Norwegian University of
Science and Technology

Vectorized 128-bit Input FP16/FP32/FP64 Floating-Point Multiplier

Espen Stenersen

Master of Science in Electronics
Submission date: June 2008
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Torstein Dybdal, ARM Norway AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

In 3D graphics, several floating-point formats are used in computations. The task is to make a floating-point multiplier with the current features:

- 256-bit input vector and 128-bit output.
- Supporting FP16/FP32/FP64 inputs.
- IEEE754 conforming.
- 5 step pipeline.
- Simple handshake interface.

Depending on input formats the following operations should be performed:

- Vec4 FP16 multiply uses a 128-bit input vector, and produces a 64-bit output vector.
- Vec4 FP32 multiply uses a 256-bit input vector, and produces a 128-bit output vector.
- Vec2 FP64 multiply uses a 256-bit input vector, and produces a 128-bit output vector.

The assignment is a continuation of the project task, where different floating-point multiplier architectures were proposed, analyzed and evaluated. Based on this, further analysis has to be made before an architecture is chosen. Implement the chosen architecture at register transfer level, for testing and synthesis.

Assignment given: 15. January 2008
Supervisor: Per Gunnar Kjeldsberg, IET

Abstract

3D graphic accelerators are often limited by their floating-point performance. A Graphic Processing Unit (GPU) has several specialized floating-point units to achieve high throughput and performance. The floating-point units consume a large part of total area and power consumption, and hence architectural choices are important to evaluate when implementing the design. GPUs are specially tuned for performing a set of operations on large sets of data. The task of a 3D graphic solution is to render a image or a scene. The scene contains geometric primitives as well as descriptions of the light, the way each object reflects light and the viewer position and orientation.

This thesis evaluates four different pipelined, vectorized floating-point multipliers, supporting 16-bit, 32-bit and 64-bit floating-point numbers. The architectures are compared concerning area usage, power consumption and performance. Two of the architectures are implemented at Register Transfer Level (RTL), tested and synthesized, to see if assumptions made in the estimation methodologies are accurate enough to select the best architecture to implement given a set of architectures and constraints. The first architecture trades area for lower power consumption with a throughput of 38.4 $Gbit/s$ at $300MHz$ clock frequency, and the second architecture trades power for smaller area with equal throughput. The two architectures are synthesized at $200MHz$, $300MHz$ and $400MHz$ clock frequency, in a $65nm$ low-power standard cell library and a $90nm$ general purpose library, and for different input data format distributions, to compare area and power results at different clock frequencies, input data distributions and target technology.

Architecture one has lower power consumption than architecture two at all clock frequencies and input data format distributions. At $300MHz$, architecture one has a total power consumption of $1.9210mW$ at $65nm$, and $15.4090mW$ at $90nm$. Architecture two has a total power consumption of $7.3569mW$ at $65nm$, and $17.4640mW$ at $90nm$. Architecture two requires less area than architecture one at all clock frequencies. At $300MHz$, architecture one has a total area of $59816.4414\mu m^2$ at $65nm$, and $116362.0625\mu m^2$ at $90nm$. Architecture two has a total area of $50843.0\mu m^2$ at $65nm$, and $95242.0469\mu m^2$ at $90nm$.

Preface

This thesis concludes my Master's degree in Electrical Engineering at Norwegian University of Science and Technology (NTNU), and is a continuation of my 2007 autumn project. The assignment is given by ARM Norway, and involves research, implementation, testing and synthesis of a vectorized floating-point multiplier. The work was carried out from January 2008 to June 2008, and the topic was interesting, challenging and very instructive.

I spent a lot of time researching floating-point implementations in hardware, especially floating-point rounding, in addition to power consumption in sub-micron technologies. IEEE specifies a detailed standard for binary floating-point arithmetic, but leaves the implementation completely to the designer. Two different vectorized floating-point multipliers was implemented using the Verilog Hardware Description Language, which I had little knowledge of before starting this assignment. A significant amount of time was spent developing a sufficient testplan for the designs, and by researching and understanding the tools used for synthesis and the Tcl scripting language. Working on this thesis, I learned much about floating-point arithmetic in hardware, the synthesis and optimization process, and power consumption in different target technologies. I also gained further knowledge of digital design in general, and the Verilog Hardware Description Language.

A special thank goes to my supervisors, Associate Professor Per Gunnar Kjeldsberg (NTNU), and Torstein Hernes Dybdahl (ARM) for their guidance, feedback and interest in this assignment. I would also like to thank my fellow students for comments and constructive questions, my family and friends.

Espen Stenersen
Trondheim, June 2008.

Contents

1 Introduction **1**
1.1 Floating-Point Multiplication 3
1.2 Power and Area Optimized Designs 4
1.2.1 Low Power Design 4
1.2.2 Area Optimized Design 6
1.3 High-Speed Multiplication 6
1.4 Architecture Search-space Exploration 7
1.4.1 Power Consumption 7
1.4.2 Area Usage 8
1.4.3 Throughput and Delay 8
1.5 Proposed Architectures 9
1.5.1 Architecture One 9
1.5.2 Architecture Two 10
1.5.3 Architecture Three 10
1.5.4 Architecture Four 11
1.6 Thesis Organization and Main Contributions 11

2 Architecture Estimations **13**
2.1 Power Estimation Methodology 13
2.2 Power Estimation 17
2.3 Area Estimation 19
2.4 Performance Estimation 22
2.5 Trade-Off Considerations 23

3 Implementation **25**
3.1 Choosing Architecture 25
3.2 Vectorized Floating-Point Multiplier 26
3.2.1 Inputs 27
3.2.2 Outputs 29
3.2.3 Architecture Description 29
3.3 Testing and Simulation 36
3.3.1 Reference Circuit 37
3.3.2 Simulations 37

4 Synthesis Results **39**
4.1 Synopsys® 39
4.1.1 Static Power 40
4.1.2 Dynamic Power 40
4.1.3 Capturing Switching Activity for Synthesis 40
4.1.4 Setting Design Constraints 41
4.2 Architecture One 41
4.2.1 Power 42
4.2.2 Area 49
4.3 Architecture Two 50
4.3.1 Power 50
4.3.2 Area 57
4.4 Power Comparison 59
4.5 Area Comparison 63

5 Conclusions **67**
5.1 Estimation Methodologies 68
5.2 Power Results 68
5.3 Area Results 69
5.4 Future Work 69

A Architecture One Verilog Sources **73**

B Architecture Two Verilog Sources **119**

C Test Data Generator **145**

D Simulation Sources **151**
D.1 Vectorized DesignWare floating-point multiplier Source 151
D.2 Testbench Sources 158
D.3 Switching Activity Simulation Source 171

List of Tables

2.1 Normalized leakage current for logic gates [1]. 14
2.2 Significand multipliers static power consumption. 16
2.3 Significand multipliers dynamic power consumption. 16
2.4 Static power estimation of proposed architectures. 17
2.5 Total power consumption, 256-bit input vector. 18
2.6 Total power consumption, 128-bit input vector. 19
2.7 Architecture area comparison, FA-cells and equivalent register-size. 21

3.1 Trade-off considerations. 26
3.2 Format encoding. 28
3.3 Rounding modes encoding. 28
3.4 Rounding mode reduction. 36

4.1 Architecture one, 65*nm* CMOS total power consumption. . . 42
4.2 Architecture one, 65*nm* CMOS building blocks power consumption. 43
4.3 Architecture one, 90*nm* CMOS total power consumption. . . 44
4.4 Architecture one, 90*nm* CMOS building blocks power consumption. 44
4.5 Architecture one, 65*nm* CMOS area usage. 49
4.6 Architecture one, 90*nm* CMOS area usage. 50
4.7 Architecture two, 65*nm* CMOS total power consumption. . . 51
4.8 Architecture two, 65*nm* CMOS building blocks power consumption. 51
4.9 Architecture two, 90*nm* CMOS total power consumption. . . 52
4.10 Architecture two, 90*nm* CMOS building blocks power consumption. 52
4.11 Architecture two, 65*nm* CMOS area usage. 58
4.12 Architecture two, 90*nm* CMOS area usage. 58

List of Figures

2.1 Full-Adder gate-level model. 15
2.2 Ratio of leakage power to total power in a 65nm CMOS library at different process corners, supply voltages and temeratures [2]. 15
2.3 Architecture power comparison. 20
2.4 Architecture area comparison. 21
2.5 Architecture latency comparison. 23

3.1 Vectorized floating-point multiplier block diagram. 27
3.2 First input vector layout. 27
3.3 Second input vector layout. 27
3.4 Clear register layout. 28
3.5 Product vector layout. 29
3.6 Exception register layout. 29
3.7 Vectorized floating-point multiplier simple timing diagram. 29
3.8 Vectorized floating-point multiplier architecture drawing. 30
3.9 Architecture one exponent unit. 32
3.10 Architecture two exponent unit. 32
3.11 Architecture one significand multiplier unit. 34
3.12 Architecture two significand multiplier unit. 34
3.13 Architecture one rounding and exception unit. 35
3.14 Architecture two rounding and exception unit. 36
3.15 DW_vec_fp_mult block diagram. 37

4.1 Architecture one, 65nm CMOS power consumption. 45
4.2 Architecture one, 90nm CMOS power consumption. 47
4.3 Architecture one, 90nm and 65nm CMOS power comparison. 48
4.4 Architecture two, 65nm CMOS power consumption. 54
4.5 Architecture two, 90nm CMOS power consumption. 55
4.6 Architecture two, 90nm and 65nm CMOS power comparison. 56
4.7 65nm architecture power comparison. 60
4.8 90nm architecture power comparison. 61
4.9 Estimated vs. real power comparison. 62

4.10 65nm CMOS architecture area comparison. 65
4.11 90nm CMOS architecture area comparison. 66

Chapter 1

Introduction

Floating-point numbers are frequently used in scientific calculations, digital signal processing applications and in 3D graphics. In 3D graphics, floating-point performance are especially demanding and several floating-point number formats are used in computations. 3D graphics accelerators have a highly parallel structure that makes them more efficient for certain algorithms than general purpose processors. The 16-bit, 32-bit and 64-bit floating-point formats FP16, FP32 and FP64 are used for high dynamic range textures, that is, where light and dark textures are spanned over a large area. All formats can be used as vertex coordinates, and the FP64 format is the minimum for graphic processing units (GPUs) to be used in scientific calculations.

ARM Norway develops hardware graphic accelerators, specifically tuned for embedded system environments, supporting the OpenGL ES and OpenVG APIs, which focus on high performance and low power consumption [3]. Mali™ 200 with GP2 fully supports OpenGL ES v2.0, v1.1 and OpenVG v1.0. Detailed information about the Mali™ 3D Graphics System Solution can be found in [4]. OpenGL ES is a royalty-free cross-platform API for full function 2D and 3D graphics on embedded systems [5], and OpenVG is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG [6].

The purpose of this floating-point multiplier is to support three different floating-point number formats, FP16, FP32 and FP64. It is a vectorized floating-point multiplier in the sense that the input vector is a vector of operands, where three different types input vectors are supported. For the FP16 format, the input vector should be

$$[127:0] = [D1, D0, C1, C0, B1, B0, A1, A0]$$

and the output will become

$$[63:0] = [D1 \times D0, C1 \times C0, B1 \times B0, A1 \times A0]$$

For the FP32 format, the input vector should be

$$[255 : 0] = [D1, D0, C1, C0, B1, B0, A1, A0]$$

and the output will become

$$[127 : 0] = [D1 \times D0, C1 \times C0, B1 \times B0, A1 \times A0]$$

For the FP64 format, the input vector should be

$$[255 : 0] = [D1, D0, C1, C0]$$

and the output will become

$$[127 : 0] = [D1 \times D0, C1 \times C0]$$

Depending on the input vector data format, the output vector will be a 64-bit or 128-bit vector of floating-point products on the IEEE 754 format.

This thesis is a continuation of my 2007 autumn project [7]. [7] presents four possible vectorized floating-point multiplier architectures with different area, power and throughput profiles. These architectures are evaluated and compared concerning area, power, throughput and latency. This thesis will further investigate power consumption of the four architectures and two architectures will be selected for RTL implementation. The implemented architectures will be tested and synthesized to see if assumptions and methodologies used to compare area and power are sufficient to select the best alternative given a set of constraints.

In a 3D graphic processing application, throughput are very important because it is operating on large data sets describing a frame or scene, where for example shading, lighting, positions and viewers perspective are considered. In any hardware implementation, area and power are usually important constraints. In a handheld, battery powered device, both area and power consumption are very important. Because of highly parallel computations, and the pipelined architecture of graphic accelerators, clock frequency is typically much lower than in a modern general purpose CPU.

This Chapter will first present the floating-point multiplication algorithm in Section 1.1. Then design strategies for low power and small area will be discussed in Section 1.2. In Section 1.3 some high-speed multiplier schemes are presented, and in Section 1.4, the vectorized floating-point multiplier architecture search-space will be explored. Section 1.5 presents the architectures evaluated in [7], and in Section 1.6, the outline and main contributions of this thesis are presented.

1.1 Floating-Point Multiplication

The IEEE standard for binary floating-point arithmetic specifies a detailed standard for floating-point representation in computers [8]. Floating-point numbers are represented by a sign, an exponent and a significand, and are written as follows

$$floating - point\ number = (-1)^s \times f \times \beta^{e-bias}, \tag{1.1}$$

where s represents the sign, f the significand, e the exponent and β the base or radix. In IEEE 754 the base is always 2. Floating-point numbers in IEEE 754 format are biased to ensure that the exponent is always greater than zero, and thus making comparison between numbers easier. The exponent represents the range, and the significand the precision of the number.

Given to floating point numbers $n_1 = (-1)^{s_1} \times f_1 \times 2^{e_1}$ and $n_2 = (-1)^{s_2} \times f_2 \times 2^{e_2}$. The floating-point product is computed as

$$n = (-1)^{s_1+s_2} \times (f_1 \times f_2) \times 2^{e_1+e_2-bias}$$

This can be achieved by a simple algorithm. The floating-point multiplication algorithm is straight-forward; exponents are added and bias subtracted, significands are multiplied, and signs computed by an XOR-operation. Because the result of the significand multiplication is of width $2n$, where n is the width of each significand, rounding has to be performed to obtain a final product in the IEEE specified format. The algorithm is given below.

```
1   // Sign, exponent and significand computation.
2   sign = sign_1 ^ sign_2;
3   exponent = exponent_1 + exponent_2 - bias;
4   significand = significand_1 * significand_2;
5
6   // Normalizing.
7   if (normalize)
8   {
9       significand = significand >> 1
10      exponent = e + 1;
11  }
12
13  // Rounding.
14  if (roundup)
15  {
16      significand = signficand + 1;
17  }
18
19  // Post-normalizing.
20  if (postnormalize)
21  {
22      significand = significand >> 1;
23      exponent = exponent + 1;
24  }
25
26  product = {sign, exponent, significand};
```

For numbers on scientific notation, the fractional part has to be normalized if it is outside the interval [0, 10). Normalizing is performed by incrementing the exponent by one and dividing the fractional part by ten. Likewise, in binary IEEE arithmetic, if the significand is outside the interval [0,2) it has to be normalized, and normalizing is performed by incrementing the exponent by one and dividing the significand by two. The decision for normalizing is simple; if the most significant bit in the result after significand multiplication equals one, every bit in the significand are shifted one position to the right and the exponent incremented by one. If significand is to be rounded, a '1' is added to the significand. If the significand is not normalized after rounding, post-normalizing occurs. The bits in the significand are shifted one position to the right, and exponent incremented by one.

IEEE specifies four rounding modes, round-to-nearest even, round-to positive infinity, round-to negative infinity and round-to zero. The rounding decision are based on rounding mode and guard digits. In round-to-nearest even mode, three guard digits are needed. In round-to positive infinity and round-to negative infinity, two guard digits in addition to sign bit are needed for making correct rounding decision. Rounding decisions for the different rounding modes and guard digits are further described in [9].

1.2 Power and Area Optimized Designs

Low power and small area can be contradicting requirements, but both is very important in handheld devices. Low power design exploit numerous techniques such as dynamic voltage and frequency scaling as well as different coding schemes and number representations to reduce the overall power consumption. Low area can be achieved by for example resource sharing, but is a trade-off between area, speed and latency.

1.2.1 Low Power Design

The average power dissipation in a CMOS circuit is given by the equation [10]

$$\begin{aligned} P_{avarage} &= P_{static} + P_{short-circuit} + P_{dynamic} \\ &= V_{DD}I_{static} + V_{DD}I_{short-circuit} + \alpha C_L V_{DD}^2 f_{clk} \end{aligned} \tag{1.2}$$

where α corresponds to the average number of $0 \rightarrow 1$ transitions at a given node each clock cycle, V_{DD} the supply voltage, C_L the capacitive load switched each cycle and f_{clk} the clock frequency.

Static Power Consumption

The static power dissipation is technology dependent, and increases as the transistor dimension and threshold voltages decreases. The static power consumption is an increasing problem in deep sub-micron technologies, and proportional to the amount of transistors in a given design. I_{static} is composed of leakage currents due to tunneling effects and sub-threshold conduction. Static power dissipation can be reduced by optimizing the supply voltage and threshold voltage, or by reducing the amount of transistors, and hence area. Other techniques such as channel engineering and changing the doping profile of the transistors may also be used. In order to eliminate the static power dissipation, the supply voltage needs to be turned off when parts of the circuit is not used.

Short-circuit Power Consumption

The short-circuit current contributes to the average power consumption when both the PMOS and NMOS transistor conduct simultaneously, creating a direct path from V_{DD} to ground. Short-circuit currents can be minimized by designing PMOS and NMOS transistors with equal fall- and rise times.

Dynamic Power Consumption

The dynamic power dissipation is the main contributor to the average power consumption when the circuit is operating. To reduce the power consumption, either the switching activity, the capacitive load, the supply voltage, the clock frequency or a combination of these can be reduced. [11] describes three architectural techniques to reduce the power consumption in CMOS circuits, trading area for lower power dissipation through hardware duplication, pipelining or a combination of these. Through hardware duplication both supply voltage and clock frequency can be reduced at the cost of additional registers at the input, and a multiplexer at the output. Through hardware pipelining, the clock frequency or supply voltage can be reduced while still maintaining the same throughput as a similar non-pipelined circuit at a higher supply voltage. In graphic processing implementations, pipelining is often used to improve throughput. [11] also describes techniques for reducing the switching activity through algorithmic optimization. Statistical knowledge of the input data can be exploited to lower the power dissipation, through choosing the best number representation, and hence lower the switching activity.

In floating-point multipliers, the significand multipliers consume the larger part of the area. Therefore, these should be implemented as area and power efficient as possible in order to minimize both static and dynamic power

dissipation. Numerous techniques for multiplier designs represents different power consumptions and area usage.

1.2.2 Area Optimized Design

Area can be reduced at the expense of larger latency and lower throughput, or by reusing or sharing computational units efficiently. If throughput or latency is an absolute demand due to some timing constraints, there may be a limit to how much area can be reduced without violating those constraints. Area is an important design parameter in handheld devices due to the size of the devices, and the energy consumption.

In floating-point multipliers supporting several formats, area can be reduced mainly by using one multiplier computing the significands for each supported format. This affects the power consumption as well, the dynamic power consumption increases unless measures are taken to minimize this, and the static power consumption is reduced due to less transistors.

1.3 High-Speed Multiplication

Multiplication involves two basic operations, generating partial products and accumulation of the partial products. The time to perform a multiplication can be reduced by either reducing the number of partial products or speed-up their accumulation [9]. High-speed multipliers can be divided into two different categories, bit-parallel- and bit-serial multipliers. Bit-parallel multipliers can be further divided into three different categories [12].

- Shift-and-add multipliers.
- Parallel multipliers.
- Array multipliers.

Shift-and-add multipliers generates partial products sequentially and accumulates them successively. This type of multiplier require the least amount of area, but is also the slowest. It can be implemented using only one bit-parallel adder and successively adding the partial products row- or column-wise. The shift-and-add multiplier requires n^2 AND operations, and $n-1$ shift operations, where n is the with of the operands.

Parallel multipliers generates all partial products in parallel, and uses an adder-tree for their accumulation. Thus it can be partitioned into three parts, partial product generation or reduction, partial product accumulation (carry-free addition) and carry-propagation addition for the final result. Partial product reduction is most often performed by some version

of Booth's algorithm, and partial product accumulation by a Dadda [13] or Wallace [14] tree. The carry-propagation addition is often performed by a carry-lookahead adder. Tree-based multipliers have a latency proportional to $O(log_2(n))$, where n is the with of the operands.

Array multipliers consists of almost identical cells for the generation of partial products and their accumulation. Compared to three-based multipliers, the array multiplier utilizes the least amount of area, but has larger latency. Array multipliers are good candidates for pipelining, and relatively easy to implement. The cells for partial product generation and accumulation are adders, most often implemented as carry-save adders to make them more efficient. Array multipliers have a latency proportional to $O(n)$, where n is the with of the operands.

High-speed multipliers and multiplier schemes are further described and elaborated in [7].

1.4 Architecture Search-space Exploration

Given the specification, a vectorized, pipelined IEEE compliant floating-point multiplier supporting 16-bit, 32-bit and 64-bit floating-point numbers, there is a minimum requirement of computational units. One significand multiplier, exponent adder and rounding and exception logic capable of handling every supported format is required. In addition to input- and output registers, and pipeline registers.

1.4.1 Power Consumption

Power consumption consists of both a static and a dynamic component. The static component is hard to estimate because it is strongly technology dependent, but is directly related to the chip area. The dynamic component depends on variables such as switching activity and glitching. Glitching activity can be much higher than functional activity in certain datapath modules such as adders and multipliers, and in a 32-bit multiplier, the power dissipation due to glitches can be three times higher than that due to functional activities [15]. Glitching can be reduced by balancing signal paths, and hence reducing uneven arrival times.

The optimized minimum power solution is difficult to obtain because the probability distribution of the different formats is unknown, and because static power dissipation can be a large contributor to the overall power and energy consumption. The choice of using only one significand multiplier for every supported format, or several significand multipliers for every supported format is crucial for both the power and energy consumption as well as the

area usage and throughput. However, the FP32 format is assumed to be the main data format, and used frequently compared the FP16 and FP64 formats. Because the use of the different supported formats is unknown, and only assumptions can be made, it is difficult to optimize the overall floating-point multiplier concerning power consumption. If FP16 computations are performed very infrequently compared to FP32 computations, the FP16 significand computations can be performed in the FP32 significand multiplier with little power overhead in the long run. This favors a solution with at least one 24-bit significand multiplier for the FP32 (and FP16) format, and one 53-bit significand multiplier for the FP64 format. However, even if the power dissipation seems to be low, the total energy consumption by computing an entire input vector has to be considered.

Reducing the input vector also reduces area requirements due to less computational units and registers, and hence less static power dissipation and total power dissipation. However, energy consumption is not significantly reduced. Because of reduced throughput, additional cycles are needed to compute an entire input vector.

1.4.2 Area Usage

A minimum area solution would have only one XOR-gate computing the sign, one exponent adder and one significand multiplier supporting every format, rounding and normalizing logic supporting all three formats and a 256-bit input register and an 128-bit output register, in addition to exception logic handling exceptions raised during computation. Pipeline registers will infer a significant increase in area, and should not be used in a minimum area solution. This architecture will suffer from very low throughput and clock-speed, in addition to a high power consumption due to glitching in very long and possible uneven signal paths plus functional switching. This floating-point multiplier is inefficient and energy consuming, and will not be suited for a battery powered graphic solution.

Power and energy consumption, as well as throughput and critical path delay, can be improved at the expense of additional pipeline registers. The area consuming part of any floating-point multiplier is the significand multiplier. Different multiplier schemes may be used to reduce the overall area usage. Amongst bit-parallel multipliers, the array multiplier requires the least amount of area, but is also the slowest [12].

1.4.3 Throughput and Delay

A vectorized floating-point multiplier, optimized concerning throughput and delay requires pipelining to reduce the critical path delay and parallel com-

putation to increase the data processed each cycle. However, parallelizing the computations requires additional computational units, which increases both area and static power dissipation significantly. In a graphic processing application, high throughput is an important criteria, however in a battery powered graphic processing application performance has to be a compromise between throughput, area and energy consumption.

To maximize the throughput, at least two significand multipliers and exponent adders supporting the FP32 and FP16 format, and two significand multipliers and exponent adders supported every format are needed, in addition to four XOR-gates computing the signs, and rounding and normalizing logic capable of handling four products in parallel. The exception logic also needs to be able to handle exceptions from four products simultaneously. An 128-bit input bus may not only reduce area and power consumption, it may also reduce the throughput.

Critical path delay is limited by the FP64 significand multiplier, assuming registers at the input and output of this multiplier. Techniques for fast multiplication can be applied to speed up the multiplication. Compression multipliers such as Dadda [13] and Wallace [14], or versions of this, in addition to techniques for reducing partial products, speeds up the multiplication at the expense of area and possibly power overhead.

1.5 Proposed Architectures

The architectures presented in [7] lies somewhere in between the solutions discussed above, and have different power consumptions, area, throughputs and latencies, where latency is measured in cycles before a product vector is ready at the output. Four architectures are presented.

1.5.1 Architecture One

This architecture attempts to be a throughput and power optimized solution at the cost of increased area. Achieving a high throughput requires parallel computation of input vectors. To minimize the dynamic power consumption, two 53-bit multipliers, four 24-bit multipliers and four 11-bit multipliers are used to compute the significands of the FP64, FP32 and FP16 formats respectively. In addition, two 11-bit bit adders and subtractors, four 8-bit bit adders and subtractors and four 5-bit bit adders and subtractors to compute the exponents of the FP64, FP32 and FP16 formats respectively. Four XOR-gates are used to compute the signs. By using components that exactly fit the operand widths, unnecessary switching is reduced when computing the different formats. Architecture one has a latency of four cycles, assuming a 256-bit input bus, and throughput is 256 bits per clock cycle. But, if

input bus is reduced to 128-bit, throughput reduces to 128 bits per cycle and latency increases to five cycles. In addition, only one 53-bit multiplier, two 24-bit multipliers and two 16-bit multipliers, one 11-bit exponent adder and subtractor, two 8-bit adders and subtractors and two 5-bit adders and subtractors are needed if input bus is reduced to 128-bit. An architectural drawing of architecture one is given in [7].

1.5.2 Architecture Two

Architecture two attempts to be a throughput and area optimized solution by using more general significand multipliers and exponent adders than architecture one. Two 53-bit multipliers and two 24-bit multipliers are used to compute the significands of all supported formats. Two 11-bit adders and subtractors and two 8-bit adders and subtractors to compute the exponents of the FP16, FP32 and FP64 data formats, in addition to four XOR-gates computing the signs. By reducing the area, static power dissipation is reduced, but dynamic power is increased due to functional switching. Significands have to be extended to fit the with of the multipliers for the FP32 and FP16 formats. The 11-bit exponent adders have to support subtraction of three different bias values, and the 8-bit exponent adders have to support subtraction of the FP16 and FP32 bias values. As architecture one, this architecture has a latency of four cycles and a throughput of 256 bits per cycle, assuming 256-bit input bus. If input bus is reduced to 128-bit, latency increases to five cycles, and throughput decreases to 128 bits per cycle. As for architecture one, number of significand multipliers and exponent adders and subtractors are halved. The architectural layout of architecture two is also given in [7].

1.5.3 Architecture Three

Architecture three attempts to be an area and power optimized architecture, where throughput is traded for smaller area. One 53-bit multiplier, one 24-bit multiplier and one 11-bit multiplier computes the significands of the FP64, FP32 and FP16 formats respectively. One 11-bit adder and subtractor, one 8-bit adder and subtractor and one 5-bit adder and subtractor are used to compute the exponents of the FP64, FP32 and FP16 formats respectively. One XOR-gate is used to compute the signs. By reducing area, static power is reduced, and by using components that fit the operand width of their designated format, functional switching is reduced and hence dynamic power consumption. This architecture has a latency of six cycles, assuming 256-bit input bus. The throughput of this architecture is 64 bits per cycle. If input bus is reduced to 128-bit, neither latency or throughput is reduced because only one product is computed each cycle. However, input register size may be reduced and hence area and static power dissipation. Architecture three

should have a 64-bit input bus to avoid wait cycles, and hence reducing registers required, and area further. The architectural layout of architecture three is given in [7].

1.5.4 Architecture Four

This architecture is close to an area optimized solution, and almost identical to architecture three, except only one 53-bit multiplier is used to compute the significands of all supported formats, one 11-bit adder and subtractor is used to compute the exponents, and one XOR-gate computing the sign. The Exponent subtractor supports FP16, FP32 and FP64 bias values. By reducing area to a minimum of components needed for computing the products of all formats, static power is reduced even further but a the cost of functional switching. As architecture three, this architecture has a latency of six cycles and a throughput of 64 bits per cycle, assuming 256-bit input bus. If input bus is reduced, latency and throughput are unaffected.

As discussed in [7], and above, area, and hence static power, can be reduced for architecture one and two by reducing the input bus from 256-bit to 128-bit. This does not change the overall energy consumption significantly because an additional cycle is needed to compute an entire input vector. Architecture three and four are not affected by reducing the input bus. The rounding, normalizing/post-normalizing and exception logic are equal for all four architectures presented in [7].

1.6 Thesis Organization and Main Contributions

The rest of this thesis is organized as follows. In Chapter 2, a power estimation methodology is presented and used to compare the the architectures presented in [7] concerning power consumption. The architectures are further compared concerning area usage, and performance including latency and throughput. Chapter 2 also discusses trade-off considerations when choosing an architecture to implement. In Chapter 3, two architectures are selected for implementation, and the implemented architectures are presented and described. In addition, testing and simulation of the two architectures are discussed. Chapter 4 describes how synthesis has been performed, and presents the synthesis power and area results. The two architectures are further compared concerning power consumption and area usage. Chapter 5 concludes this thesis.

The main contributions of this thesis are:

- A power estimation methodology for comparing the relative differences

in power consumption of the architectures proposed in [7].

- Comprehensive RTL implementation of two vectorized floating-point multiplier architectures.
- Synthesis results of the two architectures realized in a $65nm$ low-power library, and a $90nm$ general purpose library, for comparison with estimations performed in this thesis, and in [7], in two different target technologies.

Chapter 2

Power, Area and Performance Estimation

Power, area and performance estimations are important to consider when choosing an architecture to implement. Especially power can be hard to estimate because it is strongly technology dependent, and both static and dynamic power dissipation have to be taken into account. When moving into deep sub-micron technology, static power dissipation can be a significant contributor to the total power consumption.

This Chapter will first present a power estimation methodology based on power dissipated by significand multipliers in Section 2.1. In Section 2.2, this power estimation methodology will be used to compare power consumption of the four architectures presented in Section 1.5. Section 2.3 compares area requirements of the proposed architectures, and in Section 2.4, latency, throughput and clock frequency of the proposed architectures will be discussed. Trade-off considerations that should be considered when choosing an architecture to implement are presented in Section 2.5.

2.1 Power Estimation Methodology

The significand multiplier is the major computational unit in any floating-point multiplier. Therefore, estimating the power consumed by the significand multipliers will give a good indication of the total power consumption of the overall floating-point multiplier.

When computing the resulting significand of two floating-point numbers of size n-bit, the n most significant bits of the $n \times n$-bit product are the bits of interest. This means that for example if a FP32 significand is computed in a FP64 significand multiplier, the FP32 significand has to be extended to fit the width of the FP64 significand multiplier. If additional bits are

appended as the most significant bits, shifting has to be performed after the multiplication, or multiplexers connected to the output register of the significand multiplier has to select the correct bits for further computations such as rounding and normalizing. Alternatively, additional bits can be appended as the least significant bits, and avoid the shifting or multiplexing.

In order to estimate the power consumption, both static and dynamic power consumption, a power model or methodology is needed. In [1] simulations of leakage currents for different logic gates are performed for a $65nm$ CMOS library, with standard threshold transistors and standard cells with a driving force of one. The result is given in Table 2.1. In [2], simulations are performed to analyze the ratio of static power dissipation to total power dissipation. The simulations are performed in a $65nm$ CMOS library for different process corners and different supply voltages and temperatures. In the simulated circuit it is assumed that 95% of the gates are quiet and 5% are switching. The simulation result is given in Figure 2.2.

Input	NAND	AND	XOR
L L	1	5.3	17.9
L H	5.9	10.2	17.9
H L	7.1	11.4	9.1
H H	4.5	14.5	9.1

Table 2.1: Normalized leakage current for logic gates [1].

As can be seen from Table 2.1, static power can be reduced by setting unused bits to other values than zero. However, this simple power model aims to highlight the relative differences between the four architectures, and not their exact power consumptions. As seen from Equation 1.2, static power is given by $I_{static} \times V_{DD}$. Assuming equal V_{DD} for all architectures, V_{DD} can be eliminated from the equation, and total static power of a Full-Adder can be computed as

$$2 \times 17.9 + 2 \times 1 + 1 \times 4.5 = \underline{42.3}$$

assuming unused bits are set to '0'. The Full-Adder model used for the static power computation is given in Figure 2.1, and differs from the Full-Adder model used in [7, 16]. The model used in Figure 2.1 utilizes 6 transistors less, and therefore reduces the area, and in addition makes it possible to use the simulated data from Table 2.1.

Figure 2.2 shows that for a typical $65nm$ CMOS process the static power dissipation is approximately 30% of total dissipated power. Hence if the static power is 42.3, the dynamic power will be

$$42.3 \times (7/3) = \underline{98.7}$$

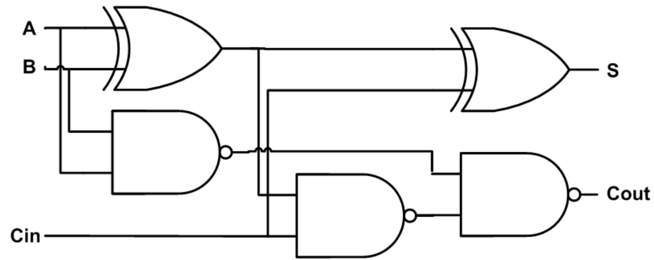


Figure 2.1: Full-Adder gate-level model.

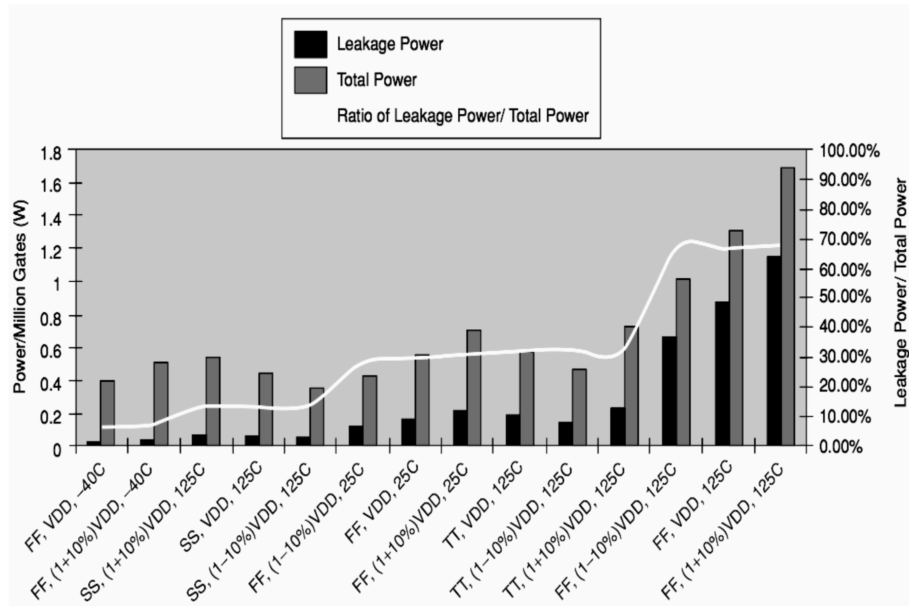


Figure 2.2: Ratio of leakage power to total power in a 65nm CMOS library at different process corners, supply voltages and temeratures [2].

Assuming that the significand multipliers are implemented as array multipliers as described in [7], the FP16 multiplier requires

$$11 \times 10 = \underline{110}$$

FAs, the FP32 multiplier

$$24 \times 23 = \underline{552}$$

FAs, and the FP64 multiplier

$$53 \times 52 = \underline{2756}.$$

FAs. The static power consumption of the three different multipliers are given in Table 2.2, normalized to the value of the FP16 multiplier, assuming every input-bits equals '0'.

Multiplier	Static Power	Normalized Static Power
FP16	4653.0	1.0
FP32	23349.6	5.0
FP64	116578.8	25.1

Table 2.2: Significand multipliers static power consumption.

As shown in Table 2.2, the 53-bit FP64 significand multiplier dissipates 25.1 times more static power than the 11-bit FP16 significand multiplier, and the 24-bit FP32 significand multiplier 5 times more than the FP16 multiplier. Assuming dynamic dissipated power equals approximately 70% of total power consumption, dynamic power consumption is computed and given in Table 2.3, where the dynamic power is normalized to the static power dissipation of the FP16 significand multiplier.

Multiplier	Dynamic Power	Normalized Dynamic Power
FP16	10857.0	2.3
FP32	54482.4	11.7
FP64	272017.2	58.5

Table 2.3: Significand multipliers dynamic power consumption.

This estimation methodology has several sources of error, which may lead to the wrong conclusions. The most severe source of error in this methodology, is probably the assumption that 95% of the gates are quit during the simulations given in Figure 2.2. Floating-point multiplications are frequently performed in a graphic application, and in the proposed architectures 95% of the gates will not be quiet during computation. In addition static power

consumption is very technology dependent, and may be different for a low-power and a general purpose CMOS process, and may even vary between vendors as well. Because leakage current simulations are performed by [1], and static power consumption by [2] this may enhance the error, and lead to not choosing the best architecture for implementation given a set of area, power and throughput constraints.

2.2 Power Estimation

The architectures presented in [7] have different power consumptions, areas and throughputs. In Table 2.4, the static power dissipation for each of the four architectures is computed, assuming none of the significand multipliers are performing any computation.

$$P_{static} = \begin{array}{l} \#\text{ FP64 multipliers} \times P_{static}(\text{FP64}) + \\ \#\text{ FP32 multipliers} \times P_{static}(\text{FP32}) + \\ \#\text{ FP16 multipliers} \times P_{static}(\text{FP16}) \end{array} \tag{2.1}$$

The values in Table 2.4 are computed according to Equation 2.1, and the values are normalized to architecture four.

Architecture	Static Power	Normalized Static Power
One	345168.0	3.0
Two	279856.8	2.4
Three	144581.4	1.2
Four	116578.8	1.0

Table 2.4: Static power estimation of proposed architectures.

The total power consumption is given by both the static power consumption and the dynamic power consumption, where the dynamic power consumption is given by

$$P_{dynamic} = \begin{array}{l} \#\text{ FP64 multipliers} \times P_{dynamic}(\text{FP64}) + \\ \#\text{ FP32 multipliers} \times P_{dynamic}(\text{FP32}) + \\ \#\text{ FP16 multipliers} \times P_{dynamic}(\text{FP16}) \end{array} \tag{2.2}$$

, and the total power consumption given by

$$P_{total} = P_{static} + P_{dynamic} \tag{2.3}$$

The methodology presented in Section 2.1, is a simplified and inaccurate methodology. However, the relative differences between the architectures

evaluated in [7] and described in Section 1.5 are well highlighted through this simple methodology. The static and dynamic power consumption computed in Table 2.2 and Table 2.3 are used to compute the total significand multiplier power consumption for each of the four architectures. In Table 2.5 and 2.6, the total power dissipated per cycle, and total power dissipated per multiplication are computed for the different supported formats. Total power per multiplication is important because if the input bus is reduced to 128-bit, two cycles are needed to compute the significands of an entire input vector for architecture one and two. It is also important to consider how input data format affects power dissipation of the four architectures, because the input data distribution is unknown. It can only be assumed that the FP32 format are frequently used compared to the FP16 and FP64 format. This knowledge may be important when choosing architecture. Total power includes both static and dynamic power dissipation, where the values are normalized to the purely static power dissipation of architecture four.

Data format	Architecture	Total Power	Normalized Power per Cycle	Normalized Power per Multiplication
FP16	One	388596.0	3.33	3.33
	Two	932856.0	8.00	8.00
	Three	155438.4	1.33	5.33
	Four	388596.0	3.33	13.33
FP32	One	563097.6	4.83	4.83
	Two	932856.0	8.00	8.00
	Three	199063.8	1.71	6.83
	Four	388596.0	3.33	13.33
FP64	One	889202.4	7.63	7.63
	Two	823891.2	7.07	7.07
	Three	416598.6	3.57	14.29
	Four	388596.0	3.33	13.33

Table 2.5: Total power consumption, 256-bit input vector.

Figure 2.3 displays the differences in power consumption per cycle, and power consumption per multiplication for the four architectures. It shows that in addition to reducing the overall chip area, reducing the input bus also reduces the power dissipated each clock cycle for architecture one and two. The total power dissipated by architecture three and four is unchanged by reducing the input bus. This is because the amount of computational units are not reduced as for architecture one and two. However, even if total power consumption per cycle is reduced for architecture one and two, the power consumption per multiplication is not reduced because an additional

Data format	Architecture	Total Power	Normalized Power per Cycle	Normalized Power per Multiplication
FP16	One	194298.0	1.67	3.33
	Two	466428.0	4.00	8.00
	Three	155438.4	1.33	5.33
	Four	388596.0	3.33	13.33
FP32	One	281548.8	2.42	4.83
	Two	466428.0	4.00	8.00
	Three	199063.8	1.71	6.83
	Four	388596.0	3.33	13.33
FP64	One	444601.2	3.81	7.63
	Two	411945.6	3.53	7.07
	Three	416598.6	3.57	14.29
	Four	388596.0	3.33	13.33

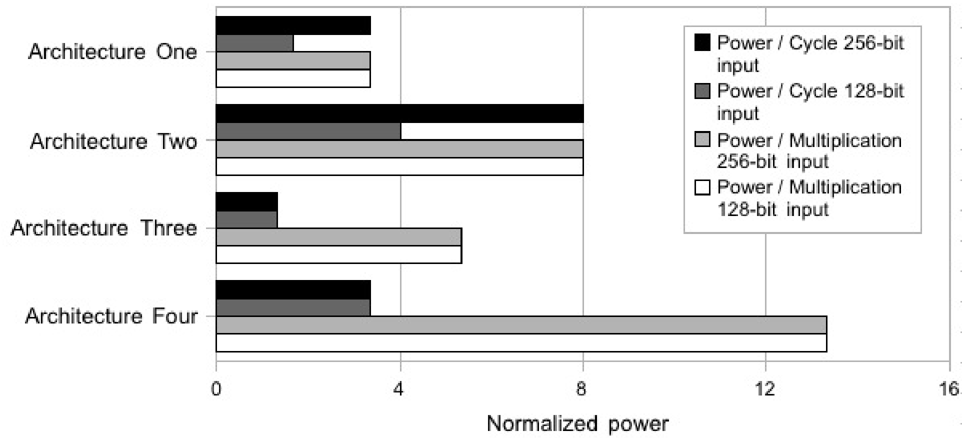
Table 2.6: Total power consumption, 128-bit input vector.

cycle is needed to compute an entire 256-bit input vector.

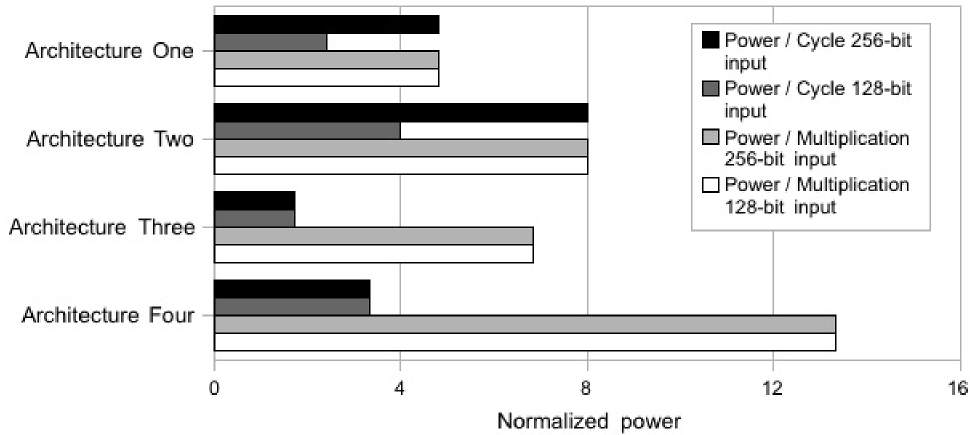
The relative differences in power consumption of the four architectures are well highlighted in Figure 2.3. Architecture three and four dissipates the least amount of power per cycle, but suffers from high total power consumption when computing an entire input vector compared to architecture one and two. Because only one product is computed each cycle, four cycles are needed to compute an entire input vector. Architecture one dissipates slightly more power than architecture three per cycle assuming a 128-bit input bus, but has significantly lower total power consumption when an entire input vector is considered. Architecture one has lowest power consumption per multiplication for all data formats, except FP64. Because the FP32 format is assumed to be the most used data format this should be an important consideration when choosing the architectures to implement. Total power consumption per multiplication is more important to consider than power dissipation per cycle. Because the rounding and exception logic, which is a significant part of the architectures, are not considered when computing power consumption, the relative differences may be greater or smaller.

2.3 Area Estimation

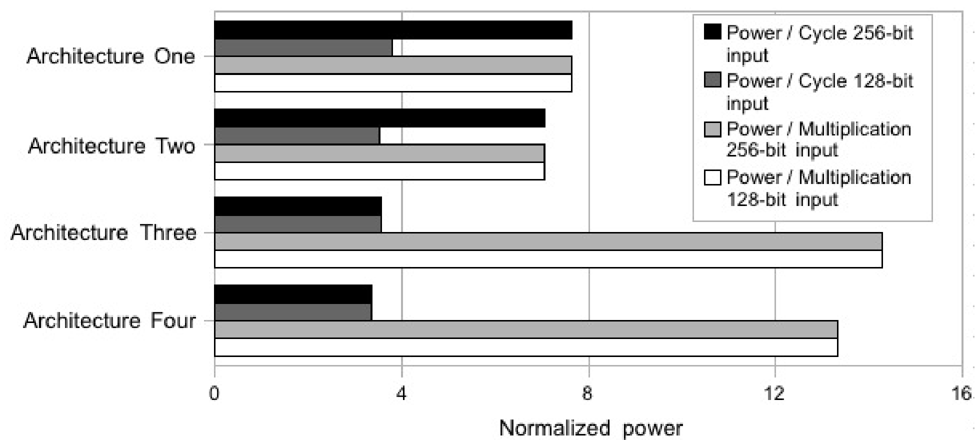
Area estimations are performed following the methodology described in [7]. Number of Full-Adders and equivalent 1-bit register cells are used to compute the total area requirements. Control logic and additional computational logic



(a) Power estimation, only FP16 input data.



(b) Power estimation, only FP32 input data.



(c) Power estimation, only FP64 input data.

Figure 2.3: Architecture power comparison.

requires little area compared to the significand multipliers, exponent adders and registers. The rounding logic differs somewhat for the architectures evaluated. Architecture one and two requires additional rounding logic due to parallel computing of product vectors. The ratio of transistors required by the Full-Adder model in Figure 2.1 and the register model presented in [7] is given by Equation 2.4.

$$transistor\ ratio = \frac{\#\ transistors\ in\ FA}{\#\ transistors\ in\ register} = \frac{40}{36} = \underline{1.11} \qquad (2.4)$$

Architecture	Input-bus	# FA-cells	Eq. register-size	# Transistors
One	256-bit	8160	924	9990.7
	128-bit	4080	530	5063.3
Two	256-bit	6616	1134	8485.1
	128-bit	3308	635	4310.6
Three	256-bit	3418	612	4409.8
	128-bit	3418	484	4281.8
Four	256-bit	2756	612	3674.2
	128-bit	2756	484	3546.2

Table 2.7: Architecture area comparison, FA-cells and equivalent register-size.

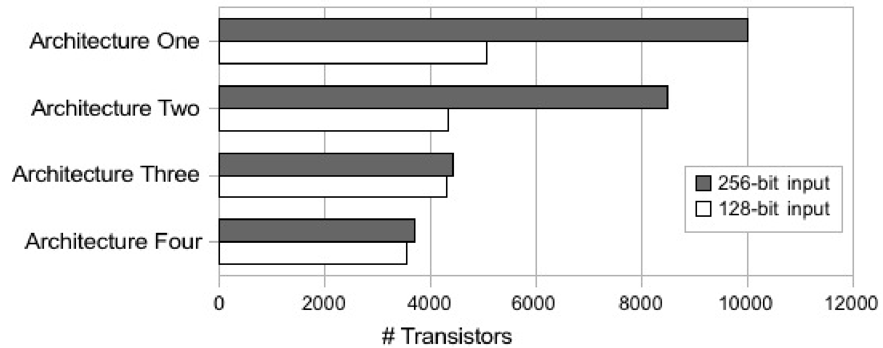


Figure 2.4: Architecture area comparison.

Figure 2.4 illustrates the area usage of the different architectures as a function of required transistors as presented in Table 2.3. Figure 2.4 shows that for an 256-bit input bus, architecture one requires more than twice as much transistors as architecture three and four, and architecture two approximately almost twice as much as architecture three. For an 128-bit input bus,

the relative differences are much smaller, and not more than approximately 1000 transistors. Area reduction of architecture one and two is large compared to architecture three and four, because number of computational units such as significand multipliers and exponent adders are reduced, while only number of equivalent 1-bit register cells is reduced in architecture three and four. From an area point of view, an 128-bit input bus is favored.

Because logic not included in this area estimation methodology differs somewhat for the different architectures, this is a source of error. The largest computational unit not considered in this methodology are the rounding and exception unit, and because this unit is larger, and equal, for architecture one and two compared to architecture three and four, the differences will be greater than displayed in Figure 2.4. However, the relative difference in area usage by the proposed architectures are still well highlighted because the rounding and exception logic are small compared to the significand multipliers.

2.4 Performance Estimation

Performance is measured by clock frequency and data processed each clock cycle. The maximum clock frequency will be approximately equal for all architectures, and determined by the critical path delay. The clock frequency is given by the inverse of the delay trough the 53-bit significand multiplier.

The data processed each cycle, or the throughput, is determined by the ability to process data in parallel. The architectures described in [7] have different throughputs and latencies. Throughput is measured in how many products computed each clock cycle. Reducing the input bus also reduces the throughput for architecture one and two, but not for architecture three and four. The ARM 3D graphic solutions typically runs at 300*MHz* clock frequency. Assuming a clock frequency of 300*MHz*, and a 256-bit input bus, the throughput of architecture one and two will be

$$256\ bit \times 300\ MHz = 76800\ \frac{Mbit}{s},$$

and for architecture three and four

$$64\ bit \times 300\ MHz = 19200\ \frac{Mbit}{s}$$

If the input bus is reduced to 128-bit, the throughput of architecture one and two will become

$$128\ bit \times 300\ MHz = 38400\ \frac{Mbit}{s},$$

and for architecture three and four the throughput will be unchanged.

The computations above shows that architecture one and two have higher throughput than architecture three and four. However, if the input bus is reduced to 128-bit, architecture one and two still have higher throughput, but reduced by 50% compared to an 256-bit input bus, while the throughput of architecture three and four remains the same.

Latency is in this context defined as the number of clock cycles from a vector arrives at the input to the product vector are ready at output. The latencies for the different architectures are given in Figure 2.5.

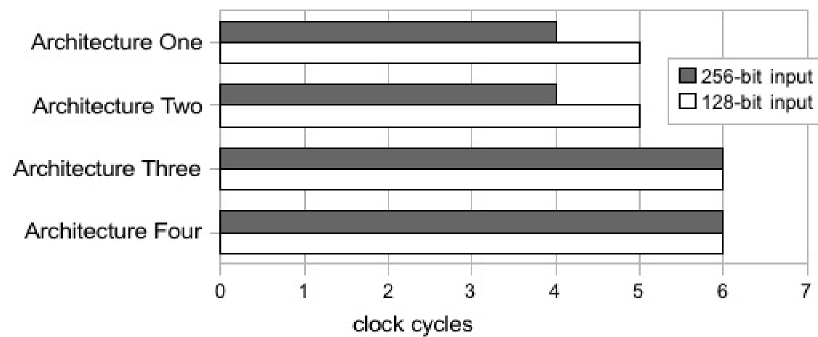


Figure 2.5: Architecture latency comparison.

The delay through the 53-bit significand multiplier is equal to the inverse of the delay through 106 full-adder cells, assuming the multiplier is implemented as an array multiplier. For a typical low-power $65nm$ CMOS process the delay through one full-adder cell equals $0.11ns$, which gives a maximum clock frequency of $90.9MHz$. To achieve higher clock frequencies, the significand multipliers must be implemented using a faster multiplier scheme. The Dadda or Wallace multiplier, with or without Booth recoded input will achieve this as described in Section 1.3. In the power and area estimations, significand multipliers are assumed implemented as array multipliers. However, changing the significand multiplier scheme does not changes the relative difference between the architectures, as long as the change is equal for all four architectures.

2.5 Trade-Off Considerations

When choosing the architecture to implement, design constraints have to be considered. Because throughput is very important in a graphic application, throughput should be kept as high as possible. In a handheld, battery pow-

ered device, area and power are also very important. Hence, the decision of which architectures to implement should be a trade-off between area, power and throughput. A weight-function could be used to help the decision, where area usage, power consumption and throughput are weighted according to importance. But, because total power consumption has a static and a dynamic component, where the dynamic component are dependent of which format being computed, and the static power component directly related to area usage, the weight-function can become complex. In addition, data format distribution may vary from user to user, which makes the decision even harder. However, the FP32 format is expected to be the most used data format. Thus, this should be weighted as more important than the FP16 and FP64 formats.

Because of error sources in the area and power estimation methodologies, such as logic not considered and the assumptions of quiet gates in the static power consumption calculation as described in Section 2.1, this should be kept in mind when choosing architecture. Because of the error sources in the estimation methodologies, two architectures should be implemented and compared to see how well the area and power methodologies predicted the relative differences in area usage and power consumption.

Chapter 3

Implementation

An IEEE compliant, pipelined, vectorized floating-point multiplier is to be implemented RTL for testing and synthesis. In Section 3.1, two architectures are selected for implementation based on the analysis and trade-off considerations performed in Chapter 2. Section 3.2 presents the implemented architectures, describes the differences between them, and provides user information. In Section 3.3, testing are discussed. Section 3.3 describes the testing and simulation, and what have been tested.

3.1 Choosing Architecture

The width of the input bus affects area usage, power consumption and throughput for the evaluated architectures. Area and power consumption can be significantly reduced, if the input bus is reduced from 256-bit to 128-bit. However, this lowers the throughput and increases the latency. The total power consumption by computing an entire input vector does not change, if the input bus is reduced from 256-bit to 128-bit, following the assumptions made in the methodologies presented in Chapter 2. The total energy consumption may be reduced somewhat if the input data is highly correlated, however this can not be assumed. By reducing the input bus both area and power consumption are reduced significantly for architecture one and two. Area is slightly reduced for architecture three and four as well. Table 3.1, presents a summary of estimated area usage, power consumption, latency and throughput (at $300MHz$) for architecture one, two, three and four, assuming 128-bit input. Power is presented for only FP16 computations, only FP32 computations and only FP64 computations, where power dissipated by computing an entire input vector is considered.

Total power consumption by computing an entire input vector and area is the most important criteria when choosing an architecture to implement, in addition to the throughput. But, as seen from Table 3.1, dissipated power

	Architecture			
	One	Two	Three	Four
Area	5063.3	4310.6	4281.8	3546.2
FP16 Power	3.33	8.00	5.33	13.33
FP32 Power	4.83	8.00	6.83	13.33
FP64 Power	7.63	7.07	14.29	13.33
Throughput	38400	38400	19200	19200
Latency	5	5	6	6

Table 3.1: Trade-off considerations.

is dependent of which format being computed, and input data format distribution should be considered when choosing architecture. Architecture one has lower power consumption than the other architectures for only FP16 and FP32 computations. But when only FP64 computations are performed, architecture two has lower power consumption than architecture one. This is because of static power dissipated by the significand multipliers in architecture one. If static power is modeled to high, architecture one might have lower power consumption than architecture one for only FP64 computations as well. Architecture one and two have lower latency and higher throughput than architecture three and four. Architecture one and two have larger area than architecture three and four, but architecture four suffers from significantly higher power consumption. Architecture three has higher power consumption than architecture one for all input formats, but lower than architecture two for FP16 and FP32 input data.

Based on the analysis above, and the estimations performed in Section 2.2, 2.3 and 2.4, the input bus should be 128-bit and architecture one should be implemented to minimize the trade-off between area and power consumption, while keeping a relatively high throughput. Because only power dissipated in the multipliers are considered, and the sources of error discussed in Chapter 2, the differences may be greater or less due to power dissipated in registers, logic not considered, and fan-out effects in multiplexers. To see if the analysis made in Chapter 2 are accurate enough to make a correct implementation decision, given a set of constraints, architecture one and two should be implemented and compared concerning area and power.

3.2 Vectorized Floating-Point Multiplier

Two partially IEEE compliant, vectorized floating-point multipliers have been implemented. Architecture one and two was selected for implementation in RTL. The vectorized floating-point multipliers does not support denormalized inputs. If denormalized input vectors are provided to the

floating-point multiplier, these are treated as zero. Otherwise, the floating-point multiplier complies to the IEEE 754 specifications concerning delivering the correct result and exception generations.

The general block diagram of the vectorized floating-point multiplier top-module is given in Figure 3.1.

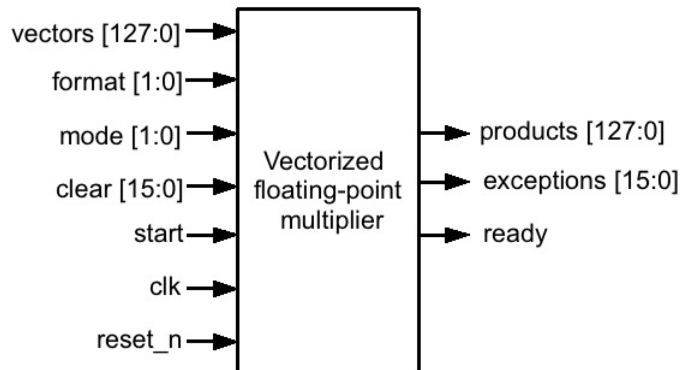


Figure 3.1: Vectorized floating-point multiplier block diagram.

3.2.1 Inputs

The vectorized floating-point multipliers have five inputs, *vectors*, *format*, *mode*, *clear* and *start* in addition to *clock* and *reset* inputs. The *format* input tells the floating-point multiplier which format to compute, FP16, FP32 or FP64, and the *mode* input tells which rounding mode to apply. The *clear* input is used to clear exceptions, and the *start* input tells the floating-point multiplier that vectors are ready at the input. *start* must be kept high as long as input vectors are ready at the input. Input vectors should be given as shown in Figure 3.2 and Figure 3.3.

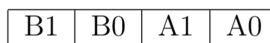


Figure 3.2: First input vector layout.

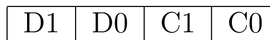


Figure 3.3: Second input vector layout.

Because the input bus is 128-bit and the input vector for the FP32 and FP64 formats are 256-bit, the input vector has to be provided in two cycles

where A0, A1, B0 and B1 should be given in the first cycle, and C0, C1, D0 and D1 should be given in the second cycle. The same has been done for the FP16 format, therefore the upper 64-bits of the input vector should be set to zero when FP16 computations are performed.

Data formats and rounding modes are encoded as given in Table 3.2 and Table 3.3 respectively.

Format	Encoding
FP16	00
FP32	01
FP64	10

Table 3.2: Format encoding.

Mode	Encoding
Round-to-nearest even	00
Round-to-plus infinity	01
Round-to-minus infinity	10
Round-to zero	11

Table 3.3: Rounding modes encoding.

Exceptions should be cleared by setting the correct bits on the *clear* input bus to one. The layout of the the clear register is given in Figure 3.4.

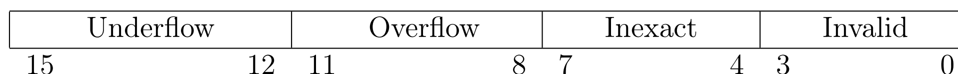


Figure 3.4: Clear register layout.

$invalid[0]$ correspond to the product $A0 \times A1$, $invalid[1]$ to the product $B0 \times B1$, $invalid[2]$ to the product $C0 \times C1$ and $invalid[3]$ to the product $D0 \times D1$. Likewise for the inexact, underflow and overflow exceptions, except the index should be incremented as shown in Figure 3.4. However, this functionality has not been implemented properly, and exceptions are not cleared as specified in the IEEE 754 standard.

3.2.2 Outputs

The vectorized floating-point multiplier has three outputs, *products*, *exceptions* and *ready*. The ready output is set to one whenever a product vector is ready at the output. Products are laid out as given in Figure 3.5.

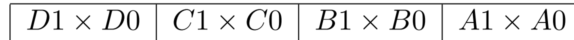


Figure 3.5: Product vector layout.

The exception layout is exactly the same as the clear register layout, and as given in Figure 3.6.

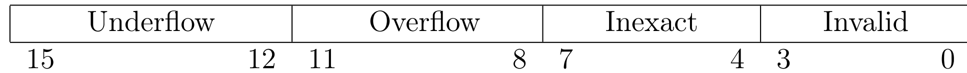


Figure 3.6: Exception register layout.

$invalid[0]$, $inexact[4]$, $overflow[8]$ and $underflow[12]$ corresponds to the product $A0 \times A1$. Exceptions for the products $B0 \times B1$, $C0 \times C1$ and $D0 \times D1$ are found by incrementing the index.

A typical scenario with only one input vector pair is given in Figure 3.7.

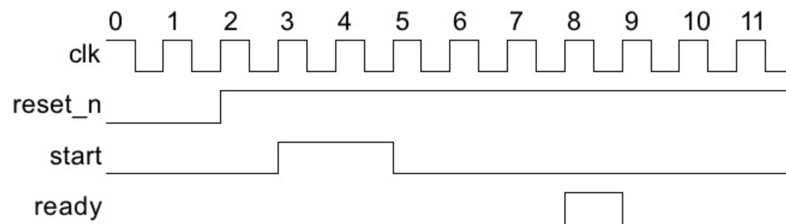


Figure 3.7: Vectorized floating-point multiplier simple timing diagram.

3.2.3 Architecture Description

The architectures have some minor changes from the ones described in [7]. These changes does not affect the relative differences between the area, power and performance estimations performed in Chapter 2 of the two implemented architectures. Figure 3.8 shows a more detailed architecture diagram than provided in [7].

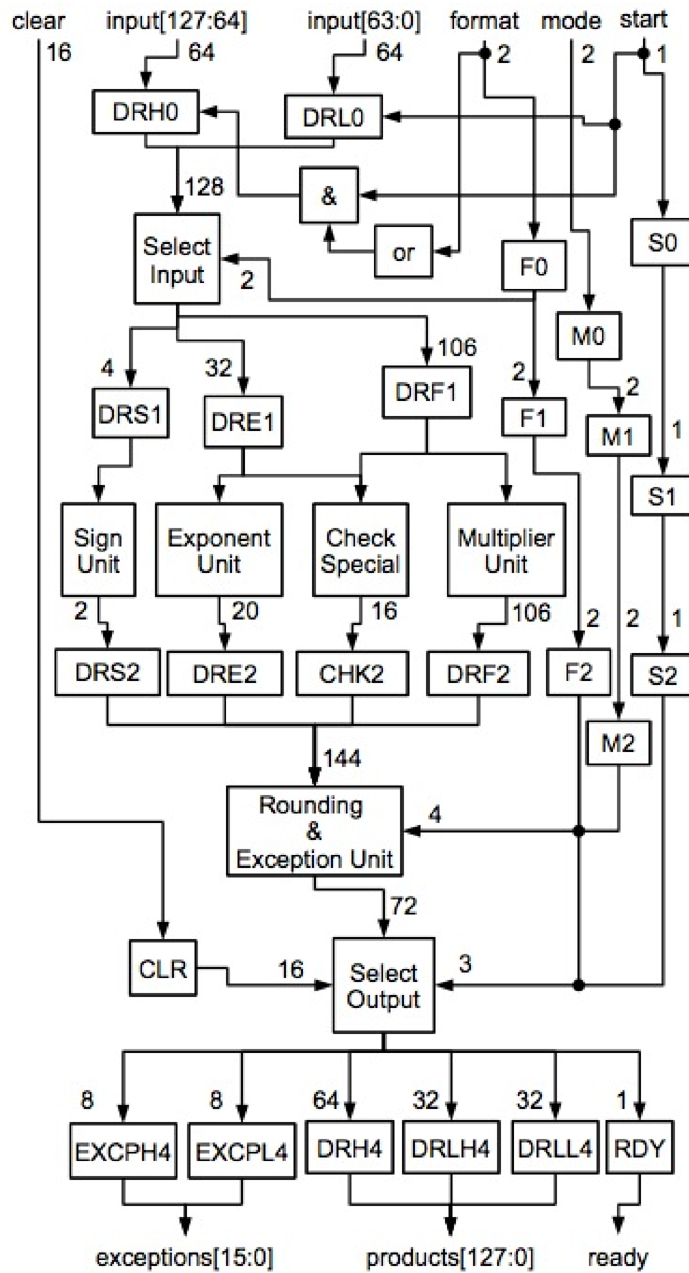


Figure 3.8: Vectorized floating-point multiplier architecture drawing.

Building Blocks

The major building blocks in the design are the *select input* demultiplexer, the *sign unit*, *exponent unit*, *multiplier unit*, *check special unit*, *rounding and exception unit* and the *select output* demultiplexer.

The *select input* demultiplexer provides the *sign unit*-, *exponent unit*- and *multiplier unit*- registers with correct data, and selects parts of the input registers based on which data format being computed. The *sign unit*, *exponent unit* and *multiplier unit* computes the resulting signs, exponents and significands respectively. The *check special unit* checks for special inputs like NaNs, infinities and zeroes used by the *rounding and exception unit* to generate correct result and exceptions. The *select output* demultiplexer select which part of the exception registers and output registers to load in addition to setting the correct value of the ready register. These units are equal for both implemented architectures.

There are some differences between the architectures of the two implementations. In architecture two, the *exponent unit* and the *multiplier unit* needs to know which format being computed, in addition to the actual content of the building blocks as described in Section 1.5. As described in [7], the bus width, and register size, between the multiplier unit and the computed significands register changes between the two architectures. In architecture one this is 106-bits, and in architecture two this is 154-bits because the products of the 53-bit significand multiplier and the 24-bit significand multiplier equals 154-bit. This also infers a wider bus between the computed signs, exponents and significands registers and the rounding and exception unit. The exception and rounding unit differs for the two architectures, and will be discussed later.

Exponent Unit

In the exponent unit, exponent adders and subtractors are implemented using carry-lookahead adders in the DesignWare® library form synopsys [17]. One adder computes the sum of the two exponents, and a subtractor computes the sum minus the bias. The exponent unit are different in architecture one and two. In architecture one, one 11-bit adder and subtractor, two 8-bit adders and subtractors and two 5-bit adders and subtractors are used for computing the resulting exponents of the FP64, FP32 and FP16 formats respectively. The exponent unit of architecture one is given in Figure 3.9.

In architecture two, one 11-bit adder and subtractor and one 8-bit adder and subtractor are used to compute the exponents. The 11-bit subtractor supports subtraction of all FP16, FP32 and FP64 bias values, and the 8-bit

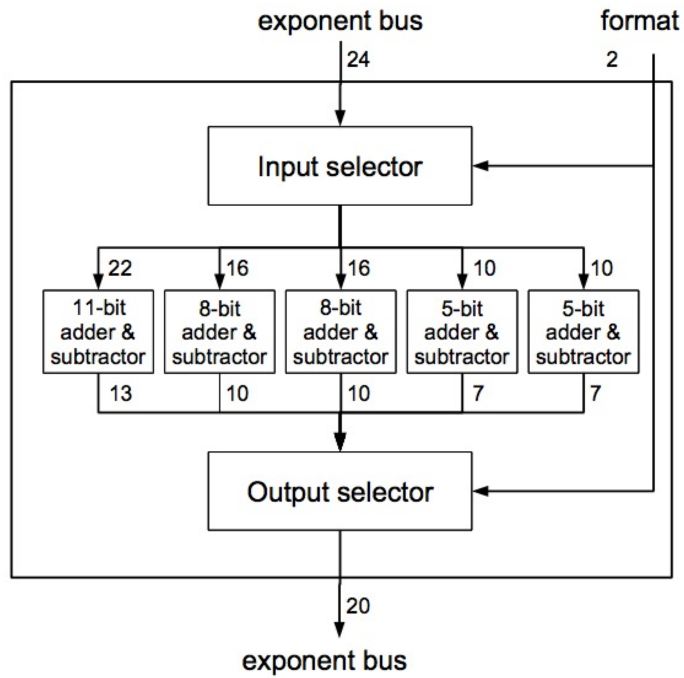


Figure 3.9: Architecture one exponent unit.

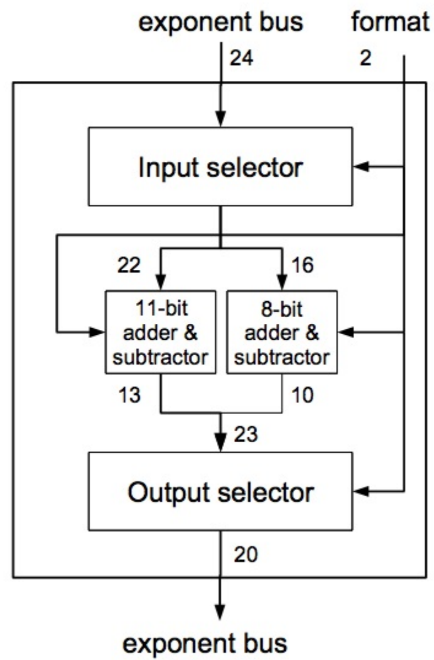


Figure 3.10: Architecture two exponent unit.

subtractor supports subtraction of FP16 and FP32 bias values. Exponent unit of architecture two is given in Figure 3.10.

The output bus from the exponent unit includes four extra bits in addition to the actual exponents. These are overflow bits from the exponent additions and bias subtractions used by the rounding and exception unit to generate correct exceptions, and are equal for both architectures. An input demultiplexer selects the input bits from the exponent register to supply the correct adder, and an output multiplexer puts the result from correct adder on the output bus.

Multiplier Unit

The significand multipliers are implemented as unsigned parallel-prefix multipliers provided by the DesignWare® datapath and building block IP library [18] from Synopsys to obtain higher clock frequencies than 90.9 MHz as estimated in Section 2.4. This implementation is flexible, and dynamically generated based on context, e.g., area and timing constraints, and technology library. It exploits the characteristics of different implementations and generates the optimal architecture [19]. The content of the multiplier unit differs for the two architectures. In architecture one, one 53-bit, two 24-bit and two 11-bit unsigned multipliers are used to compute the significands of the FP64, FP32 and FP16 data formats respectively. Architecture one significand multiplier unit are given in Figure 3.11.

In architecture two, one 53-bit multiplier and one 24-bit multiplier are used to compute the significands. The 53-bit multiplier are used to compute the resulting significand of all formats, and the 24-bit multiplier is used to compute the resulting significand of the FP32 and FP16 formats. The significand multiplier unit in architecture two are given in Figure 3.12.

An input demultiplexer selects which bits should go to which multiplier. In architecture one, an output multiplexer selects which multiplier group result FP16, FP32 or FP64 should be put on the output bus. In architecture two, the significands are extended to fit the width of the 53-bit and 24-bit multiplier input buses. Zeroes are appended as least significant bits to avoid shifting or demultiplexing in the rounding and exception unit.

Rounding and Exception Unit

In the original architecture proposals in [7], the rounding and exception unit is equal for all four architectures. However, this has been implemented differently in architecture one and two to better highlight the differences between them concerning power. In architecture one, specialized rounding

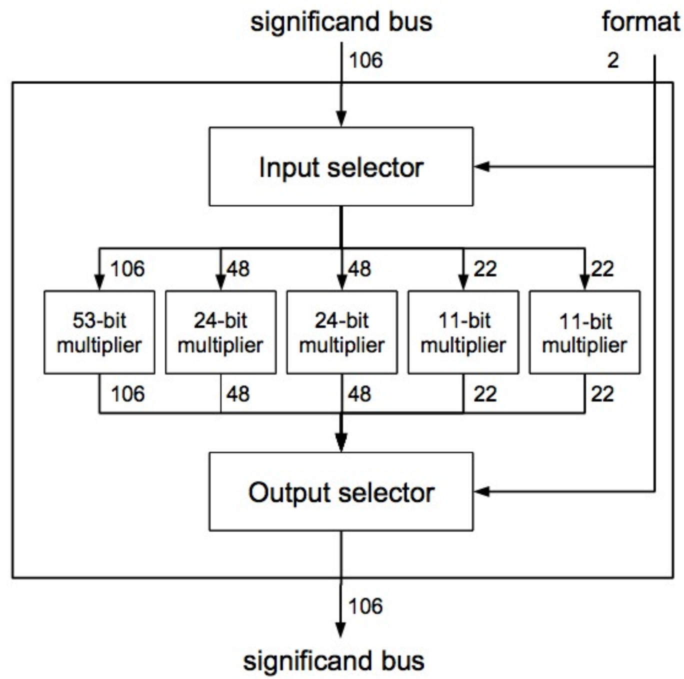


Figure 3.11: Architecture one significand multiplier unit.

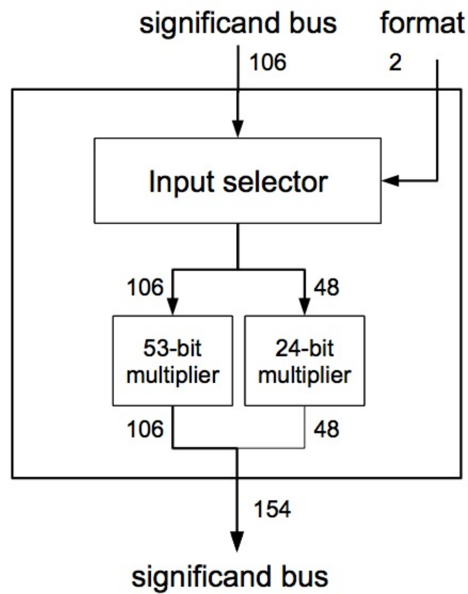


Figure 3.12: Architecture two significand multiplier unit.

units for each format are used, as for the multiplier unit and exponent unit. One rounding and exception block handling the FP64 format, two rounding and exception blocks handling the FP32 format and two handling the FP16 format. The rounding and exception unit in architecture one is given in Figure 3.13. In architecture two, one rounding and exception block handles every format, and one handling the FP32 and FP16 formats. A simple rounding algorithm has been implemented. The rounding and exception unit of architecture two is given in Figure 3.14.

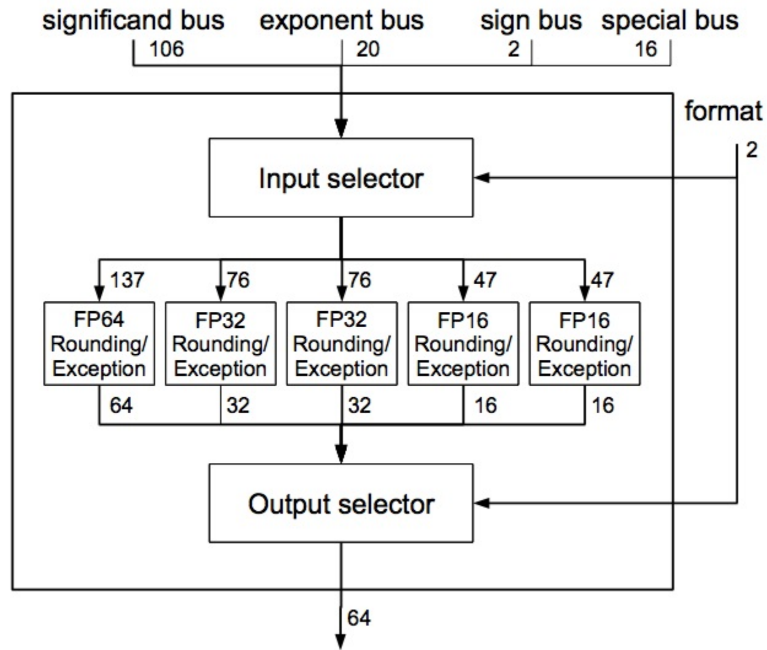


Figure 3.13: Architecture one rounding and exception unit.

The implemented rounding algorithm is basically the same as the one presented in Section 1.1, except a demultiplexer is used in the post-normalizing step to select the appropriate significand bits as the simple algorithm presented in [20]. Rounding could be performed faster and more efficiently, if for example the QFT algorithm presented in [20] is used. However, this requires a significand multiplier that outputs the sum and carry vectors as separated carry-save encoded vectors. The four rounding modes, round-to-nearest even, round-to positive infinity, round-to negative infinity and round-to zero have been reduced to three, round-to-nearest even, round-to infinity and round-to zero as in [21]. Round-to positive infinity, round-to negative infinity and round-to zero can be reduced to round-to infinity and round-to zero based on the sign as given in Table 3.4.

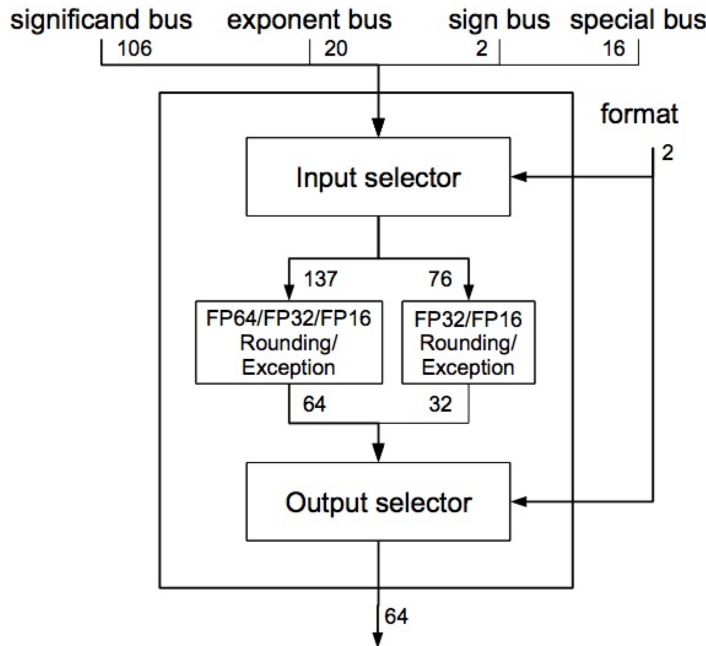


Figure 3.14: Architecture two rounding and exception unit.

IEEE Rounding mode	Positive Number	Negative Number
Round-to-nearest even	Round-to-nearest even	
Round-to positive infinity	Round-to infinity	Round-to zero
Round-to negative infinity	Round-to zero	Round-to infinity
Round-to zero	Round-to zero	

Table 3.4: Rounding mode reduction.

In the rounding and exception unit, a demultiplexer supplies the different rounding and exception blocks with signs, exponents and significands computed in their respective units from their registers, as well as information computed in the check special unit.

3.3 Testing and Simulation

Testing have been performed using the open source Verilog simulator and synthesis tool, Icarus Verilog [22]. Test cases have been generated using the C-code in Appendix C. “Random” floating-point numbers are created, and special values are included “randomly” to ensure simulation of exceptional cases like NaN times any number, and zero times infinity. 500,000 test cases have been simulated for both architectures and for all supported data formats and rounding modes.

3.3.1 Reference Circuit

The DesginWare® library from Synopsys provides a simulation model of a fully IEEE compliant floating-point multiplier [23, 24]. This has been used to create a vectorized version, that computes four products in parallel. The block diagram of the DesignWare vectorized floating-point multiplier is given in Figure 3.15.

The Verilog code for the DesignWare vectorized floating-point multiplier can be found in Appendix D.1. In addition, because the DesignWare floating-point multiplier supports denormalized numbers, the output is set to zero if denormalized product, and an inexact exception is generated. The correctness of the DesignWare vectorized multiplier can easily be verified by looking at the code.

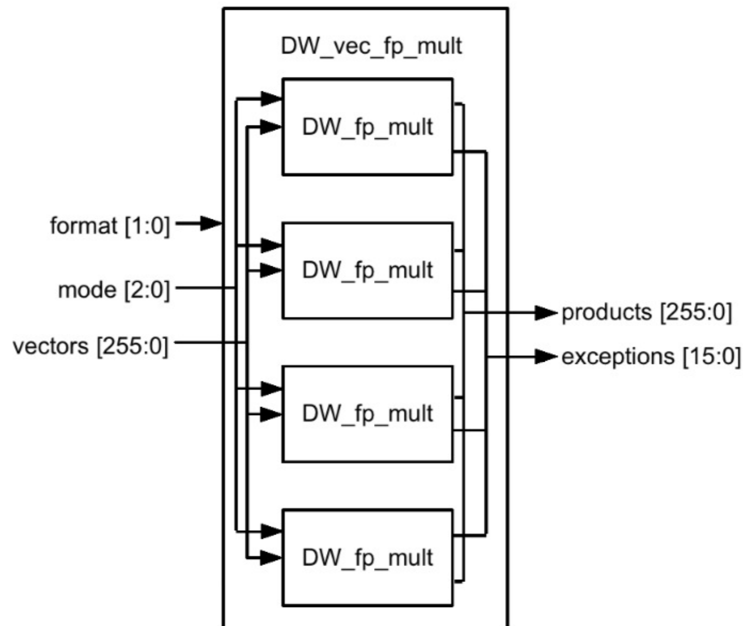


Figure 3.15: DW_vec_fp_mult block diagram.

3.3.2 Simulations

Whenever a product is ready, the computed product and exceptions is compared to the product vector and exceptions computed by the DesignWare floating-point multiplier. The testbench used for simulating the two architectures can be found in Appendix D.2.

Both architectures have been tested with FP16, FP32 and FP64 input vectors. For each format, the rounding modes round-to-nearest even, round-

to positive infinity, round-to negative infinity and round-to zero have been tested with 500,000 test cases. The testbench does not try to change data format or rounding mode during simulation time, however this is believed to work. To verify correct operation of the implemented architectures, this has to be tested. However, the emphasis of this assignment is not verification, but rather highlighting the differences between the architectures concerning power and area. The testbench used for simulation prints statistics about input vectors, output vectors and exceptions generated when finished to ensure every exceptional cases have been covered by input vectors during simulation. In addition, behavioral testing has been performed at module level, to ensure correct behavior of lower level modules such as rounding and exceptions unit, exponent unit, demultiplexers etc.

One error in the rounding unit has been detected with the FP16 format in round-to-nearest even and round-to positive infinity mode. This error is believed to be format independent, but has not been detected when the FP32 and FP64 formats have been tested. The error arises in post-normalization when result should be rounded to the smallest representable normalized number, but is flushed to zero instead. This error has not been corrected because the emphasis of this thesis lies on power and area comparison, and choosing the best architecture to implement, given a set of constraints. The correction of this error will probably not infer any significant increase in area nor power consumption.

Chapter 4

Synthesis Results

Synopsys Design Compiler^{TM} [25] and Power compiler^{TM} [26] are used to synthesize the designs, and perform area, timing and power analysis. A typical general purpose low-power standard cell library is used to map the design into a 65*nm* technology, and a general purpose standard cell library to map the design into a 90*nm* technology. Because the 65*nm* library is a low-power library, and the 90*nm* library is a general purpose library, somewhat different power results are expected. However, this will in addition highlight the differences in which target technology the architectures are realized in.

This Chapter will first present how Synopsys Design Compiler^{TM} and Power Compiler^{TM} calculates power, how to capture switching activity in the implemented architectures, and how design constraints are set to optimize the result, in Section 4.1. The power consumption and area usage of architecture one is presented in Section 4.2. In Section 4.3 power consumption and area usage of architecture two will be presented. In Section 4.4 the power consumption of the two architectures will be compared, and in Section 4.5 area usage of the two architectures will be compared.

4.1 Synopsys®

It is important to understand how Synopsys models and computes power to obtain useful information from the synthesis reports. The following describes how static and dynamic power is computed and are taken from the Power Products Reference Manual [27] by Synopsys. The power analysis tool calculates and reports power based on equations given in [27]. DesignPower and Power Compiler^{TM} use these equations and information modeled in the technology library to evaluate the power of the design.

4.1.1 Static Power

Static power is the power dissipated by a gate when it is not switching. It is dissipated in several ways, mostly due to source-to-drain leakage currents caused by reduced threshold voltages preventing the gate from completely turning off. Other currents leaks also contributes, and hence it is often called leakage power. For designs that are active most of the time, leakage power is less than 1% of the total power.

4.1.2 Dynamic Power

Dynamic power dissipates when the circuit is active. Dynamic power has two sources, internal power and switched power. Internal power is any power dissipated within the boundary of a cell. During switching, a circuit dissipates internal power by the charging or discharging of any existing capacitances internal to the cell. The definition of internal power includes power dissipated by a momentary short circuit between the PMOS and NMOS transistors of a gate, called short circuit power. The switching power of a driving cell is the power dissipated by the charging and discharging of the load capacitance at the output of the cell. The total load capacitance at the output of a driving cell is the sum of the net and gate capacitances on the driving output.

4.1.3 Capturing Switching Activity for Synthesis

Synopsys provides several ways of including simulated switching activity into the power calculations. These are described in the Power Products Reference Manual [27]. The testbench used for capture the switching activity of the different nets in the two architectures are given in Appendix D.3. This testbench has been used for simulating typical switching activity, only FP16 computations switching activity, only FP32 switching activity and only FP64 switching activity. When typical switching activity is captured, FP32 computations are assumed to be performed 60% of the time, FP16 computations 20% of the time and FP64 computations 20% of the time. This distribution is chosen to ensure switching in all nets and registers, and is not given by ARM or any other. But, the FP32 format has been indicated to be the main format used in computations. To capture switching activity, the method described in Power Products Reference Manual Appendix B has been used. The function `rtl2saif` creates a switching activity file (SAIF) from the Verilog RTL design files in the Synopsys *dc_shell*. *dc_shell* is the Synopsys tools command line interface. The UNIX utility `saif2trace` is used to create a forward-annotation trace file based on the information about non-combinational and combinational elements in the SAIF file. This file is included in the testbench to generate switching information as a value change dump file (VCD) of the different design elements. The

VCD file is converted to a backward-annotation SAIF file by the UNIX utility `vcd2saif`, that uses the `set_switching_activity` command in *dc_ shell* to set the static probability and toggle rate for elements in the design. The backward-annotation SAIF file is read in the *dc_ shell* before compilation by the function `read_saif`, which incorporates information about switching activity into the compilation and optimization process performed by the Design Compiler™ and Power Compiler™.

4.1.4 Setting Design Constraints

Area and power constraints are set by the *dc_ shell* commands `set_max_area`, and `set_max_total_power`. Maximum dynamic power and leakage power may be set individually by the commands `set_max_dynamic_power` and `set_max_leakage_power`, respectively. Timing constraints can be set by the `set_max_transition` command from input ports or pins to output ports or pins. However, if the design is clocked, Design Compiler™ assumes single cycle datapaths between registers and the `create_clock` command can be used to set timing. To synthesize and optimize the design the `set_max_area` and `set_max_total_power` have been set to zero. To set timing constraints of combinational logic between registers, the `create_clock` command has been used.

Architecture one and two have been synthesized for $200MHz$, $300MHz$ and $400MHz$ clock frequency, and for different input data format distributions, as described in Section 4.1.3. When simulating switching activity, all four rounding modes have been simulated for each format to capture switching in every register and combinational units. The `compile_ultra` command from the *dc_ shell* enables the Design Compiler Ultra optimizations available from Synopsys as described in [28], which, i.a., includes advanced arithmetic optimization and obtains better quality of result for timing and area. Design Compiler Ultra and Power Compiler works side by side. Power Compiler optimizes for timing, area and power simultaneously and includes switching activity information to obtain better results concerning power.

4.2 Architecture One

Architecture one attempts to be a power optimized vectorized floating-point multiplier. In this Section, the area usage and power consumption of this architecture, realized in $65nm$ and $90nm$ CMOS, will be investigated. Power units are given in mW, and area units in μm^2.

4.2.1 Power

Table 4.1 and 4.3 presents internal-, switching-, leakage- and total power dissipated by architecture one in $65nm$ and $90nm$ CMOS technology respectively, with typical input data distribution, as described in Section 4.1.3 at $200MHz$, $300MHz$ and $400MHz$ clock frequency. Table 4.2 and 4.4 shows which part of the circuit that dissipates the largest amount of power.

Clock frequency	Power			
200 MHz	Internal	1.0800	85.17	% of dynamic power
	Switching	0.1880	14.83	% of dynamic power
	Leakage	0.0185	1.44	% of total power
	Total	1.2860	100.00	% of total power
300 MHz	Internal	1.6190	85.30	% of dynamic power
	Switching	0.2790	14.70	% of dynamic power
	Leakage	0.0228	1.19	% of total power
	Total	1.9210	100.00	% of total power
400 MHz	Internal	2.1700	84.83	% of dynamic power
	Switching	0.3880	15.17	% of dynamic power
	Leakage	0.0286	1.11	% of total power
	Total	2.5870	100.00	% of total power

Table 4.1: Architecture one, $65nm$ CMOS total power consumption.

From Table 4.1 it can be seen that, in $65nm$ low power CMOS, leakage power is much less than estimated, on average 1.25% of total power compared to the estimated value of 30%. This is because no idle simulation has been performed as in [2], and because target library is optimized for low power. The major power component, internal power, is due to charging and discharging of capacitive loads internal to the cells, where the cells represents the instantiated Verilog modules. The average increase in total power consumption equals $0.6505mW/100MHz$. From Table 4.2 it can be seen that over 85% of total power is consumed by registers in the $65nm$ circuit. Significand multipliers only accounts for 4.63% of total power on average. This is a surprising result, which contradicts the assumptions made in the power estimation methodology, that the significand multipliers are the most power consuming units in the design. However, this result is partially because of datapath optimizations performed by the Synopsys tools, it is also possible that the sequential elements are not optimized for low power in the same manner as the datapath elements. This should be investigated further.

From Table 4.3 it can be seen that, in $90nm$ CMOS, power consumption is much larger than in $65nm$ CMOS. All power components are increased in size, internal, switching and leakage. The most important increase are the

Clock frequency	Power	% of total power
200 MHz	Registers	87.6
	Multiplier Unit	4.3
	Rounding Unit	0.7
	Select Output	3.0
	Select Input	3.9
300 MHz	Registers	86.9
	Multiplier Unit	4.7
	Rounding Unit	0.9
	Select Output	3.1
	Select Input	3.6
400 MHz	Registers	86.6
	Multiplier Unit	4.9
	Rounding Unit	0.9
	Select Output	3.0
	Select Input	3.8

Table 4.2: Architecture one, $65nm$ CMOS building blocks power consumption.

increase in ratio of switching power to total dynamic power and the ratio of leakage power to total power. Switching power is on average, at the different clock frequencies, 36% of total dynamic power, and leakage power 9.62% of total power. The large increase in power consumption is partially because the $65nm$ library is a low-power library, and Synopsys Design Compiler™ and Power Compiler™ exploits features in the low-power library to obtain lower power consumption, and hence internal-, switching- and leakage power is reduced. The average increase in total power is $5.1850mW/100MHz$. In Table 4.4 power dissipated by major units, when realized in $90nm$ CMOS, are presented. The results presented in Table 4.4 are more as expected, where the significand multipliers accounts for the larger part of the total power consumption. Approximately 60% of total power is dissipated in the significand multipliers, and approximately 28% by the registers, compared to the $65nm$ results where on average 87% is dissipated in registers and 4.6% in multipliers.

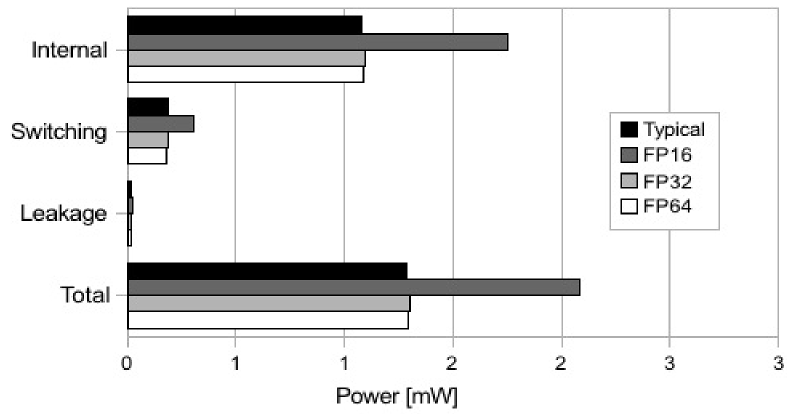
Figure 4.1 shows power consumption of architecture one at $200MHz$, $300MHz$ and $400MHz$ in $65nm$ CMOS and for typical input data distribution, only FP16 input data, only FP32 input data and only FP64 input data. At $200MHz$ a strange case occurs. When only FP16 computations are performed, power consumption is much larger than when the other input data distributions are computed. From the synthesis report it can be seen that the large power consumption is mostly due to high internal and switching power in the 53-bit and one of the 24-bit multipliers. Architecture one

Clock frequency	Power			
200 MHz	Internal	5.5030	64.15	% of dynamic power
	Switching	3.0760	35.85	% of dynamic power
	Leakage	1.1500	11.81	% of total power
	Total	9.7340	100.00	% of total power
300 MHz	Internal	8.9450	64.17	% of dynamic power
	Switching	4.9950	35.83	% of dynamic power
	Leakage	1.4700	9.54	% of total power
	Total	15.4090	100.00	% of total power
400 MHz	Internal	11.8440	63.70	% of dynamic power
	Switching	6.7500	36.30	% of dynamic power
	Leakage	1.5100	7.51	% of total power
	Total	20.1040	100.00	% of total power

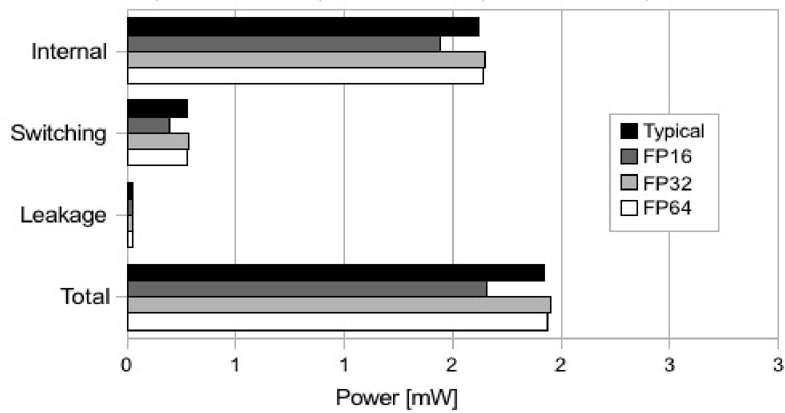
Table 4.3: Architecture one, $90nm$ CMOS total power consumption.

Clock frequency	Power	% of total power
200 MHz	Registers	28.2
	Multiplier Unit	61.1
	Rounding Unit	7.0
	Select Output	1.0
	Select Input	1.2
300 MHz	Registers	28.4
	Multiplier Unit	60.0
	Rounding Unit	8.0
	Select Output	1.0
	Select Input	1.1
400 MHz	Registers	27.0
	Multiplier Unit	60.4
	Rounding Unit	9.3
	Select Output	0.9
	Select Input	1.1

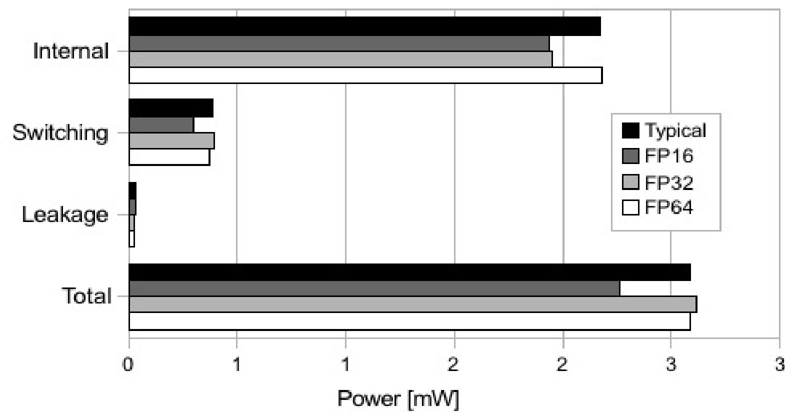
Table 4.4: Architecture one, $90nm$ CMOS building blocks power consumption.



(a) Power consumption at 200MHz.



(b) Power consumption at 300MHz.



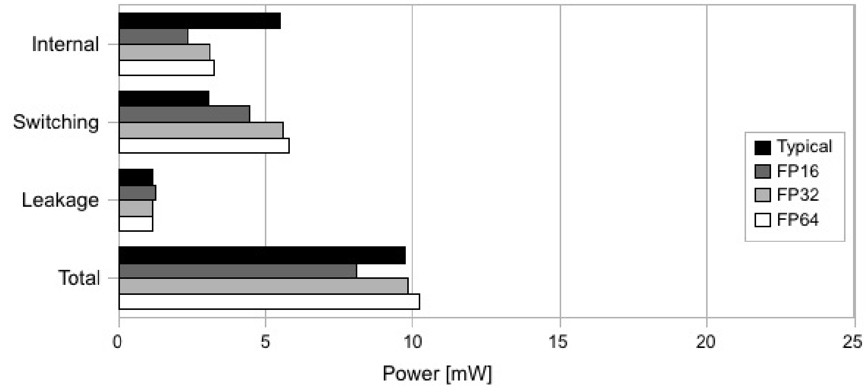
(c) Power consumption at 400MHz.

Figure 4.1: Architecture one, 65nm CMOS power consumption.

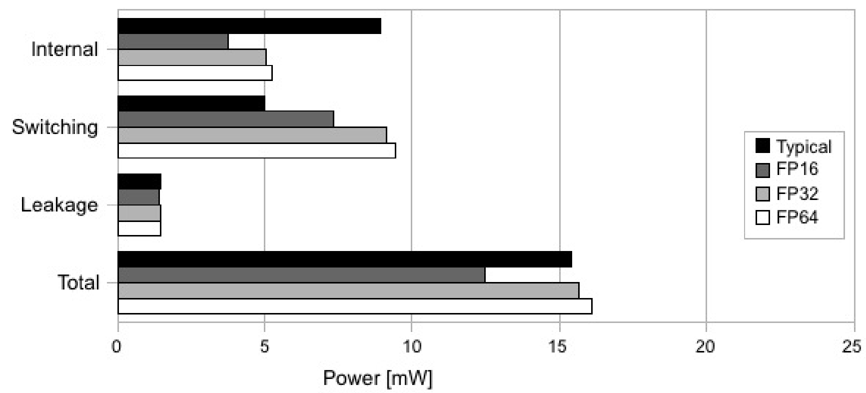
has been simulated and synthesized at 200*MHz* for only FP16 computations several times to locate the reason for this strange behavior without luck. The behavior is strange because this does not happen at either 300*MHz* or 400*MHz* clock frequency, where power consumption when performing FP16 computations are as expected. It may have happened because of insufficient control in the synthesis process, because only power, area and timing constraints are set, unexpected optimizations may have occurred. Figure 4.1 shows that FP32 computations are the most power consuming, except the strange case when performing FP16 computations at 200*MHz*. However, what is important to remember is that switching activity information from the simulation are included in the optimization process performed by Design Compiler^{TM} and the Power Compiler^{TM}, which may lead to somewhat different circuits and hence power consumptions.

Figure 4.2 shows power consumption of architecture one at 200*MHz*, 300*MHz* and 400*MHz* in 90*nm* CMOS and for typical input data distribution, only FP16 input data, only FP32 input data and only FP64 input data. Figure 4.2 better highlights the effect of increasing clock frequency than Figure 4.1 because the 90*nm* library is a general purpose library and not optimized for low power. Figure 4.2 shows that for any of the three clock frequencies, FP16 computations are the least power consuming. Typical input data distribution is the second least power consuming, FP32 computations the second largest and FP64 computations the largest. Internal power is significantly higher for typical input data distribution than for any other because the capacitance switched internal to the multiplier unit is higher, and switching power is significantly lower because several multipliers are now driving the output. When performing only FP16, FP32 or FP64 computations the internal load capacitance is reduced because only some multipliers are used, and hence internal power is reduced. Switching power is increased because the output of the used multipliers have to drive a wide bus, and the gates connected to the bus.

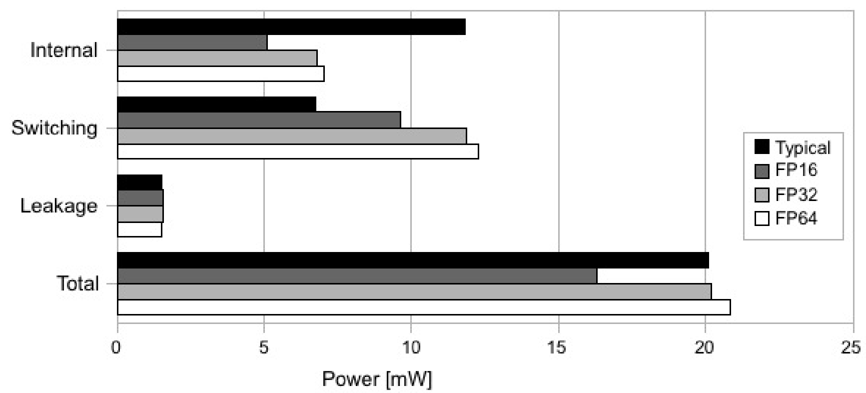
Figure 4.3 compares dissipated power by architecture one in 65*nm* and 90*nm* CMOS assuming typical input data distribution at 200*MHz*, 300*MHz* and 400*MHz*. The differences are large. In 90*nm* CMOS, on average at the different clock frequencies, 7.79 times more power is dissipated than in 65*nm* CMOS. It can also be seen that in 90*nm* CMOS, switching power is a significantly larger part of the total dynamic power consumption. In addition, leakage power is much higher in the 90*nm* circuit. However, many of the differences are probably mostly due to that the 90*nm* library is a general purpose library not optimized for low power, as the 65*nm* library is. Hence, different optimizations are performed by the Synopsys tools to meet the constraints of lowest possible total power consumption and smallest possible area at a given clock frequency.



(a) Power consumption at 200MHz.

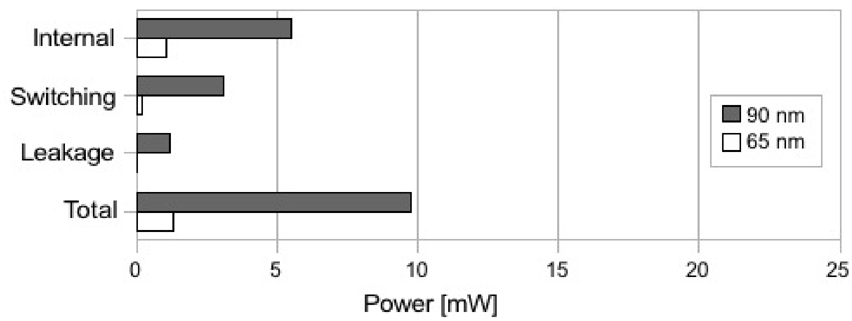


(b) Power consumption at 300MHz.

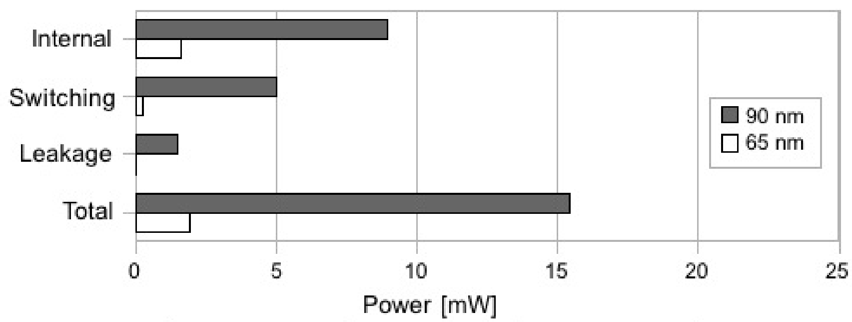


(c) Power consumption at 400MHz.

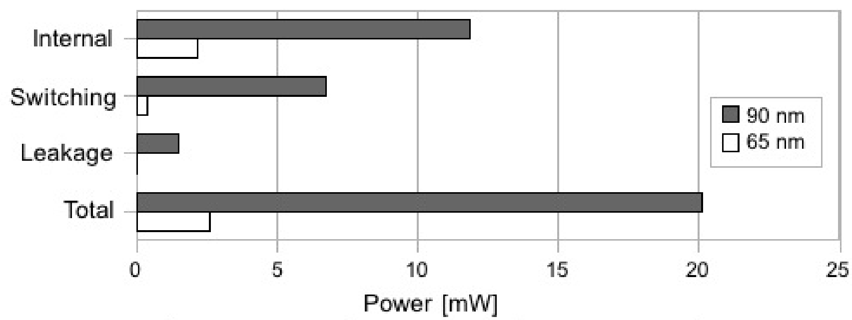
Figure 4.2: Architecture one, 90nm CMOS power consumption.



(a) Power comparison at 200MHz.



(b) Power comparison at 300MHz.



(c) Power comparison at 400MHz.

Figure 4.3: Architecture one, 90nm and 65nm CMOS power comparison.

4.2.2 Area

Table 4.5 and 4.6 presents registers, significand multiplier unit, exponent unit, rounding and exception unit and total area usage by architecture one in 65*nm* and 90*nm* CMOS technology, with typical input data distribution.

Clock frequency	Area			
200 MHz	Registers	5347.6470	9.46	% of total area
	Multiplier unit	42684.6992	75.50	% of total area
	Exponent unit	827.8377	1.47	% of total area
	Rounding unit	4378.4097	7.74	% of total area
	Total	56536.7109	100	% of total area
300 MHz	Registers	5357.0068	8.96	% of total area
	Multiplier unit	45084.0664	75.37	% of total area
	Exponent unit	829.3976	1.39	% of total area
	Rounding unit	5220.324	8.73	% of total area
	Total	59816.4414	100	% of total area
400 MHz	Registers	5406.4063	8.6	% of total area
	Multiplier unit	47699.4180	75.84	% of total area
	Exponent unit	828.8778	1.32	% of total area
	Rounding unit	5557.2886	8.84	% of total area
	Total	62899.0469	100	% of total area

Table 4.5: Architecture one, 65*nm* CMOS area usage.

Differences in area usage for the four input data distributions considered in Section 4.2.1 are small compared to differences in power consumption and around 1% at the different clock frequencies. The registers, multiplier unit and rounding and exception unit are the most area consuming building blocks in architecture one when realized in both 65*nm* and 90*nm* CMOS technology. Significand multipliers are by far the largest building block, which together with registers and the rounding and exception logic accounts for over 90% of total area. The ratio of registers to total area, multiplier unit to total area and rounding and exception unit to total area does not change significantly when realized in 65*nm* CMOS or 90*nm* CMOS. On average, at 200*MHz*, 300*MHz* and 400*MHz*, the 90*nm* circuit is 1.91 times larger than the 65*nm* circuit. This is a bit larger than expected, because the gate length in 90*nm* CMOS is approximately 1.4 times larger than in 65*nm* CMOS. However, area is also dependent of available gates and marco blocks in the target library. Area usage is, as power consumption, dependent on clock frequency, because area is traded to meet the timing constraints, mainly in the 53-bit significand multiplier. The increase in total area are more linearly increasing with clock frequency in the 65*nm* circuit than the 90*nm* circuit.

Clock frequency	Area			
200 MHz	Registers	9983.7744	9.17	% of total area
	Multiplier unit	83370.6250	76.58	% of total area
	Exponent unit	1564.0778	1.44	% of total area
	Rounding unit	8086.0181	7.43	% of total area
	Total	108863.2578	100	% of total area
300 MHz	Registers	9995.8486	8.59	% of total area
	Multiplier unit	89636.4297	77.03	% of total area
	Exponent unit	1571.7610	1.35	% of total area
	Rounding unit	8897.1494	7.65	% of total area
	Total	116362.0625	100	% of total area
400 MHz	Registers	9989.2627	8.47	% of total area
	Multiplier unit	89948.2031	76.27	% of total area
	Exponent unit	1568.4681	1.33	% of total area
	Rounding unit	9979.4297	8.46	% of total area
	Total	117933.8281	100	% of total area

Table 4.6: Architecture one, $90nm$ CMOS area usage.

4.3 Architecture Two

Architecture two trades power for area, and attempts to be an area and throughput optimized vectorized floating-point multiplier. This Section investigates power consumption and area usage of architecture two, realized in $65nm$ and $90nm$ CMOS at $200MHz$, $300MHz$ and $400MHz$ clock frequency. As for architecture one, power units are given in mW, and area units in μm^2.

4.3.1 Power

Table 4.7 and 4.9 presents internal-, switching-, leakage- and total power dissipated by architecture two realized in a $65nm$ low-power CMOS and $90nm$ CMOS, with typical input data distribution as described in Section 4.1.3.

In the $65nm$ circuit, power is mostly dissipated by charging and discharging of capacitances internal to the cells, and charging and discharging of capacitances at the output of the cells. Leakage power is very low, and the ratio of leakage power to total power decreases with increasing clock frequency because the dynamic power component grows faster than the static power component. The estimated leakage power is over 90 times larger on average, at the different clock frequencies. Table 4.8 shows which building blocks in the design that dissipates the most power. The registers, significand multipliers and the rounding and exception logic accounts for approximately 95% of total power consumption, where the multiplier unit is the most power consuming building block. The average increase in power is

Clock frequency	Power			
200 MHz	Internal	2.4660	57.46	% of dynamic power
	Switching	1.8260	42.54	% of dynamic power
	Leakage	0.0168	0.39	% of total power
	Total	4.3088	100	% of total power
300 MHz	Internal	4.2240	57.59	% of dynamic power
	Switching	3.1100	42.41	% of dynamic power
	Leakage	0.0229	0.31	% of total power
	Total	7.3569	100	% of total power
400 MHz	Internal	5.3430	56.61	% of dynamic power
	Switching	4.0950	43.39	% of dynamic power
	Leakage	0.0215	0.23	% of total power
	Total	9.4595	100	% of total power

Table 4.7: Architecture two, $65nm$ CMOS total power consumption.

Clock frequency	Power	% of total power
200 MHz	Registers	36.2
	Multiplier Unit	52.9
	Rounding Unit	6.8
	Select Output	1.6
	Select Input	1.2
300 MHz	Registers	35.3
	Multiplier Unit	54.8
	Rounding Unit	7.2
	Select Output	1.5
	Select Input	1.1
400 MHz	Registers	33.2
	Multiplier Unit	54.2
	Rounding Unit	8.1
	Select Output	1.5
	Select Input	1.1

Table 4.8: Architecture two, $65nm$ CMOS building blocks power consumption.

Clock frequency	Power			
200 MHz	Internal	6.4050	64.16	% of dynamic power
	Switching	3.5780	35.84	% of dynamic power
	Leakage	0.9980	9.09	% of total power
	Total	10.9810	100.00	% of total power
300 MHz	Internal	10.5920	64.97	% of dynamic power
	Switching	5.7120	35.03	% of dynamic power
	Leakage	1.1600	6.64	% of total power
	Total	17.4640	100.00	% of total power
400 MHz	Internal	14.1970	64.08	% of dynamic power
	Switching	7.9570	35.92	% of dynamic power
	Leakage	1.3100	5.58	% of total power
	Total	23.4620	100.00	% of total power

Table 4.9: Architecture two, $90nm$ CMOS total power consumption.

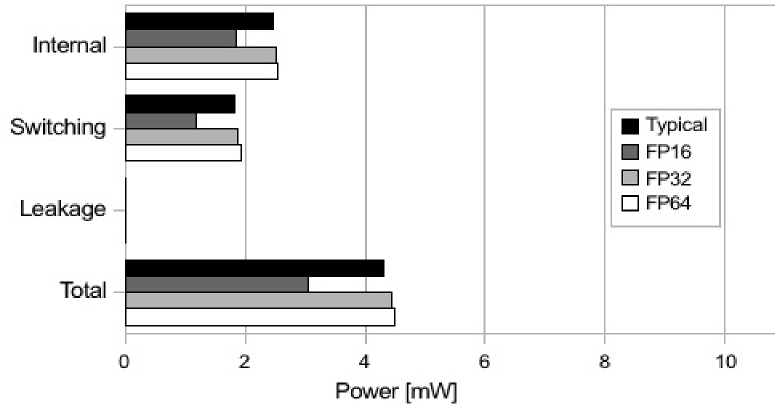
Clock frequency	Power	% of total power
200 MHz	Registers	26.8
	Multiplier Unit	63.2
	Rounding Unit	5.7
	Select Output	1.3
	Select Input	1.1
300 MHz	Registers	25.1
	Multiplier Unit	61.3
	Rounding Unit	7.8
	Select Output	1.4
	Select Input	1.0
400 MHz	Registers	25.4
	Multiplier Unit	63.0
	Rounding Unit	7.8
	Select Output	1.3
	Select Input	1.0

Table 4.10: Architecture two, $90nm$ CMOS building blocks power consumption.

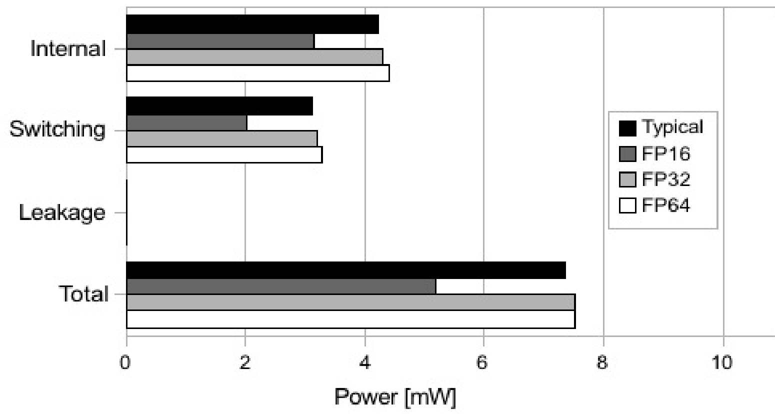
$2.5735mW/100MHz$.

Table 4.9 presents internal-, switching-, leakage- and total power dissipated by architecture two realized in 90*nm* CMOS. Compared to the 65*nm* circuit, the 90*nm* circuit has significantly higher total power consumption at all clock frequencies. The internal power percentage is higher, while the switching power percentage is lower. Leakage power is also increased compared to the 65*nm* circuit, and on average responsible for 7.10% of total power consumption. The ratio leakage to total power is reduced as clock frequency increases, because internal and switching power grows faster than leakage power. Leakage power is independent of clock frequency but proportional to area. As clock frequency increases, larger area are required by mainly the significand multipliers as a tradeoff between area, timing and power. Average increase in power is $6.2405mW/100MHz$, which is $3.6670mW$ higher than the 65*nm* circuit. Table 4.10 shows which part of the circuit that dissipates most power. Compared to the 65*nm* circuit, less power is consumed by the registers, and more power is consumed by the multiplier unit. Average increase in power consumed by the multiplier unit is 8.53%, and average reduction in power consumed by the registers are 9.13%. As for the 65*nm* circuit, the rounding unit is the third most power consuming unit. The differences in leakage-, internal- and switching power of the 65*nm* circuit and the 90*nm* circuit are probably due to different optimizations performed by the Synopsys tools based on available cells in the target library.

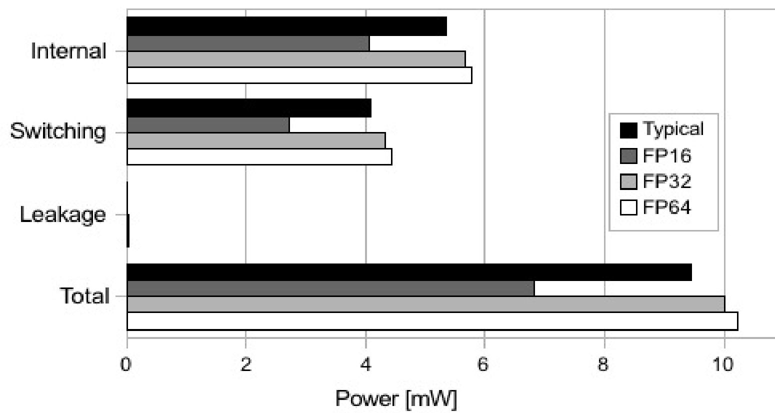
Figure 4.4 shows power consumption of architecture two at 200*MHz*, 300*MHz* and 400*MHz* in 65*nm* CMOS and for typical input data distribution, only FP16 input data, only FP32 input data and only FP64 input data. Internal-, switching-, leakage- and total power are included to show how input data distribution affects power consumption of architecture two. FP16 computations dissipates the least amount of power at all clock frequencies, mostly because of less charging and discharging of load capacitances both internal in multiplier cells and at their outputs. The differences of the four input data distributions increases with clock frequency, and are clearest at 400*MHz*. Figure 4.5 shows power consumption of architecture two realized in 90*nm* CMOS, which shows even larger differences in power consumption at increasing clock frequency compared to the 65*nm* circuit. The effect of which data format used are somewhat different in the 65*nm* circuit and the 90*nm* circuit. The most significant difference are the internal- and switching power component of the two circuits. In the 65*nm* circuit, the internal power are almost equal for typical input data, FP32 input data and FP64 input data, however in the 90*nm* circuit, the internal power is significantly larger for typical input data than for FP32- and FP64 input data. In both circuits, the internal power are smallest for FP16 input data. In the 65*nm* circuit, the switching power are almost equal for typical input data, FP32- and FP64



(a) Power consumption at 200MHz.

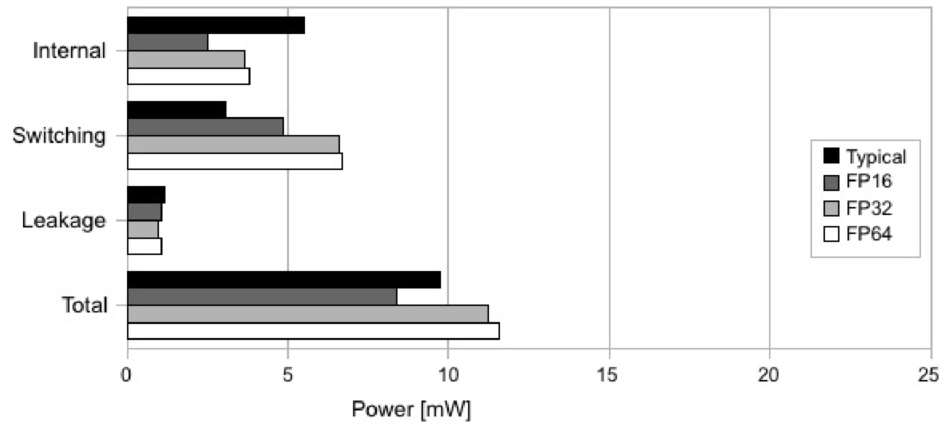


(b) Power consumption at 300MHz.

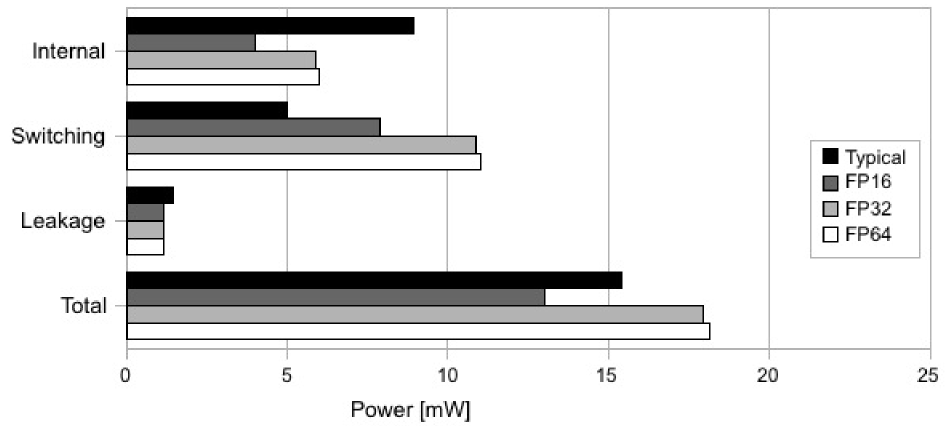


(c) Power consumption at 400MHz.

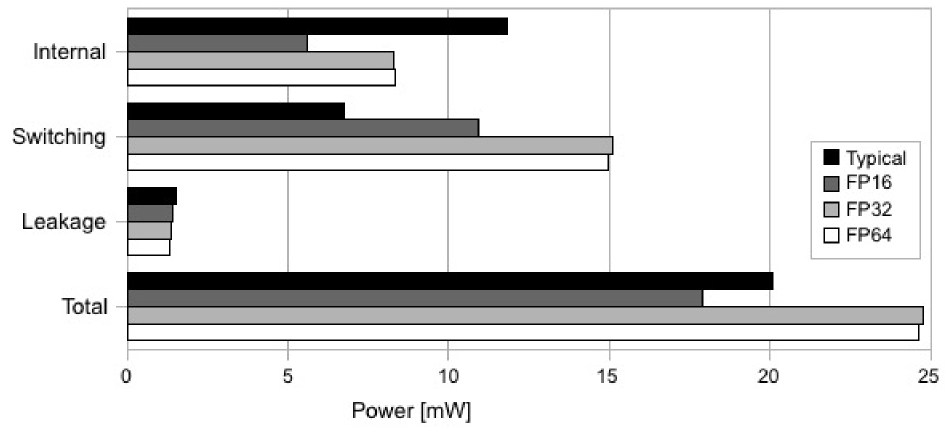
Figure 4.4: Architecture two, 65nm CMOS power consumption.



(a) Power consumption at $200MHz$.

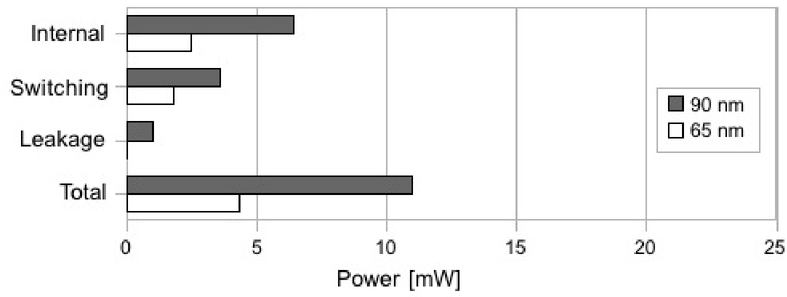


(b) Power consumption at $300MHz$.

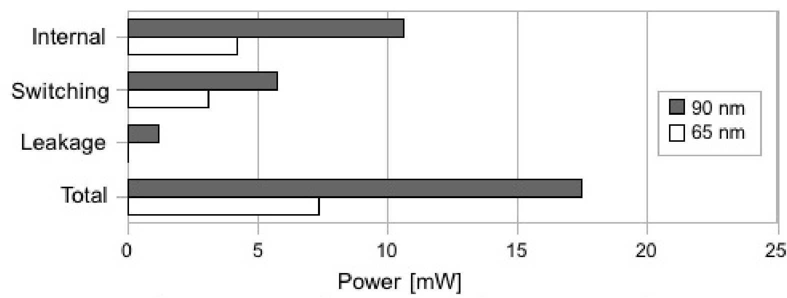


(c) Power consumption at $400MHz$.

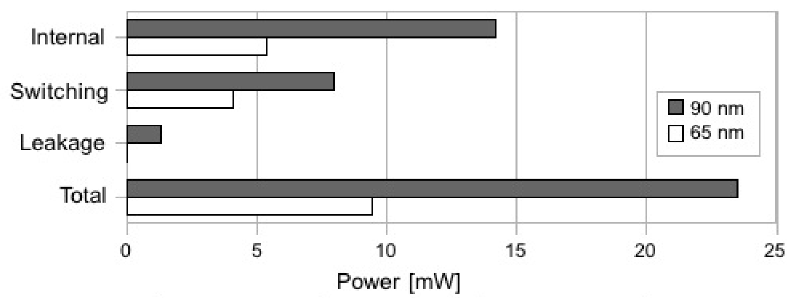
Figure 4.5: Architecture two, $90nm$ CMOS power consumption.



(a) Power comparison at 200MHz.



(b) Power comparison at 300MHz.



(c) Power comparison at 400MHz.

Figure 4.6: Architecture two, 90nm and 65nm CMOS power comparison.

input data as well, but in the $90nm$ circuit typical input data has the lowest switching power, then FP16- input data, and FP32- and FP64 input data have almost equal switching power. These differences are probably due to available cells in the target library, and hence optimization performed in the datapaths. In the simplified power estimation methodology used to compare the architectures, architecture two was estimated to have equal power consumption for only FP16, only FP32 and only FP64 computations. This is not the case. In the estimation methodology, is was assumed that every bit in the multipliers had equal switching for all formats. As seen from Figure 4.4 and Figure 4.5, FP16 computations have both less switching and internal power compared to FP32 and FP64 computations. This should be considered if the power estimation methodology is to be improved. Concerning total power, both circuits have power consumptions where FP16 computations requires the least amount of power, then typical input data computations, and FP32- and FP64 computations have almost equal total power consumptions.

Figure 4.3 compares dissipated power by architecture two in $65nm$ and $90nm$ CMOS assuming typical input data distribution. Power consumption by the $90nm$ circuit is on average $10.26mW$ higher than the $65nm$ circuit, at 200*MHz*, 200*MHz* and 400*MHz* clock frequency. Differences in leakage power of the two circuits are well highlighted in Figure 4.3. Internal- and switching power are closer to equal for the $65nm$ circuit than the $90nm$ circuit. This is due to more switching internal in the cells in the $90nm$ circuit, and less switching of outputs. These differences are probably due to different optimizations because of available cells in the target library.

4.3.2 Area

The area usage of architecture two is presented in Table 4.11 and 4.12, where area required by registers, the multiplier unit, the exponent unit and the exception and rounding unit, in addition to total area are included.

Differences in area usage for the four input data distributions are small compared to differences in power consumption and around 1 % at the different clock frequencies. The significand multiplier unit is by far the largest unit in both the $65nm$ circuit and the $90nm$ circuit. The ratio of multiplier unit area to total area is approximately 2% larger in the $90nm$ circuit, compared to the $65nm$ circuit. The $90nm$ circuit are on average 1.93 times larger than the $65nm$ circuit and $200MHz$, $300MHz$ and $400MHz$ clock frequency. The area usage of architecture two increases more linearly with clock frequency when realized in the $90nm$ general purpose library. When realized in the $65nm$ low-power library, the largest increase in area occurs when going from $200MHz$ to $300MHz$ clock frequency. When going from

Clock frequency	Area			
200 MHz	Registers	5739.7617	12.27	% of total area
	Multiplier unit	33117.7344	70.77	% of total area
	Exponent unit	656.7595	1.40	% of total area
	Rounding unit	3883.3469	8.30	% of total area
	Total	46796.8789	100	% of total area
300 MHz	Registers	5775.6426	11.36	% of total area
	Multiplier unit	36380.5391	71.55	% of total area
	Exponent unit	659.3595	1.30	% of total area
	Rounding unit	4614.4980	9.08	% of total area
	Total	50843.0000	100	% of total area
400 MHz	Registers	5800.0767	11.53	% of total area
	Multiplier unit	35370.0977	70.30	% of total area
	Exponent unit	675.4794	1.34	% of total area
	Rounding unit	4896.8638	9.73	% of total area
	Total	50314.1602	100	% of total area

Table 4.11: Architecture two, $65nm$ CMOS area usage.

Clock frequency	Area			
200 MHz	Registers	10668.7070	11.7	% of total area
	Multiplier unit	66626.7500	73.06	% of total area
	Exponent unit	1250.1659	1.37	% of total area
	Rounding unit	6541.7031	7.17	% of total area
	Total	91194.0938	100	% of total area
300 MHz	Registers	10696.1465	11.23	% of total area
	Multiplier unit	68993.0703	72.44	% of total area
	Exponent unit	1247.9708	1.31	% of total area
	Rounding unit	8062.9741	8.47	% of total area
	Total	95242.0469	100	% of total area
400 MHz	Registers	10759.8096	10.81	% of total area
	Multiplier unit	71746.6641	72.10	% of total area
	Exponent unit	1255.6536	1.26	% of total area
	Rounding unit	8995.9385	9.04	% of total area
	Total	99506.2188	100	% of total area

Table 4.12: Architecture two, $90nm$ CMOS area usage.

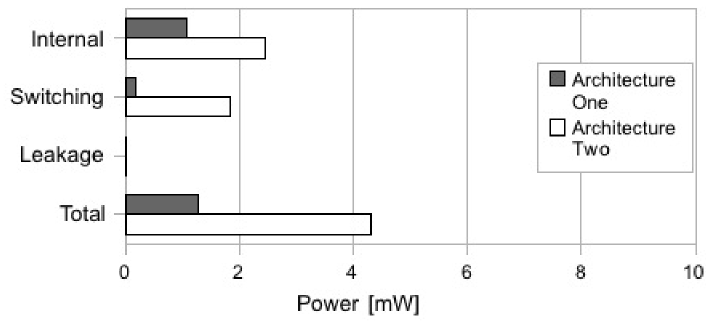
$300MHz$ to $400MHz$, area is reduced somewhat for the $65nm$ circuit. This is probably because at $300MHz$, the $65nm$ circuit has traded area for better power results.

4.4 Power Comparison

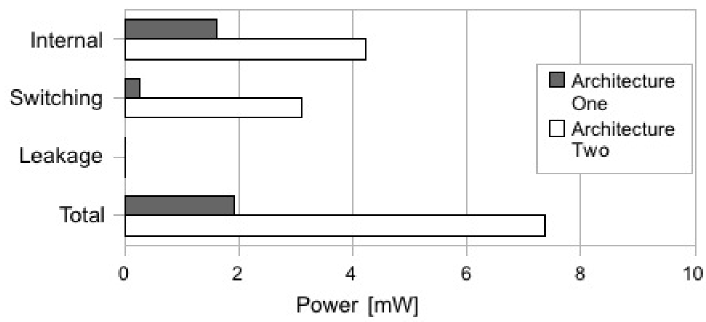
Because architecture one trades area for better power results, and architecture two trades power for better area results, different power and area results were expected, as estimated in Section 2.2 and 2.3. This Section will compare the power results of the implemented architectures. In addition, because the $65nm$ circuits are realized using a low-power CMOS process, and the $90nm$ circuits are realized using a general purpose CMOS process, differences in target process will be highlighted. Figure 4.7 compares dissipated power by architecture one and two at 200*MHz*, 300*MHz* and 400*MHz* for typical input data distribution realized in a $65nm$ low-power CMOS process, and Figure 4.8 compares dissipated power by the two architectures realized in a $90nm$ general purpose CMOS process.

From Figure 4.7 it can be seen that architecture one has much better power results than architecture two. On average, at 200*MHz*, 300*MHz* and 400*MHz*, power consumed by architecture two is $5.1104mW$ larger than by architecture one. Both internal power and switching power are significantly lower in architecture one, due to reduced switching inside the cells and switching at the their outputs. Hence, to obtain the best power results architecture one should be chosen. The difference in power consumption by the two architectures grows larger as clock frequency increases. However, because the vectorized floating-point multipliers are reaching the limit of how much clock frequency can be increased without introducing pipeline registers in the significand multipliers, or multicycle multipliers, power results may be different at higher frequencies. Because of the surprising result that registers are more power consuming the $65nm$ circuit of architecture one, power may be further reduced if low-power registers are used, assuming this is the cause for the result.

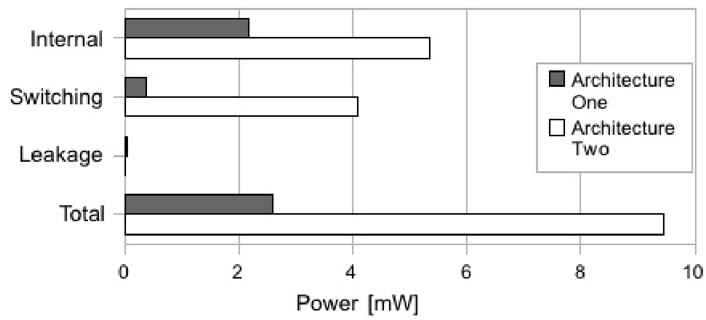
If the two architectures are realized in a $90nm$ general purpose CMOS process, the differences between architecture one and architecture two are less distinct, as seen in Figure 4.8. On average, at the different clock frequencies, architecture two consumes $2.2200mW$ more power than architecture one. The difference in power consumption of the two architectures are much less when realized in a general purpose process than if a low-power process is used. However, the difference grows larger as clock frequency increases because dynamic power becomes more dominant over leakage power. At



(a) Power comparison at $200MHz$.

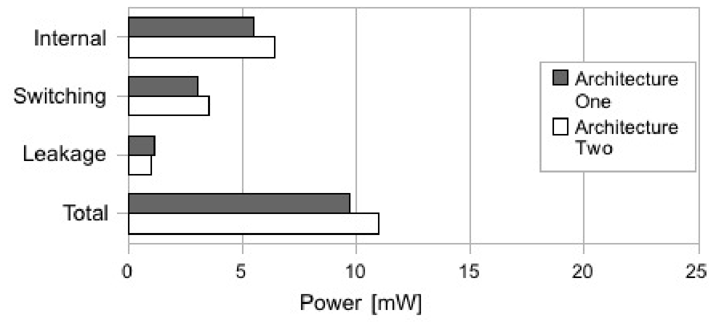


(b) Power comparison at $300MHz$.

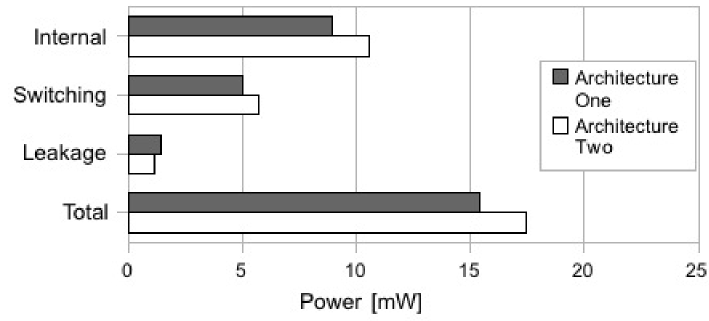


(c) Power comparison at $400MHz$.

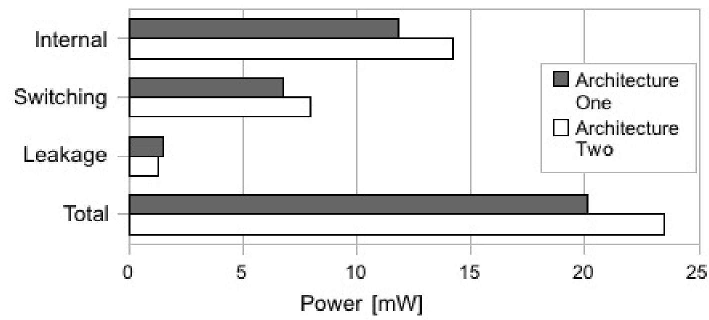
Figure 4.7: $65nm$ architecture power comparison.



(a) Power comparison at 200MHz.



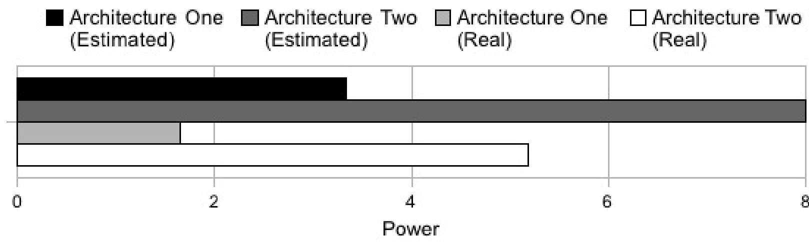
(b) Power comparison at 300MHz.



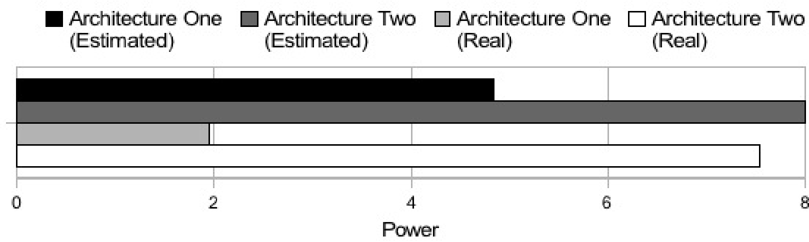
(c) Power comparison at 400MHz.

Figure 4.8: 90nm architecture power comparison.

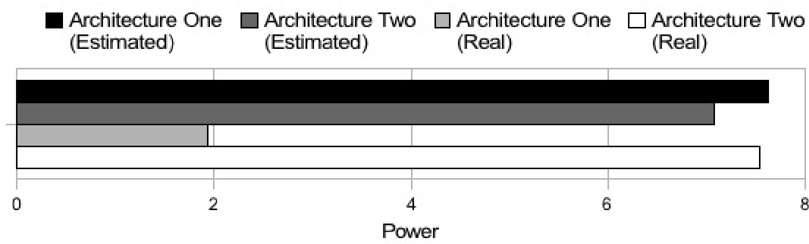
$400MHz$, the difference in power consumption of the two architectures equals approximately $4mW$, while at $200MHz$ the difference is approximately $1mW$. As for the $65nm$ circuits, to obtain the best power result, architecture one should be chosen.



(a) Estimated vs. real power, only FP16 input data.



(b) Estimated vs. real power, only FP32 input data.



(c) Estimated vs. real power, only FP64 input data.

Figure 4.9: Estimated vs. real power comparison.

Architecture one and two was selected for implementation based on estimations performed in Chapter 2. Figure 4.9 compares the estimated power consumption of architecture one and two to the real power consumption obtained from synthesis, for only FP16 input data, only FP32 input data and only FP64 input data. The numbers for the real power consumption are from the $65nm$ circuits at $300MHz$, but the same relative difference between the architectures would be obtained at $200MHz$ and $400MHz$, and by the $90nm$ circuits, except at $200MHz$ in the $65nm$ architecture one circuit where power consumption is surprisingly high when computing only FP16 input data. As seen from Figure 4.9, the estimated power consumption gives a

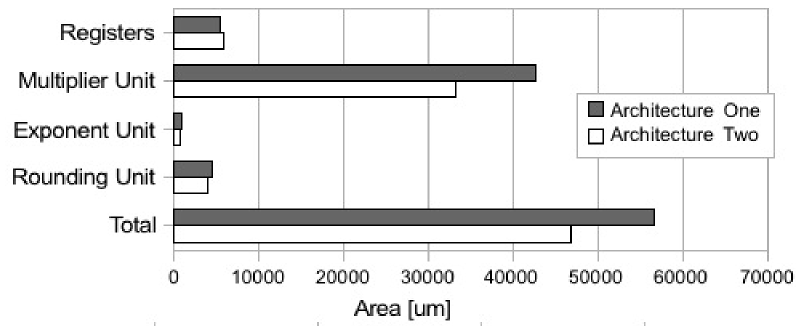
good picture of the relative difference between the two architectures, except when only FP64 input data are computed. When only FP64 input data are computed, architecture one is estimated to have higher power consumption than architecture two. This is because static power was assumed to be 30% of total power consumption, which is much higher than in both the 65nm and the 90nm circuits, where average static power is less than 1.5% and less 10%, respectively. In addition, significand multipliers are not implemented as array multipliers as assumed in the estimation methodology but parallel-prefix multipliers which exploits low-power features in the target library to obtain better power results. The relative estimated difference in power consumption by the two architectures, is largest when only FP16 input data is computed, and decreases when only FP32 data is computed. But, as seen in Figure 4.9, the real difference in power consumption grows larger larger when only FP32 and only FP64 input data is computed compared to only FP16 input data. Hence, the estimation methodology predicted correctly in two of three cases, and has a fidelity of 66%.

4.5 Area Comparison

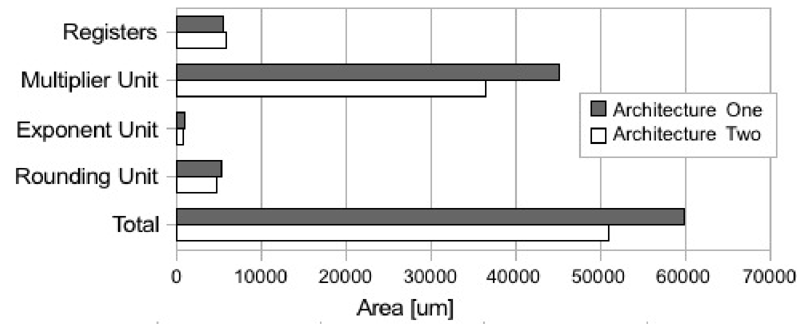
Architecture one trades area for better power results, and architecture two trades power for better area results. By using multipliers, adders and subtractors, and rounding and exception logic that exactly fit the width of the operands being computed, architecture one reduces total power consumption at the cost of additional multipliers, adders and subtractors, and rounding and exception logic. This additional logic increases total area significantly, compared to architecture two. Figure 4.10 compares the 65nm circuits concerning area usage, and Figure 4.11 compares the 90nm circuits. The relative difference in area usage of the two architectures are approximately equal in the 65nm and 90nm circuits. On average, in the 65nm realization of architecture one and two, architecture one is 10432.7200 μm^2 larger than architecture two. In the 90nm realization, architecture one is 19072.2630 μm^2 larger than architecture two. On average, at different clock frequencies, the multiplier unit in architecture two accounts for approximately 70% of total area in 90nm CMOS, and approximately 72% in 65nm CMOS. In architecture one, the multiplier unit accounts for approximately 76% of total area in both 65nm and 90nm CMOS. Hence, the multiplier unit is the largest unit in both architectures. Figure 4.10 and 4.11 shows that the differences in the multiplier unit accounts for almost all the difference between the architectures. The difference in area usage by registers, exponent unit and rounding and exception unit is very small. As discussed in Section 4.2.2, area of architecture one increases more linearly with clock frequency in the 65nm circuits than in the 90nm circuits. For architecture two, area increases more linearly with increasing clock frequency in the 90nm circuits than in

the 65nm circuits. This is probably due to the nature of the architectures and optimization performed by the Synopsys tools based on target library.

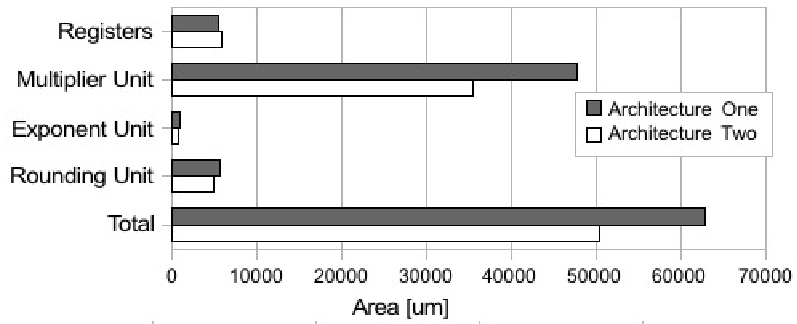
As can be seen from Table 4.5, 4.6, 4.11 and 4.12, significand multipliers accounts for more than 70% of total area, and registers for more than 9% of total area. The estimations performed in Section 2.3 is based on transistors used in significand multipliers, exponent adders and registers, assuming multipliers implemented as array multipliers. In the synthesized circuits, significand multipliers are implemented as parallel-prefix multiplier from the DesignWare® library provided by Synopsys. In the estimations performed in Section 2.3, architecture one is estimated to require approximately 15% larger area than architecture two. In the 90nm realization of the two architectures, architecture one requires, on average at the different clock frequencies, 16.7% larger area than architecture two. In the 65nm circuits, on average, architecture one requires 17.5% larger area than architecture two. Hence, the estimation methodology has a fidelity of 100%, even if significand multipliers are implemented differently than assumed. This is because the multipliers are by far the largest building blocks of the design, and together with the registers accounts for approximately 80% of total area at different clock frequencies and target technologies.



(a) Area usage at $200MHz$.

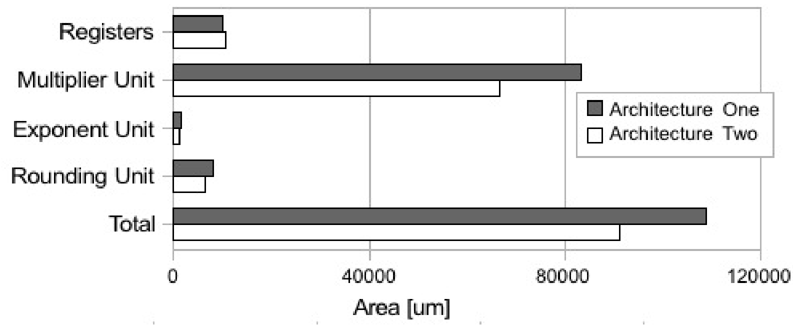


(b) Area usage at $300MHz$.

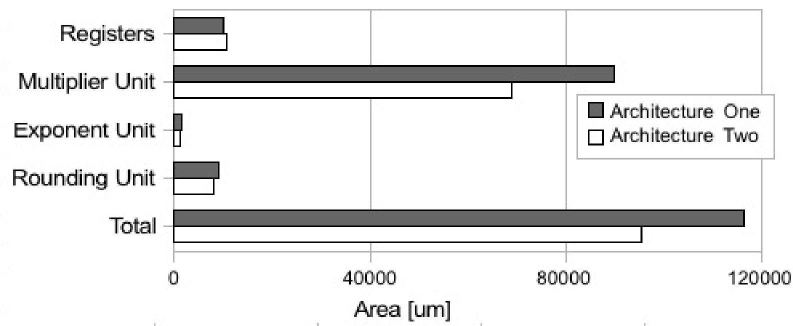


(c) Area usage at $400MHz$.

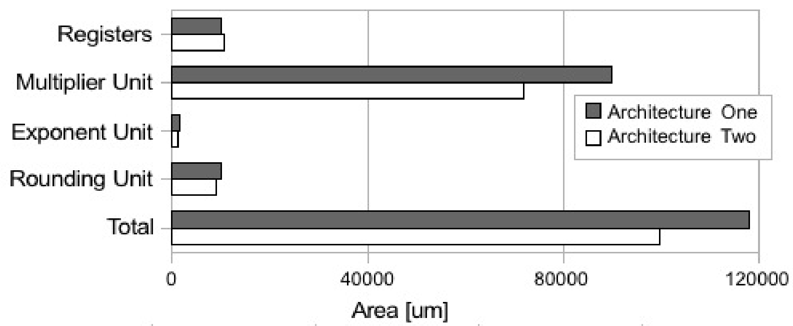
Figure 4.10: $65nm$ CMOS architecture area comparison.



(a) Area usage at 200MHz.



(b) Area usage at 300MHz.



(c) Area usage at 400MHz.

Figure 4.11: 90nm CMOS architecture area comparison.

Chapter 5

Conclusions

This Chapter concludes the thesis. Four, partially IEEE compliant, pipelined, vectorized floating-point multipliers supporting FP16, FP32 and FP64 input data was proposed in [7], and evaluated concerning area, power, latency and throughput. A methodology for estimating power has been developed to help choosing the best architecture to implement given a set of constraints. Two architectures with different area usage and power consumption have been implemented in RTL. Architecture one trades area for better power results, and architecture two trades power for smaller area. The two architectures have equal latency and throughput. The implemented architectures have a latency of five clock cycles, and a throughput of 38400*Mbit/s* at 300*MHz* clock frequency.

The architectures have been tested with 500,000 testcases for each supported format and rounding mode to ensure correct behavior according to the IEEE standard for binary floating-point arithmetic. The simulations revealed an error in the rounding logic, which in rare cases rounds the product to zero when it should be rounded to the smallest representable normalized number in round-to-nearest even and round-to positive infinity mode. The error is believed to be format independent, but has only been detected when performing FP16 computations.

Architecture one and two have been synthesized at 200*MHz*, 300*MHz* and 400*MHz* clock frequency, and for typical input data distribution, assuming 20% FP16 computations, 60% FP32 computations and 20% FP64 computations. In addition, the architectures have been synthesized for only FP16 computations, only FP32 computations and only FP64 computations to see how input data distribution affects power consumption. The architectures have been synthesized using a 65*nm* low-power standard cell library, and a 90*nm* general purpose standard cell library, to see how target technology affects the architectures concerning power.

5.1 Estimation Methodologies

An area estimation methodology was developed in [7], and a power estimation methodology has been developed in this thesis. The estimation methodologies have been used to select architecture one and two for implementation.

Power is estimated based on power dissipated by the significand multipliers, and simulation results from [1] and [2]. Static power consumed by a Full-Adder cell was computed using results from [1]. Assuming 30% static power consumption, in accordance with simulations performed in [2], dynamic power was computed from static power. The total power dissipated by the significand multipliers in the four proposed architectures was computed assuming multipliers implemented as array multipliers, using full-adders. Because static power is strongly technology dependent, and varies between process technologies, this estimation methodology has several uncertainties and sources of error. The power estimation methodology predicted architecture one to have the lowest power consumption for FP16 and FP32 input data, and architecture two to have the lowest power consumption for FP64 input data. From synthesis power reports it is seen that architecture one has lower power consumption than architecture two for all input data distributions, clock frequencies, and in both $65nm$ and $90nm$ technology. Hence, the power estimation methodology predicted correctly in two of the three estimated input data cases and has a fidelity of 66%.

Because area required by registers and significand multipliers accounts for the larger part of the vectorized floating-point multipliers proposed in [7], this is used to compare the architectures concerning area. Area estimations are performed based on the transistor count in significand multipliers and registers. The ratio of number of transistors in a Full-Adder cell to number of transistors in a 1-bit register, is used to compute the area required by the significand multipliers and registers of the different architectures. This gives a good picture of the the relative difference in area usage by the four architectures. Architecture one was estimated to be 15% larger than architecture two. From synthesis area reports it is seen that architecture one is 16.7% larger than architecture two in $90nm$ technology, and 17.5% in $65nm$ technology, on average at $200MHz$, $300MHz$ and $400MHz$ clock frequency. Hence, the estimation methodology predicted a quite accurate relative difference between the architectures, and has a fidelity of 100%.

5.2 Power Results

The implemented architectures are designed to have different power consumptions, independent of target technology. But, because the $65nm$ li-

brary is a low-power process, and the $90nm$ library is a general purpose process, this highlights the differences in power consumption by the two architectures depending on target technology, in addition the the architectural differences. When realized in a $65nm$ low-power library, architecture one has a total power consumption of $1.9200mW$ at $300MHz$, and architecture two a total power consumption of $7.3569mW$. The average increase in total power consumption is $0.6505mW/100MHz$ for architecture one, and $2.5735mW/100MHz$ for architecture two. When realized in a $90nm$ general purpose library, architecture one has a total power consumption of $15.4090mW$ at $300MHz$, and an average increase in total power consumption equal to $5.1850mW/100MHz$. Architecture two has a total power consumption of $17.4640mW$, and an average increase of $6.2405mW/100MHz$.

The difference in power consumption by the two architectures are higher when realized in a low-power process than in a general purpose process technology. The difference in power consumption at $300MHz$ is $5.4369mW$. This is because the synthesis tools exploits the low-power properties of the library when performing circuit optimization. When realized in a general purpose library, the difference in total power consumption is $2.0550mW$ at $300MHz$. Hence, to fully obtain the best power result, architecture one should be realized in a low-power process.

5.3 Area Results

Because architecture one trades area for better power results it was estimated to use 15% larger area than architecture two. When realized in the $65nm$ library, architecture one area usage is $59816.4414\mu m^2$ at $300MHz$, architecture two area usage is $50843.0000\mu m^2$. When realized in the $90nm$ library, architecture one area usage is $116362.0625\mu m^2$, and architecture two area usage is $95242.0469\mu m^2$. Area is affected by clock frequency because area is traded to meet timing constraints, mainly in the 53-bit significand multiplier. In the $65nm$ circuits, architecture one is 17.5% larger than architecture two, and in the $90nm$ circuits, architecture one is 16.7% larger than architecture one. Hence, the relative difference in area usage are approximately equal when realized in a low-power library and a general purpose library.

5.4 Future Work

The implemented architectures have several improvements. Sticky exceptions, and clearing of exceptions have not been implemented properly. The implemented vectorized floating-point multipliers generates exceptions according to the IEEE standard for binary floating-point arithmetic, but the standard requires that exceptions shall be sticky and explicitly cleared by

user. By writing to a clear-register, exceptions should be cleared. This has not been implemented according to the standard, and should be implemented to comply the IEEE standard.

An error in the rounding logic has been detected when simulating only FP16 input data in rounding-to-nearest even and round-to positive infinity mode. The error is believed to be format independent, but is only successfully detected by the FP16 input vectors. Result should be rounded to the smallest representable normalized number, but is rounded to zero. This error also has to be corrected to make the vectorized floating-point multipliers IEEE compliant. Rounding could be performed more effectively if the QFT algorithm presented in [20] is used. This requires the sum and carry form the significand multipliers to be delivered as carry-save encoded vectors. The DesignWare® library provides a multiplier with carry-save encoded sum and carry output [29], which could be used when implementing this algorithm.

Because the power estimation methodology did not predict correct relative difference in power consumption in all cases, this should be improved. To improve the power estimation methodology, target technology has to be taken into account, because static power differs significantly for a low-power library and a general purpose library. In addition, power consumption of architecture two is are not equal for FP16 computations, FP32 computations and FP64 computations as estimated. As seen from the 65nm synthesis results of architecture one, the multiplier unit is not the most power consuming. This should be investigated further. If this is the case, the power estimation methodology can not be based on the significand multipliers alone, power dissipated by registers also have to be included. A weight-function should be developed, where input format distribution and target technology are included when estimating power. Power, area and throughput should be weighted for a given set of architectures and constraints to give a better basis for choosing the best architecture to implement. Clock frequency should perhaps be included in the methodology as well, because differences in power consumption by the two architectures grows larger with increasing clock frequency.

The architectures are generically implemented, and can relatively easy be changed to a 256-bit input vectorized floating-point multiplier. The differences in area usage will be greater, and it might be interesting to look at power consumption of the two architectures, especially in a general purpose process where static power is a significant contributor to total power. Hence, architecture two might have better power results than architecture two due to lower static power dissipation, and because FP16, FP32 and FP64 computations have not equal dynamic power consumption as assumed in the power estimation methodology.

Bibliography

[1] S. T. Oskuii, *Design of Low-Power Reduction-Trees in Parallel Multipliers*. PhD thesis, Norwegian University of Science and Technology, 2008.

[2] Q. X. et al., "Efficient subthreshold leakage current optimization - leakage current optimization and layout migration for 90- and 65- nm asic libraries," *Circuits and Devices Magazine, IEEE*, vol. 22, no. 5, pp. 39–47, Sept.-Oct. 2006.

[3] ARM, "Mali™ graphics solution." http://www.arm.com/products/esd/multimediagraphics_malioverview.html.

[4] A. Stevens, "Arm® mali™ 3d graphics system solution." http://www.arm.com/miscPDFs/16514.pdf, December 2006.

[5] Khronos, "Opengl es - the standard for embedded accelerated 3d graphics." http://www.khronos.org/opengles/.

[6] Khronos, "Openvg - the standard for vector graphics acceleration." http://www.khronos.org/openvg/.

[7] E. Stenersen, "Vectorized 256-bit input fp16/fp32/fp64 floating-point multiplier." Norwegian University of Science and Technology, 2007.

[8] IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.

[9] I. Koren, *Computer Arithmetic Algorithms*. Natick, MA, USA: A. K. Peters, Ltd., 2001.

[10] T. Njølstad, "Introduction to sie40aa low power digital design," *NTNU*, 2002.

[11] R. B. Anantha P. Chandrakasan, *Low Power Digital CMOS Design*. Springer, 1995.

[12] L. Wanhammar, *DSP Integrated Circuits*. Academic Press, 1999.

[13] L. DADDA, "Some schemes for parallel multipliers," *Alta Frequenza 34*, pp. 349–356, May 1965.

[14] C. WALLACE, "A suggestion for a fast multiplier," *EEE Trans. Electron. Comp.*, pp. 14–17, Feb. 1964.

[15] M. Pedram, "Power minimization in ic design: principles and applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 1, pp. 3–56, 1996.

[16] D. D. Gajski, *Principles of Digital Design*. Prentice Hall, 1997.

[17] Synopsys, "Designware®." http://www.synopsys.com/dw/buildingblock.php.

[18] Synopsys, "Designware®." http://www.synopsys.com/products/designware/docs/doc/dwf/datasheets/dw02_mult.pdf.

[19] Synopsys, "Designware®." http://www.synopsys.com/products/designware/dwtb/articles/multiplier_bldg_block/multiplier_bldg_block.html.

[20] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for ieee floating-point multiplication," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 638–650, 2000.

[21] N. T. Quach, N. Takagi, and M. J. Flynn, "Systematic ieee rounding method for high-speed floating-point multipliers," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 5, pp. 511–521, 2004.

[22] S. Williams, "Icarus verilog." http://www.icarus.com/eda/verilog/.

[23] Synopsys, "Designware®." http://www.synopsys.com/products/designware/docs/doc/dwf/datasheets/dw_fp_mult.pdf.

[24] Synopsys, "Designware®." http://www.synopsys.com/products/designware/docs/doc/dwf/datasheets/fp_overview2.pdf.

[25] Synopsys, "Design compiler™." http://www.synopsys.com/products/logic/design_compiler.html.

[26] Synopsys, "Power compiler™." http://www.synopsys.com/products/power/power_ds.html.

[27] Synopsys, *Power Products Reference Manual*, 1999.

[28] Synopsys, "Design compiler ultra performance capabilities." http://www.analogy.com/products/logic/adc_ultratech_bgr.pdf.

[29] Synopsys, "Partial product multiplier." http://synopsys.com/products/designware/docs/doc/dwf/datasheets/dw02_multp.pdf.

Appendix A

Architecture One Verilog Sources

The defines file contains definitions used in the design files.

```
 1 // ------------------------------------------------------------------
 2 // File.......:  defines.v
 3 // Author.....:  Espen Stenersen
 4 // Date.......:  Wed May 14 11:45:28 CEST 2008
 5 // Revision...:  1.0
 6 // Description:  Contains definitions used in the design files.
 7 //               Openrand widths, exponent widths, significand widths,
 8 //               bias values and bus widths.
 9 // ------------------------------------------------------------------
10
11 `define FP16         0
12 `define FP32         1
13 `define FP64         2
14
15 `define FP16W        16
16 `define FP32W        32
17 `define FP64W        64
18
19 `define FP16SW       10
20 `define FP32SW       23
21 `define FP64SW       52
22
23 `define FP16EW       5
24 `define FP32EW       8
25 `define FP64EW       11
26
27 `define FP16BIAS     15
28 `define FP32BIAS     127
29 `define FP64BIAS     1023
30
31 `define FRACBUS      2*(`FP64SW+1)
32 `define EXPBUS       4*`FP32EW
33 `define SIGNBUS      4
34
35 `define BUS          128
36
37 `define EVEN         0
38 `define PINF         1
39 `define NINF         2
```

```
40 `define ZERO          3
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: vec_fp_mult.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 10:30:15 CEST 2008
5 // Revision...: 1.0
6 // Description: Vectorized FP16/FP32/FP64 floating-point multiplier
7 //              top module. Assembles the architecture.
8 // ---------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module vec_fp_mult
13 (
14    start,          // Input. Starts computation.
15    vectors,        // Input. FP vectors to be computed.
16    format,         // Input. Format of vectors.
17    mode,           // Input. Rounding mode.
18    clear,          // Input. Clears specified exceptions.
19    products,       // Output. Computed products.
20    exceptions,     // Output. Exceptions raised.
21    ready,          // Output. Output vector ready.
22    clk,
23    reset_n
24 );
25
26    // input(s)
27    input                      clk;
28    input                      reset_n;
29    input                      start;
30    input [`BUS-1:0]           vectors;
31    input [1:0]                format;
32    input [1:0]                mode;
33    input [15:0]               clear;
34
35    // output(s)
36    output [`BUS-1:0]          products;
37    output [15:0]              exceptions;
38    output                     ready;
39
40    // wire(s)
41    wire                       reset;
42    wire [`BUS/2-1:0]          DRH_to_stage2;
43    wire [`BUS/2-1:0]          DRL_to_stage2;
44    wire [1:0]                 IR_to_stage2;
45    wire [1:0]                 IR_to_stage3;
46    wire [1:0]                 IR_to_stage4;
47    wire [1:0]                 M_to_stage2;
48    wire [1:0]                 M_to_stage3;
49    wire [1:0]                 M_to_stage4;
50    wire [15:0]                S0_to_stage4;
51    wire [`FRACBUS-1:0]        DRF_to_stage3;
52    wire [`FRACBUS-1:0]        DRF_to_stage4;
53    wire [`EXPBUS-1:0]         DRE_to_stage3;
54    wire [`EXPBUS/2+3:0]       DRE_to_stage4;
55    wire [`SIGNBUS-1:0]        DRS_to_stage3;
56    wire [`SIGNBUS/2-1:0]      DRS_to_stage4;
57    wire                       start_to_stage1;
58    wire                       start_to_stage2;
59    wire                       start_to_stage3;
60    wire                       start_to_stage4;
61    wire                       load_ST0;
62
```

```
63
64    // ------------------------------------------------------------------
65    // Module instantiation.
66    // ------------------------------------------------------------------
67
68    // Registers. Keep track of start signal to set ready signal when
69    // needed.
70
71    reg_enable #(1) ST0
72    (
73       .d         (start),              // Data in.
74       .q         (start_to_stage1),    // Data out.
75       .enable    (load_ST0),           // Enable bit.
76       .clk       (clk),
77       .reset     (reset)
78    );
79
80    ff #(1) ST1
81    (
82       .d         (start_to_stage1),    // Data in.
83       .q         (start_to_stage2),    // Data out.
84       .clk       (clk),
85       .reset     (reset)
86    );
87
88    ff #(1) ST2
89    (
90       .d         (start_to_stage2),    // Data in.
91       .q         (start_to_stage3),    // Data out.
92       .clk       (clk),
93       .reset     (reset)
94    );
95
96    ff #(1) ST3
97    (
98       .d         (start_to_stage3),    // Data in.
99       .q         (start_to_stage4),    // Data out.
100      .clk       (clk),
101      .reset     (reset)
102   );
103
104   // Pipeline stage 1.
105   stage1 stage1
106   (
107      .vectors         (vectors),           // Input. Vectors.
108      .start           (start),             // Input. Start computing.
109      .format          (format),            // Input. Data format.
110      .mode            (mode),              // Input. Rounding mode.
111      .DRH0_out        (DRH_to_stage2),     // Output. [127:64] of input.
112      .DRL0_out        (DRL_to_stage2),     // Output. [63:0] of input.
113      .IR0_out         (IR_to_stage2),      // Output. Format.
114      .M0_out          (M_to_stage2),       // Output. Rounding mode.
115      .clk             (clk),
116      .reset           (reset)
117   );
118
119   // Pipeline stage 2.
120   stage2 stage2
121   (
122      .DRH0            (DRH_to_stage2),     // Input from input register
             DRH0.
```

```
123         .DRL0           (DRL_to_stage2),   // Input from input register
                DRL0.
124         .format         (IR_to_stage2),    // Input from format register
                IR0.
125         .mode           (M_to_stage2),     // Input from mode register M0.
126         .DRF0_out       (DRF_to_stage3),   // Output to significand mults.
127         .DRE0_out       (DRE_to_stage3),   // Output to exponent adders.
128         .DRS0_out       (DRS_to_stage3),   // Output to sign computation.
129         .IR1_out        (IR_to_stage3),    // Output.
130         .M1_out         (M_to_stage3),     // Output.
131         .clk            (clk),
132         .reset          (reset)
133     );
134
135     // Pipeline stage 3.
136     stage3 stage3
137     (
138         .DRF0           (DRF_to_stage3),   // Input from register DRF0.
139         .DRE0           (DRE_to_stage3),   // Input from register DRE0.
140         .DRS0           (DRS_to_stage3),   // Input from register DRS0.
141         .format         (IR_to_stage3),    // Input from format register.
142         .mode           (M_to_stage3),     // Input from mode register.
143         .DRS1_out       (DRS_to_stage4),   // Output to sign register.
144         .DRE1_out       (DRE_to_stage4),   // Output to exponent register.
145         .DRF1_out       (DRF_to_stage4),   // Output to fraction register.
146         .S0_out         (S0_to_stage4),    // Output to special register.
147         .M2_out         (M_to_stage4),     // Output to mode register.
148         .IR2_out        (IR_to_stage4),    // Output to format register.
149         .clk            (clk),
150         .reset          (reset)
151     );
152
153     // Pipeline stage 4 & 5.
154     stage4 stage4
155     (
156         .start          (start_to_stage4),// Input.
157         .DRF1           (DRF_to_stage4),   // Input from register DRF1.
158         .DRE1           (DRE_to_stage4),   // Input from register DRE1.
159         .DRS1           (DRS_to_stage4),   // Input from register DRS1.
160         .specials       (S0_to_stage4),    // Input form register S0.
161         .format         (IR_to_stage4),    // Input from register IR2.
162         .mode           (M_to_stage4),     // Input from regiser M2.
163         .clear_excps    (clear),           // Input. Clear exceptions.
164         .products       (products),        // Output. Final result.
165         .exceptions     (exceptions),      // Output. Exceptions.
166         .ready          (ready),           // Output. Result ready.
167         .clk            (clk),
168         .reset          (reset)
169     );
170
171
172     // Internal active high reset.
173     assign reset = !reset_n;
174     assign load_ST0 = start;
175
176 endmodule // vec_fp_mult
```

```
1  // -----------------------------------------------------------------
2  // File.......: stage1.v
3  // Author.....: Espen Stenersen
4  // Date.......: Fri Apr 18 16:11:23 CEST 2008
5  // Revision...: 1.0
6  // Description: Stage one in pipeline.
7  // -----------------------------------------------------------------
8
9  `include "defines.v"
10
11 module stage1
12 (
13     vectors,      // Input. Vectors.
14     start,        // Input. Start computing.
15     format,       // Input. Data format.
16     mode,         // Input. Rounding mode.
17     DRH0_out,     // Output. [127:64] of input.
18     DRL0_out,     // Output. [63:0] of input.
19     IR0_out,      // Output. Format.
20     M0_out,       // Output. Rounding mode.
21     clk,
22     reset
23 );
24
25     // input(s)
26     input  [`BUS-1:0]      vectors;
27     input  [0:0]           start;
28     input  [1:0]           format;
29     input  [1:0]           mode;
30     input                  clk;
31     input                  reset;
32
33     // output(s)
34     output [`BUS/2-1:0]    DRH0_out;
35     output [`BUS/2-1:0]    DRL0_out;
36     output [1:0]           IR0_out;
37     output [1:0]           M0_out;
38
39     // wire(s)
40     wire                   load_drh;
41     wire                   load_drl;
42     wire                   load_ir0;
43     wire                   load_m0;
44
45     // reg(s)
46
47     // -------------------------------------------------------------
48     // Module instantiation.
49     // -------------------------------------------------------------
50
51     // Registers.
52     reg_enable #(`BUS/2) DRH0
53     (
54         .d       (vectors[`BUS-1:`BUS/2]),  // Data in.
55         .q       (DRH0_out),                // Data out.
56         .enable  (load_drh),                // Enable bit.
57         .clk     (clk),
58         .reset   (reset)
59     );
60
61     reg_enable #(`BUS/2) DRL0
62     (
```

```
63        .d         (vectors[`BUS/2-1:0]),   // Data in.
64        .q         (DRL0_out),              // Data out.
65        .enable    (load_drl),              // Enable bit.
66        .clk       (clk),
67        .reset     (reset)
68    );
69
70    reg_enable #(2) M0
71    (
72        .d         (mode),      // Data in.
73        .q         (M0_out),    // Data out.
74        .enable    (load_m0),   // Enable bit.
75        .clk       (clk),
76        .reset     (reset)
77    );
78
79    reg_enable #(2) IR0
80    (
81        .d         (format),     // Data in.
82        .q         (IR0_out),    // Data out.
83        .enable    (load_ir0),   // Enable bit.
84        .clk       (clk),
85        .reset     (reset)
86    );
87
88    // ------------------------------------------------------------------
89    // Assigns.
90    // ------------------------------------------------------------------
91
92    assign load_drh = start & (|format);
93    assign load_drl = start;
94    assign load_m0 = start;
95    assign load_ir0 = start;
96
97 endmodule // stage1
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: stage2.v
3 // Author.....: Espen Stenersen
4 // Date.......: Fri Apr 18 16:29:31 CEST 2008
5 // Revision...: 1.0
6 // Description: Stage two of pipeline.
7 // ---------------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module stage2
12 (
13    DRH0,         // Input from input register DRH0.
14    DRL0,         // Input from input register DRL0.
15    format,       // Input from format register IR0.
16    mode,         // Input from rounding mode register M0.
17    DRF0_out,     // Output to significand multipliers.
18    DRE0_out,     // Output to exponent adders.
19    DRS0_out,     // Output to sign computation.
20    IR1_out,      // Output.
21    M1_out,       // Output.
22    clk,
23    reset
24 );
25
26    // input(s)
27    input [`BUS/2-1:0]        DRH0;
28    input [`BUS/2-1:0]        DRL0;
29    input [1:0]               format;
30    input [1:0]               mode;
31    input                     clk;
32    input                     reset;
33
34    // output(s)
35    output [`FRACBUS-1:0]     DRF0_out;
36    output [`EXPBUS-1:0]      DRE0_out;
37    output [`SIGNBUS-1:0]     DRS0_out;
38    output [1:0]              IR1_out;
39    output [1:0]              M1_out;
40
41    // wire(s)
42    wire [`FRACBUS-1:0]       fracs;
43    wire [`EXPBUS-1:0]        exps;
44    wire [`SIGNBUS-1:0]       signs;
45
46
47    // reg(s)
48
49    // -------------------------------------------------------------------------
50    // Module instantiations.
51    // -------------------------------------------------------------------------
52
53    // Registers.
54    ff #(`FRACBUS) DRF0
55    (
56       .d        (fracs),      // Data in.
57       .q        (DRF0_out),   // Data out.
58       .clk      (clk),
59       .reset    (reset)
60    );
61
62    ff #(`EXPBUS) DRE0
```

```
63      (
64          .d          (exps),       // Data in.
65          .q          (DRE0_out),  // Data out.
66          .clk        (clk),
67          .reset      (reset)
68      );
69
70      ff #(`SIGNBUS) DRS0
71      (
72          .d          (signs),      // Data in.
73          .q          (DRS0_out),  // Data out.
74          .clk        (clk),
75          .reset      (reset)
76      );
77
78      ff #(2) IR1
79      (
80          .d          (format),     // Data in.
81          .q          (IR1_out),    // Data out.
82          .clk        (clk),
83          .reset      (reset)
84      );
85
86      ff #(2) M1
87      (
88          .d          (mode),       // Data in.
89          .q          (M1_out),     // Data out.
90          .clk        (clk),
91          .reset      (reset)
92      );
93
94      // Input mux / selector.
95      sel_input sel_input
96      (
97          .drh        (DRH0),       // Input from data-register high (DRH0).
98          .drl        (DRL0),       // Input from data-register low (DRL0).
99          .format     (format),     // Input form instrucion(format) register.
100         .signs      (signs),      // Output to sign bus.
101         .exps       (exps),       // Output to exponent bus.
102         .fracs      (fracs)       // Output to significand bus.
103     );
104
105     defparam sel_input.WIDTH    = `BUS/2;
106     defparam sel_input.SIGNBUS = `SIGNBUS;
107     defparam sel_input.EXPBUS   = `EXPBUS;
108     defparam sel_input.FRACBUS = `FRACBUS;
109
110 endmodule // stage2
```

```
1  // ----------------------------------------------------------------------------
2  // File.......: stage3.v
3  // Author.....: Espen Stenersen
4  // Date.......: Fri Apr 18 16:51:03 CEST 2008
5  // Revision...: 1.0
6  // Description: Stage three of pipeline.
7  // ----------------------------------------------------------------------------
8
9  `include "defines.v"
10
11 module stage3
12 (
13     DRF0,          // Input from fraction register.
14     DRE0,          // Input from exponent register.
15     DRS0,          // Input from sign register.
16     format,        // Input from format register.
17     mode,          // Input from rounding mode register.
18     DRS1_out,      // Output to sign register.
19     DRE1_out,      // Output to exponent register.
20     DRF1_out,      // Output to fraction register.
21     S0_out,        // Output to special values register.
22     M2_out,        // Output to rounding mode register.
23     IR2_out,       // Output to format register.
24     clk,
25     reset
26 );
27
28     // input(s)
29     input  [`FRACBUS-1:0]      DRF0;
30     input  [`EXPBUS-1:0]       DRE0;
31     input  [`SIGNBUS-1:0]      DRS0;
32     input  [1:0]               format;
33     input  [1:0]               mode;
34     input                      clk;
35     input                      reset;
36
37     // output(s)
38     output [`FRACBUS-1:0]      DRF1_out;
39     output [`SIGNBUS/2-1:0]    DRS1_out;
40     output [`EXPBUS/2+3:0]     DRE1_out; // + overflow/underflow bits.
41     output [15:0]              S0_out;
42     output [1:0]               M2_out;
43     output [1:0]               IR2_out;
44
45     // wire(s)
46     wire   [`FRACBUS-1:0]      prods;
47     wire   [`SIGNBUS/2-1:0]    signs;
48     wire   [`EXPBUS/2+3:0]     sums;
49     wire   [15:0]              specials;
50     wire   [3:0]               ints;
51     wire   [3:0]               infs;
52     wire   [3:0]               nans;
53     wire   [3:0]               zeroes;
54
55     // reg(s)
56
57     // ------------------------------------------------------------------------
58     // Module instantiations.
59     // ------------------------------------------------------------------------
60
61     // Registers.
62     ff #(`FRACBUS) DRF1
```

```
63     (
64        .d         (prods),       // Data in.
65        .q         (DRF1_out),    // Data out.
66        .clk       (clk),
67        .reset     (reset)
68     );
69
70     ff #(`EXPBUS/2+4) DRE1
71     (
72        .d         (sums),        // Data in.
73        .q         (DRE1_out),    // Data out.
74        .clk       (clk),
75        .reset     (reset)
76     );
77
78     ff #(`SIGNBUS/2) DRS1
79     (
80        .d         (signs),       // Data in.
81        .q         (DRS1_out),    // Data out.
82        .clk       (clk),
83        .reset     (reset)
84     );
85
86     ff #(16) S0
87     (
88        .d         (specials),    // Data in.
89        .q         (S0_out),      // Data out.
90        .clk       (clk),
91        .reset     (reset)
92     );
93
94     ff #(2) IR2
95     (
96        .d         (format),      // Data in.
97        .q         (IR2_out),     // Data out.
98        .clk       (clk),
99        .reset     (reset)
100    );
101
102    ff #(2) M2
103    (
104       .d         (mode),        // Data in.
105       .q         (M2_out),      // Data out.
106       .clk       (clk),
107       .reset     (reset)
108    );
109
110    // Computational units.
111    chk_special #(`FRACBUS, `EXPBUS) chk_special
112    (
113       .fracs     (DRF0),        // Input from significand bus.
114       .exps      (DRE0),        // Input from exponent bus.
115       .format    (format),      // Input.
116       .infs      (infs),        // Output.
117       .ints      (ints),        // Output.
118       .nans      (nans),        // Output.
119       .zeroes    (zeroes)       // Output.
120    );
121
122    mult_unit mult_unit
123    (
124       .fracs     (DRF0),        // Input from significand bus.
```

```
125         .format   (format),    // Input from instruction register.
126         .prods    (prods)      // Output to significand bus.
127     );
128
129     exp_unit exp_unit
130     (
131         .exps     (DRE0),      // Input from exponent bus.
132         .format   (format),    // Input from instruction register.
133         .sums     (sums)       // Output to exponent bus.
134     );
135
136     sign_unit sign_unit
137     (
138         .signs       (DRS0),   // Input signs from sign bus.
139         .signs_comp  (signs)   // Output to sign bus.
140     );
141
142     assign specials = {ints, zeroes, infs, nans};
143
144 endmodule // stage3
```

```
1 // ------------------------------------------------------------------
2 // File.......: sel_output_tb.v
3 // Author.....: Espen Stenersen
4 // Date.......: Thu Apr 24 21:39:26 CEST 2008
5 // Revision...: 1.0
6 // Description: For testing select output logic.
7 // ------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module stage4
12 (
13    start,        // Input.
14    DRF1,         // Input from fraction register DRF1.
15    DRE1,         // Input from exponent register DRE1.
16    DRS1,         // Input from sign register DRS1.
17    specials,     // Input form special values register S0.
18    format,       // Input from format register IR2.
19    mode,         // Input from rounding mode regiser M2.
20    clear_excps,// Input. Clear exceptions.
21    products,     // Output. Final result.
22    exceptions,   // Output. Exceptions.
23    ready,        // Output. Result ready.
24    clk,
25    reset
26 );
27
28    // Input(s)
29    input                     start;
30    input [`FRACBUS-1:0]      DRF1;
31    input [`EXPBUS/2+3:0]     DRE1; // + overflow/underflow bits.
32    input [`SIGNBUS/2-1:0]    DRS1;
33    input [15:0]              specials;
34    input [1:0]               format;
35    input [1:0]               mode;
36    input [15:0]              clear_excps;
37    input                     clk;
38    input                     reset;
39
40    // Output(s)
41    output [`BUS-1:0]         products;
42    output [15:0]             exceptions;
43    output                    ready;
44
45    // wire(s)
46    wire [`BUS-1:0]           products;
47    wire [15:0]               exceptions_tmp;
48    wire                      load_drh;
49    wire                      load_drlh;
50    wire                      load_drll;
51    wire                      load_excep_l;
52    wire                      load_excep_h;
53    wire [15:0]               ex;
54    wire [15:0]               clear;
55    wire [7:0]                clear_l;
56    wire [7:0]                clear_h;
57    wire [7:0]                ex_h_in;
58    wire [7:0]                ex_l_in;
59    wire [7:0]                ex_h_out;
60    wire [7:0]                ex_l_out;
61    wire [`BUS-1:0]           prods;
62    wire                      ready_tmp;
```

```
63    wire [`BUS/2-1:0]          result;
64    wire [7:0]                 exceps;
65    wire [1:0]                 format;
66
67    // reg(s)
68
69
70    // -----------------------------------------------------------------
71    // Module instantiation.
72    // -----------------------------------------------------------------
73
74    // Product registers.
75    reg_enable #(32) DRLL
76    (
77       .d              (prods[31:0]),        // Data in.
78       .q              (products[31:0]),     // Data out.
79       .enable         (load_drll),          // Enable bit.
80       .clk            (clk),
81       .reset          (reset)
82    );
83
84    reg_enable #(32) DRLH
85    (
86       .d              (prods[63:32]),       // Data in.
87       .q              (products[63:32]),    // Data out.
88       .enable         (load_drlh),          // Enable bit.
89       .clk            (clk),
90       .reset          (reset)
91    );
92
93    reg_enable #(64) DRH
94    (
95       .d              (prods[127:64]),      // Data in.
96       .q              (products[127:64]),   // Data out.
97       .enable         (load_drh),           // Enable bit.
98       .clk            (clk),
99       .reset          (reset)
100   );
101
102   // Exception registers.
103   reg_enable #(8) EXCPL
104   //reg_excep #(8) EXCPL
105   (
106      .d              (ex_l_in),          // Data in.
107      .q              (ex_l_out),         // Data out.
108      .enable         (load_excep_l),     // Enable bit.
109      //.clear        (clear_l),
110      .clk            (clk),
111      .reset          (reset)
112   );
113
114   reg_enable #(8) EXCPH
115   //reg_excep #(8) EXCPH
116   (
117      .d              (ex_h_in),          // Data in.
118      .q              (ex_h_out),         // Data out.
119      .enable         (load_excep_h),     // Enable bit.
120      //.clear        (clear_h),
121      .clk            (clk),
122      .reset          (reset)
123   );
124
```

```
125     // Clear register. Written to in order to clear exceptions.
126     // [unf p3 .. p0, ovf p3 .. p0, inx p3 .. p0, nan p3 .. p0]
127     ff #(16) CLEAR
128     (
129        .d              (clear_excps), // Data in.
130        .q              (clear),       // Data out.
131        .clk            (clk),
132        .reset          (reset)
133     );
134
135     // Ready register.
136     reg_set #(1) READY
137     (
138        .set            (ready_tmp),
139        .q              (ready),
140        .clk            (clk),
141        .reset          (reset)
142     );
143
144     // Rounding unit.
145     rne_unit rne_unit
146     (
147        .fracs          (DRF1),       // Input from fraction bus.
148        .exps           (DRE1),       // Input form exponent bus.
149        .signs          (DRS1),       // Input from sign bus.
150        .format         (format),     // Input from instrucion register.
151        .special        (specials),   // Input form check special.
152        .mode           (mode),       // Input from mode register.
153        .exceps         (exceps),     // Output exceptions.
154        .result         (result)      // Output. Rounded result.
155     );
156
157     // Output selector.
158     sel_output sel_output
159     (
160        .result         (result),        // Input from rounding logic.
161        .exceps         (exceps),        // Input from rounding logic.
162        .format         (format),        // Input from format register.
163        .start          (start),         // Input from start register.
164        .products       (prods),         // Output to output register.
165        .load_drh       (load_drh),      // Output to output register.
166        .load_drlh      (load_drlh),     // Output to output register.
167        .load_drll      (load_drll),     // Output to output register.
168        .exceptions     (ex),            // Output to exception register.
169        .load_excep_l   (load_excep_l),// Output to exception register.
170        .load_excep_h   (load_excep_h),// Output to exception register.
171        .reset          (reset),
172        .clk            (clk)
173     );
174
175     // ----------------------------------------------------------------
176     // Assigns.
177     // ----------------------------------------------------------------
178
179     assign ready_tmp =
180        (format == `FP16) ? load_drlh&start : load_drh&start;
181
182     assign clear_l =
183        {clear[13:12], clear[9:8], clear[5:4], clear[1:0]};
184     assign clear_h =
185        {clear[15:14], clear[11:10], clear[7:6], clear[3:2]};
186
```

```
187    assign ex_l_in = ex[7:0]&~clear_l;
188    assign ex_h_in = ex[15:8]&~clear_h;
189
190    assign exceptions =
191        (format == `FP64) ?
192        {1'b0, 1'b0, ex_h_out[6], ex_l_out[6],
193         1'b0, 1'b0, ex_h_out[4], ex_l_out[4],
194         1'b0, 1'b0, ex_h_out[2], ex_l_out[2],
195         1'b0, 1'b0, ex_h_out[0], ex_l_out[0]} :
196
197        {ex_h_out[7:6], ex_l_out[7:6],
198         ex_h_out[5:4], ex_l_out[5:4],
199         ex_h_out[3:2], ex_l_out[3:2],
200         ex_h_out[1:0], ex_l_out[1:0]};
201
202 endmodule // stage4
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: chk_special.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:30:08 CEST 2008
5 // Revision...: 1.0
6 // Description: Checks if inputs equals special values such as
7 //              infinity, nan, zero or int. Result is used for
8 //              exception generation.
9 // ---------------------------------------------------------------------------
10
11 `include "defines.v"
12
13 module chk_special
14 (
15    fracs,   // Input from significand bus.
16    exps,    // Input from exponent bus.
17    format,  // Input.
18    infs,    // Output.
19    ints,    // Output.
20    nans,    // Output.
21    zeroes   // Output.
22 );
23
24    parameter FRACBUS = `FRACBUS;
25    parameter EXPBUS = `EXPBUS;
26
27    // input(s)
28    input [FRACBUS-1:0]  fracs;
29    input [EXPBUS-1:0]   exps;
30    input [1:0]          format;
31
32    // output(s)
33    output [3:0]     infs;
34    output [3:0]     ints;
35    output [3:0]     nans;
36    output [3:0]     zeroes;
37
38    // wire(s)
39    wire [EXPBUS/2-1:0]   exponent_a;
40    wire [EXPBUS/2-1:0]   exponent_b;
41    wire [FRACBUS/2-1:0]  significand_a;
42    wire [FRACBUS/2-1:0]  significand_b;
43
44    wire              nan_a0;
45    wire              nan_a1;
46    wire              nan_b0;
47    wire              nan_b1;
48    wire              inf_a0;
49    wire              inf_a1;
50    wire              inf_b0;
51    wire              inf_b1;
52    wire              int_a0;
53    wire              int_a1;
54    wire              int_b0;
55    wire              int_b1;
56    wire              zero_a0;
57    wire              zero_a1;
58    wire              zero_b0;
59    wire              zero_b1;
60
61
62    // reg(s)
```

```
63
64   // ———————————————————————————————————————————————
65   // Combinational assigns.
66   // ———————————————————————————————————————————————
67
68   // fracs[1*(`FP16SW+1)-2:0*(`FP16SW+1) because significands are
69   // now extended to 11, 24 and 53 bits included the implict bit.
70
71   // Assign invalid inputs.
72   assign nan_a0 =
73       (format == `FP16) ?
74           (&exps[1*(`FP16EW)-1:0*(`FP16EW)]) &
75           (|fracs[1*(`FP16SW+1)-2:0*(`FP16SW+1)]) :
76       (format == `FP32) ?
77           (&exps[1*(`FP32EW)-1:0*(`FP32EW)]) &
78           (|fracs[1*(`FP32SW+1)-2:0*(`FP32SW+1)]) :
79       (format == `FP64) ?
80           (&exps[1*(`FP64EW)-1:0*(`FP64EW)]) &
81           (|fracs[1*(`FP64SW+1)-2:0*(`FP64SW+1)]) : 1'b0;
82
83   assign nan_b0 =
84       (format == `FP16) ?
85           (&exps[2*(`FP16EW)-1:1*(`FP16EW)]) &
86           (|fracs[2*(`FP16SW+1)-2:1*(`FP16SW+1)]) :
87       (format == `FP32) ?
88           (&exps[2*(`FP32EW)-1:1*(`FP32EW)]) &
89           (|fracs[2*(`FP32SW+1)-2:1*(`FP32SW+1)]) :
90       (format == `FP64) ?
91           (&exps[2*(`FP64EW)-1:1*(`FP64EW)]) &
92           (|fracs[2*(`FP64SW+1)-2:1*(`FP64SW+1)]) : 1'b0;
93
94   assign nan_a1 =
95       (format == `FP16) ?
96           (&exps[3*(`FP16EW)-1:2*(`FP16EW)]) &
97           (|fracs[3*(`FP16SW+1)-2:2*(`FP16SW+1)]) :
98       (format == `FP32) ?
99           (&exps[3*(`FP32EW)-1:2*(`FP32EW)]) &
100          (|fracs[3*(`FP32SW+1)-2:2*(`FP32SW+1)]) :
101      (format == `FP64) ? 1'b0 : 1'b0;
102
103  assign nan_b1 =
104      (format == `FP16) ?
105          (&exps[4*(`FP16EW)-1:3*(`FP16EW)]) &
106          (|fracs[4*(`FP16SW+1)-2:3*(`FP16SW+1)]) :
107      (format == `FP32) ?
108          (&exps[4*(`FP32EW)-1:3*(`FP32EW)]) &
109          (|fracs[4*(`FP32SW+1)-2:3*(`FP32SW+1)]) :
110      (format == `FP64) ? 1'b0 : 1'b0;
111
112
113  // Assign infinity inputs.
114  assign inf_a0 =
115      (format == `FP16) ?
116          (&exps[1*(`FP16EW)-1:0*(`FP16EW)]) &
117          (~|fracs[1*(`FP16SW+1)-2:0*(`FP16SW+1)]) :
118      (format == `FP32) ?
119          (&exps[1*(`FP32EW)-1:0*(`FP32EW)]) &
120          (~|fracs[1*(`FP32SW+1)-2:0*(`FP32SW+1)]) :
121      (format == `FP64) ?
122          (&exps[1*(`FP64EW)-1:0*(`FP64EW)]) &
123          (~|fracs[1*(`FP64SW+1)-2:0*(`FP64SW+1)]) : 1'b0;
124
```

```
125     assign inf_b0 =
126         (format == `FP16) ?
127             (&exps[2*(`FP16EW)-1:1*(`FP16EW)]) &
128             (~|fracs[2*(`FP16SW+1)-2:1*(`FP16SW+1)]) :
129         (format == `FP32) ?
130             (&exps[2*(`FP32EW)-1:1*(`FP32EW)]) &
131             (~|fracs[2*(`FP32SW+1)-2:1*(`FP32SW+1)]) :
132         (format == `FP64) ?
133             (&exps[2*(`FP64EW)-1:1*(`FP64EW)]) &
134             (~|fracs[2*(`FP64SW+1)-2:1*(`FP64SW+1)]) : 1'b0;
135
136     assign inf_a1 =
137         (format == `FP16) ?
138             (&exps[3*(`FP16EW)-1:2*(`FP16EW)]) &
139             (~|fracs[3*(`FP16SW+1)-2:2*(`FP16SW+1)]) :
140         (format == `FP32) ?
141             (&exps[3*(`FP32EW)-1:2*(`FP32EW)]) &
142             (~|fracs[3*(`FP32SW+1)-2:2*(`FP32SW+1)]) :
143         (format == `FP64) ? 1'b0 : 1'b0;
144
145     assign inf_b1 =
146         (format == `FP16) ?
147             (&exps[4*(`FP16EW)-1:3*(`FP16EW)]) &
148             (~|fracs[4*(`FP16SW+1)-2:3*(`FP16SW+1)]) :
149         (format == `FP32) ?
150             (&exps[4*(`FP32EW)-1:3*(`FP32EW)]) &
151             (~|fracs[4*(`FP32SW+1)-2:3*(`FP32SW+1)]) :
152         (format == `FP64) ? 1'b0 : 1'b0;
153
154
155     // Assign zero inputs.
156     assign zero_a0 =
157         (format == `FP16) ?
158             (~|exps[1*(`FP16EW)-1:0*(`FP16EW)]) &
159             (~|fracs[1*(`FP16SW+1)-2:0*(`FP16SW+1)]) :
160         (format == `FP32) ?
161             (~|exps[1*(`FP32EW)-1:0*(`FP32EW)]) &
162             (~|fracs[1*(`FP32SW+1)-2:0*(`FP32SW+1)]) :
163         (format == `FP64) ?
164             (~|exps[1*(`FP64EW)-1:0*(`FP64EW)]) &
165             (~|fracs[1*(`FP64SW+1)-2:0*(`FP64SW+1)]) : 1'b0;
166
167     assign zero_b0 =
168         (format == `FP16) ?
169             (~|exps[2*(`FP16EW)-1:1*(`FP16EW)]) &
170             (~|fracs[2*(`FP16SW+1)-2:1*(`FP16SW+1)]) :
171         (format == `FP32) ?
172             (~|exps[2*(`FP32EW)-1:1*(`FP32EW)]) &
173             (~|fracs[2*(`FP32SW+1)-2:1*(`FP32SW+1)]) :
174         (format == `FP64) ?
175             (~|exps[2*(`FP64EW)-1:1*(`FP64EW)]) &
176             (~|fracs[2*(`FP64SW+1)-2:1*(`FP64SW+1)]) : 1'b0;
177
178     assign zero_a1 =
179         (format == `FP16) ?
180             (~|exps[3*(`FP16EW)-1:2*(`FP16EW)]) &
181             (~|fracs[3*(`FP16SW+1)-2:2*(`FP16SW+1)]) :
182         (format == `FP32) ?
183             (~|exps[3*(`FP32EW)-1:2*(`FP32EW)]) &
184             (~|fracs[3*(`FP32SW+1)-2:2*(`FP32SW+1)]) :
185         (format == `FP64) ? 1'b0 : 1'b0;
186
```

```
187    assign zero_b1 =
188        (format == `FP16) ?
189            (~|exps[4*(`FP16EW)-1:3*(`FP16EW)]) &
190            (~|fracs[4*(`FP16SW+1)-2:3*(`FP16SW+1)]) :
191        (format == `FP32) ?
192            (~|exps[4*(`FP32EW)-1:3*(`FP32EW)]) &
193            (~|fracs[4*(`FP32SW+1)-2:3*(`FP32SW+1)]) :
194        (format == `FP64) ? 1'b0 : 1'b0;
195
196
197    // Assign integer inputs.
198    assign int_a0 =
199        (format == `FP16) ?
200            (|exps[1*(`FP16EW)-1:0*(`FP16EW)]) &
201            (~|fracs[1*(`FP16SW+1)-2:0*(`FP16SW+1)]) :
202        (format == `FP32) ?
203            (|exps[1*(`FP32EW)-1:0*(`FP32EW)]) &
204            (~|fracs[1*(`FP32SW+1)-2:0*(`FP32SW+1)]) :
205        (format == `FP64) ?
206            (|exps[1*(`FP64EW)-1:0*(`FP64EW)]) &
207            (~|fracs[1*(`FP64SW+1)-2:0*(`FP64SW+1)]) : 1'b0;
208
209    assign int_b0 =
210        (format == `FP16) ?
211            (|exps[2*(`FP16EW)-1:1*(`FP16EW)]) &
212            (~|fracs[2*(`FP16SW+1)-2:1*(`FP16SW+1)]) :
213        (format == `FP32) ?
214            (|exps[2*(`FP32EW)-1:1*(`FP32EW)]) &
215            (~|fracs[2*(`FP32SW+1)-2:1*(`FP32SW+1)]) :
216        (format == `FP64) ?
217            (|exps[2*(`FP64EW)-1:1*(`FP64EW)]) &
218            (~|fracs[2*(`FP64SW+1)-2:1*(`FP64SW+1)]) : 1'b0;
219
220    assign int_a1 =
221        (format == `FP16) ?
222            (|exps[3*(`FP16EW)-1:2*(`FP16EW)]) &
223            (~|fracs[3*(`FP16SW+1)-2:2*(`FP16SW+1)]) :
224        (format == `FP32) ?
225            (|exps[3*(`FP32EW)-1:2*(`FP32EW)]) &
226            (~|fracs[3*(`FP32SW+1)-2:2*(`FP32SW+1)]) :
227        (format == `FP64) ? 1'b0 : 1'b0;
228
229    assign int_b1 =
230        (format == `FP16) ?
231            (|exps[4*(`FP16EW)-1:3*(`FP16EW)]) &
232            (~|fracs[4*(`FP16SW+1)-2:3*(`FP16SW+1)]) :
233        (format == `FP32) ?
234            (|exps[4*(`FP32EW)-1:3*(`FP32EW)]) &
235        (~|fracs[4*(`FP32SW+1)-2:3*(`FP32SW+1)]) :
236            (format == `FP64) ? 1'b0 : 1'b0;
237
238
239    // Assign outputs.
240    assign infs[0] = inf_a0;
241    assign infs[1] = inf_b0;
242    assign infs[2] = inf_a1;
243    assign infs[3] = inf_b1;
244    assign ints[0] = int_a0;
245    assign ints[1] = int_b0;
246    assign ints[2] = int_a1;
247    assign ints[3] = int_b1;
248    assign nans[0] = nan_a0;
```

```
249     assign nans[1] = nan_b0;
250     assign nans[2] = nan_a1;
251     assign nans[3] = nan_b1;
252     assign zeroes[0] = zero_a0;
253     assign zeroes[1] = zero_b0;
254     assign zeroes[2] = zero_a1;
255     assign zeroes[3] = zero_b1;
256
257 endmodule // chk_special
```

```
1 // ————————————————————————————————————————————————————————————————
2 // File.......: exp_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:40:17 CEST 2008
5 // Revision...: 1.0
6 // Description: Exponent adder unit.
7 // ————————————————————————————————————————————————————————————————
8
9 `include "defines.v"
10
11 module exp_unit
12 (
13    exps,     // Input from exponent bus.
14    format,   // Input from instruction register.
15    sums      // Output to exponent bus.
16 );
17
18
19    // input(s)
20    input  [`EXPBUS-1:0]       exps;
21    input  [1:0]               format;
22
23    // output(s)
24    output [`EXPBUS/2+3:0]     sums;
25
26    // wire(s)
27    wire [`FP16EW-1:0]         fp16_a_0;
28    wire [`FP16EW-1:0]         fp16_b_0;
29    wire [`FP16EW-1:0]         fp16_sum_0;
30    wire                       fp16_ovf_ab_0;
31    wire                       fp16_ovf_biased_0;
32    wire [`FP16EW-1:0]         fp16_a_1;
33    wire [`FP16EW-1:0]         fp16_b_1;
34    wire [`FP16EW-1:0]         fp16_sum_1;
35    wire                       fp16_ovf_ab_1;
36    wire                       fp16_ovf_biased_1;
37    wire [`FP32EW-1:0]         fp32_a_0;
38    wire [`FP32EW-1:0]         fp32_b_0;
39    wire [`FP32EW-1:0]         fp32_sum_0;
40    wire                       fp32_ovf_ab_0;
41    wire                       fp32_ovf_biased_0;
42    wire [`FP32EW-1:0]         fp32_a_1;
43    wire [`FP32EW-1:0]         fp32_b_1;
44    wire [`FP32EW-1:0]         fp32_sum_1;
45    wire                       fp32_ovf_ab_1;
46    wire                       fp32_ovf_biased_1;
47    wire [`FP64EW-1:0]         fp64_a_0;
48    wire [`FP64EW-1:0]         fp64_b_0;
49    wire [`FP64EW-1:0]         fp64_sum_0;
50    wire                       fp64_ovf_ab_0;
51    wire                       fp64_ovf_biased_0;
52
53    // reg(s)
54
55    // ————————————————————————————————————————————————————————————
56    // Module instantiation.
57    // ————————————————————————————————————————————————————————————
58    exp_add #(`FP16EW, `FP16BIAS) fp16_add_0
59    (
60       .a          (fp16_a_0),
61       .b          (fp16_b_0),
62       .sum        (fp16_sum_0),
```

```
63      .ovf_ab      (fp16_ovf_ab_0),
64      .ovf_biased  (fp16_ovf_biased_0)
65  );
66
67  exp_add #(`FP16EW, `FP16BIAS) fp16_add_1
68  (
69      .a           (fp16_a_1),
70      .b           (fp16_b_1),
71      .sum         (fp16_sum_1),
72      .ovf_ab      (fp16_ovf_ab_1),
73      .ovf_biased  (fp16_ovf_biased_1)
74  );
75
76  exp_add #(`FP32EW, `FP32BIAS) fp32_add_0
77  (
78      .a           (fp32_a_0),
79      .b           (fp32_b_0),
80      .sum         (fp32_sum_0),
81      .ovf_ab      (fp32_ovf_ab_0),
82      .ovf_biased  (fp32_ovf_biased_0)
83  );
84
85  exp_add #(`FP32EW, `FP32BIAS) fp32_add_1
86  (
87      .a           (fp32_a_1),
88      .b           (fp32_b_1),
89      .sum         (fp32_sum_1),
90      .ovf_ab      (fp32_ovf_ab_1),
91      .ovf_biased  (fp32_ovf_biased_1)
92  );
93
94  exp_add #(`FP64EW, `FP64BIAS) fp64_add_0
95  (
96      .a           (fp64_a_0),
97      .b           (fp64_b_0),
98      .sum         (fp64_sum_0),
99      .ovf_ab      (fp64_ovf_ab_0),
100     .ovf_biased  (fp64_ovf_biased_0)
101 );
102
103 // ------------------------------------------------------------------------
104 // Combinational assign.
105 // ------------------------------------------------------------------------
106
107 // Input demux.
108 assign fp16_a_0 =
109     (format == `FP16) ? exps[1*`FP16EW-1:0*`FP16EW] : 0;
110 assign fp16_b_0 =
111     (format == `FP16) ? exps[2*`FP16EW-1:1*`FP16EW] : 0;
112 assign fp16_a_1 =
113     (format == `FP16) ? exps[3*`FP16EW-1:2*`FP16EW] : 0;
114 assign fp16_b_1 =
115     (format == `FP16) ? exps[4*`FP16EW-1:3*`FP16EW] : 0;
116 assign fp32_a_0 =
117     (format == `FP32) ? exps[1*`FP32EW-1:0*`FP32EW] : 0;
118 assign fp32_b_0 =
119     (format == `FP32) ? exps[2*`FP32EW-1:1*`FP32EW] : 0;
120 assign fp32_a_1 =
121     (format == `FP32) ? exps[3*`FP32EW-1:2*`FP32EW] : 0;
122 assign fp32_b_1 =
123     (format == `FP32) ? exps[4*`FP32EW-1:3*`FP32EW] : 0;
124 assign fp64_a_0 =
```

```
125         (format == `FP64) ? exps[1*`FP64EW-1:0*`FP64EW] : 0;
126     assign fp64_b_0 =
127         (format == `FP64) ? exps[2*`FP64EW-1:1*`FP64EW] : 0;
128
129     // Output mux.
130     assign sums =
131         (format == `FP16) ?
132             {fp16_ovf_ab_1, fp16_ovf_biased_1, fp16_sum_1,
133              fp16_ovf_ab_0, fp16_ovf_biased_0, fp16_sum_0} :
134         (format == `FP32) ?
135             {fp32_ovf_ab_1, fp32_ovf_biased_1, fp32_sum_1,
136              fp32_ovf_ab_0, fp32_ovf_biased_0, fp32_sum_0} :
137         (format == `FP64) ?
138             {fp64_ovf_ab_0, fp64_ovf_biased_0, fp64_sum_0} : 0;
139
140 endmodule // exp_unit
```

```
1  // ---------------------------------------------------------------------------
2  // File.......:  exp_add.v
3  // Author.....:  Espen Stenersen
4  // Date.......:  Tue Apr 15 10:44:27 CEST 2008
5  // Revision...:  1.0
6  // Description:  Exponent adder. Adds the two inputs, and subtracts
7  //               the bias.
8  // ---------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module exp_add
13 (
14    a,            // Input operand.
15    b,            // Input operand.
16    sum,          // Output sum.
17    ovf_ab,       // Overflow after addition.
18    ovf_biased    // Overflow after subtraction.
19 );
20
21    parameter WIDTH = 1;
22    parameter BIAS = `FP32BIAS;
23
24    // input(s)
25    input  [WIDTH-1:0]      a;
26    input  [WIDTH-1:0]      b;
27
28    // output(s)
29    output  [WIDTH-1:0]     sum;
30    output                  ovf_ab;
31    output                  ovf_biased;
32
33    // wire(s)
34    wire   [WIDTH:0]        a_plus_b_tmp;
35    wire   [WIDTH:0]        biased_tmp;
36
37    assign a_plus_b_tmp = a + b;
38    assign biased_tmp = a_plus_b_tmp - BIAS;
39
40    assign sum = biased_tmp[WIDTH-1:0];
41    assign ovf_ab = a_plus_b_tmp[WIDTH];
42    assign ovf_biased = biased_tmp[WIDTH];
43
44 endmodule // exp_add
```

```
1 // ------------------------------------------------------------
2 // File.......: mult_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:37:56 CEST 2008
5 // Revision...: 1.0
6 // Description: Significand multiplier unit.
7 // ------------------------------------------------------------
8
9 `include "defines.v"
10
11 module mult_unit
12 (
13    fracs,    // Input from significand bus.
14    format,   // Input from instruction register.
15    prods     // Output to significand bus.
16 );
17
18    // input(s)
19    input [`FRACBUS-1:0]  fracs;
20    input [1:0]           format;
21
22    // output(s)
23    output [`FRACBUS-1:0]    prods;
24
25    // wire(s)
26    wire [`FP16SW:0]       fp16_a_0;
27    wire [`FP16SW:0]       fp16_b_0;
28    wire [2*`FP16SW+1:0]   fp16_p_0;
29    wire [`FP16SW:0]       fp16_a_1;
30    wire [`FP16SW:0]       fp16_b_1;
31    wire [2*`FP16SW+1:0]   fp16_p_1;
32    wire [`FP32SW:0]       fp32_a_0;
33    wire [`FP32SW:0]       fp32_b_0;
34    wire [2*`FP32SW+1:0]   fp32_p_0;
35    wire [`FP32SW:0]       fp32_a_1;
36    wire [`FP32SW:0]       fp32_b_1;
37    wire [2*`FP32SW+1:0]   fp32_p_1;
38    wire [`FP64SW:0]       fp64_a_0;
39    wire [`FP64SW:0]       fp64_b_0;
40    wire [2*`FP64SW+1:0]   fp64_p_0;
41
42    // reg(s)
43
44    //
      ------------------------------------------------------------
45    // Module instantiations.
46    //
      ------------------------------------------------------------
47
48    uns_mult #(`FP16SW+1) uns_mult_fp16_0
49    (
50       .a(fp16_a_0),
51       .b(fp16_b_0),
52       .p(fp16_p_0)
53    );
54
55    uns_mult #(`FP16SW+1) uns_mult_fp16_1
56    (
57       .a(fp16_a_1),
58       .b(fp16_b_1),
```

```
59          .p(fp16_p_1)
60      );
61
62      uns_mult #(`FP32SW+1) uns_mult_fp32_0
63      (
64          .a(fp32_a_0),
65          .b(fp32_b_0),
66          .p(fp32_p_0)
67      );
68
69      uns_mult #(`FP32SW+1) uns_mult_fp32_1
70      (
71          .a(fp32_a_1),
72          .b(fp32_b_1),
73          .p(fp32_p_1)
74      );
75
76      uns_mult #(`FP64SW+1) uns_mult_fp64_0
77      (
78          .a(fp64_a_0),
79          .b(fp64_b_0),
80          .p(fp64_p_0)
81      );
82
83      // ---------------------------------------------------------------------
84      // Combinaional assigns.
85      // ---------------------------------------------------------------------
86
87      // Input demux.
88      assign fp16_a_0 =
89          (format == `FP16) ? fracs[1*(`FP16SW+1)-1:0*(`FP16SW+1)] : 0;
90      assign fp16_b_0 =
91          (format == `FP16) ? fracs[2*(`FP16SW+1)-1:1*(`FP16SW+1)] : 0;
92      assign fp16_a_1 =
93          (format == `FP16) ? fracs[3*(`FP16SW+1)-1:2*(`FP16SW+1)] : 0;
94      assign fp16_b_1 =
95          (format == `FP16) ? fracs[4*(`FP16SW+1)-1:3*(`FP16SW+1)] : 0;
96
97      assign fp32_a_0 =
98          (format == `FP32) ? fracs[1*(`FP32SW+1)-1:0*(`FP32SW+1)] : 0;
99      assign fp32_b_0 =
100         (format == `FP32) ? fracs[2*(`FP32SW+1)-1:1*(`FP32SW+1)] : 0;
101     assign fp32_a_1 =
102         (format == `FP32) ? fracs[3*(`FP32SW+1)-1:2*(`FP32SW+1)] : 0;
103     assign fp32_b_1 =
104         (format == `FP32) ? fracs[4*(`FP32SW+1)-1:3*(`FP32SW+1)] : 0;
105
106     assign fp64_a_0 =
107         (format == `FP64) ? fracs[1*(`FP64SW+1)-1:0*(`FP64SW+1)] : 0;
108     assign fp64_b_0 =
109         (format == `FP64) ? fracs[2*(`FP64SW+1)-1:1*(`FP64SW+1)] : 0;
110
111     // Output mux.
112     assign prods =
113         (format == `FP16) ? {fp16_p_1, fp16_p_0} :
114         (format == `FP32) ? {fp32_p_1, fp32_p_0} :
115         (format == `FP64) ? fp64_p_0 : 0;
116
117 endmodule // mult_unit
```

```
1 // ------------------------------------------------------------------------
2 // File.......: uns_mult.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 10:40:36 CEST 2008
5 // Revision...: 1.0
6 // Description: Unsigned multiplier used for significand
7 //              multiplication.
8 // ------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module uns_mult
13 (
14    a, // Input, multiplicand.
15    b, // Input, multiplier.
16    p  // Output, product.
17 );
18
19
20    parameter WIDTH = `FP64SW+1;
21
22    // input(s)
23    input [WIDTH-1:0]     a;
24    input [WIDTH-1:0]     b;
25
26    // output(s)
27    output [2*WIDTH-1:0] p;
28
29    assign p = a * b;
30
31 endmodule // uns_mult
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: sign_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:42:25 CEST 2008
5 // Revision...: 1.0
6 // Description: Sign computation unit.
7 // ---------------------------------------------------------------------------
8
9 module sign_unit
10 (
11    signs,        // Input signs from sign bus.
12    signs_comp    // Output to sign bus.
13 );
14
15    parameter SIGNBUS = 4;
16
17    // input(s)
18    input  [SIGNBUS-1:0]      signs;
19
20    // output(s)
21    output [SIGNBUS/2-1:0]    signs_comp;
22
23    // wire(s)
24
25    // reg(s)
26
27    assign signs_comp[0] = signs[0] ^ signs[1];
28    assign signs_comp[1] = signs[2] ^ signs[3];
29
30 endmodule // sign_unit
```

```
1 // ----------------------------------------------------------------
2 // File.......: rne_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 12:19:50 CEST 2008
5 // Revision...: 1.0
6 // Description: Rounding, normalizing and exception unit.
7 // ----------------------------------------------------------------
8
9 `include "defines.v"
10
11 module rne_unit
12 (
13    fracs,    // Input from fraction bus.
14    exps,     // Input form exponent bus.
15    signs,    // Input from sign bus.
16    format,   // Input from instrucion register.
17    special,  // Input form check special.
18    mode,     // Input from mode register.
19    exceps,   // Output exceptions.
20    result    // Output. Rounded result.
21 );
22
23    // input(s)
24    input [`FRACBUS-1:0]          fracs;
25    input [`EXPBUS/2+3:0]         exps;
26    input [`SIGNBUS/2-1:0]        signs;
27    input [1:0]                   format;
28    input [1:0]                   mode;
29    input [15:0]                  special;
30
31    // output(s)
32    output [`BUS/2-1:0]           result;
33    output [7:0]                  exceps;
34
35    // wire(s)
36    wire [2*`FP64SW+1:0]          frac_rne_0;
37    wire                          sign_rne_0;
38    wire [`FP64EW+1:0]            exp_rne_0;
39    wire [7:0]                    specials_rne_0;
40    wire [`FP64SW+`FP64EW:0]      result_rne_0;
41    wire [3:0]                    exceps_rne_0;
42    wire [2*`FP64SW+1:0]          frac_rne_1;
43    wire                          sign_rne_1;
44    wire [`FP64EW+1:0]            exp_rne_1;
45    wire [7:0]                    specials_rne_1;
46    wire [`FP64SW+`FP64EW:0]      result_rne_1;
47    wire [3:0]                    exceps_rne_1;
48    wire [1:0]                    fp16_overflow;
49    wire [1:0]                    fp16_underflow;
50    wire [1:0]                    fp16_inexact;
51    wire [1:0]                    fp16_invalid;
52    wire [1:0]                    fp32_overflow;
53    wire [1:0]                    fp32_underflow;
54    wire [1:0]                    fp32_inexact;
55    wire [1:0]                    fp32_invalid;
56    wire [1:0]                    fp64_overflow;
57    wire [1:0]                    fp64_underflow;
58    wire [1:0]                    fp64_inexact;
59    wire [1:0]                    fp64_invalid;
60    wire [3:0]                    exceps_fp16_rne_0;
61    wire [3:0]                    exceps_fp16_rne_1;
62    wire [3:0]                    exceps_fp32_rne_0;
```

```
63    wire [3:0]              exceps_fp32_rne_1;
64    wire [3:0]              exceps_fp64_rne_0;
65    wire [7:0]              fp16_exceps;
66    wire [7:0]              fp32_exceps;
67    wire [7:0]              fp64_exceps;
68    wire [2*`FP16SW+1:0]    frac_fp16_rne_0;
69    wire [2*`FP16SW+1:0]    frac_fp16_rne_1;
70    wire [2*`FP32SW+1:0]    frac_fp32_rne_0;
71    wire [2*`FP32SW+1:0]    frac_fp32_rne_1;
72    wire [2*`FP64SW+1:0]    frac_fp64_rne_0;
73    wire [`FP16EW+1:0]      exp_fp16_rne_0;
74    wire [`FP16EW+1:0]      exp_fp16_rne_1;
75    wire [`FP32EW+1:0]      exp_fp32_rne_0;
76    wire [`FP32EW+1:0]      exp_fp32_rne_1;
77    wire [`FP64EW+1:0]      exp_fp64_rne_0;
78    wire [`FP16W-1:0]       result_fp16_rne_0;
79    wire [`FP16W-1:0]       result_fp16_rne_1;
80    wire [`FP32W-1:0]       result_fp32_rne_0;
81    wire [`FP32W-1:0]       result_fp32_rne_1;
82    wire [`FP64W-1:0]       result_fp64_rne_0;
83    wire [2*`FP16W-1:0]     result_fp16_rne;
84    wire [2*`FP32W-1:0]     result_fp32_rne;
85    wire [2*`FP64W-1:0]     result_fp64_rne;
86
87    // reg(s)
88
89    // ------------------------------------------------------------------------
90    // Module instantiation.
91    // ------------------------------------------------------------------------
92
93    rne #(`FP16SW, `FP16EW) fp16_rne_0
94    (
95        .frac       (frac_fp16_rne_0),
96        .sign       (sign_rne_0),
97        .exp        (exp_fp16_rne_0),
98        .specials   (specials_rne_0),
99        .mode       (mode),
100       .result     (result_fp16_rne_0),
101       .exceps     (exceps_fp16_rne_0)
102   );
103
104   rne #(`FP16SW, `FP16EW) fp16_rne_1
105   (
106       .frac       (frac_fp16_rne_1),
107       .sign       (sign_rne_1),
108       .exp        (exp_fp16_rne_1),
109       .specials   (specials_rne_1),
110       .mode       (mode),
111       .result     (result_fp16_rne_1),
112       .exceps     (exceps_fp16_rne_1)
113   );
114
115   rne #(`FP32SW, `FP32EW) fp32_rne_0
116   (
117       .frac       (frac_fp32_rne_0),
118       .sign       (sign_rne_0),
119       .exp        (exp_fp32_rne_0),
120       .specials   (specials_rne_0),
121       .mode       (mode),
122       .result     (result_fp32_rne_0),
123       .exceps     (exceps_fp32_rne_0)
124   );
```

```
125
126     rne #(`FP32SW, `FP32EW) fp32_rne_1
127     (
128         .frac        (frac_fp32_rne_1),
129         .sign        (sign_rne_1),
130         .exp         (exp_fp32_rne_1),
131         .specials    (specials_rne_1),
132         .mode        (mode),
133         .result      (result_fp32_rne_1),
134         .exceps      (exceps_fp32_rne_1)
135     );
136
137     rne #(`FP64SW, `FP64EW) fp64_rne_0
138     (
139         .frac        (frac_fp64_rne_0),
140         .sign        (sign_rne_0),
141         .exp         (exp_fp64_rne_0),
142         .specials    (specials_rne_0),
143         .mode        (mode),
144         .result      (result_fp64_rne_0),
145         .exceps      (exceps_fp64_rne_0)
146     );
147
148
149     // ---------------------------------------------------------------------
150     // Combinalional assign.
151     // ---------------------------------------------------------------------
152
153     // Inputs to rounding logic.
154     assign frac_fp16_rne_0 = fracs[2*(`FP16SW+1)-1:0*(`FP16SW+1)];
155     assign frac_fp16_rne_1 = fracs[4*(`FP16SW+1)-1:2*(`FP16SW+1)];
156     assign frac_fp32_rne_0 = fracs[2*(`FP32SW+1)-1:0*(`FP32SW+1)];
157     assign frac_fp32_rne_1 = fracs[4*(`FP32SW+1)-1:2*(`FP32SW+1)];
158     assign frac_fp64_rne_0 = fracs[2*(`FP64SW+1)-1:0*(`FP64SW+1)];
159
160     // Two msb bits represents the overflow bits during exponent
161     // addition.
162     assign exp_fp16_rne_0 = exps[1*(`FP16EW+1):0*(`FP16EW+2)];
163     assign exp_fp16_rne_1 = exps[2*(`FP16EW+1)+1:1*(`FP16EW+2)];
164     assign exp_fp32_rne_0 = exps[1*(`FP32EW+1):0*(`FP32EW+2)];
165     assign exp_fp32_rne_1 = exps[2*(`FP32EW+1)+1:1*(`FP32EW+2)];
166     assign exp_fp64_rne_0 = exps[1*(`FP64EW+1):0*(`FP64EW+2)];
167
168     assign sign_rne_0 = signs[0];
169     assign sign_rne_1 = signs[1];
170
171     assign specials_rne_0 =
172         {special[13], special[12], special[9], special[8],
173          special[5], special[4], special[1], special[0]};
174
175     assign specials_rne_1 =
176         {special[15], special[14], special[11], special[10],
177          special[7], special[6], special[3], special[2]};
178
179
180     // Outputs from rounding logic.
181     assign result_fp16_rne = {result_fp16_rne_1, result_fp16_rne_0};
182     assign result_fp32_rne = {result_fp32_rne_1, result_fp32_rne_0};
183     assign result_fp64_rne = result_fp64_rne_0;
184
185     assign fp16_underflow =
186         {exceps_fp16_rne_1[3], exceps_fp16_rne_0[3]};
```

```
187     assign fp16_overflow =
188         {exceps_fp16_rne_1[2], exceps_fp16_rne_0[2]};
189     assign fp16_inexact =
190         {exceps_fp16_rne_1[1], exceps_fp16_rne_0[1]};
191     assign fp16_invalid =
192         {exceps_fp16_rne_1[0], exceps_fp16_rne_0[0]};
193
194     assign fp32_underflow =
195         {exceps_fp32_rne_1[3], exceps_fp32_rne_0[3]};
196     assign fp32_overflow =
197         {exceps_fp32_rne_1[2], exceps_fp32_rne_0[2]};
198     assign fp32_inexact =
199         {exceps_fp32_rne_1[1], exceps_fp32_rne_0[1]};
200     assign fp32_invalid =
201         {exceps_fp32_rne_1[0], exceps_fp32_rne_0[0]};
202
203     assign fp64_underflow =
204         {1'b0, exceps_fp64_rne_0[3]};
205     assign fp64_overflow =
206         {1'b0, exceps_fp64_rne_0[2]};
207     assign fp64_inexact =
208         {1'b0, exceps_fp64_rne_0[1]};
209     assign fp64_invalid =
210         {1'b0, exceps_fp64_rne_0[0]};
211
212     assign fp16_exceps =
213         {fp16_underflow, fp16_overflow, fp16_inexact, fp16_invalid};
214     assign fp32_exceps =
215         {fp32_underflow, fp32_overflow, fp32_inexact, fp32_invalid};
216     assign fp64_exceps =
217         {fp64_underflow, fp64_overflow, fp64_inexact, fp64_invalid};
218
219     assign exceps =
220         (format == `FP16) ? fp16_exceps :
221         (format == `FP32) ? fp32_exceps :
222         (format == `FP64) ? fp64_exceps : 0;
223
224     assign result =
225         (format == `FP16) ? result_fp16_rne :
226         (format == `FP32) ? result_fp32_rne :
227         (format == `FP64) ? result_fp64_rne : 0;
228
229 endmodule // rne_unit
```

```
1 // ----------------------------------------------------------------------------
2 // File.......: rne.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:10:54 CEST 2008
5 // Revision...: 1.0
6 // Description: Rounding and exception unit. Rounds, normalizes and
7 //              postnormlizes the result from the computation, and
8 //              generates exceptions if needed.
9 // ----------------------------------------------------------------------------
10
11 `include "defines.v"
12
13 module rne
14 (
15    frac,       // Input. Fractional part from significand
        multiplication.
16    sign,       // Input. Sign from sign computation.
17    exp,        // Input. Biased exponent from exponent addition.
18    specials,   // Input. NaNs, infinities, zeros..
19    mode,       // Input. Rounding mode.
20    result,     // Output. Rounded result or special value.
21    exceps      // Output. Exceptions.
22 );
23
24    parameter SW = 52;
25    parameter EW = 11;
26
27    // input(s)
28    input [2*SW+1:0]   frac;
29    input [EW+1:0]     exp;
30    input              sign;
31    input [7:0]        specials;
32    input [1:0]        mode;
33
34    // output(s)
35    output [SW+EW:0]   result;
36    output [3:0]       exceps;
37
38    // wire(s)
39    wire               normalize;
40    wire               postnormalize;
41    wire               lsb;
42    wire               round;
43    wire               sticky;
44    wire               roundup;
45    wire               rounded;
46    wire               ovf_ab;
47    wire               ovf_biased;
48    wire               ovf_postnorm;
49    wire               round_to_nearest_even;
50    wire               round_to_infinity;
51    wire               round_to_zero;
52    wire               nan_a;
53    wire               nan_b;
54    wire               int_a;
55    wire               int_b;
56    wire               inf_a;
57    wire               inf_b;
58    wire               zero_a;
59    wire               zero_b;
60    wire               int_times_inf;
61    wire               invalid;
```

```
62    wire                overflow;
63    wire                overflow_tmp;
64    wire                underflow;
65    wire                underflow_tmp;
66    wire                inexact;
67    wire                exp_zero;
68    wire  [SW:0]        significand;
69    wire  [SW:0]        significand_tmp;
70    wire  [SW:0]        significand_plus_ulp;
71    wire  [EW:0]        exponent;
72    wire  [SW+EW:0]     result_tmp;
73    wire  [SW+EW:0]     product_nan;
74    wire  [SW+EW:0]     product_zero;
75    wire  [SW+EW:0]     product_large;
76    wire  [SW+EW:0]     product_overflow;
77
78    // reg(s)
79
80    // Round and normalize / Postnormalize.
81    // ---------------------------------------------------------------------
82
83    // Normalize if result from multiplier lies in [2,4)
84    assign normalize = frac[2*SW+1];
85
86    assign significand_tmp =
87        normalize ?
88        frac[2*SW:SW] >> 1: frac[2*SW:SW];
89
90    assign exponent =
91        normalize ?
92        exp[EW-1:0] + 1 : exp[EW-1:0];
93
94
95    // Assign rounding bits.
96    assign lsb =
97        normalize ?
98        frac[SW+1] :
99        frac[SW];
100
101   assign round =
102       normalize ?
103       frac[SW] :
104       frac[SW-1];
105
106   assign sticky =
107       normalize ?
108       |frac[SW-1:0] :
109       |frac[SW-2:0];
110
111   // Reduce to three rounding modes.
112   assign round_to_nearest_even =
113       (round & (lsb | sticky)) & !(|mode);
114
115   assign round_to_infinity =
116       (!sign&(!mode[1]&mode[0])|sign&(mode[1]&!mode[0])) &
117       (round|sticky);
118
119   assign round_to_zero =
120       (sign&(~mode[1]&mode[0])|~sign&(mode[1]&~mode[0]))|&mode;
121
122   // Round-up if necessary.
123   assign significand_plus_ulp = significand_tmp + 1'b1;
```

```
124     assign roundup = round_to_infinity | round_to_nearest_even;
125     assign significand =
126        roundup ?
127           significand_plus_ulp : significand_tmp;
128
129     // Post-normalize if result after rounding lies in [2,4).
130     assign postnormalize = !significand[SW] & significand_tmp[SW];
131     assign result_tmp =
132        postnormalize ?
133        {sign, exponent[EW-1:0] + 1'b1, significand[SW:1]} :
134        {sign, exponent[EW-1:0], significand[SW-1:0]};
135
136     // Inexact if result was rounded.
137     assign rounded = round | sticky;
138
139     assign ovf_postnorm =
140        exponent[EW] | &exponent[EW-1:0]&(normalize|postnormalize);
141
142
143     // Generate exceptions.
144     // ----------------------------------------------------------------
145
146     assign ovf_ab = exp[EW+1];
147     assign ovf_biased = exp[EW];
148
149     // Invalid inputs from chk_special.
150     assign nan_a = specials[0];
151     assign nan_b = specials[1];
152     assign inf_a = specials[2];
153     assign inf_b = specials[3];
154     assign zero_a = specials[4];
155     assign zero_b = specials[5];
156     assign int_a = specials[6];
157     assign int_b = specials[7];
158
159
160     // Generate exceptions.
161     assign int_times_inf = (int_a&inf_b)|(int_b&inf_a);
162
163     assign invalid =
164        (nan_a | nan_b)|
165        (zero_a&inf_b | zero_b&inf_a)|
166        (inf_a | inf_b)&!int_times_inf;
167
168     assign inexact =
169        (rounded & (!invalid)|
170        overflow_tmp|
171        round_to_zero&overflow_tmp|
172        underflow&(!(zero_a|zero_b)))&!int_times_inf;
173
174     assign underflow =
175        (~ovf_ab&ovf_biased)|
176        (~|result_tmp[SW+EW-1:SW])&!(ovf_ab&ovf_biased|ovf_postnorm) &
177        !overflow&!invalid|(zero_a|zero_b)&!(nan_a|nan_b|inf_a|inf_b);
178
179     // If overflow occurs and rounding mode equals round-to zero,
180     // result shall be rounded to largest representative number.
181     // e.x 0111101111111111.
182
183     assign overflow_tmp =
184        ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
185       &result_tmp[SW+EW-1:SW]&!underflow)&!invalid;
```

```
186
187     assign overflow = overflow_tmp&!round_to_zero | int_times_inf;
188
189
190     // Compute special results.
191     assign product_nan =
192         {1'b0, {EW{1'b1}}, {(SW-1){1'b0}}, 1'b1};
193
194     assign product_zero =
195         {result_tmp[SW+EW], {(SW+EW){1'b0}}};
196
197     assign product_overflow =
198         {result_tmp[SW+EW], {EW{1'b1}}, {(SW){1'b0}}};
199
200     assign product_large =
201         {result_tmp[SW+EW], {(EW-1){1'b1}}, 1'b0, {(SW){1'b1}}};
202
203     // Final product decided by exceptions.
204     assign result =
205         invalid ? product_nan :
206         overflow ? product_overflow :
207         underflow ? product_zero :
208         round_to_zero & overflow_tmp & !int_times_inf ? product_large :
209         result_tmp;
210
211     assign exceps[0] = invalid;
212     assign exceps[1] = inexact;
213     assign exceps[2] = overflow;
214     assign exceps[3] = underflow;
215
216 endmodule // rne
```

```
1 // ------------------------------------------------------------------------
2 // File.......: sel_input.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 10:54:28 CEST 2008
5 // Revision...: 1.0
6 // Description: Selects data from the input registers and puts it on
7 //              the exponent, significand and sign busses.
8 // ------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module sel_input
13 (
14    drh,     // Input from data-register high (DRH0).
15    drl,     // Input from data-register low (DRL0).
16    format,  // Input form instrucion(format) register.
17    signs,   // Output to sign bus.
18    exps,    // Output to exponent bus.
19    fracs    // Output to significand bus.
20 );
21
22    parameter WIDTH      = `BUS/2;
23    parameter SIGNBUS    = `SIGNBUS;
24    parameter EXPBUS     = `EXPBUS;
25    parameter FRACBUS    = `FRACBUS;
26
27    // input(s)
28    input [WIDTH-1:0]      drh;
29    input [WIDTH-1:0]      drl;
30    input [1:0]            format;
31
32    // output(s)
33    output [SIGNBUS-1:0]  signs;
34    output [EXPBUS-1:0]   exps;
35    output [FRACBUS-1:0]  fracs;
36
37    // wire(s)
38
39    // reg(s)
40    reg [SIGNBUS-1:0]      signs_tmp;
41    reg [EXPBUS-1:0]       exps_tmp;
42    reg [FRACBUS-1:0]      fracs_tmp;
43
44
45    // ---------------------------------------------------------------------
46    // Combinational logic.
47    // ---------------------------------------------------------------------
48
49    always @ (drh or drl or format) begin
50       signs_tmp = 0;
51       exps_tmp = 0;
52       fracs_tmp = 0;
53
54       case (format)
55          `FP16: begin
56             signs_tmp =
57                {drl[4*`FP16W-1], drl[3*`FP16W-1],
58                 drl[2*`FP16W-1], drl[1*`FP16W-1]};
59
60             exps_tmp =
61                {drl[4*(`FP16W)-2:3*`FP16W+`FP16SW],
62                 drl[3*(`FP16W)-2:2*`FP16W+`FP16SW],
```

```
63                      drl[2*(`FP16W)-2:1*`FP16W+`FP16SW],
64                      drl[1*(`FP16W)-2:0*`FP16W+`FP16SW]};
65
66                fracs_tmp =
67                    {1'b1, drl[4*`FP16W-`FP16EW-2:3*`FP16W],
68                     1'b1, drl[3*`FP16W-`FP16EW-2:2*`FP16W],
69                     1'b1, drl[2*`FP16W-`FP16EW-2:1*`FP16W],
70                     1'b1, drl[1*`FP16W-`FP16EW-2:0*`FP16W]};
71            end
72            `FP32: begin
73                signs_tmp =
74                    {drh[2*`FP32W-1], drh[1*`FP32W-1],
75                     drl[2*`FP32W-1], drl[1*`FP32W-1]};
76
77                exps_tmp =
78                    {drh[2*(`FP32W)-2:1*`FP32W+`FP32SW],
79                     drh[1*(`FP32W)-2:0*`FP32W+`FP32SW],
80                     drl[2*(`FP32W)-2:1*`FP32W+`FP32SW],
81                     drl[1*(`FP32W)-2:0*`FP32W+`FP32SW]};
82
83                fracs_tmp =
84                    {1'b1, drh[2*`FP32W-`FP32EW-2:1*`FP32W],
85                     1'b1, drh[1*`FP32W-`FP32EW-2:0*`FP32W],
86                     1'b1, drl[2*`FP32W-`FP32EW-2:1*`FP32W],
87                     1'b1, drl[1*`FP32W-`FP32EW-2:0*`FP32W]};
88            end
89            `FP64: begin
90                signs_tmp =
91                    {1'b0, 1'b0, drh[1*`FP64W-1], drl[1*`FP64W-1]};
92
93                exps_tmp =
94                    {drh[1*(`FP64W)-2:0*`FP64W+`FP64SW],
95                     drl[1*(`FP64W)-2:0*`FP64W+`FP64SW]};
96
97                fracs_tmp =
98                    {1'b1, drh[1*`FP64W-`FP64EW-2:0*`FP64W],
99                     1'b1, drl[1*`FP64W-`FP64EW-2:0*`FP64W]};
100           end
101       endcase
102   end
103
104
105   assign signs = signs_tmp;
106   assign exps = exps_tmp;
107   assign fracs = fracs_tmp;
108
109 endmodule // sel_input
```

```
1  // ---------------------------------------------------------------------
2  // File.......: sel_output.v
3  // Author.....: Espen Stenersen
4  // Date.......: Thu Apr 24 23:42:46 CEST 2008
5  // Revision...: 1.0
6  // Description: Loads the correct locations in output register and
7  //              exception register.
8  // ---------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module sel_output
13 (
14    result,         // Input from rounding logic.
15    exceps,         // Input from rounding logic.
16    format,         // Input from format register.
17    start,          // Input from start register.
18    products,       // Output to output register.
19    load_drh,       // Output to output register.
20    load_drlh,      // Output to output register.
21    load_drll,      // Output to output register.
22    exceptions,     // Output to exception register.
23    load_excep_l,   // Output to exception register.
24    load_excep_h,   // Output to exception register.
25    reset,
26    clk
27 );
28
29
30    // input(s)
31    input [`BUS/2-1:0]    result;
32    input [7:0]           exceps;
33    input [1:0]           format;
34    input                 start;
35    input                 clk;
36    input                 reset;
37
38    // output(s)
39    output [`BUS-1:0]     products;
40    output [15:0]         exceptions;
41    output                load_drh;
42    output                load_drlh;
43    output                load_drll;
44    output                load_excep_l;
45    output                load_excep_h;
46
47
48    // wire(s)
49
50    // reg(s)
51    reg [`BUS-1:0]        products;
52    reg [15:0]            exceptions;
53    reg                   load_drh;
54    reg                   load_drlh;
55    reg                   load_drll;
56    reg                   load_excep_l;
57    reg                   load_excep_h;
58    reg                   counter;
59
60
61    always @ (posedge clk) begin
62       if (reset) begin
```

```
63            counter <= 0;
64         end
65         else begin
66            if (start) begin
67               counter <= counter + 1;
68            end
69            else begin
70               counter <= 0;
71            end
72         end
73      end
74
75      always @ (result or exceps or format or counter or start) begin
76         products = 0;
77         exceptions = 0;
78         load_drh = 0;
79         load_drlh = 0;
80         load_drll = 0;
81         load_excep_l = 0;
82         load_excep_h = 0;
83
84         case (format)
85            `FP16: begin
86               case (counter)
87                  0: begin
88                     products[31:0] = result[31:0];
89                     exceptions[7:0] = exceps;
90                     load_drll = 1;
91                     load_excep_l = 1;
92                  end
93                  1: begin
94                     products[63:32] = result[31:0];
95                     exceptions[15:8] = exceps;
96                     load_drlh = 1;
97                     load_excep_h = 1;
98                  end
99               endcase
100           end
101           `FP32: begin
102              case (counter)
103                 0: begin
104                    products[63:0] = result[63:0];
105                    exceptions[7:0] = exceps;
106                    load_drll = 1;
107                    load_drlh = 1;
108                    load_excep_l = 1;
109                 end
110                 1: begin
111                    products[127:64] = result[63:0];
112                    exceptions[15:8] = exceps;
113                    load_drh = 1;
114                    load_excep_h = 1;
115                 end
116              endcase
117           end
118           `FP64: begin
119              case (counter)
120                 0: begin
121                    products[63:0] = result[63:0];
122                    exceptions[7:0] = exceps;
123                    load_drll = 1;
124                    load_drlh = 1;
```

```
125                     load_excep_l = 1;
126                 end
127                 1: begin
128                     products[127:64] = result[63:0];
129                     exceptions[15:8] = exceps;
130                     load_drh = 1;
131                     load_excep_h = 1;
132                 end
133             endcase
134         end
135     endcase
136 end
137
138 endmodule // sel_output
```

```
1 // ---------------------------------------------------------------------
2 // File.......: reg_enable.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 10:31:28 CEST 2008
5 // Revision...: 1.0
6 // Description: Generic register with synchronous reset and enable.
7 // ---------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module reg_enable
12 (
13     d,        // Data in.
14     q,        // Data out.
15     enable,   // Enable bit.
16     clk,
17     reset
18 );
19
20     parameter WIDTH = `BUS;
21
22     // input(s)
23     input [WIDTH-1:0]      d;
24     input                  enable;
25     input                  clk;
26     input                  reset;
27
28     // output(s)
29     output [WIDTH-1:0]     q;
30
31     // wire(s)
32
33     // reg(s)
34     reg [WIDTH-1:0]        q;
35
36     always @ (posedge clk) begin
37         if (reset) begin
38             q <= 0;
39         end
40         else if (enable) begin
41             q <= d;
42         end
43     end
44
45 endmodule // reg_enable
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: reg_set.v
3 // Author.....: Espen Stenersen
4 // Date.......: Thu Apr 24 23:07:57 CEST 2008
5 // Revision...: 1.0
6 // Description: Generic register with syncrhonous set.
7 // ---------------------------------------------------------------------------
8
9 module reg_set
10 (
11    set,       // Input.
12    q,         // Output.
13    clk,
14    reset
15 );
16
17    parameter WIDTH = 1;
18
19    // input(s)
20    input     set;
21    input     clk;
22    input     reset;
23
24    // output(s)
25    output    q;
26
27    // wire(s)
28
29    // reg(s)
30    reg       q;
31
32    always @ (posedge clk) begin
33       if (reset) begin
34          q <= 0;
35       end
36       else if (set) begin
37          q <= 1;
38       end
39       else begin
40          q <= 0;
41       end
42    end
43
44 endmodule // reg_set
```

```
1 // ─────────────────────────────────────────────────────────────
2 // File.......: ff.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:45:16 CEST 2008
5 // Revision...: 1.0
6 // Description: Generic register with synchronous reset.
7 // ─────────────────────────────────────────────────────────────
8
9 module ff
10 (
11     d,          // Data in.
12     q,          // Data out.
13     clk,
14     reset
15 );
16
17     parameter WIDTH = 1;
18
19     // input(s)
20     input [WIDTH-1:0]     d;
21     input                 clk;
22     input                 reset;
23
24     // output(s)
25     output [WIDTH-1:0]    q;
26
27     // reg(s)
28     reg [WIDTH-1:0]       q;
29
30     always @ (posedge clk) begin
31         if (reset)
32             q <= 0;
33         else
34             q <= d;
35     end
36
37 endmodule // ff
```

Appendix B

Architecture Two Verilog Sources

Only sources that differs between the two architectures are included in this Chapter, exponent unit building blocks, multiplier unit building blocks and rounding and exception unit building blocks.

```
1  // ---------------------------------------------------------------------------
2  // File.......:  defines.v
3  // Author.....:  Espen Stenersen
4  // Date.......:  Wed May 14 11:45:28 CEST 2008
5  // Revision...:  1.0
6  // Description:  Contains definitions used in the design files.
7  //               Openrand widths, exponent widths, significand widths,
8  //               bias values and bus widths.
9  // ---------------------------------------------------------------------------
10
11 `define FP16          0
12 `define FP32          1
13 `define FP64          2
14
15 `define FP16W         16
16 `define FP32W         32
17 `define FP64W         64
18
19 `define FP16SW        10
20 `define FP32SW        23
21 `define FP64SW        52
22
23 `define FP16EW        5
24 `define FP32EW        8
25 `define FP64EW        11
26
27 `define FP16BIAS      15
28 `define FP32BIAS      127
29 `define FP64BIAS      1023
30
31 `define FRACBUS       2*(`FP64SW+1)
32 `define FRACBUSOUT    154
33 `define EXPBUS        4*`FP32EW
34 `define EXPBUSOUT     20
35 `define SIGNBUS       4
36
```

```
37 `define BUS      128
38
39 `define EVEN     0
40 `define PINF     1
41 `define NINF     2
42 `define ZERO     3
```

```
1 // ------------------------------------------------------------------------
2 // File.......: exp_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:40:17 CEST 2008
5 // Revision...: 1.0
6 // Description: Exponent adder unit.
7 // ------------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module exp_unit
12 (
13     exps,     // Input from exponent bus.
14     format,   // Input from instruction register.
15     sums      // Output to exponent bus.
16 );
17
18
19     // input(s)
20     input  [`EXPBUS-1:0]      exps;
21     input  [1:0]              format;
22
23     // output(s)
24     output [`EXPBUSOUT-1:0]  sums;
25
26     // wire(s)
27     wire [`FP32EW-1:0]        fp32_a_0;
28     wire [`FP32EW-1:0]        fp32_b_0;
29     wire [`FP32EW-1:0]        fp32_sum_0;
30     wire                      fp32_ovf_ab_0;
31     wire                      fp32_ovf_biased_0;
32     wire [`FP64EW-1:0]        fp64_a_0;
33     wire [`FP64EW-1:0]        fp64_b_0;
34     wire [`FP64EW-1:0]        fp64_sum_0;
35     wire                      fp64_ovf_ab_0;
36     wire                      fp64_ovf_biased_0;
37
38     // reg(s)
39
40     // --------------------------------------------------------------------
41     // Module instantiation.
42     // --------------------------------------------------------------------
43
44     exp_add8 #(`FP32EW) fp32_add_0
45     (
46         .a            (fp32_a_0),
47         .b            (fp32_b_0),
48         .sum          (fp32_sum_0),
49         .format       (format),
50         .ovf_ab       (fp32_ovf_ab_0),
51         .ovf_biased   (fp32_ovf_biased_0)
52     );
53
54     exp_add11 #(`FP64EW) fp64_add_0
55     (
56         .a            (fp64_a_0),
57         .b            (fp64_b_0),
58         .format       (format),
59         .sum          (fp64_sum_0),
60         .ovf_ab       (fp64_ovf_ab_0),
61         .ovf_biased   (fp64_ovf_biased_0)
62     );
```

```
63
64    // ------------------------------------------------------------
65    // Combinational assign.
66    // ------------------------------------------------------------
67
68    // Input demux.
69
70    assign fp32_a_0 =
71        (format == `FP16) ?
72        {{(`FP32EW-`FP16EW){1'b0}}, exps[1*`FP16EW-1:0*`FP16EW]} :
73        (format == `FP32) ?
74        exps[1*`FP32EW-1:0*`FP32EW] : 0;
75    assign fp32_b_0 =
76        (format == `FP16) ?
77        {{(`FP32EW-`FP16EW){1'b0}}, exps[2*`FP16EW-1:1*`FP16EW]} :
78        (format == `FP32) ?
79        exps[2*`FP32EW-1:1*`FP32EW] : 0;
80
81    assign fp64_a_0 =
82        (format == `FP16) ?
83        {{(`FP64EW-`FP16EW){1'b0}}, exps[3*`FP16EW-1:2*`FP16EW]} :
84        (format == `FP32) ?
85        {{(`FP64EW-`FP32EW){1'b0}}, exps[3*`FP32EW-1:2*`FP32EW]} :
86        (format == `FP64) ?
87        exps[1*`FP64EW-1:0*`FP64EW] : 0;
88
89    assign fp64_b_0 =
90        (format == `FP16) ?
91        {{(`FP64EW-`FP16EW){1'b0}}, exps[4*`FP16EW-1:3*`FP16EW]} :
92        (format == `FP32) ?
93        {{(`FP64EW-`FP32EW){1'b0}}, exps[4*`FP32EW-1:3*`FP32EW]} :
94        (format == `FP64) ?
95        exps[2*`FP64EW-1:1*`FP64EW] : 0;
96
97    // Output mux.
98    assign sums =
99        (format == `FP16) ?
100       {fp64_ovf_ab_0, fp64_ovf_biased_0, fp64_sum_0[`FP16EW-1:0],
101       fp32_ovf_ab_0, fp32_ovf_biased_0, fp32_sum_0[`FP16EW-1:0]} :
102       (format == `FP32) ?
103       {fp64_ovf_ab_0, fp64_ovf_biased_0, fp64_sum_0[`FP32EW-1:0],
104       fp32_ovf_ab_0, fp32_ovf_biased_0, fp32_sum_0[`FP32EW-1:0]} :
105       (format == `FP64) ?
106       {fp64_ovf_ab_0, fp64_ovf_biased_0, fp64_sum_0[`FP64EW-1:0]} :
107       0;
108
109 endmodule // exp_unit
```

```
1 // ---------------------------------------------------------------------------
2 // File.......:  exp_add8.v
3 // Author.....:  Espen Stenersen
4 // Date.......:  Tue Apr 15 10:44:27 CEST 2008
5 // Revision...:  1.0
6 // Description:  Exponent adder. Adds the two inputs, and subtracts
7 //               the bias.
8 // ---------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module exp_add8
13 (
14    a,            // Input operand.
15    b,            // Input operand.
16    format,       // Input.
17    sum,          // Output sum.
18    ovf_ab,       // Overflow after addition.
19    ovf_biased    // Overflow after subtraction.
20 );
21
22    parameter WIDTH = `FP32EW;
23
24    // input(s)
25    input [WIDTH-1:0]      a;
26    input [WIDTH-1:0]      b;
27    input [1:0]            format;
28
29    // output(s)
30    output [WIDTH-1:0]     sum;
31    output                 ovf_ab;
32    output                 ovf_biased;
33
34    // wire(s)
35    wire   [WIDTH:0]       a_plus_b_tmp;
36    wire   [WIDTH:0]       biased_tmp;
37
38    // Exponent1 + exponent2
39    assign a_plus_b_tmp = a + b;
40
41    // Subtract bias.
42    assign biased_tmp =
43       (format == `FP16) ? a_plus_b_tmp - `FP16BIAS :
44       (format == `FP32) ? a_plus_b_tmp - `FP32BIAS : 0;
45    // Selcet part of sum.
46    assign sum =
47       (format == `FP16) ? biased_tmp[`FP16EW-1:0] :
48       (format == `FP32) ? biased_tmp[`FP32EW-1:0] : 0 ;
49
50    // Compute overflow / underflow detection bits.
51    assign ovf_ab =
52       (format == `FP16) ? a_plus_b_tmp[`FP16EW] :
53       (format == `FP32) ? a_plus_b_tmp[`FP32EW] : 0;
54
55    // Compute overflow / underflow detection bits.
56    assign ovf_biased =
57       (format == `FP16) ? biased_tmp[`FP16EW] :
58       (format == `FP32) ? biased_tmp[`FP32EW] :  0;
59
60
61 endmodule // exp_add8
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: exp_add11.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 10:44:27 CEST 2008
5 // Revision...: 1.0
6 // Description: Exponent adder. Adds the two inputs, and subtracts
7 //              the bias.
8 // ---------------------------------------------------------------------------
9
10 `include "defines.v"
11
12 module exp_add11
13 (
14    a,          // Input operand.
15    b,          // Input operand.
16    format,     // Input.
17    sum,        // Output sum.
18    ovf_ab,     // Overflow after addition.
19    ovf_biased  // Overflow after subtraction.
20 );
21
22    parameter WIDTH = `FP64EW;
23
24    // input(s)
25    input  [WIDTH-1:0]    a;
26    input  [WIDTH-1:0]    b;
27    input  [1:0]          format;
28
29    // output(s)
30    output  [WIDTH-1:0]   sum;
31    output                ovf_ab;
32    output                ovf_biased;
33
34    // wire(s)
35    wire   [WIDTH:0]      a_plus_b_tmp;
36    wire   [WIDTH:0]      biased_tmp;
37
38    // Exponent1 + exponent2
39    assign a_plus_b_tmp = a + b;
40
41    // Subtract bias.
42    assign biased_tmp =
43       (format == `FP16) ? a_plus_b_tmp - `FP16BIAS :
44       (format == `FP32) ? a_plus_b_tmp - `FP32BIAS :
45       (format == `FP64) ? a_plus_b_tmp - `FP64BIAS : 0;
46
47    // Selcet part of sum.
48    assign sum =
49       (format == `FP16) ? biased_tmp[`FP16EW-1:0] :
50       (format == `FP32) ? biased_tmp[`FP32EW-1:0] :
51       (format == `FP64) ? biased_tmp[`FP64EW-1:0] : 0 ;
52
53    // Compute overflow / underflow detection bits.
54    assign ovf_ab =
55       (format == `FP16) ? a_plus_b_tmp[`FP16EW] :
56       (format == `FP32) ? a_plus_b_tmp[`FP32EW] :
57       (format == `FP64) ? a_plus_b_tmp[`FP64EW] : 0;
58
59    // Compute overflow / underflow detection bits.
60    assign ovf_biased =
61       (format == `FP16) ? biased_tmp[`FP16EW] :
62       (format == `FP32) ? biased_tmp[`FP32EW] :
```

```
63          (format == `FP64) ? biased_tmp[`FP64EW] : 0;
64
65
66 endmodule // exp_add11
```

```
1 // ------------------------------------------------------------------------
2 // File.......: mult_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:37:56 CEST 2008
5 // Revision...: 1.0
6 // Description: Significand multiplier unit.
7 // ------------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module mult_unit
12 (
13     fracs,    // Input from significand bus.
14     format,   // Input from instruction register.
15     prods     // Output to significand bus.
16 );
17
18     // input(s)
19     input  [`FRACBUS-1:0]        fracs;
20     input  [1:0]                 format;
21
22     // output(s)
23     output [`FRACBUSOUT-1:0]     prods;
24
25     // wire(s)
26     wire  [`FP32SW:0]            fp32_a_0;
27     wire  [`FP32SW:0]            fp32_b_0;
28     wire  [2*`FP32SW+1:0]        fp32_p_0;
29     wire  [`FP64SW:0]            fp64_a_0;
30     wire  [`FP64SW:0]            fp64_b_0;
31     wire  [2*`FP64SW+1:0]        fp64_p_0;
32
33     // reg(s)
34
35     // --------------------------------------------------------------------
36     // Module instantiations.
37     // --------------------------------------------------------------------
38
39     uns_mult #(`FP32SW+1) uns_mult_fp32_0
40     (
41         .a(fp32_a_0),
42         .b(fp32_b_0),
43         .p(fp32_p_0)
44     );
45
46     uns_mult #(`FP64SW+1) uns_mult_fp64_0
47     (
48         .a(fp64_a_0),
49         .b(fp64_b_0),
50         .p(fp64_p_0)
51     );
52
53     // --------------------------------------------------------------------
54     // Combinaional assigns.
55     // --------------------------------------------------------------------
56
57     // Input demux.
58
59     assign fp32_a_0 =
60         (format == `FP16) ?
61             {fracs[1*(`FP16SW+1)-1:0*(`FP16SW+1)],
62             {(`FP32SW-`FP16SW){1'b0}}} :
```

```
63         (format == `FP32) ?
64             fracs[1*(`FP32SW+1)-1:0*(`FP32SW+1)] : 0;
65
66     assign fp32_b_0 =
67         (format == `FP16) ?
68             {fracs[2*(`FP16SW+1)-1:1*(`FP16SW+1)],
69             {(`FP32SW-`FP16SW){1'b0}}} :
70         (format == `FP32) ?
71             fracs[2*(`FP32SW+1)-1:1*(`FP32SW+1)] : 0;
72
73
74     assign fp64_a_0 =
75         (format == `FP16) ?
76             {fracs[3*(`FP16SW+1)-1:2*(`FP16SW+1)],
77             {(`FP64SW-`FP16SW){1'b0}}} :
78         (format == `FP32) ?
79             {fracs[3*(`FP32SW+1)-1:2*(`FP32SW+1)],
80             {(`FP64SW-`FP32SW){1'b0}}} :
81         (format == `FP64) ?
82             fracs[1*(`FP64SW+1)-1:0*(`FP64SW+1)] : 0;
83
84     assign fp64_b_0 =
85         (format == `FP16) ?
86             {fracs[4*(`FP16SW+1)-1:3*(`FP16SW+1)],
87             {(`FP64SW-`FP16SW){1'b0}}} :
88         (format == `FP32) ?
89             {fracs[4*(`FP32SW+1)-1:3*(`FP32SW+1)],
90             {(`FP64SW-`FP32SW){1'b0}}} :
91         (format == `FP64) ?
92             fracs[2*(`FP64SW+1)-1:1*(`FP64SW+1)] : 0;
93
94     assign prods = {fp64_p_0, fp32_p_0};
95
96 endmodule // mult_unit
```

```
 1 // ------------------------------------------------------------------------
 2 // File.......: uns_mult.v
 3 // Author.....: Espen Stenersen
 4 // Date.......: Tue Apr 15 10:40:36 CEST 2008
 5 // Revision...: 1.0
 6 // Description: Unsigned multiplier used for significand
 7 //              multiplication.
 8 // ------------------------------------------------------------------------
 9
10 `include "defines.v"
11
12 module uns_mult
13 (
14    a, // Input, multiplicand.
15    b, // Input, multiplier.
16    p  // Output, product.
17 );
18
19
20    parameter WIDTH = `FP64SW+1;
21
22    // input(s)
23    input [WIDTH-1:0]     a;
24    input [WIDTH-1:0]     b;
25
26    // output(s)
27    output [2*WIDTH-1:0] p;
28
29    assign p = a * b;
30
31 endmodule // uns_mult
```

```
1 // ----------------------------------------------------------------------
2 // File.......: rne_unit.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 12:19:50 CEST 2008
5 // Revision...: 1.0
6 // Description: Rounding, normalizing and exception unit.
7 // ----------------------------------------------------------------------
8
9 `include "defines.v"
10
11 module rne_unit
12 (
13    fracs,    // Input from fraction bus.
14    exps,     // Input form exponent bus.
15    signs,    // Input from sign bus.
16    format,   // Input from instrucion register.
17    special,  // Input form check special.
18    mode,     // Input from mode register.
19    exceps,   // Output exceptions.
20    result    // Output. Rounded result.
21 );
22
23    // input(s)
24    input [`FRACBUSOUT-1:0]      fracs;
25    input [`EXPBUSOUT-1:0]       exps;
26    input [`SIGNBUS/2-1:0]       signs;
27    input [1:0]                  format;
28    input [1:0]                  mode;
29    input [15:0]                 special;
30
31    // output(s)
32    output [`BUS/2-1:0]          result;
33    output [7:0]                 exceps;
34
35    // wire(s)
36    wire                         sign_rne_0;
37    wire                         sign_rne_1;
38    wire [2*`FP32SW+1:0]         frac_rne_0;
39    wire [2*`FP64SW+1:0]         frac_rne_1;
40    wire [`FP32EW+1:0]           exp_rne_0;
41    wire [`FP64EW+1:0]           exp_rne_1;
42    wire [`FP32W-1:0]            result_rne_0;
43    wire [`FP64W-1:0]            result_rne_1;
44    wire [7:0]                   specials_rne_0;
45    wire [7:0]                   specials_rne_1;
46    wire [3:0]                   exceps_rne_0;
47    wire [3:0]                   exceps_rne_1;
48    wire [1:0]                   overflow;
49    wire [1:0]                   underflow;
50    wire [1:0]                   inexact;
51    wire [1:0]                   invalid;
52    // reg(s)
53
54    // ------------------------------------------------------------------
55    // Module instantiation.
56    // ------------------------------------------------------------------
57
58    rne32 #(`FP32SW, `FP32EW) rne_0
59    (
60       .frac        (frac_rne_0),
61       .sign        (sign_rne_0),
62       .exp         (exp_rne_0),
```

```
63         .specials    (specials_rne_0),
64         .mode        (mode),
65         .format      (format),
66         .result      (result_rne_0),
67         .exceps      (exceps_rne_0)
68     );
69
70     rne64 #(`FP64SW, `FP64EW) rne_1
71     (
72         .frac        (frac_rne_1),
73         .sign        (sign_rne_1),
74         .exp         (exp_rne_1),
75         .specials    (specials_rne_1),
76         .mode        (mode),
77         .format      (format),
78         .result      (result_rne_1),
79         .exceps      (exceps_rne_1)
80     );
81
82
83
84     // ----------------------------------------------------------------
85     // Combinalional assign.
86     // ----------------------------------------------------------------
87
88     // Inputs to rounding logic.
89     assign frac_rne_0 = fracs[2*(`FP32SW+1)-1:0];
90     assign frac_rne_1 = fracs[`FRACBUSOUT-1:2*(`FP32SW+1)];
91
92     // Two msb bits represents the overflow bits during exponent
93     // addition.
94     assign exp_rne_0 =
95         (format == `FP16) ?
96             exps[`FP16EW+1:0] :
97         (format == `FP32) ?
98             exps[`FP32EW+1:0] :
99         0;
100
101    assign exp_rne_1 =
102        (format == `FP16) ?
103            exps[`EXPBUSOUT-1:`FP16EW+2] :
104        (format == `FP32) ?
105            exps[`EXPBUSOUT-1:`FP32EW+2] :
106        (format == `FP64) ?
107            exps[`FP64EW+1:0] :
108        0;
109
110    assign sign_rne_0 = signs[0];
111
112    assign sign_rne_1 =
113        (format == `FP64) ?
114            signs[0] :
115        signs[1];
116
117    assign specials_rne_0 =
118        {special[13:12], special[9:8], special[5:4], special[1:0]};
119
120    assign specials_rne_1 =
121        (format == `FP64) ?
122            {special[13:12], special[9:8], special[5:4], special[1:0]} :
123        {special[15:14], special[11:10], special[7:6], special[3:2]};
124
```

```
125
126     assign underflow =
127         (format == `FP64) ?
128             {1'b0, exceps_rne_1[3]} :
129         {exceps_rne_1[3], exceps_rne_0[3]};
130
131     assign overflow =
132         (format == `FP64) ?
133             {1'b0, exceps_rne_1[2]} :
134         {exceps_rne_1[2], exceps_rne_0[2]};
135
136     assign inexact =
137         (format == `FP64) ?
138             {1'b0, exceps_rne_1[1]} :
139         {exceps_rne_1[1], exceps_rne_0[1]};
140
141     assign invalid =
142         (format == `FP64) ?
143             {1'b0, exceps_rne_1[0]} :
144         {exceps_rne_1[0], exceps_rne_0[0]};
145
146     assign exceps = {underflow, overflow, inexact, invalid};
147
148     assign result =
149         (format == `FP16) ?
150             {result_rne_1[`FP16SW + `FP16EW:0],
151             result_rne_0[`FP16SW + `FP16EW:0]} :
152         (format == `FP32) ?
153             {result_rne_1[`FP32SW + `FP32EW:0],
154             result_rne_0[`FP32SW + `FP32EW:0]} :
155         (format == `FP64) ?
156             result_rne_1[`FP64SW + `FP64EW:0] :
157         0;
158
159 endmodule // rne_unit
```

```
1 // ----------------------------------------------------------------------
2 // File.......: rne32.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:10:54 CEST 2008
5 // Revision...: 1.0
6 // Description: Rounding and exception unit. Rounds, normalizes and
7 //              postnormlizes the result from the computation, and
8 //              generates exceptions if needed.
9 // ----------------------------------------------------------------------
10
11 `include "defines.v"
12
13 module rne32
14 (
15     frac,        // Input. Fractional part from multiplication.
16     sign,        // Input. Sign from sign computation.
17     exp,         // Input. Biased exponent from exponent addition.
18     specials,    // Input. NaNs, infinities, zeros..
19     format,      // Input.
20     mode,        // Input. Rounding mode.
21     result,      // Output. Rounded result or special value.
22     exceps       // Output. Exceptions.
23 );
24
25     parameter SW = `FP32SW;
26     parameter EW = `FP32EW;
27
28     // input(s)
29     input [2*SW+1:0]    frac;
30     input [EW+1:0]      exp;
31     input               sign;
32     input [7:0]         specials;
33     input [1:0]         mode;
34     input [1:0]         format;
35
36     // output(s)
37     output [SW+EW:0]    result;
38     output [3:0]        exceps;
39
40     // wire(s)
41     wire                normalize;
42     wire                postnormalize;
43     wire                lsb;
44     wire                round;
45     wire                sticky;
46     wire                roundup;
47     wire                rounded;
48     wire                ovf_ab;
49     wire                ovf_biased;
50     wire                ovf_postnorm;
51     wire                round_to_nearest_even;
52     wire                round_to_infinity;
53     wire                round_to_zero;
54     wire                nan_a;
55     wire                nan_b;
56     wire                int_a;
57     wire                int_b;
58     wire                inf_a;
59     wire                inf_b;
60     wire                zero_a;
61     wire                zero_b;
62     wire                int_times_inf;
```

```
63      wire                invalid;
64      wire                overflow;
65      wire                overflow_tmp;
66      wire                underflow;
67      wire                inexact;
68      wire  [SW:0]        significand;
69      wire  [SW:0]        significand_tmp;
70      wire  [SW:0]        significand_plus_ulp;
71      wire  [EW:0]        exponent;
72      wire  [EW:0]        exponent_tmp;
73      wire  [SW+EW:0]     result_tmp;
74      wire  [SW+EW:0]     product_nan;
75      wire  [SW+EW:0]     product_zero;
76      wire  [SW+EW:0]     product_large;
77      wire  [SW+EW:0]     product_overflow;
78
79      // reg(s)
80
81
82      // Round and normalize / Postnormalize.
83      // -----------------------------------------------------------------------
84
85      // Normalize if result from multiplier lies in [2,4)
86      assign normalize = frac[2*SW+1];
87
88      assign significand_tmp =
89          normalize ?
90          frac[2*SW:SW] >> 1 :
91          frac[2*SW:SW];
92
93      assign exponent_tmp =
94          (format == `FP16) ?
95          normalize ?
96          exp[`FP16EW-1:0] + 1 : exp[`FP16EW-1:0] :
97          (format == `FP32) ?
98          normalize ?
99          exp[`FP32EW-1:0] + 1 : exp[`FP32EW-1:0] : 0;
100
101
102     // Assign rounding bits.
103     assign lsb =
104         (format == `FP16) ?
105         normalize ?
106         frac[37] :
107         frac[36] :
108         (format == `FP32) ?
109         normalize ?
110         frac[24] :
111         frac[23] : 0;
112
113
114     assign round =
115         (format == `FP16) ?
116         normalize ?
117         frac[36] :
118         frac[35] :
119         (format == `FP32) ?
120         normalize ?
121         frac[23] :
122         frac[22] : 0;
123
124     assign sticky =
```

```
125        (format == `FP16) ?
126        normalize ?
127        |frac[35:26] :
128        |frac[34:25] :
129        (format == `FP32) ?
130        normalize ?
131        |frac[22:1] :
132        |frac[21:0] : 0;
133
134    // Reduce to three rounding modes.
135    assign round_to_nearest_even =
136        (round & (lsb | sticky)) & !(|mode);
137
138    assign round_to_infinity =
139        (!sign&(!mode[1]&mode[0])|sign&(mode[1]&!mode[0])) &
140        (round|sticky);
141
142    assign round_to_zero =
143        (sign&(~mode[1]&mode[0])|~sign&(mode[1]&~mode[0]))|&mode;
144
145    // Round-up if necessary.
146    assign significand_plus_ulp =
147        (format == `FP16) ?
148        significand_tmp[SW:SW-`FP16SW] + 1'b1 :
149        (format == `FP32) ?
150        significand_tmp[SW:SW-`FP32SW] + 1'b1 : 0;
151
152    assign roundup = round_to_infinity | round_to_nearest_even;
153    assign significand =
154        (format == `FP16) ?
155        roundup ?
156        significand_plus_ulp : significand_tmp[SW:SW-`FP16SW] :
157        (format == `FP32) ?
158        roundup ?
159        significand_plus_ulp : significand_tmp[SW:SW-`FP32SW] : 0;
160
161    // Post-normalize if result after rounding lies in [2,4).
162    assign postnormalize =
163        (format == `FP16) ?
164        !significand[`FP16SW]&significand_tmp[SW] :
165        (format == `FP32) ?
166        !significand[`FP32SW]&significand_tmp[SW] : 0;
167
168    assign exponent =
169        postnormalize ?
170        exponent_tmp + 1 :
171        exponent_tmp;
172
173    assign result_tmp =
174        (format == `FP16) ?
175        postnormalize ?
176        {sign, exponent[`FP16EW-1:0], significand[`FP16SW-1:0]} :
177        {sign, exponent[`FP16EW-1:0], significand[`FP16SW-1:0]} :
178        (format == `FP32) ?
179        postnormalize ?
180        {sign, exponent[`FP32EW-1:0], significand[`FP32SW-1:0]} :
181        {sign, exponent[`FP32EW-1:0], significand[`FP32SW-1:0]} : 0;
182
183
184    // Inexact if result was rounded.
185    assign rounded = round | sticky;
186
```

```
187     assign ovf_postnorm =
188         (format == `FP16) ?
189             exponent[`FP16EW] |
190             &exponent[`FP16EW-1:0]&(normalize|postnormalize) :
191         (format == `FP32) ?
192             exponent[`FP32EW] |
193             &exponent[`FP32EW-1:0]&(normalize|postnormalize) :
194         0;
195
196
197     // Generate exceptions.
198     // -----------------------------------------------------------------------------
199
200     assign ovf_ab =
201         (format == `FP16) ?
202         exp[`FP16EW+1] :
203         (format == `FP32) ?
204         exp[`FP32EW+1] : 0;
205
206     assign ovf_biased =
207         (format == `FP16) ?
208         exp[`FP16EW] :
209         (format == `FP32) ?
210         exp[`FP32EW] : 0;
211
212     // Invalid inputs from chk_special.
213     assign nan_a = specials[0];
214     assign nan_b = specials[1];
215     assign inf_a = specials[2];
216     assign inf_b = specials[3];
217     assign zero_a = specials[4];
218     assign zero_b = specials[5];
219     assign int_a = specials[6];
220     assign int_b = specials[7];
221
222
223     // Generate exceptions.
224     assign int_times_inf = (int_a&inf_b)|(int_b&inf_a);
225
226     assign invalid =
227         (nan_a | nan_b)|
228         (zero_a&inf_b | zero_b&inf_a)|
229         (inf_a | inf_b)&!int_times_inf;
230
231     assign inexact =
232         (rounded & (!invalid)|
233         overflow_tmp|
234         round_to_zero&overflow_tmp|
235         underflow&(!(zero_a|zero_b)))&!int_times_inf;
236
237     assign underflow =
238         (format == `FP16) ?
239             (~ovf_ab&ovf_biased)|
240             (~|result_tmp[`FP16SW+`FP16EW-1:`FP16SW]) &
241             !(ovf_ab&ovf_biased|ovf_postnorm) &
242             !overflow&!invalid|(zero_a | zero_b) &
243             !(nan_a|nan_b|inf_a|inf_b) :
244         (format == `FP32) ?
245             (~ovf_ab&ovf_biased)|
246             (~|result_tmp[`FP32SW+`FP32EW-1:`FP32SW]) &
247             !(ovf_ab&ovf_biased|ovf_postnorm) &
248             !overflow&!invalid|(zero_a | zero_b) &
```

```
249         !(nan_a|nan_b|inf_a|inf_b) : 0;
250
251     // If overflow occurs and rounding mode equals round-to zero,
252     // result shall be rounded to largest representative number.
253     // e.x 0111101111111111.
254     assign overflow_tmp =
255        (format == `FP16) ?
256           ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
257          &result_tmp[`FP16SW+`FP16EW-1:`FP16SW]&!underflow) &
258           !invalid :
259        (format == `FP32) ?
260           ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
261          &result_tmp[`FP32SW+`FP32EW-1:`FP32SW]&!underflow) &
262           !invalid : 0;
263
264     assign overflow = overflow_tmp&!round_to_zero | int_times_inf;
265
266
267     // Compute special results.
268     assign product_nan =
269        (format == `FP16) ?
270           {1'b0, {`FP16EW{1'b1}}, {(`FP16SW-1){1'b0}}, 1'b1} :
271        (format == `FP32) ?
272           {1'b0, {`FP32EW{1'b1}}, {(`FP32SW-1){1'b0}}, 1'b1} :
273           0;
274
275     assign product_zero =
276        (format == `FP16) ?
277           {result_tmp[`FP16SW+`FP16EW], {(`FP16SW+`FP16EW){1'b0}}} :
278        (format == `FP32) ?
279           {result_tmp[`FP32SW+`FP32EW], {(`FP32SW+`FP32EW){1'b0}}} :
280           0;
281
282     assign product_overflow =
283        (format == `FP16) ?
284           {result_tmp[`FP16SW+`FP16EW],
285           {`FP16EW{1'b1}}, {(`FP16SW){1'b0}}} :
286        (format == `FP32) ?
287           {result_tmp[`FP32SW+`FP32EW],
288           {`FP32EW{1'b1}}, {(`FP32SW){1'b0}}} :
289           0;
290
291     assign product_large =
292        (format == `FP16) ?
293           {result_tmp[`FP16SW+`FP16EW],
294           {(`FP16EW-1){1'b1}}, 1'b0, {(`FP16SW){1'b1}}} :
295        (format == `FP32) ?
296           {result_tmp[`FP32SW+`FP32EW],
297           {(`FP32EW-1){1'b1}}, 1'b0, {(`FP32SW){1'b1}}} :
298           0;
299
300     // Final product decided by exceptions.
301     assign result =
302        invalid ? product_nan :
303        overflow ? product_overflow :
304        underflow ? product_zero :
305        round_to_zero & overflow_tmp & !int_times_inf ? product_large :
306        result_tmp;
307
308     assign exceps[0] = invalid;
309     assign exceps[1] = inexact;
310     assign exceps[2] = overflow;
```

```
311     assign exceps[3] = underflow;
312
313 endmodule // rne32
```

```
1 // ---------------------------------------------------------------------------
2 // File.......: rne64.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 15 11:10:54 CEST 2008
5 // Revision...: 1.0
6 // Description: Rounding and exception unit. Rounds, normalizes and
7 //              postnormlizes the result from the computation, and
8 //              generates exceptions if needed.
9 // ---------------------------------------------------------------------------
10
11 `include "defines.v"
12
13 module rne64
14 (
15     frac,       // Input. Fractional part from multiplication.
16     sign,       // Input. Sign from sign computation.
17     exp,        // Input. Biased exponent from exponent addition.
18     specials,   // Input. NaNs, infinities, zeros..
19     format,     // Input.
20     mode,       // Input. Rounding mode.
21     result,     // Output. Rounded result or special value.
22     exceps      // Output. Exceptions.
23 );
24
25     parameter SW = 52;
26     parameter EW = 11;
27
28     // input(s)
29     input [2*SW+1:0]    frac;
30     input [EW+1:0]      exp;
31     input               sign;
32     input [7:0]         specials;
33     input [1:0]         mode;
34     input [1:0]         format;
35
36     // output(s)
37     output [SW+EW:0]    result;
38     output [3:0]        exceps;
39
40     // wire(s)
41     wire                normalize;
42     wire                postnormalize;
43     wire                lsb;
44     wire                round;
45     wire                sticky;
46     wire                roundup;
47     wire                rounded;
48     wire                ovf_ab;
49     wire                ovf_biased;
50     wire                ovf_postnorm;
51     wire                round_to_nearest_even;
52     wire                round_to_infinity;
53     wire                round_to_zero;
54     wire                nan_a;
55     wire                nan_b;
56     wire                int_a;
57     wire                int_b;
58     wire                inf_a;
59     wire                inf_b;
60     wire                zero_a;
61     wire                zero_b;
62     wire                int_times_inf;
```

```
63      wire                invalid;
64      wire                overflow;
65      wire                overflow_tmp;
66      wire                underflow;
67      wire                underflow_tmp;
68      wire                inexact;
69      wire                exp_zero;
70      wire  [SW:0]        significand;
71      wire  [SW:0]        significand_tmp;
72      wire  [SW:0]        significand_plus_ulp;
73      wire  [EW:0]        exponent;
74      wire  [EW:0]        exponent_tmp;
75      wire  [SW+EW:0]     result_tmp;
76      wire  [SW+EW:0]     product_nan;
77      wire  [SW+EW:0]     product_zero;
78      wire  [SW+EW:0]     product_large;
79      wire  [SW+EW:0]     product_overflow;
80      wire  [SW+EW:0]     product_min;
81
82      // reg(s)
83
84
85      // Round and normalize / Postnormalize.
86      // ------------------------------------------------------------------------------
87
88      // Normalize if result from multiplier lies in [2,4)
89      assign normalize = frac[2*SW+1];
90
91      assign significand_tmp =
92          normalize ?
93          frac[2*SW:SW] >> 1: frac[2*SW:SW];
94
95      assign exponent_tmp =
96          (format == `FP16) ?
97          normalize ?
98          exp[`FP16EW-1:0] + 1 : exp[`FP16EW-1:0] :
99          (format == `FP32) ?
100         normalize ?
101         exp[`FP32EW-1:0] + 1 : exp[`FP32EW-1:0] :
102         (format == `FP64) ?
103         normalize ?
104         exp[`FP64EW-1:0] + 1 : exp[`FP64EW-1:0] : 0;
105
106
107     // Assign rounding bits.
108     assign lsb =
109         (format == `FP16) ?
110         normalize ?
111         frac[95] :
112         frac[94] :
113         (format == `FP32) ?
114         normalize ?
115         frac[82] :
116         frac[81] :
117         (format == `FP64) ?
118         normalize ?
119         frac[53] :
120         frac[52] : 0;
121
122
123     assign round =
124         (format == `FP16) ?
```

```
125         normalize ?
126         frac[94] :
127         frac[93] :
128         (format == `FP32) ?
129         normalize ?
130         frac[81] :
131         frac[80] :
132         (format == `FP64) ?
133         normalize ?
134         frac[52] :
135         frac[51] : 0;
136
137     assign sticky =
138         (format == `FP16) ?
139         normalize ?
140         |frac[93:84] :
141         |frac[92:83] :
142         (format == `FP32) ?
143         normalize ?
144         |frac[80:59] :
145         |frac[79:58] :
146         (format == `FP64) ?
147         normalize ?
148         |frac[51:1] :
149         |frac[50:0] : 0;
150
151     // Reduce to three rounding modes.
152     assign round_to_nearest_even =
153         (round & (lsb | sticky)) & !(|mode);
154
155     assign round_to_infinity =
156         (!sign&(!mode[1]&mode[0])|sign&(mode[1]&!mode[0])) &
157         (round|sticky);
158
159     assign round_to_zero =
160         (sign&(~mode[1]&mode[0])|~sign&(mode[1]&~mode[0]))|&mode;
161
162     // Round-up if necessary.
163     assign significand_plus_ulp =
164         (format == `FP16) ?
165         significand_tmp[SW:SW-`FP16SW] + 1'b1 :
166         (format == `FP32) ?
167         significand_tmp[SW:SW-`FP32SW] + 1'b1 :
168         (format == `FP64) ?
169         significand_tmp[SW:SW-`FP64SW] + 1'b1 : 0;
170
171     assign roundup = round_to_infinity | round_to_nearest_even;
172     assign significand =
173         (format == `FP16) ?
174         roundup ?
175         significand_plus_ulp : significand_tmp[SW:SW-`FP16SW] :
176         (format == `FP32) ?
177         roundup ?
178         significand_plus_ulp : significand_tmp[SW:SW-`FP32SW] :
179         (format == `FP64) ?
180         roundup ?
181         significand_plus_ulp : significand_tmp[SW:SW-`FP64SW] : 0;
182
183     // Post-normalize if result after rounding lies in [2,4).
184     assign postnormalize =
185         (format == `FP16) ?
186         !significand[`FP16SW]&significand_tmp[SW] :
```

```
187         (format == `FP32) ?
188         !significand[`FP32SW]&significand_tmp[SW] :
189         (format == `FP64) ?
190         !significand[`FP64SW]&significand_tmp[SW] : 0;
191
192     assign exponent =
193         postnormalize ?
194         exponent_tmp + 1 :
195         exponent_tmp;
196
197
198     assign result_tmp =
199         (format == `FP16) ?
200         postnormalize ?
201             {sign, exponent[`FP16EW-1:0], significand[`FP16SW-1:0]} :
202             {sign, exponent[`FP16EW-1:0], significand[`FP16SW-1:0]} :
203         (format == `FP32) ?
204         postnormalize ?
205             {sign, exponent[`FP32EW-1:0], significand[`FP32SW-1:0]} :
206             {sign, exponent[`FP32EW-1:0], significand[`FP32SW-1:0]} :
207         (format == `FP64) ?
208         postnormalize ?
209             {sign, exponent[`FP64EW-1:0], significand[`FP64SW-1:0]} :
210             {sign, exponent[`FP64EW-1:0], significand[`FP64SW-1:0]} :
211         0;
212
213     // Inexact if result was rounded.
214     assign rounded = round | sticky;
215
216     assign ovf_postnorm =
217         (format == `FP16) ?
218             exponent[`FP16EW] |
219             &exponent[`FP16EW-1:0]&(normalize|postnormalize) :
220         (format == `FP32) ?
221             exponent[`FP32EW] |
222             &exponent[`FP32EW-1:0]&(normalize|postnormalize) :
223         (format == `FP64) ?
224             exponent[`FP64EW] |
225             &exponent[`FP64EW-1:0]&(normalize|postnormalize) :
226         0;
227
228
229     // Generate exceptions.
230     //
        ------------------------------------------------------------------------------------------
231
232     assign ovf_ab =
233         (format == `FP16) ?
234             exp[`FP16EW+1] :
235         (format == `FP32) ?
236             exp[`FP32EW+1] :
237         (format == `FP64) ?
238             exp[`FP64EW+1] : 0;
239
240     assign ovf_biased =
241         (format == `FP16) ?
242             exp[`FP16EW] :
243         (format == `FP32) ?
244             exp[`FP32EW] :
245         (format == `FP64) ?
246             exp[`FP64EW] : 0;
```

```
247
248    // Invalid inputs from chk_special.
249    assign nan_a = specials[0];
250    assign nan_b = specials[1];
251    assign inf_a = specials[2];
252    assign inf_b = specials[3];
253    assign zero_a = specials[4];
254    assign zero_b = specials[5];
255    assign int_a = specials[6];
256    assign int_b = specials[7];
257
258
259    // Generate exceptions.
260    assign int_times_inf = (int_a&inf_b)|(int_b&inf_a);
261
262    assign invalid =
263       (nan_a | nan_b)|
264       (zero_a&inf_b | zero_b&inf_a)|
265       (inf_a | inf_b)&!int_times_inf;
266
267    assign inexact =
268       (rounded & (!invalid)|
269       overflow_tmp|
270       round_to_zero&overflow_tmp|
271       underflow&(!(zero_a|zero_b)))&!int_times_inf;
272
273    assign underflow =
274       (format == `FP16) ?
275          (~ovf_ab&ovf_biased)|
276          (~|result_tmp[`FP16SW+`FP16EW-1:`FP16SW]) &
277          !(ovf_ab&ovf_biased|ovf_postnorm) &
278          !overflow&!invalid|(zero_a | zero_b) &
279          !(nan_a|nan_b|inf_a|inf_b) :
280       (format == `FP32) ?
281          (~ovf_ab&ovf_biased)|
282          (~|result_tmp[`FP32SW+`FP32EW-1:`FP32SW]) &
283          !(ovf_ab&ovf_biased|ovf_postnorm) &
284          !overflow&!invalid|(zero_a | zero_b) &
285          !(nan_a|nan_b|inf_a|inf_b) :
286       (format == `FP64) ?
287          (~ovf_ab&ovf_biased)|
288          (~|result_tmp[`FP64SW+`FP64EW-1:`FP64SW]) &
289          !(ovf_ab&ovf_biased|ovf_postnorm) &
290          !overflow&!invalid|(zero_a | zero_b) &
291          !(nan_a|nan_b|inf_a|inf_b) : 0;
292
293    // If overflow occurs and rounding mode equals round-to zero,
294    // result shall be rounded to largest representative number.
295    // e.x 0111101111111111.
296    assign overflow_tmp =
297       (format == `FP16) ?
298          ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
299          &result_tmp[`FP16SW+`FP16EW-1:`FP16SW]&!underflow)&!invalid :
300       (format == `FP32) ?
301          ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
302          &result_tmp[`FP32SW+`FP32EW-1:`FP32SW]&!underflow)&!invalid :
303       (format == `FP64) ?
304          ((ovf_ab&ovf_biased|ovf_postnorm&!underflow)|
305          &result_tmp[`FP64SW+`FP64EW-1:`FP64SW]&!underflow)&!invalid :
306       0;
307
308    assign overflow = overflow_tmp&!round_to_zero | int_times_inf;
```

```
309
310
311     // Compute special results.
312     assign product_nan =
313         (format == `FP16) ?
314             {1'b0, {`FP16EW{1'b1}}, {(`FP16SW-1){1'b0}}, 1'b1} :
315         (format == `FP32) ?
316             {1'b0, {`FP32EW{1'b1}}, {(`FP32SW-1){1'b0}}, 1'b1}:
317         (format == `FP64) ?
318             {1'b0, {`FP64EW{1'b1}}, {(`FP64SW-1){1'b0}}, 1'b1} :
319         0;
320
321     assign product_zero =
322         (format == `FP16) ?
323             {result_tmp[`FP16SW+`FP16EW], {(`FP16SW+`FP16EW){1'b0}}} :
324         (format == `FP32) ?
325             {result_tmp[`FP32SW+`FP32EW], {(`FP32SW+`FP32EW){1'b0}}} :
326         (format == `FP64) ?
327             {result_tmp[`FP64SW+`FP64EW], {(`FP64SW+`FP64EW){1'b0}}} :
328         0;
329
330     assign product_overflow =
331         (format == `FP16) ?
332             {result_tmp[`FP16SW+`FP16EW],
333             {`FP16EW{1'b1}}, {(`FP16SW){1'b0}}} :
334         (format == `FP32) ?
335             {result_tmp[`FP32SW+`FP32EW],
336             {`FP32EW{1'b1}}, {(`FP32SW){1'b0}}}:
337         (format == `FP64) ?
338             {result_tmp[`FP64SW+`FP64EW],
339             {`FP64EW{1'b1}}, {(`FP64SW){1'b0}}} :
340         0;
341
342     assign product_large =
343         (format == `FP16) ?
344             {result_tmp[`FP16SW+`FP16EW],
345             {(`FP16EW-1){1'b1}}, 1'b0, {(`FP16SW){1'b1}}} :
346         (format == `FP32) ?
347             {result_tmp[`FP32SW+`FP32EW],
348             {(`FP32EW-1){1'b1}}, 1'b0, {(`FP32SW){1'b1}}}:
349         (format == `FP64) ?
350             {result_tmp[`FP64SW+`FP64EW],
351             {(`FP64EW-1){1'b1}}, 1'b0, {(`FP64SW){1'b1}}} :
352         0;
353
354     assign product_min =
355         (format == `FP16) ?
356             {result_tmp[`FP16SW+`FP16EW],
357             {(`FP16EW-1){1'b0}}, 1'b1, {(`FP16SW){1'b0}}} :
358         (format == `FP32) ?
359             {result_tmp[`FP32SW+`FP32EW],
360             {(`FP32EW-1){1'b0}}, 1'b1, {(`FP32SW){1'b0}}}:
361         (format == `FP64) ?
362             {result_tmp[`FP64SW+`FP64EW],
363             {(`FP64EW-1){1'b0}}, 1'b1, {(`FP64SW){1'b0}}} :
364         0;
365
366     // Final product decided by exceptions.
367     assign result =
368         invalid ? product_nan :
369         overflow ? product_overflow :
370         underflow ? product_zero :
```

```
371         round_to_zero & overflow_tmp & !int_times_inf ? product_large :
372         result_tmp;
373
374     assign exceps[0] = invalid;
375     assign exceps[1] = inexact;
376     assign exceps[2] = overflow;
377     assign exceps[3] = underflow;
378
379 endmodule // rne64
```

Appendix C

Test Data Generator

```
1  // ---------------------------------------------------------------------------
2  // Author:         Espen Stenersen.
3  // Date:           Spring 2008.
4  // Description:    Generates FP16, FP32 and FP64 testvectors including
5  //                 "random" special inputs such as infinity x zero,
6  //                 nans ... Testfiles are written to fp16testfiles.txt,
7  //                 fp32testfiles.txt and fp64testfiles.txt.
8  // ---------------------------------------------------------------------------
9
10 #include <stdio.h>
11 #include <stdarg.h>
12 #include <string.h>
13 #include <xlocale.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16
17 #define FP16 0
18 #define FP32 1
19 #define FP64 2
20 #define FP16WIDTH 16;
21 #define FP32WIDTH 32;
22 #define FP64WIDTH 64;
23 #define FP16EXPONENT 5;
24 #define FP32EXPONENT 8;
25 #define FP64EXPONENT 11;
26
27 void usage();
28 void generate(int format, int testcases);
29 void generate_nan(int width, int exponent);
30 void generate_zero(int width, int exponent);
31 void generate_infinity(int width, int exponent);
32 void generate_special(int width, int exponent);
33 void generate_random(int width, int exponent);
34
35 FILE *f;
36
37 int main (int argc, char const *argv[])
38 {
39    int format;
40
41    // Initializes the random generator.
42    srand(time(0) * getpid());
43
```

```
44     if (argc < 3) usage();
45     else
46     {
47        if ( strcmp("-fp16", argv[1]) == 0 ) format = FP16;
48        else if ( strcmp("-fp32", argv[1]) == 0 ) format = FP32;
49        else if ( strcmp("-fp64", argv[1]) == 0 ) format = FP64;
50        else usage();
51
52        generate(format, atoi(argv[2]));
53     }
54     return 0;
55 }
56
57 void generate(int format, int testcases)
58 {
59     int i = 0;
60     int width = 0;
61     int exponent = 0;
62     int random = 0;
63
64     switch( format )
65     {
66        case FP16: width = FP16WIDTH; exponent = FP16EXPONENT; f = fopen
              ("fp16testcases.txt", "wt"); break;
67        case FP32: width = FP32WIDTH; exponent = FP32EXPONENT; f = fopen
              ("fp32testcases.txt", "wt"); break;
68        case FP64: width = FP64WIDTH; exponent = FP64EXPONENT; f = fopen
              ("fp64testcases.txt", "wt"); break;
69     }
70
71     // Genrates nan x nan.
72     generate_nan(width, exponent);
73     generate_nan(width, exponent);
74
75     // Generates zero x infinity.
76     generate_infinity(width, exponent);
77     generate_infinity(width, exponent);
78
79     // Generates zero x zero.
80     generate_zero(width, exponent);
81     generate_zero(width, exponent);
82
83     // Generates zero z infinity.
84     generate_zero(width, exponent);
85     generate_infinity(width, exponent);
86
87     // Generates infinity x nan.
88     generate_infinity(width, exponent);
89     generate_nan(width, exponent);
90
91     // Generates zero x nan.
92     generate_zero(width, exponent);
93     generate_nan(width, exponent);
94
95     i = 12;
96     while (i < testcases)
97     {
98        random = rand() % 999;
99        if (random == 14)
100       {
101          generate_special(width, exponent);
102       }
```

```
103         else
104         {
105             generate_random(width, exponent);
106         }
107         i++;
108     }
109     fclose (f);
110 }
111
112 void generate_random(int width, int exponent)
113 {
114     int j = 0;
115     int normalized = 0;
116     int bit = 0;
117
118     while (j < width)
119     {
120         bit = rand() % 2;
121         if (j < 1) fprintf(f, "%d", bit);
122         else if (j < exponent)
123         {
124             if (bit == 1) normalized++;
125             fprintf(f, "%d", bit);
126         }
127         else
128         {
129             if (j == exponent)
130             {
131                 if (normalized < 1)
132                     fprintf(f, "%d", 1);
133                 else
134                     fprintf(f, "%d", bit);
135             }
136             else
137                 fprintf(f, "%d", bit);
138         }
139         j++;
140     }
141     fprintf(f, "\n");
142 }
143 // Generate random special input vectos. e.x zero x infinity.
144 void generate_special(int width, int exponent)
145 {
146     int random = rand() % 6;
147
148     // Genrates nan x nan.
149     if (random == 0)
150     {
151         generate_nan(width, exponent);
152         generate_nan(width, exponent);
153     }
154     // Generates zero x infinity.
155     else if (random == 1)
156     {
157         generate_infinity(width, exponent);
158         generate_infinity(width, exponent);
159     }
160     // Generates zero x zero.
161     else if (random == 2)
162     {
163         generate_zero(width, exponent);
164         generate_zero(width, exponent);
```

```
165     }
166     // Generates zero z infinity.
167     else if (random == 3)
168     {
169         generate_zero(width, exponent);
170         generate_infinity(width, exponent);
171     }
172     // Generates infinity x nan.
173     else if (random == 4)
174     {
175         generate_infinity(width, exponent);
176         generate_nan(width, exponent);
177     }
178     // Generates zero x nan.
179     else if (random == 5)
180     {
181         generate_zero(width, exponent);
182         generate_nan(width, exponent);
183     }
184 }
185
186 // Generate NaN vectors.
187 void generate_nan(int width, int exponent)
188 {
189     int i = 0;
190     int bit = rand() % 2;
191     while (i < width)
192     {
193         if (i < 1) fprintf(f, "%d", bit);
194         else if (i < exponent + 1) fprintf(f, "%d", 1);
195         else if (i < width - 1) fprintf(f, "%d", 0);
196         else fprintf(f, "%d", 1);
197         i++;
198     }
199     fprintf(f, "\n");
200 }
201
202 // Generate zero vectors.
203 void generate_zero(int width, int exponent)
204 {
205     int i = 0;
206     int bit = rand() % 2;
207     while (i < width)
208     {
209         if (i < 1) fprintf(f, "%d", bit);
210         else fprintf(f, "%d", 0);
211         i++;
212     }
213     fprintf(f, "\n");
214 }
215
216 // Generate infinity vectors.
217 void generate_infinity(int width, int exponent)
218 {
219     int i = 0;
220     int bit = rand() % 2;
221     while (i < width)
222     {
223         if (i < 1) fprintf(f, "%d", bit);
224         else if (i < exponent + 1) fprintf(f, "%d", 1);
225         else fprintf(f, "%d", 0);
226         i++;
```

```
227     }
228     fprintf(f, "\n");
229 }
230
231 // Prints user info.
232 void usage()
233 {
234     printf("\nGenerates␣normalized␣IEEE␣conforming␣testvectors␣of␣
            desired␣format.\n");
235     printf("\nUsage:␣generatetestvectors␣<format>␣<number␣of␣testcases
            >\n");
236     printf("␣␣␣␣␣␣-fp16:␣16-bit␣floating-point␣vectors.\n");
237     printf("␣␣␣␣␣␣-fp32:␣32-bit␣floating-point␣vectors.\n");
238     printf("␣␣␣␣␣␣-fp64:␣64-bit␣floating-point␣vectors.\n");
239     printf("\n␣␣␣Ex:␣generatetestvectors␣-fp16␣10000\n\n");
240     printf("\n");
241 }
```

Appendix D

Simulation Sources

D.1 Vectorized DesignWare floating-point multiplier Source

```
1 // ----------------------------------------------------------------
2 // File.......: dw_vec_fp16_mult.v
3 // Author.....: Espen Stenersen
4 // Date.......: Thu Apr 24 16:40:38 CEST 2008
5 // Revision...: 1.0
6 // Description: Vectorized FP16 floating-point multiplier based on
7 //              the DesignWare simulation model.
8 // ----------------------------------------------------------------
9
10 `include "defines.v"
11
12 module dw_vec_fp_mult
13 (
14    dw_vectors,      // Input from testbench.
15    dw_mode,         // Input from testbench.
16    format,       // Input from testbench.
17    dw_products,     // Output to testbench.
18    dw_exceptions    // Output to testbench.
19 );
20
21
22    // input(s)
23    input [2*`BUS-1:0]              dw_vectors;
24    input [2:0]                     dw_mode;
25    input [1:0]                     format;
26
27    // output(s)
28    output [`BUS-1:0]               dw_products;
29    output [15:0]                   dw_exceptions;
30
31    // wire(s)
32    wire [7:0]                      fp16_mult0_status;
33    wire [7:0]                      fp16_mult1_status;
34    wire [7:0]                      fp16_mult2_status;
35    wire [7:0]                      fp16_mult3_status;
36    wire [`FP16SW+`FP16EW:0]        fp16_mult0_z;
37    wire [`FP16SW+`FP16EW:0]        fp16_mult1_z;
38    wire [`FP16SW+`FP16EW:0]        fp16_mult2_z;
```

```
39      wire [`FP16SW+`FP16EW:0]       fp16_mult3_z;
40      wire [`FP16SW+`FP16EW:0]       fp16_mult0_a;
41      wire [`FP16SW+`FP16EW:0]       fp16_mult0_b;
42      wire [`FP16SW+`FP16EW:0]       fp16_mult1_a;
43      wire [`FP16SW+`FP16EW:0]       fp16_mult1_b;
44      wire [`FP16SW+`FP16EW:0]       fp16_mult2_a;
45      wire [`FP16SW+`FP16EW:0]       fp16_mult2_b;
46      wire [`FP16SW+`FP16EW:0]       fp16_mult3_a;
47      wire [`FP16SW+`FP16EW:0]       fp16_mult3_b;
48      wire [`FP16SW+`FP16EW:0]       fp16_mult0_z_tmp;
49      wire [`FP16SW+`FP16EW:0]       fp16_mult1_z_tmp;
50      wire [`FP16SW+`FP16EW:0]       fp16_mult2_z_tmp;
51      wire [`FP16SW+`FP16EW:0]       fp16_mult3_z_tmp;
52
53      wire [7:0]                     fp32_mult0_status;
54      wire [7:0]                     fp32_mult1_status;
55      wire [7:0]                     fp32_mult2_status;
56      wire [7:0]                     fp32_mult3_status;
57      wire [`FP32SW+`FP32EW:0]       fp32_mult0_z;
58      wire [`FP32SW+`FP32EW:0]       fp32_mult1_z;
59      wire [`FP32SW+`FP32EW:0]       fp32_mult2_z;
60      wire [`FP32SW+`FP32EW:0]       fp32_mult3_z;
61      wire [`FP32SW+`FP32EW:0]       fp32_mult0_a;
62      wire [`FP32SW+`FP32EW:0]       fp32_mult0_b;
63      wire [`FP32SW+`FP32EW:0]       fp32_mult1_a;
64      wire [`FP32SW+`FP32EW:0]       fp32_mult1_b;
65      wire [`FP32SW+`FP32EW:0]       fp32_mult2_a;
66      wire [`FP32SW+`FP32EW:0]       fp32_mult2_b;
67      wire [`FP32SW+`FP32EW:0]       fp32_mult3_a;
68      wire [`FP32SW+`FP32EW:0]       fp32_mult3_b;
69      wire [`FP32SW+`FP32EW:0]       fp32_mult0_z_tmp;
70      wire [`FP32SW+`FP32EW:0]       fp32_mult1_z_tmp;
71      wire [`FP32SW+`FP32EW:0]       fp32_mult2_z_tmp;
72      wire [`FP32SW+`FP32EW:0]       fp32_mult3_z_tmp;
73
74      wire [7:0]                     fp64_mult0_status;
75      wire [7:0]                     fp64_mult1_status;
76      wire [`FP64SW+`FP64EW:0]       fp64_mult0_z;
77      wire [`FP64SW+`FP64EW:0]       fp64_mult1_z;
78      wire [`FP64SW+`FP64EW:0]       fp64_mult0_a;
79      wire [`FP64SW+`FP64EW:0]       fp64_mult0_b;
80      wire [`FP64SW+`FP64EW:0]       fp64_mult1_a;
81      wire [`FP64SW+`FP64EW:0]       fp64_mult1_b;
82      wire [`FP64SW+`FP64EW:0]       fp64_mult0_z_tmp;
83      wire [`FP64SW+`FP64EW:0]       fp64_mult1_z_tmp;
84
85      // reg(s)
86
87
88      // ------------------------------------------------------------------
89      // Module instantiation.
90      // ------------------------------------------------------------------
91
92      dw_fp_mult #(`FP16SW, `FP16EW, 1) fp16_mult0
93      (
94          .a        (fp16_mult0_a),
95          .b        (fp16_mult0_b),
96          .rnd      (dw_mode),
97          .z        (fp16_mult0_z),
98          .status   (fp16_mult0_status)
99      );
100     dw_fp_mult #(`FP16SW, `FP16EW, 1) fp16_mult1
```

```
101     (
102         .a          (fp16_mult1_a),
103         .b          (fp16_mult1_b),
104         .rnd        (dw_mode),
105         .z          (fp16_mult1_z),
106         .status     (fp16_mult1_status)
107     );
108     dw_fp_mult #(`FP16SW, `FP16EW, 1) fp16_mult2
109     (
110         .a          (fp16_mult2_a),
111         .b          (fp16_mult2_b),
112         .rnd        (dw_mode),
113         .z          (fp16_mult2_z),
114         .status     (fp16_mult2_status)
115     );
116     dw_fp_mult #(`FP16SW, `FP16EW, 1) fp16_mult3
117     (
118         .a          (fp16_mult3_a),
119         .b          (fp16_mult3_b),
120         .rnd        (dw_mode),
121         .z          (fp16_mult3_z),
122         .status     (fp16_mult3_status)
123     );
124
125     dw_fp_mult #(`FP32SW, `FP32EW, 1) fp32_mult0
126     (
127         .a          (fp32_mult0_a),
128         .b          (fp32_mult0_b),
129         .rnd        (dw_mode),
130         .z          (fp32_mult0_z),
131         .status     (fp32_mult0_status)
132     );
133     dw_fp_mult #(`FP32SW, `FP32EW, 1) fp32_mult1
134     (
135         .a          (fp32_mult1_a),
136         .b          (fp32_mult1_b),
137         .rnd        (dw_mode),
138         .z          (fp32_mult1_z),
139         .status     (fp32_mult1_status)
140     );
141     dw_fp_mult #(`FP32SW, `FP32EW, 1) fp32_mult2
142     (
143         .a          (fp32_mult2_a),
144         .b          (fp32_mult2_b),
145         .rnd        (dw_mode),
146         .z          (fp32_mult2_z),
147         .status     (fp32_mult2_status)
148     );
149     dw_fp_mult #(`FP32SW, `FP32EW, 1) fp32_mult3
150     (
151         .a          (fp32_mult3_a),
152         .b          (fp32_mult3_b),
153         .rnd        (dw_mode),
154         .z          (fp32_mult3_z),
155         .status     (fp32_mult3_status)
156     );
157
158     dw_fp_mult #(`FP64SW, `FP64EW, 1) fp64_mult0
159     (
160         .a          (fp64_mult0_a),
161         .b          (fp64_mult0_b),
162         .rnd        (dw_mode),
```

```
163        .z         (fp64_mult0_z),
164        .status    (fp64_mult0_status)
165    );
166    dw_fp_mult #(`FP64SW, `FP64EW, 1) fp64_mult1
167    (
168        .a         (fp64_mult1_a),
169        .b         (fp64_mult1_b),
170        .rnd       (dw_mode),
171        .z         (fp64_mult1_z),
172        .status    (fp64_mult1_status)
173    );
174
175
176    // ----------------------------------------------------------------
177    // Input selections.
178    // ----------------------------------------------------------------
179    assign fp16_mult0_a = dw_vectors[1*`FP16W-1:0*`FP16W];
180    assign fp16_mult0_b = dw_vectors[2*`FP16W-1:1*`FP16W];
181    assign fp16_mult1_a = dw_vectors[3*`FP16W-1:2*`FP16W];
182    assign fp16_mult1_b = dw_vectors[4*`FP16W-1:3*`FP16W];
183    assign fp16_mult2_a = dw_vectors[5*`FP16W-1:4*`FP16W];
184    assign fp16_mult2_b = dw_vectors[6*`FP16W-1:5*`FP16W];
185    assign fp16_mult3_a = dw_vectors[7*`FP16W-1:6*`FP16W];
186    assign fp16_mult3_b = dw_vectors[8*`FP16W-1:7*`FP16W];
187    assign fp32_mult0_a = dw_vectors[1*`FP32W-1:0*`FP32W];
188    assign fp32_mult0_b = dw_vectors[2*`FP32W-1:1*`FP32W];
189    assign fp32_mult1_a = dw_vectors[3*`FP32W-1:2*`FP32W];
190    assign fp32_mult1_b = dw_vectors[4*`FP32W-1:3*`FP32W];
191    assign fp32_mult2_a = dw_vectors[5*`FP32W-1:4*`FP32W];
192    assign fp32_mult2_b = dw_vectors[6*`FP32W-1:5*`FP32W];
193    assign fp32_mult3_a = dw_vectors[7*`FP32W-1:6*`FP32W];
194    assign fp32_mult3_b = dw_vectors[8*`FP32W-1:7*`FP32W];
195    assign fp64_mult0_a = dw_vectors[1*`FP64W-1:0*`FP64W];
196    assign fp64_mult0_b = dw_vectors[2*`FP64W-1:1*`FP64W];
197    assign fp64_mult1_a = dw_vectors[3*`FP64W-1:2*`FP64W];
198    assign fp64_mult1_b = dw_vectors[4*`FP64W-1:3*`FP64W];
199
200
201
202    // ----------------------------------------------------------------
203    // Set exceptions.
204    // ----------------------------------------------------------------
205
206    // Invalid. dw_status[2].
207    assign dw_exceptions[0] =
208        (format == `FP16) ?
209        fp16_mult0_status[2] :
210        (format == `FP32) ?
211        fp32_mult0_status[2] :
212        (format == `FP64) ?
213        fp64_mult0_status[2] :
214        0;
215
216    assign dw_exceptions[1] =
217        (format == `FP16) ?
218        fp16_mult1_status[2] :
219        (format == `FP32) ?
220        fp32_mult1_status[2] :
221        (format == `FP64) ?
222        fp64_mult1_status[2] :
223        0;
224
```

```
225     assign dw_exceptions[2] =
226         (format == `FP16) ?
227         fp16_mult2_status[2] :
228         (format == `FP32) ?
229         fp32_mult2_status[2] :
230         (format == `FP64) ?
231         0 : 0;
232
233     assign dw_exceptions[3] =
234         (format == `FP16) ?
235         fp16_mult3_status[2] :
236         (format == `FP32) ?
237         fp32_mult3_status[2] :
238         (format == `FP64) ?
239         0 : 0;
240
241     // Inexact. dw_status[5].
242     assign dw_exceptions[4] =
243         (format == `FP16) ?
244         fp16_mult0_status[5]|fp16_mult0_status[3] :
245         (format == `FP32) ?
246         fp32_mult0_status[5]|fp32_mult0_status[3] :
247         (format == `FP64) ?
248         fp64_mult0_status[5]|fp64_mult0_status[3] :
249         0;
250
251     assign dw_exceptions[5] =
252         (format == `FP16) ?
253         fp16_mult1_status[5]|fp16_mult1_status[3] :
254         (format == `FP32) ?
255         fp32_mult1_status[5]|fp32_mult1_status[3] :
256         (format == `FP64) ?
257         fp64_mult1_status[5]|fp64_mult1_status[3] :
258         0;
259
260     assign dw_exceptions[6] =
261         (format == `FP16) ?
262         fp16_mult2_status[5]|fp16_mult2_status[3] :
263         (format == `FP32) ?
264         fp32_mult2_status[5]|fp32_mult2_status[3] :
265         (format == `FP64) ?
266         0 : 0;
267
268     assign dw_exceptions[7] =
269         (format == `FP16) ?
270         fp16_mult3_status[5]|fp16_mult3_status[3] :
271         (format == `FP32) ?
272         fp32_mult3_status[5]|fp32_mult3_status[3] :
273         (format == `FP64) ?
274         0 : 0;
275
276     // Overflow. dw_status[1].
277     assign dw_exceptions[8] =
278         (format == `FP16) ?
279         fp16_mult0_status[1] :
280         (format == `FP32) ?
281         fp32_mult0_status[1] :
282         (format == `FP64) ?
283         fp64_mult0_status[1] :
284         0;
285
286     assign dw_exceptions[9] =
```

```
287         (format == `FP16) ?
288         fp16_mult1_status[1] :
289         (format == `FP32) ?
290         fp32_mult1_status[1] :
291         (format == `FP64) ?
292         fp64_mult1_status[1] :
293         0;
294
295     assign dw_exceptions[10] =
296         (format == `FP16) ?
297         fp16_mult2_status[1] :
298         (format == `FP32) ?
299         fp32_mult2_status[1] :
300         (format == `FP64) ?
301         0 : 0;
302
303     assign dw_exceptions[11] =
304         (format == `FP16) ?
305         fp16_mult3_status[1] :
306         (format == `FP32) ?
307         fp32_mult3_status[1] :
308         (format == `FP64) ?
309         0 : 0;
310
311     // Underflow. dw_status[0] |Â dw_status[3] (underflow|denormal).
312     assign dw_exceptions[12] =
313         (format == `FP16) ?
314         fp16_mult0_status[0]|fp16_mult0_status[3] :
315         (format == `FP32) ?
316         fp32_mult0_status[0]|fp32_mult0_status[3] :
317         (format == `FP64) ?
318         fp64_mult0_status[0]|fp64_mult0_status[3] :
319         0;
320
321     assign dw_exceptions[13] =
322         (format == `FP16) ?
323         fp16_mult1_status[0]|fp16_mult1_status[3] :
324         (format == `FP32) ?
325         fp32_mult1_status[0]|fp32_mult1_status[3] :
326         (format == `FP64) ?
327         fp64_mult1_status[0]|fp64_mult1_status[3] :
328         0;
329
330     assign dw_exceptions[14] =
331         (format == `FP16) ?
332         fp16_mult2_status[0]|fp16_mult2_status[3] :
333         (format == `FP32) ?
334         fp32_mult2_status[0]|fp32_mult2_status[3] :
335         (format == `FP64) ?
336         0 : 0;
337
338     assign dw_exceptions[15] =
339         (format == `FP16) ?
340         fp16_mult3_status[0]|fp16_mult3_status[3] :
341         (format == `FP32) ?
342         fp32_mult3_status[0]|fp32_mult3_status[3] :
343         (format == `FP64) ?
344         0 : 0;
345
346     // Flush product to zero if denormal output from dw_dp_mult.
347     assign fp16_mult0_z_tmp = fp16_mult0_status[3] ?
348         {fp16_mult0_z[`FP16W-1], fp16_mult0_z[`FP16W-2:0]&1'b0} :
```

```
349         fp16_mult0_z;
350
351     assign fp16_mult1_z_tmp = fp16_mult1_status[3] ?
352         {fp16_mult1_z[`FP16W-1], fp16_mult1_z[`FP16W-2:0]&1'b0} :
353         fp16_mult1_z;
354
355     assign fp16_mult2_z_tmp = fp16_mult2_status[3] ?
356         {fp16_mult2_z[`FP16W-1], fp16_mult2_z[`FP16W-2:0]&1'b0} :
357         fp16_mult2_z;
358
359     assign fp16_mult3_z_tmp = fp16_mult3_status[3] ?
360         {fp16_mult3_z[`FP16W-1], fp16_mult3_z[`FP16W-2:0]&1'b0} :
361         fp16_mult3_z;
362
363     assign fp32_mult0_z_tmp = fp32_mult0_status[3] ?
364         {fp32_mult0_z[`FP32W-1], fp32_mult0_z[`FP32W-2:0]&1'b0} :
365         fp32_mult0_z;
366
367     assign fp32_mult1_z_tmp = fp32_mult1_status[3] ?
368         {fp32_mult1_z[`FP32W-1], fp32_mult1_z[`FP32W-2:0]&1'b0} :
369         fp32_mult1_z;
370
371     assign fp32_mult2_z_tmp = fp32_mult2_status[3] ?
372         {fp32_mult2_z[`FP32W-1], fp32_mult2_z[`FP32W-2:0]&1'b0} :
373         fp32_mult2_z;
374
375     assign fp32_mult3_z_tmp = fp32_mult3_status[3] ?
376         {fp32_mult3_z[`FP32W-1], fp32_mult3_z[`FP32W-2:0]&1'b0} :
377         fp32_mult3_z;
378
379     assign fp64_mult0_z_tmp = fp64_mult0_status[3] ?
380         {fp64_mult0_z[`FP64W-1], fp64_mult0_z[`FP64W-2:0]&1'b0} :
381         fp64_mult0_z;
382
383     assign fp64_mult1_z_tmp = fp64_mult1_status[3] ?
384         {fp64_mult1_z[`FP64W-1], fp64_mult1_z[`FP64W-2:0]&1'b0} :
385         fp64_mult1_z;
386
387
388     // ---------------------------------------------------------------
389     // Output mux.
390     // ---------------------------------------------------------------
391     assign dw_products = (format == `FP16) ?
392         {fp16_mult3_z_tmp, fp16_mult2_z_tmp,
393         fp16_mult1_z_tmp, fp16_mult0_z_tmp} :
394         (format == `FP32) ?
395         {fp32_mult3_z_tmp, fp32_mult2_z_tmp,
396         fp32_mult1_z_tmp, fp32_mult0_z_tmp} :
397         (format == `FP64) ?
398         {fp64_mult1_z_tmp, fp64_mult0_z_tmp} : 0;
399 endmodule // dw_vec_fp_mult
```

D.2 Testbench Sources

```
1 // ---------------------------------------------------------------------------
2 // File.......: vec_fp_mult_tb.v
3 // Author.....: Espen Stenersen
4 // Date.......: Thu Apr 17 13:49:28 CEST 2008
5 // Revision...: 1.0
6 // Description: Testbench for top module vec_fp_mult.
7 // ---------------------------------------------------------------------------
8
9 `include "../rtl/defines.v"
10
11 //`timescale
12 `define CLK_PERIOD 1
13
14 module vec_fp_mult_tb;
15
16    `include "../tb/defines_tb.v"
17    `include "../tb/debug.v"
18
19    parameter W            = `FP16W;
20    parameter SW           = `FP16SW;
21    parameter EW           = `FP16EW;
22    parameter FORMAT       = `FP16;
23    parameter MODE         = `ZERO;
24    parameter VECTORS      = 100000;
25
26    // wire(s)
27    wire [`BUS-1:0]        products;
28    wire [15:0]            exceptions;
29    wire                   ready;
30    wire                   exceptions_failed;
31    wire                   products_failed;
32    wire [3:0]             nan, inf, zero;
33
34    wire [`BUS-1:0]        dw_products;
35    wire [15:0]            dw_exceptions;
36    wire [2*`BUS-1:0]      dw_vectors;
37
38    // reg(s)
39    reg [W-1:0]            testmem [0:VECTORS-1];
40    reg [`BUS-1:0]         vectors;
41    reg [W-1:0]            A0, B0, A1, B1;
42    reg [1:0]              format;
43    reg [1:0]              mode;
44    reg [15:0]             clear;
45    reg                    start;
46    reg                    clk;
47    reg                    reset_n;
48
49
50    reg [2:0]              dw_mode;
51
52    // Counters.
53    integer                i_vec, i_ans, i_passed, i_failed, i_total;
54    integer                i_nan, i_zero, i_inf, i_inf_times_zero;
55    integer                step, i_ovf, i_unf, i_inv, i_inx;
56    integer                i_nan_times_any;
57
58
59    // ------------------------------------------------------------------------
```

```
60  // Module instantiation.
61  // ------------------------------------------------------------
62  vec_fp_mult DUT
63  (
64      .start          (start),        // Input. Starts computation.
65      .vectors        (vectors),      // Input. FP vectors.
66      .format         (format),       // Input. Format of vectors.
67      .mode           (mode),         // Input. Rounding mode.
68      .clear          (clear),        // Input. Clears exceptions.
69      .products       (products),     // Output. Computed products.
70      .exceptions     (exceptions),   // Output. Exceptions raised.
71      .ready          (ready),        // Output. Output ready.
72      .clk            (clk),
73      .reset_n        (reset_n)
74  );
75
76  dw_vec_fp_mult dw_vec_fp_mult
77  (
78      .dw_vectors     (dw_vectors),     // Input from testbench.
79      .dw_mode        (dw_mode),        // Input from testbench.
80      .format         (format),         // Input from testbench.
81      .dw_products    (dw_products),    // Output to testbench.
82      .dw_exceptions  (dw_exceptions)   // Output to testbench.
83  );
84
85
86  // ------------------------------------------------------------
87  // Initials.
88  // ------------------------------------------------------------
89
90  // Generate stimuli.
91  initial begin
92
93      // --------------------------------------------------------
94      // Verbosity levels:
95      //     0: Only final report.
96      //     1: Signal events and updates.
97      //     2: Error messages.
98      //     3: Elaborated error messages with product vectors,
99      //        exception vectors and input vectors that caused the
100     //        error.
101     //     4: 1 and 3 combined.
102     // --------------------------------------------------------
103
104     verbosity(3);
105
106     initialize; // Call to initialize task.
107
108
109     @(posedge clk) // wait cycle.
110     @(posedge clk) // wait cycle.
111     @(posedge clk) reset_n = 1;
112     @(posedge clk) // wait cycle.
113     @(posedge clk) // wait cycle.
114
115     for (i_vec = 0; i_vec < VECTORS; i_vec = i_vec + step) begin
116        @(posedge clk) begin
117           start = 1;
118
119           if (format == `FP16) begin
120              vectors[1*`FP16W-1:0*`FP16W] <= testmem[i_vec + 0];
121              vectors[2*`FP16W-1:1*`FP16W] <= testmem[i_vec + 1];
```

```
122                 vectors[3*`FP16W-1:2*`FP16W] <= testmem[i_vec + 2];
123                 vectors[4*`FP16W-1:3*`FP16W] <= testmem[i_vec + 3];
124                 A0 <= testmem[i_vec + 0];
125                 B0 <= testmem[i_vec + 1];
126                 A1 <= testmem[i_vec + 2];
127                 B1 <= testmem[i_vec + 3];
128             end
129             else if (format == `FP32) begin
130                 vectors[1*`FP32W-1:0*`FP32W] <= testmem[i_vec + 0];
131                 vectors[2*`FP32W-1:1*`FP32W] <= testmem[i_vec + 1];
132                 vectors[3*`FP32W-1:2*`FP32W] <= testmem[i_vec + 2];
133                 vectors[4*`FP32W-1:3*`FP32W] <= testmem[i_vec + 3];
134                 A0 <= testmem[i_vec + 0];
135                 B0 <= testmem[i_vec + 1];
136                 A1 <= testmem[i_vec + 2];
137                 B1 <= testmem[i_vec + 3];
138             end
139             else if (format == `FP64) begin
140                 vectors[1*`FP64W-1:0*`FP64W] <= testmem[i_vec + 0];
141                 vectors[2*`FP64W-1:1*`FP64W] <= testmem[i_vec + 1];
142                 A0 <= testmem[i_vec + 0];
143                 B0 <= testmem[i_vec + 1];
144             end
145             else begin
146                 vectors <= 0;
147                 A0 <= 0;
148                 B0 <= 0;
149                 A1 <= 0;
150                 B1 <= 0;
151             end
152         end
153     end
154     // Empty pipeline.
155     @(posedge clk) // wait cycle.
156     @(posedge clk) // wait cycle.
157     start = 0;
158     @(posedge clk) // wait cycle.
159     @(posedge clk) // wait cycle.
160     @(posedge clk) // wait cycle.
161     print_report;
162     $finish;
163 end
164
165 // ---------------------------------------------------------------------
166 // Sequential test logic.
167 // ---------------------------------------------------------------------
168
169 // clock generator.
170 always #`CLK_PERIOD clk = !clk;
171
172
173
174 // Monitor / checker.
175 always @(ready) begin
176     if (reset_n == 0) begin
177         i_ans <= 0;
178         i_total <= 0;
179     end
180
181     // When products and exceptions are ready at output.
182     if (ready == 1) begin
183         i_total <= i_total + 1;
```

```
184
185          if (format == `FP64) begin
186             i_ans <= i_ans + 4;
187          end
188          else begin
189             i_ans <= i_ans + 8;
190          end
191
192
193          if ((products != dw_products)|
194             (exceptions != dw_exceptions)) begin
195             i_failed = i_failed + 1;
196          end
197          else begin
198             i_passed = i_passed + 1;
199          end
200       end
201    end
202
203
204    // ------------------------------------------------------------------------
205    // Combinaional test logic.
206    // ------------------------------------------------------------------------
207
208    // Clears exceptions when arised.
209 /* always @ (ready) begin
210       if (ready) begin
211          clear = 'hffff;
212       end
213       else if (!ready) begin
214          clear = 'h0000;
215       end
216    end*/
217
218
219    // ------------------------------------------------------------------------
220    // Produces test statistics.
221    // ------------------------------------------------------------------------
222    // Counts infity inputs.
223    integer i0;
224    always @ (inf or start) begin
225       for (i0 = 0; i0 < 4; i0 = i0 + 1) begin
226          if (inf[i0]&start) i_inf = i_inf + 1;
227       end
228    end
229    // Counts zero inputs.
230    integer i1;
231    always @ (zero or start) begin
232       for (i1 = 0; i1 < 4; i1 = i1 + 1) begin
233          if (zero[i1]&start) i_zero = i_zero + 1;
234       end
235    end
236    // Counts invalid inputs.
237    integer i2;
238    always @ (nan or start) begin
239       for (i2 = 0; i2 < 4; i2 = i2 + 1) begin
240          if (nan[i2]&start) i_nan = i_nan + 1;
241       end
242    end
243    // Counts infinity times zero inputs.
244    integer i3;
245    always @ (inf or zero or start) begin
```

```
246         for (i3 = 0; i3 < 2; i3 = i3 + 2) begin
247             if (inf[i3]&zero[i3+1]&start)
248                 i_inf_times_zero = i_inf_times_zero + 1;
249         end
250         for (i3 = 0; i3 < 2; i3 = i3 + 2) begin
251             if (inf[i3+1]&zero[i3]&start)
252                 i_inf_times_zero = i_inf_times_zero + 1;
253         end
254     end
255     // Counts invalid times any number (not invalid times invalid).
256     integer i4;
257     always @ (nan or start) begin
258         for (i4 = 0; i4 < 2; i4 = i4 + 1) begin
259             if (nan[i4]&!nan[i4+1]&start)
260                 i_nan_times_any = i_nan_times_any + 1;
261         end
262     end
263     // Counts underflows.
264     integer o0;
265     always @ (ready or exceptions) begin
266         for (o0 = 0; o0 < 4; o0 = o0 + 1) begin
267             if (exceptions[12 + o0]&ready) i_unf = i_unf + 1;
268         end
269     end
270     // Counts overflows.
271     integer o1;
272     always @ (ready or exceptions) begin
273         for (o1 = 0; o1 < 4; o1 = o1 + 1) begin
274             if (exceptions[8 + o1]&ready) i_ovf = i_ovf + 1;
275         end
276     end
277     // Counts inexacts.
278     integer o2;
279     always @ (ready or exceptions) begin
280         for (o2 = 0; o2 < 4; o2 = o2 + 1) begin
281             if (exceptions[4 + o2]&ready) i_inx = i_inx + 1;
282         end
283     end
284     // Counts invalids.
285     integer o3;
286     always @ (ready or exceptions) begin
287         for (o3 = 0; o3 < 4; o3 = o3 + 1) begin
288             if (exceptions[0 + o3]&ready) i_inv = i_inv + 1;
289         end
290     end
291
292
293     // ------------------------------------------------------------------------
294     // Assigns.
295     // ------------------------------------------------------------------------
296
297     assign dw_vectors =
298         (format == `FP16) ?
299         {testmem[i_ans + 7], testmem[i_ans + 6],
300         testmem[i_ans + 5], testmem[i_ans + 4],
301         testmem[i_ans + 3], testmem[i_ans + 2],
302         testmem[i_ans + 1], testmem[i_ans + 0]} :
303         (format == `FP32) ?
304         {testmem[i_ans + 7], testmem[i_ans + 6],
305         testmem[i_ans + 5], testmem[i_ans + 4],
306         testmem[i_ans + 3], testmem[i_ans + 2],
307         testmem[i_ans + 1], testmem[i_ans + 0]} :
```

```
308        (format == `FP64) ?
309        {testmem[i_ans + 3], testmem[i_ans + 2],
310        testmem[i_ans + 1], testmem[i_ans + 0]} : 0;
311
312
313     assign nan[0] = (&A0[W-2:SW])&(|A0[SW-1:0]);
314     assign inf[0] = (&A0[W-2:SW])&(~|A0[SW-1:0]);
315     assign zero[0] = (~|A0[W-2:SW])&(~|A0[SW-1:0]);
316
317     assign nan[1] = (&A1[W-2:SW])&(|A1[SW-1:0]);
318     assign inf[1] = (&A1[W-2:SW])&(~|A1[SW-1:0]);
319     assign zero[1] = (~|A1[W-2:SW])&(~|A1[SW-1:0]);
320
321     assign nan[2] = (&B0[W-2:SW])&(|B0[SW-1:0]);
322     assign inf[2] = (&B0[W-2:SW])&(~|B0[SW-1:0]);
323     assign zero[2] = (~|B0[W-2:SW])&(~|B0[SW-1:0]);
324
325     assign nan[3] = (&B1[W-2:SW])&(|B1[SW-1:0]);
326     assign inf[3] = (&B1[W-2:SW])&(~|B1[SW-1:0]);
327     assign zero[3] = (~|B1[W-2:SW])&(~|B1[SW-1:0]);
328
329
330
331     // ---------------------------------------------------------------------------
332     // Tasks.
333     // ---------------------------------------------------------------------------
334
335
336     task initialize;
337        begin
338           set_mode(MODE); format = FORMAT;
339
340           clk = 1; reset_n = 0; clear = 0; start = 0;
341           A0 = 0; B0 = 0; B1 = 0; A1 = 0; vectors = 0;
342           i_vec = 0; i_ans = 0; i_passed = 0; i_failed = 0;
343           i_nan = 0; i_zero = 0; i_inf = 0; i_inf_times_zero = 0;
344           i_nan_times_any = 0; i_ovf = 0; i_unf = 0; i_inv = 0;
345           i_inx = 0;
346
347           // Opens correct testcase readfile.
348           if (format == `FP16) begin
349              step = 4;
350              $readmemb(`FP16TESTCASES, testmem);
351           end
352           else if (format == `FP32) begin
353              step = 4;
354              $readmemb(`FP32TESTCASES, testmem);
355           end
356           else if (format == `FP64) begin
357              step = 2;
358              $readmemb(`FP64TESTCASES, testmem);
359           end
360           $display("");
361        end
362     endtask
363
364
365     // Sets rounding mode for both floating-point multipliers.
366     task set_mode;
367        input [1:0] r_mode;
368        begin
369           case (r_mode)
```

```
370             `EVEN: begin
371                 mode = `EVEN;
372                 dw_mode = `DW_EVEN;
373             end
374             `PINF: begin
375                 mode = `PINF;
376                 dw_mode = `DW_PINF;
377             end
378             `NINF: begin
379                 mode = `NINF;
380                 dw_mode = `DW_NINF;
381             end
382             `ZERO: begin
383                 mode = `ZERO;
384                 dw_mode = `DW_ZERO;
385             end
386             default: begin
387                 $display("Not valid rounding mode!");
388                 $finish;
389             end
390         endcase
391     end
392 endtask
393
394 endmodule // vec_fp_mult_tb
```

```
 1 // ------------------------------------------------------------------------
 2 // File.......: debug.v
 3 // Author.....: Espen Stenersen
 4 // Date.......: Sat Apr 26 01:04:00 CEST 2008
 5 // Revision...: 1.0
 6 // Description: Tasks for debuging design. Reports errors and signal
 7 //              status at different verbosity level.
 8 // ------------------------------------------------------------------------
 9
10 reg v1;
11 reg v2;
12 reg v3;
13 reg v4;
14
15 // ----------------------------------------------------------------------
16 // Verbosity levels:
17 // 0: Only final report.
18 // 1: Signal events and updates.
19 // 2: Error messages.
20 // 3: Elaborated error messages with product vectors, exception
21 //    vectors and input vectors that caused the error.
22 // 4: 1 and 3 combined.
23 // ----------------------------------------------------------------------
24
25 task verbosity;
26     input [2:0] verbosity;
27     begin
28         case (verbosity)
29             0: begin
30                 v1 = 0; v2 = 0; v3 = 0; v4 = 0;
31             end
32             // Signal updates.
33             1: begin
34                 v1 = 1;
35                 print_header;
36             end
```

```
37
38              // Error messages.
39              2: begin
40                 v2 = 1;
41              end
42
43              // Elaborated messages.
44              3: begin
45                 v3 = 1;
46              end
47
48              // Elaborated messages with signal updates.
49              4: begin
50                 v4 = 1;
51              end
52
53              default: begin
54                 v1 = 0; v2 = 0; v3 = 0; v4 = 0;
55              end
56           endcase
57        end
58     endtask
59
60     // ------------------------------------------------------------
61     // Prints error elaborated error messages.
62     // ------------------------------------------------------------
63
64     always @ (ready) begin
65        if ((v2|v3)&(ready == 1)) begin
66           if ((products != dw_products)|
67              (exceptions != dw_exceptions)) begin
68              print_error;
69           end
70        end
71     end
72
73
74     // ------------------------------------------------------------
75     // Updates the signal status print-out.
76     // ------------------------------------------------------------
77     always @ (reset_n) begin
78        if (v1|v4)
79           $display("@ %0d\t\t| reset_n\t\t| %b", ($time)/2, reset_n);
80     end
81     always @ (start) begin
82        if (v1|v4)
83           $display("@ %0d\t\t| start\t\t\t| %b", ($time)/2, start);
84     end
85     always @ (ready) begin
86        if (v1)
87           $display("@ %0d\t\t| ready\t\t\t| %b", ($time)/2, ready);
88        if (v4&ready) print_error;
89        if (v4&!ready)
90           $display("@ %0d\t\t| ready\t\t\t| %b", ($time)/2, ready);
91     end
92     always @ (clear) begin
93        if (v1|v4)
94           $display("@ %0d\t\t| clear\t\t\t| %b %b %b %b", ($time)/2,
95                    clear[15:12], clear[11:8], clear[7:4], clear[3:0]);
96     end
97     always @ (ready) begin
98        if (ready == 1) begin
```

```
99          if ((v1)&(products != dw_products))
100             $display("@ %0d\t\t| products\t\t| ERROR!", ($time)/2);
101       end
102    end
103    always @ (ready) begin
104       if (ready == 1) begin
105          if ((v1)&(exceptions != dw_exceptions))
106             $display("@ %0d\t\t| exceptions\t\t| ERROR!", ($time)/2);
107       end
108    end
109    always @ (format) begin
110       if (v1|v4) begin
111          case (format)
112             `FP16: begin
113                $display("@ %0d\t\t| format\t\t| 16-bit floating-point 
                      (\'b%b)",
114                ($time)/2, format);
115             end
116             `FP32: begin
117                $display("@ %0d\t\t| format\t\t| 32-bit floating-point 
                      (\'b%b)",
118                ($time)/2, format);
119             end
120             `FP64: begin
121                $display("@ %0d\t\t| format\t\t| 64-bit floating-point 
                      (\'b%b)",
122                ($time)/2, format);
123             end
124          endcase
125       end
126    end
127    always @ (mode) begin
128       if (v1|v4) begin
129          case (mode)
130             `EVEN: begin
131                $display("@ %0d\t\t| mode\t\t\t| Round-to-nearest even 
                      (\'b%b)",
132                ($time)/2, mode);
133             end
134             `PINF: begin
135                $display("@ %0d\t\t| mode\t\t\t| Round-to-positive 
                      infinity (\'b%b)",
136                ($time)/2, mode);
137             end
138             `NINF: begin
139                $display("@ %0d\t\t| mode\t\t\t| Round-to-negative 
                      infinity (\'b%b)",
140                ($time)/2, mode);
141             end
142             `ZERO: begin
143                $display("@ %0d\t\t| mode\t\t\t| Round-to zero (\'b%b)"
                      ,
144                ($time)/2, mode);
145             end
146          endcase
147       end
148    end
149    always @ (exceptions) begin
150       if (v4) begin
151          $display("@ %0d\t\t| exceptions\t\t| %b %b %b %b", $time/2,
152          exceptions[15:12], exceptions[11:8], exceptions[7:4],
                exceptions[3:0]);
```

```
153         end
154     end
155
156
157     // ----------------------------------------------------------------
158     // Prints error messages.
159     // ----------------------------------------------------------------
160
161     task print_error;
162         begin
163             if (i_failed == 1) print_header;
164
165             if ((products != dw_products)) begin
166                 $write("------------------------------------------");
167                 $write("------------------------------------------\n");
168                 $display("@ %0d\t\t| products\t\t| ERROR: Products faild."
                    ,
169                 $time/2);
170                 $write("------------------------------------------");
171                 $write("------------------------------------------\n");
172             end
173             else if ((exceptions != dw_exceptions)) begin
174                 $write("------------------------------------------");
175                 $write("------------------------------------------\n");
176                 $display("@ %0d\t\t| exceptions\t\t| ERROR: Exceptions
                    failed.",
177                 $time/2);
178                 $write("------------------------------------------");
179                 $write("------------------------------------------\n");
180             end
181             if (v4) begin
182                 $display("@ %0d\t\t| ready\t\t\t| %b", $time/2, ready);
183                 $write("------------------------------------------");
184                 $write("------------------------------------------\n");
185             end
186             if ((v3|v4)) begin
187                 if (format == `FP64) begin
188                     $display("A0\t\t| %b", testmem[4*i_total + 0]);
189                     $display("B0\t\t| %b", testmem[4*i_total + 1]);
190                     $display("A1\t\t| %b", testmem[4*i_total + 2]);
191                     $display("B1\t\t| %b", testmem[4*i_total + 3]);
192                     $write("------------------------------------------");
193                     $write("------------------------------------------\n");
194                 end
195                 else begin
196                     $display("A0\t\t| %b", testmem[8*i_total + 0]);
197                     $display("B0\t\t| %b", testmem[8*i_total + 1]);
198                     $display("A1\t\t| %b", testmem[8*i_total + 2]);
199                     $display("B1\t\t| %b", testmem[8*i_total + 3]);
200                     $display("C0\t\t| %b", testmem[8*i_total + 4]);
201                     $display("D0\t\t| %b", testmem[8*i_total + 5]);
202                     $display("C1\t\t| %b", testmem[8*i_total + 6]);
203                     $display("D1\t\t| %b", testmem[8*i_total + 7]);
204                     $write("------------------------------------------");
205                     $write("------------------------------------------\n");
206                 end
207             end
208
209             $display("DUT [127:64]\t|Â %b", products[127:64]);
210             $display("DUT [ 63:0 ]\t|Â %b", products[63:0]);
211             $write("------------------------------------------");
212             $write("------------------------------------------\n");
```

```
213         $display("DW␣␣[127:64]\t|Â %b", dw_products[127:64]);
214         $display("DW␣␣[␣63:0␣]\t|Â %b", dw_products[63:0]);
215         $write("—————————————————————————————");
216         $write("—————————————————————————————\n");
217         $display("DUT␣[15:0]\t|Â %b␣%b␣%b␣%b␣(underflow␣overflow␣
                inexact␣invalid)",
218         exceptions[15:12], exceptions[11:8], exceptions[7:4],
                exceptions[3:0]);
219         $write("—————————————————————————————");
220         $write("—————————————————————————————\n");
221         $display("DW␣␣[15:0]\t|Â %b␣%b␣%b␣%b␣(underflow␣overflow␣
                inexact␣invalid)",
222         dw_exceptions[15:12], dw_exceptions[11:8], dw_exceptions
                [7:4], dw_exceptions[3:0]);
223         $write("—————————————————————————————");
224         $write("—————————————————————————————\n");
225      end
226   endtask
227   // ————————————————————————————————————————————————————
228   // Prints final report.
229   // ————————————————————————————————————————————————————
230
231   task print_report;
232      begin
233         $display("\n\n");
234         $write("****************************************");
235         $write("****************************************\n");
236         $display("\nFINAL␣REPORT\n");
237         print_format(format);
238         print_mode(mode);
239         $display("Input␣statistics");
240         $write("—————————————————————————————");
241         $write("—————————————————————————————\n");
242         $display("Total␣invalid␣inputs\t\t:␣%0d", i_nan);
243         $display("Total␣zero␣inputs\t\t:␣%0d", i_zero);
244         $display("Total␣infinity␣inputs\t\t:␣%0d", i_inf);
245         $display("Total␣infintiy␣times␣zero\t:␣%0d", i_inf_times_zero
                );
246         $display("Total␣invalid␣times␣any␣number\t:␣%0d",
                i_nan_times_any);
247         $write("—————————————————————————————");
248         $write("—————————————————————————————\n");
249         $display("Total␣input␣vectors\t\t:␣%0d", VECTORS);
250         $write("—————————————————————————————");
251         $write("—————————————————————————————\n");
252         $display("");
253         $display("Ouput␣statistics");
254         $write("—————————————————————————————");
255         $write("—————————————————————————————\n");
256         $display("Total␣overflowed␣products\t:␣%0d", i_ovf);
257         $display("Total␣underflowed␣products\t:␣%0d", i_unf);
258         $display("Total␣invalid␣products\t\t:␣%0d", i_inv);
259         $display("Total␣inexact␣products\t\t:␣%0d", i_inx);
260         $write("—————————————————————————————");
261         $write("—————————————————————————————\n");
262         if (format == `FP64) begin
263            // Times two because each product vector consists of
264            // two products
265            $display("Total␣products\t\t\t:␣%0d", 2*i_total);
266            //$display("Total products passed\t\t: %0d", 2*i_passed);
267            //$display("Total products failed\t\t: %0d", 2*i_failed);
268            //$write("—————————————————————————————");
```

```
269             //$write("—————————————————————————————————————————\n");
270             $display("Total product vectors\t\t: %0d", i_total);
271             $display("Total products vectors passed\t: %0d", i_passed)
                    ;
272             $display("Total products vectors failed\t: %0d", i_failed)
                    ;
273         end
274         else begin
275             // Times four because each product vector consists of
276             // four products
277             $display("Total products\t\t\t: %0d", 4*i_total);
278             //$display("Total products passed\t\t: %0d", 4*i_passed);
279             //$display("Total products failed\t\t: %0d", 4*i_failed);
280             //$write("—————————————————————————————————————————");
281             //$write("—————————————————————————————————————————\n");
282             $display("Total product vectors\t\t: %0d", i_total);
283             $display("Total products vectors passed\t: %0d", i_passed)
                    ;
284             $display("Total products vectors failed\t: %0d", i_failed)
                    ;
285         end
286         $write("—————————————————————————————————————————");
287         $write("—————————————————————————————————————————\n");
288         $display("\n\n");
289         if (i_failed > 0) begin
290             $display("Test finished without success!");
291         end
292         else begin
293             $display("Test finished successfully!");
294         end
295         $display("");
296         $write("***************************************");
297         $write("***************************************\n");
298     end
299 endtask
300
301
302 task print_header;
303     begin
304         $write("=========================================");
305         $write("=========================================\n");
306         $display("Time (cycle)\t| Signal\t\t|Â Event");
307         $write("=========================================");
308         $write("=========================================\n");
309     end
310 endtask
311
312
313 // ————————————————————————————————————————————————————————————
314 // Prints rounding mode.
315 // ————————————————————————————————————————————————————————————
316 task print_mode;
317     input [1:0] mode;
318     begin
319         case (mode)
320             `EVEN: begin
321                 $display("Rounding mode\t\t\t: Round-to-nearest even\n"
                        );
322             end
323             `PINF: begin
324                 $display("Rounding mode\t\t\t: Round-to-positive
                        infinity\n");
```

```
325                 end
326                 `NINF: begin
327                     $display("Rounding mode\t\t\t: Round-to-negative 
                            infinity\n");
328                 end
329                 `ZERO: begin
330                     $display("Rounding mode\t\t\t: Round-to zero\n");
331                 end
332             endcase
333         end
334     endtask
335 
336     // ----------------------------------------------------------------
337     // Prints data format tested.
338     // ----------------------------------------------------------------
339     task print_format;
340         input [5:0] data_format;
341         begin
342             if (data_format == `FP16) begin
343                 $display("Data format\t\t\t: 16-bit floating-point (FP16)"
                        );
344             end
345             else if (data_format == `FP32) begin
346                 $display("Data format\t\t\t: 32-bit floating-point (FP32)"
                        );
347             end
348             else if (data_format == `FP64) begin
349                 $display("Data format\t\t\t: 64-bit floating-point (FP64)"
                        );
350             end
351         end
352     endtask
```

D.3 Switching Activity Simulation Source

```
1 // ----------------------------------------------------------------------
2 // File.......: vec_fp_mult_stimuli_tb.v
3 // Author.....: Espen Stenersen
4 // Date.......: Tue Apr 29 14:19:15 CEST 2008
5 // Revision...: 1.0
6 // Description: Generates switching activity information to the
7 //              synopsys power analasys tools.
8 // ----------------------------------------------------------------------
9
10
11 `include "defines.v"
12
13 `timescale 1ps/1ps
14 `define CLK_PERIOD 5000
15
16
17 module vec_fp_mult_stimuli_tb;
18
19    parameter          FP16STEP = 4;
20    parameter          FP32STEP = 4;
21    parameter          FP64STEP = 2;
22    parameter          FP16VECTORS = 100;
23    parameter          FP32VECTORS = 0;
24    parameter          FP64VECTORS = 0;
25
26    // wire(s)
27    wire [127:0]       products;
28    wire [15:0]        exceptions;
29    wire               ready;
30
31    // reg(s)
32    reg                start;
33    reg [127:0]        vectors;
34    reg [1:0]          format;
35    reg [1:0]          mode;
36    reg [15:0]         clear;
37    reg                clk;
38    reg                reset_n;
39
40    reg [`FP16W-1:0]   fp16testmem [0:FP16VECTORS];
41    reg [`FP32W-1:0]   fp32testmem [0:FP32VECTORS];
42    reg [`FP64W-1:0]   fp64testmem [0:FP64VECTORS];
43
44    integer            i_vec;
45
46    // ------------------------------------------------------------------
47    // Module instantiation.
48    // ------------------------------------------------------------------
49    vec_fp_mult DUT
50    (
51       .start      (start),        // Input. Starts computation.
52       .vectors    (vectors),      // Input. FP vectors to be computed.
53       .format     (format),       // Input. Format of vectors.
54       .mode       (mode),         // Input. Rounding mode.
55       .clear      (clear),        // Input. Clears exceptions.
56       .products   (products),     // Output. Computed products.
57       .exceptions (exceptions),   // Output. Exceptions raised.
58       .ready      (ready),        // Output. Output vector ready.
59       .clk        (clk),
```

```
60        .reset_n      (reset_n)
61    );
62
63
64    initial begin
65        clk = 1; reset_n = 0; start = 0; vectors = 0; clear = 0;
66        format = 0; mode = `EVEN;
67        `include "d1_tracefile.v"
68        $dumpfile("toggle1_200_fp16.vcd");
69        $readmemb("fp16testcases.txt", fp16testmem);
70        $readmemb("fp32testcases.txt", fp32testmem);
71        $readmemb("fp64testcases.txt", fp64testmem);
72
73        @ (posedge clk) // wait cycle.
74        @ (posedge clk) // wait cycle.
75        @ (posedge clk) reset_n = 1;
76        @ (posedge clk) // wait cycle.
77        @ (posedge clk) // wait cycle.
78
79        // Round-to-nearest even.
80        for (i_vec = 0; i_vec < FP16VECTORS; i_vec = i_vec + FP16STEP)
              begin
81           @ (posedge clk) begin
82              format = `FP16;
83              start = 1;
84              vectors[1*`FP16W-1:0*`FP16W] <= fp16testmem[i_vec + 0];
85              vectors[2*`FP16W-1:1*`FP16W] <= fp16testmem[i_vec + 1];
86              vectors[3*`FP16W-1:2*`FP16W] <= fp16testmem[i_vec + 2];
87              vectors[4*`FP16W-1:3*`FP16W] <= fp16testmem[i_vec + 3];
88           end
89        end
90        for (i_vec = 0; i_vec < FP32VECTORS; i_vec = i_vec + FP32STEP)
              begin
91           @ (posedge clk) begin
92              format = `FP32;
93              start = 1;
94              vectors[1*`FP32W-1:0*`FP32W] <= fp32testmem[i_vec + 0];
95              vectors[2*`FP32W-1:1*`FP32W] <= fp32testmem[i_vec + 1];
96              vectors[3*`FP32W-1:2*`FP32W] <= fp32testmem[i_vec + 2];
97              vectors[4*`FP32W-1:3*`FP32W] <= fp32testmem[i_vec + 3];
98           end
99        end
100       for (i_vec = 0; i_vec < FP64VECTORS; i_vec = i_vec + FP64STEP)
              begin
101          @ (posedge clk) begin
102             format = `FP64;
103             start = 1;
104             vectors[1*`FP64W-1:0*`FP64W] <= fp64testmem[i_vec + 0];
105             vectors[2*`FP64W-1:1*`FP64W] <= fp64testmem[i_vec + 1];
106          end
107       end
108       // Round-to-positive infinity.
109       mode = `PINF;
110       for (i_vec = FP16VECTORS; i_vec < 2*FP16VECTORS; i_vec = i_vec +
               FP16STEP) begin
111          @ (posedge clk) begin
112             format = `FP16;
113             start = 1;
114             vectors[1*`FP16W-1:0*`FP16W] <= fp16testmem[i_vec + 0];
115             vectors[2*`FP16W-1:1*`FP16W] <= fp16testmem[i_vec + 1];
116             vectors[3*`FP16W-1:2*`FP16W] <= fp16testmem[i_vec + 2];
117             vectors[4*`FP16W-1:3*`FP16W] <= fp16testmem[i_vec + 3];
```

```
118             end
119         end
120         for (i_vec = FP32VECTORS; i_vec < 2*FP32VECTORS; i_vec = i_vec +
                FP32STEP) begin
121             @ (posedge clk) begin
122                 format = `FP32;
123                 start = 1;
124                 vectors[1*`FP32W-1:0*`FP32W] <= fp32testmem[i_vec + 0];
125                 vectors[2*`FP32W-1:1*`FP32W] <= fp32testmem[i_vec + 1];
126                 vectors[3*`FP32W-1:2*`FP32W] <= fp32testmem[i_vec + 2];
127                 vectors[4*`FP32W-1:3*`FP32W] <= fp32testmem[i_vec + 3];
128             end
129         end
130         for (i_vec = FP64VECTORS; i_vec < 2*FP64VECTORS; i_vec = i_vec +
                FP64STEP) begin
131             @ (posedge clk) begin
132                 format = `FP64;
133                 start = 1;
134                 vectors[1*`FP64W-1:0*`FP64W] <= fp64testmem[i_vec + 0];
135                 vectors[2*`FP64W-1:1*`FP64W] <= fp64testmem[i_vec + 1];
136             end
137         end
138         // Round-to-negative inifity.
139         mode = `NINF;
140         for (i_vec = 2*FP16VECTORS; i_vec < 3*FP16VECTORS; i_vec = i_vec
                + FP16STEP) begin
141             @ (posedge clk) begin
142                 format = `FP16;
143                 start = 1;
144                 vectors[1*`FP16W-1:0*`FP16W] <= fp16testmem[i_vec + 0];
145                 vectors[2*`FP16W-1:1*`FP16W] <= fp16testmem[i_vec + 1];
146                 vectors[3*`FP16W-1:2*`FP16W] <= fp16testmem[i_vec + 2];
147                 vectors[4*`FP16W-1:3*`FP16W] <= fp16testmem[i_vec + 3];
148             end
149         end
150         for (i_vec = 2*FP32VECTORS; i_vec < 3*FP32VECTORS; i_vec = i_vec
                + FP32STEP) begin
151             @ (posedge clk) begin
152                 format = `FP32;
153                 start = 1;
154                 vectors[1*`FP32W-1:0*`FP32W] <= fp32testmem[i_vec + 0];
155                 vectors[2*`FP32W-1:1*`FP32W] <= fp32testmem[i_vec + 1];
156                 vectors[3*`FP32W-1:2*`FP32W] <= fp32testmem[i_vec + 2];
157                 vectors[4*`FP32W-1:3*`FP32W] <= fp32testmem[i_vec + 3];
158             end
159         end
160         for (i_vec = 2*FP64VECTORS; i_vec < 3*FP64VECTORS; i_vec = i_vec
                + FP64STEP) begin
161             @ (posedge clk) begin
162                 format = `FP64;
163                 start = 1;
164                 vectors[1*`FP64W-1:0*`FP64W] <= fp64testmem[i_vec + 0];
165                 vectors[2*`FP64W-1:1*`FP64W] <= fp64testmem[i_vec + 1];
166             end
167         end
168         // Round-to zero.
169         mode = `ZERO;
170         for (i_vec = 3*FP16VECTORS; i_vec < 4*FP16VECTORS; i_vec = i_vec
                + FP16STEP) begin
171             @ (posedge clk) begin
172                 format = `FP16;
173                 start = 1;
```

```
174             vectors[1*`FP16W-1:0*`FP16W] <= fp16testmem[i_vec + 0];
175             vectors[2*`FP16W-1:1*`FP16W] <= fp16testmem[i_vec + 1];
176             vectors[3*`FP16W-1:2*`FP16W] <= fp16testmem[i_vec + 2];
177             vectors[4*`FP16W-1:3*`FP16W] <= fp16testmem[i_vec + 3];
178          end
179       end
180       for (i_vec = 3*FP32VECTORS; i_vec < 4*FP32VECTORS; i_vec = i_vec
             + FP32STEP) begin
181          @(posedge clk) begin
182             format = `FP32;
183             start = 1;
184             vectors[1*`FP32W-1:0*`FP32W] <= fp32testmem[i_vec + 0];
185             vectors[2*`FP32W-1:1*`FP32W] <= fp32testmem[i_vec + 1];
186             vectors[3*`FP32W-1:2*`FP32W] <= fp32testmem[i_vec + 2];
187             vectors[4*`FP32W-1:3*`FP32W] <= fp32testmem[i_vec + 3];
188          end
189       end
190       for (i_vec = 3*FP64VECTORS; i_vec < 4*FP64VECTORS; i_vec = i_vec
             + FP64STEP) begin
191          @(posedge clk) begin
192             format = `FP64;
193             start = 1;
194             vectors[1*`FP64W-1:0*`FP64W] <= fp64testmem[i_vec + 0];
195             vectors[2*`FP64W-1:1*`FP64W] <= fp64testmem[i_vec + 1];
196          end
197       end
198       // Empty pipeline.
199       @(posedge clk) // wait cycle.
200       @(posedge clk) // wait cycle.
201       start = 0;
202       @(posedge clk) // wait cycle.
203       @(posedge clk) // wait cycle.
204       @(posedge clk) // wait cycle.
205       $finish;
206    end
207
208    // Toggles clearing of exceptions.
209    always @(ready) begin
210       if (ready == 1) begin
211          clear = 16'b1111111111111111;
212       end
213       else begin
214          clear = 16'b0000000000000000;
215       end
216    end
217
218
219    // clock generator.
220    always #(`CLK_PERIOD/2) clk = !clk;
221
222 endmodule // vec_fp_mult_stimuli_tb
```