Examensarbete

Low Cost Floating-Point Extensions to a Fixed-Point SIMD Datapath

Examensarbete utfört i Datorteknik vid Tekniska högskolan vid Linköpings universitet av

Gaspar Kolumban

LiTH-ISY-EX--13/4733--SE

Linköping 2013
Low Cost Floating-Point Extensions to a Fixed-Point SIMD Datapath

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av
Gaspar Kolumban

LiTH-ISY-EX--13/4733--SE

Handledare: Andréas Karlsson
ISY, Linköpings universitet

Examinator: Andreas Ehliar
ISY, Linköpings universitet

Linköping, 21 november 2013
Titel
Title
Low Cost Floating-Point Extensions to a Fixed-Point SIMD Datapath

Författare
Author
Gaspar Kolumban

Sammanfattning
Abstract

The ePUMA architecture is a novel master-multi-SIMD DSP platform aimed at low-power computing, like for embedded or hand-held devices for example. It is both a configurable and scalable platform, designed for multimedia and communications.

Numbers with both integer and fractional parts are often used in computers because many important algorithms make use of them, like signal and image processing for example. A good way of representing these types of numbers is with a floating-point representation. The ePUMA platform currently supports a fixed-point representation, so the goal of this thesis will be to implement twelve basic floating-point arithmetic operations and two conversion operations onto an already existing datapath, conforming as much as possible to the IEEE 754-2008 standard for floating-point representation. The implementation should be done at a low hardware and power consumption cost. The target frequency will be 500MHz. The implementation will be compared with dedicated DesignWare components and the implementation will also be compared with floating-point done in software in ePUMA.

This thesis presents a solution that on average increases the VPE datapath hardware cost by 15% and the power consumption increases by 15% on average. Highest clock frequency with the solution is 473MHz. The target clock frequency of 500MHz is thus not achieved but considering the lack of register retiming in the synthesis step, 500MHz can most likely be reached with this design.

Nyckelord
Keywords
ePUMA, floating-point, SIMD, VPE, fixed-point datapath, IEEE 754
Abstract

The ePUMA architecture is a novel master-multi-SIMD DSP platform aimed at low-power computing, like for embedded or hand-held devices for example. It is both a configurable and scalable platform, designed for multimedia and communications.

Numbers with both integer and fractional parts are often used in computers because many important algorithms make use of them, like signal and image processing for example. A good way of representing these types of numbers is with a floating-point representation. The ePUMA platform currently supports a fixed-point representation, so the goal of this thesis will be to implement twelve basic floating-point arithmetic operations and two conversion operations onto an already existing datapath, conforming as much as possible to the IEEE 754-2008 standard for floating-point representation. The implementation should be done at a low hardware and power consumption cost. The target frequency will be 500MHz. The implementation will be compared with dedicated DesignWare components and the implementation will also be compared with floating-point done in software in ePUMA.

This thesis presents a solution that on average increases the VPE datapath hardware cost by 15% and the power consumption increases by 15% on average. Highest clock frequency with the solution is 473MHz. The target clock frequency of 500MHz is thus not achieved but considering the lack of register retiming in the synthesis step, 500MHz can most likely be reached with this design.
Acknowledgments

I would like to thank Research Fellow Andreas Ehliar for allowing me to do this thesis. I would also like to thank Ph.D Student Andréas Karlsson for excellent supervision throughout the course of this thesis and for many interesting discussions. A thanks also to Martin Nielsen-Lönn, for keeping me company during the summer and for many fun discussions. Thanks also to Erik Karlsson, Jeremia Nyman and Robert Norlander for many fun times during our university years.

And finally a thanks to you, the Reader, I hope you can find something of value in this thesis.

Linköping, November 2013
Gaspar Kolumban
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation</td>
<td>ix</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Goal</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Scope</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Method</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>3</td>
</tr>
<tr>
<td>2 ePUMA</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Master Processor</td>
<td>7</td>
</tr>
<tr>
<td>2.3 VPE</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1 Data vectors</td>
<td>8</td>
</tr>
<tr>
<td>2.3.2 Datapath</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Further Information</td>
<td>9</td>
</tr>
<tr>
<td>3 Floating-Point</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Format</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Rounding</td>
<td>13</td>
</tr>
<tr>
<td>3.3 NaN</td>
<td>14</td>
</tr>
<tr>
<td>3.4 Exceptions</td>
<td>15</td>
</tr>
<tr>
<td>4 Operations</td>
<td>17</td>
</tr>
<tr>
<td>4.1 Addition</td>
<td>18</td>
</tr>
<tr>
<td>4.1.1 Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>4.1.2 Implementation</td>
<td>20</td>
</tr>
<tr>
<td>4.2 Multiplication</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1 Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 Implementation</td>
<td>23</td>
</tr>
<tr>
<td>4.2.3 VPE adaptation</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Division</td>
<td>26</td>
</tr>
<tr>
<td>4.3.1 Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>4.3.2 Implementation</td>
<td>28</td>
</tr>
</tbody>
</table>
# Notation

## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACR</td>
<td>Accumulator Register</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor/Processing</td>
</tr>
<tr>
<td>ePUMA</td>
<td>embedded Parallel DSP processor with Unique Memory Access</td>
</tr>
<tr>
<td>FP</td>
<td>Floating-Point</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-Point Unit</td>
</tr>
<tr>
<td>FPgen</td>
<td>Floating-point test generator</td>
</tr>
<tr>
<td>GTP</td>
<td>Generic Test Plan</td>
</tr>
<tr>
<td>HR</td>
<td>Help Register</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>LUT</td>
<td>Lookup Table</td>
</tr>
<tr>
<td>LVM</td>
<td>Local Vector Memory</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply and Accumulate</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NaN</td>
<td>Not a Number</td>
</tr>
<tr>
<td>qNaN</td>
<td>Quiet NaN</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RNE</td>
<td>Round to nearest even</td>
</tr>
<tr>
<td>RNU</td>
<td>Round to nearest up</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>sNaN</td>
<td>Signaling NaN</td>
</tr>
<tr>
<td>SRF</td>
<td>Special Register File</td>
</tr>
<tr>
<td>VPE</td>
<td>Vector Processing Element</td>
</tr>
<tr>
<td>VRF</td>
<td>Vector Register File</td>
</tr>
</tbody>
</table>
Many important algorithms today need to be able to support numbers that have both integer and fractional components, like image and signal processing for example. There are different ways of representing these kinds of numbers. One way is through the use of a fixed-point representation, where there is a variable amount of integer bits combined with a variable amount of fractional bits that has an implicit radix point which can alternate position depending on the program context. For example 0110 in Q1.3 notation (the version with the explicit sign) would have zero integer bits and three fractional bits and have a decimal value of 0.75. Further reading about fixed-point representation can be found in [10] and [15].

Depending on the application, higher precision and/or larger dynamic range might be needed and in order to accommodate this need of increased dynamic range and precision, the designer can increase the width of the integer/fractional bits in the fixed-point representation, but this is very expensive since the amount of bits required to represent large numbers and/or have high precision quickly escalates. A better approach is to implement support for another type of number representation, a floating-point representation, that has large dynamic range and precision. The representation chosen for this thesis is the IEEE standard 754 single precision floating-point number format.

1.1 Goal

The goal of this thesis is to implement support for some basic floating-point arithmetic operations for the ePUMA DSP platform that is being researched and designed at the Computer Engineering Department of Linköping University. The
modifications will be done to a fixed-point datapath. It is implied that the implement-
mentation needs to have a small hardware and power consumption cost and to not affect the max clock frequency too much. The target clock frequency will be 500MHz.
The operations that are to be supported are the following:

<table>
<thead>
<tr>
<th>#</th>
<th>Operation</th>
<th>Rounding</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FPADD</td>
<td>Yes</td>
<td>a + b</td>
</tr>
<tr>
<td>2</td>
<td>FPSUB</td>
<td>Yes</td>
<td>a - b</td>
</tr>
<tr>
<td>3</td>
<td>FPMUL</td>
<td>Yes</td>
<td>a * b</td>
</tr>
<tr>
<td>4</td>
<td>FPABS</td>
<td>No</td>
<td>abs(a)</td>
</tr>
<tr>
<td>5</td>
<td>FNEG</td>
<td>No</td>
<td>-a</td>
</tr>
<tr>
<td>6</td>
<td>FMIN</td>
<td>No</td>
<td>min(a,b)</td>
</tr>
<tr>
<td>7</td>
<td>FMAX</td>
<td>No</td>
<td>max(a,b)</td>
</tr>
<tr>
<td>8</td>
<td>FPABS</td>
<td>Yes</td>
<td>abs(a - b)</td>
</tr>
<tr>
<td>9</td>
<td>CVTWF2FP</td>
<td>No</td>
<td>Fixed to floating-point conversion</td>
</tr>
<tr>
<td>10</td>
<td>CVTFP2W</td>
<td>No</td>
<td>Float to fixed-point conversion</td>
</tr>
<tr>
<td>11</td>
<td>FPDIV</td>
<td>Yes</td>
<td>a / b</td>
</tr>
<tr>
<td>12</td>
<td>FPRECI</td>
<td>No</td>
<td>1 / a (faster, less precision)</td>
</tr>
<tr>
<td>13</td>
<td>FMINM</td>
<td>No</td>
<td>a if</td>
</tr>
<tr>
<td>14</td>
<td>FMAXM</td>
<td>No</td>
<td>a if</td>
</tr>
</tbody>
</table>

Table 1.1: List of operations to be implemented together with their function.

A rounding mode of round to nearest even (RNE) will also be implemented for FPADD, FPSUB, FPMUL, FPABS and FPDIV. FPRECI will not support rounding since this operation already produces a result that is inexact.

1.2 Scope

The IEEE standard for floating-point numbers [1], has a variety of features that will not be supported in this thesis, such as support for denormal numbers, proper handling of sNaN/qNaN and support for all rounding modes. Any operation that yields a NaN or ∞ will not raise any flags or exceptions. The result will just be saved as is and it will be up to the programmer to decide on how to handle these results.

The operations will support only single precision floating-point numbers. This means that each floating-point number is represented in a 32-bit format with 1 sign bit, 8 exponent bits and 23 mantissa bits.

For more information on the representation of single precision floating-point numbers in the IEEE standard 754, see Chapter 3 or [1].
1.3 Method

The thesis will be divided into three phases. In the first phase the initial ground work will be done; to figure out how these operations should be implemented and then, to verify the chosen algorithms, a simple simulator will be written to test the operations. The programming language chosen to write this simulator is C. The test data used comes from the generic test-plan (GTP) from the FPgen package from IBM. For the operations that does not have test data from IBM; self made test data will be used. For more information on FPgen, see Section 6.1, [5] and [6].

In phase two the mission will be to modify an already existing cycle-true and pipeline accurate simulator, written in C++, to be able to run the chosen algorithms. This phase will focus on trying to reduce the amount of extra hardware that is introduced in order to keep the hardware cost low.

In phase three it will be time to write the register-transfer level (RTL) code of what was accomplished in phase two and evaluate the implementation costs. The RTL code will be written in SystemVerilog. For further information on SystemVerilog see [16] and [14]. In this phase minimizations of the hardware can also occur.

1.4 Outline

This thesis contains the following chapters:

- **Chapter 1 - Introduction**: Some background information, goal and scope of the thesis and how the thesis will be done.
- **Chapter 2 - ePUMA**: An overview of the ePUMA architecture.
- **Chapter 3 - Floating-Point**: Details on how floating-point numbers are represented.
- **Chapter 4 - Operations**: Information on the different theoretical requirements for the different operations with the chosen solutions for each operation.
- **Chapter 5 - Hardware Implementation**: An overview of the unified datapath that can handle all operations.
- **Chapter 6 - Verification**: On how the design was tested.
- **Chapter 7 - Synthesis**: A synthesis comparison between floating-point and non floating-point datapath.
- **Chapter 8 - Evaluation**: Determine if and when the proposed design is worth adding to the datapath.
- **Chapter 9 - Conclusions**: Conclusions from the thesis and what can be done in the future.
The ePUMA architecture is a novel master-multi-SIMD DSP platform aimed at low-power computing, like for embedded or hand-held devices for example. It is both a configurable and scalable platform, designed for multimedia and communications.

The general idea in this type of architecture is to have a single master processor together with a number of co-processors. The single master processors purpose is to control program flow, run programs or parts of programs that are not suited for parallelization and delegate tasks to the co-processors. The smaller co-processors main purpose is to run arithmetic operations on vector data. This allows this type of architecture to achieve very high throughput. A notable example of this type of architecture is the Cell Broadband Engine, that was developed by Sony Computer Entertainment, Toshiba Corporation and IBM. Famous for being the main CPU in the PlayStation 3™gaming console, it also has other commercial applications. For more information on the Cell Broadband Engine, refer to [4].

The focus of this thesis is to modify the VPE (see Section 2.3) datapath to be able to handle some basic single precision floating-point arithmetic operations. This chapter will not present a detailed description of the entire architecture. Only a brief overview will be presented in order to bring things into context.

### 2.1 Overview

The ePUMA architecture is meant to be modular in order to make it easier to design towards a specific application. A master processor can be combined with any number of cores, called Slave clusters, in the ePUMA case. Inside the Slave clusters themselves, different kinds of accelerators may be present together with
a variable number of VPE cores. In order for ePUMA to run at all, there are a number of on-chip networks so that different parts of the ePUMA architecture can communicate with each other when needed.

An example configuration with a master processor, 4 Slave clusters, a ring network and a star network is shown in Figure 2.1.

![Figure 2.1: An overview of an example ePUMA configuration together with off-chip external memory.](image)

Some of the main components are as follows:

- **Master processor**: One master processor to communicate with the off-chip main memory and to delegate tasks to the Slave cluster cores when needed.

- **Slave clusters**: A core that contains a main processor (called Slave controller and is visible as "SC" in Figure 2.1) together with a variable number of VPE cores and accelerators.

- **VPE cores**: A Vector Processing Element that does arithmetic computations on vectors.

- **Accelerators**: A Slave cluster can contain any number of highly specialized accelerators to speed up sophisticated algorithms.

- **On-chip network**: In order for the ePUMA to function, there are different types of on-chip networks. One is a ring network which allows the different Slave cluster cores to communicate with each other. This will enable
different Slave clusters to perform different computational tasks in large and complex algorithms. Another type of network is a star network, which is used to move data and program code from main memory to Slave clusters and back to main memory.

2.2 Master Processor

At the very top level of the ePUMA architecture is a master processor. This processor has many different tasks to perform, for example to coordinate the activities of the different Slave clusters and to perform smaller computational tasks that are not efficient to perform in the Slave clusters.

Inside each Slave cluster is also a type of master processor (although not to be confused with the "Master" in the top level) called Slave controller (see "SC" in Figure 2.1). The job of this controller is in many ways the same as for the master processor one level up, only that now it is responsible for coordinating the activities of the VPE cores and the accelerators instead of the Slave clusters.

2.3 VPE

VPE stands for Vector Processing Element. It is in this module that vectors are operated on. The VPE internal structure can be seen in Figure 2.2. Some of its main components are the following:

- **PM**: The program memory contains the current program that is executing. This memory is filled by the master processor (see Figure 2.1).

- **LVM**: There are three local vector memories in each VPE core. They are the main data memories where all the operand, intermediate and output vectors are stored. Out of the three LVMs, only two are accessible during program execution. The third one is used to transfer data to and from the VPE core while a program is being executed, so that the next program in many cases already has its data inputs present in memory when the current program finishes. This scheme allows for highly efficient memory management, which is important because memories tend to be bottlenecks when trying to achieve a high degree of parallelization.

- **VRF**: The vector register file contains a configurable number of data vectors.

- **SRF**: The special register file contains various address registers for addressing the LVMs and other memories. It also contains the top and bottom registers for modulo addressing.

- **Datapath**: Where all the arithmetic operations are done, it takes two data vectors that are each 128-bits wide and produces a single 128-bit output data vector.
2.3.1 Data vectors

In a Slave cluster core there are a variable amount of VPE modules present. Each VPE module operates on 128-bit length data vectors. This is what allows the ePUMA architecture to have such huge potential throughput. A data vector consist of scalars (of different sizes) that are packed together. Some of the different formats can be seen in Figure 2.3.

The different sizes are 8-bit scalars (bytes), 16-bit scalars (words) and 32-bit scalars (double words) which can yield 16, 8 or 4 simultaneous results respectively. The VPE module also has support for complex numbers and some of their formats can also be seen in Figure 2.3.

The floating-point extension that are to be implemented in this thesis will make use of the double word vector format.

2.3.2 Datapath

The datapath of the VPE module will be the focus of this thesis project, since all of the arithmetic hardware is in there. The datapath is divided into three major stages and has four pipeline steps. A basic overview of the datapath can be seen in Figure 2.4. It does not depict all the existing hardware in the datapath in order to keep the explanation as simple as possible.

The first stage is the multiplier stage. It contains sixteen 16x16-bit multipliers. These multipliers can multiply two's complement numbers, either signed or unsigned. This stage has one pipeline step.
The second stage contains the adder tree. This stage has most of the adders in them, in three "rows" (see Figure 2.4) where each row is eight adders wide. They are used for basic ALU operations like add, sub, min, max, abs. This stage is two pipeline steps.

The third and final stage is the MAC stage. It has another ALU together with an accumulator register and other post processing units, like shifting, rounding and saturation for each lane (see Figure 2.4) and there are eight identical lanes in this stage. It is also in this stage that the flags for different operations are set. This stage is one pipeline step.

Not all instructions require so much hardware before producing a useful result, which is why there are two kinds of operations, short and long datapath operations. Long operations make use of all three stages in the datapath while short operations bypass the two first stages (MUL and Adder stages) and only make use of the final stage, the MAC stage.

### 2.4 Further Information

This chapter has only provided a brief overview of the ePUMA architecture in order to get things into context. A more detailed description of the ePUMA architecture (although slightly outdated) can be seen in [7]. Some performance numbers can also be seen in [9].
Figure 2.4: A basic overview of the VPE datapath.
The focus of this thesis project is to implement support for some arithmetic operations on floating-point numbers. The chosen floating-point representation is the IEEE 754-2008 standard single precision format. It is thus important to know how these floating-point numbers work and how they are represented. This chapter will serve as an introduction to the IEEE 754-2008 standard [1] on floating-point numbers, with an emphasis on the single precision format.

3.1 Format

With a fixed-point notation, it is possible to represent a wide range of numbers centered around 0. By assuming a radix point at a fixed position, this format allows the representation of fractional numbers as well. This notation however has limitations because very large numbers or very small fractions cannot be represented. In the decimal number system, this problem is avoided by using scientific notation. With scientific notation large numbers, for example 123 000 000 000 000 can be represented as $1.23 \times 10^{17}$, and very small fractions, for example 0.000000000000000123 can be represented as $1.23 \times 10^{-17}$. This type of notation allows very large numbers and very small fractions to be represented with only a few digits.

The IEEE 754-2008 standard uses the same approach to represent floating-point binary numbers. In general, a floating-point binary number will look like this:

$$(-1)^S \cdot M \cdot 2^E$$

All the floating-point binary formats in the IEEE 754-2008 standard consists of three parts, which are the following:
• **Sign S**: Signifies whether the number is positive or negative. A 0 means positive and a 1 means that the number is negative.

• **Exponent E**: Is the exponent of the number. This exponent is biased in the IEEE 754-2008 standard.

• **Mantissa M**: The significand (or coefficient) of the number.

The IEEE 754-2008 standard defines binary formats for 16, 32, 64, 128-bits and in any 32-bit multiple larger than 128-bits. The parameters for each format is shown in Table 3.1. For a more detailed table, refer to [1].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>bin16</th>
<th>bin32</th>
<th>bin64</th>
<th>bin128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
</tr>
<tr>
<td>Sign bits</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Exponent bits</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Mantissa bits</td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>112</td>
</tr>
<tr>
<td>Total</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
</tbody>
</table>

*Table 3.1: Table of different IEEE 754-2008 floating-point binary formats.*

The number of mantissa bits shown in Table 3.1 is one less than its actual precision. This is because the leading digit is always a 1 and this leading 1 is never explicitly stored. There are numbers when this does not apply. These numbers are called denormal/subnormal numbers and the leading digit in these types of numbers is a 0. Subnormal/denormal numbers fill the gap that exists between the smallest normalized number and zero (see Table 3.2). Subnormal/denormal numbers are outside the scope of this thesis and will not be supported. These types of numbers will be considered equal to zero.

A biased exponent means that the real value of the exponent is offset by a value that is known as the *exponent bias*. This is done because exponent values need to be signed in order to represent small and large numbers, but using two's complement for example, is not good as this makes comparisons harder because it would require special floating-point hardware for comparisons. With this setup (sign first, exponent second, mantissa last) a comparison of two floating-point values can be done with fixed-point hardware. Biased exponents solve this by moving the negative exponent values below the *exponent bias* so that comparisons become easier. For instance the number 1 normally has an exponent of 0 but because of the biasing, the stored exponent is 127 for single precision floating-point format (see Table 3.1) and 0.5 has a (biased) exponent of 126.

The chosen format for this thesis is the single precision (32-bit) format. In Table 3.2 a number of example values are shown. Note that denormalized (as previously mentioned) numbers do not have an implicit 1. A further requirement for
denormalized numbers are that the (biased) exponent is 0 (-127 unbiased exponent). Also note that when the (biased) exponent is 255, the implicit 1 does not matter (denoted by the x’s in Table 3.2). This is because when the (biased) exponent is 255, the number is either infinity or NaN (not a number, see Section 3.3) and these numbers are never used for any actual calculations so their implicit 1 never matters. Also note that zero can be represented both as positive and negative, depending on the sign bit.

<table>
<thead>
<tr>
<th>Number</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>0111 1111 (127)</td>
<td>(1)000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>Zero</td>
<td>0000 0000 (0)</td>
<td>(0)000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>Denormalized value</td>
<td>0000 0000 (0)</td>
<td>(0)000 0000 1000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>Max normalized value</td>
<td>1111 1110 (254)</td>
<td>(1)111 1111 1111 1111 1111 1111 1111 1111 1111</td>
</tr>
<tr>
<td>Min normalized value</td>
<td>0000 0001 (1)</td>
<td>(1)000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>Inf</td>
<td>1111 1111 (255)</td>
<td>(x)000 0000 0000 0000 0000 0000 0000 0000 0000</td>
</tr>
<tr>
<td>NaN</td>
<td>1111 1111 (255)</td>
<td>(x)000 0000 0000 1100 0000 0000 0000 0000 0000</td>
</tr>
</tbody>
</table>

Table 3.2: A few examples of single precision (32-bit) format values (sign bit omitted).

Any future references to exponents will mean the biased exponent unless explicitly noted otherwise!

The floating-point representation presented in this section has its downsides too, it has to do with the range and precision. Unlike fixed-point notation that has a constant precision (the distance between two consecutive numbers) in all of its range, this kind of floating-point representation does not. The best precision is for numbers closest to the value 0, with exponent 1, but as the exponent becomes larger, precision is successively lost. This is illustrated in Table 3.3, where it can be seen that the loss of precision quickly escalates when the exponent gets larger and larger.

3.2 Rounding

Many of the floating-point operations that are in focus in this thesis, can produce results that might need many more bits to represent exactly then is available for the format. Since the focus of this thesis is single precision floating-point numbers, the maximum available accuracy in the mantissa is 24 bits (including the implicit 1). Because of this limitation a rounding scheme is needed to reach a final result as close as possible to the exact result.

For a satisfactory rounding scheme, the following demands have to be satisfied [2] (x,y below are numbers of infinite precision):

1. If \( x \leq y \) then \( \text{rnd}(x) \leq \text{rnd}(y) \).
<table>
<thead>
<tr>
<th>Exponent</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.4012985 · 10^{−45}</td>
</tr>
<tr>
<td>54</td>
<td>1.2621774 · 10^{−29}</td>
</tr>
<tr>
<td>94</td>
<td>1.3877788 · 10^{−17}</td>
</tr>
<tr>
<td>117</td>
<td>1.1641532 · 10^{−10}</td>
</tr>
<tr>
<td>127</td>
<td>1.1920928955078125 · 10^{−7}</td>
</tr>
<tr>
<td>137</td>
<td>1.2207031 · 10^{−4}</td>
</tr>
<tr>
<td>160</td>
<td>1.024 · 10^{3}</td>
</tr>
<tr>
<td>200</td>
<td>1.1258999 · 10^{15}</td>
</tr>
<tr>
<td>254</td>
<td>2.028241 · 10^{31}</td>
</tr>
</tbody>
</table>

Table 3.3: Precision of different single precision exponent values.

2. If $x$ can be represented exactly in the chosen floating-point representation then $\text{rnd}(x) = x$.

3. If $F_1$ and $F_2$ are two consecutive floating-point numbers (of finite precision) such that $F_1 \leq x \leq F_2$ then $\text{rnd}(x)$ should be either $F_1$ or $F_2$.

The IEEE 754-2008 standard lists four different rounding modes to satisfy these demands. They are the following:

- **Round to nearest even**: The result is rounded to the nearest representable number, in the case of a tie, the result is rounded to the nearest even number. This is the chosen rounding scheme for this thesis because it has a bias of 0 on average [8]. This is also the reason why it is the default rounding scheme in the IEEE 754-2008 standard [1].

- **Round toward $+\infty$**: The result is rounded up towards positive infinity.

- **Round toward $-\infty$**: The result is rounded down towards negative infinity.

- **Round toward zero**: The result is rounded towards zero. Also known as truncation.

This section only serves to inform broadly of the different kinds of rounding schemes and the demands on them to fulfill their tasks. For a more in-depth presentation of round to nearest even, together with its implementation details, refer to Section 4.7.

### 3.3 NaN

NaN stands for not a number and is a special type of value, dedicated for undefined or unrepresentable numbers (For example the result of $\frac{\infty}{\infty}$). In the single
precision format, a NaN is stored in the following manner:

\[ s \ 11111111 \ axxxxxxxxxxxxxxxxxxxxxx \]

Where \( a \) determines which kind of NaN it is and the \( x \):s are an extra payload used for debug information. The sign bit \( s \) is most often ignored in applications.

There are two kinds of NaN in the IEEE 754-2008 standard, quiet NaN \((a = 1)\) and signaling NaN \((a = 0 \text{ and } x \neq 0)\). The difference between them is that floating-point operations that encounter a quiet NaN (qNaN) normally propagates it whereas when a signaling NaN (sNaN) is encountered, an invalid operation exception (see Section 3.4) is supposed to be signaled and if the operation produces a floating-point result it is supposed to produce a qNaN as its output.

In the scope of this thesis, there will be no difference made between the two kinds of NaN. Both will be treated as qNaN, namely that most operations will propagate it and none of the operations will raise any exception flags.

### 3.4 Exceptions

The IEEE 754-2008 standard defines five different kind of exceptions [1]. Normally when these exceptions occur, a status flag is set and the computation continues. Additionally a trap handler can be implemented, that is called when an exception occurs. The five exceptions are the following:

- **Invalid operation**: This exception is signaled when a NaN is produced.
- **Division by zero**: This exception is signaled when division by zero is attempted.
- **Overflow**: This exception is signaled when the rounded result exceeds the largest finite number.
- **Underflow**: This exception is signaled when the rounded result is too small to be represented.
- **Inexact**: This exception is signaled when the infinite result is different from final represented number. So basically when rounding or overflow has occurred.

In the scope of this thesis, there are no status flags set anywhere that the programmer can manipulate in any way. The exceptions are detected and signaled internally (because they’re needed to get the right result in many cases) during the lifetime of the computation, but they’re not stored in any way after a computation is done. This is another break with the IEEE 754-2008 standard which requires the existence of some sort of status flags.
This chapter will cover the theoretical background for each operation together with their chosen implementation. A section of this chapter is also dedicated to the chosen rounding scheme (round to nearest even, see Section 4.7). This chapter will not attempt to reach a unified hardware solution being capable of doing all operations. This is reserved for Chapter 5.

The chapter is divided into a number of sections, they each cover the following operations:

- **Addition**: This section explains how the addition, subtraction and absolute difference operations are implemented.
- **Multiplication**: This section explains how multiplication is done.
- **Division**: This section explains how division and reciprocal is implemented.
- **Compare operations**: This section explains how the minimum, maximum, minimum magnitude and maximum magnitude operations are implemented.
- **Conversion operations**: This section explains how the operations for conversion from *word format* to *floating-point format* (and vice versa) are implemented.
- **Sign operations**: This section will cover how the absolute and negate operations are implemented.
- **Rounding**: This section goes over the details on how the chosen rounding scheme (round to nearest even) is implemented.
4.1 Addition

The algorithm and implementation of FPADD, FPSUB and FPABSD operations will be the subject of this section. Addition and subtraction are intimately linked when it comes to algorithm and implementation. This is because depending on the operand signs and intended operation, the actual (effective) operation might change. As the following example will illustrate, when addition becomes subtraction:

\[ 5 + (-3) = 5 - 3 \]

The next section will go through the basic algorithm to perform an addition or subtraction. The operation of FP absolute difference is trivial once addition and subtraction is done. All that is required is to set the sign to zero (positive sign), so this operation will not be mentioned further in this section. For more information on sign operations, refer to Section 4.6.

There are a couple of special operand combinations that need to be detected and dealt with, since normal calculations would not give the right answer. They are shown below (\(x\) is a floating-point number):

1. \(+\infty - +\infty = qNaN\)
2. \(+\infty + -\infty = qNaN\)
3. \(-\infty + +\infty = qNaN\)
4. \(-\infty - -\infty = qNaN\)
5. \(\pm x \pm NaN_b = NaN\)
6. \(NaN_a \pm +\infty = NaN_a\)
7. \(NaN_a \pm NaN_b = NaN_x\)

According to the IEEE 754-2008 standard [1] when an operand is a NaN, that operand needs to be propagated (as shown in case 5 and 6 above). When both operands are NaN (case 7 above) it is not specified which operand is to be propagated. It is up to the designer.

4.1.1 Algorithm

The algorithm in this subsection is given in generic terms and will only be about the calculation aspects. IEEE considerations, like handling NaN and infinities, will be mentioned in the next subsection (4.1.2).

Let \(x, y\) and \(z\) be three floating-point numbers, fulfilling the operation \(z = x \pm y\). The operands consist of \((S_x, E_x, M_x)\) and \((S_y, E_y, M_y)\) for sign, exponent and mantissa (also called significand). The result will have the corresponding \((S_z, E_z, M_z)\) and any intermediate results will be denoted by \((S^{*}_z, E^{*}_z, M^{*}_z)\). Some boolean operations used: "\(\land\)" means logical and, "\(\lor\)" means logical or, \(\oplus\) is XOR and lines above signals means a negation of that signal.

The general algorithm consists of the following steps (adapted from [2]):

1. Add/subtract the mantissas and set the exponent (also detect special operand
combinations):

\[
(S_z =) S_z^* = (S_x \land g\ell) \lor ((op \oplus S_y) \land (S_x \lor (\bar{g}\ell \land \bar{eq})))
\]

\[
E_z^* = \max(E_x, E_y)
\]

\[
M_z^* = \begin{cases} 
(M_x \pm (M_y \cdot 2^{(E_y-E_x)})) & \text{if } E_x \geq E_y \\
((M_x \cdot 2^{(E_x-E_y)}) \pm M_y) & \text{if } E_x < E_y
\end{cases}
\]

Equation 4.3 shows that the smaller mantissa is shifted right by the difference between the exponents. This is called alignment or pre-normalization. It is also in this stage that the sign of the result is determined (Equation 4.1). The \textit{op} signal is 0 for addition and 1 for subtraction. The sign is determined directly, meaning that the intermediate sign is the same as the final sign. The \textit{gt} signal signals 1 if \(x\) is larger than \(y\) otherwise 0. The \textit{eq} signal signals 1 if \(x = y\) otherwise 0.

2. Normalization: The result of step 1 might not be normalized in which case it needs to be normalized and the exponent needs to be adjusted. Three situations can follow:

(a) The result is already normalized: no action is required.

(b) When the effective operation (EOP) is addition, there might be an overflow of the result. This requires the following actions:

- Shift the mantissa right by one \((M_z^* = M_z^* \gg 1)\).
- Increment the exponent by one \((E_z^* = E_z^* + 1)\).

(c) When the effective operation is subtraction, there might be a variable amount of leading zeros \((lz)\) in the result. This requires the following actions:

- Shift left the mantissa by the same amount as the number of leading zeros \((M_z^* = M_z^* \ll lz)\).
- Decrement the exponent by the same amount \((E_z^* = E_z^* - lz)\).

3. Perform rounding of the normalized result (for details see Section 4.7). The rounding might again produce an unnormalized result, in which case it is needed to normalize the result again according to Option B above.

4. Determine special values like exponent overflow, exponent underflow or zero result value. If anything out of bounds is detected or any special operand combinations were previously detected then the result is replaced with something appropriate like \(qNaN\) or \(\infty\) for example.

Finally, \(S_z = S_z^*, E_z = E_z^*, M_z = M_z^*\).
4.1.2 Implementation

The implementation in this subsection is based on the algorithm explained in Section 4.1.1 with the special cases in mind as well. A figure of the chosen implementation can be seen in Figure 4.1. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

\[ (Ex, Mx) \quad (Ey, My) \]
\[ MUX \]
\[ Ex \quad Ey \]
\[ SWAP \]
\[ Expo \]
\[ Ex \quad Ey \quad gt \]
\[ Subtract \]
\[ SWAP \]
\[ Mantissa \]
\[ Mx \quad My \quad gt \]
\[ Shift \]
\[ Adder \]
\[ Sx \quad Sy \quad op \]
\[ EOP \]
\[ Adjust \]
\[ Normalize \]
\[ Round \]
\[ Normalize 2 \]
\[ Special \]
\[ Finalize \]
\[ (Sx, Ez, Mx) \]

**Figure 4.1:** Implementation of floating-point addition and subtraction.

The operation is divided into the following steps:

1. The first step is to determine which of the operands is the largest (the \( gt \) signal) or if the operands are equal (the \( eq \) signal). These values are very important since they are used in many other places (Figure 4.1).

2. In this step, the sign is determined by using the \( S_x, S_y, op, gt \) and \( eq \) signals. The mantissa of the largest number and largest exponent are also determined for the two SWAP boxes. The difference between exponents are...
also calculated, in the "Subtract" box. The largest exponent will be used as the $S_z$.

3. After the exponent difference is calculated, that value will be used to align the smaller of the two mantissas. The effective operation (EOP) will also be determined in this step.

4. Now the addition or subtraction (depending on the value of EOP) of the two mantissas are done. In this step, all the special cases values can also be determined, that is to check for special operand combinations.

5. After the addition/subtraction, the result might be unnormalized. In this step, the result is normalized and the exponent adjusted according to the algorithm described in step 2 in Section 4.1.1.

6. After normalizing the result, it is time to round the value if needed (rounding will be covered in detail in Section 4.7). The rounding might have produced another unnormalized value (only overflow in this step, see Step 2b in Section 4.1.1). This is rectified with another normalizing step.

7. In the final step it is time to assemble the final floating-point number. It is also in this step that exponent overflow, exponent underflow and zero result value is checked. If the result overflowed it will be forced to ±∞. In the case of underflow or zero result, the result is forced to 0. If any of the special operand combinations occurred, the result is forced to either qNaN or one of the input operands $x$ or $y$.

A significant cost in this implementation is the way the biggest operand is found, the two operands are compared in a 31-bit comparator (the "gt" box in Figure 4.1). Normally, only the exponents are compared but if the exponents are the same and the effective operation is subtraction, a situation can occur where $M_z = M_x - M_y < 0$. This is rectified by negating the sign bit and two’s complementing $M_z$ (see [11] and [12]). In the current datapath this will not be detected until the second pipeline stage, that is why this type of solution was initially discarded to keep the sign logic simpler and at the same pipeline stage for all FP operations.

Another (widely used) way of doing addition/subtraction can be seen in [3] and [13]. These divide the datapath for addition/subtraction in two, called the CLOSE/NEAR path and the FAR path. The different paths calculate based on the difference of exponent, large difference means the FAR path is chosen otherwise the CLOSE path. Both paths generally contain an alignment block, adder and a normalization block. The difference is that the alignment block in the CLOSE path is smaller because few shifts are required, the opposite is true in the FAR path where more shifts are needed. The adder performs a mantissa addition/subtraction in both paths. The normalization is more complex in the CLOSE path because two numbers that are close to each other can render a result with potentially many leading zeros and this needs to be handled. Some CLOSE path designs also has an LZA (leading zero anticipation) unit to speed up this aspect of the CLOSE path. The normalization requires at worst a few shifts in the FAR path. This
4.2 Multiplication

This section will describe the algorithm and implementation of the FPMUL operation. Multiplication for floating-point numbers is in many ways less complex than addition/subtraction (4.1). Things like alignment of the smaller operand, is not necessary for example.

Similarly to addition/subtraction, there are a number of special operand combinations that need to be detected. They are the following (x is a floating-point number):

1. $\pm\infty \cdot \pm 0 = qNaN$
2. $\pm 0 \cdot \pm\infty = qNaN$
3. $\pm x \cdot NaN_b = NaN_b$
4. $NaN_a \cdot \pm x = NaN_a$
5. $NaN_a \cdot NaN_b = NaN_x$
6. $\pm\infty \cdot \pm x = \pm\infty$
7. $\pm x \cdot \pm\infty = \pm\infty$

The propagation of NaNs (cases 3-5 above) works in the same way as for addition/subtraction. That is, if any of the operands are NaNs, that operand gets propagated unaltered. If both operands are NaNs, one of the operands gets propagated unaltered. Which of the operands that gets propagated is not decided by the IEEE 754-2008 standard [1]. It is up to the designer.

4.2.1 Algorithm

The algorithm in this subsection is given in generic terms and will only be about the calculation aspects. IEEE considerations, like handling NaN and infinities, will be mentioned in the next subsection (4.2.2).

Let $x$, $y$ and $z$ be three floating-point numbers, fulfilling the operation $z = x \cdot y$. The operands consist of $(S_x, E_x, M_x)$ and $(S_y, E_y, M_y)$ for sign, exponent and mantissa (also called significand). The result will have the corresponding $(S_z, E_z, M_z)$ and any intermediate results will be denoted by $(S^*_z, E^*_z, M^*_z)$. The XOR operation is denoted by the $\oplus$ sign.

The general algorithm consists of the following steps (adapted from [2]):

1. Multiply the mantissas and set the exponent (also detect special operand combinations):

\[
(S_z =) \quad S^*_z = S_x \oplus S_y \quad (4.4)
\]

\[
E^*_z = E_x + E_y - bias \quad (4.5)
\]
\[ M_z^* = M_x \cdot M_y \] 

(4.6)

The subtraction of the bias value (see Chapter 3) in Equation 4.5 is done because otherwise the bias value would be counted twice in the intermediate exponent, since the bias value is in both \( E_x \) and \( E_y \) from the beginning.

The sign is determined by a simple XOR operation between \( S_x \) and \( S_y \). This XOR operation will determine the sign directly, meaning that the intermediate sign is the same as the final sign (see Equation 4.4).

2. Normalization: The result of step 1 might be unnormalized, in which case it needs to be normalized and the exponent needs to be adjusted. Two situations can occur:

   (a) The result is already normalized: no action is required.

   (b) Because both the operands are in the range \([1,2)\) the result is in the range \([1,4)\). Depending on the result, the following actions might be required:

      - Shift the mantissa right by one \((M_z^* = M_z^* \gg 1)\).
      - Increment the exponent by one \((E_z^* = E_z^* + 1)\).

If denormal numbers were supported and one (or both) of the operands were denormal then there might be leading zeros (as in the case of addition) in the result.

3. Perform rounding of the normalized result (for details see Section 4.7). The rounding might again produce an unnormalized result, in which case it is needed to normalize the result again according to Option B above.

4. Determine special values like exponent overflow, exponent underflow or zero result value. If anything out of bounds is detected or any special operand combinations were previously detected then the result is replaced with something appropriate like \( qNaN \) or \( \infty \) for example.

Finally, \( S_z = S_z^* \), \( E_z = E_z^* \), \( M_z = M_z^* \).

### 4.2.2 Implementation

The implementation in this subsection is based on the algorithm explained in Section 4.2.1 with the special cases in mind as well. A figure of the chosen implementation can be seen in Figure 4.2. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

The operation is divided into the following steps:

1. In the first step the initial addition of the two exponents is done. The sign is determined and the multiplication between the two mantissas is done (see Section 4.2.3). It is also in this step that the various special operand combinations are detected.
2. In the second step the second part of the exponent calculation is done, to subtract the bias from the sum of the two exponents, so that the bias is not counted twice.

3. After the multiplication of the mantissas, the result might be unnormalized. In this step, the result is normalized and the exponent adjusted according to the algorithm described in step 2 in Section 4.2.1.

4. After normalizing the result, it is time to round the value if needed. The "Sticky" box is also related to rounding (rounding will be covered in detail in Section 4.7). The rounding might have produced another unnormalized value (only overflow in this step, see Step 2b in Section 4.2.1). This is rectified with another normalizing step.

5. In the final step it is time to assemble the final floating-point number. It
is also in this step that exponent overflow, exponent underflow and zero result value is checked. If the result overflowed the result will be forced to \( \infty \). In the case of underflow or zero result, the result is forced to 0. If any of the special operand combinations occurred, the result is forced to either \( qNaN \) or one of the input operands \( x \) or \( y \).

### 4.2.3 VPE adaptation

The operation requires the multiplication of the two 24-bit mantissas (see Figure 4.2) but the datapath only contains 16-bit multipliers, so the multiplication of the mantissas is done as illustrated by the following example.

Let \( H1 \) and \( H2 \) be bits 23-8 from operand 1 and 2. Similarly let \( L1 \) and \( L2 \) be bits 7-0 from operand 1 and 2 shifted to the left by 8 steps, so that \( L1 = [L1, 8'0] \) and \( L2 = [L2, 8'0] \). These values are then fed to the multipliers as shown in Figure 4.3, which shows how multiplication of the mantissas progresses through the VPE datapath for one floating-point lane.

As can be seen in Figure 4.3, after the MUL stage multiplications, there are three additional additions. Two of the additions performed are called \( \text{ADDSHIFT-B} \), the other is called \( \text{ADDSHIFT-A} \). The addshifts performs the following operation inside the adder blocks:

\[
\text{ADDSHIFT} - A : \quad (A \gg 16) + B \\
\text{ADDSHIFT} - B : \quad A + (B \gg 16)
\]
After the second level of adders (The adder block with the final ADDSHIFT-B operation in Figure 4.3) the sticky bit is added on the right position (see Section 4.7 for more information on Rounding). After the sticky block, the multiplication of the mantissas is finished and the final adder block is simply bypassed.

### 4.3 Division

This section will describe the algorithm and implementation of the FPDIV and FPRECI operations. Division for floating-point numbers is very similar to multiplication (4.2), as many of the components from multiplication can be reused (see Section 4.3.1 and 4.3.2). The difference between FPDIV and FPRECI in the scope of this thesis is that FPRECI has less precision (thus being faster) and that the numerator (dividend) is statically set to 1. FPDIV has the same precision as 32-bit single precision floating-point format [1], whereas FPRECI will only support 16-bits precision (compared to 24-bits for FPDIV) in the mantissa and FPRECI will not support rounding either.

There are a number of special operand combinations that need to be detected. They are the following \((x)\) is a floating-point number):

1. \[
\frac{NaN_a}{\pm x} = NaN_a
\]
2. \[
\frac{\pm x}{NaN_b} = NaN_b
\]
3. \[
\frac{NaN_a}{NaN_b} = NaN_x
\]
4. \[
\frac{\pm \infty}{\pm \infty} = qNaN
\]
5. \[
\frac{\pm 0}{\pm 0} = qNaN
\]
6. \[
\frac{\pm \infty}{\pm x} = \pm \infty
\]
7. \[
\frac{\pm x}{\pm \infty} = \pm 0
\]
8. \[
\frac{\pm 0}{\pm x} = \pm 0
\]
9. \[
\frac{\pm x}{\pm 0} = \pm \infty
\]

Cases 1-3 shows the NaN propagation rules, they work same way as for add/sub/mul. If any single operand is a NaN, that operand gets propagated unaltered. If both operands are NaN, one of the operands gets propagated unaltered. Which of the operands that is propagated is up to the designer of the hardware.
4.3 Division

4.3.1 Algorithm

The algorithm in this subsection is given in generic terms and will only be about the calculation aspects. IEEE considerations, like handling NaN and infinities, will be mentioned in the next subsection (4.2.2).

Let $x$, $y$ and $z$ be three floating-point numbers, fulfilling the operation $z = \frac{x}{y}$ (In the case of the reciprocal operation, $x$ is forced to 1). The operands consist of $(S_x, E_x, M_x)$ and $(S_y, E_y, M_y)$ for sign, exponent and mantissa (also called significand). The result will have the corresponding $(S_z, E_z, M_z)$ and any intermediate results will be denoted by $(S_z^*, E_z^*, M_z^*)$. Some boolean operations used: "$\wedge$" means logical and, "$\lor$" means logical or, $\oplus$ is XOR and lines above signals means a negation of that signal.

The general algorithm consists of the following steps (adapted from [2]):

1. Divide the mantissas and set the exponent (also detect special operand combinations):

   \[ S_z = S_x \oplus S_y \]  
   \[ E_z^* = E_x - E_y + \text{bias} \] \hspace{1cm} (4.7)  
   \[ M_z^* = \frac{M_x}{M_y} \] \hspace{1cm} (4.8)  

   The addition of the bias value (see Chapter 3) in Equation 4.8 is done because otherwise the bias value would be counted twice in the intermediate exponent, since the bias value is in both $E_x$ and $E_y$ from the beginning.

   The sign is determined by a simple XOR operation between $S_x$ and $S_y$. This XOR operation will determine the sign directly, meaning that the intermediate sign is the same as the final sign (see Equation 4.7).

2. Normalization: The result of step 1 might be unnormalized, in which case it needs to be normalized and the exponent needs to be adjusted. Two situations can follow:

   (a) The result is already normalized: no action is required.

   (b) There might be an underflow of the result. This requires the following actions:
       - Shift the mantissa left by one ($M_z^* = M_z^* << 1$).
       - Decrement the exponent by one ($E_z^* = E_z^* - 1$).

3. Perform rounding of the normalized result (for details see Section 4.7). The rounding might again produce an unnormalized result, an overflow. This is rectified by the following actions:
   - Shift the mantissa right by one ($M_z^* = M_z^* >> 1$).
   - Increment the exponent by one ($E_z^* = E_z^* + 1$).
4. Determine special values like exponent overflow, exponent underflow or zero result value. If anything out of bounds is detected or any special operand combinations were previously detected then the result is replaced with something appropriate like qNaN or \( \infty \) for example.

Finally, \( S_z = S_z^* \), \( E_z = E_z^* \), \( M_z = M_z^* \).

### 4.3.2 Implementation

The implementation in this subsection is based upon the algorithm explained in Section 4.3.1 with the special cases in mind as well. A figure of the chosen implementation can be seen in Figure 4.4. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

![Diagram of floating-point division](image)

**Figure 4.4: Implementation of floating-point division.**

The operation is divided into the following steps:
1. In the first step the initial subtraction is done between the two exponents. The sign is determined and the division of the two mantissas is done. The chosen method for division is called restoring division and will be further explained in Section 4.3.3. It is also in this step that the various special operand combinations are detected.

2. In the second step the second part of the exponent calculation is done, to add the bias from the subtraction of the two exponents, so that the bias is not counted twice.

3. After the division of the mantissas, the result might be unnormalized. In this step, the result is normalized and the exponent adjusted according to the algorithm described in step 2 in Section 4.3.1.

4. After normalizing the result, it is time to round the value if needed (rounding will be covered in detail in Section 4.7). The rounding might have produced another unnormalized value (only overflow in this step, see Step 3 in Section 4.3.1), this is rectified with another normalizing step.

5. In the final step it is time to assemble the final floating-point number. It is also in this step that exponent overflow, exponent underflow and zero result value is checked. If the result overflowed the result will be forced to $\infty$. In the case of underflow or zero result, the result is forced to 0. If any of the special operand combinations occurred, it is here that the result is forced to either qNaN or one of the input operands $x$ or $y$.

### 4.3.3 Restoring division

The method used for dividing the two mantissas is a method called restoring division. Restoring division produces one bit of the answer each cycle of the algorithm and the algorithm is the following:

Let $x$, $y$ and $z$ be floating-point mantissas fulfilling the operation $z = \frac{x}{y}$.

```
loop
    temp = x - y
    if(temp < 0):
        z = z \& \bar{1} \ (\text{add zero to LSB of } z)
    else:
        z = z \lor 1 \ (\text{add one to LSB of } z)
        x = temp
    rmndr = x
    x = x \cdot 2 \ (\text{Try a larger } x \text{ value})
    z = z << 1 \ (\text{Shift to make room for next bit})
endloop
```

This loop can be run as many times as is needed to achieve a desired precision. The `rmndr` (remainder) variable is used in the calculation of the sticky bit (see...
4.4 Compare operations

The algorithm and implementation of FPMIN, FPMAX, FPMINM and FPMAXM operations will be the subject of this section. These compare operations turn out to be relatively simple to implement. This is because the floating-point format (with sign, followed by exponent and then mantissa) allows comparisons to be done by interpreting the floating-point values as signed magnitude. In the case of FPMINM and FPMAXM, things become a little more complicated. This is because when both numbers have the same magnitude ($|x| = |y|$) the IEEE 754-2008 standard states that the returned value should consider signs as well. For example:

\[
\text{fpminm}(-2, +2) = -2 \\
\text{fpmaxm}(-2, +2) = +2
\]

There are a couple of special cases to consider for these operations as well. They are the following (\(op\) can be either \(fpmin\), \(fpmax\), \(fpminm\) and \(fpmaxm\)):

1. \(op(NaN_a, x) = x\)
2. \(op(x, NaN_b) = x\)
3. \(op(NaN_a, NaN_b) = NaN_x\)

Unlike the other operations, NaN operands are not propagated. The only time these operations produce a NaN is when both operands are NaNs. Also infinities do not need to be handled specially because of the IEEE 754-2008 floating-point format. Infinite operands have the largest exponent and as such they will be considered the largest operand automatically without any special handling.

Since these operations are not so complex, the algorithm and implementation are combined into a single section.

4.4.1 Min/Max algorithm and implementation

Let \(x, y\) and \(z\) be three floating-point numbers, fulfilling the operation \(z = fpmin(x, y)\) or \(z = fpmax(x, y)\). They consist of \((S_x, E_x, M_x)\), \((S_y, E_y, M_y)\) and \((S_y, E_y, M_y)\) for sign, exponent and mantissa (also called significand). Some boolean operations used: "\&" means logical and, "\|" means logical or and lines above signals means a negation of that signal. A figure of the chosen implementation can be seen in Figure 4.5. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

The general algorithm consists of the following steps:

1. Perform a normal two's complement subtraction (note that all of \(x\) and \(y\)
Figure 4.5: Implementation of floating-point minimum and maximum.

with sign, exponent and mantissa is used as a two’s complement number):

\[ temp = x - y \]

Also detect special cases in this step.

2. Determine and signal if there was wrap-around (33rd bit of temp is 1). Wrap-around means that \( y \) is larger than \( x \).

3. Use this signal \( wr \) for wrap-around, together with the operand signs to determine which operand is smaller:

\[ smaller = (S_x \land \overline{wr}) \lor (S_y \land wr) \]

When the smaller signal equals 0, operand \( x \) is smallest. When smaller equals 1, operand \( y \) is smallest.

For the FP MAX operation, the smaller signal is inverted to determine which operand is largest instead.
4. Output the correct operand. The result of the smaller signal might be over-ridden if any of the special cases scenarios occurred that were detected in step 1 (refer to Section 4.4).

### 4.4.2 Magnitude algorithm and implementation

Let $x, y$ and $z$ be three floating-point numbers, fulfilling the operation $z = \text{fpmin}(x, y)$ or $z = \text{fpmax}(x, y)$. They consist of $(S_x, E_x, M_x)$, $(S_y, E_y, M_y)$ and $(S_z, E_z, M_z)$ for sign, exponent and mantissa (also called significand). Some boolean operations used: "∧" means logical and, "∨" means logical or and lines above signals means a negation of that signal. A figure of the chosen implementation can be seen in Figure 4.6. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

![Figure 4.6: Implementation of floating-point minimum and maximum magnitude.](image)

The general algorithm consists of the following steps:
1. Perform a normal two's complement subtraction, without the sign bits:
   \[ \text{temp} = x[30:0] - y[30:0] \]
   Also detect special cases in this step.

2. Determine and signal if both operands were equal (\( \text{temp} = 0 \)) or if there was wrap-around (32nd bit of \( \text{temp} \) is 1). Wrap-around means that \( y \) is larger than \( x \).

3. Use these signals (\( \text{eq} \) for equal and \( \text{wr} \) for wrap-around) together with the operand signs to determine which operand is smaller:
   \[ \text{smaller} = (\overline{\text{eq} \land \text{wr}}) \lor (\text{eq} \land S_y) \lor (\text{eq} \land \overline{S_x}) \]
   When the \( \text{smaller} \) signal equals 0, operand \( x \) is smallest. When \( \text{smaller} \) equals 1, operand \( y \) is smallest.

   For the FP MAXM operation, the \( \text{smaller} \) signal is inverted to determine which operand is largest instead.

4. Output the correct operand. The result of the \( \text{smaller} \) signal might be overridden if any of the special cases scenarios occurred that were detected in step 1 (refer to Section 4.4).

### 4.5 Conversion operations

This section will be about the algorithm and implementation of the CVTW2FP and CVTFP2W operations. These operations convert words (16-bit) fixed-point numbers (in \( Q1.15 \) format, which can represent \([-1, 1]) to single precision (32-bit) floating-point format and vice versa.

In the case of CVTW2FP there are no special cases to consider since all the numbers that are representable in 16-bit fixed-point format are representable in single precision floating-point format. The CVTFP2W operation does however need some special value handling. Any negative FP numbers that have an exponent (biased) larger than 127 (meaning numbers that are equal to -2 or more) and any positive number with an exponent equal to 127 (meaning a number that is equal to \([1, 2]) are rounded to the largest negative or positive two's complement numbers. 0111 1111 1111 1111 (0.99...) for positive numbers and 1000 0000 0000 0000 (-1) for negative numbers. No special consideration is taken to NaNs, since fixed-point number formats can not represent NaNs. NaNs are therefore also rounded to largest representable (positive or negative) value.

#### 4.5.1 W to FP algorithm and implementation

Let \( x \) be a fixed-point operand and \( z \) the floating-point result, fulfilling the operation \( z = cvtw2fp(x) \). It consist of \((S_z, E_z, M_z)\) for sign, exponent and mantissa (also called significand). Intermediate exponent and mantissa will be called \( E_z^* \) and \( M_z^* \). A figure of the chosen implementation can be seen in Figure 4.7. The
numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

\[ x[15] \quad 127 \quad x[15] \quad x \]

2-Comp
\[ \text{Shift 8 Left} \]
\[ \text{Adjust} \]
\[ \text{Normalize} \]
\[ \text{Finalize} \]

**Figure 4.7: Implementation of fixed to floating-point conversion.**

The general algorithm consists of the following steps:

1. In the first step, the sign \( S_2 \) is taken from the 16th bit from \( x \) (i.e. \( x[15] \)). The exponent \( E_2^* \) is set to 127. The fixed-point value \( x \) is in a two’s complement encoding, as opposed to the single precision floating-point encoding, which is in a sign magnitude encoding, so when \( x \) is negative, \( x \) needs to be two complemented. Regardless of whether \( x \) was two’s complemented or not, it is shifted left by 8 because the fixed-point format only has a 16-bit precision and this needs to be shifted up to the 24th bit in the floating-point precision value.

2. In the second step, the intermediate result will be normalized to adjust for any leading zeros (\( l_z \)) that might be present in the fixed-point format. The
4.5 Conversion operations

exponent will also be adjusted accordingly:

\[ M^*_z = M^*_z \ll \lz \]

\[ E^*_z = E^*_z - \lz \]

3. In the final step, the final floating-point number is assembled and outputted.

\[ z = \{x[15] = S_z, E_z^*, M_z^*\} \]

4.5.2 FP to W algorithm and implementation

Let \( x \) be a floating-point operand and \( z \) the fixed-point result, fulfilling the operation \( z = \text{cvtfp}_2^w(x) \). They consist of \((S_x, E_x, M_x)\) and \((S_z)\) for sign, exponent and mantissa (also called significand).

A figure of the chosen implementation can be seen in Figure 4.8. The numbers next to various boxes correspond to the numbers in the list of steps below, where they are explained.

\[\begin{align*}
\text{Special} & \quad 127 \quad \text{Ex} \\
\text{Shift} & \quad 1 \\
\text{2-Comp} & \quad 2 \\
\text{Round} & \quad 3 \\
\text{SAT} & \quad 4 \\
\text{x} & \quad 1 \\
\text{Adder} & \quad 1 \\
S_x & \\
M_x & \quad 1 \\
S_z &
\end{align*}\]

Figure 4.8: Implementation of floating to fixed-point conversion.
The general algorithm consists of the following steps:

1. First step is to subtract the exponent from 127 and use that difference to shift $M_x$ by this amount. It is also in this step that overflow values are detected (in the “Special” box in Figure 4.8).

2. In the second step, the mantissa gets two’s complemented if the sign is negative. This is because floating-point format is in sign-magnitude format whereas the fixed-point format uses a two’s complement format.

3. In the third step, the intermediate mantissa is rounded, but only if $x$ is positive. This produces the same result as if rounding was done before the two’s complementation. It had to be implemented in this way because all the adders where two’s complementation can be done are placed before the rounding unit in the datapath. More on rounding for CVTFP2W in Section 4.7.

4. If the rounding caused the intermediate result to overflow, this is detected in the saturation unit and the result is forced to the largest positive or negative value. It is also in the saturation unit that the intermediate result is forced to either the largest positive or negative value if the floating-point number was too big to begin with in step 1.

### 4.6 Sign operations

This small section will be about the FPABS and FPNEG operations. These operations are trivial to implement, because of the IEEE 754-2008 [1] floating-point format. There is a dedicated bit that decides the sign of the number. In the case of FPABS, all that is needed is to clear the sign bit of the operand. In the case of FPNEG, all that is needed is to negate the sign bit of the operand. It is also stated in the IEEE 754-2008 standard that these operations do not consider if the operand is NaN or not, meaning that these operations will treat NaN like any other number. The sign bit in NaN is normally not used for anything anyway.

### 4.7 Rounding

In Section 3.2 the different rounding schemes of IEEE 754-2008 standard [1] were briefly mentioned. They are:

- Round to nearest even
- Round toward $+\infty$
- Round toward $-\infty$
- Round toward zero

Another rounding mode, that is relatively easy to implement, is called Round to nearest up (RNU). Whenever there is a tie, the number gets rounded up. For ex-
ample \( rnu(2.5) = 3 \). The rounding scheme however has a bias (not to be confused with exponent bias) towards larger values. This is particularly bad when there is a long sequence of calculations where each calculation is dependent on the previous calculations result. In order to minimize this problem another rounding scheme is often chosen instead, \textit{round to nearest even} which has an average bias of zero because for half of the numbers the rounding scheme rounds up and for the other half it rounds down. For example \( rne(2.5) = 2 \) and \( rne(3.5) = 4 \).

This is also the reason why it is the default rounding mode in the IEEE 754-2008 standard and the rounding mode of choice for this thesis. \textit{Round to nearest odd} (which rounds to nearest odd in the tie case) also has a bias of zero, but RNE is preferred because it leads to less error when the result is divided by 2, which is a common computation \cite{2}.

Different operations have slightly different requirements to fulfill the demands that were postulated in Section 3.2, but for simplicity, all operations implemented use the exact same method to implement RNE. The only operation that does not use RNE is the CVTFP2W instruction which uses Round to nearest up (RNU).

### 4.7.1 Algorithm and implementation

In order to represent a floating-point number exactly, infinite precision is required but for the single precision floating-point format only 24 mantissa bits are available for the final result, but not all bits produced as an intermediate result are needed to implement proper rounding to fulfill the requirements postulated in Section 3.2. So how many extra bits are required? It turns out that 2 extra accurate bits and 1 indicator bit is enough to be able to implement the RNE rounding scheme \cite{8}, \cite{2}. The two accurate extra bits are called G (guard) and R (round). The extra indicator bit is called T (sticky). Their positions are the following:

\[ \cdots \text{xxxxxxxxxxxxxLGRT} \]

L is the LSB (24th) bit (counting from MSB) of the intermediate result produced by an operation. G and R are two extra bits that have either been shifted beyond L in some pre-normalization (in the case of addition/subtraction) or have their place there after a finished operation (in the case of multiplication or division). The T bit is an indicator bit. It has a value of 1 if any bit beyond R has ever had a non-zero value. For example:

\[ \cdots \text{xxxxxxxxxxxxxL100000101} \Rightarrow \cdots \text{xxxxxxxxxxxxxL101} \]

Where \( G = 1, R = 0 \) and \( T = 1 \).

When it is time to round for the different operations, a combinatorial box determines whether to round or not. The boolean expression that must be fulfilled to round a number according to RNE is the following ("\( \land \)" means logical and, "\( \lor \)" means logical or):

\[ \text{rnd} = G \land (L \lor R \lor T) \]
When \( rnd \) is 1, a 1 is added in the L position of the number that is to be rounded. Some operation specific details need to be clarified that were not discussed in the sections for the different operations.

**Addition**

In the *pre-normalization* stage for the addition/subtraction operation, the shift unit (that is used to *align* the smaller operand, see Figure 4.1) needs to be modified to able to handle the T (sticky) bit, so that when a non-zero value is shifted beyond the R bit, the T bit is set to 1.

**Multiplication**

There are no *pre-normalization* shifts required for multiplication, however, the multiplication does produce a 48-bit result. The 27th bit needs to be modified to behave as a T bit, this is done in the "Sticky" box in Figure 4.2.

**Division**

The method for the division of the two mantissas is called *restoring division* (refer to Section 4.3.3). This method produces one bit per clock cycle. In order to make RNE work for division the method needs to produce 27 bits which includes the 24 normal bits together with the G, R and T bits. The correct T bit is produced by ("\( \lor \)" means logical or):

\[
T = T \lor \text{remainder}
\]

*(remainder is the remainder, if any, from the *restoring division* algorithm)*.

**CVTFP2W**

The rounding scheme for the CVTFP2W is RNU instead of RNE. RNU requires only one additional bit, the G bit. And a 1 is always added to the G position to get the correctly rounded result.

**Denormal numbers**

Multiplication and division can produce numbers that are temporarily denormalized. They are detected by having an *exponent* = 0 (in the case of division) and the 48th bit (only for multiplication) is also 0. In this case the combinatorial box that decides whether to round or not has the following boolean expression to satisfy ("\( \land \)" means logical and, "\( \lor \)" means logical or):

\[
rnd = (G \ll 1) \land ((L \ll 1) \lor (R \ll 1) \lor (T \ll 1))
\]

This expression basically says that the L, G, R and T bits are taken one step to the left of where they are normally taken. The 1 that is normally added to the L position, is now added to the \((L \ll 1)\) position.
This chapter will present a unified hardware design that will be able to handle all the operations presented previously. The datapath is separated into three distinct stages (the MUL stage, the Adder stage and the MAC stage) and this chapter is divided into sections according to them, with each section going over the changes made to that stage in order to handle floating-point operations.

The width of the datapath can support computation on four different values simultaneously. Each computation is done in something that is called a lane and each lane has access to its own set of hardware in the different datapath stages. All the presentations, including figures, in this chapter will be about one lane (because all lanes are identical) unless explicitly written otherwise.

The sections will not present an exact hardware setup, but a rather general overview of the most important hardware blocks.

5.1 The MUL stage

An overview of the MUL pipeline stage can be seen in Figure 5.1. The figure shows one of the four lanes capable of handling floating-point operations in the MUL stage. The Preformatting and Output Formatting blocks are common for all four lanes.

Here is an overview of the most significant changes done for FP adaptation:

- The preformatting block has been altered to be able to handle multiplication properly. There is detection for if any of the two operands are zero or NaN. This affects whether the hidden one gets added (to the 24th bit) or not, if any of the operands are zero or NaN the hidden one does not get added
Figure 5.1: Overview of the MUL pipeline stage.

to the mantissa of that number.

- The FP registers block contains registers for important signals and values that are generated in the FP operations block. The sign, initial exponent, effective operation (EOP) for addition/subtraction and which (if any) special operand combination has occurred. These signals need to be saved for components further down the datapath.

- The FP operation block was added to handle all pre-calculation aspects of floating-point numbers (with the exception of multiplication). This block is responsible for calculating the sign and the initial exponent for all floating-point operations. It also handles detection of any of the special operand combinations for the various FP operations. Some mantissa manipulation is also done in this block: the pre-alignment of the smaller mantissa in the case of addition/subtraction. The FP operations block will be presented in more detail.

An overview of the FP operations block can be seen in Figure 5.2. The "Shift" block is used for the pre-alignment of the smaller mantissa and is only used for the FPADD/FPSUB/FPABSD operations.
5.1 The MUL stage

The MUL stage includes the MUL pipeline stage, which processes floating-point operations. The SP block contains the detection of the different special operand combinations. The signal coming out from the SP block (also called SP in the future) is capable of setting the sign, exponent and mantissas to particular values depending on the operation and special operand combination detected.

Figure 5.2: Overview of the FP operations block of the MUL pipeline stage.

Figure 5.3: All the different options for the sign.
The sign, expo and mantissa blocks in Figure 5.2 contain all the different options that the floating-point sign, exponent and mantissa can have. The sign block, which can be seen in Figure 5.3, produces a final result, meaning it will not have to be changed again during the course of the calculation. The Comb1 block in the figure determines the sign for the addition/subtraction operation. The signmode signal determines which alternative is chosen as the sign, but if the SP block has detected some special operand combination the normal signmode no longer applies. This is done in the Comb2 block.

The exponent block produces an initial exponent value that might be changed later depending on the normalizations required. Its contents can be seen in Figure 5.4. It contains two adders in series with all the options to be able to handle the different operations.

![Diagram](image.png)

*Figure 5.4: Contents of the Exponent block.*

For division the exponent was calculated as $E_x - E_y + \text{bias}$. In the case of the FPRECI operation, this has been simplified to $254 - E_y$ because $E_x$ is a static value (127) and $\text{bias}$ is also a static value (127). So they were combined into one
value, $127 + 127 = 254$ and placed in the $A\ MUX$. The "126" and "10" values in the $B2\ MUX$ are there because values in the different stages are left adjusted. How this affects some operations will be covered in more detail in Section 5.4.

The mantissa block can be seen in Figure 5.5. The "0x800000" value in the $L\ MUX$, is the value used as the numerator (dividend) in the FPRECI operation.

![Figure 5.5: Contents of the Mantissa block.](image)

5.2 The Adder stage

The adder stage of the datapath is only slightly modified (see Figure 2.4 in Section 2.3.2). The only modifications are the addition of some extra operations in the ALU blocks. And the addition of a "Sticky" block, after the second level of adders, that is used in the FPMUL operation (see Section 4.2.3 for more information).

The operations that have been added in the ALU blocks are the following:

- **fpadder**: This ALU operation is used for the FPADD/FPSUB/FPABSD operations and it performs an addition/subtraction of the mantissas, depending on the value of the $EOP$ signal (see Section 4.1 for more information). This ALU operation is done in the first level of adders (for FPADD, FPSUB and FPABSD) and is not available in any other ALU block.

- **fpabs**: This ALU operation is used for the CVTFP2W operation and performs an absolute value of the incoming floating-point mantissa depending on the sign of that floating-point number. This ALU operation is done in the second level of adders (for CVTFP2W) and is not available in any other ALU block.

- **fpdiv**: This ALU operation is used for the FPDIV/FPRECI operations and it performs the subtractions of the mantissas in the restoring division algorithm (see Section 4.3.3 for more information). This ALU operation is done in the ALU block in the MAC pipeline stage (for FPDIV and FPRECI) and is not available in any other ALU block.
• The FPMIN, FMAX, FMINM and FMAXM operations are handled entirely inside the ALU blocks (and nowhere else) according to the algorithm described in Section 4.4. These ALU operations are done normally in the first level of adders, but are available in all ALU blocks.

5.3 The MAC stage

An overview of the MAC pipeline stage can be seen in Figure 5.6. The figure shows one of the four lanes capable of handling floating-point operations in the MAC stage. The Data Switches and Output formatting blocks are common for all four lanes.

Figure 5.6: Overview of the MAC pipeline stage.

Here is an overview of the most significant changes:

• The FP registers block contains the same registers as were already explained in Section 5.1.
• The ACCROP block contains the accumulator hardware. An overview of the accumulator can be seen in Figure 5.7. The accumulator register (ACR) contains accumulated values from various other operations (outside the scope of this thesis) and it also contains the restoring division remainder (see Section 4.3.3 for more information) during the FPDIV and FPRECI operations.

![Figure 5.7: Overview of the accumulator hardware.](image)

The help register (HR) is only used by the restoring division algorithm and contains the answer of the division. Each clock cycle a subtraction takes place in the adder (Figure 5.7). If there was a wrap-around (meaning that $A - B < 0$) or not is signaled to the "Comb" block which generates the 1 or 0 depending on the result from the adder block. For more information on restoring division, refer to Section 4.3.3.

The "S & S" and "<< 6" options at the MUX 1 block are also for the division operations. The "S & S" block is used in the FPDIV operation and takes the value in the ACR register, generates a sticky from it and puts it in the correct
position on the value from the HR register. This result is then shifted left by five because the answer needs to be *left adjusted* (more on this in Section 5.4). The "<< 6" block is used in the FPRECI operation and it only shifts the result in the HR register left by six, also to make sure the answer is *left adjusted*.

- The "FF1" block tells the "Scaler" block how much the mantissa needs to be shifted, either left or right. The value is in a two's complement format. Three cases can occur:
  - If there is overflow of the mantissa, a negative value is returned. Since none of the operations (in the scope of this thesis) can produce a result that has more than one bit of overflow, this value can only be "−1" in practice.
  - If there is underflow of the mantissa, a positive value is returned. This value is variable since a variable amount of leading zeros can occur in the FPADD, FPSUB and FPABSD operations.
  - The mantissa is already normalized, then "0" is returned from the FF1 block.

- The "Scaler" block does the first normalization step in all operations that require normalization. It is a variable step shifter that can shift both to the left and to the right. In the scope of this thesis it takes the amount of steps to shift from the FF1 block, but can it can take this from other sources for other operations.

- The "Rounder" block performs the rounding according to the method described in Section 4.7. If any of the operands in the MUL stage were detected as NaN and need to be propagated, no rounding is done.

- The "SAT" (or Saturation) block is only used for the CVTFP2W operation, in the scope of this thesis. Its purpose is to detect when the "Rounding" block may have caused the result to overflow in the process of rounding (RNU in this operation), and in that case, this block will force the result to the largest positive (0x7fff) value or the largest negative (0x8000) value.

- The *SP* signal going into the "FP2W Special" block, is generated in the MUL Stage (see Section 5.1) and it signals if the original floating-point number was too big (either in positive or negative terms). If it was the result will also be forced to the largest positive or negative value, regardless if the rounding operation overflowed or not.

- The "FP Operation" block, in many ways, works in the same way as the FP operation block in the MUL Stage (see Section 5.1). An overview of the block can be seen in Figure 5.8.

Some of its components:
5.3 The MAC stage

- The "Exponent Calc" block calculates a new exponent for the floating-point number based on normalizations, according to this formula:

\[ E^*_z = E^*_z - FF1_{res} + \text{fpmant}_{25} \]

The negation of \( FF1_{res} \) is because a right shift (increase exponent) means that \( FF1_{res} \) is negative and a left shift (decrease exponent) means that \( FF1_{res} \) is positive. If \( \text{fpmant}_{25} \) (the 25th bit) is set to 1, then that means that the rounding operation overflowed and the result needs to be normalized again. This means that the exponent needs to be increased by one, which is also done in this block.

- The "Special Signals" block determines if the result that has been calculated up to this point is a result within the allowed range. If overflow, underflow or zero result has occurred, it is detected and signaled to the "Exponent MUX" and "Mantissa MUX" which values should be chosen.

In the case of the exponent mux, only three values can occur, the calculated exponent from the "Exponent Calc" block, 0 or 255 (for NaN or \( \infty \)).

- The "Mantissa MUX" block can be seen in Figure 5.9.

The final normalization (if rounding overflow occurred) is done at "MUX 1". If the 25th bit of the mantissa is set to 1, the mantissa chosen is the one that is shifted to the right by one step, otherwise not. The mantissa can take on two other values also, 0 and 0x400000. The 0x400000 value is forced mantissa value in case an special operand combination occurs which has a result of qNaN. The SS signal comes from the "Special Signals" block (see Figure 5.8), this signal together with the calculated exponent from the "Exponent Calc" block determine the exponent value to be used in the "Mantissa MUX" block.

**Figure 5.8:** Overview of the "FP Operation" block of the MAC pipeline stage.
with the SP signal from the FP operation block from the MUL stage, will decide which will be the final mantissa value.

5.4 Left adjustment

All the values are left adjusted in the VPE datapath, by the "Output Formatting" block in Figure 5.1. What it means, in the scope of this thesis, is that the 24 bits of the mantissa, together with the guard, round and sticky bits, are in the upper region of 32-bit registers. For example:

\[ M000000000000000000000000000000LGRT00000 \]

Where \( M \) is the most significant bit of the mantissa, (usually) the 24th bit is in the bit 32 position. \( L \) is the least significant bit, (usually) the 0th bit is now in the 9th bit position. Followed by the Guard, Round and Sticky bits.

5.4.1 Multiplication

Among the options to the "B2 MUX" in Figure 5.4, there is an option that is "126". This is used in the exponent calculation of the multiplication (FPMUL) operation, according to the formula \( E_x + E_y - \text{bias} \). This 126 value corresponds to that \( \text{bias} \) value, which is normally 127.

The reason why it is 126 in the case of multiplication is because of the left adjustment in the datapath. The two 24-bit mantissas generate a 48-bit result, where the 48th bit would indicate an overflow of the result. Because of the left adjustment, the upper 24 bits are left adjusted to 32-bits, meaning the potential overflow will be put in the 32nd bit instead of the normal 33rd bit. In order to rectify this, \(-126\) is used instead (in the exponent formula) to make the exponent bigger than normal by 1, because if there is no overflow from the multiplication, then
the $M$ bit will be in bit position 31 but in the FF1 block this is treated as an underflow and it will be shifted left by one step in the Scaler. This "normalization" step also subtracts 1 from the exponent resulting in $-126 - 1 = -127$, which was the original bias and the $E_x + E_y - 127$ formula is satisfied.

If there was overflow on the other hand, then the exponent needs to be increased by 1, which was already done by using $-126$ in the exponent formula.

This is done to minimize hardware at the output of the adder stage.

### 5.4.2 Reciprocal

Among the options to the "B2 MUX" in Figure 5.4, there is an option that is "10". This is added to the normal exponent formula of FPRECI $(254 - E_y)$ because the precision of the FPRECI operation is 16-bits and this is shifted left six steps, so that the most significant bit will be in bit position 22 (see Section 5.3 on the "ACCROP" block).

This value will be assumed to be left adjusted for 32 bits in the FF1 block (even though it is not), which will detect $32 - 22 = 10$ leading zeros and shift left the mantissa ten steps, which will decrease the exponent by 10. This will cancel out the 10 added in the exponent block in the MUL stage.
This chapter will briefly go over how the design was tested and also with what kind of inputs it was tested with. For most of the operations, test cases were taken from a test generation framework called FPgen. The operations that did not have test cases from FPgen were tested with self made test cases. The same test cases were applied to all three phases of the development of this design.

This chapter will first give some information on FPgen and then go over which operations were covered by FPgen. An overview will be given over the different kinds of test cases used for the operations that were not covered by FPgen. Finally a short section will be about how the tests were made.

6.1 FPgen

FPgen is a test generation framework used to verify floating-point datapaths. It is a generic tool for verification of FPUs that targets architectures that conform to the IEEE 754 standard [1]. The framework has been developed over a number of years by IBM and for detailed information, refer to [5] and [6].

6.1.1 Coverage models

The part of FPgen that is of interest to this thesis is the so called generic test-plan (GTP). It comprises of different coverage models that are meant to target different aspects of the floating-point datapath. They each contain test cases with input operands and expected result values. The coverage models that are of use for this thesis, are the following:

1. **Add: Cancellation**: This model tests every possible value for cancellation, for a 24-bit mantissa. This model contains 49 test cases.
2. **Add: Shift and Special Significands**: This model tests combinations of different shift values between the input operands together with special patterns in the significand of the input operands. This model contains 32,507 test cases.

3. **Add: Shift**: This model tests every possible value for a shift between the input operands. This model contains 112 test cases.

4. **Basic Types: Inputs**: This model checks some regular input operands, for example Infinity, qNaN, MaxNorm. This model contains 4,388 test cases.

5. **Basic Types: Intermediate**: Same as above, but the tests target intermediate results instead. This model contains 90 test cases.

6. **Rounding**: This model tests all combinations of the sign, LSB, guard and sticky bit of the intermediate significand. This model contains 111 test cases.

7. **Corner Rounding**: This model checks for a few corner cases in rounding. This model contains 8 test cases.

8. **Vicinity of Rounding Boundaries**: This model targets numbers that are on the edge of a rounding boundary. This model contains 104 test cases.

9. **Hamming Distance**: This model checks final results that are very close, measured in Hamming Distance, to some specified boundary values. This model contains 85 test cases.

10. **Input Special Significand**: This model checks special patterns in the significands of the input operands. This model contains 1,000 test cases.

11. **Overflow**: This model checks for different kinds of overflow. This model contains 315 test cases.

12. **Underflow**: This model checks for different kinds of underflow. This model contains 119 test cases.

13. **Divide: Divide by Zero**: This model will test the divide by zero exception. (See Section 3.4 for more information). This model contains 28 test cases.

14. **Divide: Trailing Zeros**: This model has various test cases to test the sticky bit in intermediate results. This model contains 20 test cases.

15. **Compare Different Input Field Relations**: This model checks various possible differences between the two input operands. This model contains 317 test cases.

These coverage models contain a grand total of 39,253 test cases that were used to test the solutions of the different phases (see Section 1.3).

The different operations that are covered within these coverage models are the following: FPADD, FPSUB, FPMUL, FPABS, FPNEG, FPMIN, FPMAX, FPDIV, FPMAXM.
The operations that were not covered by FPgen are the following: **FPABSD, CVTW2FP, CVTFP2W, FPRECI, FPMINM.**

### 6.2 Other test cases

Among the operations that have no test cases from FPgen, many of them have large similarities with other, tested, operations. Because of this they are considered to be implicitly covered as well.

FPABSD is basically the same operation as FPSUB, the only difference is that the sign always gets set to 0 (positive value). The difference between FPRECI and FPDIV is that FPRECI has 16-bit precision instead of the full 24-bits and the numerator is a constant 1. In the current implementation of FPMINM and FPMAXM, the only difference is that the signal choosing which is smaller is inverted to determine the largest value.

The entire number space of 16-bit fixed-point numbers can be represented with a single precision floating-point representation, so there are few corner cases to consider. What needs to be tested are that negative two’s complement numbers get properly converted to a sign magnitude format and that the mantissa is properly shifted so that the eventual exponent is right.

In the case of CVTFP2W there are some more corner cases to consider. Negative and positive overflow are tested (numbers outside the $-1 \leq x < 1$ range). Similarly to CVTW2FP, there needs to be a format conversion for negative numbers. This time from sign magnitude format to a two's complement format, this is also tested. Since the entire number space of the floating-point format cannot be represented in the fixed-point format a rounding is performed (round to nearest up). Because of the rounding the answer might overflow, so in this case a saturation scheme is required. Both the saturation and rounding are tested.

When making test cases for CVTW2FP and CVTFP2W, a simple branch coverage testing scheme was performed. Naturally the testing can be done much more stringently but time did not allow for a more advanced approach.

### 6.3 Testing

The different testing activities during the three phases:

- **Phase one:** The test data from FPgen was converted to a more suitable (less complex) format for this thesis. Test cases for the operations that did not have test cases from FPgen were also made. Finally the algorithms that were written in C were tested and passed.

- **Phase two:** In this phase the cycle-true and pipeline accurate simulator was tested by generating assembler programs with the test data in them, to test the different operations. All test cases passed.
• Phase three: In the final phase the RTL code was tested by generating test benches with the test data as inputs. The results were then logged and compared to the correct answers. All test cases passed.

During all three phases, the same test data was used.
To show how this solution affects the existing datapath, in terms of extra area and power consumption, the datapath was synthesized with and without floating-point support. The synthesis tool provides both the area and power usage numbers.

The ASIC synthesis was done with the following settings:

- **Process**: 65nm
- **Voltage**: 1.2V
- **Tool**: Design Compiler
- **Register retiming**: No
- **Max clock fanout**: 16
- **Target frequencies**: 50MHz, 100MHz, 150MHz, 200MHz, 250MHz, 300MHz, 350MHz, 400MHz, 450MHz, 500MHz

The RTL code is designed for register retiming in mind but there was huge difficulties in getting it to work in the synthesis tool, so the RTL code was eventually synthesized without register retiming. This will affect the results, especially when the design starts to reach its peak when it comes to clock frequency.

The initial synthesis results can be seen in Table 7.1. To better illustrate the differences, a comparison between area and power of the floating-point (FP) datapath and the non-FP datapath can be seen in Figure 7.1 and Figure 7.2.
<table>
<thead>
<tr>
<th>Area ($\mu m^2$)</th>
<th>Power (mW)</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>191 405</td>
<td>3.08</td>
<td>50</td>
</tr>
<tr>
<td>210 485</td>
<td>6.69</td>
<td>100</td>
</tr>
<tr>
<td>223 631</td>
<td>11.19</td>
<td>150</td>
</tr>
<tr>
<td>241 371</td>
<td>17.05</td>
<td>200</td>
</tr>
<tr>
<td>265 162</td>
<td>25.39</td>
<td>250</td>
</tr>
<tr>
<td>282 516</td>
<td>31.63</td>
<td>300</td>
</tr>
<tr>
<td>387 287</td>
<td>46.79</td>
<td>350</td>
</tr>
<tr>
<td>456 298</td>
<td>61.43</td>
<td>414</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area ($\mu m^2$)</th>
<th>Power (mW)</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>169 587</td>
<td>2.69</td>
<td>50</td>
</tr>
<tr>
<td>188 469</td>
<td>5.97</td>
<td>100</td>
</tr>
<tr>
<td>193 685</td>
<td>9.35</td>
<td>150</td>
</tr>
<tr>
<td>214 607</td>
<td>14.39</td>
<td>200</td>
</tr>
<tr>
<td>233 589</td>
<td>22.19</td>
<td>250</td>
</tr>
<tr>
<td>245 427</td>
<td>27.11</td>
<td>300</td>
</tr>
<tr>
<td>275 810</td>
<td>39.82</td>
<td>350</td>
</tr>
<tr>
<td>306 076</td>
<td>50.39</td>
<td>400</td>
</tr>
<tr>
<td>389 889</td>
<td>70.73</td>
<td>450</td>
</tr>
</tbody>
</table>

Table 7.1: Initial ASIC synthesis results.

![Area comparison between "FP" and "No FP" datapath.](image)

Figure 7.1: Area comparison between "FP" and "No FP" datapath.
As can be seen in Figure 7.1, the lack of register retiming really starts to affect the results at around 350MHz. Up to 300MHz, the average increase in area was around 13.5% when adding on the floating-point support. The increase in area for 350MHz is around 40% and the increase in area between the max FP frequency of 414MHz compared to the 400MHz of the non-FP datapath is around 49%. The average increase in power consumption is about 17%.

In order to compensate for the lack of register retiming and to make sure that this floating-point design can scale reasonably well, pipeline registers were moved between the second and third level adders. It was moved there because separate synthesis of the different pipeline stages revealed that the adder stage is the bottleneck in the design without register retiming. The synthesis results for this setup can be seen in Table 7.2.

Again, to illustrate the differences better, a comparison between area and power of the floating-point (FP) datapath and the non-FP datapath can be seen in Figure 7.3 and Figure 7.4. As can be seen in Figure 7.3 the FP area follows the behavior of the non-FP datapath better. This is reflected in the numbers as well, with an average increase in area of about 14.7%. The increase in area between the 442MHz FP datapath and the 450MHz non-FP datapath is around 24%. The increase in area between the 473MHz FP datapath (max frequency) and the 500MHz non-FP datapath is around 23%. This sudden increase most likely again has to do with the manual moving of the pipeline registers not being optimal, the synthesis tool with register retiming could do a much better job of identifying bottlenecks and moving around registers as needed. The average increase in power consumption...
<table>
<thead>
<tr>
<th>Area ($\mu m^2$)</th>
<th>Power (mW)</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>188 730</td>
<td>2.98</td>
<td>50</td>
</tr>
<tr>
<td>203 724</td>
<td>6.40</td>
<td>100</td>
</tr>
<tr>
<td>215 361</td>
<td>10.71</td>
<td>150</td>
</tr>
<tr>
<td>228 568</td>
<td>14.56</td>
<td>200</td>
</tr>
<tr>
<td>244 500</td>
<td>22.33</td>
<td>250</td>
</tr>
<tr>
<td>264 439</td>
<td>30.34</td>
<td>300</td>
</tr>
<tr>
<td>297 720</td>
<td>39.66</td>
<td>350</td>
</tr>
<tr>
<td>348 825</td>
<td>52.05</td>
<td>400</td>
</tr>
<tr>
<td>442 674</td>
<td>70.65</td>
<td>442</td>
</tr>
<tr>
<td>452 231</td>
<td>82.33</td>
<td>473</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No FP</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>166 417</td>
<td>2.67</td>
<td>50</td>
</tr>
<tr>
<td>178 962</td>
<td>5.65</td>
<td>100</td>
</tr>
<tr>
<td>185 681</td>
<td>8.96</td>
<td>150</td>
</tr>
<tr>
<td>201 419</td>
<td>13.00</td>
<td>200</td>
</tr>
<tr>
<td>213 018</td>
<td>19.14</td>
<td>250</td>
</tr>
<tr>
<td>225 881</td>
<td>25.46</td>
<td>300</td>
</tr>
<tr>
<td>259 567</td>
<td>36.00</td>
<td>350</td>
</tr>
<tr>
<td>281 671</td>
<td>44.90</td>
<td>400</td>
</tr>
<tr>
<td>359 493</td>
<td>59.39</td>
<td>450</td>
</tr>
<tr>
<td>449 793</td>
<td>80.84</td>
<td>500</td>
</tr>
</tbody>
</table>

**Table 7.2:** ASIC synthesis results with moved pipeline registers.

is about 15%, this has also improved over the previous synthesis.

Clearly, by either moving around the pipeline registers manually or by using register retiming, the differences in area and power consumption can improve.

It is worth noting that the power consumption numbers in Table 7.1 and 7.2 are taken directly from Design Compiler without any switching tests on any of the inputs. As such they should be considered only as estimates.
Figure 7.3: Area comparison between "FP" and "No FP" datapath with moved pipeline registers.

Figure 7.4: Power comparison between "FP" and "No FP" datapath with moved pipeline registers.
The focus of this chapter will be to evaluate if and under which circumstances the proposed extensions can be worth adding to the existing datapath.

8.1 Separate component comparison

Another approach to handling floating-point numbers in ePUMA would be to add dedicated floating-point cores to the different slave clusters. For example one core for addition, one core for multiplication, one core for division and so on. To demonstrate how much this approach would affect ePUMA in terms of area some DesignWare components were synthesized as standalone components with registers on all inputs and outputs and their area was compared to the added area of the solution presented in this thesis. The DesignWare components were synthesized for different target clock frequencies, the area results were then multiplied by 4 because the VPE datapath is capable of doing four floating-point operations in parallel. For simplicity, only addition and multiplication cores were synthesized. The results can be seen in Table 8.1 below.

The "VPE FP Extra area" column in Table 8.1 is the difference in area between the FP and NoFP VPE datapaths for the different clock frequencies (calculated from Table 7.2). Also remember that the "Total" column corresponds to the area of the addition and multiplication together, multiplied by 4 \((\text{Area}_{\text{Add}} + \text{Area}_{\text{Mul}}) \cdot 4\), this is done because four floating-point calculations can be done in VPE, in parallel.

As can be seen in Table 8.1 the difference is quite significant even though only addition together with multiplication was compared with the 14 operations that have been implemented so far for the VPE. In the case of 400MHz, the reference
Table 8.1: Comparison between DesignWare components and proposed VPE extensions.

<table>
<thead>
<tr>
<th>Frequency (MHz)</th>
<th>DW Components</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3 476</td>
<td>6 574</td>
</tr>
<tr>
<td>200</td>
<td>5 144</td>
<td>7 331</td>
</tr>
<tr>
<td>300</td>
<td>6 065</td>
<td>8 364</td>
</tr>
<tr>
<td>350</td>
<td>5 602</td>
<td>9 567</td>
</tr>
<tr>
<td>400</td>
<td>7 129</td>
<td>9 872</td>
</tr>
</tbody>
</table>

8.2 Software and Hardware comparison

Under which circumstances is it worthwhile to implement these proposed VPE extensions? In an attempt to answer that question, a comparison will be made between doing floating-point calculations in software and doing them in hardware and trying to correlate that cost to the extra area that is required.

Doing floating-point calculations in software has the benefit of not requiring extra hardware but has the downside of requiring more time to complete each calculation. Determining the exact number of cycles that is required to perform a floating-point operation with a software implementation is hard and depends on many different factors like data dependencies among different operations operands, what kind of hardware support there is and so on. So for simplicity floating-point addition and multiplication will be looked closer on, mainly because these operations are the most commonly used among floating-point operations.

A rough estimate on performing an addition or multiplication is 60 cycles in soft-
8.2Software and Hardware comparison

ware, for the VPE. Addition can potentially be much worse (around 200 cycles) if there is no FF1 hardware support and the normalization step needs to be looped entirely in software. So 60 cycles is a quite generous estimate.

Using this cycle cost, Equations 8.1-8.3 are presented:

\[ C = 60 \]  \hspace{1cm} (8.1)

\[ D = F \cdot C + (1 - F) \cdot 1 \]  \hspace{1cm} (8.2)

\[ Score = \frac{\text{Calculations}}{D \cdot \text{Area}_{FP, NoFP}} = \frac{\text{Calculations}}{D \cdot \text{Area}_{FP, NoFP}} \]  \hspace{1cm} (8.3)

An explanation for each equation:

- **Equation 8.1**: \( C \) is the estimated cost of doing a floating-point operation in the VPE without hardware support.

- **Equation 8.2**: \( D \) is a delay factor for an arbitrary application, giving an indication of how much an application is slowed down because floating-point calculations have to be done in software. \( F \) stands for how much of the application contains floating-point operations, ranging from 0 to 1, multiplied by the cycle cost \( C \). The \( (1 - F) \) part represents the contribution from the operations that are non floating-point calculations, they have a cost of 1.

  This delay factor has a best value of 1, which means that floating-point calculations are done as fast as other calculations, i.e \( C = 1 \). This is the case with a floating-point VPE datapath. The worst value is when all instructions are floating-point operations and \( F = 1 \), then \( D = C \). Which is 60 for a non floating-point datapath.

- **Equation 8.3**: \( Score \) will serve as a benchmark for different setups. \( \text{Calculations} \) tells how many calculations can be done per second, in an ideal situation. For example if the VPE is run at 400MHz then 400 million calculations can be done each second. The idealized \( \text{Calculations} \) variable is divided by the delay factor \( D \) to indicate how many calculations are actually done each second. This in turn is correlated to area by dividing by either with \( \text{Area}_{FP} \) (the area of the floating-point datapath) or \( \text{Area}_{NoFP} \) (the area of the non floating-point datapath).

  This equation basically says how many actual calculations that are done each second per \( \mu m^2 \). A higher score is more desirable than a lower score.

Equation 8.3 will be used to compare software solutions with the proposed extensions presented in this thesis. A comparison will be made for 400MHz and then continue with some general remarks. Area values are taken from Table 7.2. Table 8.2 shows the \( Score \) values for differing \( F \) values. A figure can also be seen in Figure 8.1.

Note that \( Score_{FP} \) is the same for all \( F \) values because floating-point and non
floating-point calculations can be done in one clock cycle in the floating-point VPE datapath. Therefore $C = 1 \Rightarrow D = 1$ for all variations of $F$ in that case.

<table>
<thead>
<tr>
<th>$F$</th>
<th>$\text{Score}_{\text{NoFP}}$</th>
<th>$\text{Score}_{\text{FP}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1420</td>
<td>1147</td>
</tr>
<tr>
<td>0.1</td>
<td>206</td>
<td>1147</td>
</tr>
<tr>
<td>0.2</td>
<td>111</td>
<td>1147</td>
</tr>
<tr>
<td>0.3</td>
<td>76</td>
<td>1147</td>
</tr>
<tr>
<td>0.4</td>
<td>58</td>
<td>1147</td>
</tr>
<tr>
<td>0.5</td>
<td>47</td>
<td>1147</td>
</tr>
<tr>
<td>0.6</td>
<td>39</td>
<td>1147</td>
</tr>
<tr>
<td>0.7</td>
<td>34</td>
<td>1147</td>
</tr>
<tr>
<td>0.8</td>
<td>29</td>
<td>1147</td>
</tr>
<tr>
<td>0.9</td>
<td>26</td>
<td>1147</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>1147</td>
</tr>
</tbody>
</table>

Table 8.2: Different Score values for different $F$ values at 400MHz. Higher Score value is better.

Figure 8.1: Comparison between hardware and software solutions at 400MHz.
As can be seen in Figure 8.1 the score for the NoFP datapath is, not surprisingly, better when no part of an application contains floating-point operations. But the score dips very fast as soon as floating-point operations are added. From the figure it can be seen that the breakpoint occurs somewhere when $F$ is between 0 and 0.1, but exactly where? This will be deduced mathematically. Equation 8.3 will be used to find a breakpoint value of $D$ called $D_{Br}$. That value of $D_{Br}$ will together with Equation 8.2 then be used to find a breakpoint value of $F$ called $F_{Br}$. These steps can be seen in Equations 8.4 and 8.5.

$$\frac{Calculations}{D_{Br} \cdot Area_{NoFP}} = \frac{Calculations}{Area_{FP}} \Rightarrow D_{Br} = \frac{Area_{FP}}{Area_{NoFP}}$$ (8.4)

$$D_{Br} = F_{Br} \cdot C + (1 - F_{Br}) \Leftrightarrow D_{Br} - 1 = F_{Br}(C - 1) \Rightarrow F_{Br} = \frac{D_{Br} - 1}{C - 1}$$ (8.5)

Combining these two equations into one for clarity:

$$F_{Br} = \frac{D_{Br} - 1}{C - 1} = \frac{Area_{FP}}{Area_{NoFP} - 1}$$ (8.6)

From Equation 8.6 and with area numbers for 400MHz (from Table 7.2) the breakpoint in Figure 8.1 occurs at $F_{Br} \approx 0.004$. This means that for any given application, if more than 0.4% of the calculations are floating-point calculations then it is better to use the VPE datapath (run at 400MHz) with the proposed extensions instead of doing the calculations in software. Some $F_{Br}$ values for various frequencies can be seen in Table 8.3.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>$F_{Br}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.002</td>
</tr>
<tr>
<td>200</td>
<td>0.002</td>
</tr>
<tr>
<td>300</td>
<td>0.003</td>
</tr>
<tr>
<td>400</td>
<td>0.004</td>
</tr>
</tbody>
</table>

Table 8.3: Breakpoint values at other frequencies.

Even if a more generous $C$ value is chosen, for example 10. It still yields $F_{Br} \approx 0.03$ in the case of 400MHz, which is still very good.

As a concluding remark it can be said that under these comparisons, the extensions to the VPE datapath was generally worthwhile.
The goal of this thesis was to implement support for some floating-point arithmetic operations, while trying to keep down the hardware and power consumption costs. This has largely been accomplished, the target frequency of 500MHz was not achieved. But with the lack of register retiming in the synthesis step, 500MHz can likely be achieved in the future. Improvements can be done though. For example, the "FP operations" block in the MUL stage (see Section 5.1 and Figure 5.1) can be optimized further with regards to its usage of adders. Particularly in the addition/subtraction of floating-point numbers (see Section 4.1.2), the exponent together with mantissa of OpA is checked to be greater than the exponent and mantissa of OpB (the "gt" block in Figure 4.1). And separately the exponents are subtracted inside another adder. These can be combined into one addition but would complicate the sign calculation (see Section 4.1.2) and by requiring another delay register. So that the adder stage knows which floating-point operation is done, to make sure that the sign bit is not altered for other floating-point operations by mistake.

This implementation supports NaN, infinities and only supports a very narrow band of subnormal numbers, pertaining to rounding in multiplication and division (see Section 4.7) but it has some costs too. In most real world applications, the need for handling NaN and Inf is extremely rare, the same is true for subnormal numbers as well. If the support for these features were scrapped, hardware costs could be further reduced.

Another useful improvement would be to support some square-root operation, either $\sqrt{x}$ or $\frac{1}{\sqrt{x}}$, because some form of square-rooting is done frequently (in multimedia applications). The cost could potentially be small, since the algorithm has similarities to floating-point division. This operation was considered at the
start of this thesis, but was eventually scrapped due to time considerations.

A frequently used instruction is the MAC (multiply-and-accumulate) operation, this operation was never considered for implementation because of the heavy modifications that would be needed but would maybe be interesting for future projects.

9.1 Future Work

Some suggestions for future work related to this thesis:

- Implement flags for some of the operations, so that control flow manipulations can be done using floating-point arithmetic also. Much of this work can be done now, without much effort (see Section 3.4).

- Investigate if a 16-bit floating-point representation can be useful and, implement support for it.

- Implement support for additional floating-point operations, like square-root or multiply-and-accumulate for example.

- Further optimize the "FP operations" block in the MUL stage.


ACCROP block, 45
ACR, x, 45
Adder stage, 43
Addition, 18
Addition algorithm, 18
Addition implementation, 20
Addition rounding, 38
Alignment, 19
ALU, x, 43, 44
ASIC, x

Cell Broadband Engine, 5
CLOSE path, 21
Compare operations, 30
Conversion operations, 33
Coverage models, 51
CPU, x
CVTFP2W, 35, 43, 46
CVTFP2W rounding, 38
CVTW2FP, 33

Denormal rounding, 38
Division, 26
Division algorithm, 27
Division implementation, 28
Division rounding, 38
DSP, x, 5

ePUMA, x, 1
ePUMA Overview, 5
Exponent MUX, 47

FAR path, 21
FF1 block, 46

Floating-point exceptions, 15
Floating-point exponent, 12
Floating-point format, 11
Floating-point mantissa, 12
Floating-point precision, 13
Floating-point rounding schemes, 13
Floating-point sign, 12
FP Operation block, 40, 46
FP Registers block, 40, 44
FPABS, 36
fpabs, 43
FPABSD, 18, 43
FPADD, 18, 43
fpadder, 43
FPDIV, 26, 43, 45
fpdiv, 43
FPGA, x
FPgen, x, 3, 51
FPMAX, 30, 44
FPMAXM, 30, 44
FPMIN, 30, 44
FPMINM, 30, 44
FPMUL, 22
FPNEG, 36
FPRECI, 26, 42, 43, 46, 49
FPSUB, 18, 43
FPU, x

Goal, 1
GTP, x, 3, 51

HR, x, 45

Infinity, 13, 47, 52
Lane, 39
Left adjustment, 48
LSB, x
LUT, x
LVM, x, 7

MAC, x, 9
MAC stage, 44
Mantissa MUX, 47
Master Processor, 7
MSB, x
MUL stage, 39
Multiplication, 22, 25, 48
Multiplication algorithm, 22
Multiplication implementation, 23
Multiplication rounding, 38

NaN, x, 2, 13, 14, 18, 39, 46, 47
NEAR path, 21

PM, x, 7

Q-notation, 1, 33
qNaN, x, 2, 47, 52

Restoring division, 29
Restoring division remainder, 30, 45
RISC, x
RNE, x
RNU, x
RNU Implementation, 36
Rounder block, 46
RTL, x, 3

Saturation block, 46
Scaler block, 46
Scope, 2
Sign operations, 36
SIMD, x, 5
Slave cluster, 6, 7
sNaN, x, 2
SRF, x, 7

Unified design, 39

VPE, x, 7
VPE data vectors, 8
VPE datapath, 7, 8

VPE pipeline, 8
VRF, x, 7
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Gaspar Kolumban