NTNU

Norwegian University of
Science and Technology

# Low-Cost FPU

Specification, Implementation and Verification

**Daniel Hornæs**

Master of Science in Electronics

Submission date: June 2010

Supervisor: Bjørn B. Larsen, IET

Co-supervisor: Einar Fredriksen, Atmel Norway

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

# Problem Description

Based on previous work, this project will continue the work on a low-cost floating-point unit, suitable for inclusion in an Atmel AVR 8-bit microcontroller.

The design aims to implement the minimum requirements of the IEEE-754 1985 standard for floating-point arithmetic, using simple algorithms with similar functional requirements.
An important goal for the project is to provide a design that may offer the convenience of floating-point computations to the microcontroller domain, without a huge impact on hardware consumption or the slow execution speed of a software implementation.

Implementation and specification is prioritized, but verification through simulation should be performed, in order to demonstrate the correctness of the final implementation.

Assignment given: 18. January 2010
Supervisor: Bjørn B. Larsen, IET

# Abstract

This report aims to provide a complete specification of an IEEE-754 1985 compliant design, as well as a working, synthesizable implementation in Verilog HDL. The report is based on a preliminary project, which analyzed the IEEE-754 standard and suggested a set of algorithms suitable for a compact realization.

Through traditional methods of both algorithmic analysis and dataflow analysis, requirements of functional units are derived, and operations are scheduled.

A set of functional simulations assert the correctness of the design, while area and performance analysis provides information on the speedup gained, versus the hardware cost.

Finally, the results obtained are compared to existing implementations, in both hardware and software.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

As microcontrollers are assigned more and more complex tasks, the requirement of low-end, yet efficient floating-point computations becomes relevant. Even though most computations involving fractions can be performed using traditional fixed-point math, floating-point math is convenient for programmers, especially as the size of software projects increases.

Most implementations of the IEEE-754 1985 standard for binary floating-point arithmetic (or simply *the standard*) focus on high performance, targeting complex scientific calculations, or heavy multimedia processing. Low-end platforms with floating-point support mostly relies on pure software-implementations, which typically provide rather poor performance.

In an attempt to bridge this gap, this project aims to derive a compact hardware implementation of the standard, that achieves a large enough speedup over a software implementation to justify the additional hardware cost. The target domain is 8-bit microcontrollers, more specifically the Atmel AVR 8-bit architecture [3].

This project is based on a preliminary project [5].

## 1.2 What was Covered in the Preliminary Project

The preliminary project aimed to extract the requirements of a compliant implementation from the IEEE-754 1985 standard for binary floating-point arithmetic. This involved identifying the representation format, along with the required operations and exceptional cases covered in the standard. Then, a variety of algorithms

capable of performing these operations were discussed and compared. Finally, a set of algorithms was chosen, based on their functional equivalence and the absence of complex internal operations. The motivation behind this was to allow maximum sharing of resources, as well as keeping the functional units as simple as possible. Based on these algorithms, an architecture was suggested, along with some rough timing estimates.

## 1.3 What will be Covered in this Report

This project builds upon the preliminary project, aiming to provide a complete implementation of the architecture suggested in the previous report, along with area and execution time estimates. This task includes detailed specification, along with solving some problems not covered by the previous work. A few notable examples is the performing of rounding in accordance with the standard, as well as detecting and dealing with exceptions.

As the time-frame for the project is limited, some tasks must be prioritized. As the goal of the project is to acquire data on area consumption and execution time of an IEEE-754 1985 implementation, the features that contribute to area consumption and execution time will be dealt with first. This involves any operation that will require dedicated functionality, or additional control steps. Operations such as overflow detection are less likely to introduce any significant hardware consumption, thus they are assigned a lower priority.

## 1.4 What will not be Covered in this Report

This report assumes that the reader is familiar with IEEE single-precision floating-point numbers, as well as the basic mathematics behind the associated operations. For more information on this, please refer to the preliminary project report [5], the standard itself [4] or other resources [11], [6].

As the purpose of this project is to create a minimal, low-cost implementation, only features that are required by the standard are considered. Thus, additional features such as conversion from character strings to floating-point values, and comparison instructions will not be discussed.

In addition, certain features are omitted due to their complex nature. Especially support for denormal numbers is fairly complex to implement, thus it is not featured in this report even though it is required by the standard. As the method for handling denormal numbers is very much dependent on the surrounding architecture, no choices have been made for this. One possible method is to flush all denormal inputs and outputs to zero.

No mechanism for software traps are discussed, as this is beyond the scope of this

project. However, this may be used to handle denormal numbers in software, thus any subsequent work may want to consider supporting traps. The floating-point remainder operation is not featured here either.

Finally, the treatment of *Not-a-Number* values (*NaNs*) is very brief in this report. For instance, *quiet NaNs* and *signaling NaNs* are not treated individually in any way.

## 1.5 Structure of this Report

Chapter 2 will give a brief overview of some of the theory that was left out in the preceding report, or simply wasn't discussed in sufficient detail. Note that this is not a comprehensive walk-through of all the theory behind this project, please refer to [5] for more details.

Following the theory chapter, the design and specification of the various modules will be discussed in Ch.3. This chapter is the bulk of this report, covering both algorithmic as well as architectural design considerations. The various functional units that will form the building blocks of the complete design is discussed separately, before they are assembled to form the final architecture. In order to make this chapter tidy and comprehensible, the design of each operation is discussed separately. Finally, they are merged together into the final design.

Chapter 4 on simulation and verification will discuss some means to verify the behavior of the design, as well as illustrate some important concepts through simulation.

Then, Ch. 5 will present the synthesis results, as well as the resulting clock cycle usage of the various operations.

Finally, the results from the preceding sections will be discussed in Ch.6, and the report will be concluded. Suggestions for future work is included here as well.

# Chapter 2

# Theory

This chapter will introduce some theory that was insufficiently covered in the preliminary project, or not covered at all.

## 2.1 IEEE Floating-Point Numbers

This report deals with floating-point numbers, that correspond to the representation presented in the IEEE-754 1986 Standard for Binary Floating-Point Arithmetics [4]. To limit this report, only single precision, normalized numbers are discussed.

As mentioned, this report assumes that the reader has some knowledge of the standard as well as the various floating-point operations. Several resources that provide information on this was listed in the introduction.

This report will refer to the various components of a floating-point value as *sign bit* ($s$), *exponent* ($E$) and *significand, fraction or mantissa* ($F$).

## 2.2 Count Leading Zeros

The *count leading zeros* operation is included as a CPU instruction in several architectures. It takes a binary number, and returns the number of leading zeros, counting from the MSB towards the LSB. If the input value consists of $N$ bits, the answer can be represented in $\log 2(N)$ bits, given that $N$ is a power of two.

For instance, $CLZ(00001010)$ returns the value $100_{two} = 4_{ten}$.

This section describes a way to implement the CLZ operation, for input values with

a power-of-two bit width.

First, extract the upper half of the input word, and compare it to zero. If this comparison is true, it means that the leading one is located in the lower half of the input word. Thus, the number of leading zeros is at least half of the input word length. The upper half of the input word is discarded, and the same procedure is applied to the lower half. The upper bit of the result word is set to one.

If the comparison was false, the leading one must be located in the upper half of the input word. This means that the number of leading zeros must be less than half the word length of the input word. The upper bit in the result is set to zero, and the procedure is applied to the upper half of the input word.

This subdivision will generate a binary tree with a depth of $log2(N)$, generating one result bit per subdivision starting from the MSB. It should be noted that the case of an all-zero input value requires special care. This can be implemented as a relatively simple comparison of the result generated, which introduces very little extra logic.

This algorithm is represented as a flowchart in Fig.2.1.

## 2.3 Rounding

This section will elaborate the theory behind the various forms of rounding included in the IEEE-754 1985. The different rounding modes will be reviewed shortly, and methods for performing the actual rounding will be discussed. Note that this section does not discuss the actual implementation of the rounding schemes; this is done in ch.3.9.

For more details on the mathematics behind rounding, see [11].

### 2.3.1 Prerequisites for rounding

As concluded in [5], some additional information is required in order to round a fraction in accordance with the specification. More specifically, this is two additional bits of precision — *guard* and *round* — in the internal representation, as well as a *sticky bit*. The former bits are fairly easy to implement, as long as the datapath is wide enough to support it.

The *sticky bit* represents what could be in the bits to the right of the least significand bits, had they not been discarded. If a *one* is ever shifted into the sticky bit, it "sticks" to *one*, and remains high for the rest of the operation. The generation of the *sticky bit* will require some additional logic, this is discussed in Ch.3.9.

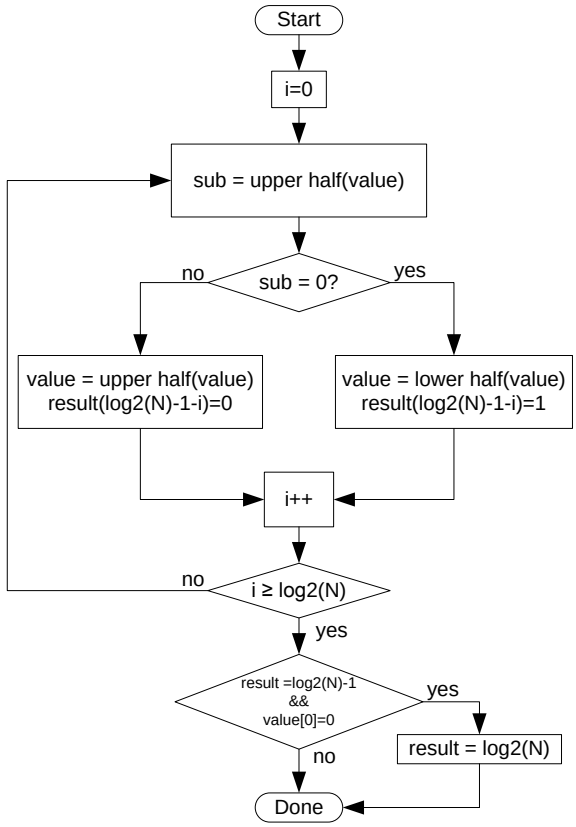


Figure 2.1: Flowchart Representation of the Count-Leading-Zeros Algorithm

Mathematically speaking, the rounding can be described as

$$frac_{rounded} = round(frac, s)$$

where frac is on the form

$$1.\underbrace{XXXXXXXXXXXXXXXXXXXXXXX}_{23\ bit}\ g\ r$$

and the resulting $frac_{rounded}$ is on the form

$$1.\underbrace{XXXXXXXXXXXXXXXXXXXXXXX}_{23\ bit}$$

The 23 fractional bits denoted $X$ in the latter expression are the bits that will be kept in the final representation of the result, leading *one* excluded.

### 2.3.2 Round Towards Zero (Truncate)

This rounding mode is by far the simplest to implement, as it only requires a truncation of the significand. An implementation that only supported this rounding mode could be made more compact and have a shorter execution time of operations that require rounding. Example:

$$round(1.11000000000000000000000\ 1\ 1\ 0) = 1.11000000000000000000000$$

### 2.3.3 Round Towards $+\infty$

In this mode, the value of the fraction will be rounded towards $+\infty$, regardless of its value. In the case of negative numbers, this is identical to the round-towards-zero mode; just truncate the number. Example:

$$round(-1.11000000000000000000000\ 1\ 1\ 0) = 1.11000000000000000000000$$

In the case of a positive significand, the value must be increased if any of the additional bits are set. This can be solved by performing a logical OR between the *guard*, *round* and *sticky* bits. The resulting bit can be used to determine the rounding: if it is 1, one LSB must be added to the significand. If it is 0, the fraction can be truncated.

Examples:

$$round(1.11000000000000000000000\ 1\ 0\ 0) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 1\ 0) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 0\ 1) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 0\ 0) = 1.11000000000000000000000$$

### 2.3.4 Round Towards $-\infty$

The round-towards-minus-infinity rounding mode will, as the name implies, round the fraction towards minus $\infty$. This is fairly similar to the previous rounding mode, except that we switch the treatment of the positive and negative fractions: if the fraction is positive, it is truncated. If it is negative, the extra bits will determine its value. Example:

$$round(1.11000000000000000000000\ 1\ 1\ 0) = 1.11000000000000000000000$$

Again, if any of the extra bits are set, a LSB must be added to the absolute value of the fraction. Note that the sign bit is kept out of the addition itself; the purpose

of this addition is to bring the absolute fraction value away from zero. Examples:

$$round(1.11000000000000000000000\ 1\ 0\ 0) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 1\ 0) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 0\ 1) = 1.11000000000000000000001$$
$$round(1.11000000000000000000000\ 0\ 0\ 0) = 1.11000000000000000000000$$

### 2.3.5 Round To Nearest Even

This rounding mode is by far the most complicated to implement, however it is also the one specified as default by the standard. The principle is to round the fraction to the value that is closest to the unrounded value. For instance,

$$round(10.11) = 11.00$$

, as the fraction is closer to 11 than it is to 10. The problem with the algorithm is the treatment of the half-way case, where the numeric distance to the two alternatives is equal. For instance, 10.10 is half-way between 10.00 and 11.00. In this case, the standard requires the number to be rounded to the value that has a zero in its LSB. In the previous example, the correct value after rounding would be 10.00.

Thus, the implementation of this rounding mode will require knowledge of the least significant bit of the significand, namely the bit to the left of the *guard* and *round* bits.

If we compute an intermediate *sticky-bit* $s' = g|s$, the effective rounding operations required for the *round-to-nearest-even* scheme can be summarized as in Tab.2.1 (adapted from [11]). *ulp* denotes *unit in the last place*, and is simply a high bit

Table 2.1: Round-to-Nearest-Even: Required Rounding Operations

| LSB | r | s' | Operation |
|---|---|---|---|
| 0 | 0 | 0 | $+0$ |
| 0 | 0 | 1 | $+0$ |
| 0 | 1 | 0 | $+0$ |
| 0 | 1 | 1 | $+0.5ulp$ |
| 1 | 0 | 0 | $+0$ |
| 1 | 0 | 1 | $+0$ |
| 1 | 1 | 0 | $+0.5ulp$ |
| 1 | 1 | 1 | $+0.5ulp$ |

with the same weight as the least significant bit in the operand. For instance, $1.0 + 0.5ulp$ is performed as:

$$1.00000000000000000000000\ g\ r$$
$$+0.00000000000000000000000\ 1\ 0$$

### 2.3.6 Floating-Point Exceptions

This section will give a brief review of the various exceptions included in the standard, as well as highlighting some important implementation aspects. Note that the implementation of the exception handling is discussed in Ch.3.10, in context of the actual design organization.

#### Exceptions Present in the IEEE 754-1985 Standard

The standard defines the following exceptions:

1. Invalid operation
2. Division by Zero
3. Inexact
4. Overflow
5. Underflow

The exceptions were defined in the preliminary report, along with examples of cases that will trigger them. The important thing to note here, is that these exceptions can be divided into two classes: The first is exceptions that can be detected and dealt with upon the very beginning of an operation, the second is the kind of exceptions that occur at some point during the execution of the operation.

The former class of exceptions will be referred to as *init-time exceptions* in this report, the latter will be referred to as *run-time exceptions*.

This difference affects how exceptions will be detected and dealt with in the implementation, thus it is necessary to identify which exceptions belong to which group. This is listed in Tab.2.2.

Table 2.2: IEEE 754-1985 Exceptions

| Exception | Detectable at init-time? |
|---|---|
| Invalid operation | Yes |
| Division by Zero | Yes |
| Inexact | No |
| Overflow | No |
| Underflow | No |

In accordance with this classification, the implementation will treat *invalid operation* and *division by zero* at init-time, while the other exceptions will be detected within the arithmetic stages ("run-time"). The causes for these exceptions will be derived per operation in the later chapters, before they are summarized in Ch.3.10.

# Chapter 3

# Design and Specification

This chapter will deal with the design and specification of various parts of the system. The chapter is divided into separate sections for the various operations, as well as separate sections for rounding and exception handling. The reason behind this organization is to manage complexity, as well as making it easier to extract a single operation from the design, and implement it by itself. The design is loosely based on design principles found in [13] and [13].

## 3.1 General Considerations

The general design was derived in the preliminary project, an abstract overview of the architecture is given in Fig.3.1.

The architecture can be summarized as two distinct scalar pipelines, sharing a common control unit. In addition to this, an external multiplier is connected to the significand pipeline. Certain operations require some transfer of data between the two pipelines, hence they are interconnected by a few data wires.

The design will implement the floating-point operations required by the IEEE-754 1985, by a careful selection of algorithms. As the design is similar to a general-purpose CPU pipeline, it is obvious that the chosen algorithms will share characteristics with existing software implementations if the IEEE-754. The speedup over a software implementation is mainly achieved through a more suitable data width, as well as utilizing two pipelines along with some hardwired routing of data.

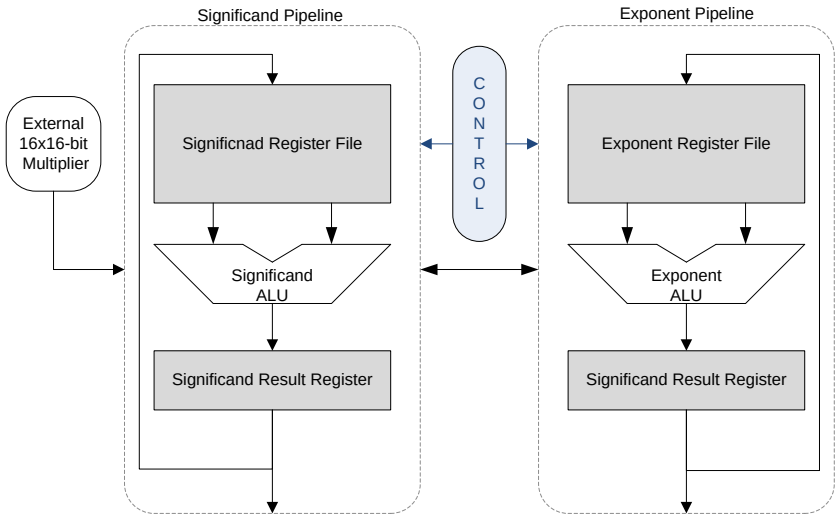


Figure 3.1: Abstract View of the System Architecture

### 3.1.1 System-level interface

Most of the operations specified in the standard take two inputs and produce one output. The rest are unary, thus consuming one operand and produce one result. The system-level interface of the FPU includes two input data ports, and one output data port. Additional input signals are the instruction opcode, as well as the active rounding mode. All input operands are assumed to be 32-bit floating-point values, with the exception of the integer-to-floating-point conversion operands. These will be interpreted as 32 bit integer values, either unsigned or signed two's compliment, depending on the instruction. Integer values smaller than 32 bit are often used in microcontrollers, but they can simply be sign extended in order to correspond with the format assumed by the FPU.

The output ports consist of a 32-bit data result, several status flags and a flag that indicated that the current instruction is completed. The interface signals of the FPU is listed in Tab.3.1.

The system-level interface of the FPU is illustrated in fig.3.2.

## 3.2 Functional Units

This section will describe the functional units that form the basic blocks of the FPU architecture. The requirements and specifications of the various units are

Table 3.1: Floating-Point Unit - Interface Signals

| Signal | Name |
|---|---|
| Operand A | Input A |
| Operand B | Input B |
| OpCode | Specifies the active operation |
| RoundingMode | Decides the active rounding mode |
| Result | The FPU result output |
| ResultReady | Indicates that a result is ready, and the unit is ready for a new instruction |
| Invalid operation | Indicates that an invalid operation was performed |
| Division by zero | Indicates that a division by zero occured |
| Oveflow | Indicates that an overflow occured |
| Underflow | Indicates that an underflow occured |
| Inexact | Indicates that precision was lost during the operation |

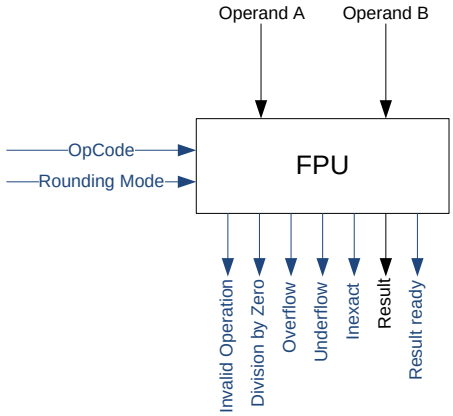


Figure 3.2: The System-Level Interface of the FPU

governed by the choice of algorithms and architecture in general. Thus, several aspects of the functional units presented here will be elaborated in the subsequent sections, which deal with each floating-point operation separately.

### 3.2.1 Storage Elements

This section will describe the internal storage elements of the floating-point unit. The organization of the internal registers is determined by the amount of storage

required by the chosen algorithms, as well as critical path considerations.

At minimum, the design needs three registers per pipeline. More specifically, each ALU needs two registers to provide the input values, and a result register. Note that the result in many cases can be stored by overwriting one of the input operands. In other words, a dedicated result register is only needed in a few operations.

However, the result registers can be utilized to reduce the critical path of the design, as well as serve as output buffers. Thus, we chose to implement them separately instead of incorporating them in a register file along with the input registers.

In addition to the general-purpose registers (*GPRs*), a set of constant-valued registers is also required. Examples of such are a register to hold the bias value specified by the standard, as well as constants used in normalization of results. In order to keep the design tidy, the constant registers are contained in a register file, along with two general-purpose registers. Note that the constant registers are not user-writable, as opposed to the GPRs. As the constants used in the design usually consist of mostly zeros, the constant values can be generated by relatively simple combinatorial logic.

### Significand-Related Registers

Figure 3.3 shows the interface of the register file that is connected to the significand ALU. The two write ports are connected to GPR R0 and R1, the *read select* signals choose which internal register value to forward to the corresponding output port. The *shift enable* signal enables left-shifting of register R0, by one digit.

As the various algorithms featured in this design require a selection of specific constants, a set of constant registers have been included in the register files. Table 3.2 lists the constants featured in the significand result register.

Table 3.2: Significand Register File - Constants

| Name | Value | Description |
|---|---|---|
| Zero | 32'd0 | All zeros |
| One | 32'd1 | 1 |
| Two | 32'd2 | 2 |
| ULP Round | 32'd128 | ULP used during rounding |
| Bias | 32'd127 | Exponent bias value |
| Five | 32'd5 | 5 |
| Six | 32'd6 | 6 |
| NaN Sig. | 32'h20000000 | Significand corresponding to a NaN result |
| Ones | 32'hFFFFFFFF | All ones |

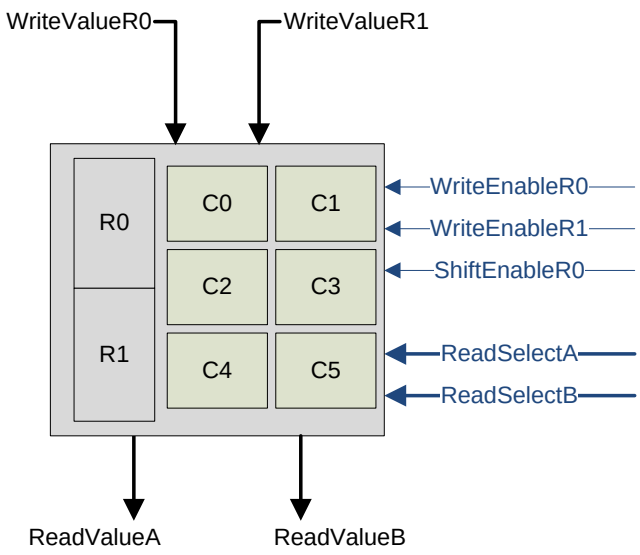


Figure 3.3: Significand Register File Interface

#### Exponent-Related Registers

The interface of the exponent pipeline register file is given in fig.3.4. Note that it is slightly simpler than its significand counterpart, as the shift-capabilities are not required here.

Again, a set of constant values are required in the various algorithms. The constant register values included in the exponent register file is given in tab.3.3.

Table 3.3: Exponent Register File - Constants

| Name | Value | Description |
|---|---|---|
| Zero | 0 | All zeros |
| One | 9'd1 | 1 |
| RPP | 9'd31 | Significand radix point position |
| I2FP | 9'd158 | Used in int->float conversion |
| Bias | 9'd127 | Exponent bias |
| Ones | 9'd511 | All ones |

### 3.2.2 Detection of Special Representation Values

The standard [4] defines several special representation values for floating-point numbers, which have a great impact on the implementation. Detecting and iden-

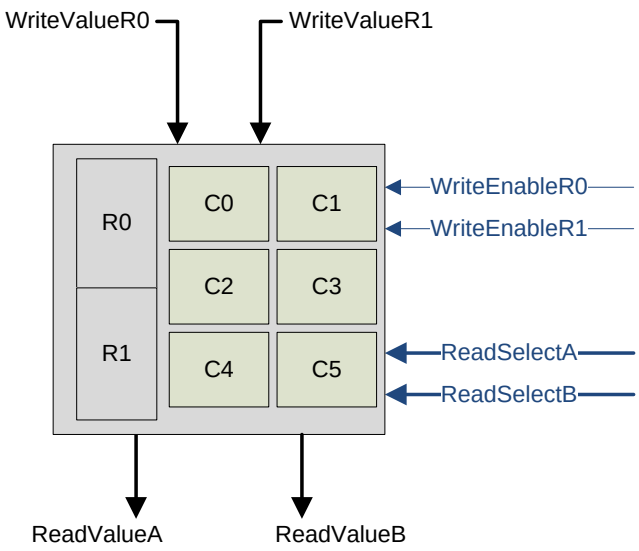


Figure 3.4: Exponent Register File Interface

tifying these values are an important part of floating-point exception handling. In addition it is possible to increase performance by treating certain special cases, such as multiplication between zero and a number.

The circuitry needed for detecting special representation values were derived in the preliminary project, and repeated here for convenience. Implementation-wise, the operation will consist of two combinatorial gate-networks connected to the input ports of the FPU itself. The logic will assert a set of status flags, depending on the value of the inputs. These status flags will be forwarded to the control unit, and used to determine the subsequent control flow.

Figure 3.5 shows the interface of the detection logic, Tab.3.4 specifies the interface signals. For more information on the internals of this unit, please refer to the preliminary project and the actual implementation source code.

### 3.2.3 Interface to the External Multiplier

As the floating-point unit will require multiplication of larger bit-widths than the existing hardware multiplier supports, it is necessary to split the multiplication into several smaller multiplication, and accumulate them. This section will describe the logic required to feed the multiplier with data, invoke a multiplication and finally align the partial product, in order to prepare it for the accumulation step.

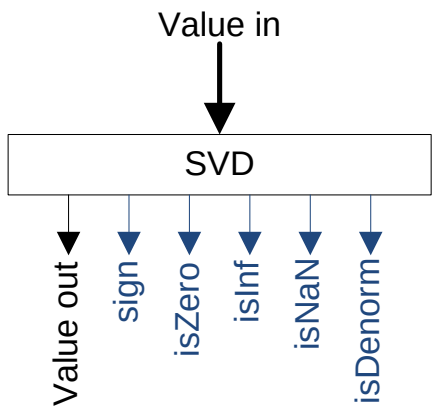


Figure 3.5: SVD Unit Interface

Table 3.4: Special Value Detection Unit - Interface

| Signal | Bit Width | Description |
|---|---|---|
| value in | 32 | Single precision floating-point value |
| sign | 1 | The sign of the input |
| isZero | 1 | input equal to $\pm 0$? |
| isInf | 1 | input equal to $\pm\infty$? |
| isNan | 1 | input is Not-a-Number? |
| isDenorm | 1 | Input is a denormal value? |
| value out | 33 | Input with leading significand digit appended |

### Slicing the Input

The *mask and shift* unit takes two 24-bit operands, and returns two 16-bit operands which can be sent to the multiplier input ports. Exactly how the slicing is done is determined by an opcode.

Note that extracting the higher bits of a word, and outputting them on the lower bits of the output ports, the numeric value of the operands are changed. This must be compensated for, after the multiplication is performed. This task is handled by the *shift and extend* unit.

The different operations of the unit is summarized in Tab.3.5

### The External Multiplier

The external multiplier is not a part of this project, however a behavioral model is included for simulation purposes. It is simply a pipelined multiplier that consumes

Table 3.5: *Shift-and-Mask Unit*

| Operation | OpCode | Description |
|---|---|---|
| A8C8 | 00 | Extracts the upper 8 bits from both operands |
| A8D16 | 01 | Extracts the upper 8 bits from the first operand and the 16 lower bits from the second operand |
| B16C8 | 10 | Extracts the 16 lower bits of the first operand and the 8 lower bits from the second operand |
| B16D16 | 11 | Extracts the lower 16 bits from both operands |

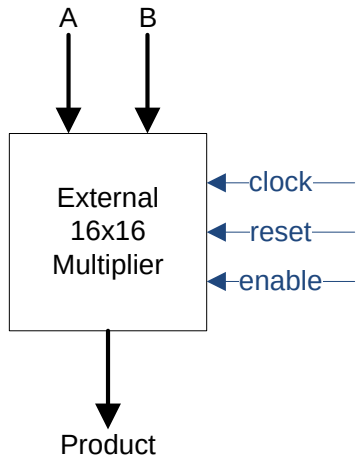


Figure 3.6: External Multiplier Interface

two cycles computing a 32-bit product from two 16-bit inputs. Consecutive multiplications can be started in consecutive cycles, allowing the multiplier to calculate $N$ multiplications in $N + 1$ cycles.

The interface of the external multiplier is illustrated in Fig.3.6.

Note that the area contribution of the external multiplier should be subtracted from the synthesis results, as it is not a part of this design.

### Extracting, Shifting and Extending the Partial Products

The *shift and extend* unit is responsible for converting a partial product from the external multiplier into a representation suitable for accumulation. This involves

shifting the partial product into the right position, and zero-extending the value. The shift amount depends on the operation of the *mask and shift* unit. Note that since the multiplier consumes two cycles per product, the operation of this unit must lag one cycle behind the one of the *mask and shift* unit.

For instance, if the former unit extracts the upper half of the 24-bit significands, and right-shifts the resulting bits by 12, the partial product needs to be left-shifted by 24 in order to obtain the correct numeric value. However, as we discard the lower 16 bits of the significand product, the actual operation of the *shift and extend* unit needs to be a 8-bit left-shift. See Ch.3.3 for more details on this.

The different operations of the unit is summarized in tab.3.6

Table 3.6: *Shift-and-Extend Unit*

| Operation | OpCode | Description |
|---|---|---|
| SHIFT_16_BIT_AND_EXTEND | 00 | Shifts the input 16 bits to the left |
| SHIFT_0_BIT_AND_EXTEND | 01 | Shifts the input 8 bits to the left, and zero extends the result |
| SHIFT_TRUNC_AND_EXTEND | 11 | Truncates the lower 16 bits of the input, and zero extends the result |

### 3.2.4 Arithmetic-Logic Units

The arithmetic-logic units are responsible for the bulk of the operations performed on data within the floating-point unit. Thus they need to be flexible and generic, while maintaining a low level of complexity in order to keep the area consumption as low as possible. As mentioned previously, the design revolves around two ALUs; one for the significand calculations and one for the exponent calculations.

The motivation behind this choice is that several of the micro-operations in the chosen algorithms can be performed independent on the significand and the exponent. Thus it is possible to exploit a certain amount of parallelism with very little effort. All the operations this design will implement could have been performed by a single ALU, indeed this is how it is done in most software implementations of the standard. Still, the addition of a second pipeline will provide a significant speedup at a low cost.

#### Significand ALU

This is the largest of the two ALUs, and also the one with the largest amount of operations. Hence, it will be a major factor in determining the total system cost.

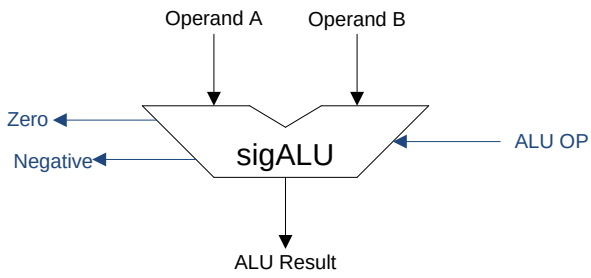


Figure 3.7: Significand ALU Interface

The interface of the significand ALU is shown in Fig.3.7, the interface signals are specified in Tab.3.7.

Table 3.7: Significand ALU Interface Signals

| Signal name | Bit Width | Description |
|---|---|---|
| Operand A | 32 | ALU Input A |
| Operand B | 32 | ALU Input B |
| ALU Result | 32 | The result of the current operation |
| ALU OP | 4 | The current operation |
| Zero | 1 | 1 if the result was zero, 0 otherwise |
| Negative | 1 | 1 if the result was negative, 0 otherwise |

The operations that are included in the significand ALU are summarized in Tab.3.8.

### Exponent ALU

The interface of the exponent ALU is shown in Fig.3.8, the interface signals are specified in Tab.3.9.

The operations that are included in the exponent ALU are summarized in Tab.3.10.

Table 3.8: Significand ALU Operations

| Operation | OpCode | Operation | Comment |
|---|---|---|---|
| SIG_ALU_OP_NOP | 0000 | Result ← 0 | No operation |
| SIG_ALU_OP_MOVA | 0001 | Result ← A | Moves A through the ALU |
| SIG_ALU_OP_NEGB | 0010 | Result ← -B | Negates B |
| SIG_ALU_OP_ADD | 0011 | Result ← A + B | Adds A and B |
| SIG_ALU_OP_SUB | 0100 | Result ← A - B | Subtracts B from A |
| SIG_ALU_OP_SHRA | 0101 | Result ← A »> B | Arithmetic right-shift of A by B bits |
| SIG_ALU_OP_SHRL | 0101 | Result ← A » B | Logical right-shift of A by B bits |
| SIG_ALU_OP_SHLL | 0110 | Result ← A « B | Logical left-shift of A by B bits |
| SIG_ALU_OP_CLZ | 1000 | CLZ(A) | Returns the number of leading zeroes in A, in the range [0,32] |

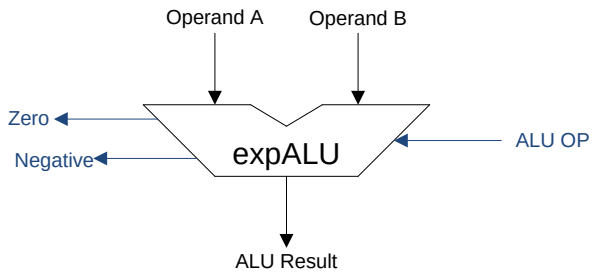


Figure 3.8: Exponent ALU Interface

Table 3.9: Exponent ALU Interface Signals

| Signal name | Bit Width | Description |
|---|---|---|
| Operand A | 32 | ALU Input A |
| Operand B | 32 | ALU Input B |
| ALU Result | 32 | The result of the current operation |
| ALU OP | 3 | The current operation |
| Zero | 1 | 1 if the result was zero, 0 otherwise |
| Negative | 1 | 1 if the result was negative, 0 otherwise |

Table 3.10: Exponent ALU Operations

| Operation | OpCode | Operation | Comment |
|---|---|---|---|
| EXP_ALU_OP_NOP | 000 | Result ← 0 | No operation |
| EXP_ALU_OP_MOVA | 001 | Result ← A | Moves A through the ALU |
| EXP_ALU_OP_NEGB | 010 | Result ← (-B) | Negates B |
| EXP_ALU_OP_ADD | 011 | Result ← A + B | Adds A and B |
| EXP_ALU_OP_SUB | 100 | Result ← A - B | Subtracts B from A |
| EXP_ALU_OP_SHL | 101 | Result ← A « B | Logical left-shift of A by B bits |

## 3.3 Floating-Point Multiplication

Multiplication differs from the other operations, as it is the only operation that is based on existing hardware; namely a 16x16bit integer multiplier. Thus it is the least flexible operation in terms of design space exploration and will be discussed before the others.

### 3.3.1 Algorithm and Design Considerations

The basic algorithm for floating-point multiplication was described in [5]. The algorithm can be summarized with the following steps:

1. Add the exponents
2. Subtract *bias* in order to obtain the correct exponent
3. Perform signed multiplication of the input significands
4. Normalize and round the result. This is easy, because of the constrained range of the multiplication result
5. Calculate the output sign bit as the logical XOR operation between the input exponent bits

The only complex operation in this algorithm is the significand multiplication, which will be performed by the existing 16x16-bit multiplicator, along with an accumulator.

As the input significands consist of 24-bit fixed-point numbers with a 1:23 bit distribution (integer:fraction), the complete multiplication of these values will yield a 48-bit result, with a 2:46 bit distribution. Thus the minimum required size of the accumulator and result register is 48 bit. This will result in a significant increase in bit width of several units, which will have a negative impact on the total area consumption. It is highly desirable to reduce this requirement, in order to find a compact solution.

As the IEEE-754 only requires a certain amount of precision, it is possible to discard the least significant bits of the product. By careful scheduling of the partial product multiplications, the required bit width of the accumulator and result register can be reduced to 32 bit. This is a 50% reduction compared the direct computation.

The required word slicing is illustrated in eq.3.1

$$\underbrace{AA}_{A_{high}}\underbrace{BBCC}_{A_{low}} \times \underbrace{DD}_{B_{high}}\underbrace{EEFF}_{B_{low}} \tag{3.1}$$

Eq.3.2 shows the order of operations.

$$\begin{array}{ll} & \underline{\quad AABBCC \times DDEEFF} \\ & AADD \\ + & \quad AAEEFF \\ + & \quad DDBBCC \\ + & \quad\quad BBCCEEFF \\ = & \overline{PPPPPPPPPPPP} \end{array} \tag{3.2}$$

Note that the bits that will be discarded from the final partial product can not be completely ignored; they may affect the rounding of the final result. Hence it is necessary to determine if any of the discarded bits were set high; in this case the *sticky bit* must be asserted. More details on this can be found in chapter 3.9, which deals with the implementation of rounding.

$$Multiplication\ result = \underbrace{PPPPPPPP}_{32\ MSB\ of\ product} \quad \underbrace{PPPP}_{Sticky\ bit\ data}$$

### 3.3.2 Organization

Figure 3.9 shows a suggested schedule for performing floating-point multiplication at a minimal hardware cost. Note that this architecture is affected by the algorithms chosen for all the floating-point operations, in the preliminary project. Thus, it is most likely not ideal if you consider the multiplication operation by itself.

Blue wires represent control signals, black wires represent data. The gray blocks are storage elements, the white rectangles represent functional units and combinatorial units. The registers were discussed in detail in ch.3.2.1. The two units labeled *sigALU* and *expALU* are the arithmetic-logic units that deals with significand and exponent computations, respectively. The blue control lines are unconnected in the figure; the control unit was left out of the figure to keep it more readable.

The exponent pipeline simply accepts two input exponents, add them together and subtract *bias* in order to obtain the correct numerical result. The significand ALU accepts partial products from the multiplier chain shown in the left part of the diagram. The significand ALU works together with the corresponding register file, functioning as an accumulator. This enables the significand multiplication, in accordance with the method presented in the previous section.

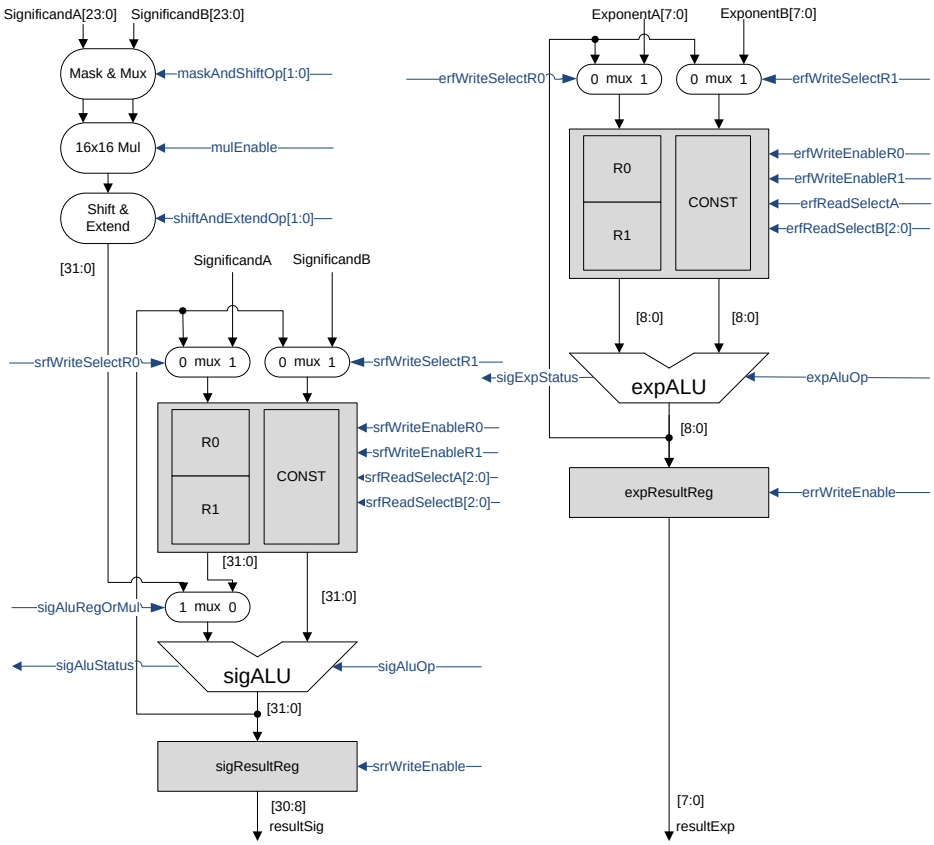


Figure 3.9: Floating-Point Multiplication Architecture

### 3.3.3 Scheduling and Control

#### Dataflow

The algorithm can be represented in a dataflow diagram (DFG) as shown in Fig.3.10

#### Functional Unit Binding

The functional unit binding is relatively simple: the multiplication is shared among the external multiplier and the significand ALU, while the exponent calculations are performed by the exponent ALU.

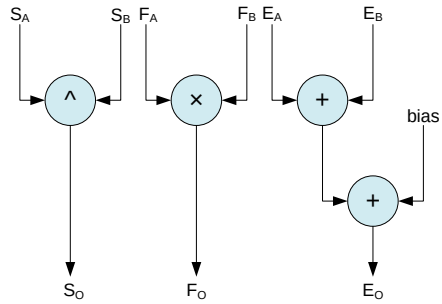


Figure 3.10: Floating-Point Multiplication DFG

### Scheduling

Table 3.11 shows the schedule of the floating-point multiplication, using the given architecture. Note how the calculations of the exponent and the significand can be performed independently. The sign bit calculation is not shown in this table, as it is included in the control unit itself.

Table 3.11: Floating-Point Multiplication - Schedule

| Cycle | Multiplier | Significand ALU | Exponent ALU |
|---|---|---|---|
| 1 | $P0$ | | $E_A + E_B$ |
| 2 | $P1/P0$ | | $(E_A + E_B) - bias$ |
| 3 | $P2/P1$ | $0 + P0$ | |
| 4 | $P3/P2$ | $P0 + P1$ | |
| 5 | $P3$ | $(P0 + P1) + P2$ | |
| 6 | | $(P0 + P1 + P2) + P3$ | |

### Register Allocation

Table 3.12 presents an alternative view of the schedule, namely the register values after each cycle. This is included in order to illustrate the internal data flow. Note how the final significand and exponent are placed in their respective R0s. This is common to all operations, as the normalize and round operations expect the value they act upon to be present in these registers.

### Control

Table 3.13 lists all the control signals present in the floating-point multiplication design. Please refer to fig.3.9 for details on how the signals are connected to the

Table 3.12: Floating-Point Multiplication - Internal Register Values

| Cycle | sig.R0 | sig.R1 | sig. result | exp.R0 | exp.R1 | exp. result |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $E_A$ | $E_B$ | 0 |
| 1 | 0 | 0 | 0 | $E_A + E_B$ | $E_B$ | 0 |
| 2 | 0 | 0 | 0 | $(E_A + E_B) - bias$ | $E_B$ | 0 |
| 3 | $P0$ | 0 | 0 | $(E_A + E_B) - bias$ | $E_B$ | 0 |
| 4 | $P0 + P1$ | 0 | 0 | $(E_A + E_B) - bias$ | $E_B$ | 0 |
| 5 | $(P0 + P1) + P2$ | 0 | 0 | $(E_A + E_B) - bias$ | $E_B$ | 0 |
| 6 | $(P0 + P1 + P2) + P3$ | 0 | 0 | $(E_A + E_B) - bias$ | $E_B$ | 0 |

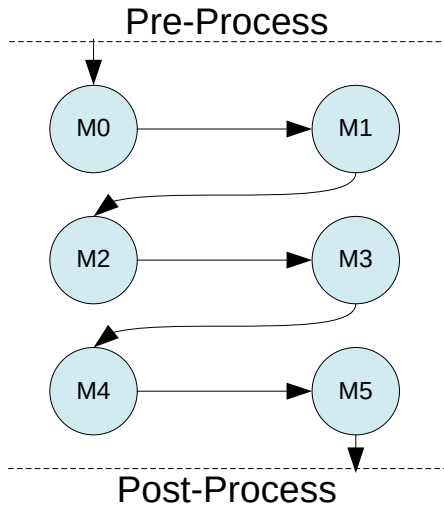


Figure 3.11: Floating-Point Multiplication - Control Flow/State Chart

various functional units.

The control flow of this operation is fairly simple, as it contains no branches, and consumes the same amount of clock cycles every time. The control flow/state chart for the floating-point multiplication is shown in Fig.3.11. Note that this state chart assumes that all input operands are placed in the appropriate registers upon start. This is referred to as pre-process, which also deals with detection of invalid operations and inputs.

In addition, the control steps for the normalization, rounding and final exception checking is not included. These steps are referred to as post-process. Normalization is discussed in Ch.3.8, rounding is discussed in Ch.2.3 and 3.9.

Finally, the sequence of control signals that generate the mentioned behavior must

Table 3.13: Floating-Point Multiplication - Control Signals

| Name | Bit Width | Default | Description |
|---|---|---|---|
| maskAndShiftOp | 2 | 00 | |
| mulEnable | 1 | 0 | Enable multiplier? |
| shiftAndExtendOp | 2 | 00 | Chooses how to shift and zero-extend the multiplier output |
| srfWriteSelectR0 | 1 | 0 | Muxes between the ALU result and the input port |
| srfWriteSelectR1 | 1 | 0 | Muxes between the ALU result and the input port |
| srfWriteEnableR0 | 1 | 0 | Enable write to s.R0? |
| srfWriteEnableR1 | 1 | 0 | Enable write to s.R1? |
| srfReadSelectA | 3 | 000 | Chooses which register to output on port A |
| srfReadSelectB | 3 | 001 | Chooses which register to output on port B |
| sigAluRegOrMul | 1 | 0 | Muxes between an extended partial product and register file, port A |
| sigAluOp | 4 | 0000 | Significand ALU OpCode |
| srrWriteEnable | 1 | 0 | Enable write to the significand result register? |
| erfWriteSelectR0 | 1 | 0 | Muxes between the ALU result and the input port |
| erfWriteSelectR1 | 1 | 0 | Muxes between the ALU result and the input port |
| erfWriteEnableR0 | 1 | 0 | Enable write to e.R0? |
| erfWriteEnableR1 | 1 | 0 | Enable write to e.R1? |
| erfReadSelectA | 3 | 000 | Chooses which register to output on port A |
| erfReadSelectB | 3 | 001 | Chooses which register to output on port B |
| expAluOp | 3 | 000 | Exponent ALU OpCode |
| errWriteEnable | 1 | 0 | Enable write to the exponent result register? |

be specified. Table 3.14 lists all states relevant to this operation, and specifies the control signals in each state. To make the table more readable, only values that differ from their default values are listed. Thus, this table should be compared with tab.3.13 for a complete understanding of which control signals are set to what value, in a given state.

### 3.3.4 Exceptions

The floating-point multiplication can trigger several exceptions, a notable example is the multiplication between zero and infinity.

Table 3.14: Floating-Point Multiplication - State Specification

| Signal | Value |
|---|---|
| **State: M1** | |
| maskAndShiftOp | MASK_AND_SHIFT_A8C8 |
| mulEnable | 1 |
| erfWriteEnableR0 | 1 |
| expAluOp | ADD |
| **State: M2** | |
| maskAndShiftOp | MASK_AND_SHIFT_A8D16 |
| mulEnable | 1 |
| sigAluRegOrMul | 1 |
| erfWriteEnableR0 | 1 |
| erfReadSelectB | 110 |
| expAluOp | SUB |
| **State: M3** | |
| maskAndShiftOp | MASK_AND_SHIFT_B16C8 |
| mulEnable | 1 |
| srfWriteEnableR0 | 1 |
| sigAluRegOrMul | 1 |
| sigAluOp | MOVA |
| **State: M4** | |
| maskAndShiftOp | MASK_AND_SHIFT_B16D16 |
| mulEnable | 1 |
| shiftAndExtendOp | SHIFT_0_BIT_AND_EXTEND |
| srfWriteEnableR0 | 1 |
| srfReadSelectB | 000 |
| sigAluRegOrMul | 1 |
| sigAluOp | ADD |
| **State: M5** | |
| mulEnable | 1 |
| shiftAndExtendOp | SHIFT_0_BIT_AND_EXTEND |
| srfWriteEnableR0 | 1 |
| srfReadSelectB | 000 |
| sigAluRegOrMul | 1 |
| sigAluOp | ADD |
| **State: M6** | |
| mulEnable | 1 |
| shiftAndExtendOp | SHIFT_TRUNC_AND_EXTEND |
| srfWriteEnableR0 | 1 |
| srfReadSelectB | 000 |
| sigAluRegOrMul | 1 |
| sigAluOp | ADD |

Table 3.15 lists all the exceptions that may be caused by this operation. Note that some exceptional cases —such as operations on a *NaN* —are shared among all floating-point operations. These cases were discussed in Ch.2.3.6. Please refer to Ch.3.10 for more details on the actual implementation of the exception handling.

Table 3.15: Floating-Point Multiplication - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Invalid operation | $\pm 0 \times \infty$ or $\infty \times \pm 0$ | Yes |
| Inexact | Fraction before rounding differs from fraction after rounding | No |
| Overflow | Result too large to be represented | No |
| Underflow | Result too small to be represented | No |

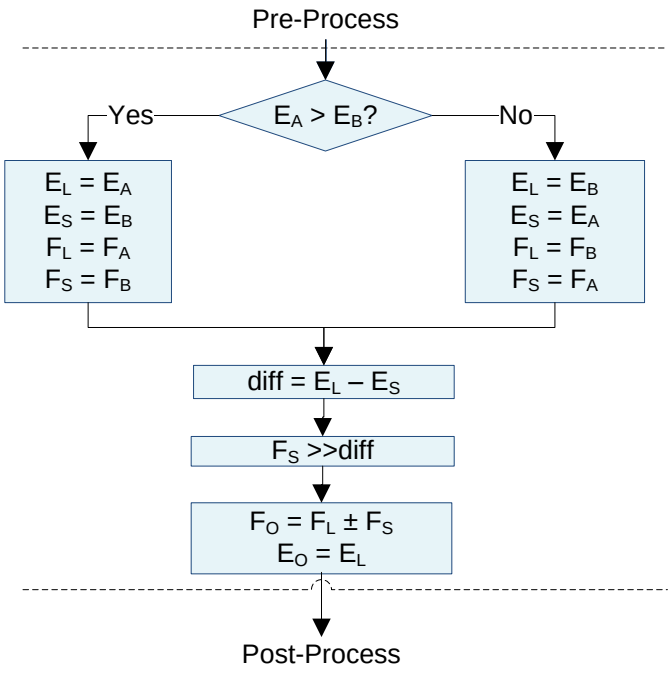


Figure 3.12: Floating-Point Addition/Subtraction Algorithm

## 3.4 Floating-Point Addition and Subtraction

The floating-point addition and subtraction are two closely related operations, at least in the case of signed operands. Hence, they will be both discussed and implemented together even though they are assigned unique op-codes in the FPU design.

### 3.4.1 Algorithm and Design Considerations

The addition and subtraction operations are far more complex than the multiplication operation, at least in terms of control. This is mainly due to the necessary adjustment of the input exponents, as well as several conditional operations present in the control path.

Figure 3.12 shows the algorithm for floating-point addition and subtraction. The figure is adapted from [5].

The algorithm can be summarized in the following steps:

Table 3.16: Effective Addition or Effective Subtraction

| Operation | Sign(A) | Sign(B) | Effective Operation |
|---|---|---|---|
| Add | + | + | A+B |
| Add | + | - | A-B |
| Add | - | + | -A+B |
| Add | - | - | -A-B |
| Sub | + | + | A-B |
| Sub | + | - | A+B |
| Sub | - | + | -A-B |
| Sub | - | - | -A+B |

1. Subtract the input exponents, in order to compare them
2. Right-shift the significand that belongs to the smallest exponent, by the absolute exponent difference
3. Add or subtract the two operands
4. Negate the sum generated in the previous step, if it yielded a negative result
5. Keep the largest exponent as the result exponent
6. Normalize and round the result. This is more complex than in the multiplication case, as the number of leading zeros in the result is harder to predict

Please refer to [11] and [5] for more details on the algorithm.

An important implementation consideration is the concept of *effective addition* and *effective subtraction*. As we are dealing with signed operands, it is necessary to determine which operation is actually going to be performed. This is further complicated by the fact that the standard requires both input and output values to be represented as *sign-magnitude* instead of *two's compliment* notation.

Table 3.16 shows the possible combinations of operations and operand signs, and the corresponding effective operation. The determination of the effective operation can be performed according to Fig.3.13. These figures show that a negation of at least one operand is required, in order to perform all possible combinations of operations. This introduces a problem — namely the concept of negative numbers — which was not present in the floating-point multiplication operation.

One possible way of dealing with effective subtraction is to sort the operands, and always subtract the smaller operand from the larger one. This will always yield a positive result, and the sign can be kept track of in the control unit. One problem with this approach is that the magnitude of the significands will be affected by the exponents, due to the pre-adjustment mentioned previously. This means that the comparison of the significands must be delayed until this adjustment has been performed, thus prolonging the execution time of the entire operation.

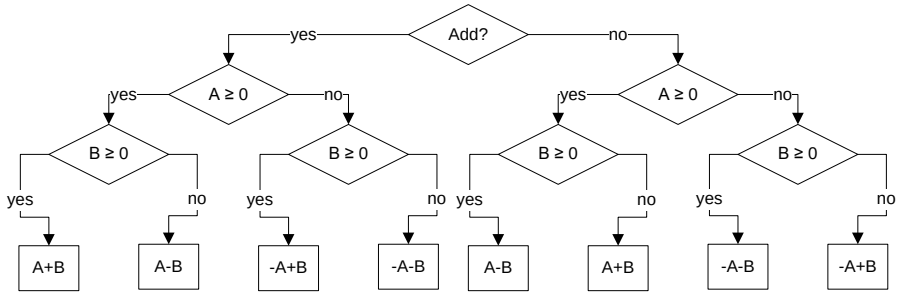


Figure 3.13: Determination of the Effective Operation

Another approach is to negate one of more of the operands, by converting it into a negative number in two's compliment notation. The addition or subtraction can then be performed directly, yielding a result in two's compliment notation. The result must then be converted back into sign-magnitude representation, if it is to conform with the IEEE-754 representation format.

From Fig.3.13, it it clear that all possible combinations can be performed by negating at most one operand. For instance, $-A - B$ can be performed by negating A, then subtracting B from the result. As the significand operations are dependent on the exponent operations, there are several free time slots available for negating one of the operands, without using more cycles in total. Hence, this approach will be chosen: significand $F_A$ will be negated if necessary, while the eventual negation of $F_B$ will be handled by the subtraction operation in the significand ALU.

### 3.4.2 Organization

Figure 3.14 shows the proposed architecture for the two operations. It is very similar to the proposed architecture for performing floating-point multiplication that was given in Fig.3.9, page 25. Notable differences are the absence of the external multiplier interface, as well as the newly introduced connections between the two ALU result registers and the ALUs themselves. The new data connections need some explanation: as the algorithm requires one of the significands to be shifted by the absolute difference between the input exponents, the exponent subtraction result must be relayed to the significand ALU in order to use it as a shift amount. In a similar fashion, the output of the significand ALU's *count leading zeros* operation must be available to the exponent ALU, in order to perform a generic normalization. See ch.3.8 and Ch.2.2 for details on this.

All of the storage elements are identical to the ones introduced in the previous section. The ALUs, however, are slightly more complicated. Unlike the multiplication case, the ALUs now need to incorporate generic shift operations. This is required for both adjustment of input as well as normalization of the result. In order to

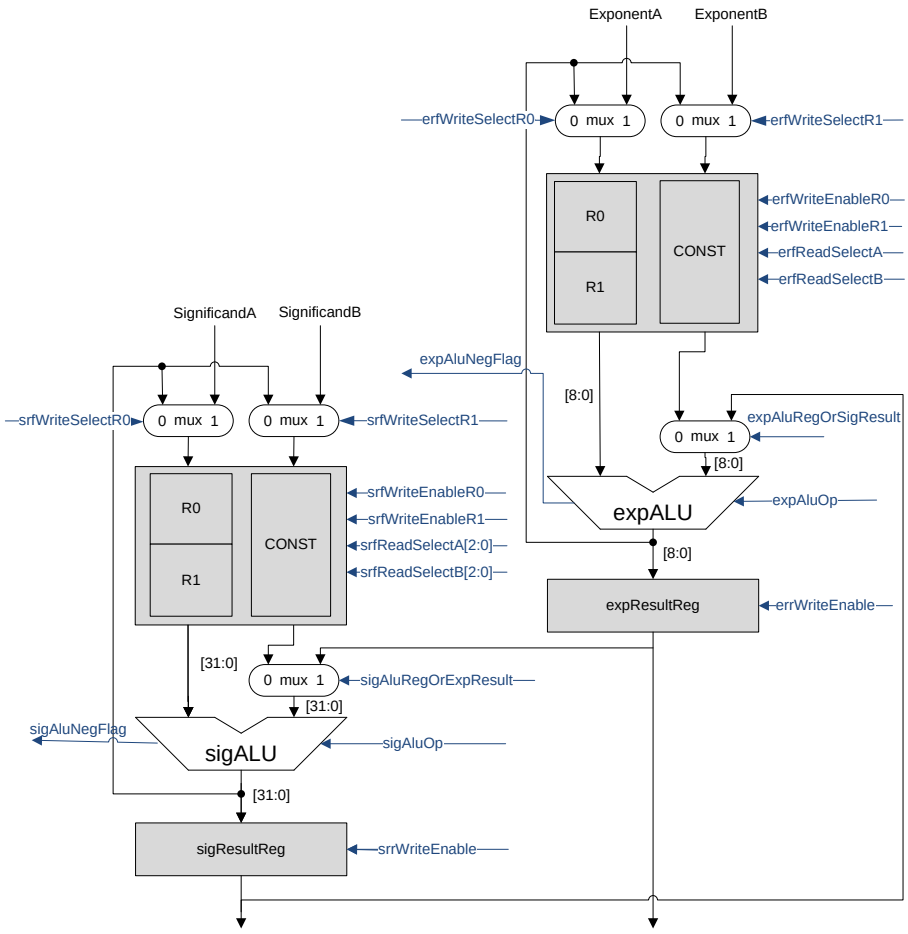


Figure 3.14: Floating-point Addition/Subtraction - Architecture

function correctly, the right-shift used to adjust one of the significands must be an arithmetic shift, due to the fact that we may negate the significand before it is shifted.

Another new feature of the ALUs is the negation operation. This operation performs a two's compliment negation (inverting all the bits and adding one) on one of the operands. The operation is used both to find the absolute value of the difference between the input exponents, as well as negating certain operands in the case of signed addition or subtraction. This was elaborated in the previous section. Negating a number is functionally similar to a subtraction, which is performed by adding a negated operand. To save hardware resources, the negation operation can only be performed on ALU operand B. This allows sharing of resources between

the subtraction and the negation operation.

### 3.4.3 Scheduling and Control

#### Dataflow

The abstract flowchart given in Fig.3.12 can be fitted to the given architecture, the result is shown in Fig.3.15. Only the arithmetic stage of the operation is presented, pre-processing and post-processing is discussed separately. Note how

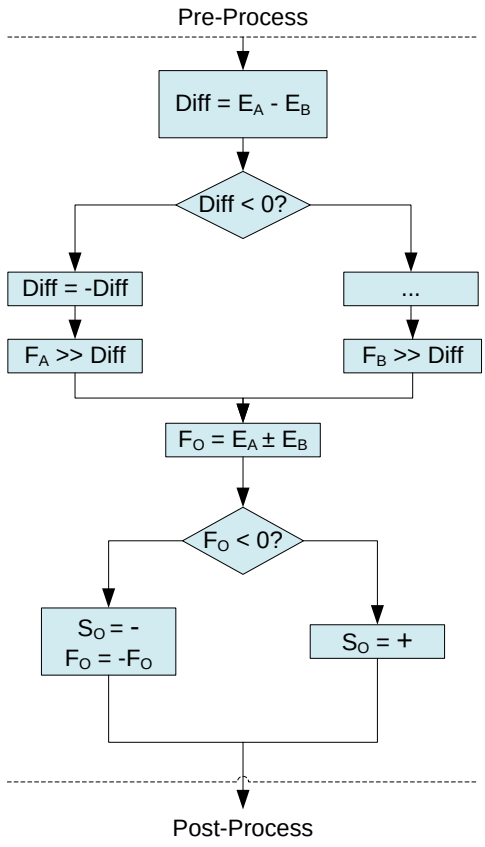


Figure 3.15: Floating-point Addition/Subtraction Algorithm - Fitted to the Proposed Architecture

the comparison of the two exponents is performed as a subtraction, followed by a sign test. The swapping of operands is omitted, by splitting the control flow into

two separate paths, this is more flexible than moving data around in a register-constrained environment.

Based on the architecture-fitted representation of the algorithm, we can derive the dataflow graph for the operation. To keep the DFGs unconditional, two versions are given in Fig.3.16; one shows the case where $E_A \geq E_B$, the other shows the case where $E_A < E_B$.

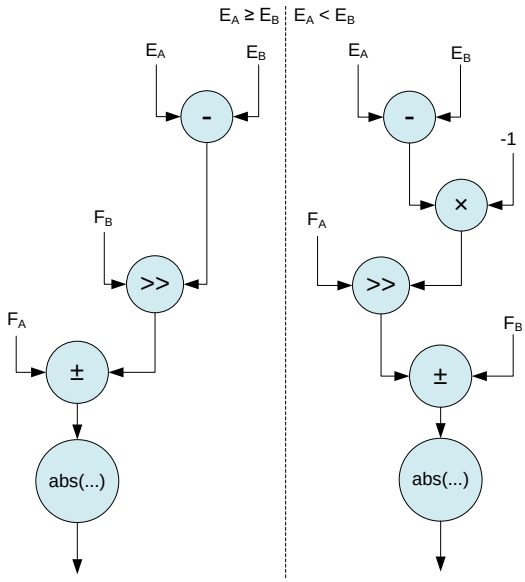


Figure 3.16: Floating-Point Addition/Subtraction - Unconditional DFGs

### Functional Unit Binding

The functional unit binding is fairly simple: operations on the significand will be performed in the significand ALU, while operations on the exponent will be performed by the exponent ALU. Up to three negation operations are required during the execution: one negation of $F_A$, one negation of $E_A - E_B$ and finally the negation of $F_A \pm F_B$. These negations are conditional, and will be controlled by the control unit. In order to keep the dataflow consistent between different control paths, empty nodes or NOPs have been inserted in the cases where no negation is needed. This might slow the operation down a cycle or two in a some cases. Thus, it is a good place to start of optimization of the architecture is required.

### Scheduling

Table 3.17 and 3.18 shows which operation is performed by which functional unit in a given cycle. Note that a few cycles are being wasted, especially cycle 2 in the case where $E_A \geq E_B$.

Table 3.17: Floating-Point Addition/Subtraction - Schedule ($E_A \geq E_B$)

| Cycle | Significand ALU | Exponent ALU |
|---|---|---|
| 1 | neg($F_A$)? | $E_A - E_B$ |
| 2 | | |
| 3 | $F_B >> (E_A - E_B)$ | |
| 4 | $F_A \pm F_B$ | |
| 5 | $neg(F_A \pm F_B)$? | $(E_A - E_B) + 1$ |

Table 3.18: Floating-Point Addition/Subtraction - Schedule ($E_A < E_B$)

| Cycle | Significand ALU | Exponent ALU |
|---|---|---|
| 1 | neg($F_A$)? | $E_A - E_B$ |
| 2 | | $neg(E_A - E_B)$ |
| 3 | $F_A >> (E_A - E_B)$ | |
| 4 | $F_A \pm F_B$ | |
| 5 | neg($F_A \pm F_B$)? | $(E_A - E_B) + 1$ |

### Register Allocation

Just as for multiplication, the register allocation is quite simple. Most of the operations read two operands from the corresponding register file, and overwrites one of there registers with the new result. A notable exception is the case of results that must be transmitted to the significand pipeline from the exponent pipeline, and vice versa. Due to the way these two pipelines are interconnected, the result must be written to the corresponding result register, in order for it to be accessible from the other pipeline. See Fig.3.14 for an illustration of this interconnection.

As the pre-adjustment of the input operands requires different updates of data, the registers transfers will be different as well. The difference is illustrated in Tab.3.19 and Tab.3.20, which show the register contents after a given cycle. $diff$ denotes the expression $E_A - E_B$, while $sum$ denotes the summation between the shifted and the unshifted significand.

### Control

Based on the scheduling of the operation, along with the register allocation, the control path of the addition and subtraction operations can be implemented ac-

Table 3.19: Floating-Point Add/Sub - Internal Register Values ($E_A \geq E_B$)

| Cycle | sig.R0 | sig.R1 | sig. result | exp.R0 | exp.R1 | exp. result |
|---|---|---|---|---|---|---|
| 0 | $F_A$ | $F_B$ | 0 | $E_A$ | $E_B$ | 0 |
| 1 | $\pm F_A$ | - | - | - | $diff$ | 0 |
| 2 | - | - | - | - | - | $diff$ |
| 3 | - | $F_B >> diff$ | - | - | - | - |
| 4 | $\pm F_A \pm (F_B >> diff)$ | - | - | - | - | - |
| 5 | - | - | $\pm sum$ | - | - | $E_A$ |

Table 3.20: Floating-Point Add/Sub - Internal Register Values ($E_A < E_B$)

| Cycle | sig.R0 | sig.R1 | sig. result | exp.R0 | exp.R1 | exp. result |
|---|---|---|---|---|---|---|
| 0 | $F_A$ | $F_B$ | 0 | $E_A$ | $E_B$ | 0 |
| 1 | $\pm F_A$ | - | - | $diff$ | - | 0 |
| 2 | - | - | - | - | - | $diff$ |
| 3 | $F_A >> diff$ | - | - | - | - | - |
| 4 | $(\pm(F_A >> diff) \pm F_B$ | - | - | - | - | - |
| 5 | - | - | $\pm sum$ | - | - | $E_B$ |

cording to Fig.3.17. The state chart includes two conditional transitions, the first is governed the sign bit of the exponent difference, the latter is determined by the sign of the effective addition or subtraction. As mentioned, some combinations of operation and input signs will require $F_A$ to be negated, into a two's compliment representation. This is determined by the sign of the inputs, and the actual opcode. Thus, this operation will not require a separate state, but rather be performed in EXP_SUB.

The conversion of a negative addition or subtraction, however, requires a branch in the state machine, as the actual operation is determined by the sign bit in the previous state, which is not preserved across clock boundaries. This is the motivation behind the states SUM_NEG and SUM_POS. In these two states, the control unit will set the sign of the final result, which is stored inside of the control unit itself.

Table 3.21 lists all the relevant control signals that are present in the floating-point addition/subtraction architecture.

Table 3.22 lists all states included in the arithmetic stage of the floating-point addition or subtraction. Again, only control signal values that differ from their default value are specified.

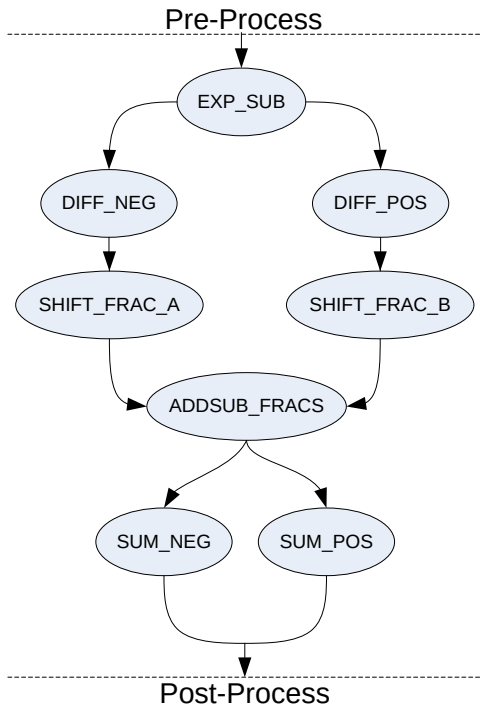


Figure 3.17: Floating-Point Addition/Subtraction Control Flow/State Chart

### 3.4.4 Exceptions

Like the other operations, addition and subtraction may lead to several exceptional cases. These are highlighted in tab.3.23.

Table 3.21: Floating-Point Add/Sub - Control Signals

| Signal | Bit Width | Default | Description |
|---|---|---|---|
| srfWriteSelectR0 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteSelectR1 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteEnableR0 | 1 | 0 | Update significand register R0? |
| srfWriteEnableR1 | 1 | 0 | Update significand register R1? |
| srfReadSelectA | 4 | 0000 | Chooses which value to output on significand register file read port A |
| srfReadSelectB | 4 | 0001 | Chooses which value to output on srf read port B |
| sigAluRegOrExpResult | 1 | 0 | Forward SRF read port B, or the ERR value to the significand ALU? |
| sigAluOp | 4 | 0000 | Significand ALU OpCode |
| srrWriteEnable | 1 | 0 | Update the significand result register (SRR)? |
| erfWriteSelectR0 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteSelectR1 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteEnableR0 | 1 | 0 | Update exponent register R0? |
| erfWriteEnableR1 | 1 | 0 | Update exponent register R1? |
| erfReadSelectA | 3 | 000 | Chooses which value to output on exponent register file read port A |
| erfReadSelectB | 3 | 001 | Chooses which value to output on exponent register file read port B |
| expAluRegOrSigResult | 1 | 0 | Forward SRF read port B, or the SRR value to the significand ALU? |
| expAluOp | 3 | 000 | Exponent ALU OpCode |
| errWriteEnable | 1 | 0 | Update the exponent result register (ERR)? |
| resultReady | 1 | 0 | Flag that a result is ready, and the unit is ready for a new operation |

Table 3.22: Floating-Point Add/Sub - State Specification

| Signal | Value |
|---|---|
| EXP_SUB | |
| erfWriteEnableR0 | $(E_A - E_B) > 0$ |
| srfReadSelectB | 0 |
| sigAluOp | NOP/NEGB |
| srfWriteEnableR0 | 1 / 0 |
| srfWriteEnableR1 | 0 / 1 |
| DIFF_NEG | |
| erfReadSelectB | 0000 |
| expAluOp | NEGB |
| errWriteEnable | 1 |
| DIFF_POS | |
| erfReadSelectA | 0001 |
| expAluOp | MOVA |
| errWriteEnable | 1 |
| SHIFT_FRAC_A | |
| erfReadSelectA | 001 |
| expAluOp | MOVA |
| erfWriteEnableR0 | 1 |
| srfReadSelectA | 0000 |
| sigAluRegOrExpResult | 1 |
| sigAluOp | SHRA |
| srfWriteEnableR0 | 1 |
| SHIFT_FRAC_B | |
| srfReadSelectA | 0001 |
| sigAluRegOrExpResult | 1 |
| sigAluOp | SHRA |
| srfWriteEnableR0 | 1 |
| ADDSUB_FRACS | |
| sigAluOp | ADD/SUB |
| srfWriteEnableR0 | 1 |
| SUM_POS | |
| - | - |
| SUM_NEG | |
| srfReadSelectB | 0000 |
| sigAluOp | NEGB |
| srfWriteEnableR0 | 1 |

Table 3.23: Floating-Point Addition/Subtraction - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Invalid operation | $+\infty + -\infty$, $-\infty + +\infty$, $+\infty - +\infty$ or $-\infty - -\infty$ | Yes |
| Inexact | Fraction before rounding differs from fraction after rounding | No |
| Overflow | Result too large to be represented | No |
| Underflow | Result too small to be represented | No |

## 3.5 Floating-Point Division

The division operation is, along with the square root operation, slightly different from the other operations. The main reason for this is that is relies on a sequential, bit-serial algorithm to produce the final result. This will have some impacts on the organization of the design.

### 3.5.1 Algorithm and Design Considerations

The basic algorithm behind floating-point division is fairly simple:

1. The inputs are read into their respective registers
2. *bias* is added to $E_A$
3. $E_B$ is subtracted from $E_A + bias$, in order to produce the final exponent
4. The signifcand division is performed by a suitable algorithm
5. The result is normalized and rounded

The challenge is to perform the significand division itself, the treatment of the exponent is trivial.

A variety of division algorithms were discussed in the preliminary project [5]. The project concluded that a sequential, bit-serial algorithm is most suited for this design. Such an approach will result in a low hardware-cost, at the expense of execution speed. Two such algorithms were presented, namely the *restoring division algorithm* and the *non-restoring division algorithm* [11].

Both of these algorithms generate $n$ bits of precision in $n$ iterations. The former is very straight-forward, and generated a usable answer immediately. However, the algorithm has some conditional execution issues, which may cause it to consume two cycles per iteration This issue can be solved by implementing the non-restoring division scheme instead, however the answer produced must be corrected and a few special cases must be dealt with. See [5] for details.

To avoid the disadvantages of these two algorithms, a compromise is proposed: the division is performed according to the restoring division scheme. However, the partial remainder is NOT updated when a trial-subtraction results in a negative value, thus eliminating the need for a separate correction step. It is important to note that the left-shifting of the partial remainder still must take place, in order to produce the correct result. This suggests that the left-shift of the partial remainder must be performed inside the register itself, not as a part of the datapath between the registers output and input.

This approach has two potential pitfalls: First, it introduces a slightly more complicated control unit. However, the control unit of this design is already quite complex, so the difference should be negligible. Secondly, the area usage of a register that allows in-place left-shifting may be larger than a corresponding left-shift performed by suitable wiring between the register and the ALU.

To conclude, this hybrid approach gives a execution time lower than the non-restoring division, as well as being easier to implement and debug. The potential pitfall is that the area consumption may be higher. If this is indeed the case, a non-restoring implementation is to be preferred, as concluded in [5].

In general, a sequential division algorithm requires three storage elements: the temporary remainder, the divisor and the partial quotient. This corresponds well with the existing architecture, as the current design indeed features two input registers and a result register. A typical division architecture is given in [13], upon which the solution in the preceding report was based.

As mentioned, this operation introduces the need for shift registers in the design. Since the algorithm itself requires that the numerator is left-shifted relative to the denominator, we must be able to left-shift this value. This could be performed by utilizing the existing shift-capabilities of the significand ALU, however this will result in a structural hazard, forcing the execution time to be twice as long; the ALU is already assigned a subtraction operation per iteration. Thus, this must be solved by adding shift capabilities to one of the registers in the significand register file, namely $R0$.

A similar problem arises when we look at the way the answer is generated. A new result bit is determined each clock cycle, namely the digit to the right of the one generated in the previous cycle. Again, this suggests adding shift-register capabilities to the significand result register, which allows a new result bit to be shifted into its LSB.

The final issue with the implementation is the normalization and rounding of the significand. The former operation is rather simple to perform by utilizing the shift-capabilities of the result register, however the rounding operation requires the result to pass through the ALU (see Ch.3.9 for details on this). Thus, it is necessary to introduce a feedback from the significand result register output, back to the significand ALU input.

### 3.5.2 Organization

The resulting architecture for performing floating-point division is shown in fig.3.18. The organization is mostly similar to the one presented in the previous sections, with a few notable exceptions: most important is the introduction of a bus from the significand result register, back to one of the inputs of the significand ALU. As mentioned, this is done to accommodate rounding and normalization in accordance

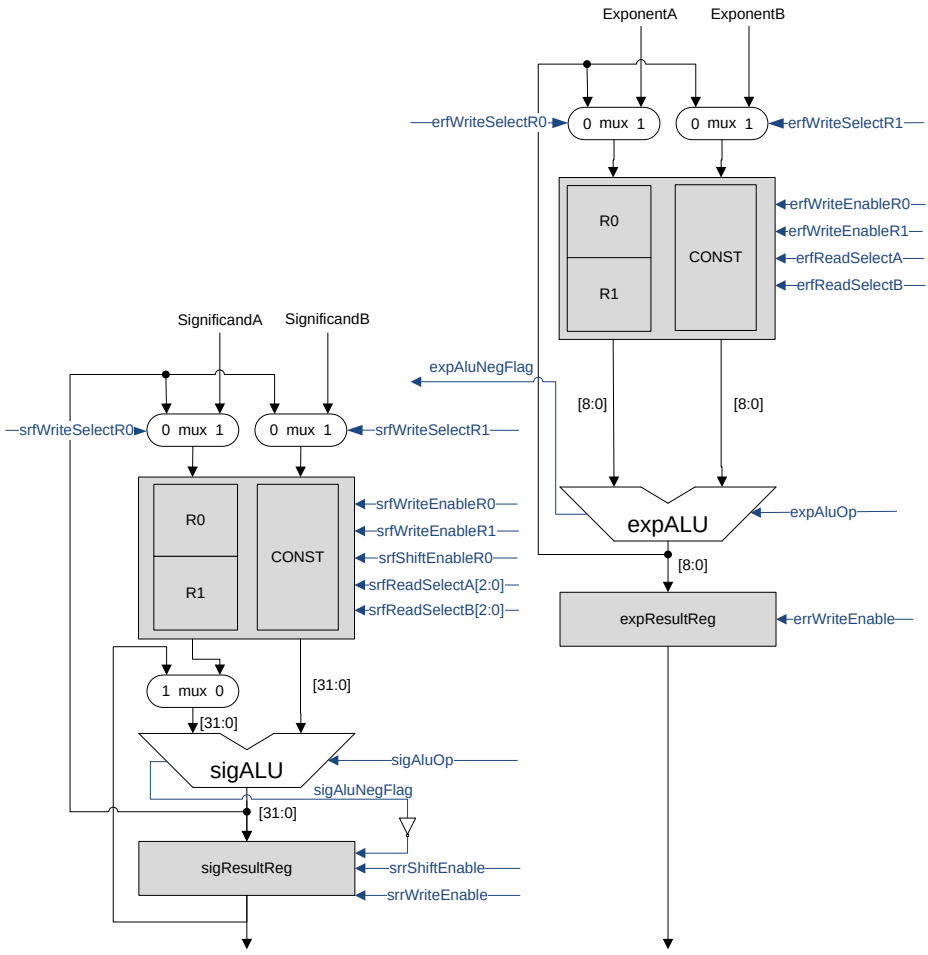


Figure 3.18: Floating-Point Division Architecture

with the other operations.

Note the introduction of shift capabilities of both the significand result register and significand register R0. These will perform necessary updates of the various operands, during the execution of the sequential algorithm. Thus, this functionality can also be utilized for other algorithms with functional similarities, such as square root extraction.

Similar to the multiplication operation, there is no exchange of data between the significand pipeline and the exponent pipeline. The only communication between them is through the shared control unit.

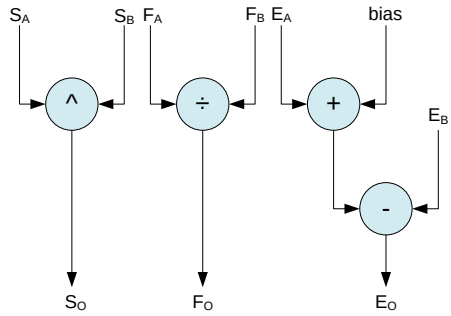


Figure 3.19: Floating-Point Division DFG

### 3.5.3 Scheduling and Control

#### Dataflow

The dataflow of the floating-point division operation is relatively simple. The exponent treatment is more or less identical to the one for multiplication, except that the input exponents are subtracted instead of added together. This also introduces a slight change in the biasing of the result.

As the significand calculation consists of iterating a certain operation many times, this is fairly simple to control as well.

Figure 3.19 shows the top-level DFG of the division operation. Again, pre-processing and post-processing are dealt with in a separate section.

#### Functional Unit Binding

The majority of this algorithm is based on an iterative approach, hence the functional unit binding is relatively simple. The treatment of the exponent is very similar to the multiplication case, while the significand is centered around the significand ALU and the attached registers.

#### Scheduling

Table 3.24 shows the scheduling of the operations. Note how most of the steps are spent dealing with the significand division. The left-shift of the numerator is performed inside its register, hence it is not included in this table of functional unit operations.

Table 3.24: Floating-Point Division - Schedule

| Cycle | Significand ALU | Exponent ALU |
|---|---|---|
| 1 | $srf.R0 - srf.R1$ | $E_A + bias$ |
| 2 | $srf.R0 - srf.R1$ | $(E_A + bias) - E_B$ |
| 3→26 | $srf.R0 - srf.R1$ | - |

#### Register Allocation

Just like the functional unit binding, the register allocation is simple and fairly regular. Table 3.25 shows the details.

Table 3.25: Floating-Point Division - Internal Register Values

| Cycle | sig.R0 | sig.R1 | sig.result | exp.R0 | exp.R1 | exp.result |
|---|---|---|---|---|---|---|
| 0 | Tmp.Rem. | Denom. | Tmp.Quotient | $E_A$ | $E_B$ | 0 |
| 1 | Tmp.Rem. | Denom. | Tmp.Quotient | $E_A + bias$ | $E_B$ | 0 |
| 2 | Tmp.Rem. | Denom. | Tmp.Quotient | $(E_A + bias) - E_B$ | $E_B$ | 0 |
| 3→26 | Tmp.Rem. | Denom. | Tmp.Quotient | $(E_A + bias) - E_B$ | $E_B$ | 0 |

#### Control

The control path of the division operation is the simplest yet: the first two steps will require some operations on the exponents, but most of the control steps will only issue subtraction operations to the significand ALU, and determine the update of the temporary remainder based on the corresponding sign flag. The control flow will be governed by an iteration counter, causing the normalization stage to be invoked after the correct number of iterations. Note that the division algorithm requires a slightly different approach to normalization, as the result bits are located all the way to the right in the register, unlike the previous operations. More details on this can be found in Ch.3.8.

Table 3.26 lists all control signals that are relevant to this operation. Again, this is fairly similar to the preceding operations, however the inclusion of shift capabilities in some registers have caused some new control signals to be added.

Figure 3.20 shows the state chart that will control the division operation.

### 3.5.4 Relevant Exceptions

Table 3.28 lists the relevant exceptions for the floating-point division operation.

Table 3.26: Floating-Point Division - Control Signals

| Signal | Bit Width | Default | Description |
|---|---|---|---|
| srfWriteSelectR0 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteSelectR1 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteEnableR0 | 1 | 0 | Update significand register R0? |
| srfWriteEnableR1 | 1 | 0 | Update significand register R1? |
| srfShiftEnableR0 | 1 | 0 | Shift significand register one digit to the left? |
| srfReadSelectA | 4 | 0000 | Chooses which value to output on significand register file read port A |
| srfReadSelectB | 4 | 0001 | Chooses which value to output on srf read port B |
| sigAluRegOrMul | 1 | 0 | Forward srf read port A, or the multiplier output to the ALU? |
| sigAluSrr | 1 | 0 | Forward the result of the decision on the line above, or the SRR value to the significand ALU? |
| sigAluOp | 4 | 0000 | Significand ALU OpCode |
| srrWriteEnable | 1 | 0 | Update the significand result register (SRR)? |
| srrShiftEnable | 1 | 0 | Shift the significand result register one digit to the left? |
| erfWriteSelectR0 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteSelectR1 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteEnableR0 | 1 | 0 | Update exponent register R0? |
| erfWriteEnableR1 | 1 | 0 | Update exponent register R1? |
| erfReadSelectA | 3 | 000 | Chooses which value to output on exponent register file read port A |
| erfReadSelectB | 3 | 001 | Chooses which value to output on exponent register file read port B |
| expAluOp | 3 | 000 | Exponent ALU OpCode |
| errWriteEnable | 1 | 0 | Update the exponent result register (ERR)? |
| resultReady | 1 | 0 | Flag that a result is ready, and the unit is ready for a new operation |

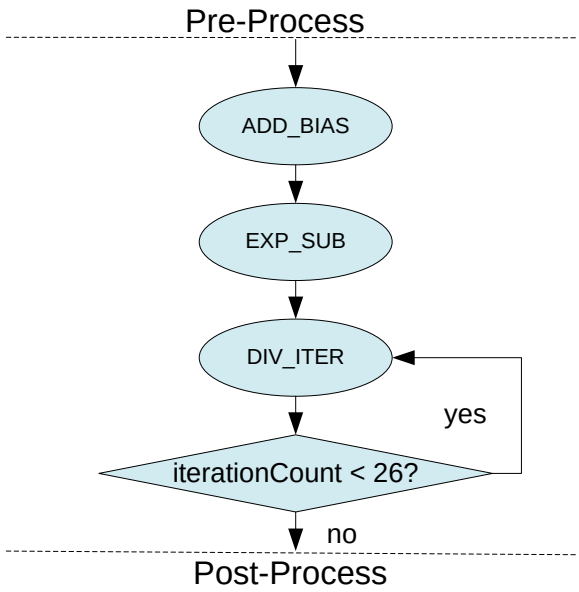


Figure 3.20: Floating-Point Division Control Flow/State Chart

Table 3.27: Floating-Point Division - State Specification

| Signal | Value |
|---|---|
| **State: ADD_BIAS** | |
| erfReadSelectB | ERF_REG_BIAS |
| expAluOp | ADD |
| erfWriteEnableR0 | 1 |
| srfShiftEnableR0 | 1 |
| sigAluOp | SUB |
| srfWriteEnableR0 | 1 |
| srrShiftEnable | 1 |
| **State: SUB_EXP** | |
| expAluOp | SUB |
| erfWriteEnableR0 | 1 |
| srfShiftEnableR0 | 1 |
| sigAluOp | SUB |
| srfWriteEnableR0 | 1 |
| srrShiftEnable | 1 |
| **State: DIV_ITER** | |
| srfShiftEnableR0 | 1 |
| sigAluOp | SUB |
| srfWriteEnableR0 | 1 |
| srrShiftEnable | 1 |

Table 3.28: Floating-Point Division - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Invalid operation | $\pm\infty / \pm\infty$ | Yes |
| Invalid operation | $\pm 0 / \pm 0$ | Yes |
| Division by Zero | Any value other than $\pm\infty$ divided by $\pm 0$ | Yes |
| Inexact | Fraction before rounding differs from fraction after rounding | No |
| Overflow | Result too large to be represented | No |
| Underflow | Result too small to be represented | No |

## 3.6 Square Root

Due to insufficient time, the square root operation has not been implemented into this design. Hence, this chapter will not be able to provide a detailed overview of the implementation and specification of floating-point square root. However, some pointers and ideas on how to include this functionality into the existing architecture will be presented.

### 3.6.1 Algorithm and Design Considerations

This section assumes that the reader is familiar with digital square root extraction. For more information on this topic, please refer to [5] and [11]. As concluded in the preliminary report, the square root operation features an exponent treatment that is somewhat similar to the addition/subtraction case, as well as a significand calculation that resembles the one found in the division operation.

Mathematically speaking, the exponent calculation involves dividing the input exponent by two. In a digital system this can be achieved by right-shifting the input exponent. This is only valid in the case where the input exponent is an even number, thus it might be necessary to adjust the exponent/significand pair prior to this operation; by right-shifting the significand by one digit while incrementing the exponent, the input is in the proper format.

The significand calculation is more complicated. According to [11], digital division can be regarded as division with a variable denominator. Thus, the challenge is to perform the necessary operations on the expression that will act as the denominator.

If we assume a *restoring division* scheme, the expression to subtract from the partial remainder simply consists of the partial quotient, with the digits 01 appended at the right end. In addition to this, the concatenation of quotient and 01 must be properly shifted according to the temporary remainder. This does not fit well with the way the quotient is currently being generated (see Ch.3.5 for details on this), as the quotient-constant pair will require a variable left-shift before they are subtracted from the temporary remainder.

A possible solution, that will introduce very little hardware resources, is to feed the temporary quotient back from the significand result register, feed it through the significand ALU, and perform a shift-operation. Then, the shifted expression is written to significand register R1. In the next cycle, the temporary remainder — which is assumed to reside in significand register R0 — is forwarded to the ALU, and the previously generated expression is subtracted from it. Then, the updated temporary quotient is fed back, 01 is appended and the expression is left-shifted. The sequence of operations is iterated, generating a quotient bit every other cycle.

Figure 3.21 illustrates the data flow during the significand square root extraction. Alternating cycles is spent updating the subtrahend by appending and shifting

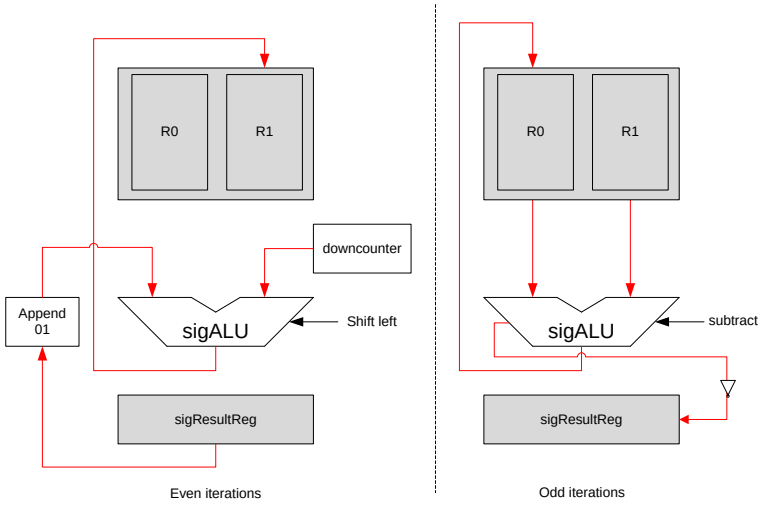


Figure 3.21: Floating-Point Square Root Extraction: Proposed Architecture (Abstract)

the temporary quotient (even cycles), and reducing the temporary remainder by subtraction (odd cycles). As for division, a negative subtraction result yields a 0 in the quotient, while a positive subtraction result yields a 1 in the quotient.

To summarize, this approach allows the current architecture to be left almost unmodified, while still being able to perform the square root extraction. The disadvantage of the method is that is consumes twice the amount of clock cycles, compared to a more specialized architecture. If square root performance is important, there are a few alternatives: one is to add generic shift logic into the datapath, between the temporary quotient and the significand ALU. This will most likely increase the critical path, as well as increase the hardware consumption of the entire design. Another possibility is to create a more specialized architecture, a good starting point is the design presented in [14].

### 3.6.2 Relevant Exceptions

Table 3.29 lists exceptions that are relevant to the floating-point square root operation.

Table 3.29: Floating-Point Square Root - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Invalid operation | Input is less than $-0$ | Yes |
| Inexact | Fraction before rounding differs from fraction after rounding | No |

## 3.7 Conversion Operations

A floating-point unit in an embedded system context would be pretty much useless without the means to convert between integer and floating-point formats; after all most data acquisition is performed as integer values read from an ADC, coefficients may arrive in the form of fixed-point numbers and so on. Hence, it is crucial to be able to convert these numbers to a floating-point representation before the relevant calculations are performed, and back again afterwards.

As stated in [5], the standard requires the implementation to be able to convert between all supported integer formats, and all supported floating-point formats. This project only aims to implement single-precision numbers, however the list of supported integer formats is harder to define. In general, a microcontroller has a narrow data width, but is able to operate on larger operands by splitting calculations into multiple passes. Thus, a microcontroller that is programmed using C may easily support integers of widths such as 64 and 128 bits.

However, as the size of both input and output ports of the floating-point unit is specified as 32 bits, we will focus on integers of 32 bit word size. This is enough to convert the output of most data converters, which typically produce a result of 10 to 14 bits [2]. Any integer smaller than 32 bit can be sign extended in order to make it conform to the format expected by the FPU. Thus, any microcontroller with a native data with less than or equal to 32 will be compatible with this choice.

The conversion operations will be discussed in a single section of this report. The motivation behind this is as follows: the operations share many common aspects (in fact they can be viewed as inverse operations), and they can be performed by the same functional units. In addition, no additional architectural features is required, in addition to those already presented. Thus, only control logic needs to be added in order to support the various conversion operations.

### 3.7.1 Overview

#### Floating-Point to Integer

Converting a floating-point number into an integer involves several obstacles. First, the value of the number must be rounded to an integer-valued number. Next, the range of the number must be taken into consideration: floating-point numbers may have a numeric value that is far greater than what can be represented by the use of an integer representation, even though their word lengths are equal.

To further complicate things, several floating-point values have no logical representation as integers, namely the *infinity* and *NaN* representations. These cases should trigger the invalid operation exception. The sign of 0 will be ignored when converting to integer.

To summarize, there are three cases:

1. Convertible values
2. Non-convertible values, due to range
3. Non-convertible values, due to special representation values

Let's discuss the general case first: If we interpret the a floating-point value as the $n$-bit significand sequence $2^0+2^{-1}+2^{-2}+\ldots+2^{-n}\times 2^E$ where $E$ is the true, unbiased exponent, the actual value of the significand is $2^{N-0}+2^{N-1}+2^{N-2}+\ldots+2^{N-n}$. The purpose of the conversion is to remove any digits that correspond to a weight of less than 0, in other words chop off any digits where $N-i$ is less than zero ($i$ denotes the position of a given digit in the sequence).

Assuming normalized fractions, we know that the weight of the first digit is always $2^0=1$. By placing the input significand to the left in a general-purpose register, we can remove bits with weights less than $2^0$ by right-shifting the operand. The number of shifts depends on the true exponent of the input, as well as the size of the result register. The important thing is to shift the digit with a weight of $2^0$ down in the LSB of the result register.

If the true exponent is too large, it is impossible to convert the value to an integer. This would cause the result to be too wide for the result register, rendering the operation impossible. This corresponds to case 2 in the list above, and should trigger an exception.

A problem with the method described above is that it does not support rounding. By simply chopping off bits, we effectively truncate the value, causing the floating-point number 1.9 to be converted to 1, even when the *round towards* $+\infty$ mode is active. This can be solved in the following way:

1. Find the true exponent by un-biasing the input exponent
2. Calculate the true position of the radix point
3. Left-shift the fraction, so that the digits with weights $2^{-1}$ and $2^{-2}$ is located in the *guard* and *round* locations in the register
4. Perform normal rounding, according to the current rounding-mode. See Ch.2.3 and Ch.3.9 for details on this
5. After rounding, right-shift the operand, in order to move the LSB of the integer part into the LSB of the result register

The difference from the first approach is that we split the right-shift into two parts, allowing the already present rounding logic to be utilized.

This might seem a bit confusing, so an example is appropriate: Assume a 8-bit register configuration, with the *guard* and *round* located in the two lowest bits. Assume the number $10.75_{ten} = 1010.11_{two}$ is represented as $1.0101100_{two} \times 2^{3_{ten}}_{ten}$. This exponent is unbiased, so we can skip the first step of the algorithm.

The next step is to calculate the true position of the radix point, which can be calculated as

$$(radix\ point\ position) - (true\ exponent)$$

In this case, the expression yields $7-3 = 4$. Thus, the actual value is $1010.1100_{two}$.

The next part is to right-shift the number, in order to place the proper bits in the guard and round positions. This corresponds to moving the true radix point to position two, as the $g$ and $r$ bits are assumed to be located in bit 1 and 0 before the rounding takes place. Hence, the shift amount is

$$(true\ RPP) - (RPP\ assumed\ before\ rounding) = 4_{ten} - 2_{ten} = 2_{ten}$$

where *RPP* denotes the *radix point position* Thus, the register content before rounding is $001010.11_{two}$. This corresponds to a fractional part of $.75_{ten}$, which will cause the integer part to be rounded to $1011_{two}$ in the default rounding mode.

Finally, the properly rounded integer answer can be generated, by right-shifting the rounded value all the way to the right in the register, in this case two digits. The final answer is $00001011_{two} = 11_{ten}$.

It should be noted that since the size of the fraction is restricted to 24 bits, the lower 8 bits of the input value will always be zero. This allows the rounding part to be skipped, in cases where the *guard* and *round* bits would be zero anyway.

The problem with too large floating-point values can be detected in the following manner: If the true exponent is larger than 31, the fraction would have to be left-shifted in order to represent the true radix point. This is not possible, and implies that any true exponent value larger than 31 should trigger an exception.

Luckily, the final case can be detected already at the input-stage. Special representation values can be detected in the first cycle of the operation, and the appropriate exception can be triggered without invoking the arithmetic core of this operation. This is a pure control-unit problem, which will be dealt with along with the other exception handling.

### Integer to Floating-Point

Converting an integer into a floating-point number is easier than the opposite. A 32 bit integer can not overflow a single-precision floating-point value, and no cases such as *infinity* or *NaN* can arise. It should be noted that due to the limited size of the significand, integer values which exceeds 24 bits can not be represented accurately, and the conversion will result in a loss of precision.

This operation takes a 32 bit integer number, either signed or unsigned, and converts it to a floating-point number. For example, $i2fp(64_{ten})$ will yield $64.0_{ten} = 1.0_{two} \times 2^6_{ten}$.

It is necessary to distinguish between signed and unsigned operands. The former will be assumed to have a two's compliment representation. As the floating-point result generated by the operation will have a separate sign-bit, the first step of the algorithm is to convert any signed integers to sign-magnitude. The next step is to find the weight of the most significand bit of the operand. For instance, the integer value $1010_{two}$ will have a MSB-weight of $2^3_{ten} = 8_{ten}$. The goal is to reduce this weight to one, by multiplying it with a suitable exponent and adjust the value accordingly. For example, the integer value $1010_{two} = 10_{ten}$ can be represented as $1.010_{two} \times 2^3_{ten}$. Note how the latter representation corresponds with the representation of floating-point values.

To summarize, a given integer can be converted to a floating-point value by extracting the 24 most significant bits of the integer, and calculating the corresponding exponent. This exponent value was derived as $(integer\ word\ length) - (number\ of\ leadingzeros\ in\ operand) + (bias - 1)$ in [5]. A more accurate way of describing this relation is $(radix\ point\ position) - (number\ of\ leading\ zeroes\ in\ operand) + bias$.

In the example above, the correct exponent would be $31 - 28 + 127 = 130$, which corresponds to an exponent value of 3. The resulting significand would be 1.00000000000000000000000, leading *one* included.

Thus, the final algorithm is:

1. Convert signed operand to sign-magnitude
2. Count leading zeros of the operand
3. Calculate the corresponding exponent according to the expression above
4. Extract the 24 most significant bits of the input operand, and place them in the significand

#### Floating-Point to an Integer-Valued Floating-Point Value

This operation is quite similar to the floating-point to integer operation, with one notable exception: instead of right-shifting the rounded intermediate value, it will be left-shifted back to yield a normalized fraction.

### 3.7.2 Organization

As mentioned previously, no additional hardware is required for these operations. Thus, the suggested organization is identical to the one presented in the previous

section (see fig.3.14 for details).

Important architectural features are:

- The ability to count leading zeros in a significand
- The ability to perform generic left- and right-shifts of a significand
- The ability to increase and decrease the exponent, according to values generated by the significand ALU

All of these operations were covered in the previous sections, as the conversion operations share characteristics with the input-adjustment and normalization operation performed during addition and subtraction.

### 3.7.3 Control

As the conversion operations introduce no extra functional units, they don't require any additional control signals either. Thus, the control signal specification given in the preceding chapters cover the conversion operations as well.

#### Floating-Point to Integer

In this implementation - due to a lack of time - only the truncate rounding mode was implemented. Hence, the right-shift of the input significand is performed in one step, instead of two.

The resulting operations are:

1. Read the input values into the proper registers
2. Calculate the true exponent by subtracting *bias*, left-align the input significand
3. Calculate the necessary shift-amount, by subtracting the true exponent from the true radix point position (31)
4. Negate the shifted significand if the input sign is high

This behavior can be created with the state sequence illustrated in fig.3.22. The states are specified in tab.3.30.

Table 3.30: Integer ↔ Floating-Point Conversion - State Specification

| Integer to Floating-Point | | Floating-Point to Signed Integer | |
|---|---|---|---|
| Signal | Value | Signal | Value |
| TEST_SIGN | | UNBIAS | |
| sigAluOp | MOVA | erfReadSelectB | 110 |
| NEGATE | | expAluOp | SUB |
| srfReadSelectB | 0000 | erfWriteEnableR0 | 1 |
| sigAluOp | NEGB | srfReadSelectB | 0100 |
| srfWriteEnableR0 | 1 | sigAluOp | SHLL |
| CLZ | | srfWriteEnableR0 | 1 |
| sigAluOp | CLZ | CALC_ADJ | |
| srfWriteEnableR1 | 1 | erfReadSelectA | 100 |
| srrWriteEnable | 1 | erfReadSelectB | 001 |
| ADJ1 | | expAluOp | SUB |
| erfReadSelectA | 101 | errWriteEnable | 1 |
| expAluRegOrSigResult | 1 | PREROUND_RSH | |
| expAluOp | SUB | sigAluRegOrExpResult | 1 |
| erfWriteEnableR0 | 1 | sigAluOp | SHRL |
| sigAluOp | SHLL | srfWriteEnableR0 | 1 |
| srfWriteEnableR0 | 1 | NEGATE | |
| ADJ2 | | srfReadSelectA | 0000 |
| expAluOp | MOVA | srfReadSelectB | 0000 |
| erfWriteEnableR0 | 1 | sigAluOp | MOVA/NEGB |
| srfReadSelectB | 0011 | srrWriteEnable | 1 |
| sigAluOp | SHRL | | |
| srfWriteEnableR0 | 1 | | |

### Integer to Floating-Point

The control steps can be shared among both signed and unsigned integer conversions. The only difference is that signed integers must be negated if their numerical value is negative. This step is unnecessary if we assume the integer to be unsigned, thus this operation can skip directly to the conversion itself.

The sign-test can be performed by moving the input integer through the significand ALU, and test the *negative* status flag.

### Floating-Point to Integer-Valued Floating-Point

This operations has not been implemented, again due to insufficient time. However, it shares many characteristics with the floating-point to integer conversion. Thus, it can be added to the design with only a small increase in hardware consumption.

### 3.7.4 Exceptions

#### Floating-Point to Integer

The relevant exceptions for conversions between floating-point values and integers are listed in Tab.3.31.

Table 3.31: Floating-Point to Integer Conversion - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Invalid operation | Attempting to convert any special representation value to integer | Yes |
| Inexact | Fraction before rounding differs from fraction after rounding | No |
| Overflow | Input too large to represent as integer | No |

### 3.7.5 Integer to Floating-Point

The exceptional cases that may arise when converting an integer to a floating-point number is summarized in Tab.3.32.

Table 3.32: Integer to Floating-Point Conversion - Exceptions

| Exception | Cause | "Init-Time"? |
|---|---|---|
| Inexact | Loss of precision due to finite significand size (input integer larger than 24 bits) | No |

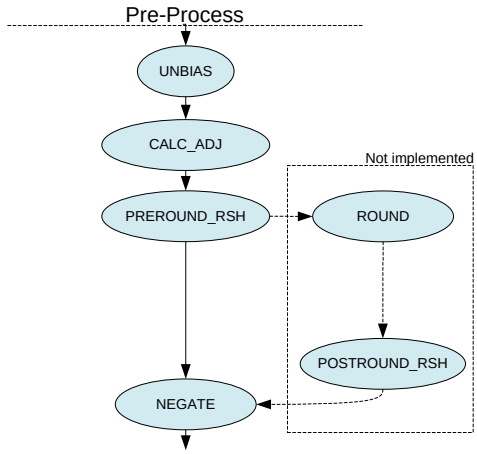


Figure 3.22: Floating-Point to Signed Integer Conversion - Control Flow/State Chart

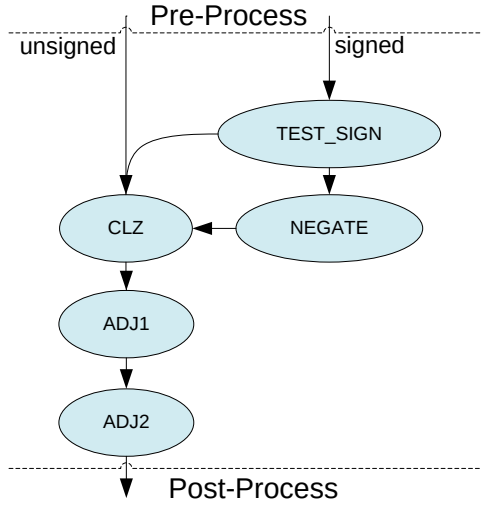


Figure 3.23: Integer to Floating-Point Conversion - Control Flow/State Chart

## 3.8 Normalization

Normalization of a floating-point number consists of placing the leading *one* of the significand in a given position in the result register, and adjusting the exponent accordingly.

In general, there are two things that decide how the normalization will act:

1. The radix point position in the actual register
2. The value range of the result to be normalized

After the normalization, the most significant bit of the significand must be located in a given position. This is required, as the subsequent rounding operation assumes a specified location.

In this implementation the leading one is assumed to be located in bit 30, counting from 0. This leaves one bit to the left of the leading one, which is required for rounding without the risk of losing information through overflow. See Ch.2.3 for more information about this. It should be noted that this emphpost-rounding normalization step is not implemented in the design.

### 3.8.1 Normalization of Multiplication

The normalization of a multiplication result is easy, as the value to be normalized is in the range $[1.0, 4.0)$. As stated in Ch.3.3, the radix point position after the accumulation of the partial products is located to the right of the two most significand bits.

Hence, the register contents will be on the form $2 : 30$ (integer:fraction), where the two most significand bits will be either 01, 10 or 11. To conform with the specified register content layout, as well as preserving the numerical value of the result, there are two possible normalization operations:

1. If the significand product is in the range $[1.0, 2.0)$, there is no need for any normalization
2. If the significand is in the range $[2.0, 4.0)$, the significand must be shifted one digit to the right, and the exponent increased by one

### 3.8.2 Normalization of Addition and Subtraction: Generic Normalization

The normalization of addition and subtraction results are more complex than in the previous case. The reason for this is that the result can have any number of

leading zeros, and the shift-amount required for normalization must be calculated. This can be performed in the following manner:

1. The number of leading zeros in the significand result must be counted, while the exponent value must be incremented by one

2. The required shift-amount must be calculated. With the specified register content layout, $(number\ of\ leading\ zeros) - 1$ yields the correct value

3. The significand must be left-shifted by this amount, while the exponent is decreased by the same amount

This will consume three cycles in the current architecture.

### 3.8.3 Normalization of Division

Similar to multiplication, the normalization of division can be simplified due to the constrained range of the result quotient. In this case, the result is in the range $(0.5, 1.99)$. Thus, the two most significant digits of the quotient result is either 01 or 10.

In contrast to the previous operations, the result of the significand division operation is right-aligned in the result register. The shift-amount required for normalization depends on the number of digits in the result, 26 in this implementation.

To summarize, there are two possible normalization operations:

1. If the significand product is in the range $(0.5, 1.0)$, the significand result must be left-shifted by 6 digits, while the exponent is increased by one

2. If the significand product is in the range $[1.0, 2.0)$, the significand result must be left-shifted by 5 digits, while the exponent is left untouched

It should be noted that this approach can be utilized in the case of floating-point square root normalization.

### 3.8.4 Normalization of Integer-to-Floating-Point

As this operations is based on shifting of the significand, the normalization is included in the normal operation. Hence, there is no need for a separate normalization step here.

## 3.9 Rounding

The mathematical theory behind the various rounding operations were described in Ch.2.3. This section will discuss how to actually implement this behavior.

### 3.9.1 Overview

The current *rounding mode* will be chosen by a separate input signal, which requires the rounding mode to be specified for each individual operation. An alternative approach is to keep the active rounding mode in a user-writable register, which is preserved between operations. The latter is probably preferred, as it makes each floating-point instruction more compact. This is however a minor implementation detail.

As the rounding operation takes place after the normalization, it can assume that the normalized significand is located in the same location for every operation. Thus, this stage can be shared among all the operations that require rounding of the result. Just like normalization, the input to this operation will be assumed to be located in general-purpose register R0, in the corresponding register file.

As seen in Ch.2.3, the operation required for rounding is either an addition by $0.5ulp$, an addition by $ulp$ or no modification of the significand at all. Hence, the rounding can be implemented as a function in the control unit which decides the required operation. A problem with rounding is that it may overflow the input significand, thus de-normalizing the value. This is a rare case, yet it must be accounted for. Because of this, it might be necessary to perform a *post-rounding normalization step*. This will simply consist of shifting the significand one place to the right, and increasing the exponent by one.

To make this possible, it is necessary to have an extra bit to the left of the leading one in the significand register, to allow the value to overflow without losing any data. In addition, this allows the need for a post-rounding normalization to be determined by testing the MSB of the significand register, after rounding.

Thus, the significand register contents before rounding is assumed to be:

$$01.\underbrace{xxxxxxxxxxxxxxxxxxxxxxx}_{23bits}\ g\ r\ \underbrace{xxxxx}_{extra\ bits}$$

This format must match the output of the preceding normalization operation, see ch.3.8 for details on this. Note that the last five bits will be used for a *sticky bit* calculation.

### 3.9.2 Generating the Data Required for Rounding

#### *Guard* and *Round*

The *Guard* and *Round* bits were defined in ch.2.3. They are simply additional bits of precision, that can be generated in a very simple manner: as the significand register size is large enough to contain extra precision, both addition, subtraction and multiplication operations get this information for free. The bit-serial algorithms of the division and square root operations simply require two additional iterations, in order to get the required level of precision.

#### The *Sticky Bit*

The sticky bit was also defined in Ch.2.3, however it is slightly more complicated to implement. In general, we must test any bits located to the right of the *guard* and *round* bits, that are discarded during any operation, and see if any of them are high. If so, the *sticky bit* is set high, and kept high for the remainder of the active operation.

There are three places where data is discarded in the suggested architecture:

1. Inside the significand ALU, each time a right-shift is performed
2. In the multiplication chain, in the *shift-and-extend* unit
3. During the rounding operation itself, as no bits to the right of *guard* and *round* are being considered here

Thus, some comparison logic must be appended to these three places. Luckily, the comparison itself is fairly simple: we only need to perform a logical *or* between the discarded bits.

The first case is the most complicated: we need to take the logical *or* between the $n$ lowest bits, in the case of an $n$-bit right-shift. The bits that were shifted out is *or-reduced*, and the resulting bit is transmitted to the control unit. This is currently not implemented in the design, due to insufficient time.

The two next cases are easier to implement: we only need to extract a fixed amount of bits from the multiplier output and the significand ALU output, respectively, and *or-reduce* them. Note that these sticky bit calculations are only valid in certain states, hence the update of the *sticky-bit* register inside the control unit must be state-dependent. Currently, only the last of these two operations is implemented in the actual design, again due to a lack of time.

### 3.9.3 How to Determine the Rounding Operation

Based on Ch.2.3 and [11], the effective rounding operation can be determined through simple combinatorial operations. The determination of the rounding operation depends on the active rounding mode:

#### Round-to-zero (truncate)

No rounding operation is required, only truncate the significand to the required word length.

#### Round-to-plus-infinity

If the number is negative, it can just be truncated. This will effectively round a negative value towards $+\infty$. If the number is positive, we add $ulp$ if $g$, $r$ or $s$ are high, otherwise we truncate.

#### Round-to-minus-infinity

Similar to the previous rounding-mode, apart from the sign. If the number is positive, we truncate it. If the number is negative, we add $ulp$ in the cases where $g$, $r$ or $s$ are high, otherwise we truncate.

#### Round-to-nearest-even

The effective operation is determined according to tab.2.1 on page 9. In most cases we just truncate the answer, in the remaining cases we add $0.5ulp$ to the significand before we truncate to the required word length. Note that this rounding-mode is not implemented in this design, due to insufficient time. According to [11], the addition of $0.5ulp$ is to be performed if the boolean expression $r \cdot (s' + LSB)$ equals 1.

## 3.10 Exception Handling

Due to the many special cases that may arise during floating-point computation — both numerical and logic — it is important to deal with exceptions. Thankfully, most exceptional cases are easy both to detect and treat, given a thorough specification with respect to the standard.

It should be noted that the floating-point exception handling was not prioritized in this project; due to a limited amount of time, the features that contribute to

hardware consumption and clock cycle usage were prioritized. Since the exception handling mostly relies on simple tests in the control unit, the arithmetic operations was given more attention. Thus, not all exceptional cases are handled in the current implementation, yet it should require little effort to extend the design into doing so.

### 3.10.1 Summary of Exceptional Cases

Based on Ch.2.3.6 and the preceding design chapters, this section will summarize the possible exceptional cases.

#### "Init-Time" Exceptions

These exceptions only rely on the inputs to the FPU, thus they are easy to detect at the init stage. The detection and treatment of these exceptions has been implemented in the current design.

**Invalid Operation** Table 3.33 shows the combinations of operation and input that will trigger an *invalid operation* exception.

Table 3.33: Invalid Operations

| Operation | Operand A | Operand B |
|---|---|---|
| Mul | 0 | $\pm\infty$ |
| Mul | $\pm\infty$ | 0 |
| Add | $\pm\infty$ | $\mp\infty$ |
| Sub | $\pm\infty$ | $\pm\infty$ |
| Div | 0 | 0 |
| Div | $\pm\infty$ | $\pm\infty$ |
| Sqrt | $input < -0$ | - |
| fp2int | $NaN$ | - |
| fp2int | $\pm\infty$ | - |

**Division by Zero** This exception only occurs when the user tries to divide any number other than zero, by zero.

**Infinity Arithmetic** Not really an exception, but the treatment of this case is very similar to exception handling. In the standard, infinity arithmetic is assumed to be precise. For instance, $+\infty + 3.0 = +\infty$, with no exceptions triggered. If this was processed as normal in the arithmetic stage, an overflow exception would have occurred. Thus, it is necessary to detect and treat these cases.

**"Run-Time" Exceptions**

These exceptions depend on conditions that occur within the arithmetic stage of a given operation. Hence, it is necessary to include additional logic in order to detect these, as well as memory elements in order to keep track of which exceptions were triggered during the execution of the operation. The assertion of status flags, as well as generating the correct output, is then performed after the operation itself.

These operations require more careful specification and planning than the "init-time" exceptions, and have not been implemented in the current design. As mentioned, it should be easy to extend the design based on the subsequent section, as well as information in the standard itself [4].

**Inexact** This exception is triggered when there is a difference between the rounded and the unrounded result, indicating that there was a loss of precision due to the finite length of the representation format. This exception is triggered fairly often, and is usually ignored. The exception is easy to detect: any time the significand is modified during the rounding step, this exception shall be triggered. Thus it is unnecessary to perform any actual significand comparisons, the exception can be asserted in the corresponding states.

**Overflow** This exception is triggered whenever a result is produced that is too large to represent. For instance, addition of very large numbers may trigger this exception. It can be detected when a number is impossible to normalize, without overflowing the exponent. The output from an overflowing operation is a signed infinity.

**Underflow** Similar to the previous one, except that this occurs whenever a result is too small to represent. An important aspect here is the treatment of denormal values; if denormal values are supported (in accordance with the standard), the underflow exception is not triggered until the denormal representation range is exhausted as well. In implementations that does not deal with denormal numbers, a possible solution is to flush the result to a signed zero, and trigger the exception. It might be convenient to include a separate output flag, that signals that the result would normally end up in the denormal range.

### 3.10.2 Implementation Considerations

In general, the implementation must provide two things when it comes to exceptions:

1. Signal the Corresponding Status Flags

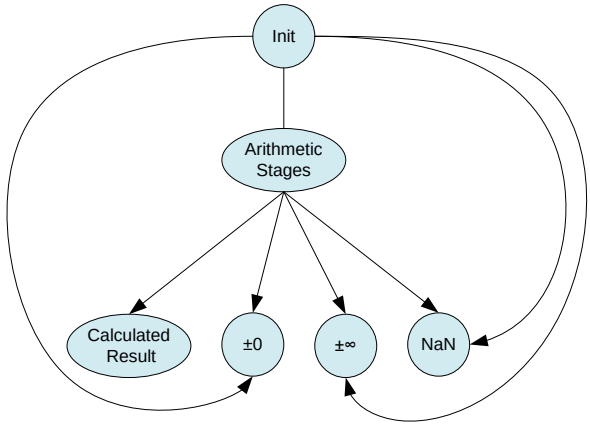


Figure 3.24: Output Nodes: Dealing With Output in Exceptional Cases

2. Generate the Correct Output

The first task is fairly easy, just forward the results generated by the exception detection logic within the control unit. The latter, however, is more complicated. There are several possible outputs from an operation:

- The result generated by the operation
- $\pm 0$
- $\pm \infty$
- $NaN$

Any of these possible output values may be produced together with one or more exception flags. The actual implementation will generate the correct output, by reading the required data from constant registers. For instance, an output of $+\infty$ can be generated by reading 11111111 from the exponent register file and 0 from the significand register file.

This will be organized by separating the different output possibilities into separate nodes in the control graph. Most operations will finish their execution in the "normal result" state, while exceptions may cause the control flow to end up in a different node, such as "output infinity". Exceptions that are detected at init-time will bypass the arithmetic stage, and go directly to the corresponding output node. The concept is illustrated in Fig.3.24.

## 3.11 The Final Design

This section will present the final architecture, which is a combination of the different architectures presented in the previous chapters. An overview of what is included in the design will be given, along with a list of requirements that is not yet fulfilled.

### 3.11.1 Organization

The organization is pretty much the union between all the previous architectures, and included all the functionality required for performing the specified operations. Figure 3.25 shows a block diagram of the architecture.

The final architecture looks less clean than the individual ones, mainly due to the various multiplexers that are needed to route different pieces data during different operations.

### 3.11.2 Control

The control unit is implemented as a simple state machine, with a few internal registers. The state machine will encapsulate the arithmetic stage control paths that were described in the previous chapters. Figure 3.26 shows the state diagram of the final control unit. The double-lined nodes symbolize a encapsulated control path: the node corresponds to a small sub-graph. Please refer to the chapters on individual operation design for details on the encapsulated states.

The exception handling is controlled according to Ch.3.10.

Table 3.34 shows the specification of all the inter-module control signals present in the final architecture. These values are generated by the control unit, and forwarded to the respective units. Refer to Fig.3.25 for details on how these signals are interconnected.

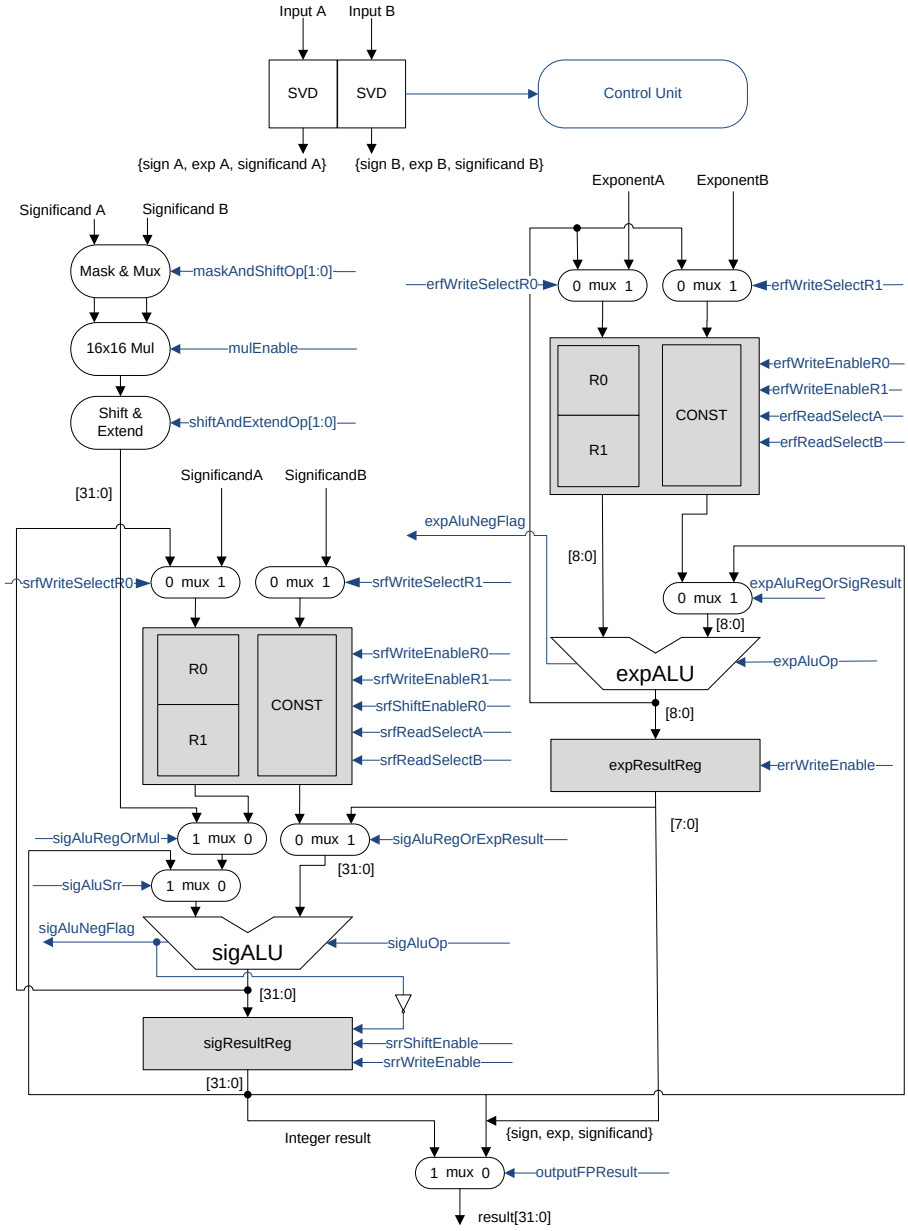


Figure 3.25: The Final FPU Data Path

Table 3.34: Floating-Point Unit - Control Signal Specification

| Signal | Width | Default | Description |
|---|---|---|---|
| readFPInput | 1 | 1 | Interpret the input as floating-point or integer? |
| maskAndShiftOp | 2 | 00 | Determines how to slice and shift the multiplier input |
| mulEnable | 1 | 0 | Enable multiplier? |
| shiftAndExtendOp | 2 | 00 | Determines how to shift and extend the multiplier output |
| srfWriteSelectR0 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteSelectR1 | 1 | 0 | Muxes between new input and the significand ALU result |
| srfWriteEnableR0 | 1 | 0 | Update significand register R0? |
| srfWriteEnableR1 | 1 | 0 | Update significand register R1? |
| srfShiftEnableR0 | 1 | 0 | Shift significand register one digit to the left? |
| srfReadSelectA | 4 | 0000 | Chooses which value to output on significand register file read port A |
| srfReadSelectB | 4 | 0001 | Chooses which value to output on srf read port B |
| sigAluRegOrMul | 1 | 0 | Forward srf read port A, or the multiplier output to the ALU? |
| sigAluSrr | 1 | 0 | Forward the result of the decision on the line above, or the SRR value to the significand ALU? |
| sigAluRegOrExpResult | 1 | 0 | Forward SRF read port B, or the ERR value to the significand ALU? |
| sigAluOp | 4 | 0000 | Significand ALU OpCode |
| srrWriteEnable | 1 | 0 | Update the significand result register (SRR)? |
| srrShiftEnable | 1 | 0 | Shift the significand result register one digit to the left? |
| erfWriteSelectR0 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteSelectR1 | 1 | 0 | Muxes between new input and the exponent ALU result |
| erfWriteEnableR0 | 1 | 0 | Update exponent register R0? |
| erfWriteEnableR1 | 1 | 0 | Update exponent register R1? |
| erfReadSelectA | 3 | 000 | Chooses which value to output on exponent register file read port A |
| erfReadSelectB | 3 | 001 | Chooses which value to output on exponent register file read port B |
| expAluRegOrSigResult | 1 | 0 | Forward SRF read port B, or the SRR value to the significand ALU? |
| expAluOp | 3 | 000 | Exponent ALU OpCode |
| errWriteEnable | 1 | 0 | Update the exponent result register (ERR)? |
| outputFPResult | 1 | 1 | Generate a floating-point result, or forward the entire significand result register as output? |
| resultReady | 1 | 0 | Flag that a result is ready, and the unit is ready for a new operation |

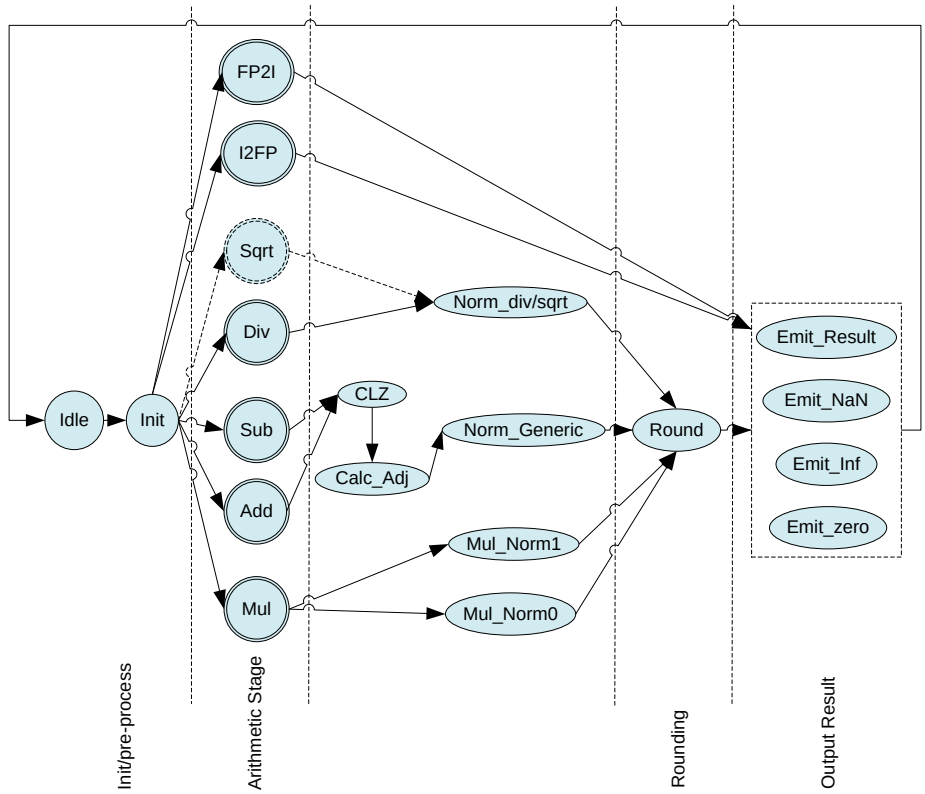


Figure 3.26: The Final FPU State Chart

# Chapter 4

# Simulation and Verification

This chapter will feature both functional simulations of the design, as well as some notes on how the implementation has been verified.

## 4.1 Simulation

This section will provide several functional simulations of the design. The purpose of this section is to underline some important aspects of the theory and algorithms presented in the previous chapters, as well as demonstrating that the implementation is in accordance with the specification.

### 4.1.1 Simulation of Functional Units

The functional units that are utilized in the architecture is relatively standard; arithmetic-logic units and register files are pretty much "textbook designs"[13]. Still, a few simulations are included, both to demonstrate a few quirks such as the shifting of significand register R0, and for the sake of completeness.

#### Register Files

As the register files utilized in the design share many aspects, only a simulation of the significand register file is shown here (Fig.4.1).

1. The input values are written to internal register R0 and R1
2. Register R0 is shifted one digit to the left

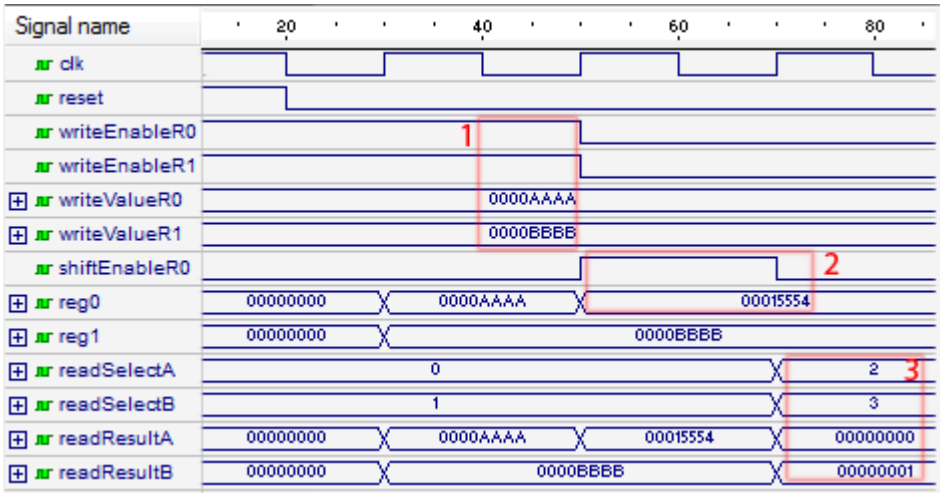


Figure 4.1: Significand Register File

3. The read select signals choose constant register *zero* and *one*, which are transmitted to the output ports

### Arithmetic-Logic Units

The arithmetic-logic units featured in the design are mostly similar to each other, the only major difference is the native word size, and the fact that some operations has been left out of the exponent ALU. Due to the similarities, only a simulation run of the significand ALU is given here (Fig.4.2).

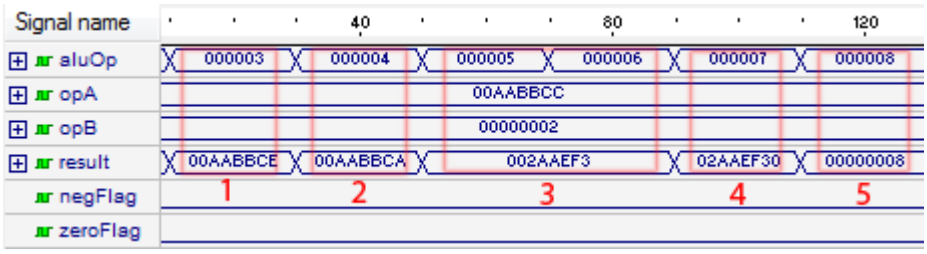


Figure 4.2: Significand ALU

1. Addition
2. Subtraction

3. Right-shifts: arithmetic and logic
4. Left-shift
5. Count-leading zeros

### 4.1.2 Simulation of Individual Operations

This section will show several simulations of individual operations. The purpose of this is to illustrate the manner of operation for the included instructions, as well as demonstrate that the design is indeed implemented according to the given algorithms. Hence, this chapter should be studied along with the design considerations presented in Ch.3.

To maintain readability, the simulation runs are split into several different figures. In general, the exponent and significand calculations will be shown separately, as they are mostly independent from each other.

#### Floating-Point Multiplication

**Exponent Calculation** Figure 4.3 shows how the input exponents are added, and the summation result properly biased.

1. The new exponents are read from the input port, and placed in the exponent register file
2. The exponents are added together, and the addition result is placed in exponent register R0
3. *bias* is subtracted from the exponent sum, in order to obtain the correct exponent. The result is again stored in exponent register R0

**Partial Product Calculation** Figure 4.4 shows how the input significands are sliced, multiplied and shifted in order to create a single, partial product.

1. The input fractions are sliced and extended, according to *maskAndShiftOp*
2. The sliced input is fed to the multiplier, which emits a partial product two cycles later
3. The partial product is shifted and zero-padded, according to *shiftAndExtendOp*

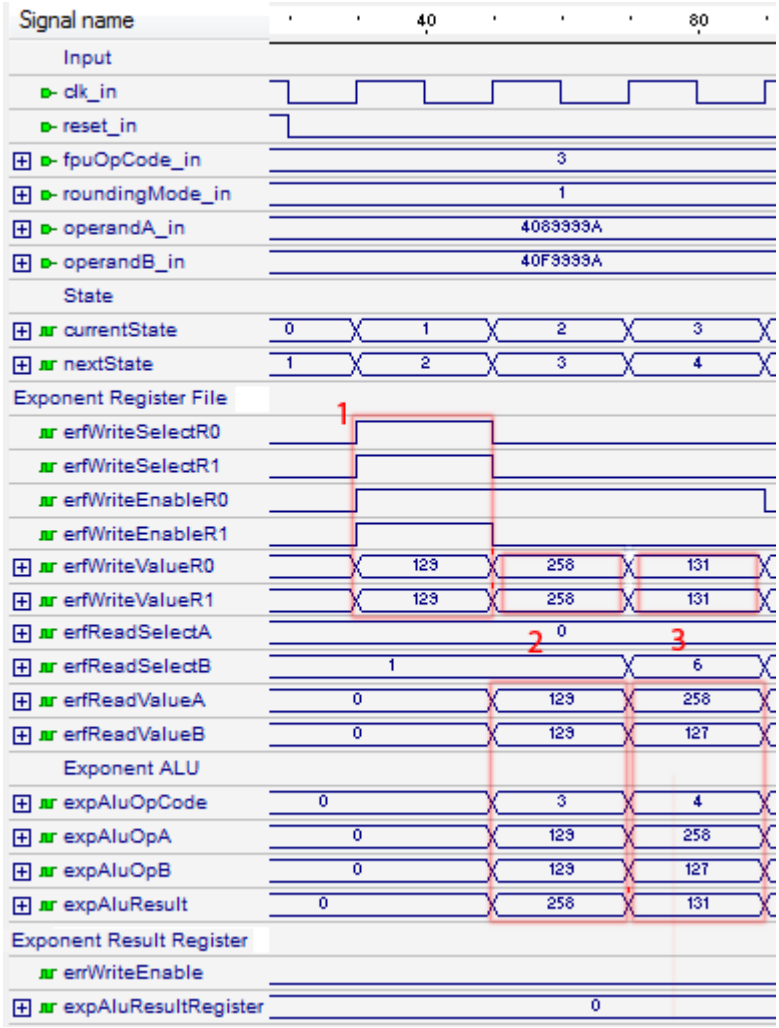


Figure 4.3: Floating-Point Multiplication: Exponent Calculation

**Partial Product Accumulation** The accumulation of in total four partial products is illustrated in fig.4.5.

1. ALU input A is fed data from the multiplier chain
2. The first partial product is moved through the ALU, the three last partial products are accumulated
3. The ALU results are stored in significand register R0

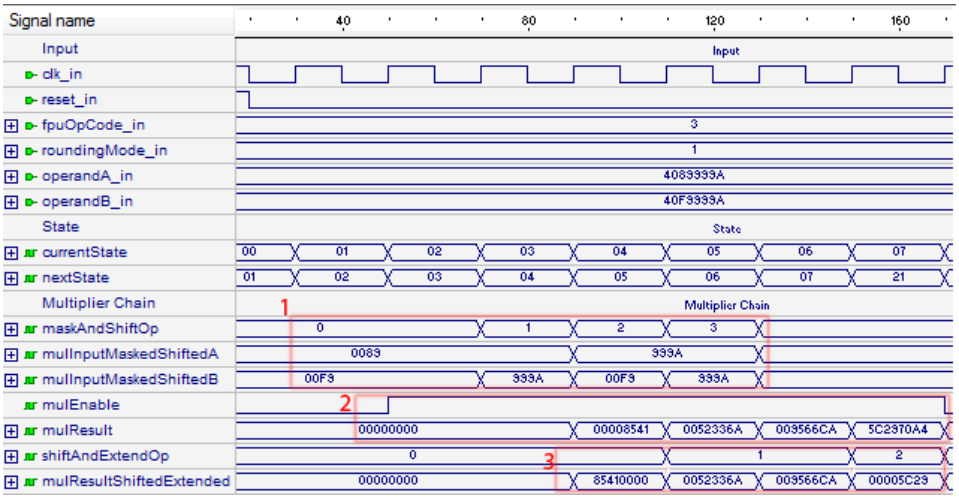


Figure 4.4: Floating-Point Multiplication: Partial Product Calculation

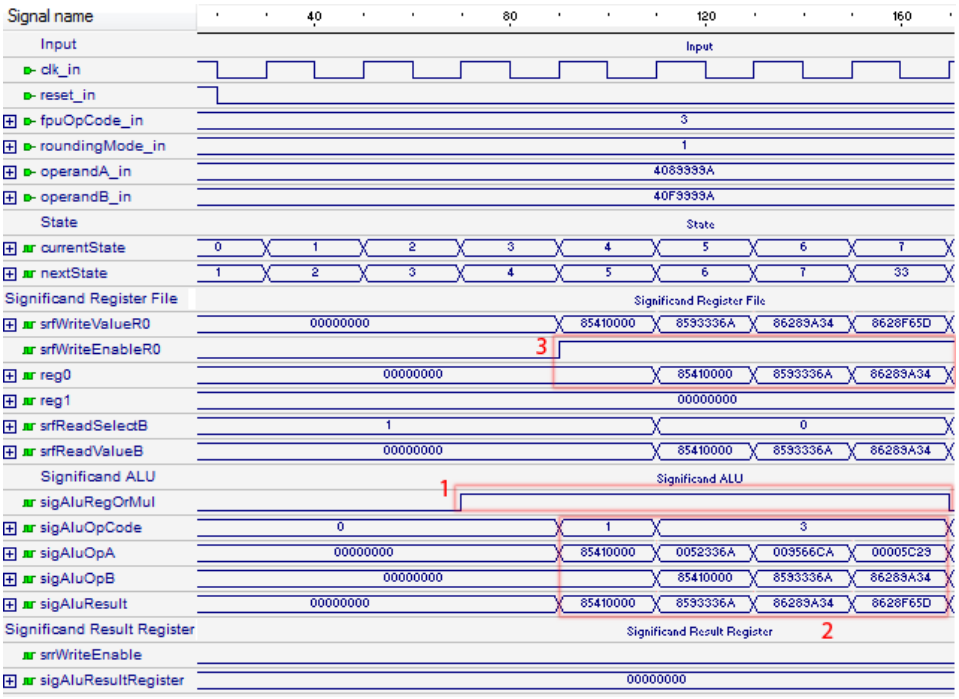


Figure 4.5: Floating-Point Multiplication: Partial Product Accumulation

**Floating-Point Addition and Subtraction: $E_A \geq E_B$, effective addition**

**Exponent Calculation** Figure 4.6 shows how the exponents are compared, and the exponent difference is relayed to the significand pipeline. In this case $E_A$ is larger than $E_B$, and $E_A$ is kept as the result exponent.

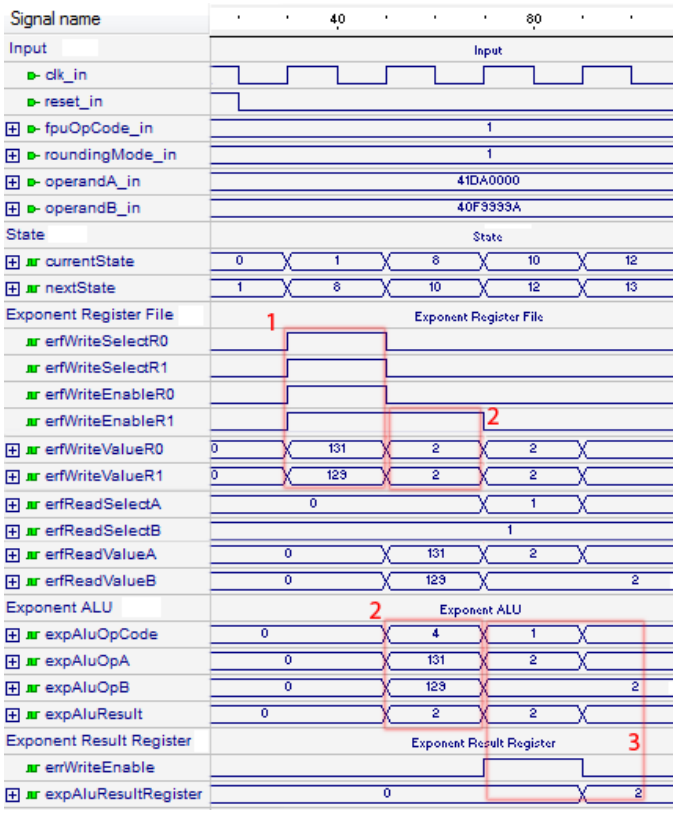


Figure 4.6: Floating-Point Addition: Exponent Calculation

1. The input exponents are read into the exponent register file
2. $E_B$ is subtracted from $E_A$, the difference is stored in exponent register R1
3. The exponent difference is moved to the exponent result register, thus made available to the significand ALU

**Significand Calculation** Figure 4.7 illustrates how the input is read into the corresponding registers, and how $F_B$ is adjusted in order to prepare the signifi-

cands for the subsequent addition. The values are then added and prepared for normalization.

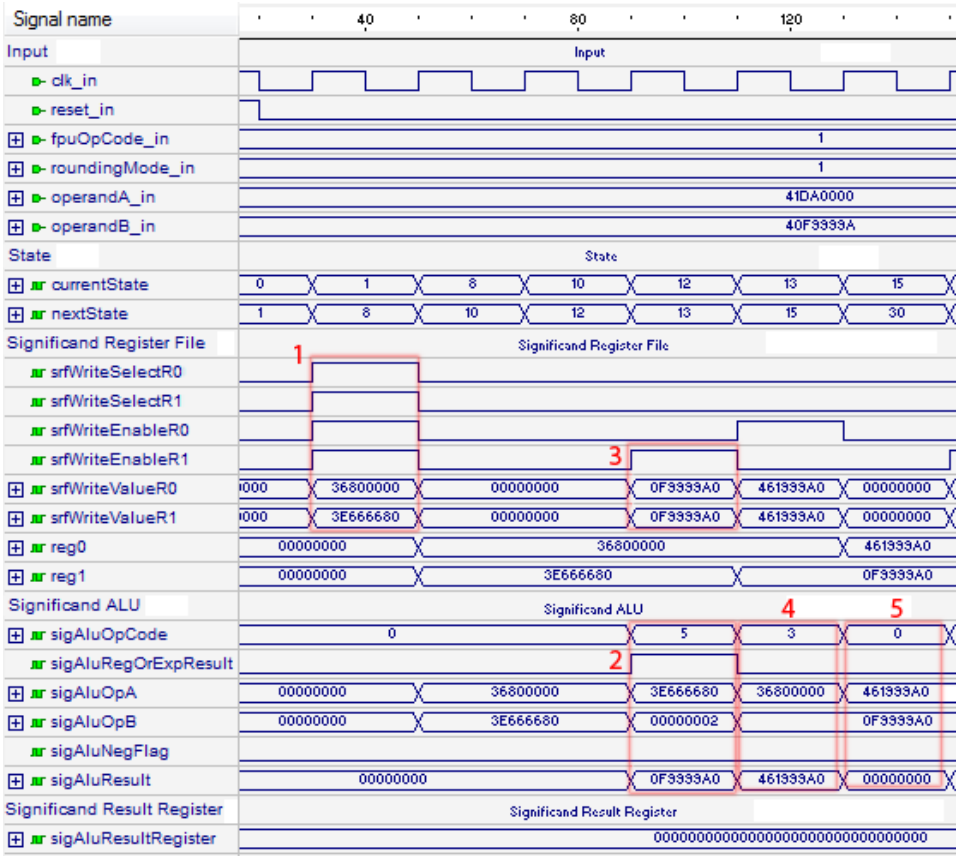


Figure 4.7: Floating-Point Addition: Significand Calculation

1. The input significands are read into the significand register file
2. Input value B has the smallest exponent, thus it is right-shifted by the absolute exponent difference
3. The adjusted significand is written back into its corresponding register
4. The adjusted significands are added, the result is stored in significand register R0
5. As the addition yielded a positive result, no negation operation is required here

**Floating-Point Addition and Subtraction: $E_A < E_B$, effective subtraction**

**Exponent Calculation** In this case, $E_B$ is larger than $E_A$. This requires the exponent comparison to include a negation. Again, the absolute difference between the input exponents is transmitted to the significand ALU. (Fig.4.8)

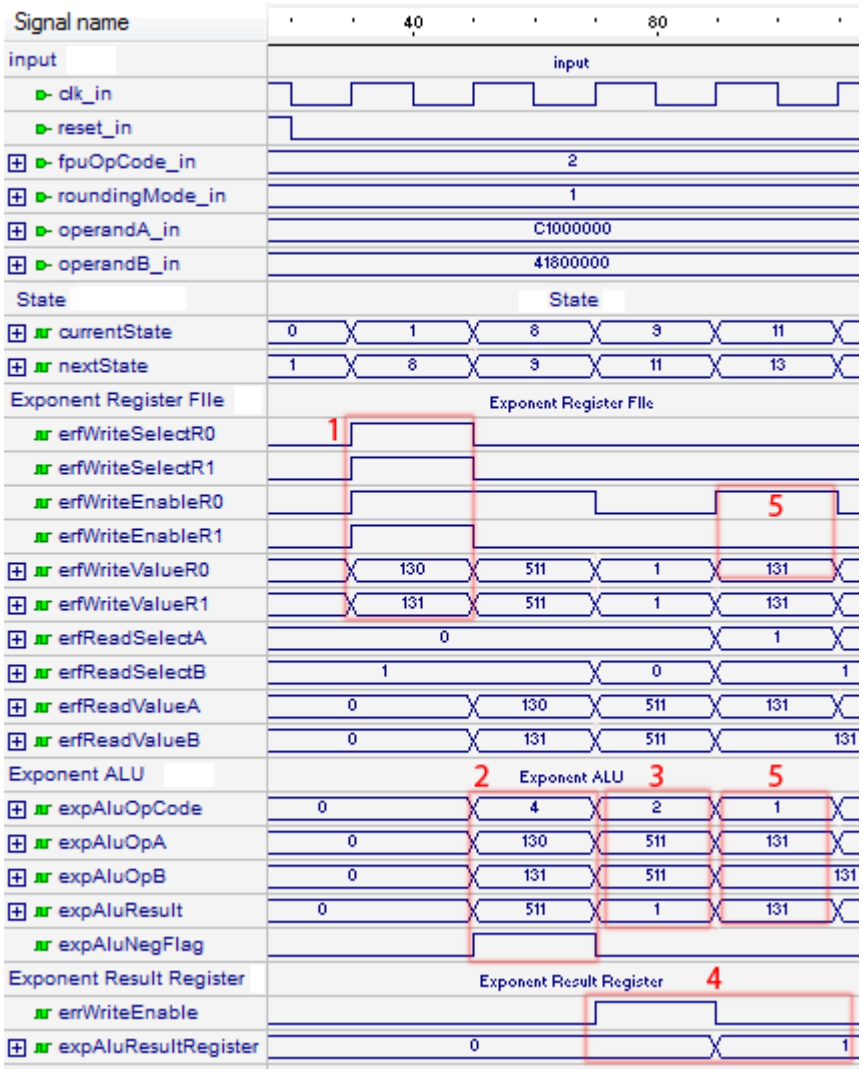


Figure 4.8: Floating-Point Subtraction: Exponent Calculation

1. The input exponents are read into the exponent register file

2. $E_B$ is subtracted from $E_A$, the difference is stored in exponent register R0
3. As the previous subtraction yielded a negative result, the contents of exponent register R0 is negated
4. The absolute difference between the exponents are stored in the exponent result register, thus made available to the significand ALU
5. The final exponent value is moved to exponent register R0, in order to be ready for the subsequent normalization operation

**Significand Calculation** In Fig.4.9, the effective operation is subtraction. In addition, the sign of the first operand requires it to be negated. Again, the significand belonging to the smallest exponent is right-shifted by the absolute exponent difference.

1. The input significands are read into the significand register file
2. $E_A$ is negated, as it's sign bit is set to high and the effective operation is $-E_A - E_B$
3. $E_A$ is right-shifted by the absolute exponent difference calculated by the exponent ALU
4. $E_B$ is subtracted from $E_A$
5. As the subtraction yielded a negative result, the value is negated in order to convert it to a sign-magnitude representation
6. All adjustments of $E_A$ as well as the summation itself, is stored in significand register R0

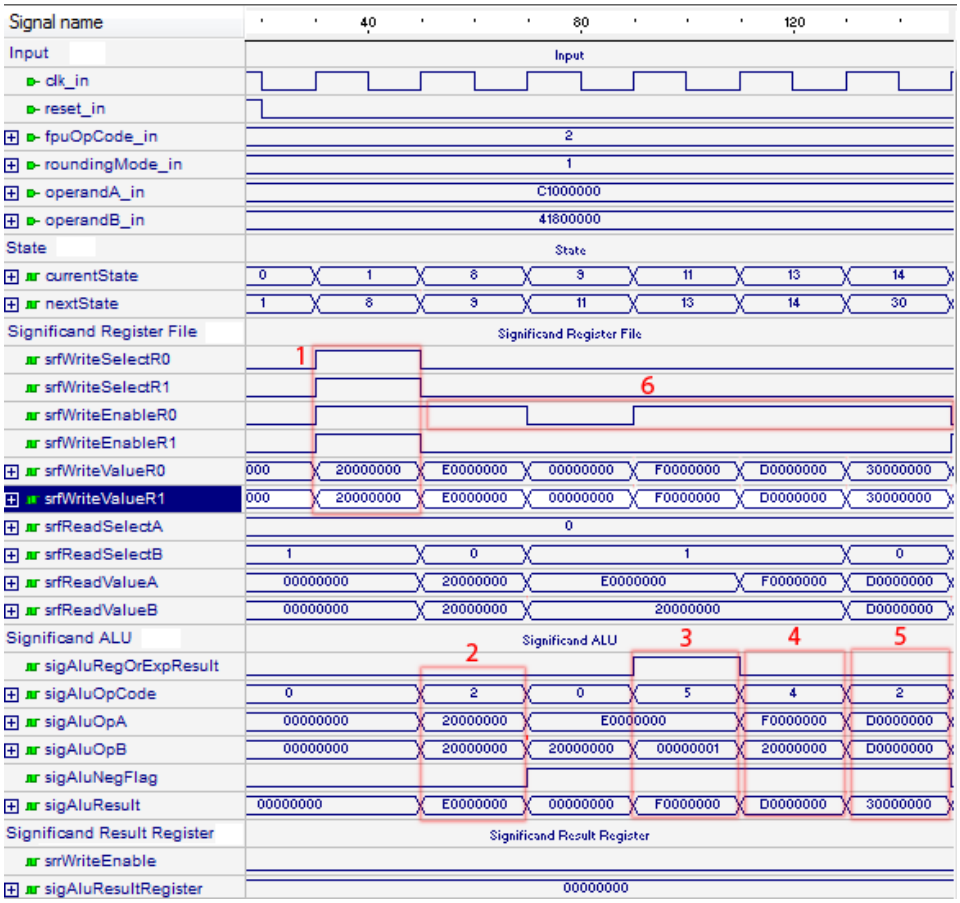


Figure 4.9: Floating-Point Subtraction: Significand Calculation

### Floating-Point Division

**Exponent Calculation** The treatment of the exponents in the case of floating-point division — illustrated in Fig.4.10 — is very similar to the one featured in the floating-point multiplication operation.

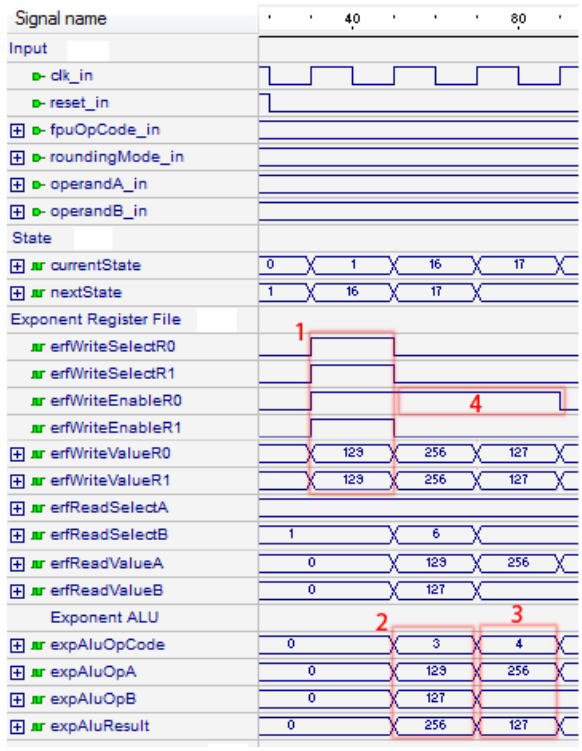


Figure 4.10: Floating-Point Division: Exponent Calculation

1. The input exponents are read into the exponent register file
2. *bias* added to $E_A$
3. $E_B$ is subtracted from the previous sum
4. All calculations are written back to exponent register R0

**Significand Calculation** The significand division is performed by 26 subsequent subtractions. A few of these iterations are shown in fig.4.11.

1. The iteration counter controls the number of iterations

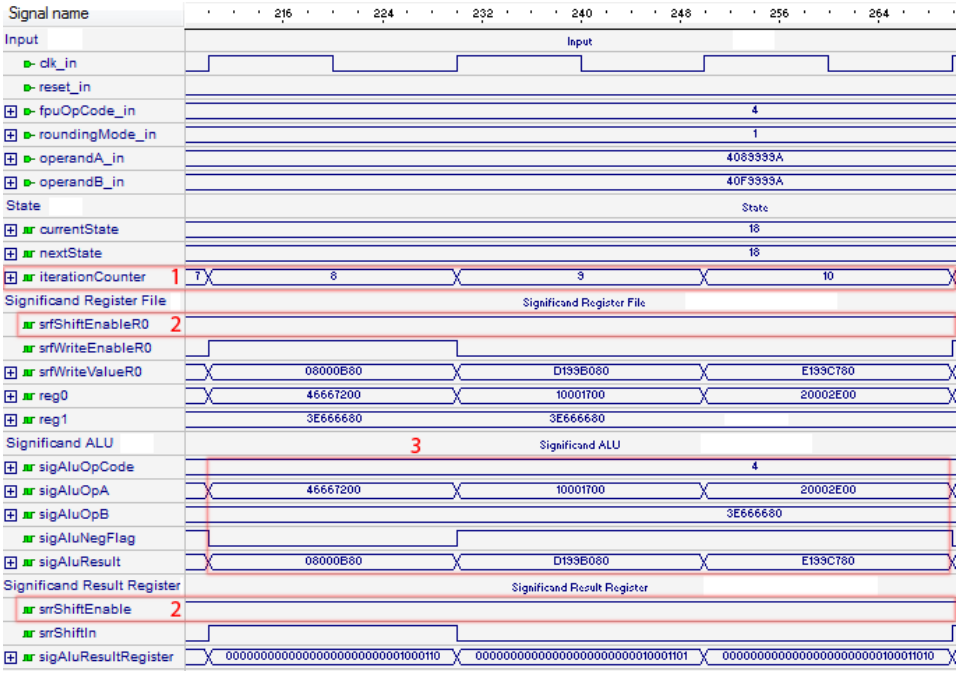


Figure 4.11: Floating-Point Division: Significand Calculation

2. Both significand register R0 and the significand result register are shifted one digit to the left each cycle

3. The significand ALU is performing one subtraction per iteration. The resulting sign from this operation determines whether of not the partial remainder is updated (4), as well as the next quotient digit (5)

### Floating-Point to Integer: Positive Input

**Calculation of Shift Amount** Figure 4.12 and Fig.4.13 shows the conversion of a positive floating-point value to an integer.

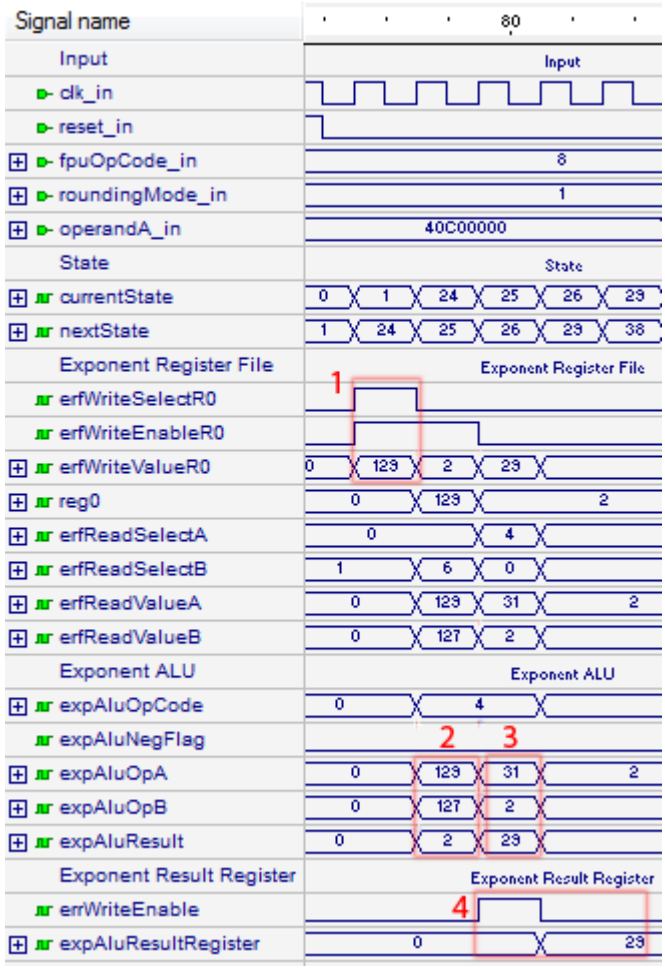


Figure 4.12: Positive Floating-Point to Signed Integer - Calculation of Shift Amount

1. The input exponent is read into exponent register R0

2. *bias* is subtracted from the input exponent, the result is the true input exponent

3. The true exponent value is subtracted from the radix point position (31), yielding the required significand shift amount

4. The significand shift amount is stored in the exponent result register, thus made available to the significand ALU

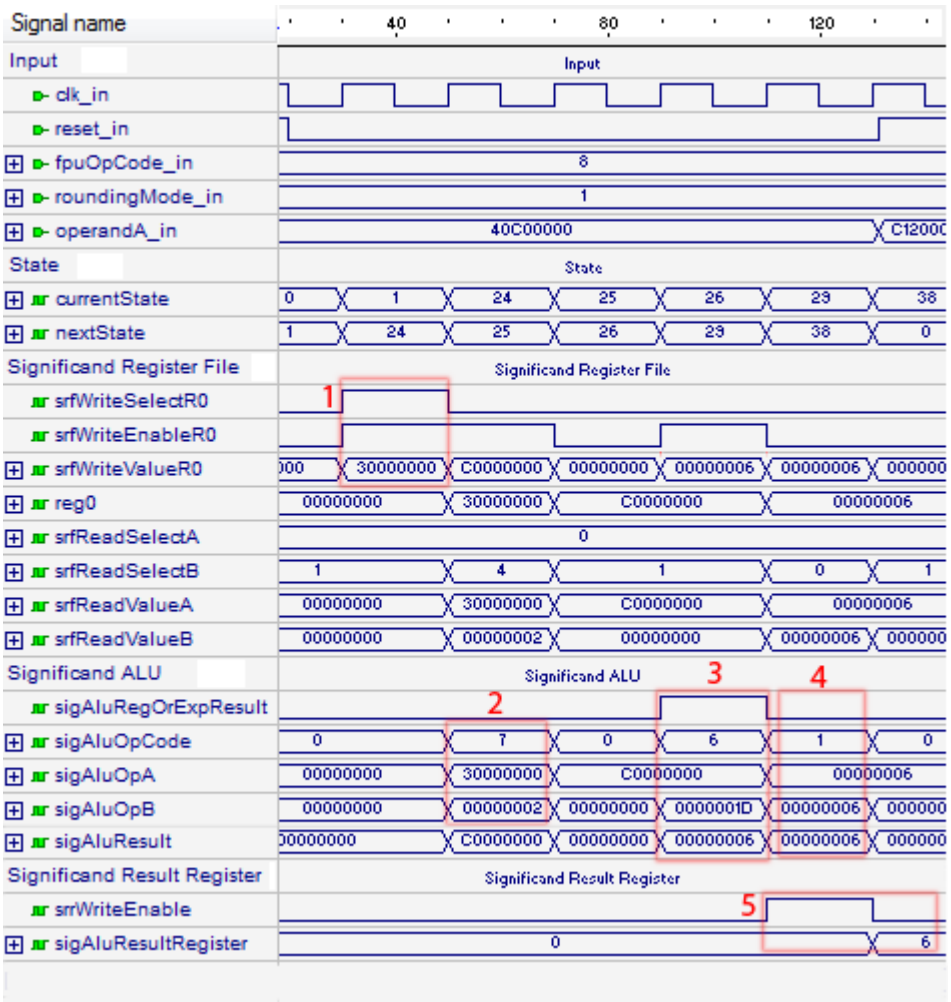


Figure 4.13: Positive Floating-Point to Signed Integer - Significand Adjustment

### Adjustment of Significand

1. The input significand is stored in significand register R0

2. The input significand is left-aligned in its register, by shifting it two places to the left

3. The significand shift amount is read from the exponent result register, and the significand is right-shifted

4. As the input was positive, there is no need for converting the integer result into a negative two's compliment representation

5. The integer result is written to the significand result register

#### Floating-Point to Integer: Negative Input

Figure 4.14 and Fig.4.15 shows how a negative floating-point value is converted to a signed integer.

#### Calculation of Shift Amount

1. The input exponent is read into exponent register R0

2. *bias* is subtracted from the input exponent, the result is the true input exponent

3. The true exponent value is subtracted from the radix point position (31), yielding the required significand shift amount

4. The significand shift amount is stored in the exponent result register, thus made available to the significand ALU

#### Adjustment of Significand

1. The input significand is stored in significand register R0

2. The input significand is left-aligned in its register, by shifting it two places to the left

3. The significand shift amount is read from the exponent result register, and the significand is right-shifted

4. As the input was negative, the integer value is negated in order to convert it into a negative two's compliment representation

5. The integer result is written to the significand result register

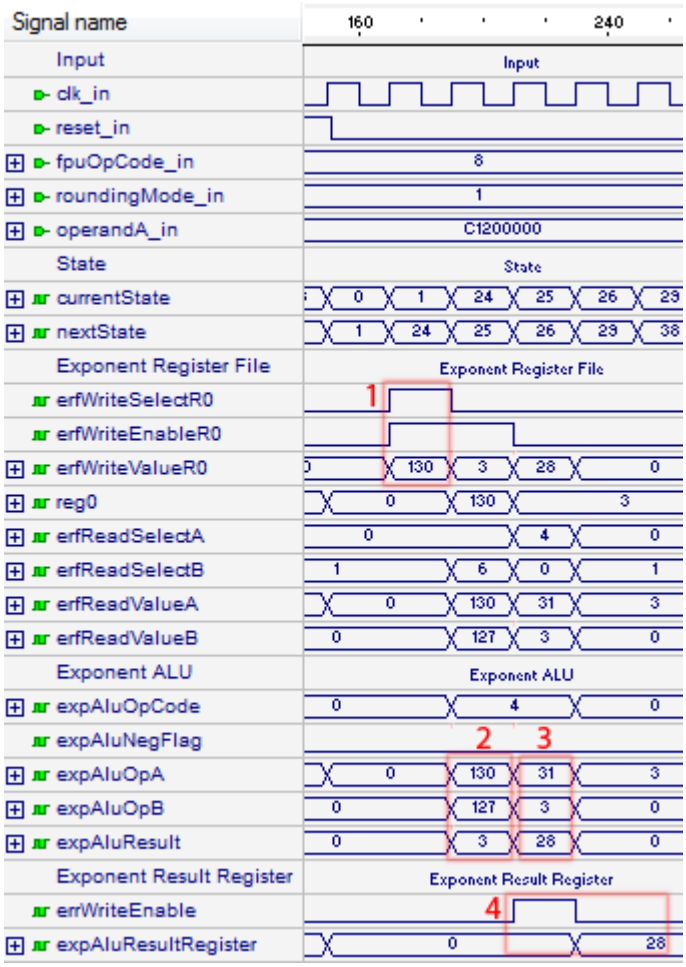


Figure 4.14: Negative Floating-Point to Signed Integer - Calculation of Shift Amount

### Unsigned Integer to Floating-Point

**Calculation of Exponent Value** Figure 4.16 and Fig.4.17 shows the conversion from an unsigned integer to a floating-point value.

1. The input integer is stored in significand register R0

2. The number of leading zeros in the input integer is counted, the amount is stored in significand register R1

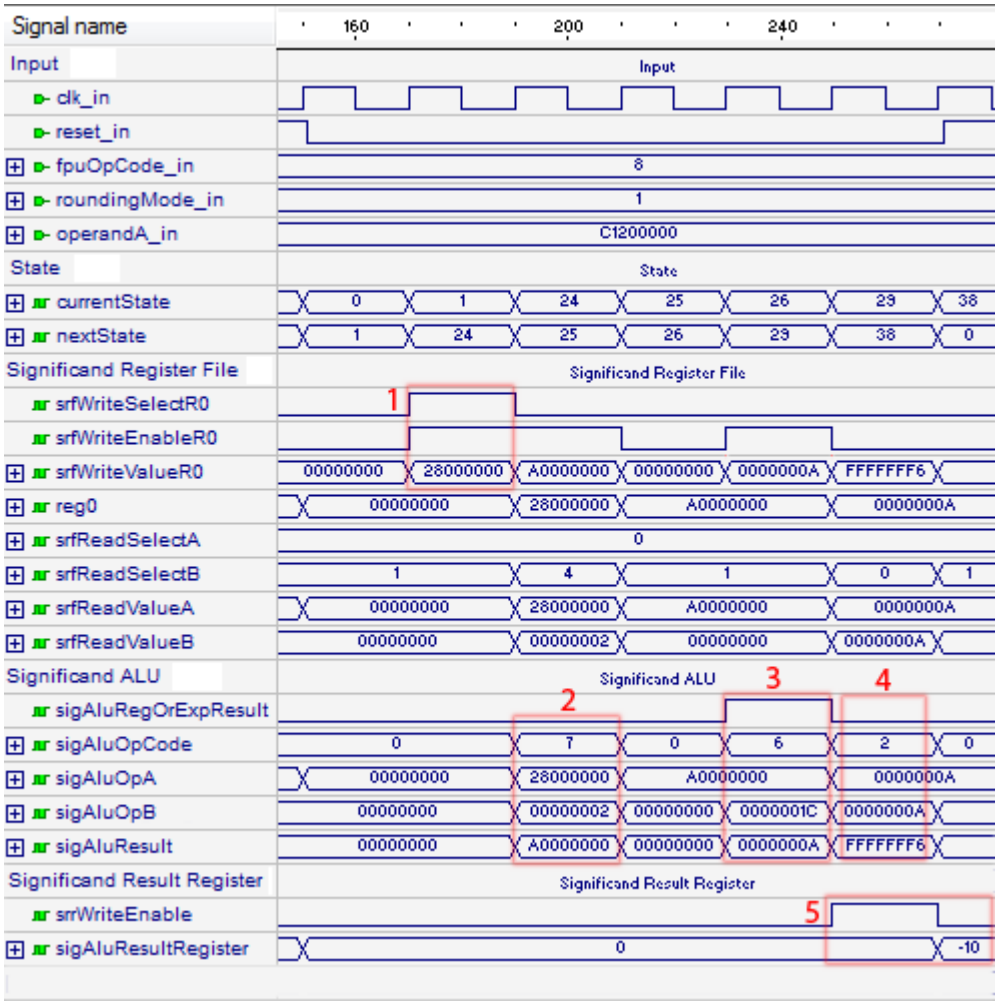


Figure 4.15: Negative Floating-Point to Signed Integer - Significand Adjustment

3. The number of leading zeros is stored in the significand result register, in order to make it available to the exponent ALU

4. The input integer is left-shifted, according to the number of leading zeros in the value

5. The shifted integer is right-shifted one digit, in order to conform with the register layout assumed by the subsequent rounding operation

6. The final significand value is written to the significand result register

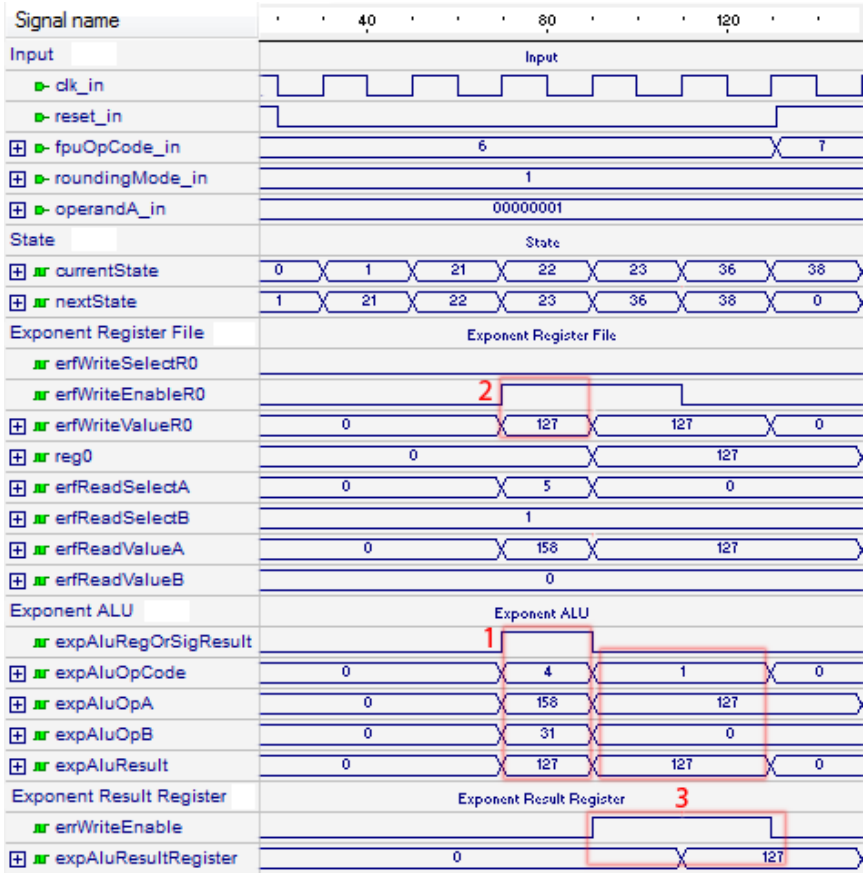


Figure 4.16: Unsigned Integer to Floating-Point: Calculation of Exponent Value

### Adjustment of Significand

1. The number of leading zeros in the input integer is read from the significand result register, and added to *bias*
2. The exponent value is written to exponent register R0
3. No adjustment of the exponent is performed during the rounding step, and the value is written to the exponent result register

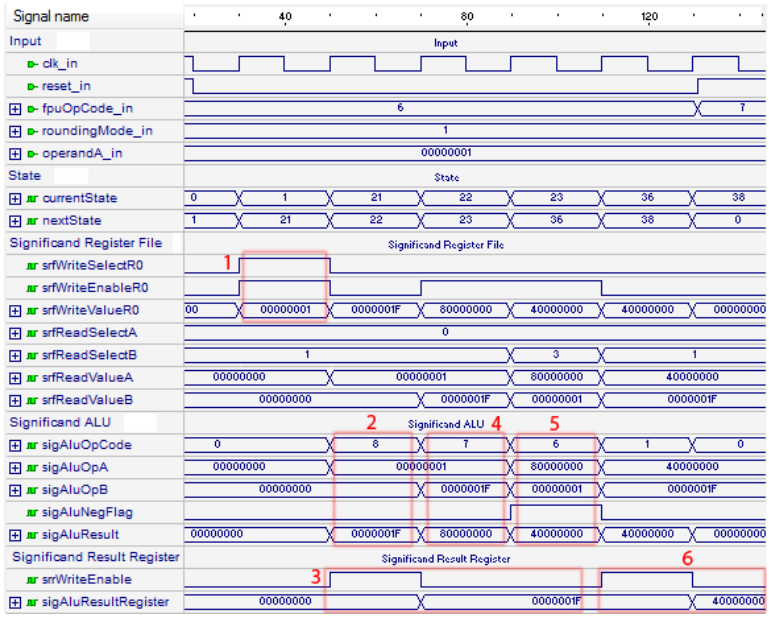


Figure 4.17: Unsigned Integer to Floating-Point: Adjustment of Significand

### 4.1.3 System-Level Simulation

This section features a few top-level simulation runs, demonstrating only the signals that are available to the user of the FPU itself.

#### System-Level Simulation of a Binary Operation

Figure 4.18 shows the system-level simulation of a multiplication instruction. Both input values are normalized values, none of them belong to the class of special representation values. Note how none of the exception flags are asserted.

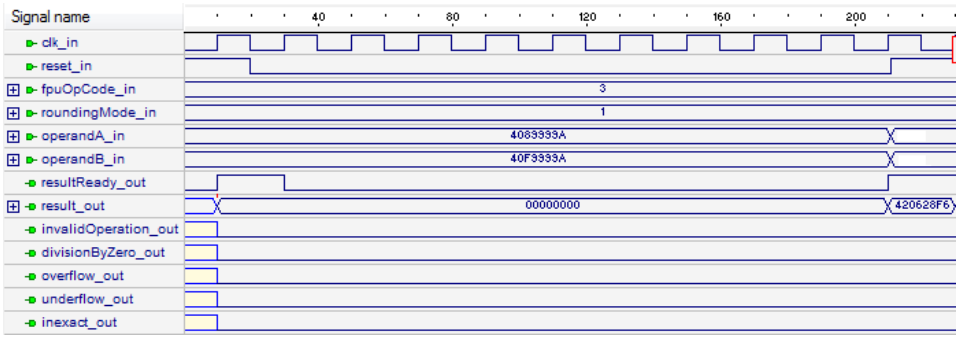


Figure 4.18: System-Level Simulation: Floating-Point Multiplication

#### System-Level Simulation of a Unary Operation

The conversion operations are unary, hence they take one input and produce one output. In this simulation run (Fig.4.19), only operand A affects the result. None of the assertion flags are asserted, and the integer result is correctly transmitted to the output port.

#### System-Level Simulation of an Invalid Operation

In this simulation (Fig.4.20), an invalid operation is performed. Note how the *invalid operation* flag is asserted, and a $NaN$ output value is generated.

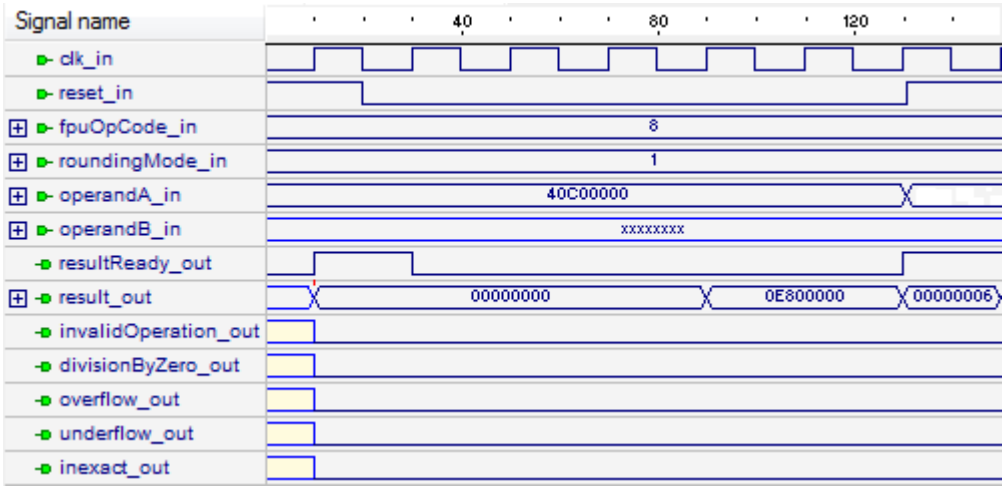


Figure 4.19: System-Level Simulation: Floating-Point to Signed Integer Conversion

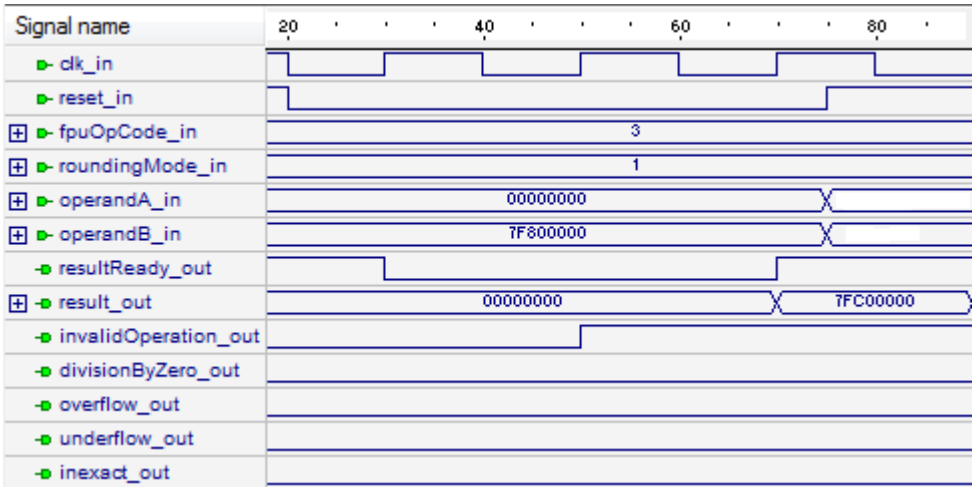


Figure 4.20: System-Level Simulation: Invalid Operation ($0 \times \infty$)

## 4.2 Verification

This section will give an overview of how the design was verified, along with some suggestions on further verification. It should be noted that the design is by no means completely tested.

### 4.2.1 Automated test benches

This design has been tested by running a set of pseudo-random test vectors through a simulation model of the entire module. A complete coverage of all input combinations, operations and rounding modes is clearly impossible. however, the number of test vectors used indicate that the implemented operations perform their tasks in accordance with both the specification and the standard itself.

The test vectors were generated using a set of C-programs, compiled with GCC. These programs are rather simple, most generate two floating-point numbers, perform an operation between them, and write both the operands and the result to a text file. The validity of these tests depend on whether or not the floating-point implementation used conforms to the IEEE-754 1985.

An example C-program for generating test vectors is listed in A.2.

### 4.2.2 Testing of Special Cases

As mentioned before, floating-point operations have several special cases. Some examples are exceptional cases, and infinity arithmetic. In order to verify that an implementation does indeed conform with the standard, it is necessary to test this behavior.

As a complete verification is beyond the scope of this project, only some simple tests have been done. These simulation runs are based on Tab.3.33 on page 67, and were inspected manually. The correct exception flags were generated, and the correct answer was produced. This indicates that the exception handling that is implemented works as specified. No figures of these simulation runs are included here, as they required too much space.

### 4.2.3 Suggestions for Future Testing and Verification

First, the number of test-vectors used in the automatic test-benches should be increased dramatically, in order to increase the coverage of the tests. A good tool for extensive testing, as well as test vector generation, is SoftFloat [9] . If the FPU is connected to a generic microprocessor capable of running C-programs, the program TestFloat [10] may be utilized as well.

As some functionality is missing from this implementation, several aspects of IEEE conformance is yet untested. This includes both the square root operations, as well as the different rounding modes. The treatment of quiet NaNs and denormal numbers is still not handled, thus the verification of this behavior must be delayed until the implementation is made more mature.

A complete verification should also perform more testing of internal signals during the various operations, not only checking input against output. For instance, all control signals should be tested in each state featured in the control unit, and checked against the specification.

Finally, *formal verification* may prove useful in the search for a thoroughly tested design. Information on this is provided in [7] and [8].

# Chapter 5

# Results

This chapter contains performance analysis results and area estimates in the form of gate counts. The results are discussed in the subsequent chapter.

## 5.1 Area Consumption

### 5.1.1 FPGA Synthesis

Even though the design is targeted at an implementation in custom silicon, an FPGA synthesis was performed. The motivation behind this was to obtain more comparable data about the hardware consumption of the design. The synthesis was performed using Altera Quartus II v.9.1 web edition. The target platform was set to Altera Cyclone I - EP1C6Q240C6, with "optimize area" as a parameter to the synthesis tool.

These settings were chosen to match the ones presented in [12] (available through `www.opencores.org` [1]), as this report presents the design of an non-pipelined FPU with FPGA synthesis results. The other design is a more traditional FPU architecture, with separate arithmetic units for each operation. Thus, it is interesting to see whether or not the architecture proposed in this project provides any significant savings in terms of hardware consumption.

Figure 5.1 shows the synthesis results for this implementation. Note that the logic cells used by the external multiplier can be subtracted from the results, as this unit is assumed to be present already.

In comparison, [12] gives the total logic cell count of the unit as 3468, not counting the square-root pipeline. More details on the logic cell consumption of the solution presented is given in Tab.5.1.

| Compilation Hierarchy Node | Logic Cells |
|---|---|
| ⊟ \|fpu_top | 1924 (308) |
| \|ExpRegisterFile:expRegisterFile\| | 33 (33) |
| \|ExponentALU:expALU\| | 87 (87) |
| ⊟ \|ExternalMul16x16:externalMul\| | 430 (67) |
| ⊟ \|lpm_mult:Mult0\| | 363 (0) |
| \|mult_21t:auto_generated\| | 363 (363) |
| \|FpuControlUnit:controlUnit\| | 159 (159) |
| \|MaskAndShift:maskAndShift\| | 34 (34) |
| \|SVDUnit:svdUnitA\| | 14 (14) |
| \|SVDUnit:svdUnitB\| | 14 (14) |
| \|ShiftAndExtend:shiftAndExtend\| | 13 (13) |
| \|SigRegisterFile:sigRegisterFile\| | 134 (134) |
| \|SignificandALU:sigALU\| | 698 (698) |

Figure 5.1: FPGA Synthesis Results

Table 5.1: Logic Cell Usage - Other Implementation [12]

| Unit | Logic Cell Count |
|---|---|
| Addition Unit | 684 |
| Multiplication Unit | 1530 |
| Division Unit | 928 |
| Square-Root Unit | 919 |
| Top-level | 326 |
| Total | 4387 |

### 5.1.2 Gate-Level Synthesis

The design was synthesized to a netlist, as the logic gate count of the design was underlined as one of the most important figures of merit for this project. The synthesis tool used was Synopsis DesignCompiler v.2009.06-SP4, the specified process technology was an Atmel CMOS process.

The resulting gate counts are listed in tab.5.2. Note that the Significand Result Register as well as the Exponent Result Register were implemented directly in the top module. Thus, the top-level gate count includes all submodules, two registers and a lot of interconnects.

An important aspect of actual area consumption is the interconnects. This has not been estimated in this project, thus the given area metrics should only be interpreted as estimates.

Table 5.2: Area Consumption by Module

| Unit | Gate Count |
|---|---|
| Top-level (total) | 3271 |
| Control unit | 77 |
| Exponent ALU | 232 |
| Exponent register file | 188 |
| Mask-and-shift unit | 53 |
| Shift-and-extend unit | 80 |
| Significand ALU | 1283 |
| Significand register file | 580 |

## 5.2 Performance

A crucial aspect of any hardware solution is whether or not it provides a large enough speedup over a software implementation to justify the increased hardware cost. Thus, the performance analysis must be studied together with the area estimates, in order to assess the quality of the design.

Table 5.3 lists the clock cycle consumption for all the implemented operations, along with the estimates made in the preliminary project as well as the clock cycle consumption of the existing software implementation.

The column labeled *normal* indicated the clock cycle consumption of a normal operation: no special-case inputs and no need for post-rounding normalization. The *worst-case* column indicated the worst possible clock-cycle consumption the operation can have, this typically includes operations that require a post-rounding normalization of the result.

The three columns labeled *estimated*, *SW (GCC)* and *SW (IAR)* are adapted from the preliminary report [5] They indicate the estimated FPU clock cycle consumption, as well as the clock cycle usage of the existing software implementations provided by the GCC and the IAR compiler, respectively. The final column presents the clock cycle consumption of the solution presented in [12]. *Note that the square root operation is not implemented in this design, thus no performance data is available.

If we define the *speedup* as in Eq.5.1, we can calculate the speedup over the existing software solutions. The results are presented in Fig.5.2. The estimated speedup as predicted in the preliminary report is included as well.

$$Speedup_{HW,SW} = \frac{Cycle\ usage, SW}{Cycle\ usage, HW} \tag{5.1}$$

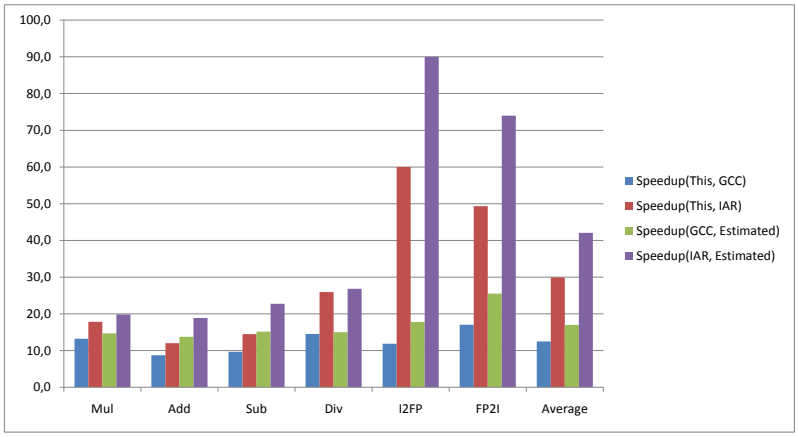


Figure 5.2: Speedup of a Hardware Implementation vs. Software Implementations

Table 5.3: Clock Cycle Usage by Operation

| Operation | Normal | Worst-Case | Estimated | SW (GC) | SW (IAR) | [12] |
|---|---|---|---|---|---|---|
| Mul | 10 | 11 | 9 | 132 | 178 | 12 |
| Add | 11 | 12 | 7 | 96 | 132 | 7 |
| Sub | 11 | 12 | 7 | 106 | 159 | 7 |
| Div | 30 | 31 | 29 | 435 | 778 | 35 |
| Sqrt* | - | - | 30 | 480 | 2612 | 35 |
| UI2FP | 5 | 5 | 4 | 71 | 360 | - |
| I2FP | 6 | 7 | 4 | 71 | 360 | - |
| FP2SI | 6 | 6 | 4 | 102 | 296 | - |

# Chapter 6

# Discussion and Conclusion

## 6.1 Discussion of Results

This discussion is based on the results presented in the previous chapter, the featured simulations, and the design presented in the preceding sections.

All the specified operations were implemented successfully, apart from the floating-point square root and the conversion from a floating-point value to an integer-valued floating-point number. The implemented operations seem to function, which has been verified through simulations and test-benches. Some basic exception handling is in place, though this part is yet a bit incomplete.

As for the results, the synthesis results gave some promising numbers. The area usage of the FPGA synthesis shows that this implementation is roughly half the size of an alternative implementation. This suggests that the architecture presented here is capable of reducing the area consumption of a floating-point unit by a substantial amount, compared to a more traditional architecture.

The disadvantage of the solution presented, however, is the large amount of structural hazards in the design itself. Thus, it is very unsuitable for any kind of pipelining, which may limit the performance. Another potential issue is the maximum clock frequency the design can achieve, this has not been derived in this project.

The performance analysis shows that the estimated performance of the design was a little too optimistic: this is mainly due to the fact that normalization and rounding are performed as ALU operations, not as dedicated logic. This typically increases the clock cycle usage of each operation by a few cycles, which explain the difference in estimated results, and real results. The speedup over a pure software-implementation is still quite good; a dedicated FPU can give a speedup of between 13x and 30x, depending on the compiler being used.

## 6.2 Future Work

The most important feature that is missing from this implementation is the square root operation. However, this can be added to the design at a relatively low cost as outlined in Ch.3.6. In addition to this, exception handling is somewhat incomplete, especially when it comes to verification. This will, however, not introduce much of an increase in neither area consumption nor execution time of the various operations.

Regarding synthesis, much work is left. This includes both detailed analysis of synthesis results, as well as tailoring the implementation to the synthesis tool being used. This may provide a more compact solution, in terms of gate count. The amount of interconnects and the final size of the implementation should be derived, and compared to other solutions.

## 6.3 Conclusion

This report has shown that a functional floating-point unit can be realized in a compact manner, by exploiting reuse of functional units, as well as simple and functionally similar algorithms. Such a design can achieve a significant speedup over a pure software implementation of the IEEE-754 1985 standard, at a low cost-penalty.

# Bibliography


[1] A resource for development and publications of free (LGPL) hardware core designs. http://www.opencores.org.

[2] Atmel, San Jose, USA. *AT90USB1287, 8-bit AVR Microcontroller with 64/128K Bytes of ISP Flash and USB Controller*, 2009.

[3] Atmel, San Jose, USA. *AVR Quick Reference Guide*, 2009.

[4] IEEE computer society. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages –, Aug 1985.

[5] Daniel Hornæs. Low-Cost FPU. December 2009.

[6] David Goldberg. What every computer scientist should know about floating-point arithmetic, 1991. http://docs.sun.com/source/806-3568/ncg_goldberg.html.

[7] John Harrison. Formal verification of floating point trigonometric functions. *Lecture Notes in Computer Science*, 2004.

[8] John Harrison. Floating-point verification using theorem proving. *Lecture Notes in Computer Science*, 2006.

[9] John Hauser. SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html.

[10] John Hauser. TestFloat. http://www.jhauser.us/arithmetic/TestFloat.html.

[11] Israel Koren. *Computer Arithmetic Algorithms, 2nd Edition.* A.K. Peters LTD, 2002.

[12] Jidan Al Eryani. Floating Point Unit. Technical report, Vienna University of Technology, 2006. http://opencores.org/project,fpu100.

[13] John L. Hennessy and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 2007.

[14] Yamin Li and Wanming Chu. Implementation of single precision floating point square root on fpgas. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:226, 1997.

# Appendix A

# Appendix

## A.1 Verilog Implementation Code

This appendix containes all the Verilog source code of the synthesizable implementation.

The included files are:

1. global.v: various constants used by most modules
2. sig_register_file.v: the significand register file
3. exp_register_file.v: the exponent register file
4. SVD_unit.v: the special value detection unit
5. mask_and_shift.v: the unit that slices and extends the input to the external multiplier
6. external_mul_16x16.v: behavioral model of the external 16-bit multiplier
7. shift_and_extend: the unit which shifts and zero-extends the multiplier output
8. significand_alu.v: the significand ALU
9. exponent_alu.v: the exponent ALU
10. fpu_control_unit.v: the complete FPU control unit
11. fpu_top.v: the top-level module of the design. Instantiates the other modules, as well as the exponent and significand result registers

Listing A.1: global.v

```
//
// FORMAT SPECIFICATIONS
//
`define FORMAT_WORD_WIDTH  32
`define FORMAT_FRAC_WIDTH  23
`define FORMAT_EXP_WIDTH   8

//
// FPU INSTRUCTIONS
//
`define FPU_INSTR_NOP   4'b0000
`define FPU_INSTR_ADD   4'b0001
`define FPU_INSTR_SUB   4'b0010
`define FPU_INSTR_MUL   4'b0011
`define FPU_INSTR_DIV   4'b0100
`define FPU_INSTR_SQRT  4'b0101
`define FPU_INSTR_UI2FP 4'b0110
`define FPU_INSTR_SI2FP 4'b0111
`define FPU_INSTR_FP2SI 4'b1000

//
// FPU ROUNDING MODES
//
`define ROUNDING_MODE_NEAREST_EVEN  2'b00
`define ROUNDING_MODE_TRUNCATE      2'b01
`define ROUNDING_MODE_POS_INF       2'b10
`define ROUNDING_MODE_NEG_INF       2'b11

//
// Significand register file port names
//
`define SRF_REG_R0      4'b0000
`define SRF_REG_R1      4'b0001
`define SRF_REG_ZERO        4'b0010
`define SRF_REG_ONE     4'b0011
`define SRF_REG_TWO     4'b0100
`define SRF_REG_ULP_ROUND   4'b0101
`define SRF_REG_BIAS        4'b0110
`define SRF_REG_FIVE        4'b0111
`define SRF_REG_SIX     4'b1000
`define SRF_REG_NANSIG      4'b1001
`define SRF_REG_ONES        4'b1111

//
// Exponent register file port names
//
`define ERF_REG_R0      3'b000
`define ERF_REG_R1      3'b001
`define ERF_REG_ZERO        3'b010
`define ERF_REG_ONE     3'b011
`define ERF_REG_RPP     3'b100
`define ERF_REG_I2FP        3'b101
`define ERF_REG_BIAS        3'b110
`define ERF_REG_ONES        3'b111

//
```

```
57 // Significand ALU opcodes
58 //
59 `define SIG_ALU_OP_NOP  4'b0000 //no operation
60 `define SIG_ALU_OP_MOVA 4'b0001 //move op.A
61 `define SIG_ALU_OP_NEGB 4'b0010 //negate op.B
62 `define SIG_ALU_OP_ADD  4'b0011 //add op.A and op.B
63 `define SIG_ALU_OP_SUB  4'b0100 //subtract op.B from op.A
64 `define SIG_ALU_OP_SHRA 4'b0101 //arithmetic right-shift of operand A,
       by "op.B" digits
65 `define SIG_ALU_OP_SHRL 4'b0110 //logic right-shift of op.A, by "op.B"
       digits
66 `define SIG_ALU_OP_SHLL 4'b0111 //logic left-shift of op.A, by "op.B"
       digits
67 `define SIG_ALU_OP_CLZ  4'b1000 //count leading zeroes of op.A
68
69 //
70 // Exponent ALU opcodes
71 //
72 `define EXP_ALU_OP_NOP  3'b000
73 `define EXP_ALU_OP_MOVA 3'b001
74 `define EXP_ALU_OP_NEGB 3'b010
75 `define EXP_ALU_OP_ADD  3'b011
76 `define EXP_ALU_OP_SUB  3'b100
77 `define EXP_ALU_OP_SHL  3'b101
78
79
80 //
81 // Mask-And-Shift unit opcodes
82 //
83 `define MASK_AND_SHIFT_A8C8 2'b00
84 `define MASK_AND_SHIFT_A8D16    2'b01
85 `define MASK_AND_SHIFT_B16C8    2'b10
86 `define MASK_AND_SHIFT_B16D16   2'b11
87
88 //
89 // Shift-And_extend unit opcodes
90 //
91 `define SHIFT_16_BIT_AND_EXTEND 2'b00
92 `define SHIFT_0_BIT_AND_EXTEND  2'b01
93 `define SHIFT_TRUNC_AND_EXTEND  2'b10
```

Listing A.2: sig_register_file.v

```
1  `timescale 1ns/1ps
2
3  `include "global.v"
4
5  module SigRegisterFile(clk_in, reset_in, writeEnableR0_in,
       writeEnableR1_in, writeValueR0_in, writeValueR1_in,
6                         shiftEnableR0_in,
7                         readSelectA_in, readSelectB_in,
                              readResultA_out, readResultB_out);
8
9      //default register width
10     parameter REGISTER_WIDTH = 'd32;
11
12     //default constant register contents
```

```
13      parameter CONST0_VALUE = 32'd0;  //zero - required for exception
            handling
14      parameter CONST1_VALUE = 32'd1;  //one - required!
15      parameter CONST2_VALUE = 32'd2;  //two - required!
16      parameter CONST3_VALUE = 32'd128;//ULP for rounding - required!
17      parameter CONST4_VALUE = 32'd127;//BIAS - required!
18      parameter CONST5_VALUE = 32'd5;  //used for normalization of div/
            sqrt
19      parameter CONST6_VALUE = 32'd6;  //used for normalization of div/
            sqrt
20      parameter CONST7_VALUE = 32'h20000000; //significand value
            corresponding to a NaN result
21      parameter CONST8_VALUE = 32'hffffffff; //all ones
22
23  //PORTS
24      input clk_in, reset_in;
25      input writeEnableR0_in, writeEnableR1_in;
26      input [REGISTER_WIDTH-1:0] writeValueR0_in, writeValueR1_in;
27      input shiftEnableR0_in;
28      input [3:0] readSelectA_in, readSelectB_in;
29      output reg [REGISTER_WIDTH-1:0] readResultA_out, readResultB_out;
30
31  //INTERNAL REGISTERS
32      //GPR
33      reg [REGISTER_WIDTH-1:0] reg0, reg1;
34
35      always @(readSelectA_in, readSelectB_in, reg0, reg1) begin
36          case (readSelectA_in)
37              4'b0000: readResultA_out = reg0;
38              4'b0001: readResultA_out = reg1;
39              4'b0010: readResultA_out = CONST0_VALUE;
40              4'b0011: readResultA_out = CONST1_VALUE;
41              4'b0100: readResultA_out = CONST2_VALUE;
42              4'b0101: readResultA_out = CONST3_VALUE;
43              4'b0110: readResultA_out = CONST4_VALUE;
44              4'b0111: readResultA_out = CONST5_VALUE;
45              4'b1000: readResultA_out = CONST6_VALUE;
46              4'b1001: readResultA_out = CONST7_VALUE;
47              4'b1111: readResultA_out = CONST8_VALUE;
48              default: begin
49                  readResultA_out = 0;
50              end
51          endcase
52
53          case (readSelectB_in)
54              4'b0000: readResultB_out = reg0;
55              4'b0001: readResultB_out = reg1;
56              4'b0010: readResultB_out = CONST0_VALUE;
57              4'b0011: readResultB_out = CONST1_VALUE;
58              4'b0100: readResultB_out = CONST2_VALUE;
59              4'b0101: readResultB_out = CONST3_VALUE;
60              4'b0110: readResultB_out = CONST4_VALUE;
61              4'b0111: readResultB_out = CONST5_VALUE;
62              4'b1000: readResultB_out = CONST6_VALUE;
63              4'b1001: readResultB_out = CONST7_VALUE;
64              4'b1111: readResultB_out = CONST8_VALUE;
65              default: begin
```

```
66
67                  readResultB_out = 0;
68              end
69          endcase
70      end
71
72
73      always @(posedge clk_in) begin
74          if (reset_in == 1'b1) begin // reset registers?
75              reg0 <= 0;
76              reg1 <= 0;
77          end else begin
78              //update reg0?
79              if (writeEnableR0_in) reg0 = writeValueR0_in;
80
81              if (shiftEnableR0_in) reg0 = reg0 << 1;
82
83               //update reg1?
84              if (writeEnableR1_in == 1'b1) reg1 <= writeValueR1_in;
85          end
86      end
87
88  endmodule
```

Listing A.3: exp_register_file.v

```
1   `timescale 1ns/1ps
2
3   `include "global.v"
4
5   module ExpRegisterFile(clk_in, reset_in, writeEnableR0_in,
        writeEnableR1_in, writeValueR0_in, writeValueR1_in,
6                          readSelectA_in, readSelectB_in,
                               readResultA_out, readResultB_out);
7
8       //default register width
9       parameter REGISTER_WIDTH = 'd9;
10
11      //default constant register contents
12      parameter CONST0_VALUE = 9'd0;   //zero
13      parameter CONST1_VALUE = 9'd1;   //one
14      parameter CONST2_VALUE = 9'd31;  //radix point position
15      parameter CONST3_VALUE = 9'd158; //exponent const used by int2fp
16      parameter CONST4_VALUE = 9'd127; //BIAS
17      parameter CONST5_VALUE = 9'd511; //all ones
18
19  //PORTS
20      input clk_in, reset_in;
21      input writeEnableR0_in, writeEnableR1_in;
22      input [REGISTER_WIDTH-1:0] writeValueR0_in, writeValueR1_in;
23      input [2:0] readSelectA_in, readSelectB_in;
24      output reg [REGISTER_WIDTH-1:0] readResultA_out, readResultB_out;
25
26  //INTERNAL REGISTERS
27      //GPR
28      reg [REGISTER_WIDTH-1:0] reg0, reg1;
29
```

```
30      always @(readSelectA_in, readSelectB_in, reg0, reg1) begin
31          case (readSelectA_in)
32              3'b000: readResultA_out = reg0;
33              3'b001: readResultA_out = reg1;
34              3'b010: readResultA_out = CONST0_VALUE;
35              3'b011: readResultA_out = CONST1_VALUE;
36              3'b100: readResultA_out = CONST2_VALUE;
37              3'b101: readResultA_out = CONST3_VALUE;
38              3'b110: readResultA_out = CONST4_VALUE;
39              3'b111: readResultA_out = CONST5_VALUE;
40          endcase
41
42          case (readSelectB_in)
43              3'b000: readResultB_out = reg0;
44              3'b001: readResultB_out = reg1;
45              3'b010: readResultB_out = CONST0_VALUE;
46              3'b011: readResultB_out = CONST1_VALUE;
47              3'b100: readResultB_out = CONST2_VALUE;
48              3'b101: readResultB_out = CONST3_VALUE;
49              3'b110: readResultB_out = CONST4_VALUE;
50              3'b111: readResultB_out = CONST5_VALUE;
51          endcase
52      end
53
54
55      always @(posedge clk_in) begin
56          if (reset_in) begin // reset registers?
57              reg0 <= 0;
58              reg1 <= 0;
59          end else begin
60              //update reg0?
61              if (writeEnableR0_in) reg0 <= writeValueR0_in;
62
63               //update reg1?
64              if (writeEnableR1_in) reg1 <= writeValueR1_in;
65          end
66      end
67  endmodule
```

Listing A.4: svd_unit.v

```
1   `timescale 1ns/1ps
2
3   `include "global.v"
4
5   module SVDUnit(operand_in, sign_out, isZero_out, isInf_out, isNan_out,
        isDenorm_out, operand_out);
6   // I/O PORTS
7       input [`FORMAT_WORD_WIDTH-1:0] operand_in;
8       output reg sign_out, isZero_out, isInf_out, isNan_out,
            isDenorm_out;
9       output reg [32:0] operand_out; //operand with leading significand
            bit included
10
11  //INTERNAL REGISTERS
12      reg sign;
13      reg [`FORMAT_EXP_WIDTH-1:0] exp;
```

```
14      reg [`FORMAT_FRAC_WIDTH-1:0] frac;
15
16      reg expIsMax, expIsNonZero, fracIsNonZero;
17
18      always @(operand_in) begin
19          //decompose input
20          sign = operand_in[`FORMAT_WORD_WIDTH-1]; //[31]
21          exp    = operand_in[(`FORMAT_WORD_WIDTH-2):(`FORMAT_WORD_WIDTH-
                `FORMAT_EXP_WIDTH-1)]; // [30:23]
22          frac   = operand_in[(`FORMAT_FRAC_WIDTH-1):0]; // [22:0]
23
24          // &-reduction / |-reduction
25          expIsMax = &(exp);
26          expIsNonZero = |(exp);
27          fracIsNonZero = |(frac);
28
29          //generate output
30          sign_out      = sign;
31          isZero_out    = ~expIsMax & ~expIsNonZero & ~fracIsNonZero;
32          isInf_out     = expIsMax & (~fracIsNonZero);
33          isNan_out     = expIsMax & fracIsNonZero;
34          isDenorm_out  = ~expIsMax & ~expIsNonZero & fracIsNonZero;
35          operand_out = (isDenorm_out) ? {sign, exp, 1'b0, frac} : {sign
                , exp, 1'b1, frac};
36    end
37  endmodule
```

Listing A.5: mask_and_shift.v

```
1   `timescale 1ns/1ps
2
3   `include "global.v"
4
5   module MaskAndShift(maskAndShiftSelect_in, operandA_in, operandB_in,
        operandA_out, operandB_out);
6       input [1:0] maskAndShiftSelect_in;
7       input [`FORMAT_FRAC_WIDTH:0] operandA_in, operandB_in; //23:0
8       output reg [15:0] operandA_out, operandB_out; //15:0
9
10      always @(maskAndShiftSelect_in, operandA_in, operandB_in) begin
11          case (maskAndShiftSelect_in)
12              `MASK_AND_SHIFT_A8C8: begin
13                  operandA_out <= {8'b0, operandA_in[23:16]};
14                  operandB_out <= {8'b0, operandB_in[23:16]};
15              end
16
17              `MASK_AND_SHIFT_A8D16: begin
18                  operandA_out <= {8'b0, operandA_in[23:16]};
19                  operandB_out <= operandB_in[15:0];
20              end
21
22              `MASK_AND_SHIFT_B16C8: begin
23                  operandA_out <= {operandA_in[15:0]};
24                  operandB_out <= {8'b0, operandB_in[23:16]};
25              end
26
27              `MASK_AND_SHIFT_B16D16: begin
```

```
                operandA_out <= operandA_in[15:0];
                operandB_out <= operandB_in[15:0];
            end
        endcase
    end
endmodule
```

Listing A.6: external_mul_16x16.v

```
`timescale 1ns/1ps

// Behavioral model of a 16x16 integer multiplier, two-stage pipeline, 32 bit result
module ExternalMul16x16(clk_in, reset_in, mulEnable, operandA_in, operandB_in, product_out);
    input clk_in, reset_in, mulEnable;
    input [15:0] operandA_in, operandB_in;
    output[31:0] product_out;

    reg [31:0] delay_reg, product_reg;

    assign product_out = product_reg;

    always @(posedge clk_in) begin
        if (reset_in == 1'b1) begin
            delay_reg   <= 32'b0;
            product_reg <= 32'b0;
        end else begin
            if (mulEnable) begin
                delay_reg   <= operandA_in * operandB_in;
                product_reg <= delay_reg;
            end else begin
                delay_reg <= delay_reg;
                product_reg <= product_reg;
            end
        end
    end

endmodule
```

Listing A.7: shift_and_extend.v

```
`timescale 1ns/1ps

`include "global.v"

module ShiftAndExtend(shiftAndExtendSelect_in, operand_in, operand_out, stickyBit_out);
    input [1:0] shiftAndExtendSelect_in;
    input [31:0] operand_in;
    output reg [31:0] operand_out; //32 bits
    output reg stickyBit_out;

    always @(shiftAndExtendSelect_in, operand_in) begin
        stickyBit_out = 1'b0;
        case (shiftAndExtendSelect_in)
```

```
14              `SHIFT_16_BIT_AND_EXTEND:    operand_out <= {operand_in
                    [15:0], 16'b0};
15              `SHIFT_0_BIT_AND_EXTEND:     operand_out <= {8'b0,
                    operand_in[23:0]};
16              `SHIFT_TRUNC_AND_EXTEND: begin
17                  operand_out <= {16'b0, operand_in[31:16]};
18                  stickyBit_out <= |(operand_in[15:0]);
19              end
20              default: begin
21                  operand_out <= 32'b0;
22                  //$display("Invalid Shift-And-Extend opcode!");
23              end
24          endcase
25      end
26  endmodule
```

Listing A.8: significand_alu.v

```
1   `timescale 1ns/1ps
2
3   `include "global.v"
4
5   module SignificandALU(aluOpCode_in, aluOpA_in, aluOpB_in,
        aluNegFlag_out, aluZeroFlag_out, aluResult_out);
6       input [3:0] aluOpCode_in;
7       input [31:0] aluOpA_in, aluOpB_in; //32 bit
8       output reg aluNegFlag_out;
9       output aluZeroFlag_out;
10      output [31:0] aluResult_out; //32 bit
11
12      reg signed [31:0] aluResult; //32 bit
13
14      assign aluResult_out = aluResult;
15      assign aluZeroFlag_out  = (aluResult == 32'b0) ? 1'b1 : 1'b0;
16
17      always @(aluOpCode_in, aluOpA_in, aluOpB_in) begin
18          //default outputs
19          aluResult    = 32'b0;
20          aluNegFlag_out = aluOpA_in[31];
21
22          case (aluOpCode_in)
23              `SIG_ALU_OP_NOP:     aluResult = 32'b0;
24              `SIG_ALU_OP_MOVA:    aluResult =  aluOpA_in;
25              `SIG_ALU_OP_NEGB:    aluResult = -aluOpB_in;
26              `SIG_ALU_OP_ADD:     {aluNegFlag_out, aluResult} =  {
                    aluOpA_in[31], aluOpA_in} + {aluOpB_in[31], aluOpB_in
                    };
27              `SIG_ALU_OP_SUB:     {aluNegFlag_out, aluResult} =  {
                    aluOpA_in[31], aluOpA_in} - {aluOpB_in[31], aluOpB_in
                    };
28              `SIG_ALU_OP_SHRA: begin
29                  aluResult =  aluOpA_in;
30                  if (aluOpB_in[0]) aluResult = aluResult >>> 1;
31                  if (aluOpB_in[1]) aluResult = aluResult >>> 2;
32                  if (aluOpB_in[2]) aluResult = aluResult >>> 4;
33                  if (aluOpB_in[3]) aluResult = aluResult >>> 8;
34                  if (aluOpB_in[4]) aluResult = aluResult >>> 16;
```

```
35              end
36              `SIG_ALU_OP_SHRL: begin
37                  aluResult =  aluOpA_in;
38                  if (aluOpB_in[0]) aluResult = aluResult >> 1;
39                  if (aluOpB_in[1]) aluResult = aluResult >> 2;
40                  if (aluOpB_in[2]) aluResult = aluResult >> 4;
41                  if (aluOpB_in[3]) aluResult = aluResult >> 8;
42                  if (aluOpB_in[4]) aluResult = aluResult >> 16;
43              end
44              `SIG_ALU_OP_SHLL: begin
45                  aluResult = aluOpA_in;
46                  if (aluOpB_in[0]) aluResult = aluResult << 1;
47                  if (aluOpB_in[1]) aluResult = aluResult << 2;
48                  if (aluOpB_in[2]) aluResult = aluResult << 4;
49                  if (aluOpB_in[3]) aluResult = aluResult << 8;
50                  if (aluOpB_in[4]) aluResult = aluResult << 16;
51              end
52
53              `SIG_ALU_OP_CLZ:    aluResult =  {26'b0, CLZ(aluOpA_in)};
54              default: begin
55                  //$display("Significand ALU: Undefined ALU OpCode!");
56              end
57          endcase
58      end
59
60
61      function [5:0] CLZ;
62              input [31:0] val32;
63              reg [15:0] val16;
64              reg [7:0] val8;
65              reg [3:0] val4;
66              reg [1:0] val2;
67              reg [4:0] result;
68          begin
69              result[4]   = (val32[31:16] == 16'b0);
70              val16       = (result[4]) ? val32[15:0] : val32[31:16];
71
72              result[3]   = (val16[15:8] == 8'b0);
73              val8        = (result[3]) ? val16[7:0] : val16[15:8];
74
75              result[2]   = (val8[7:4] == 4'b0);
76              val4        = (result[2]) ? val8[3:0] : val8[7:4];
77
78              result[1]   = (val4[3:2] == 2'b0);
79              val2        = (result[1]) ? val4[1:0] : val4[3:2];
80
81              result[0]   = (val2[1] == 1'b0);
82
83              //handle special case of input = 0
84              CLZ = ((result == 5'd31) && (val2[0] == 1'b0)) ? 6'd32 :
                    {1'b0, result};
85          end
86      endfunction
87
88  endmodule
```

Listing A.9: exponent_alu.v

```
1  `timescale 1ns/1ps
2
3  `include "global.v"
4
5  module ExponentALU(aluOpCode_in, aluOpA_in, aluOpB_in, aluNegFlag_out,
       aluZeroFlag_out, aluResult_out);
6      input [2:0] aluOpCode_in;
7      input [`FORMAT_EXP_WIDTH:0] aluOpA_in, aluOpB_in; // 9 bit
8      output aluNegFlag_out, aluZeroFlag_out;
9      output [`FORMAT_EXP_WIDTH:0] aluResult_out; // 9 bit
10
11     reg [`FORMAT_EXP_WIDTH:0] aluResult; //9 bit
12
13     assign aluResult_out = aluResult;
14     assign aluNegFlag_out   = (aluResult[`FORMAT_EXP_WIDTH] == 1'b1 )
           ? 1'b1 : 1'b0;
15     assign aluZeroFlag_out  = (aluResult == 9'b0) ? 1'b1 : 1'b0;
16
17
18     always @(aluOpCode_in, aluOpA_in, aluOpB_in) begin
19         //default outputs
20         aluResult   = 9'bx;
21
22         case (aluOpCode_in)
23             `EXP_ALU_OP_NOP:    aluResult =  9'b0;
24             `EXP_ALU_OP_MOVA:   aluResult =  aluOpA_in;
25             `EXP_ALU_OP_NEGB:   aluResult = -aluOpB_in;
26             `EXP_ALU_OP_ADD:    aluResult =  aluOpA_in + aluOpB_in;
27             `EXP_ALU_OP_SUB:    aluResult =  aluOpA_in - aluOpB_in;
28             `EXP_ALU_OP_SHL:    aluResult =  aluOpA_in << aluOpB_in;
29             default: $display("Exponent␣ALU:␣Undefined␣ALU␣OpCode!");
30         endcase
31     end
32
33 endmodule
```

Listing A.10: fpu_control_unit.v

```
1  `timescale 1ns/1ps
2
3  `include "global.v"
4
5  module FpuControlUnit(clk_in, reset_in, fpuOpCode_in, roundingMode_in,
6      signA_in, signB_in, isZeroA_in, isZeroB_in, isInfA_in, isInfB_in,
           isNanA_in,
7          isNanB_in, isDenormA_in, isDenormB_in,
8      expAluNegFlag_in, expAluZeroFlag_in,
9      sigAluNegFlag_in, sigAluZeroFlag_in,
10     guardBit_in, roundBit_in, stickyBitData_in,
11     readFPInput_out,
12     erfWriteSelectR0_out, erfWriteSelectR1_out, erfWriteEnableR0_out,
           erfWriteEnableR1_out,
13         erfReadSelectA_out, erfReadSelectB_out,
14     srfWriteSelectR0_out, srfWriteSelectR1_out, srfWriteEnableR0_out,
           srfWriteEnableR1_out,
15         srfShiftEnableR0_out, srfReadSelectA_out, srfReadSelectB_out,
```

```
16      expAluRegOrSigResult_out,
17      expAluOp_out,
18      errWriteEnable_out,
19      sigAluRegOrMul_out, sigAluSrr_out, sigAluRegOrExpResult_out,
20      sigAluOp_out,
21      srrWriteEnable_out,
22      srrShiftEnable_out,
23      srrShiftIn_out,
24      maskAndShiftOp_out,
25      mulEnable_out,
26      shiftAndExtendOp_out,
27      outputFPResult_out,
28      resultSign_out,
29      resultReady_out,
30      invalidOperationDetected_out, divisionByZeroDetected_out,
            overflowDetected_out,
31          underflowDetected_out, inexactDetected_out
32      );
33
34  //STATE DEFINITIONS
35  //Pre-process states
36      parameter STATE_IDLE                =  0; //idle state, ready for
            new input
37      parameter STATE_INIT                =  1; //read inputs, determine
             operation, detect exceptions
38
39  //Arithmetic states
40      //MUL states
41      parameter STATE_MUL_M1              =  2; //start first
            multiplication, add exponents
42      parameter STATE_MUL_M2              =  3; //start second
            multiplication, subtract bias from exponent sum
43      parameter STATE_MUL_M3              =  4; //start third
            multiplication
44      parameter STATE_MUL_M4              =  5; //start fourth
            multiplication, accumulate
45      parameter STATE_MUL_M5              =  6; //complete fourth
            multiplication, accumulate
46      parameter STATE_MUL_M6              =  7; //final accumulate
47
48      //ADDSUB states
49      parameter STATE_ADDSUB_EXPSUB       =  8; //compare input
            exponents
50      parameter STATE_ADDSUB_DIFFNEG      =  9; //find the absolute
            value of the exp. difference
51      parameter STATE_ADDSUB_DIFFPOS      = 10; //empty state - may be
            removed
52      parameter STATE_ADDSUB_SHIFTFRACA   = 11; //adjust input
            significand A
53      parameter STATE_ADDSUB_SHIFTFRACB   = 12; //adjust input
            significand B
54      parameter STATE_ADDSUB_ADDSUBFRACS  = 13; //add or subtract the
            adjusted significands
55      parameter STATE_ADDSUB_NEGSUM       = 14; //negate any negative
            sum, set the output sign to negative
56      parameter STATE_ADDSUB_POSSUM       = 15; //keep the sum, set the
            output sign to positive
```

```
57
58      //DIV states
59      parameter STATE_DIV_ADD_BIAS         = 16;
60      parameter STATE_DIV_SUB_EXP          = 17;
61      parameter STATE_DIV_ITER             = 18;
62
63      //int2fp states
64      parameter STATE_I2FP_TEST_SIGN       = 19;
65      parameter STATE_I2FP_NEGATE          = 20;
66      parameter STATE_I2FP_CLZ             = 21;
67      parameter STATE_I2FP_ADJ1            = 22;
68      parameter STATE_I2FP_ADJ2            = 23;
69
70      //fp2int states
71      parameter STATE_FP2SI_UNBIAS         = 24;
72      parameter STATE_FP2SI_CALC_ADJ       = 25;
73      parameter STATE_FP2SI_PREROUND_RSH   = 26;
74      parameter STATE_FP2SI_ROUND          = 27;
75      parameter STATE_FP2SI_POSTROUND_RSH = 28;
76      parameter STATE_FP2SI_NEGATE         = 29;
77
78  //Post-process states
79      //addsub-normalization
80      parameter STATE_CLZ                  = 30; //count leading zeros in
            a significand, needed for generic rounding
81      parameter STATE_NORMALIZE_CALC_ADJ   = 31; //calculate the required
            normalization adjustments
82      parameter STATE_NORMALIZE_GENERIC    = 32; //perform the actual
            normalization
83      //mul normalization
84      parameter STATE_MUL_NORM0            = 33; //multiplication
            normalization, case 1
85      parameter STATE_MUL_NORM1            = 34; //empty state, may be
            removed
86      //div/sqrt normalization
87      parameter STATE_NORM_DIV_SQRT        = 35; //normalize division or
            square root
88      //rounding
89      parameter STATE_ROUND                = 36; //round the result: R0 =
            round(R0)
90      parameter STATE_POST_ROUND_NORM      = 37; //correct de-normalize
            caused by rounding
91      //output states
92      parameter STATE_EMIT_RESULT          = 38;
93      parameter STATE_EMIT_ZERO            = 39;
94      parameter STATE_EMIT_INF             = 40;
95      parameter STATE_EMIT_NAN             = 41;
96
97
98  //PORTS
99      //input ports
100         input clk_in, reset_in;
101         input [3:0] fpuOpCode_in;
102         input [1:0] roundingMode_in;
103
104     //SVD values
105         input signA_in, signB_in, isZeroA_in, isZeroB_in, isInfA_in,
```

```
            isInfB_in, isNanA_in,
106           isNanB_in, isDenormA_in, isDenormB_in;
107
108     //ALU status flags
109         input expAluNegFlag_in, expAluZeroFlag_in;
110         input sigAluNegFlag_in, sigAluZeroFlag_in;
111
112     //bits 5 and 6 of the s.R0 register
113         input guardBit_in, roundBit_in;
114     //the logical OR between all discarded bits
115         input stickyBitData_in;
116
117     // input control
118     output reg readFPInput_out;  //interpret input as floating-point
            or integer?
119
120     //Register file control
121         //exponent register file
122             output reg erfWriteSelectR0_out, erfWriteSelectR1_out,
                    erfWriteEnableR0_out, erfWriteEnableR1_out;
123             output reg [2:0] erfReadSelectA_out, erfReadSelectB_out;
124         //exponent result register
125             output reg errWriteEnable_out;
126         //significand register file
127             output reg srfWriteSelectR0_out, srfWriteSelectR1_out,
                    srfWriteEnableR0_out, srfWriteEnableR1_out,
                    srfShiftEnableR0_out;
128             output reg [3:0] srfReadSelectA_out, srfReadSelectB_out;
129         //significand result register
130             output reg srrWriteEnable_out, srrShiftEnable_out,
                    srrShiftIn_out;
131
132
133     //Exponent ALU control
134         output reg expAluRegOrSigResult_out;
135         output reg [2:0] expAluOp_out;
136
137
138     //Significand ALU control
139         output reg sigAluRegOrMul_out, sigAluSrr_out;
140         output reg sigAluRegOrExpResult_out;
141         output reg [3:0] sigAluOp_out;
142
143
144     //multiplier chain
145         output reg [1:0] maskAndShiftOp_out;
146         output reg mulEnable_out;
147         output reg [1:0] shiftAndExtendOp_out;
148
149     //result related values
150         output reg outputFPResult_out;
151         output resultSign_out;
152         output reg resultReady_out;
153
154     //exception flags
155     output invalidOperationDetected_out, divisionByZeroDetected_out,
156            overflowDetected_out, underflowDetected_out,
```

```
                inexactDetected_out;
157
158
159 //INTERNAL REGISTERS
160     reg [5:0] currentState, nextState; //state registers
161     reg [3:0] fpuOpCode;               //active operation
162     reg [1:0] roundingMode;            //active rounding mode
163     reg signA, signB, isZeroA, isZeroB, isInfA, isInfB, isNanA, isNanB
            , isDenormA, isDenormB;
164     reg stickyBit;                     //current sticky bit value
165     reg resultSign;                    //result sign of active
            operation
166     reg [4:0] iterationCounter;        //counter used by DIV and SQRT
167     reg firstIterSign;                 //holds the sign generated by
            the first iteration, used for norm.
168     reg invalidOperationDetected, divisionByZeroDetected,
            overflowDetected, underflowDetected, inexactDetected;
169
170
171 //INTERNAL TEMPORARY VALUES
172     reg [3:0] fpuOpCode_val;
173     reg [1:0] roundingMode_val;
174     reg signA_val, signB_val, isZeroA_val, isZeroB_val, isInfA_val,
            isInfB_val, isNanA_val,
175         isNanB_val, isDenormA_val, isDenormB_val;
176     reg stickyBit_val;
177     reg resultSign_val;
178     reg [4:0] iterationCounter_val;
179     reg firstIterSign_val;
180     reg invalidOperationDetected_val, divisionByZeroDetected_val,
            overflowDetected_val,
181         underflowDetected_val, inexactDetected_val;
182
183 //ASSIGNMENTS
184     //output the sign of the active operation
185     assign resultSign_out = resultSign;
186     //output exception flags
187     assign invalidOperationDetected_out = invalidOperationDetected;
188     assign divisionByZeroDetected_out   = divisionByZeroDetected;
189     assign overflowDetected_out         = overflowDetected;
190     assign underflowDetected_out        = underflowDetected;
191     assign inexactDetected_out          = inexactDetected;
192
193
194 //asynchronous block
195     always @(*) begin
196     //DEFAULT VALUES
197         //preserve register values by default
198         fpuOpCode_val        = fpuOpCode;
199         roundingMode_val     = roundingMode;
200         signA_val            = signA;
201         signB_val            = signB;
202         isZeroA_val          = isZeroA;
203         isZeroB_val          = isZeroB;
204         isInfA_val           = isInfA;
205         isInfB_val           = isInfB;
206         isNanA_val           = isNanA;
```

```
207         isNanB_val            = isNanB;
208         isDenormA_val         = isDenormA;
209         isDenormB_val         = isDenormB;
210         stickyBit_val         = stickyBit;
211         resultSign_val        = resultSign;
212         iterationCounter_val  = iterationCounter;
213         firstIterSign_val     = firstIterSign;
214         invalidOperationDetected_val= invalidOperationDetected;
215         divisionByZeroDetected_val  = divisionByZeroDetected;
216         overflowDetected_val        = overflowDetected;
217         underflowDetected_val       = underflowDetected;
218         inexactDetected_val         = inexactDetected;
219
220         //input control
221         readFPInput_out = 1'b1; //interpret input as FP by default
222
223         //exponent register file
224         erfWriteEnableR0_out = 1'b0;
225         erfWriteEnableR1_out = 1'b0;
226         erfWriteSelectR0_out = 1'b0;
227         erfWriteSelectR1_out = 1'b0;
228         erfReadSelectA_out   = `ERF_REG_R0;
229         erfReadSelectB_out   = `ERF_REG_R1;
230
231         //significand register file
232         srfWriteEnableR0_out = 1'b0;
233         srfWriteEnableR1_out = 1'b0;
234         srfWriteSelectR0_out = 1'b0;
235         srfWriteSelectR1_out = 1'b0;
236         srfShiftEnableR0_out = 1'b0;
237         srfReadSelectA_out   = `SRF_REG_R0;
238         srfReadSelectB_out   = `SRF_REG_R1;
239
240         //exponent ALU
241         expAluRegOrSigResult_out= 1'b0;
242         expAluOp_out            = `EXP_ALU_OP_NOP;
243         errWriteEnable_out      = 1'b0;
244
245         //significand ALU
246         sigAluRegOrMul_out      = 1'b0;  //read op.A from register
              file or from multiplier?
247         sigAluSrr_out           = 1'b0;  //read op.A from Significand
              Result Register, og register file/multiplier?
248         sigAluRegOrExpResult_out= 1'b0;  //read op.B from register
              file or exponent result register=
249         sigAluOp_out            = `SIG_ALU_OP_NOP;
250         srrWriteEnable_out      = 1'b0;
251         srrShiftEnable_out      = 1'b0;
252         srrShiftIn_out          = 1'b0;
253
254         //multiplier chain
255         maskAndShiftOp_out   = `MASK_AND_SHIFT_A8C8;
256         mulEnable_out        = 1'b0;
257         shiftAndExtendOp_out = `SHIFT_16_BIT_AND_EXTEND;
258
259         //output related
260         outputFPResult_out = 1'b1; //output floating-point result by
```

```
            default
261         resultReady_out     = 1'b0;
262
263     //END: DEFAULT VALUES
264
265 //
    ////////////////////////////////////////////////////////////////////////////////
266         case (currentState)
267 //--------PRE-PROCESS STATES BEGIN
268             STATE_IDLE: begin
269                 nextState = STATE_INIT;
270
271                 //keep previous result valid, until a new operation is
                        started
272                 resultReady_out = 1'b1;
273                 outputFPResult_out = ~(fpuOpCode == `FPU_INSTR_FP2SI);
274             end
275             STATE_INIT: begin
276                 fpuOpCode_val = fpuOpCode_in;       //update active
                        operation
277                 roundingMode_val = roundingMode_in; //update active
                        rounding modes
278
279                 //read SVD signals
280                 signA_val       = signA_in;
281                 signB_val       = signB_in;
282                 isZeroA_val     = isZeroA_in;
283                 isZeroB_val     = isZeroB_in;
284                 isInfA_val      = isInfA_in;
285                 isInfB_val      = isInfB_in;
286                 isNanA_val      = isNanA_in;
287                 isNanB_val      = isNanB_in;
288                 isDenormA_val   = isDenormA_in;
289                 isDenormB_val   = isDenormB_in;
290
291
292                 //determine next state
293                 case (fpuOpCode_in)
294                     `FPU_INSTR_MUL: begin
295                         nextState = STATE_MUL_M1;   //default
                                nextState
296
297                         //detect input-time exceptions
298                         if (isZeroA_in)
299                             if (isInfB_in) begin // 0*Inf
300                                 $display("Invalid operation: 0*Inf");
301                                 nextState = STATE_EMIT_NAN;
302                                 invalidOperationDetected_val = 1'b1;
303                             end
304
305                         if (isZeroB_in)
306                             if (isInfA_in) begin //Inf*0
307                                 $display("Invalid operation: Inf*0");
308                                 nextState = STATE_EMIT_NAN;
309                                 invalidOperationDetected_val = 1'b1;
310                             end
```

```
311
312                         //read new inputs exponents
313                         erfWriteEnableR0_out = 1'b1;
314                         erfWriteEnableR1_out = 1'b1;
315                         erfWriteSelectR0_out = 1'b1;
316                         erfWriteSelectR1_out = 1'b1;
317                     end
318                     `FPU_INSTR_ADD: begin
319                         nextState = STATE_ADDSUB_EXPSUB; //default
                                next-state
320
321                         //detect input-time exceptions
322                         if (isInfA_in && isInfB_in) //both operands
                                Inf?
323                             if (signA_in != signB_in) begin //
                                    different signs?
324                                 $display("Invalid operation: (add)
                                        magnitude subtraction of
                                        infinities");
325                                 nextState = STATE_EMIT_NAN;
326                                 invalidOperationDetected_val = 1'b1;
327                             end
328
329                         //read new inputs exponents
330                         erfWriteEnableR0_out = 1'b1;
331                         erfWriteEnableR1_out = 1'b1;
332                         erfWriteSelectR0_out = 1'b1;
333                         erfWriteSelectR1_out = 1'b1;
334                         //read new significands
335                         srfWriteEnableR0_out = 1'b1;
336                         srfWriteEnableR1_out = 1'b1;
337                         srfWriteSelectR0_out = 1'b1;
338                         srfWriteSelectR1_out = 1'b1;
339                     end
340                     `FPU_INSTR_SUB: begin
341                         nextState = STATE_ADDSUB_EXPSUB;
342
343                         //detect input-time exceptions
344                         if (isInfA_in && isInfB_in) //both operands
                                Inf?
345                             if (signA_in == signB_in) begin //equal
                                    signs?
346                                 $display("Invalid operation: (sub)
                                        magnitude subtraction of
                                        infinities");
347                                 nextState = STATE_EMIT_NAN;
348                                 invalidOperationDetected_val = 1'b1;
349                             end
350
351                         //read new inputs exponents
352                         erfWriteEnableR0_out = 1'b1;
353                         erfWriteEnableR1_out = 1'b1;
354                         erfWriteSelectR0_out = 1'b1;
355                         erfWriteSelectR1_out = 1'b1;
356                         //read new significands
357                         srfWriteEnableR0_out = 1'b1;
358                         srfWriteEnableR1_out = 1'b1;
```

```
359                         srfWriteSelectR0_out = 1'b1;
360                         srfWriteSelectR1_out = 1'b1;
361                     end
362                     `FPU_INSTR_DIV: begin
363                         nextState = STATE_DIV_ADD_BIAS; //default next
                                -state
364
365                         //detect input-time exceptions
366                         if (isInfA_in && isInfB_in)  begin//both
                                operands Inf?
367                             $display("Invalid operation: inf/inf")
                                    ;
368                             nextState = STATE_EMIT_NAN;
369                             invalidOperationDetected_val = 1'b1;
370                         end
371                         if (isZeroB_in)
372                             if (isZeroA_in) begin // 0/0 division
373                             $display("Invalid operation: 0/0");
374                             nextState = STATE_EMIT_NAN;
375                         end else begin //division-by-zero
376                             $display("Invalid operation: division
                                    by zero");
377                             resultSign_val = signA_in ^ signB_in;
378                             nextState = STATE_EMIT_INF;
379                             divisionByZeroDetected_val = 1'b1;
380                         end
381
382                         //read new inputs exponents
383                         erfWriteEnableR0_out = 1'b1;
384                         erfWriteEnableR1_out = 1'b1;
385                         erfWriteSelectR0_out = 1'b1;
386                         erfWriteSelectR1_out = 1'b1;
387                         //read new significands
388                         srfWriteEnableR0_out = 1'b1;
389                         srfWriteEnableR1_out = 1'b1;
390                         srfWriteSelectR0_out = 1'b1;
391                         srfWriteSelectR1_out = 1'b1;
392                     end
393                     //`FPU_INSTR_SQRT: nextState = ;
394                     `FPU_INSTR_UI2FP: begin
395                         nextState = STATE_I2FP_CLZ;
396                         //read integer into s.R0
397                         readFPInput_out = 1'b0;
398                         srfWriteEnableR0_out = 1'b1;
399                         srfWriteSelectR0_out = 1'b1;
400                     end
401                     `FPU_INSTR_SI2FP: begin
402                         nextState = STATE_I2FP_TEST_SIGN;
403                         //read integer into s.R0
404                         readFPInput_out = 1'b0;
405                         srfWriteEnableR0_out = 1'b1;
406                         srfWriteSelectR0_out = 1'b1;
407                     end
408                     `FPU_INSTR_FP2SI: begin
409                         nextState = STATE_FP2SI_UNBIAS;
410
411                         erfWriteEnableR0_out = 1'b1;
```

```
412                         erfWriteSelectR0_out = 1'b1;
413                         srfWriteEnableR0_out = 1'b1;
414                         srfWriteSelectR0_out = 1'b1;
415                     end
416                     default: begin
417                         $display("Invalid instruction or not yet
                                implemented");
418                         nextState = STATE_IDLE;
419                     end
420                 endcase
421             end
422 //--------PRE-PROCESS STATES END
423
424 //
    ////////////////////////////////////////////////////////////////////////////////
425
426     //--------MUL STATES BEGIN
427             STATE_MUL_M1: begin
428                 nextState = STATE_MUL_M2;
429
430                 erfWriteEnableR0_out = 1'b1;    //store addition
                        result in R0
431                 expAluOp_out = `EXP_ALU_OP_ADD; //add exponents
432
433                 maskAndShiftOp_out = `MASK_AND_SHIFT_A8C8;
434                 mulEnable_out = 1'b1;
435
436                 resultSign_val = signA ^ signB;
437             end
438             STATE_MUL_M2: begin
439                 nextState = STATE_MUL_M3;
440
441                 erfWriteEnableR0_out = 1'b1; //write back exponent to
                        exp.R0
442                 erfReadSelectB_out = `ERF_REG_BIAS; //'d127
443                 expAluOp_out = `EXP_ALU_OP_SUB; //subtract bias
444
445                 maskAndShiftOp_out = `MASK_AND_SHIFT_A8D16;
446                 mulEnable_out = 1'b1;
447                 sigAluRegOrMul_out = 1'b1;
448             end
449             STATE_MUL_M3: begin
450                 nextState = STATE_MUL_M4;
451
452                 srfWriteEnableR0_out = 1'b1;
453                 sigAluRegOrMul_out = 1'b1;          //read partial
                        product
454                 sigAluOp_out = `SIG_ALU_OP_MOVA;    //accumulate first
                         partial product
455
456                 maskAndShiftOp_out = `MASK_AND_SHIFT_B16C8;
457                 mulEnable_out = 1'b1;
458             end
459             STATE_MUL_M4: begin
460                 nextState = STATE_MUL_M5;
461
```

```
462                 srfWriteEnableR0_out = 1'b1;
463                 srfReadSelectB_out = 3'b000;     //read B from R0
464                 sigAluRegOrMul_out = 1'b1;       //read partial product
465                 sigAluOp_out = `SIG_ALU_OP_ADD; //accumulate second
                        partial product
466
467                 maskAndShiftOp_out = `MASK_AND_SHIFT_B16D16;
468                 mulEnable_out = 1'b1;
469                 shiftAndExtendOp_out = `SHIFT_0_BIT_AND_EXTEND;
470             end
471             STATE_MUL_M5: begin
472                 nextState = STATE_MUL_M6;
473
474                 srfWriteEnableR0_out = 1'b1;
475                 srfReadSelectB_out = `ERF_REG_R0;       //read B from
                        R0
476                 sigAluRegOrMul_out = 1'b1;       //read partial product
477                 sigAluOp_out = `SIG_ALU_OP_ADD; //accumulate third
                        partial product
478
479                 mulEnable_out = 1'b1;
480                 shiftAndExtendOp_out = `SHIFT_0_BIT_AND_EXTEND;
481             end
482             STATE_MUL_M6: begin
483                 nextState = (sigAluNegFlag_in == 1'b1) ?
                        STATE_MUL_NORM0 : STATE_MUL_NORM1;
484
485                 mulEnable_out = 1'b1;
486                 shiftAndExtendOp_out = `SHIFT_TRUNC_AND_EXTEND;
487
488                 stickyBit_val = stickyBitData_in;
489
490                 srfWriteEnableR0_out = 1'b1; //write back final
                        product to R0
491                 srfReadSelectB_out = `SRF_REG_R0;   //read B from R0
492                 sigAluRegOrMul_out = 1'b1;       //read partial product
493                 sigAluOp_out = `SIG_ALU_OP_ADD; //accumulate fourth
                        partial product
494             end
495     //--------MUL STATES END
496
497 //
    ////////////////////////////////////////////////////////////////////////////////
498
499     //--------ADDSUB STATES BEGIN
500             STATE_ADDSUB_EXPSUB: begin
501                 //do we need to negate fraction A?
502                 if (signA == 1'b1) begin
503                     srfReadSelectB_out = `SRF_REG_R0; //read fractionA
                            from s.R0
504                     sigAluOp_out = `SIG_ALU_OP_NEGB;
505                     srfWriteEnableR0_out = 1'b1; //store negated
                            fraction in s.R0
506                 end
507
508                 if (expAluNegFlag_in == 1'b1) begin
```

```
509                     nextState = STATE_ADDSUB_DIFFNEG;
510                     erfWriteEnableR0_out = 1'b1;        //write back
                            subtraction result to R0
511                 end else begin
512                     nextState = STATE_ADDSUB_DIFFPOS;
513                     erfWriteEnableR1_out = 1'b1;        //write back
                            subtraction result to R1
514                 end
515
516                 expAluOp_out = `EXP_ALU_OP_SUB;         //perform E_A
                        - E_B
517             end
518             STATE_ADDSUB_DIFFNEG: begin
519                 nextState = STATE_ADDSUB_SHIFTFRACA;
520
521                 erfReadSelectB_out = `ERF_REG_R0;    //output R0 on
                        port B
522                 expAluOp_out = `EXP_ALU_OP_NEGB; //negate exponent
                        difference, to find it's absolute
523                 errWriteEnable_out = 1'b1;       //write the absolute
                        exp.diff to ERR
524             end
525             STATE_ADDSUB_DIFFPOS: begin
526                 nextState = STATE_ADDSUB_SHIFTFRACB;
527
528                 erfReadSelectA_out = `ERF_REG_R1;   //output R1 on
                        port A
529                 expAluOp_out = `EXP_ALU_OP_MOVA;//positive exponent
                        difference, move
530                 errWriteEnable_out = 1'b1;      //move the exp.diff to
                         ERR
531             end
532             STATE_ADDSUB_SHIFTFRACA: begin
533                 nextState = STATE_ADDSUB_ADDSUBFRACS;
534
535                 erfReadSelectA_out = `ERF_REG_R1;       //read E_B
                        from e.R1
536                 expAluOp_out = `EXP_ALU_OP_MOVA;//move E_B to e.R0
537                 erfWriteEnableR0_out = 1'b1;
538
539                 srfReadSelectA_out = `SRF_REG_R0; //read the
                        significand from R0
540                 sigAluRegOrExpResult_out = 1'b1; //use the ERR value (
                        exp.diff) as operand B to the sigALU
541                 sigAluOp_out = `SIG_ALU_OP_SHRA;
542                 srfWriteEnableR0_out = 1'b1; //write back shifted
                        significand to R0
543             end
544             STATE_ADDSUB_SHIFTFRACB: begin
545                 nextState = STATE_ADDSUB_ADDSUBFRACS;
546
547                 srfReadSelectA_out = `SRF_REG_R1; //read S_B from s.R1
548                 sigAluRegOrExpResult_out = 1'b1; //use the ERR value (
                        exp.diff) as operand B to the sigALU
549                 sigAluOp_out = `SIG_ALU_OP_SHRA;
550                 srfWriteEnableR1_out = 1'b1; //write back shifted
                        significand to s.R1
```

```
551             end
552             STATE_ADDSUB_ADDSUBFRACS: begin
553                 nextState = (sigAluNegFlag_in == 1'b1) ?
                        STATE_ADDSUB_NEGSUM : STATE_ADDSUB_POSSUM;
554
555                 //determine the effective operation
556                 if (fpuOpCode == `FPU_INSTR_ADD) begin
557                     if (signB == 1'b1) sigAluOp_out = `SIG_ALU_OP_SUB;
558                     else sigAluOp_out = `SIG_ALU_OP_ADD;
559                 end else if (fpuOpCode == `FPU_INSTR_SUB) begin
560                     if (((~signA)&(~signB)) | ((signA)&(~signB)))
                            sigAluOp_out = `SIG_ALU_OP_SUB;
561                     else sigAluOp_out = `SIG_ALU_OP_ADD;
562                 end else
563                     $display("Impossible case detected!");
564
565                 srfWriteEnableR0_out = 1'b1; //write back addition/
                        subtraction result to s.R0
566             end
567             STATE_ADDSUB_NEGSUM: begin
568                 nextState = STATE_CLZ;
569
570                 srfReadSelectB_out = `SRF_REG_R0;  //relay s.R0 to
                        port B
571                 sigAluOp_out = `SIG_ALU_OP_NEGB;
572                 srfWriteEnableR0_out = 1'b1; //write back negated sum
                        to s.R0
573
574                 resultSign_val = 1'b1; //set the result sign bit
575             end
576             STATE_ADDSUB_POSSUM: begin
577                 nextState = STATE_CLZ;
578
579                 resultSign_val = 1'b0; //set the result sign bit
580             end
581     //--------ADDSUB STATES END
582
583 //
        ////////////////////////////////////////////////////////////////////////////////
584
585     //--------DIV STATES BEGIN
586             STATE_DIV_ADD_BIAS: begin
587                 nextState = STATE_DIV_SUB_EXP;
588
589                 erfReadSelectB_out = `ERF_REG_BIAS; //read 'd127 (bias
                        ) from e.C4
590                 expAluOp_out = `EXP_ALU_OP_ADD; //add e.R0 to e.C4
591                 erfWriteEnableR0_out = 1'b1;    //store "unbiased"
                        exponent in e.R0
592
593                 srfShiftEnableR0_out = 1'b1;
594                 sigAluOp_out = `SIG_ALU_OP_SUB; //subtract the
                        denominator from the partial remainder
595                 srfWriteEnableR0_out = ~sigAluNegFlag_in; //only
                        update the partial remainder if the
596                                                           //
```

```
                                                                    subtraction
                                                                     gave a
                                                                    positive
                                                                     result
597                 srrShiftIn_out = ~sigAluNegFlag_in;
598                 srrShiftEnable_out = 1'b1;
599
600                 firstIterSign_val = sigAluNegFlag_in; //store the sign
                         from this iteration
601
602                 iterationCounter_val = iterationCounter + 5'd1; //
                        increment the iteration counter
603             end
604
605             STATE_DIV_SUB_EXP: begin
606                 nextState = STATE_DIV_ITER;
607
608                 expAluOp_out = `EXP_ALU_OP_SUB; //(e.R0 - e.R1)
609                 erfWriteEnableR0_out = 1'b1;    //store the exponent
                        in e.R0
610
611                 resultSign_val = signA ^ signB;
612
613                 srfShiftEnableR0_out = 1'b1;
614                 sigAluOp_out = `SIG_ALU_OP_SUB; //subtract the
                        denominator from the partial remainder
615                 srfWriteEnableR0_out = ~sigAluNegFlag_in; //only
                        update the partial remainder if the
616                                                                    //
                                                                    subtraction
                                                                     gave a
                                                                    positive
                                                                     result
617                 srrShiftEnable_out = 1'b1;
618                 srrShiftIn_out = ~sigAluNegFlag_in;
619
620                 iterationCounter_val = iterationCounter + 5'd1; //
                        increment the iteration counter
621             end
622
623             STATE_DIV_ITER: begin
624                 //determine nextState
625                 nextState = (iterationCounter >= 25) ?
                        STATE_NORM_DIV_SQRT : STATE_DIV_ITER;
626
627                 srfShiftEnableR0_out = 1'b1;
628                 sigAluOp_out = `SIG_ALU_OP_SUB; //subtract the
                        denominator from the partial remainder
629                 srfWriteEnableR0_out = ~sigAluNegFlag_in; //only
                        update the partial remainder if the
630                                                                    //
                                                                    subtraction
                                                                     gave a
                                                                    positive
                                                                     result
631                 srrShiftEnable_out = 1'b1;
632                 srrShiftIn_out = ~sigAluNegFlag_in;
```

```
633
634                 iterationCounter_val = iterationCounter + 5'd1; //
                        increment the iteration counter
635             end
636     //--------DIV STATES END
637
638
639     //--------INT2FP STATES BEGIN
640             STATE_I2FP_TEST_SIGN: begin
641                nextState = (sigAluNegFlag_in) ? STATE_I2FP_NEGATE :
                       STATE_I2FP_CLZ ;
642
643                sigAluOp_out = `SIG_ALU_OP_MOVA; //move the integer
                       through the significand ALU,
644                                                 //in order to trigger
                                                        the neg.flag
645             end
646             STATE_I2FP_NEGATE: begin
647                nextState = STATE_I2FP_CLZ;
648
649                srfReadSelectB_out = `SRF_REG_R0; //output srf.R0 on
                       srf.portB
650                sigAluOp_out = `SIG_ALU_OP_NEGB; //negate the integer,
                       in order to convert it to sign-magnitude
651                srfWriteEnableR0_out = 1'b1;      //store the negated
                       integer in srf.R0
652                resultSign_val = 1'b1;            //assert a negative
                       sign bit
653             end
654             STATE_I2FP_CLZ: begin
655                 nextState = STATE_I2FP_ADJ1;
656                 sigAluOp_out = `SIG_ALU_OP_CLZ; //perform CLZ on input
                        value
657                 srfWriteEnableR1_out = 1'b1;     //store #LZ in srf.R1
658                 srrWriteEnable_out = 1'b1;       //store #LZ in SRR in
                        order to make it available to expALU
659             end
660             STATE_I2FP_ADJ1: begin
661                 nextState = STATE_I2FP_ADJ2;
662
663                 erfReadSelectA_out = `ERF_REG_I2FP; //read (bias + 31)
                        from constant registers
664                 expAluRegOrSigResult_out = 1'b1;     //read #LZ from
                       SRR
665                 expAluOp_out = `EXP_ALU_OP_SUB;      //(bias + 31) - #
                       LZ
666                 erfWriteEnableR0_out = 1'b1;         //store exponent
                       result in e.R0
667
668                 sigAluOp_out = `SIG_ALU_OP_SHLL; //s.R0 << s.R1
669                 srfWriteEnableR0_out = 1'b1;     //store shifted value
                       in s.R0
670             end
671             STATE_I2FP_ADJ2: begin
672                 nextState = STATE_ROUND;
673
674                 expAluOp_out = `EXP_ALU_OP_MOVA;
```

```
675                 erfWriteEnableR0_out = 1'b1;
676                 //errWriteEnable_out = 1'b1;
677
678                 srfReadSelectB_out = `SRF_REG_ONE;  //read 'd1 from
                        the constant registers
679                 sigAluOp_out = `SIG_ALU_OP_SHRL;        //right-shift
                        the significand to the position assumed by the
                        round stage
680                 srfWriteEnableR0_out = 1'b1;        //store the
                        shifted value in s.R0
681                 //srrWriteEnable_out = 1'b1;
682             end
683     //--------INT2FP STATES END
684
685     //--------FP2INT STATES BEGIN
686             STATE_FP2SI_UNBIAS: begin
687                nextState = STATE_FP2SI_CALC_ADJ;
688
689                erfReadSelectB_out = `ERF_REG_BIAS;  //fetch the
                       exponent bias from the constant registers
690                expAluOp_out = `EXP_ALU_OP_SUB;      //subtract the
                       bias from the input exponent
691                erfWriteEnableR0_out = 1'b1;         //store the
                       unbiased exponent in erf.R0
692
693                //left-shift the input significand 2 digits, in order
                       to place it
694                //to the far left of the register
695                srfReadSelectB_out = `SRF_REG_TWO;
696                sigAluOp_out = `SIG_ALU_OP_SHLL;
697                srfWriteEnableR0_out = 1'b1; //store the shifted
                       significand in srf.R0
698             end
699             STATE_FP2SI_CALC_ADJ: begin
700                nextState = STATE_FP2SI_PREROUND_RSH;
701
702                //perform (31-true exponent)
703                erfReadSelectA_out = `ERF_REG_RPP;
704                erfReadSelectB_out = `ERF_REG_R0;
705                expAluOp_out = `EXP_ALU_OP_SUB;
706                errWriteEnable_out = 1'b1; //store the adjustment
                       amount in ERR
707
708             end
709             STATE_FP2SI_PREROUND_RSH: begin
710                 //TODO: split this right-shift into two parts
711                //right-shift the significand
712                sigAluRegOrExpResult_out = 1'b1; //fetch the shift
                       amount from the ERR
713                sigAluOp_out = `SIG_ALU_OP_SHRL;
714                srfWriteEnableR0_out = 1'b1;
715
716                //TEMP!
717                nextState = STATE_FP2SI_NEGATE; //no rounding yet...
718
719             end
720             STATE_FP2SI_ROUND: begin
```

```
721                 //TODO: implement this
722             end
723             STATE_FP2SI_POSTROUND_RSH: begin
724                 //TODO: implement this
725             end
726             STATE_FP2SI_NEGATE: begin
727                 nextState = STATE_EMIT_RESULT;
728 
729                 srfReadSelectA_out = `SRF_REG_R0; //read the integer
                        from srf.R0
730                 srfReadSelectB_out = `SRF_REG_R0; //read the integer
                        from srf.R0
731                 sigAluOp_out = (signA) ?  `SIG_ALU_OP_NEGB :
                        `SIG_ALU_OP_MOVA; //either move or negate the
                        integer
732                 srrWriteEnable_out = 1'b1; //write the final integer
                        to SRR
733             end
734     //--------FP2INT STATES END
735 
736 //
        ////////////////////////////////////////////////////////////////////////////////
737 
738 //--------POST-PROCESS STATES BEGIN
739             STATE_CLZ: begin
740                 nextState = STATE_NORMALIZE_CALC_ADJ;
741 
742                 //add one to the exponent, in order to maintain the
                        correct value
743                 expAluOp_out = `EXP_ALU_OP_ADD;
744                 erfReadSelectB_out = `ERF_REG_ONE;  //read 'd1 from
                        ERF
745                 erfWriteEnableR0_out = 1'b1;//write back addition
                        result to e.R0
746 
747                 sigAluOp_out = `SIG_ALU_OP_CLZ;
748                 srfWriteEnableR1_out = 1'b1; //write back CLZ result
                        to s.R1
749             end
750             STATE_NORMALIZE_CALC_ADJ: begin
751                 nextState = STATE_NORMALIZE_GENERIC;
752 
753                 //calculate the required significand shift amount
754                 srfWriteEnableR1_out = 1'b1; //store #LZ-1 result in s
                        .R1
755                 srfReadSelectA_out = `SRF_REG_R1; //read #LZ from s.R1
756                 srfReadSelectB_out = `SRF_REG_ONE; //read 'd1 from
                        constant registers
757                 sigAluOp_out = `SIG_ALU_OP_SUB;
758                 srrWriteEnable_out = 1'b1;   //store #LZ-1 result in
                        SRR
759             end
760             STATE_NORMALIZE_GENERIC: begin
761                 nextState = STATE_ROUND;
762 
763                 expAluRegOrSigResult_out = 1'b1; //read #LZ-1 from SRR
```

```
764                 expAluOp_out = `EXP_ALU_OP_SUB;  //subtract (#LZ-1)
                        from the current exponent
765                 erfWriteEnableR0_out = 1'b1;     //write back
                        normalized exponent to e.R0
766
767                 sigAluOp_out = `SIG_ALU_OP_SHLL; //left-shift the
                        significand, in order to normalize it
768                 srfWriteEnableR0_out = 1'b1;     //write back
                        normalized significand to s.R0
769             end
770             STATE_MUL_NORM0: begin
771                 nextState = STATE_ROUND;
772
773                 //the significand is already normalized, but the
                        exponent must be incremented by one
774                 erfWriteEnableR0_out = 1'b1; //write back normalized
                        exp to e.R0
775                 erfReadSelectB_out = `ERF_REG_ONE; //read 'd1 from the
                         constant registers
776                 expAluOp_out = `EXP_ALU_OP_ADD;
777
778                 srfWriteEnableR0_out = 1'b1; //write back normalized
                        frac to s.R0
779                 srfReadSelectB_out = `SRF_REG_ONE; //read 'd1 from the
                         constant registers
780                 sigAluOp_out = `SIG_ALU_OP_SHRL;
781
782             end
783             STATE_MUL_NORM1: begin
784                 nextState = STATE_ROUND;
785                 //do nothing
786             end
787             STATE_NORM_DIV_SQRT: begin
788                 if (firstIterSign) begin //answer is 0.5xxxx...
789                     erfWriteEnableR0_out = 1'b1;
790                     erfReadSelectB_out = `ERF_REG_ONE; //read 'd1 from
                             constant register
791                     expAluOp_out = `EXP_ALU_OP_SUB;
792
793                     srfReadSelectB_out = `SRF_REG_SIX; //read 'd2 from
                             the constant registers
794                     sigAluSrr_out = 1'b1; //feed SRR back to the
                            sigALU
795                     sigAluOp_out = `SIG_ALU_OP_SHLL; //shift the
                            leading one to the place expected by the round
                             unit
796                     srfWriteEnableR0_out = 1'b1; //store the
                            normalized fraction in s.R0
797                 end else begin
798                     expAluOp_out = `EXP_ALU_OP_MOVA; //keep the
                            current exponent, no need to adjust
799
800                     srfReadSelectB_out = `SRF_REG_FIVE; //read 'd1
                            from the constant registers
801                     sigAluSrr_out = 1'b1; //feed SRR back to the
                            sigALU
802                     sigAluOp_out = `SIG_ALU_OP_SHLL; //shift the
```

```
                        leading one to the place expected by the round
                         unit
803                 srfWriteEnableR0_out = 1'b1; //store the
                        normalized fraction in s.R0
804             end
805
806             nextState = STATE_ROUND;
807         end
808         STATE_ROUND: begin
809             nextState = STATE_EMIT_RESULT;  //default next state,
                    might be different...
810
811             //no rounding
812             /*expAluOp_out = `EXP_ALU_OP_MOVA;
813             sigAluOp_out = `SIG_ALU_OP_MOVA;
814             errWriteEnable_out = 1'b1;
815             srrWriteEnable_out = 1'b1;*/
816
817
818             case (roundingMode)
819                 `ROUNDING_MODE_NEAREST_EVEN: begin
820                     $display("Round towards nearest even: not yet
                            implemented");
821                 end
822                 `ROUNDING_MODE_POS_INF: begin
823                     //$display("(g,r,s): (%b,%b,%b)", guardBit_in,
                             roundBit_in, stickyBit);
824                     if ((~resultSign) & (guardBit_in | roundBit_in
                             | stickyBit)) begin
825                         //$display("->+Inf:Adding ULP");
826                         expAluOp_out = `EXP_ALU_OP_MOVA;
827                         srfReadSelectB_out = `SRF_REG_ULP_ROUND;
                                //read ULP ('d128) from constant
                                registers
828                         sigAluOp_out = `SIG_ALU_OP_ADD;
829                     end else begin
830                         //No rounding needed, just forward
831                         expAluOp_out = `EXP_ALU_OP_MOVA;
832                         sigAluOp_out = `SIG_ALU_OP_MOVA;
833                     end
834                 end
835                 `ROUNDING_MODE_NEG_INF: begin
836                     if ((resultSign) & (guardBit_in | roundBit_in
                            | stickyBit)) begin
837                         //rounding needed, add ULP
838                         expAluOp_out = `EXP_ALU_OP_MOVA;
839                         srfReadSelectB_out = `SRF_REG_ULP_ROUND;
                                //read ULP ('d128) from constant
                                registers
840                         sigAluOp_out = `SIG_ALU_OP_ADD;
841                     end else begin
842                         //No rounding needed, just forward
843                         expAluOp_out = `EXP_ALU_OP_MOVA;
844                         sigAluOp_out = `SIG_ALU_OP_MOVA;
845                     end
846                 end
847                 `ROUNDING_MODE_TRUNCATE: begin
```

```
848                         expAluOp_out = `EXP_ALU_OP_MOVA;
849                         sigAluOp_out = `SIG_ALU_OP_MOVA;
850                     end
851                 endcase
852 
853                 //generate next state, based on the MSB of the
                        significand calculation
854                 if (sigAluNegFlag_in == 1'b1) begin
855                     nextState = STATE_POST_ROUND_NORM;
856                     srfWriteEnableR0_out = 1'b1; //write rounded
                            significand back to s.R0
857                 end else begin
858                     nextState = STATE_EMIT_RESULT;
859                     errWriteEnable_out = 1'b1;
860                     srrWriteEnable_out = 1'b1;
861                 end
862             end
863             STATE_POST_ROUND_NORM: begin
864                 nextState = STATE_EMIT_RESULT;
865 
866                 //TODO
867                 expAluOp_out = `EXP_ALU_OP_MOVA;
868                 errWriteEnable_out = 1'b1;
869                 sigAluOp_out = `SIG_ALU_OP_MOVA;
870                 srrWriteEnable_out = 1'b1;
871             end
872         //output states
873             STATE_EMIT_RESULT: begin
874                 //operation finished, flag result ready
875                 resultReady_out = 1'b1;
876                 nextState = STATE_IDLE;
877 
878                 outputFPResult_out = ~(fpuOpCode == `FPU_INSTR_FP2SI);
879             end
880             STATE_EMIT_ZERO: begin
881                 erfReadSelectA_out = `ERF_REG_ZERO; //move 'd0 through
                        the ALU
882                 expAluOp_out = `EXP_ALU_OP_MOVA;
883                 errWriteEnable_out = 1'b1; //update the result
                        register
884 
885                 srfReadSelectA_out = `SRF_REG_ZERO; //move 'd0 through
                        the ALU
886                 sigAluOp_out = `SIG_ALU_OP_MOVA;
887                 srrWriteEnable_out = 1'b1; //update the result
                        register
888 
889                 nextState = STATE_IDLE;
890             end
891             STATE_EMIT_INF: begin
892                 //$display("Emitting Inf");
893                 erfReadSelectA_out = `ERF_REG_ONES; //move 'b11111111
                        through the ALU
894                 expAluOp_out = `EXP_ALU_OP_MOVA;
895                 errWriteEnable_out = 1'b1; //update the result
                        register
896 
```

```
897                 srfReadSelectA_out = `SRF_REG_ZERO; //move 'd
898                 sigAluOp_out = `SIG_ALU_OP_MOVA;
899                 srrWriteEnable_out = 1'b1; //update the result
                        register
900 
901                 nextState = STATE_IDLE;
902             end
903             STATE_EMIT_NAN: begin
904                 //$display("Emitting NaN");
905                 erfReadSelectA_out = `ERF_REG_ONES; //move 'b11111111
                        through the ALU
906                 expAluOp_out = `EXP_ALU_OP_MOVA;
907                 errWriteEnable_out = 1'b1; //update the result
                        register
908 
909                 srfReadSelectA_out = `SRF_REG_NANSIG; //move 'd
910                 sigAluOp_out = `SIG_ALU_OP_MOVA;
911                 srrWriteEnable_out = 1'b1; //update the result
                        register
912 
913                 nextState = STATE_IDLE;
914             end
915 
916 
917 //--------POST PROCESS STATES END
918 
919             default: begin
920                 //$display("Control Unit: illegal state reached!");
921                 nextState = STATE_IDLE;
922             end
923         endcase
924     end
925 
926 
927 
928 //synchronous block
929     always @(posedge clk_in) begin
930         if (reset_in == 1'b1) begin //synchronous reset
931             //reset state register
932             currentState = STATE_IDLE;
933 
934             //reset registers
935             fpuOpCode    = 4'bx;
936             roundingMode= 2'bx;
937             signA        = 1'bx;
938             signB        = 1'bx;
939             isZeroA      = 1'bx;
940             isZeroB      = 1'bx;
941             isInfA       = 1'bx;
942             isInfB       = 1'bx;
943             isNanA       = 1'bx;
944             isNanB       = 1'bx;
945             isDenormA    = 1'bx;
946             isDenormB    = 1'bx;
947             stickyBit    = 1'b0;
948             resultSign   = 1'b0;
949             iterationCounter        = 5'd0;
```

```
950             firstIterSign           = 1'b0;
951             invalidOperationDetected= 1'b0;
952             divisionByZeroDetected  = 1'b0;
953             overflowDetected        = 1'b0;
954             underflowDetected       = 1'b0;
955             inexactDetected         = 1'b0;
956 
957         end else begin
958             //update state register
959             currentState = nextState;
960 
961             //update internal registers
962             fpuOpCode    = fpuOpCode_val;
963             roundingMode= roundingMode_val;
964             signA        = signA_val;
965             signB        = signB_val;
966             isZeroA      = isZeroA_val;
967             isZeroB      = isZeroB_val;
968             isInfA       = isInfA_val;
969             isInfB       = isInfB_val;
970             isNanA       = isNanA_val;
971             isNanB       = isNanB_val;
972             isDenormA    = isDenormA_val;
973             isDenormB    = isDenormB_val;
974             stickyBit    = stickyBit_val;
975             resultSign   = resultSign_val;
976             iterationCounter        = iterationCounter_val;
977             firstIterSign           = firstIterSign_val;
978             invalidOperationDetected= invalidOperationDetected_val;
979             divisionByZeroDetected  = divisionByZeroDetected_val;
980             overflowDetected        = overflowDetected_val;
981             underflowDetected       = underflowDetected_val;
982             inexactDetected         = inexactDetected_val;
983         end
984     end
985 endmodule
```

Listing A.11: fpu_top.v

```
1  `timescale 1ns/1ps
2 
3  `include "global.v"
4 
5  module FPU_top(clk_in, reset_in, fpuOpCode_in, roundingMode_in,
       operandA_in, operandB_in, resultReady_out, result_out,
6                 invalidOperation_out, divisionByZero_out, overflow_out,
                      underflow_out, inexact_out);
7      // I/O PORTS
8      input clk_in, reset_in;
9      input [3:0] fpuOpCode_in;
10     input [1:0] roundingMode_in;
11     input [31:0] operandA_in, operandB_in;
12     output resultReady_out;
13     output [31:0] result_out;
14     //exception flags
15     output invalidOperation_out, divisionByZero_out, overflow_out,
           underflow_out, inexact_out;
```

```
16
17      // INTERNAL REGISTERS
18      reg [8:0] expAluResultRegister; //9 bit
19      reg [31:0] sigAluResultRegister; //31 bit
20
21      //WIRES
22          //input control
23          wire readFPInput; //mux between integer and floating-point
                input
24          wire [31:0] trueInputA;
25
26          //SVD output
27              //to control unit
28              wire signA, signB, isZeroA, isZeroB, isInfA, isInfB,
                    isNanA, isNanB, isDenormA, isDenormB;
29              //data
30              wire [32:0] operandAExpanded, operandBExpanded;
31
32          //exp ALU register file
33              //control
34              wire erfWriteSelectR0, erfWriteSelectR1;
35              wire erfWriteEnableR0, erfWriteEnableR1;
36              wire [2:0] erfReadSelectA, erfReadSelectB;
37              //data
38              wire [8:0] erfWriteValueR0, erfWriteValueR1; // 9 bit
39              wire [8:0] erfReadValueA, erfReadValueB; // 9 bit
40
41          //exp ALU connections
42              //control
43              wire expAluRegOrSigResult;
44              wire [2:0] expAluOpCode;
45              wire errWriteEnable;
46              //data
47              wire expAluNegFlag, expAluZeroFlag;
48              wire [8:0] expAluOpA, expAluOpB;
49              wire [8:0] expAluResult;
50
51          //large ALU register file
52              //control
53              wire srfWriteSelectR0, srfWriteSelectR1;
54              wire srfWriteEnableR0, srfWriteEnableR1;
55              wire srfShiftEnableR0;
56              wire [3:0] srfReadSelectA, srfReadSelectB;
57              //data
58              wire [31:0] srfWriteValueR0, srfWriteValueR1; // 32 bit
59              wire [31:0] srfReadValueA, srfReadValueB; // 32 bit
60
61          //significand ALU connections
62          //control
63              wire sigAluRegOrMul, sigAluSrr, sigAluRegOrExpResult;
64              wire [3:0] sigAluOpCode;
65              wire srrWriteEnable, srrShiftEnable;
66              //data
67              wire [31:0] sigAluOpA, sigAluOpB, sigAluOpA_tmp;
68              wire sigAluNegFlag, sigAluZeroFlag;
69              wire [31:0] sigAluResult;
70              wire srrShiftIn;
```

```
71
72          // multiplier chain
73              //control
74              wire [1:0] maskAndShiftOp;
75              wire mulEnable;
76              wire [1:0] shiftAndExtendOp;
77              //data
78              wire [15:0] mulInputMaskedShiftedA, mulInputMaskedShiftedB
                    ;
79              wire [31:0] mulResult;
80              wire [31:0] mulResultShiftedExtended;
81              wire stickyBitData;
82
83
84          //output control
85              wire outputFPResult; //mux between outputting a FP-result
                    or an integer result
86              wire resultSign;
87
88      // ASSIGNMENTS
89          assign result_out[31:0] = (outputFPResult) ? {resultSign,
                expAluResultRegister[7:0], sigAluResultRegister[29:7]} :
90                                                                  sigAluResultRegister
                                                                        [31:0];
91
92          //mux between input exponent and exp ALU result
93          assign erfWriteValueR0 = (erfWriteSelectR0 == 1'b0) ?
                expAluResult : operandAExpanded[31:24];
94          //mux between input exponent and exp ALU result
95          assign erfWriteValueR1 = (erfWriteSelectR1 == 1'b0) ?
                expAluResult : operandBExpanded[31:24];
96
97          //connect small register file with the small ALU
98          assign expAluOpA = erfReadValueA;
99          assign expAluOpB = (expAluRegOrSigResult == 1'b0) ?
                erfReadValueB : sigAluResultRegister[8:0];
100
101         //mux between integer input and floating-point input
102         assign trueInputA = (readFPInput) ?  {2'b0, operandAExpanded
                [23:0], 6'b0} : operandA_in[31:0];
103
104         //mux between input A and large ALU result
105         assign srfWriteValueR0 = (srfWriteSelectR0 == 1'b0) ?
                sigAluResult : trueInputA;
106         //mux between input B and large ALU result
107         assign srfWriteValueR1 = (srfWriteSelectR1 == 1'b0) ?
                sigAluResult : {2'b0, operandBExpanded[23:0], 6'b0};
108
109         //connect large register file with the large ALU
110         assign sigAluOpA_tmp = (sigAluRegOrMul) ?
                mulResultShiftedExtended : srfReadValueA;
111         assign sigAluOpA = (sigAluSrr) ? sigAluResultRegister :
                sigAluOpA_tmp;
112         assign sigAluOpB = (sigAluRegOrExpResult)?
                expAluResultRegister     : srfReadValueB;
113
```

```
114
115     //INSTANTIATIONS
116         SVDUnit svdUnitA(operandA_in, signA, isZeroA, isInfA, isNanA,
                isDenormA, operandAExpanded);
117         SVDUnit svdUnitB(operandB_in, signB, isZeroB, isInfB, isNanB,
                isDenormB, operandBExpanded);
118
119         FpuControlUnit controlUnit(.clk_in(clk_in), .reset_in(reset_in
                ), .fpuOpCode_in(fpuOpCode_in),
120                 .roundingMode_in(roundingMode_in),
121             .signA_in(signA), .signB_in(signB), .isZeroA_in(isZeroA),
                    .isZeroB_in(isZeroB),
122                 .isInfA_in(isInfA), .isInfB_in(isInfB), .isNanA_in(
                        isNanA), .isNanB_in(isNanB),
123                 .isDenormA_in(isDenormA), .isDenormB_in(isDenormB),
124             .readFPInput_out(readFPInput),
125             .expAluNegFlag_in(expAluNegFlag), .expAluZeroFlag_in(
                    expAluZeroFlag),
126             .sigAluNegFlag_in(sigAluNegFlag), .sigAluZeroFlag_in(
                    sigAluZeroFlag),
127             .guardBit_in(srfReadValueA[6]), .roundBit_in(srfReadValueA
                    [5]), .stickyBitData_in(stickyBitData),
128             .erfWriteSelectR0_out(erfWriteSelectR0), .
                    erfWriteSelectR1_out(erfWriteSelectR1),
129                 .erfWriteEnableR0_out(erfWriteEnableR0), .
                        erfWriteEnableR1_out(erfWriteEnableR1),
130                 .erfReadSelectA_out(erfReadSelectA), .
                        erfReadSelectB_out(erfReadSelectB),
131             .srfWriteSelectR0_out(srfWriteSelectR0), .
                    srfWriteSelectR1_out(srfWriteSelectR1),
132                 .srfWriteEnableR0_out(srfWriteEnableR0), .
                        srfWriteEnableR1_out(srfWriteEnableR1),
133                 .srfShiftEnableR0_out(srfShiftEnableR0), .
                        srfReadSelectA_out(srfReadSelectA),
134                 .srfReadSelectB_out(srfReadSelectB),
135             .expAluRegOrSigResult_out(expAluRegOrSigResult),
136             .expAluOp_out(expAluOpCode),
137             .errWriteEnable_out(errWriteEnable),
138             .sigAluRegOrMul_out(sigAluRegOrMul), .sigAluSrr_out(
                    sigAluSrr), .sigAluRegOrExpResult_out(
                    sigAluRegOrExpResult),
139             .sigAluOp_out(sigAluOpCode),
140             .srrWriteEnable_out(srrWriteEnable),
141             .srrShiftEnable_out(srrShiftEnable),
142             .srrShiftIn_out(srrShiftIn),
143             .maskAndShiftOp_out(maskAndShiftOp),
144             .mulEnable_out(mulEnable),
145             .shiftAndExtendOp_out(shiftAndExtendOp),
146             .outputFPResult_out(outputFPResult),
147             .resultSign_out(resultSign),
148             .resultReady_out(resultReady_out),
149             .invalidOperationDetected_out(invalidOperation_out), .
                    divisionByZeroDetected_out(divisionByZero_out),
150             .overflowDetected_out(overflow_out), .
                    underflowDetected_out(underflow_out), .
                    inexactDetected_out(inexact_out)
151             );
```

```
152
153         ExpRegisterFile expRegisterFile(clk_in, reset_in,
                erfWriteEnableR0, erfWriteEnableR1, erfWriteValueR0,
                erfWriteValueR1,
154                                         erfReadSelectA, erfReadSelectB,
                                                erfReadValueA,
                                               erfReadValueB);
155
156         ExponentALU expALU(expAluOpCode, expAluOpA, expAluOpB,
                expAluNegFlag, expAluZeroFlag, expAluResult);
157
158         SigRegisterFile sigRegisterFile(clk_in, reset_in,
                srfWriteEnableR0, srfWriteEnableR1, srfWriteValueR0,
                srfWriteValueR1,
159                                          srfShiftEnableR0,
                                                srfReadSelectA,
                                                srfReadSelectB,
                                                srfReadValueA,
                                                srfReadValueB);
160
161         SignificandALU sigALU(sigAluOpCode, sigAluOpA, sigAluOpB,
                sigAluNegFlag, sigAluZeroFlag, sigAluResult);
162
163
164         //MULTIPLIER CHAIN
165         MaskAndShift maskAndShift(maskAndShiftOp, operandAExpanded
                [23:0], operandBExpanded[23:0],
166                                   mulInputMaskedShiftedA,
                                        mulInputMaskedShiftedB);
167         ExternalMul16x16 externalMul(clk_in, reset_in, mulEnable,
                mulInputMaskedShiftedA,
168                                      mulInputMaskedShiftedB, mulResult
                                           );
169         ShiftAndExtend shiftAndExtend(shiftAndExtendOp, mulResult,
                mulResultShiftedExtended, stickyBitData);
170
171
172     // SYNCHRONOUS BLOCK - <TODO: replace with register modules?>
173         always @(posedge clk_in) begin
174             if (reset_in) begin
175                 expAluResultRegister = 0;
176                 sigAluResultRegister = 0;
177             end else begin
178                 //update exponent result register
179                 if (errWriteEnable) expAluResultRegister =
                        expAluResult;
180
181                 //update significand result register
182                 if (srrShiftEnable)
183                     sigAluResultRegister = {sigAluResultRegister
                            [30:0], srrShiftIn}; //shift a bit into SRR
184                 else begin
185                     if (srrWriteEnable) sigAluResultRegister =
                            sigAluResult;  //write ALU result into SRR
186                 end
187             end
188         end
```

```
189 endmodule
```

## A.2 Example Testbench Generation

The following C-program outputs a set of text-files with $n$ lines, each line contains two input values and the result of these operations. To make sure all combinations of input signs and rounding modes are tested, the program will generate various combinations of these. Similar programs have been written for the other floating-point operations as well.

Listing A.12: fpmul.cpp

```
1  #include <fenv.h>
2  #include <float.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdlib.h>
6
7  int main(int argc, char** argv)
8  {
9    FILE* file;
10   int testVectorCount = 1;
11   char *files[4] = {"mul-vectors-p-p.txt", "mul-vectors-p-n.txt", "mul
         -vectors-n-p.txt", "mul-vectors-n-n.txt"};
12   char *folders[4] = {"nearest/", "pinf/", "ninf/", "trunc/"};
13   int roundingModes[4] = {FE_TONEAREST, FE_UPWARD, FE_DOWNWARD,
         FE_TOWARDZERO};
14
15   if (argc>1) testVectorCount = atoi(argv[1]);
16
17   float a, b, c;
18   char buf[80];
19
20   for (int r=0; r<4; r++)
21   {
22     fesetround(roundingModes[r]);
23
24     for (int s=0; s<4; s++)
25     {
26       strcpy(buf, folders[r]);
27       strcat(buf, files[s]);
28       file = fopen(buf, "w");
29       if (file == NULL)
30       {
31     printf("Unable to create/open output file: %s\n", files[s]);
32     return -1;
33       }
34
35       //base input
36       switch (s)
37       {
38       case (0)://p-p
39           a =  4.3f;
```

```
40        b =  7.8f;
41      break;
42        case (1)://p-n
43            a =  4.3f;
44        b = -7.8f;
45      break;
46        case (2)://n-p
47            a = -4.3f;
48        b =  7.8f;
49      break;
50        case (3)://n-n
51            a = -4.3f;
52        b = -7.8f;
53      break;
54        }
55
56
57        for (int tv=0; tv<testVectorCount; tv++)
58        {
59      c = a * b;
60
61      fprintf(file, "%x\t", reinterpret_cast<int&>(a));
62      fprintf(file, "%x\t", reinterpret_cast<int&>(b));
63      fprintf(file, "%x\n", reinterpret_cast<int&>(c));
64
65      //update input
66      switch (s)
67      {
68      case (0)://p-p
69        a = 1.0073627f * a + 0.9812f;
70        b = 1.005434f * b + 0.542f;
71        break;
72      case (1)://p-n
73        a = 1.0073627f * a + 0.9812f;
74        b = 1.005434f * b - 0.542f;
75        break;
76      case (2)://n-p
77        a = 1.0073627f * a - 0.9812f;
78        b = 1.005434f * b + 0.542f;
79        break;
80      case (3)://n-n
81        a = 1.0073627f * a - 0.9812f;
82        b = 1.005434f * b - 0.542f;
83        break;
84      }
85        }
86        fclose(file);
87      }
88    }
89    return 0;
90  }
```

The resulting test-vector files can be used to test the design, by utilizing an automated testbench. An example of such a testbench is given below.

Listing A.13: fpu_top_tb.v

```
1  `timescale 1ns/1ps
2
3  `include "../global.v"
4
5  module FPU_top_tb();
6      reg clk, reset;
7      reg [3:0] opCode;
8      reg [1:0] roundingMode;
9      reg [31:0] A;
10     reg [31:0] B;
11     reg [31:0] ER; //expected result
12
13     wire resultReady;
14     wire [31:0] result;
15     wire invalidOperation, divisionByZero, overflow, underflow,
           inexact;
16
17     integer file;
18     integer mulVectorCount = 100;
19     integer addVectorCount = 100;
20     integer subVectorCount = 100;
21     integer divVectorCount = 200;
22
23     //instantiate DUT
24     FPU_top DUT(clk, reset, opCode, roundingMode, A, B, resultReady,
           result,
25                 invalidOperation, divisionByZero, overflow, underflow,
                       inexact);
26
27
28     //clock
29     parameter HCP = 10;
30     initial forever begin
31       #HCP clk =  ~clk;
32     end
33
34     initial begin
35         clk = 1'b0;
36         reset = 1'b1;
37
38         $display("----------------␣Mul␣automated␣testbench␣
               ----------------");
39         opCode = `FPU_INSTR_MUL;
40         $display("Round␣towards␣zero");
41         roundingMode = `ROUNDING_MODE_TRUNCATE;
42             file = $fopen("test/mul/trunc/mul-vectors-p-p.txt", "r");
                   runSingleFile(file, mulVectorCount);
43             file = $fopen("test/mul/trunc/mul-vectors-p-n.txt", "r");
                   runSingleFile(file, mulVectorCount);
44             file = $fopen("test/mul/trunc/mul-vectors-n-p.txt", "r");
                   runSingleFile(file, mulVectorCount);
45             file = $fopen("test/mul/trunc/mul-vectors-n-n.txt", "r");
                   runSingleFile(file, mulVectorCount);
46         $display("Round␣towards␣+Inf");
47         roundingMode = `ROUNDING_MODE_POS_INF;
48             file = $fopen("test/mul/pinf/mul-vectors-p-p.txt", "r");
                   runSingleFile(file, mulVectorCount);
```

```
49              file = $fopen("test/mul/pinf/mul-vectors-p-n.txt", "r");
                    runSingleFile(file, mulVectorCount);
50              file = $fopen("test/mul/pinf/mul-vectors-n-p.txt", "r");
                    runSingleFile(file, mulVectorCount);
51              file = $fopen("test/mul/pinf/mul-vectors-n-n.txt", "r");
                    runSingleFile(file, mulVectorCount);
52          $display("Round␣towards␣-Inf");
53          roundingMode = `ROUNDING_MODE_NEG_INF;
54              file = $fopen("test/mul/ninf/mul-vectors-p-p.txt", "r");
                    runSingleFile(file, mulVectorCount);
55              file = $fopen("test/mul/ninf/mul-vectors-p-n.txt", "r");
                    runSingleFile(file, mulVectorCount);
56              file = $fopen("test/mul/ninf/mul-vectors-n-p.txt", "r");
                    runSingleFile(file, mulVectorCount);
57              file = $fopen("test/mul/ninf/mul-vectors-n-n.txt", "r");
                    runSingleFile(file, mulVectorCount);
58          /*$display("Round towards nearest event");
59          roundingMode = `ROUNDING_MODE_NEAREST_EVEN;
60              file = $fopen("test/mul/nearest/mul-vectors-p-p.txt", "r")
                    ; runSingleFile(file, mulVectorCount);
61              file = $fopen("test/mul/nearest/mul-vectors-p-n.txt", "r")
                    ; runSingleFile(file, mulVectorCount);
62              file = $fopen("test/mul/nearest/mul-vectors-n-p.txt", "r")
                    ; runSingleFile(file, mulVectorCount);
63              file = $fopen("test/mul/nearest/mul-vectors-n-n.txt", "r")
                    ; runSingleFile(file, mulVectorCount);*/
64
65
66
67          $display("----------------␣Add/Sub␣automatic␣testbench␣
                ----------------");
68          $display("Add:");
69          opCode = `FPU_INSTR_ADD;
70              $display("Round␣towards␣zero");
71              roundingMode = `ROUNDING_MODE_TRUNCATE;
72                  file = $fopen("test/add/trunc/add-vectors-p-p.txt", "r
                        "); runSingleFile(file, addVectorCount);
73                  file = $fopen("test/add/trunc/add-vectors-p-n.txt", "r
                        "); runSingleFile(file, addVectorCount);
74                  file = $fopen("test/add/trunc/add-vectors-n-p.txt", "r
                        "); runSingleFile(file, addVectorCount);
75                  file = $fopen("test/add/trunc/add-vectors-n-n.txt", "r
                        "); runSingleFile(file, addVectorCount);
76              $display("Round␣towards␣+Inf");
77              roundingMode = `ROUNDING_MODE_POS_INF;
78                  file = $fopen("test/add/pinf/add-vectors-p-p.txt", "r"
                        );    runSingleFile(file, addVectorCount);
79                  file = $fopen("test/add/pinf/add-vectors-p-n.txt", "r"
                        );    runSingleFile(file, addVectorCount);
80                  file = $fopen("test/add/pinf/add-vectors-n-p.txt", "r"
                        );    runSingleFile(file, addVectorCount);
81                  file = $fopen("test/add/pinf/add-vectors-n-n.txt", "r"
                        );    runSingleFile(file, addVectorCount);
82              $display("Round␣towards␣-Inf");
83              roundingMode = `ROUNDING_MODE_NEG_INF;
84                  file = $fopen("test/add/ninf/add-vectors-p-p.txt", "r"
                        ); runSingleFile(file, addVectorCount);
```

```
85                  file = $fopen("test/add/ninf/add-vectors-p-n.txt", "r"
                        ); runSingleFile(file, addVectorCount);
86                  file = $fopen("test/add/ninf/add-vectors-n-p.txt", "r"
                        ); runSingleFile(file, addVectorCount);
87                  file = $fopen("test/add/ninf/add-vectors-n-n.txt", "r"
                        ); runSingleFile(file, addVectorCount);
88              /*$display("Round towards nearest even");
89              roundingMode = `ROUNDING_MODE_NEAREST_EVEN;
90                  file = $fopen("test/add/nearest/add-vectors-p-p.txt",
                        "r"); runSingleFile(file, addVectorCount);
91                  file = $fopen("test/add/nearest/add-vectors-p-n.txt",
                        "r"); runSingleFile(file, addVectorCount);
92                  file = $fopen("test/add/nearest/add-vectors-n-p.txt",
                        "r"); runSingleFile(file, addVectorCount);
93                  file = $fopen("test/add/nearest/add-vectors-n-n.txt",
                        "r"); runSingleFile(file, addVectorCount);
94              */
95          $display("----------------");
96          $display("Sub:");
97          opCode = `FPU_INSTR_SUB;
98              $display("Round␣towards␣zero");
99              roundingMode = `ROUNDING_MODE_TRUNCATE;
100                 file = $fopen("test/sub/trunc/sub-vectors-p-p.txt", "r
                        "); runSingleFile(file, subVectorCount);
101                 file = $fopen("test/sub/trunc/sub-vectors-p-n.txt", "r
                        "); runSingleFile(file, subVectorCount);
102                 file = $fopen("test/sub/trunc/sub-vectors-n-p.txt", "r
                        "); runSingleFile(file, subVectorCount);
103                 file = $fopen("test/sub/trunc/sub-vectors-n-n.txt", "r
                        "); runSingleFile(file, subVectorCount);
104             $display("Round␣towards␣+Inf");
105             roundingMode = `ROUNDING_MODE_POS_INF;
106                 file = $fopen("test/sub/pinf/sub-vectors-p-p.txt", "r"
                        );    runSingleFile(file, subVectorCount);
107                 file = $fopen("test/sub/pinf/sub-vectors-p-n.txt", "r"
                        );    runSingleFile(file, subVectorCount);
108                 file = $fopen("test/sub/pinf/sub-vectors-n-p.txt", "r"
                        );    runSingleFile(file, subVectorCount);
109                 file = $fopen("test/sub/pinf/sub-vectors-n-n.txt", "r"
                        );    runSingleFile(file, subVectorCount);
110             $display("Round␣towards␣-Inf");
111             roundingMode = `ROUNDING_MODE_NEG_INF;
112                 file = $fopen("test/sub/ninf/sub-vectors-p-p.txt", "r"
                        ); runSingleFile(file, subVectorCount);
113                 file = $fopen("test/sub/ninf/sub-vectors-p-n.txt", "r"
                        ); runSingleFile(file, subVectorCount);
114                 file = $fopen("test/sub/ninf/sub-vectors-n-p.txt", "r"
                        ); runSingleFile(file, subVectorCount);
115                 file = $fopen("test/sub/ninf/sub-vectors-n-n.txt", "r"
                        ); runSingleFile(file, subVectorCount);
116             /*$display("Round towards nearest event");
117             roundingMode = `ROUNDING_MODE_NEAREST_EVEN;
118                 file = $fopen("test/sub/nearest/sub-vectors-p-p.txt",
                        "r"); runSingleFile(file, subVectorCount);
119                 file = $fopen("test/sub/nearest/sub-vectors-p-n.txt",
                        "r"); runSingleFile(file, subVectorCount);
```

```
120                 file = $fopen("test/sub/nearest/sub-vectors-n-p.txt",
                        "r"); runSingleFile(file, subVectorCount);
121                 file = $fopen("test/sub/nearest/sub-vectors-n-n.txt",
                        "r"); runSingleFile(file, subVectorCount);
122             */
123
124         $display("----------------␣DIV␣automatic␣testbench␣
                ----------------");
125         opCode = `FPU_INSTR_DIV;
126         $display("Round␣towards␣zero");
127         roundingMode = `ROUNDING_MODE_TRUNCATE;
128             file = $fopen("test/div/trunc/div-vectors-p-p.txt", "r");
                    runSingleFile(file, 1);
129             file = $fopen("test/div/trunc/div-vectors-p-n.txt", "r");
                    runSingleFile(file, divVectorCount);
130             file = $fopen("test/div/trunc/div-vectors-n-p.txt", "r");
                    runSingleFile(file, divVectorCount);
131             file = $fopen("test/div/trunc/div-vectors-n-n.txt", "r");
                    runSingleFile(file, divVectorCount);
132         /*$display("Round towards +Inf");
133             roundingMode = `ROUNDING_MODE_POS_INF;
134                 file = $fopen("test/div/pinf/div-vectors-p-p.txt", "r
                        ");     runSingleFile(file, divVectorCount);
135                 file = $fopen("test/div/pinf/div-vectors-p-n.txt", "r
                        ");     runSingleFile(file, divVectorCount);
136                 file = $fopen("test/div/pinf/div-vectors-n-p.txt", "r
                        ");     runSingleFile(file, divVectorCount);
137                 file = $fopen("test/div/pinf/div-vectors-n-n.txt", "r
                        ");     runSingleFile(file, divVectorCount);
138             $display("Round towards -Inf");
139             roundingMode = `ROUNDING_MODE_NEG_INF;
140                 file = $fopen("test/div/ninf/div-vectors-p-p.txt", "r
                        "); runSingleFile(file, divVectorCount);
141                 file = $fopen("test/div/ninf/div-vectors-p-n.txt", "r
                        "); runSingleFile(file, divVectorCount);
142                 file = $fopen("test/div/ninf/div-vectors-n-p.txt", "r
                        "); runSingleFile(file, divVectorCount);
143                 file = $fopen("test/div/ninf/div-vectors-n-n.txt", "r
                        "); runSingleFile(file, divVectorCount); */
144             /*$display("Round towards nearest event");
145             roundingMode = `ROUNDING_MODE_NEAREST_EVEN;
146                 file = $fopen("test/div/nearest/div-vectors-p-p.txt",
                        "r"); runSingleFile(file, subVectorCount);
147                 file = $fopen("test/div/nearest/div-vectors-p-n.txt",
                        "r"); runSingleFile(file, subVectorCount);
148                 file = $fopen("test/div/nearest/div-vectors-n-p.txt",
                        "r"); runSingleFile(file, subVectorCount);
149                 file = $fopen("test/div/nearest/div-vectors-n-n.txt",
                        "r"); runSingleFile(file, subVectorCount);
150             */
151
152
153         $display("----------------");
154         #20
155         $finish;
156     end
157
```

```
158
159     task runSingleFile;
160         input integer file;
161         input integer vectorCount;
162         integer status, cnt, errorCount;
163     begin
164         cnt = 0;
165         errorCount = 0;
166         while (cnt < vectorCount) begin
167             status = $fscanf(file, "%x\t%x\t%x\n", A[31:0], B[31:0],
                    ER[31:0]);
168             #(2*HCP) reset = 1'b0;
169             @(posedge resultReady) #1;
170             if (ER !== result) begin
171                 $display("Vector %d: Wrong result!", cnt);
172                 $display("A:  %b\t%x\t%b\n", A[31], A[30:23], A[22:0])
                        ;
173                 $display("B:  %b\t%x\t%b\n", B[31], B[30:23], B[22:0])
                        ;
174                 $display("ER: %b\t%x\t%b\n", ER[31], ER[30:23], ER
                        [22:0]);
175                 $display("R:  %b\t%x\t%b\n", result[31], result
                        [30:23], result[22:0]);
176                 errorCount = errorCount + 1;
177             end else begin
178                 /*$display("Vector %d: Correct result", cnt);
179                 $display("A:  %b\t%x\t%b\n", A[31], A[30:23], A[22:0])
                        ;
180                 $display("B:  %b\t%x\t%b\n", B[31], B[30:23], B[22:0])
                        ;
181                 $display("ER: %b\t%x\t%b\n", ER[31], ER[30:23], ER
                        [22:0]);
182                 $display("R:  %b\t%x\t%b\n", result[31], result
                        [30:23], result[22:0]);*/
183             end
184             reset = 1'b1;
185             cnt = cnt + 1;
186         end
187         $display("Finished, %d vectors simulated, %d error(s)", cnt,
                errorCount);
188         $fclose(file);
189       end
190   endtask
191 endmodule
```