilinx FPGAs the total area and dynamic power consumption can be reduced by up to 96%. For, ASIC and other platforms the reduced area and dynamic power consumption are up to 91% and 82%.

For the applications tested, a maximum of 33% of the bit-width for the floating-point numbers are unnecessary, and removing these leads to great performance and area gains. By moving parts of the FPU to software, even greater gains can be accomplished.

By choosing the proper FPU, 95% of the clock cycles can be reduced for floating-point addition. For subtraction and multiplication 96% of the clock cycles are reduces, while for division 98% of the clock cycles are reduced.

Future Work

This section contains work that was not done in this project because of time constrains or limitations in the SHMAC platform. The most important future work is the implementation and test of the FPUs on the SHMAC Platform.

Implement and Test the FPUs on the SHMAC Platform

To make sure that the different FPUs adapt to the SHMAC platform it is necessary to implement and test the designs. Since the co-processor in Amber Core has been modified during the spring of 2014 the FPUs and co-processor have never been tested together.

Design a Complier for Amber Core that Supports Hardware FPU

As of today, no SHMAC compiler supports hardware FPU. However, this subject has been studied during the spring of 2014 and there may only be small adjustments needed to complete.

Continue Design and Verification of the Configurable FPU

The configurable FPU designed during this project does not include a divider and the multiplication unit is generating the wrong results when multiplying very big numbers with very small To be able to fully replace the current FPU implemented on the Amber Core, this have to be fixed. In addition, does the configurable FPU not support denormalized numbers or exceptions like underflow and overflow. This is not critical functionality, but may be included as to support the reconfigurability of the unit.

Run-Time Reconfiguration FPU

Run-time reconfiguration for FPGA designs is an increasingly important requirement for many markets. By having a run-time reconfigurable FPU, the unit can adapt, in real-time, to different applications.

Bibliography

[1] S. Borkar, "Thousand Core Chips-A Technology Perspective," *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, June 2007.

[2] S. Borkar and A. A. Chien, "The Future of Microprocessors," *Communications of the ACM*, vol. 54, pp. 67–77, May 2011.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 365–376, 2011.

[4] "Single-ISA Heterogeneous MAny-core Computer (SHMAC) Project Plan," Oct. 2013.

[5] T. A. Rodolfo, N. L. V. Calazans, and F. G. Moraes, "Floating Point Hardware for Embedded Processors in FPGAs: Design Space Exploration for Performance and Area," *International Conference on Reconfigurable Computing and FPGAs*, pp. 24–29, 2009.

[6] Y. J. Chong and S. Parameswaran, "Custom Floating-Point Unit Generation for Embedded Systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 5.

[7] "IEEE Standard for Floating-Point Arithmetic," Aug. 2008. IEEE Std 754 ™-2008.

[8] J. Al-Eryani, "Floating Point Unit." `http://opencores.org/websvn,filedetails?repname=fpu100&path=%2Ffpu100%2Ftrunk%2Fdoc%2FFPU_doc.pdf`. [Online; accessed 20-Jan-2014].

[9] ARM EABI, *Sourcery G++ Lite*, 2010.

[10] "Chapter 7 – floating point arithmetic." `http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/arith.flpt.html`. [Online; accessed 25-May-2014].

[11] "Programming languages, their environments and system software interfaces — Extensions for the programming language C to support embedded processors," Apr. 2003. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1005.pdf "[Online; accessed 16-Jan-2014]".

[12] Xilinx, *LogiCORE IP Floating-PointOperator*, 5.0 ed., Mar. 2011. http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

[13] C. V. Berkel, M. B. Josephs, and S. M. Nowick, "Applications of Asynchronous Circuits," *Proceeding of The IEEE*, vol. 87, Feb 1999.

[14] D. Lundgren, *Double Precision Floating Point Core VHDL*. opencores.org, Feb. 2010. http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf, "[Online; accessed 18-May-2014]".

[15] D. Bishop, "Floating point package user's guide." http://www.vhdl.org/fphdl/Float_ug.pdf, Oct. [Online; accessed 27-May-2014].

[16] D. Bishop, "Fixed point package user's guide." http://www.eda.org/fphdl/Fixed_ug.pdf, Oct. 2013. [Online; accessed 7-Feb-2014].

[17] H. O. Wikene, "Benchmarking SHMAC," Master's thesis, Norwegian University of Science and Technology, 2013.

[18] C. Santifort, *Amber Open Source Project, Amber 2 Core Specification*, May 2013. http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf, "[Online; accessed 3-Mar-2014]".

[19] C. Santifort, *Amber Open Source Project, Amber Project User Guide*, May 2013. http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-user-guide.pdf, "[Online; accessed 3-Mar-2014]".

[20] OpenCores Organization, *Wishbone B4, WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores*, 2010. http://cdn.opencores.org/downloads/wbspec_b4.pdf, "[Online; accessed 3-Mar-2014]".

[21] J. D. Knutsen, "Implementing a SHMAC FPU Tile," Master's thesis, Norwegian University of Science and Technology, 2014.

[22] M. L. Teilgård, "Integration of Hardware Accelerators on the SHMAC Platform," Master's thesis, Norwegian University of Science and Technology, 2014.

[23] OpenCores Organization, *OpenRISC 1000 Architecture Manual*, 2012. `http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf`, "[Online; accessed 17-Mar-2014]".

[24] OpenCores Organization, *ORPSoC User Guide*, 2011.

[25] Embecosm Limited, *Or1ksim User Guide*, July 2012.

[26] Xilinx, *Virtex-5 Family Overview*, 5.0 ed., Feb. 2009. `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, "[Online; accessed 6-Feb-2014]".

[27] Avnet Electronics Marketing, *Xilinx®Spartan ®-6 LX 16, Evaluation Kit, User Guide*, 1.00 ed., Aug. 2010.

[28] ZedBoard.org, *ZedBoard (Zynq®Evaluation and Development Hardware User's Guide*, 2.2 ed., Jan. 2014.

[29] Xilinx, *ISE Design Suite 14: Release Notes, Installation, and Licensing*, Oct. 2013.

[30] Xilinx, *XST User Guide*, v 11.3 ed., Sept. 2009.

[31] Xilinx, *ISim User Guide*, 14.3 ed., Oct. 2012.

[32] Xilinx, *Xilinx Power Tools Tutorial*, 14.5 ed., Mar. 2013.

[33] "Standard Performance Evaluation Corporation." `http://www.spec.org`. [Online; accessed 08-May-2014].

[34] "The Physical Effects of Reliability Simulator." `http://www.persim.org`. [Online; accessed 08-May-2014].

[35] D. Defour, "Collapsing floating-point operations," *Universite de Perpignan*, 2012.

[36] Y. J. Chong and S. Parameswaran, "Configurable Multimode Embedded Floating-Point Units for FPGAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, pp. 2033–2044, Sept. 2011.

[37] J. Liang, R. Tessier, and O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs," *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium*, pp. 185–194, Apr. 2003.

[38] S. Galal and M. Horowitz, "Energy-Efficient Floationg-Point Unit Design," *Computers, IEEE Transactions on*, vol. 60, pp. 913–922, July 2011.

[39] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. Cheung, "Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs," *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium*, pp. 79–88, Apr. 2004.

[40] Y. F. Tong, R. A. Rutenbar, , and D. F. Nagle, "Minimizing Floating-Point Power Dissipation Via Bit-Width Reduction," *Power-Driven Microarchitecture Workshop in conjunction with the 25th International Symposium on Computer Architecture*, 1998.

Appendices

Appendix A

Matlab Code

A.1 Floating-Point Unit Testbench Generator

```
1  function result = generateTestbench(path, exponent_length,
       mantissa_length, samples_num)
2      file_testbench = fopen(path, 'w');
3
4      for i=1:samples_num,
5          % Need this cause Matlab does not handle so big numbers
6          if exponent_length+mantissa_length+1 > 52
7              a_1 = randi([0 1],1,exponent_length+mantissa_length
                   +1);
8              a_1 = num2str(a_1(1,:));
9              a = regexprep(a_1,'[^\w'']','');
10
11             b_1 = randi([0 1],1,exponent_length+mantissa_length
                   +1);
12             b_1 = num2str(b_1(1,:));
13             b = regexprep(b_1,'[^\w'']',''); % Remove all spaces
14
15             fprintf(file_testbench, '%s %s\n', a, b);
16         else
17             range = 2^(mantissa_length+exponent_length+1);
18             a = randi(range,1);
19             b = randi(range,1);
20             a_bin = dec2bin(a, exponent_length + mantissa_length
                       + 1);
21             b_bin = dec2bin(b, exponent_length + mantissa_length
                       + 1);
22             fprintf(file_testbench, '%s %s\n', a_bin, b_bin);
23         end
24     end
```

```
25
26      result = 'Generation Complete!';
27
28      fclose(file_testbench);
29  end
```

A.2 Floating-Point Unit Testbench Check

```
1  function result = compareResults(result1, result2, samples_num)
2      file1 = fopen(result1, 'r');
3      file2 = fopen(result2, 'r');
4
5      log = fopen('log.txt', 'w');
6
7      for i=1:samples_num,
8          file1_line = fgetl(file1);
9          values1 = sscanf(file1_line, '%x %d');
10         file2_line = fgetl(file2);
11         values2 = sscanf(file2_line, '%x %d');
12
13         if(values1(2) ~= values2(2))
14             str = sprintf('Wrong operation in line %d\n', i);
15             fprintf(log, '%s', str);
16         end
17         if(values1(1) ~= values2(1))
18             str = sprintf('Various results in line %d\n', i);
19             fprintf(log, '%s', str);
20         end
21     end
22     fclose(file1);
23     fclose(file2);
24     fclose(log);
25
26     result = 'Comparing Results Complete!';
27 end
```

A.3 Calculate Value of Floating-Point Numbers

```
1  function result = calculateOperation(a,exponent_length,
       mantissa_length)
2
3      format long
4
5      bias = 2^(exponent_length-1)-1;
6      a_bin = dec2bin(hex2dec(a),exponent_length+mantissa_length
           +1);
7
8      a_exponent = 0;
9      a_mantissa = 0;
10
11     denormalized = 0;
12
13     infinity_counter = 0;
14
15     % Calculate exponent value
16     for i=2:exponent_length+1
17         if(a_bin(i) == '1')
18           infinity_counter = infinity_counter + 1;
19           a_exponent = a_exponent+2^(exponent_length-i+1);
20         end
21     end
22
23     a_exponent = a_exponent - bias;
24     result = 2^a_exponent;
25
26      if(a_exponent ~= -127)
27         % Number is normalized
28         % Add the hidden mantissa bit
29         a_mantissa = 2^0;
30      else
31         % Number is de-normalized
32         disp('De-Normalized');
33         a_mantissa = 0;
34         denormalized = 1;
35      end
36
37      % Calculate mantissa value
38      for i=exponent_length+2:exponent_length+1+mantissa_length
39            if(a_bin(i) == '1')
40                if(denormalized == 1)
41                 a_mantissa = a_mantissa + (2^(-i+exponent_length
                       +2));
42                else
```

```
43                  a_mantissa = a_mantissa + (2^(-1-i+
                        exponent_length+2));
44                end
45            end
46        end
47
48        % Is number positive or negative
49        if(a_bin(1) == '1')
50            result = -1*result*a_mantissa;
51        else
52            result = result*a_mantissa;
53        end
54
55        if(infinity_counter == exponent_length)
56            if(a_mantissa == 1)
57                result = 'Infinity';
58            else
59                result = 'NaN';
60            end
61        end
62        return
63
64    end
```

Appendix B

HDL Code

B.1 Floating-Point Design

B.1.1 Top-Level Design for Xilinx IP

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  library work;
6  use work.all;
7
8  entity system is
9    generic(
10     OPERAND_LENGTH  : integer := 32;
11     EXPONENT_LENGTH   : integer := 8;
12     MANTISSA_LENGTH   : integer := 23;
13
14     EXCEPTIONS      : boolean := false
15   );
16   port(
17     a, b        :   in std_logic_vector(OPERAND_LENGTH-1 downto
         0);
18     operation   : in std_logic_vector(2 downto 0);
19     operation_nd  : in  std_logic;
20     --operation_rfd : out std_logic;
21     clk       : in std_logic;
22     --clk_a       : in std_logic;
23     reset       : in std_logic;
24     result_ip   :   out std_logic_vector(OPERAND_LENGTH-1 downto
         0);
25     --result_conf   :   out std_logic_vector(OPERAND_LENGTH-1
         downto 0);
```

```
26      rdy         : out std_logic;
27
28      underflow    : out std_logic;
29      overflow     : out std_logic;
30      invalid_op   : out std_logic;
31      divide_by_zero  : out std_logic
32      );
33  end system;
34
35  architecture Behavioral of system is
36
37    signal ready             : std_logic := '0';
38    signal ready_conf_adder_sub : std_logic := '0';
39    signal ready_conf_mult      : std_logic := '0';
40    signal ready_ip_adder_sub   : std_logic := '0';
41    signal ready_ip_mult        : std_logic := '0';
42    signal ready_ip_div       : std_logic := '0';
43
44
45    signal result_conf_adder_sub  : std_logic_vector(
        OPERAND_LENGTH-1 downto 0) := (others => '0');
46    signal result_conf_mult     : std_logic_vector(OPERAND_LENGTH
        -1 downto 0) := (others => '0');
47    signal result_ip_adder_sub    : std_logic_vector(
        OPERAND_LENGTH-1 downto 0) := (others => '0');
48    signal result_ip_mult     : std_logic_vector(OPERAND_LENGTH-1
        downto 0) := (others => '0');
49    signal result_ip_div        : std_logic_vector(OPERAND_LENGTH
        -1 downto 0) := (others => '0');
50
51    signal fpu_operation        : std_logic_vector(2 downto 0) :=
        "000";
52    signal adder_new_data     : std_logic := '0';
53    signal mult_new_data        : std_logic := '0';
54    signal div_new_data       : std_logic := '0';
55
56    signal sclr_adder_sub     : std_logic := '1';
57    signal sclr_mult          : std_logic := '1';
58    signal sclr_div         : std_logic := '1';
59
60    signal ce_adder_sub       : std_logic := '0';
61    signal ce_mult            : std_logic := '0';
62    signal ce_div           : std_logic := '0';
63
64    -- Comment this if using configurable design
65    signal result_conf        :   std_logic_vector(OPERAND_LENGTH
        -1 downto 0);
66
67    -- Used if EXCEPTIONS is true
```

```
68    signal underflow_add, underflow_mult, underflow_div       :
          std_logic := '0';
69    signal overflow_add, overflow_mult, overflow_div          :
          std_logic := '0';
70    signal invalid_op_add, invalid_op_mult, invalid_op_div    :
          std_logic := '0';
71    signal divide_by_zero_div                                 : std_logic
          := '0';
72
73    component fpu_adder_sub
74      port(
75        a           :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
76        b           :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
77        operation   : in std_logic_vector(5 downto 0); -- 0 for
              addition and 1 for subtraction
78        operation_nd  : in  std_logic; -- New Data. Must be set
              high to indicate that operand A, B and operation is
              valid
79        --clk        : in std_logic;
80        --sclr       : in std_logic; -- Synchronous Reset. Resets
              RDY and OPERATION_RFD output. Takes priority over CE
81        --ce         : in std_logic; -- Clock enable
82        result      :   out std_logic_vector(OPERAND_LENGTH-1
              downto 0);
83        rdy         : out std_logic -- Set high when result is valid
84        --underflow  : out std_logic;
85        --overflow   : out std_logic;
86        --invalid_op  : out std_logic
87      );
88    end component;
89
90    component fpu_multiplier
91      port(
92        a           :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
93        b           :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
94        operation_nd  : in  std_logic;
95        --clk        : in std_logic;
96        --sclr       : in std_logic;
97        --ce         : in std_logic;
98        result      :   out std_logic_vector(OPERAND_LENGTH-1
              downto 0);
99        rdy         : out std_logic
100       --underflow  : out std_logic;
101       --overflow   : out std_logic;
102       --invalid_op  : out std_logic
```

```
103     );
104   end component;
105
106   component fpu_divider
107     port(
108       a          :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
109       b          :   in std_logic_vector(OPERAND_LENGTH-1 downto
              0);
110       operation_nd  : in  std_logic;
111       --clk        : in std_logic;
112       --sclr         : in std_logic;
113       --ce           : in std_logic;
114       result       :   out std_logic_vector(OPERAND_LENGTH-1
              downto 0);
115       rdy        : out std_logic
116       --underflow    : out std_logic;
117       --overflow       : out std_logic;
118       --invalid_op     : out std_logic;
119       --divide_by_zero  : out std_logic
120     );
121   end component;
122
123
124 begin
125
126     ready <= ready_ip_adder_sub or ready_ip_mult or ready_ip_div
            ;
127     rdy <= ready;
128     -- ready_conf_adder_sub or
129
130     fpu_operation <= operation(2 downto 0);
131
132     underflow <= underflow_add or underflow_mult or
            underflow_div;
133     overflow <= overflow_add or overflow_mult or overflow_div;
134     invalid_op <= invalid_op_add or invalid_op_mult or
            invalid_op_div;
135     divide_by_zero <= divide_by_zero_div;
136
137     -- Control ready signal
138     process(clk, reset, ready_conf_adder_sub, ready_ip_adder_sub
            , ready_ip_mult, ready_ip_div)
139     begin
140       if(reset = '1') then
141         --sclr_adder_sub <= '0';
142 --        sclr_mult   <= '1';
143 --        sclr_div      <= '1';
144       else
```

```
145         if(rising_edge(clk)) then
146           if(ready_conf_adder_sub = '1') then
147             result_conf <= result_conf_adder_sub;
148           elsif(ready_conf_mult = '1') then
149             result_conf <= result_conf_mult;
150           elsif(ready_ip_adder_sub = '1') then
151             result_ip <= result_ip_adder_sub;
152             --sclr_adder_sub <= '1';
153           elsif(ready_ip_mult = '1') then
154             result_ip <= result_ip_mult;
155           elsif(ready_ip_div = '1') then
156             result_ip <= result_ip_div;
157           else
158           end if;
159
160         end if;
161       end if;
162     end process;
163
164     process(clk, reset, fpu_operation, operation_nd, ready)
165     begin
166       if(reset = '1') then
167         sclr_adder_sub <= '1';
168         sclr_mult <= '1';
169         sclr_div <= '1';
170
171         ce_adder_sub <= '0';
172         ce_mult <= '0';
173         ce_div <= '0';
174       else
175         if(rising_edge(clk)) then
176           if(operation_nd = '1') then
177             -- Addition and subtraction
178             if(fpu_operation = "000" or fpu_operation ="001")
                    then
179               adder_new_data <= '1';
180               mult_new_data <= '0';
181               div_new_data <= '0';
182
183               sclr_adder_sub <= '0';
184               sclr_mult <= '1';
185               sclr_div <= '1';
186
187               ce_adder_sub <= '1';
188               ce_mult <= '0';
189               ce_div <= '0';
190             -- Multiplication
191             elsif(fpu_operation = "010") then
192               mult_new_data <= '1';
```

```
193               adder_new_data <= '0';
194               div_new_data <= '0';
195
196               sclr_adder_sub <= '1';
197               sclr_mult <= '0';
198               sclr_div <= '1';
199
200               ce_adder_sub <= '0';
201               ce_mult <= '1';
202               ce_div <= '0';
203             -- Division
204             elsif(fpu_operation = "011") then
205               div_new_data <= '1';
206               adder_new_data <= '0';
207               mult_new_data <= '0';
208
209               sclr_adder_sub <= '1';
210               sclr_mult <= '1';
211               sclr_div <= '0';
212
213               ce_adder_sub <= '0';
214               ce_mult <= '0';
215               ce_div <= '1';
216             else
217               div_new_data <= '0';
218               adder_new_data <= '0';
219               mult_new_data <= '0';
220
221               sclr_adder_sub <= '1';
222               sclr_mult <= '1';
223               sclr_div <= '1';
224
225               ce_adder_sub <= '0';
226               ce_mult <= '0';
227               ce_div <= '0';
228             end if;
229           elsif(ready = '1') then
230             sclr_adder_sub <= '1';
231             sclr_mult <= '1';
232             sclr_div <= '1';
233             ce_adder_sub <= '0';
234             ce_mult <= '0';
235             ce_div <= '0';
236
237             adder_new_data <= '0';
238             mult_new_data <= '0';
239             div_new_data <= '0';
240           else
241
```

```
242           end if;
243         end if;
244       end if;
245     end process;
246
247     local_fpu_adder_sub : fpu_adder_sub
248     port map (
249       a                 => a,          -- input [31 : 0] a
250       b                 => b,          -- input [31 : 0] b
251       operation(2 downto 0)   => operation,    -- input [5 : 0]
              operation
252       operation(5 downto 3)   => "000",
253       operation_nd          => adder_new_data,   -- input
             operation_nd
254       --clk               => clk,        -- input clk
255       --sclr                => sclr_adder_sub,
256       --ce                  => ce_adder_sub,
257       result              => result_ip_adder_sub,    -- output
             [31 : 0] result
258       rdy               => ready_ip_adder_sub      -- output
             rdy
259       --underflow           => underflow_add,
260       --overflow             => overflow_add,
261       --invalid_op           => invalid_op_add
262     );
263
264
265     local_fpu_multiplier  : fpu_multiplier
266     port map(
267       a         => a,          -- input [31 : 0] a
268       b         => b,          -- input [31 : 0] b
269       operation_nd  => mult_new_data,  -- input operation_nd
270       --clk       => clk,        -- input clk
271       --sclr        => sclr_mult,
272       --ce          => ce_mult,
273       result      => result_ip_mult,     -- output [31 : 0]
             result
274       rdy       => ready_ip_mult     -- output rdy
275       --underflow           => underflow_mult,
276       --overflow             => overflow_mult,
277       --invalid_op           => invalid_op_mult
278     );
279
280     local_fpu_divider : fpu_divider
281     port map(
282       a         => a,          -- input [31 : 0] a
283       b         => b,          -- input [31 : 0] b
284       operation_nd  => div_new_data,   -- input operation_nd
285       --clk       => clk,        -- input clk
```

```
286         --sclr          => sclr_div,
287         --ce            => ce_div,
288         result      =>  result_ip_div,     -- output [31 : 0]
                result
289         rdy         =>  ready_ip_div       -- output rdy
290         --underflow           => underflow_div,
291         --overflow              => overflow_div,
292         --invalid_op            => invalid_op_div,
293         --divide_by_zero          => divide_by_zero_div
294       );
295
296
297   end Behavioral;
```

B.1.2 Adder and Subtracter

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5  use IEEE.std_logic_unsigned.ALL;
6
7  entity configurable_adder_unsigned is
8    generic (
9      OPERAND_LENGTH : integer  := 15;
10     EXPONENT_LENGTH : integer := 5;
11     MANTISSA_LENGTH : integer := 9
12     );
13   Port ( a     : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
       ;
14       b        : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
           ;
15       clk    : in std_logic;
16       reset    : in std_logic;
17       new_data  : in std_logic;
18       operation : in std_logic_vector(2 downto 0);
19       result   : out  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto
           0);
20       rdy    : out std_logic);
21 end configurable_adder_unsigned;
22
23 architecture Behavioral of configurable_adder_unsigned is
24
25 type state_type is (s0,s1,s2,s3,s4,s5,s6,s7,s8); --type of
     state machine.
26 signal current_s,next_s: state_type; --current and next state
     declaration.
27
28 -- Signals for adder
29 signal result_exponent  : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
30
31 signal res_mantissa : std_logic_vector(MANTISSA_LENGTH+1 downto
     0) := (others => '0');
32 signal big_exponent : std_logic_vector(EXPONENT_LENGTH-1 downto
     0) := (others => '0');
33 signal small_exponent : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
34 signal big_mantissa : std_logic_vector(MANTISSA_LENGTH-1 downto
     0) := (others => '0');
35 signal small_mantissa : std_logic_vector(MANTISSA_LENGTH-1
     downto 0) := (others => '0');
36 signal normalization    : std_logic := '0';
```

```
37 signal sign          : std_logic := '0';
38 signal sub_neg       : std_logic := '0';
39
40 signal clk_node      : std_logic := '0';
41 signal delay_clk      : std_logic := '0';
42 signal normalization_factor     : std_logic_vector(1 downto 0);
43 signal sig_operation    : std_logic := '0';
44
45 begin
46
47 process (clk,reset)
48 begin
49   if (reset='1') then
50     current_s <= s0;  --default state on reset.
51   elsif (rising_edge(clk)) then
52     current_s <= next_s;   --state change.
53   end if;
54 end process;
55
56 process (current_s, a, b, new_data, operation)
57 begin
58   case current_s is
59     when s0 =>
60       rdy <= '0';
61       res_mantissa <= (others => '0');
62       big_exponent <= (others => '0');
63       small_exponent <= (others => '0');
64       big_mantissa <= (others => '0');
65       small_mantissa <= (others => '0');
66       result <= (others => '0');
67       normalization <= '0';
68       sign  <= '0';
69       sub_neg <= '0';
70
71       if new_data = '1' then
72         next_s <= s1;
73       else next_s <= s0;
74       end if;
75     when s1 =>
76       result <= (others => '0');
77       rdy <= '0';
78       res_mantissa <= (others => '0');
79       sign <= '0';
80
81
82       if(operation(0) = '0') then
83         -- order do not matter
84         next_s <= s2;
85       elsif(operation(0) = '1') then
```

```
86          -- order matter
87        end if;
88
89        -- Allocate the biggest number to the biggest exponent and
              mantissa visa verca for further
90        -- operations
91        -- B bigger then A
92        if (b(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
              EXPONENT_LENGTH) > a(OPERAND_LENGTH-2 downto
              OPERAND_LENGTH-1-EXPONENT_LENGTH)) then
93          big_exponent <= b(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
94          small_exponent <= a(OPERAND_LENGTH-2 downto
                OPERAND_LENGTH-1-EXPONENT_LENGTH);
95          big_mantissa <= b(MANTISSA_LENGTH-1 downto 0);
96          small_mantissa <= a(MANTISSA_LENGTH-1 downto 0);
97
98          if(operation(0) = '0') then
99            -- order do not matter
100           next_s <= s2;
101         elsif(operation(0) = '1') then
102           -- order matter
103           -- If minus and minus
104           if(b(OPERAND_LENGTH-1) = '1') then
105             sign <= '0';
106             sub_neg <= '1';
107           else
108             sign <= '1';
109           end if;
110         else
111         end if;
112       -- A bigger or equal to B
113       else
114
115
116         -- A is negative
117         if(a(OPERAND_LENGTH-1) = '1') then
118           sign <= '1';
119         end if;
120         big_exponent <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
121         small_exponent <= b(OPERAND_LENGTH-2 downto
                OPERAND_LENGTH-1-EXPONENT_LENGTH);
122         big_mantissa <= a(MANTISSA_LENGTH-1 downto 0);
123         small_mantissa <= b(MANTISSA_LENGTH-1 downto 0);
124       end if;
125
126       next_s <= s2;
127     when s2 =>
```

```
128         result <= (others => '0');
129         rdy <= '0';
130         res_mantissa <= (others => '0');
131
132         -- Calculate the difference between the expononents
133         result_exponent <= big_exponent - small_exponent;
134 --        a_adder(EXPONENT_LENGTH-1 downto 0) <= big_exponent;
135 --        b_adder(EXPONENT_LENGTH-1 downto 0) <= small_exponent;
136 --        ce_adder <= '1';
137 --        add_adder <= '0';
138
139         next_s <= s3;
140       when s3 =>
141         res_mantissa <= (others => '0');
142         result <= (others => '0');
143         rdy <= '0';
144         normalization <= '0';
145
146         -- Shift the smallest mantissa x places to the right if
              the difference between the exponents are bigger then 0
147         -- If the mantissa is shifted, it is denormalized, going
              from 1.xxxx to 0.xxxx
148         if(result_exponent > 0 and result_exponent <=
              MANTISSA_LENGTH-1) then
149           small_mantissa <= std_logic_vector(unsigned(
                small_mantissa) srl to_integer(unsigned(
                result_exponent)));
150           small_mantissa(MANTISSA_LENGTH-to_integer(unsigned(
                result_exponent))) <= '1';
151           normalization <= '1';
152           next_s <= s4;
153         elsif(result_exponent > (MANTISSA_LENGTH-1)) then
154           -- a >> b
155           result_exponent <= big_exponent;
156           res_mantissa(MANTISSA_LENGTH-1 downto 0) <= big_mantissa
                ;
157           next_s <= s7;
158         else
159           next_s <= s4;
160         end if;
161
162       when s4 =>
163         result <= (others => '0');
164         rdy <= '0';
165
166         -- Check for addition or subtraction
167         if (operation(0) = '0' or sub_neg = '1') then
168           if(normalization = '1') then
```

```
169             res_mantissa <= ("01" & big_mantissa) + ("00" &
                    small_mantissa);
170           else
171             res_mantissa <= ("01" & big_mantissa) + ("01" &
                    small_mantissa);
172           end if;
173
174         elsif (operation(0) = '1') then
175           if(normalization = '1') then
176             res_mantissa <= ("01" & big_mantissa(MANTISSA_LENGTH-1
                     downto 0)) - ("00" & small_mantissa(
                    MANTISSA_LENGTH-1 downto 0));
177           else
178             res_mantissa(MANTISSA_LENGTH-1 downto 0) <= a(
                    MANTISSA_LENGTH-1 downto 0) - b(MANTISSA_LENGTH-1
                    downto 0);
179           end if;
180         else
181         end if;
182
183 --        if(normalization = '0') then
184 --          res_mantissa(MANTISSA_LENGTH) <= '1';
185 --        else
186 --          normalization <= '0';
187 --        end if;
188
189         next_s <= s5;
190       when s5 =>
191         rdy <= '0';
192         --res_mantissa <= (others => '0');
193
194         -- Test if the resulting mantissa is normalized
195         if(res_mantissa(MANTISSA_LENGTH+1 downto MANTISSA_LENGTH)
                > 1) then
196           -- Needs to be normalized, mantissa bigger then 2
197           normalization <= '0';
198
199           -- Exponent has to be added
200           result_exponent <= big_exponent + res_mantissa(
                  MANTISSA_LENGTH+1 downto MANTISSA_LENGTH) - 1;
201           -- Resulting mantissa is shifted 1 to right, normalized
202           res_mantissa(MANTISSA_LENGTH-1 downto 0) <= res_mantissa
                  (MANTISSA_LENGTH downto 1);
203
204
205           -- Resulting mantissa is shifted 1 to right, normalized
206 --          add_adder <= '1';
207 --          ce_adder <= '1';
208 --          a_adder(EXPONENT_LENGTH-1 downto 0) <= big_exponent;
```

```
209 --          if(result_exponent(MANTISSA_LENGTH+1 downto
    MANTISSA_LENGTH) = "10") then
210 --            b_adder(1 downto 0) <= "01";
211 --          else
212 --            b_adder(1 downto 0) <= "10";
213 --          end if;
214           next_s <= s7;
215         elsif (res_mantissa(MANTISSA_LENGTH+1 downto
              MANTISSA_LENGTH) = 1) then
216           -- Mantissa is normalized
217           normalization <= '0';
218           result_exponent <= big_exponent;
219           --res_mantissa <= res_mantissa(MANTISSA_LENGTH-1 downto
                0);
220           next_s <= s7;
221
222         else
223           normalization <= '0';
224
225           -- Find the position of the leading one
226           -- Is this the best way to do it????
227           for i in MANTISSA_LENGTH-1 downto 0 loop
228             if(res_mantissa(i) = '1') then
229             -- Leading one found
230               if((MANTISSA_LENGTH-i) > result_exponent) then
231                 res_mantissa <= (others => '0');
232                 result_exponent <= (others => '0');
233               else
234                 res_mantissa <= std_logic_vector(unsigned(
                      res_mantissa) sll (MANTISSA_LENGTH-i));
235                 result_exponent <= big_exponent - (MANTISSA_LENGTH
                      -i);
236               end if;
237
238 --              a_adder(EXPONENT_LENGTH-1 downto 0) <=
    big_exponent;
239 --              b_adder(4 downto 0) <= std_logic_vector(
    to_unsigned(MANTISSA_LENGTH-i,5));
240 --              add_adder <= '0';
241 --              ce_adder <= '1';
242               exit;
243             else
244
245             end if;
246           end loop;
247           next_s <= s7;
248         end if;
249
250       -- This state is only needed when exponent has to be changed
```

```
251 --      when s6 =>
252 --        a_adder <= (others => '0');
253 --        b_adder <= (others => '0');
254 --        result <= (others => '0');
255 --        ce_adder <= '0';
256 --        add_adder <= '1';
257 --        rdy <= '0';
258 --
259 --        big_exponent <= result_exponent(EXPONENT_LENGTH-1 downto
        0);
260 --        next_s <= s7;
261
262       when s7 =>
263         rdy <= '1';
264
265         result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
              EXPONENT_LENGTH) <= result_exponent;
266         result(MANTISSA_LENGTH-1 downto 0) <= res_mantissa(
              MANTISSA_LENGTH-1 downto 0);
267         result(OPERAND_LENGTH-1) <= sign;
268
269         next_s <= s0;
270
271       when others =>
272         result <= (others => '0');
273         rdy <= '0';
274         res_mantissa <= (others => '0');
275         big_exponent <= (others => '0');
276         small_exponent <= (others => '0');
277         big_mantissa <= (others => '0');
278         small_mantissa <= (others => '0');
279
280         next_s <= s0;
281     end case;
282 end process;
283
284
285 end Behavioral;
```

B.1.3 Multiplier

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.std_logic_unsigned.ALL;
5  use IEEE.NUMERIC_STD.ALL;
6
7  entity configurable_multiplier is
8    generic (
9      OPERAND_LENGTH : integer  := 32;
10     EXPONENT_LENGTH : integer := 8;
11     MANTISSA_LENGTH : integer := 23
12     );
13   Port ( a     : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
       ;
14       b       : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
           ;
15       clk     : in std_logic;
16       --clk_a    : in std_logic;
17       reset   : in std_logic;
18       new_data  : in std_logic;
19       --operation : in std_logic_vector(5 downto 0);
20       result  : out  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto
           0);
21       rdy     : out std_logic);
22 end configurable_multiplier;
23
24 architecture Behavioral of configurable_multiplier is
25
26 signal a_mantissa    : std_logic_vector(MANTISSA_LENGTH downto 0)
       := (others => '0');
27 signal b_mantissa    : std_logic_vector(MANTISSA_LENGTH downto 0)
       := (others => '0');
28 signal res_mantissa : std_logic_vector(MANTISSA_LENGTH*2+1
     downto 0) := (others => '0');
29 signal res_exponent : std_logic_vector(EXPONENT_LENGTH-1 downto
     0) := (others => '0');
30 signal res_sign    : std_logic := '0';
31
32 constant ZERO        : std_logic_vector(OPERAND_LENGTH-1 downto
     0) := (others => '0');
33 constant INFINITY    : std_logic_vector(EXPONENT_LENGTH-1 downto
     0) := (others => '1');
34 constant EXPONENT_ZERO  : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
35
36 -- Signals for adder
```

```
37 --signal a_adder       : std_logic_vector(23 downto 0) := (others
       => '0');
38 --signal b_adder       : std_logic_vector(23 downto 0) := (others
       => '0');
39 --signal add_adder     : std_logic := '0';
40 --signal ce_adder      : std_logic := '0';
41 --signal result_adder : std_logic_vector(24 downto 0) := (others
       => '0');
42
43
44 type state_type is (s0,s1,s2,s3,s4); --type of state machine.
45 signal current_s,next_s: state_type; --current and next state
       declaration.
46
47 component adder
48   port(
49     a : in std_logic_vector(23 downto 0);
50     b : in std_logic_vector(23 downto 0);
51     clk : in std_logic;
52     add : in std_logic;
53     ce    : in std_logic;
54     s    : out std_logic_vector(24 downto 0)
55   );
56 end component;
57
58 begin
59
60
61 --local_fraction_adder_sub : adder
62 --  port map(
63 --    a      => a_adder,
64 --    b    =>  b_adder,
65 --    clk  =>  clk_a,
66 --    add  =>  add_adder,
67 --    ce     => ce_adder,
68 --    s    =>  result_adder
69 --  );
70
71 process (clk,reset)
72 begin
73   if (reset='1') then
74     current_s <= s0; --default state on reset.
75   elsif (rising_edge(clk)) then
76     --clk_node <= clk;
77     current_s <= next_s;   --state change.
78   --else
79   --  clk_node <= clk;
80   end if;
81 end process;
```

```
82
83  process (current_s, a, b, new_data)
84  begin
85    case current_s is
86      when s0 =>
87        a_mantissa <= (others => '0');
88        b_mantissa <= (others => '0');
89        res_mantissa <= (others => '0');
90        res_exponent <= (others => '0');
91        res_sign <= '0';
92        result <= (others => '0');
93        res_sign <= '0';
94        rdy <= '0';
95
96        if new_data = '1' then
97          next_s <= s1;
98          a_mantissa(MANTISSA_LENGTH-1 downto 0) <= a(
                MANTISSA_LENGTH-1 downto 0);
99          a_mantissa(MANTISSA_LENGTH) <= '1';
100         b_mantissa(MANTISSA_LENGTH-1 downto 0) <= b(
                MANTISSA_LENGTH-1 downto 0);
101         b_mantissa(MANTISSA_LENGTH) <= '1';
102
103         --res_mantissa <= a(MANTISSA_LENGTH-1 downto 0) * b(
                MANTISSA_LENGTH-1 downto 0);
104         next_s <= s1;
105       else next_s <= s0;
106       end if;
107     when s1 =>
108       res_mantissa <= (others => '0');
109       res_exponent <= (others => '0');
110       res_sign <= '0';
111       result <= (others => '0');
112       res_sign <= '0';
113       rdy <= '0';
114
115       res_mantissa <= a_mantissa * b_mantissa;
116       res_exponent <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
              -1-EXPONENT_LENGTH) + b(OPERAND_LENGTH-2 downto
              OPERAND_LENGTH-1-EXPONENT_LENGTH) - (2**(
              EXPONENT_LENGTH-1)-1);
117       next_s <= s2;
118       -- Check if overflow => INFINITY
119 --      if(a(OPERAND_LENGTH-1) = '1' and b(OPERAND_LENGTH-1) =
      '1') then
120 --        --res_exponent <= a(OPERAND_LENGTH-2 downto
      OPERAND_LENGTH-1-EXPONENT_LENGTH) + b(OPERAND_LENGTH-2
      downto OPERAND_LENGTH-1-EXPONENT_LENGTH) - (2**(
      EXPONENT_LENGTH-1)-1);
```

```
121 --          result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) <= INFINITY;
122 --          rdy <= '1';
123 --          next_s <= s0;
124 --        -- Check if result is 0
125 --        elsif(a(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) = EXPONENT_ZERO or  b(OPERAND_LENGTH-2
    downto OPERAND_LENGTH-1-EXPONENT_LENGTH) = EXPONENT_ZERO)
    then
126 --          --result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) <= INFINITY;
127 --          next_s <= s0;
128 --          result <= ZERO;
129 --          rdy <= '1';
130 --        else
131 --          result <= ZERO;
132 --          next_s <= s0;
133 --          rdy <= '1';
134 --        end if;
135       when s2 =>
136         a_mantissa <= (others => '0');
137         b_mantissa <= (others => '0');
138         --res_exponent <= (others => '0');
139         res_sign <= '0';
140         result <= (others => '0');
141         rdy <= '0';
142         res_sign <= a(OPERAND_LENGTH-1) xor b(OPERAND_LENGTH-1);
143         if(res_mantissa(MANTISSA_LENGTH*2+1 downto MANTISSA_LENGTH
              *2) = "01") then
144           next_s <= s3;
145         elsif(res_mantissa(MANTISSA_LENGTH*2+1 downto
              MANTISSA_LENGTH*2) = "10") then
146           res_mantissa <= std_logic_vector(unsigned(res_mantissa)
                srl 1);
147           res_exponent <= res_exponent + 1;
148           next_s <= s3;
149         elsif(res_mantissa(MANTISSA_LENGTH*2+1 downto
              MANTISSA_LENGTH*2) = "11") then
150           res_mantissa <= std_logic_vector(unsigned(res_mantissa)
                srl 1);
151           res_exponent <= res_exponent + 2;
152           next_s <= s3;
153         else
154           next_s <= s0;
155         end if;
156       when s3 =>
157         a_mantissa <= (others => '0');
158         b_mantissa <= (others => '0');
159         --res_mantissa <= res_mantissa;
```

```
160         res_exponent <= res_exponent;
161         res_sign <= res_sign;
162         result <= (others => '0');
163         --rdy <= '1';
164
165         -- Check for rounding
166         if(res_mantissa(MANTISSA_LENGTH*2-MANTISSA_LENGTH-1) =
                '1') then
167           res_mantissa(MANTISSA_LENGTH*2-1 downto MANTISSA_LENGTH
                  *2-MANTISSA_LENGTH) <= res_mantissa(MANTISSA_LENGTH
                  *2-1 downto MANTISSA_LENGTH*2-MANTISSA_LENGTH) + '1';
168         else
169         end if;
170
171         next_s <= s4;
172
173
174       when s4 =>
175         result(OPERAND_LENGTH-1) <= res_sign;
176         result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
                EXPONENT_LENGTH) <= res_exponent(EXPONENT_LENGTH-1
                downto 0);
177         result(MANTISSA_LENGTH-1 downto 0) <= res_mantissa(
                MANTISSA_LENGTH*2-1 downto MANTISSA_LENGTH*2-
                MANTISSA_LENGTH);
178         rdy <= '1';
179
180         next_s <= s0;
181       when others =>
182
183     end case;
184   end process;
185
186   end Behavioral;
```

B.1.4 Top-Level Design for Floating-Point Library

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  library IEEE_PROPOSED;
5  use IEEE_PROPOSED.float_pkg.all;
6  use IEEE_PROPOSED.fixed_float_types.all;
7
8  entity system_ieee_float is
9    port(
10     clk        : in std_logic;
11     reset      :  in std_logic;
12     operation_nd  :  in std_logic;
13     operation  : in std_logic_vector(5 downto 0);
14     a          : in std_logic_vector(float_exponent_width+
           float_fraction_width downto 0);
15     b          : in std_logic_vector(float_exponent_width+
           float_fraction_width downto 0);
16     sum        : out std_logic_vector(float_exponent_width+
           float_fraction_width downto 0);
17     ready      : out std_logic
18   );
19 end system_ieee_float;
20
21 architecture Behavioral of system_ieee_float is
22
23   signal afp, bfp, sumfp : float(float_exponent_width downto -
       float_fraction_width);
24
25   type state_type is (s0,s1,s2);  --type of state machine.
26   signal current_s,next_s: state_type;  --current and next state
       declaration.
27
28 begin
29   afp <= to_float(a, afp'high, -afp'low);
30   bfp <= to_float(b, bfp'high, -bfp'low);
31
32   process (clk,reset)
33   begin
34     if (reset='1') then
35       current_s <= s0;  --default state on reset.
36     elsif (rising_edge(clk)) then
37       current_s <= next_s;   --state change.
38     else
39     end if;
40   end process;
41
42   process (current_s, operation_nd)
```

```
43    begin
44      case current_s is
45        when s0 =>
46          ready <= '0';
47          if(operation_nd = '1') then
48            next_s <= s1;
49          else
50            next_s <= s0;
51          end if;
52        when s1 =>
53          ready <= '1';
54        when s2 =>
55      end case;
56    end process;
57
58    process(clk, reset, operation_nd, operation, afp, bfp, sumfp)
59    begin
60      if(reset = '1') then
61        sumfp <= (others => '0');
62      elsif(rising_edge(clk)) then
63        if(operation_nd = '1') then
64          if(operation = "000000") then
65            sumfp <= afp + bfp;
66          elsif(operation = "000001") then
67            sumfp <= afp - bfp;
68          elsif(operation = "000010") then
69            sumfp <= afp * bfp;
70          elsif(operation = "000011") then
71            sumfp <= afp / bfp;
72          else
73          end if;
74        end if;
75      else
76      end if;
77      sum <= to_slv(sumfp);
78    end process;
79
80
81  end Behavioral;
```

B.1.5 Top-Level Design for Fixed-Point Library

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  library IEEE_PROPOSED;
4  use IEEE_PROPOSED.fixed_float_types.all;
5  use IEEE_PROPOSED.fixed_pkg.all;
6
7
8  entity system_ieee_fixed is
9    generic(
10     BIT_WIDTH    : integer := 32;
11     INTEGER_WIDTH : integer := 10;
12     FRACTION_WIDTH  : integer := 22
13   );
14   port(
15     clk        : in std_logic;
16     reset        : in std_logic;
17     a          : in std_logic_vector(BIT_WIDTH-1 downto 0);
18     b          : in std_logic_vector(BIT_WIDTH-1 downto 0);
19     operation    : in std_logic_vector(5 downto 0);
20     operation_nd  : in std_logic;
21     sum_add_sub_o : out std_logic_vector(BIT_WIDTH downto 0);
22     sum_mult_o     : out std_logic_vector(2*INTEGER_WIDTH-1+2*
         FRACTION_WIDTH downto 0);
23     ready        : out std_logic
24   );
25 end system_ieee_fixed;
26
27 architecture Behavioral of system_ieee_fixed is
28
29   signal afp, bfp    : sfixed(INTEGER_WIDTH-1 downto -
       FRACTION_WIDTH) := (others => '0');
30   signal sum_adder_sub  : sfixed(afp'left + 1 downto afp'right)
       := (others => '0');
31   signal sum_mult    : sfixed(afp'left+bfp'left+1 downto afp'
       right+bfp'right) := (others => '0');
32   --signal sum_div       : ufixed(afp'left-bfp'right+1 downto afp
       'right-bfp'left) := (others => '0');
33   signal sum_div       : sfixed(sfixed_high (afp, '/', bfp)
       downto sfixed_low (afp, '/', bfp));
34
35   type state_type is (s0,s1,s2); --type of state machine.
36   signal current_s,next_s: state_type; --current and next state
       declaration.
37
38 begin
39
40   afp <= to_sfixed(a, afp'left, afp'right);
```

```
41    bfp <= to_sfixed(b, bfp'left, bfp'right);
42
43    process (clk,reset)
44    begin
45      if (reset='1') then
46        current_s <= s0; --default state on reset.
47      elsif (rising_edge(clk)) then
48        current_s <= next_s;  --state change.
49      else
50      end if;
51    end process;
52
53    process (current_s, operation_nd)
54    begin
55      case current_s is
56        when s0 =>
57          ready <= '0';
58          if(operation_nd = '1') then
59            next_s <= s1;
60          else
61            next_s <= s0;
62          end if;
63        when s1 =>
64          ready <= '1';
65        when s2 =>
66      end case;
67    end process;
68
69    process(clk, reset)
70    begin
71      if(reset = '1') then
72        sum_add_sub_o <= (others => '0');
73        sum_mult_o <= (others => '0');
74      elsif(rising_edge(clk)) then
75        if(operation_nd = '1') then
76          if(operation = "000000") then
77            sum_adder_sub <= afp + bfp;
78          elsif(operation = "000001") then
79            sum_adder_sub <= afp - bfp;
80          elsif(operation = "000010") then
81            sum_mult <= afp * bfp;
82          elsif(operation = "000011") then
83 --           sum_div <= divide(  l => afp,
84 --                 r => bfp,
85 --                 round_style => fixed_round,
86 --                 guard_bits  => 3);
87            --sum_div <= afp/bfp;
88          else
89          end if;
```

```
90          else
91          end if;
92        else
93        end if;
94        sum_add_sub_o <= to_slv(sum_adder_sub);
95        sum_mult_o <= to_slv(sum_mult);
96      end process;
97
98
99  end Behavioral;
```

B.1.6 Configurable Adder and Subtractor

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4  use IEEE.std_logic_unsigned.ALL;
5
6  entity configurable_adder_sub is
7    generic (
8      OPERAND_LENGTH : integer  := 32;
9      EXPONENT_LENGTH : integer := 8;
10     MANTISSA_LENGTH : integer := 23
11     );
12   Port ( a     : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
       ;
13       b       : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
           ;
14       clk     : in std_logic;
15       reset   : in std_logic;
16       new_data  : in std_logic;
17       operation : in std_logic_vector(2 downto 0);
18       result  : out  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto
           0);
19       rdy     : out std_logic);
20 end configurable_adder_sub;
21
22 architecture Behavioral of configurable_adder_sub is
23
24 type state_type is (s0,s1,s2,s3,s4,s5,s6,s7,s8); --type of
     state machine.
25 signal current_s,next_s: state_type; --current and next state
     declaration.
26
27 -- Signals for adder
28 signal res_exponent       : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
29 signal res_mantissa       : std_logic_vector(MANTISSA_LENGTH+1
     downto 0) := (others => '0');
30 signal res_sign         : std_logic := '0';
31 signal diff_exponent       : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
32
33 signal exponent_1         : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
34 signal exponent_2         : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
35 signal mantissa_1         : std_logic_vector(MANTISSA_LENGTH-1
     downto 0) := (others => '0');
```

```
36 signal mantissa_2          : std_logic_vector(MANTISSA_LENGTH-1
     downto 0) := (others => '0');
37
38 signal local_op          : std_logic_vector(2 downto 0) := (
     others => '0');
39 signal normalization         : std_logic := '0';
40 signal de_normalized_1       : std_logic := '0';
41 signal de_normalized_2       : std_logic := '0';
42
43
44
45 begin
46
47 process (clk,reset)
48 begin
49   if (reset='1') then
50     current_s <= s0;  --default state on reset.
51   elsif (rising_edge(clk)) then
52     current_s <= next_s;   --state change.
53   end if;
54 end process;
55
56 process (current_s, a, b, new_data, operation)
57 begin
58   case current_s is
59     when s0 =>
60       res_exponent <= (others => '0');
61       res_mantissa <= (others => '0');
62       res_sign <= '0';
63       diff_exponent <= (others => '0');
64       exponent_1 <= (others => '0');
65       exponent_2 <= (others => '0');
66       mantissa_1 <= (others => '0');
67       mantissa_2 <= (others => '0');
68       rdy <= '0';
69       de_normalized_1 <= '0';
70       de_normalized_2 <= '0';
71
72
73       if new_data = '1' then
74         next_s <= s1;
75       else next_s <= s0;
76       end if;
77     when s1 =>
78
79       if( (a(OPERAND_LENGTH-1) = '0' and b(OPERAND_LENGTH-1) =
             '0' and operation = "000")
80         or (a(OPERAND_LENGTH-1) = '0' and b(OPERAND_LENGTH-1) =
              '1' and operation = "001")) then
```

```
81
82          exponent_1  <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
83          exponent_2  <= b(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
84          mantissa_1  <= a(MANTISSA_LENGTH-1 downto 0);
85          mantissa_2  <= b(MANTISSA_LENGTH-1 downto 0);
86          local_op    <= "000";
87          res_sign    <= '0';
88          next_s <= s3;
89
90        elsif((a(OPERAND_LENGTH-1) = '0' and b(OPERAND_LENGTH-1) =
              '0' and operation = "001")
91          or (a(OPERAND_LENGTH-1) = '0' and b(OPERAND_LENGTH-1) =
              '1' and operation = "000")) then
92
93          exponent_1  <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
94          exponent_2  <= b(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
95          mantissa_1  <= a(MANTISSA_LENGTH-1 downto 0);
96          mantissa_2  <= b(MANTISSA_LENGTH-1 downto 0);
97          local_op    <= "001";
98          res_sign    <= '0';
99          next_s <= s2;
100
101       elsif((a(OPERAND_LENGTH-1) = '1' and b(OPERAND_LENGTH-1) =
              '0' and operation = "000")
102         or (a(OPERAND_LENGTH-1) = '1' and b(OPERAND_LENGTH-1) =
              '1' and operation = "001")) then
103
104         exponent_1  <= b(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
105         exponent_2  <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
106         mantissa_1  <= b(MANTISSA_LENGTH-1 downto 0);
107         mantissa_2  <= a(MANTISSA_LENGTH-1 downto 0);
108         local_op    <= "001";
109         res_sign <= '1';
110         next_s <= s2;
111
112       else
113
114         exponent_1  <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
115         exponent_2  <= b(OPERAND_LENGTH-2 downto OPERAND_LENGTH
                -1-EXPONENT_LENGTH);
116         mantissa_1  <= a(MANTISSA_LENGTH-1 downto 0);
117         mantissa_2  <= b(MANTISSA_LENGTH-1 downto 0);
```

```
118           local_op     <= "000";
119           res_sign     <= '1';
120           next_s <= s3;
121
122         end if;
123
124       -- Diff with sign bit
125       when s2 =>
126           if((exponent_1&mantissa_1) > (exponent_2&mantissa_2))
                then
127             res_sign <= '0';
128             diff_exponent <= exponent_1 - exponent_2;
129             next_s <= s4;
130           else
131             res_sign <= '1';
132             diff_exponent <= exponent_2 - exponent_1;
133             next_s <= s5;
134           end if;
135       -- Diff without sign bit
136       when s3 =>
137         if((exponent_1&mantissa_1) > (exponent_2&mantissa_2)) then
138           diff_exponent <= exponent_1 - exponent_2;
139           next_s <= s4;
140         else
141           diff_exponent <= exponent_2 - exponent_1;
142           next_s <= s5;
143         end if;
144
145
146       when s4 =>
147         -- Shift the smallest mantissa x places to the right if
              the difference between the exponents are bigger then 0
148         -- If the mantissa is shifted, it is denormalized, going
              from 1.xxxx to 0.xxxx
149         if(diff_exponent > 0 and diff_exponent <= MANTISSA_LENGTH
              -1) then
150           mantissa_2 <= std_logic_vector(unsigned(mantissa_2) srl
                to_integer(unsigned(diff_exponent)));
151           mantissa_2(MANTISSA_LENGTH-to_integer(unsigned(
                diff_exponent))) <= '1';
152           de_normalized_2 <= '1';
153           res_exponent <= exponent_1;
154           next_s <= s6;
155         elsif(diff_exponent > (MANTISSA_LENGTH-1)) then
156           -- a >> b
157           res_exponent <= exponent_1;
158           res_mantissa(MANTISSA_LENGTH-1 downto 0) <= mantissa_1;
159           next_s <= s8;
160         else
```

```
161           next_s <= s6;
162         end if;
163
164
165
166       when s5 =>
167         -- Shift the smallest mantissa x places to the right if
              the difference between the exponents are bigger then 0
168         -- If the mantissa is shifted, it is denormalized, going
              from 1.xxxx to 0.xxxx
169         if(diff_exponent > 0 and diff_exponent <= MANTISSA_LENGTH
              -1) then
170           mantissa_1 <= std_logic_vector(unsigned(mantissa_1) srl
                to_integer(unsigned(diff_exponent)));
171           mantissa_1(MANTISSA_LENGTH-to_integer(unsigned(
                diff_exponent))) <= '1';
172           de_normalized_1 <= '1';
173           res_exponent <= exponent_2;
174           next_s <= s6;
175         elsif(diff_exponent > (MANTISSA_LENGTH-1)) then
176           -- a >> b
177           res_exponent <= exponent_2;
178           res_mantissa(MANTISSA_LENGTH-1 downto 0) <= mantissa_2;
179           next_s <= s8;
180         else
181           next_s <= s6;
182           res_exponent <= exponent_2;
183         end if;
184
185
186       when s6 =>
187         -- Check for addition or subtraction
188         if (local_op(0) = '0') then
189           if(de_normalized_1 = '1') then
190             res_mantissa <= ("00" & mantissa_1) + ("01" &
                  mantissa_2);
191           elsif(de_normalized_2 = '1') then
192             res_mantissa <= ("01" & mantissa_1) + ("00" &
                  mantissa_2);
193           else
194             res_mantissa <= ("01" & mantissa_1) + ("01" &
                  mantissa_2);
195           end if;
196
197         elsif (local_op(0) = '1') then
198           if(de_normalized_1 = '1') then
199             res_mantissa <= ("01" & mantissa_2)-("00" & mantissa_1
                  );--("00" & mantissa_1) - ("01" & mantissa_2);
200           elsif(de_normalized_2 = '1') then
```

```
201             res_mantissa <= ("01" & mantissa_1) - ("00" &
                    mantissa_2);
202           else
203             res_mantissa(MANTISSA_LENGTH-1 downto 0) <= mantissa_1
                    (MANTISSA_LENGTH-1 downto 0) - mantissa_2(
                    MANTISSA_LENGTH-1 downto 0);
204           end if;
205         else
206         end if;
207
208         next_s <= s7;
209
210       when s7 =>
211
212         -- Test if the resulting mantissa is normalized
213         if(res_mantissa(MANTISSA_LENGTH+1 downto MANTISSA_LENGTH)
                > 1) then
214           -- Needs to be normalized, mantissa bigger then 2
215           normalization <= '0';
216
217           -- Exponent has to be added
218           res_exponent <= res_exponent + res_mantissa(
                MANTISSA_LENGTH+1 downto MANTISSA_LENGTH) - 1;
219           -- Resulting mantissa is shifted 1 to right, normalized
220           res_mantissa(MANTISSA_LENGTH-1 downto 0) <= res_mantissa
                (MANTISSA_LENGTH downto 1);
221
222         elsif (res_mantissa(MANTISSA_LENGTH+1 downto
                MANTISSA_LENGTH) = 1) then
223           -- Mantissa is normalized
224           normalization <= '0';
225           --res_mantissa <= res_mantissa(MANTISSA_LENGTH-1 downto
                0);
226
227         else
228           normalization <= '0';
229
230           -- Find the position of the leading one
231           -- Is this the best way to do it????
232           for i in MANTISSA_LENGTH-1 downto 0 loop
233             if(res_mantissa(i) = '1') then
234             -- Leading one found
235               if((MANTISSA_LENGTH-i) > res_exponent) then
236                 res_mantissa <= (others => '0');
237               else
238                 res_mantissa <= std_logic_vector(unsigned(
                        res_mantissa) sll (MANTISSA_LENGTH-i));
239                 res_exponent <= res_exponent - (MANTISSA_LENGTH-i)
                        ;
```

```
240                 end if;
241
242 --                  a_adder(EXPONENT_LENGTH-1 downto 0) <=
        big_exponent;
243 --                  b_adder(4 downto 0) <= std_logic_vector(
        to_unsigned(MANTISSA_LENGTH-i,5));
244 --                  add_adder <= '0';
245 --                  ce_adder <= '1';
246                   exit;
247                 else
248
249                 end if;
250               end loop;
251
252             end if;
253             next_s <= s8;
254
255           when s8 =>
256             result <= res_sign & res_exponent & res_mantissa(
                  MANTISSA_LENGTH-1 downto 0);
257             rdy <= '1';
258             next_s <= s0;
259           when others =>
260
261             next_s <= s0;
262       end case;
263     end process;
264
265
266     end Behavioral;
```

B.1.7 Configurable Multiplier

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.std_logic_unsigned.ALL;
5  use IEEE.NUMERIC_STD.ALL;
6
7  entity configurable_multiplier is
8    generic (
9      OPERAND_LENGTH : integer  := 32;
10     EXPONENT_LENGTH : integer := 8;
11     MANTISSA_LENGTH : integer := 23
12     );
13   Port ( a     : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
       ;
14       b       : in  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto 0)
           ;
15       clk     : in std_logic;
16       --clk_a    : in std_logic;
17       reset   : in std_logic;
18       new_data  : in std_logic;
19       --operation : in std_logic_vector(5 downto 0);
20       result  : out  STD_LOGIC_VECTOR (OPERAND_LENGTH-1 downto
           0);
21       rdy     : out std_logic);
22 end configurable_multiplier;
23
24 architecture Behavioral of configurable_multiplier is
25
26 signal a_mantissa   : std_logic_vector(MANTISSA_LENGTH downto 0)
       := (others => '0');
27 signal b_mantissa   : std_logic_vector(MANTISSA_LENGTH downto 0)
       := (others => '0');
28 signal res_mantissa : std_logic_vector(MANTISSA_LENGTH*2+1
     downto 0) := (others => '0');
29 signal res_exponent : std_logic_vector(EXPONENT_LENGTH-1 downto
     0) := (others => '0');
30 signal res_sign    : std_logic := '0';
31
32 constant ZERO        : std_logic_vector(OPERAND_LENGTH-1 downto
     0) := (others => '0');
33 constant INFINITY    : std_logic_vector(EXPONENT_LENGTH-1 downto
     0) := (others => '1');
34 constant EXPONENT_ZERO  : std_logic_vector(EXPONENT_LENGTH-1
     downto 0) := (others => '0');
35
36 -- Signals for adder
```

```
37 --signal a_adder      : std_logic_vector(23 downto 0) := (others
       => '0');
38 --signal b_adder      : std_logic_vector(23 downto 0) := (others
       => '0');
39 --signal add_adder    : std_logic := '0';
40 --signal ce_adder     : std_logic := '0';
41 --signal result_adder : std_logic_vector(24 downto 0) := (others
       => '0');
42
43
44 type state_type is (s0,s1,s2,s3,s4); --type of state machine.
45 signal current_s,next_s: state_type; --current and next state
       declaration.
46
47 component adder
48   port(
49     a : in std_logic_vector(23 downto 0);
50     b : in std_logic_vector(23 downto 0);
51     clk : in std_logic;
52     add : in std_logic;
53     ce    : in std_logic;
54     s    : out std_logic_vector(24 downto 0)
55   );
56 end component;
57
58 begin
59
60
61 --local_fraction_adder_sub : adder
62 --  port map(
63 --    a      => a_adder,
64 --    b    =>  b_adder,
65 --    clk  =>  clk_a,
66 --    add  =>  add_adder,
67 --    ce     => ce_adder,
68 --    s    =>  result_adder
69 --  );
70
71 process (clk,reset)
72 begin
73   if (reset='1') then
74     current_s <= s0; --default state on reset.
75   elsif (rising_edge(clk)) then
76     --clk_node <= clk;
77     current_s <= next_s;   --state change.
78   --else
79   --  clk_node <= clk;
80   end if;
81 end process;
```

```
82
83  process (current_s, a, b, new_data)
84  begin
85    case current_s is
86      when s0 =>
87        a_mantissa <= (others => '0');
88        b_mantissa <= (others => '0');
89        res_mantissa <= (others => '0');
90        res_exponent <= (others => '0');
91        res_sign <= '0';
92        result <= (others => '0');
93        res_sign <= '0';
94        rdy <= '0';
95
96        if new_data = '1' then
97          next_s <= s1;
98          a_mantissa(MANTISSA_LENGTH-1 downto 0) <= a(
                MANTISSA_LENGTH-1 downto 0);
99          a_mantissa(MANTISSA_LENGTH) <= '1';
100         b_mantissa(MANTISSA_LENGTH-1 downto 0) <= b(
                MANTISSA_LENGTH-1 downto 0);
101         b_mantissa(MANTISSA_LENGTH) <= '1';
102
103         --res_mantissa <= a(MANTISSA_LENGTH-1 downto 0) * b(
                MANTISSA_LENGTH-1 downto 0);
104         next_s <= s1;
105       else next_s <= s0;
106       end if;
107     when s1 =>
108       res_mantissa <= (others => '0');
109       res_exponent <= (others => '0');
110       res_sign <= '0';
111       result <= (others => '0');
112       res_sign <= '0';
113       rdy <= '0';
114
115       res_mantissa <= a_mantissa * b_mantissa;
116       res_exponent <= a(OPERAND_LENGTH-2 downto OPERAND_LENGTH
              -1-EXPONENT_LENGTH) + b(OPERAND_LENGTH-2 downto
              OPERAND_LENGTH-1-EXPONENT_LENGTH) - (2**(
              EXPONENT_LENGTH-1)-1);
117       next_s <= s2;
118       -- Check if overflow => INFINITY
119 --      if(a(OPERAND_LENGTH-1) = '1' and b(OPERAND_LENGTH-1) =
      '1') then
120 --        --res_exponent <= a(OPERAND_LENGTH-2 downto
      OPERAND_LENGTH-1-EXPONENT_LENGTH) + b(OPERAND_LENGTH-2
      downto OPERAND_LENGTH-1-EXPONENT_LENGTH) - (2**(
      EXPONENT_LENGTH-1)-1);
```

```
121 --          result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) <= INFINITY;
122 --          rdy <= '1';
123 --          next_s <= s0;
124 --        -- Check if result is 0
125 --        elsif(a(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) = EXPONENT_ZERO or  b(OPERAND_LENGTH-2
    downto OPERAND_LENGTH-1-EXPONENT_LENGTH) = EXPONENT_ZERO)
    then
126 --          --result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
    EXPONENT_LENGTH) <= INFINITY;
127 --          next_s <= s0;
128 --          result <= ZERO;
129 --          rdy <= '1';
130 --        else
131 --          result <= ZERO;
132 --          next_s <= s0;
133 --          rdy <= '1';
134 --        end if;
135       when s2 =>
136         a_mantissa <= (others => '0');
137         b_mantissa <= (others => '0');
138         --res_exponent <= (others => '0');
139         res_sign <= '0';
140         result <= (others => '0');
141         rdy <= '0';
142         res_sign <= a(OPERAND_LENGTH-1) xor b(OPERAND_LENGTH-1);
143         if(res_mantissa(MANTISSA_LENGTH*2+1 downto MANTISSA_LENGTH
              *2) = "01") then
144           next_s <= s3;
145         elsif(res_mantissa(MANTISSA_LENGTH*2+1 downto
              MANTISSA_LENGTH*2) = "10") then
146           res_mantissa <= std_logic_vector(unsigned(res_mantissa)
                srl 1);
147           res_exponent <= res_exponent + 1;
148           next_s <= s3;
149         elsif(res_mantissa(MANTISSA_LENGTH*2+1 downto
              MANTISSA_LENGTH*2) = "11") then
150           res_mantissa <= std_logic_vector(unsigned(res_mantissa)
                srl 1);
151           res_exponent <= res_exponent + 2;
152           next_s <= s3;
153         else
154           next_s <= s0;
155         end if;
156       when s3 =>
157         a_mantissa <= (others => '0');
158         b_mantissa <= (others => '0');
159         --res_mantissa <= res_mantissa;
```

```
160         res_exponent <= res_exponent;
161         res_sign <= res_sign;
162         result <= (others => '0');
163         --rdy <= '1';
164
165         -- Check for rounding
166         if(res_mantissa(MANTISSA_LENGTH*2-MANTISSA_LENGTH-1) =
                '1') then
167           res_mantissa(MANTISSA_LENGTH*2-1 downto MANTISSA_LENGTH
                  *2-MANTISSA_LENGTH) <= res_mantissa(MANTISSA_LENGTH
                  *2-1 downto MANTISSA_LENGTH*2-MANTISSA_LENGTH) + '1';
168         else
169         end if;
170
171         next_s <= s4;
172
173
174       when s4 =>
175         result(OPERAND_LENGTH-1) <= res_sign;
176         result(OPERAND_LENGTH-2 downto OPERAND_LENGTH-1-
                EXPONENT_LENGTH) <= res_exponent(EXPONENT_LENGTH-1
                downto 0);
177         result(MANTISSA_LENGTH-1 downto 0) <= res_mantissa(
                MANTISSA_LENGTH*2-1 downto MANTISSA_LENGTH*2-
                MANTISSA_LENGTH);
178         rdy <= '1';
179
180         next_s <= s0;
181       when others =>
182
183     end case;
184   end process;
185
186   end Behavioral;
```

B.2 Floating-Point Unit Testbench

```
1  `timescale 1ns / 1ps
2
3  module system_tb;
4
5    // Inputs
6    reg [19:0] a;
7    reg [19:0] b;
8    reg [2:0] operation;
9    reg operation_nd;
10   reg clk;
11   reg reset;
12
13   // Outputs
14   wire [19:0] result_ip;
15   wire rdy;
16   wire underflow;
17   wire overflow;
18   wire invalid_op;
19   wire divide_by_zero;
20
21   integer    data_file; //file handler
22   integer    data_file_result; //file handler
23   integer  scan_file; //file handler
24   reg    [2:0] state;
25   integer counter;
26
27   `define NULL 0
28
29   parameter zero=0, one=1, two=2, three=3, four=4;
30
31   // Instantiate the Unit Under Test (UUT)
32   system uut (
33     .clk(clk),
34     .reset(reset),
35     .operation_nd(operation_nd),
36     .operation(operation),
37     .a(a),
38     .b(b),
39     .result_ip(result_ip),
40     .rdy(rdy),
41     .underflow(underflow),
42     .overflow(overflow),
43     .invalid_op(invalid_op),
44     .divide_by_zero(divide_by_zero)
45   );
46
47   initial begin
```

```
48      // Initialize Inputs
49      clk = 0;
50      reset = 1;
51      a = 0;
52      b = 0;
53      operation = 0;
54      operation_nd = 0;
55      counter = 0;
56
57      data_file = $fopen("fpu_custom.dat", "r");
58      data_file_result = $fopen("../MATLAB/
            fpu_testbench_results_ip_64.txt", "w");
59      if (data_file == `NULL) begin
60        $display("data_file handle is NULL");
61        $finish;
62      end
63
64      if (data_file_result == `NULL) begin
65        $display("data_file handle is NULL");
66        $finish;
67      end
68      #200;
69
70      reset = 0;
71
72      repeat (5000) @ (posedge clk);
73      $fclose(data_file);
74      $fclose(data_file_result);
75      reset = 1;
76      #100
77      $finish;
78
79    end
80
81    always @(posedge rdy) begin
82      #20
83      $fwrite(data_file_result, "%h %d\n", result_ip, operation);
84    end
85
86    always @(posedge state == three) begin
87      if(counter == 100)
88        reset = 1;
89      else
90        counter = counter + 1;
91    end
92
93    always @(state) begin
94      case (state)
95        zero:
```

```
96          operation_nd = 0;
97        one:
98          scan_file = $fscanf(data_file, "%b %b\n", a, b);
99        two:
100         if(operation == 3)
101           operation = 0;
102         else
103           operation = operation + 1;
104       three:
105         operation_nd = 1;
106       four:
107         operation_nd = 0;
108     endcase
109   end
110
111   always @(posedge clk or posedge reset) begin
112     if (reset == 1)
113       state = zero;
114     else
115       case (state)
116         zero:
117           if(reset == 1)
118             state = zero;
119           else if(rdy == 0)
120             state = one;
121           else
122             state = zero;
123         one:
124           state = two;
125         two:
126           state = three;
127         three:
128           state = four;
129         four:
130           //state = zero;
131           if(rdy == 0)
132             state = four;
133           else
134             state = zero;
135       endcase;
136   end
137
138   always #5 clk = !clk;
139
140 endmodule
```

Appendix C

Diagrams

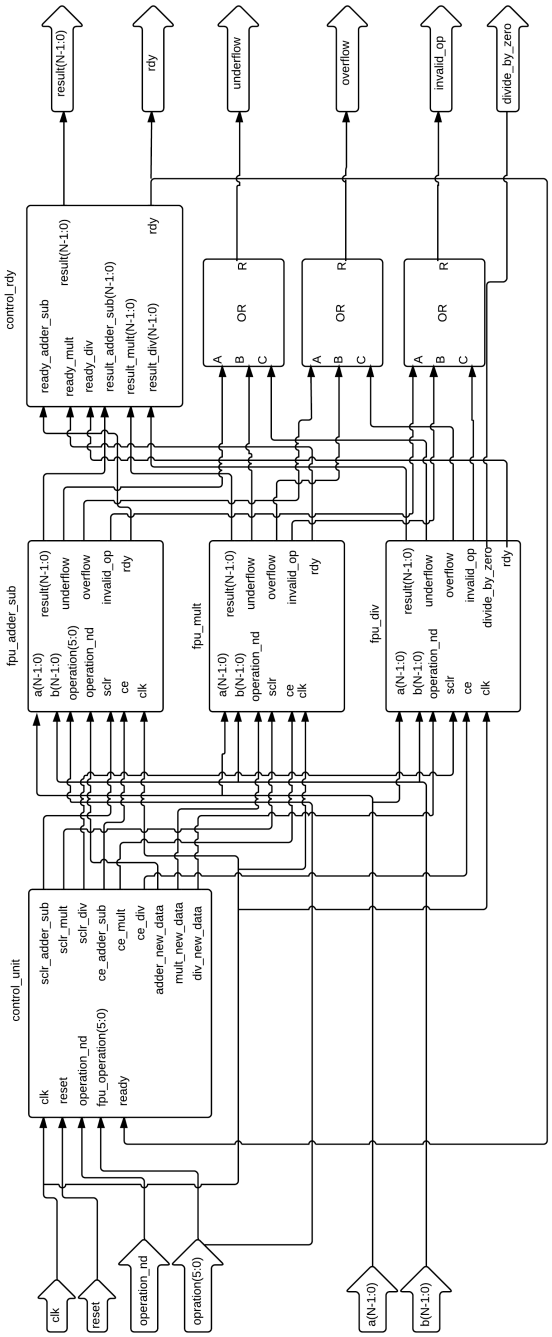

Figure C.1: Diagram of Floating-Point Implementation.

Appendix D

Calculations

D.1 Calculated Mantissa Bit-Width for Floating-Point Numbers

$$m \geq E_{U_i} - \lceil \log_2(\mid \Delta U_i \mid) \rceil + 1$$

177.mesa:

$$m \geq \lceil \log_2(9.8658) \rceil - \lceil \log_2(1E-4) \rceil + 1 = 4 - (-13) + 1 = 18$$
$$m \geq \lceil \log_2(3.21E-4) \rceil - \lceil \log_2(1E-6) \rceil + 1 = (-11) - (-19) + 1 = 9$$

179.art:

$$m \geq \lceil \log_2(99.2831228) \rceil - \lceil \log_2(1E-7) \rceil + 1 = 7 - (-23) + 1 = 31$$
$$m \geq \lceil \log_2(28.3296161) \rceil - \lceil \log_2(1E-7) \rceil + 1 = 5 - (-23) + 1 = 29$$

183.equake:

$$m \geq \lceil \log_2(32.6156) \rceil - \lceil \log_2(1E-4) \rceil + 1 = 6 - (-13) + 1 = 20$$
$$m \geq \lceil \log_2(9.04E-35) \rceil - \lceil \log_2(1E-37) \rceil + 1 = (-113) - (-122) + 1 = 10$$

188.ammp:

$$m \geq \lceil \log_2(20421.656321) \rceil - \lceil \log_2(1E-6) \rceil + 1 = 15 - (-19) + 1 = 35$$
$$m \geq \lceil \log_2(0.2290) \rceil - \lceil \log_2(1E-6) \rceil + 1 = (-2) - (-19) + 1 = 18$$

D.2 Calculated Fraction Bit-Width for Fixed-Point Numbers

$$l \geq \lceil \log_2(\mid \Delta U_i \mid) \rceil + 1$$

177.mesa:

$$l \geq \mid \lceil \log_2(1E-6) \rceil \mid +1 = 19 + 1 = 20$$

179.art:

$$l \geq \mid \lceil \log_2(1E-7) \rceil \mid +1 = 23 + 1 = 24$$

183.equake:

$$l \geq \mid \lceil \log_2(1E-37) \rceil \mid +1 = 122 + 1 = 123$$

188.ammp:

$$l \geq \mid \lceil \log_2(1E-6) \rceil \mid +1 = 19 + 1 = 20$$

Appendix E

File Hierarchy

```
master-thesis
├── fpu_core
│   └── fpu_core.xise
├── fpu_double
├── fpu100
├── MATLAB
├── Presentation
├── Report
│   ├── images
│   ├── Sources
│   └── MasterThesis.pdf
└── Result_tests
```

Attached to this thesis is a zip file containing the file hierarchy shown above. The fpu_core folder contains all HDL design. The file named fpu_core.xise can be opened in Xilinx ISE Design Suite 14.7. The folders fpu_double and fpu100 contain the double and single precision floating-point units by OpenCores. The MATLAB folder contains all Matlab scripts and functions. The Presentation folder contains two presentations that was used, presenting this thesis to younger students. The Report folder contains all images, some sources and the LaTeXfiles used to generate this article. The Result_tests folder contains all reports generated by XPower for different floating-point units.