Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application

A Thesis Presented

by

Pavle Belanović

to

Department of Electrical and Computer Engineering

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the field of

Electrical Engineering

Northeastern University

Boston, Massachusetts

May 2002

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application.

Author: Pavle Belanović.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Miriam Leeser	Date
Thesis Reader: Dana Brooks	Date
Thesis Reader: Waleed Meleis	Date
Chairman of Department: Fabrizio Lombardi	Date

Graduate School Notified of Acceptance:

Director of the Graduate School	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application.

Author: Pavle Belanović.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Miriam Leeser	Date
Thesis Reader: Dana Brooks	Date
Thesis Reader: Waleed Meleis	Date
Chairman of Department: Fabrizio Lombardi	Date

Graduate School Notified of Acceptance:

Director of the Graduate School	Date

Copy Deposited in Library:

Reference Librarian	Date

Contents

Abstract

Due to inherent limitations of the fixed-point representation, it is sometimes desirable to perform arithmetic operations in the floating-point format. Although an established standard for floating-point arithmetic exists, optimal hardware implementations of algorithms require use of floating-point formats different from the ones specified in the standard. A library of fully parameterized hardware modules for floating-point format control, arithmetic operators and conversion to and from any fixed-point format are presented. Synthesis results for arithmetic operator modules in several floating-point formats, including the IEEE single precision format, are also shown. The library supports creation of custom format floating-point pipelines, as well as hybrid fixed and floating-point implementations. A hybrid implementation of the K-means clustering algorithm for multispectral and hyperspectral image processing is presented, illustrating the use of the library. Synthesis and processing results for both implementations are shown and compared.

Acknowledgements

To Julia.

Through the course of my studies, I have been fortunate to enjoy unwavering support and encouragement of my parents, Dragan and Zora, and sister, Ana. To them I owe a debt of gratitude that can scarcely be contained in this acknowledgement.

I am grateful to my advisor, Dr. Miriam Leeser, for introducing me to the field of image processing, as well as helping me expand my knowledge in all other areas of this research. Her guidance in writing this thesis was indispensable. Most importantly, I am grateful to Dr. Leeser for giving me the chance and necessary support that made this research possible.

It has been a pleasure to cooperate with James Theiler, Maya Gokhale, Kevin McCabe and John Szymanski of Los Alamos National Laboratories and I wish to thank them for their support.

A special acknowledgement is due to Michael Estlick who introduced me to the Rapid Prototyping Laboratory, helped me get a foothold in Boston and taught me about the K-means clustering algorithm, as well as many other topics. His legacy is difficult to live up to.

I gratefully acknowledge the help of Deniz Balkan, Murat Biçer, Chris Butcher, Thomas Henley and Emre Şafak in proof-read this document, in some cases several times.

This research was funded in part by a grant from Los Alamos National Laboratory.

Chapter 1

Introduction

Many image and signal processing applications benefit from acceleration with reconfigurable hardware. This acceleration results from the exploitation of fine-grained parallelism available in reconfigurable hardware. Custom circuits built for these applications in reconfigurable hardware process values in fixed or floating-point formats. Minimizing bitwidths of signals carrying those values makes more parallel implementations possible and reduces power dissipation of the circuit. Arbitrary fixed-point formats are not difficult to implement and are in common use. Because of the inherent complexity of the floating-point representation, it is no less desirable, but much harder to implement arbitrary floating-point formats. This thesis presents a library of hardware modules that makes implementation of custom designs with arbitrary floating-point formats possible.

In this chapter, the reader is introduced to fixed and floating-point representations, reconfigurable hardware used to implement all the designs presented, motivation for this work and a survey of related work.

1.1 Fixed-Point Arithmetic

Fixed-Point Format

One of the most widely implemented formats for representing and storing numerical values in the binary number system is the fixed-point format.

Every numerical value is composed of an integer part and a fractional part and the delimitation between the two is referred to as the radix point. In the fixed-point format, the radix point is always implied to be in a predetermined position. The format thus gets its name from the fixed location of the radix point. Usually, the radix point is assumed to be located immediately to the right of the least significant digit, in which case only integer values are represented (no digits represent the fractional part).

Signed and Unsigned Representations

Two alternative schemes of representing values exist in the fixed-point format: signed and unsigned. In the unsigned fixed-point format, only values larger than or equal to zero can be represented. Thus, the sign of the value is redundant information and is not stored. For a given bitwidth n, integer values between 0 and $2^n - 1$ can be represented in the unsigned number format.

In the signed fixed-point format, however, we aim to represent both positive and negative values. Thus, the sign of the numerical value has to be stored. Signed values are represented in sign-and-magnitude, one's complement or two's complement notation, where the latter is most popular for hardware implementations. For a given bitwidth n, integer values between -2^{n-1} and $2^{n-1} - 1$ can be represented in the signed two's complement fixed-point format.

1.2 Floating-Point Arithmetic

Limited Range in Fixed-Point Format

Simplicity of arithmetic operations in the fixed-point format makes it a popular implementation choice for many algorithms. However, scientific computation often involves very large and very small values, which are not easily represented in fixed-point format. In other words, scientific applications often have larger range than fixed-point format can easily accommodate. Scaling of fixed-point numbers by placing the radix point in positions other than immediately to the right of the least significant digit is possible. This may let us represent very small or very large values, but not both without very large bitwidths. Of course, in such schemes, increased bitwidth brings about increased complexity of arithmetic operations. Hence, fixed-point formats are said to suffer from limited range.

Floating-Point Format

The floating-point format is the most common way of representing real numbers in computer systems. It ameliorates the problem of limited range and is suitable to most scientific algorithms. In essence, the floating-point format is similar to the scientific notation of numbers, such as -1.35×10^6. There are three fields in the representation of a value in the floating-point format: sign s, exponent e and fraction f. Thus, every floating-point value can be defined by

$$(-1)^s \times 1.f \times 2^{e-BIAS}$$

Please note that the exponent is *biased*, meaning that the stored value is shifted from 0 by a known bias value, depending on the bitwidth of the exponent field in the particular format. Given that the exponent bitwidth is *exp_bits*, we can represent exponent values from $-2^{exp\_bits-1}+1$ to 2^{exp_bits-1} by assigning the value $2^{exp\_bits-1}-1$ to the bias. It is worth noting that the bias value is not constant and changes with exponent bitwidth.

The bitwidth alignment of the three fields is shown in Figure 1.1. The distinction

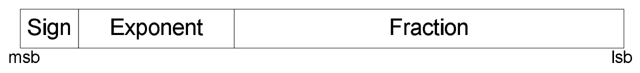

Figure 1.1: Alignment of fields in a floating-point format

between terms *fraction* and *mantissa* is that fraction represents only the portion of the mantissa to the right of the radix point (fractional part). Naturally, a tradeoff exists in total bitwidth between smaller width requiring less hardware resources and higher width providing better precision. Also, within a certain total bitwidth, it is possible to assign various combinations of values to the exponent and fraction fields (see Figure 1.1). Wider exponent fields brings higher range and wider fraction fields brings higher precision.

IEEE Standard And Other Formats

The Institute of Electrical and Electronics Engineers (IEEE) issued standard 754 in 1985, specifying details of implementing binary floating-point arithmetic. This standard details four floating-point formats - basic and extended, each in single and double precision bitwidths. Most implementations of floating-point arithmetic adhere to one or more of these standard formats, though few follow the standard absolutely.

However, optimal implementations of algorithms may not always require bitwidths as specified by the standard. In fact, it is often the case that much smaller bitwidths than those specified in the 754 standard are sufficient to provide desired precision and occupy less resources than the full standard bitwidth implementation. In custom hardware designs, it is possible, and indeed desirable, to have full control and flexibility over the floating-point format that is implemented. Reducing datapath bitwidths to their optimal values enables design of more parallel architectures and implementation of larger designs. This thesis presents a library of variable precision floating-point components to support this. These components have been implemented in reconfigurable hardware.

1.3 Reconfigurable Hardware

Field Programmable Gate Array (FPGA)

Field Programmable Gate Arrays (FPGAs) are integrated circuits with a flexible architecture, such that their structure can be programmed by the designer. FPGAs are composed of an array of hardware resources called configurable logic blocks (CLBs). The designer creates the functionality of the overall circuit by configuring CLBs to perform appropriate logic functions. Hence, FPGAs are a form of reconfigurable hardware, combining flexibility similar to software with the speed of specialized hardware.

Designs mapped to FPGAs are usually described in hardware description languages, such as VHDL or Verilog. In this project, all hardware descriptions are written in VHDL. A number of software tools exists to aid the designer in mapping the high-level description of the design in VHDL to the logic level of each CLB. Such tools perform synthesis, mapping, placing and routing of the design to the hardware. For synthesis and mapping of all designs in this project we used Synplicity Pro from Synplify. Mapping, placing and routing of the designs was done using Xilinx Alliance tools. In order to verify the fidelity of the VHDL descriptions to the intended functionality, all designs in this project were simulated with Mentor Graphics ModelSim prior to being implemented in hardware.

The Wildstar Reconfigurable Computing Engine

Reconfigurable computing is characterized by use of hardware elements that have reconfigurable architectures, as opposed to general purpose computing which uses hardware elements with fixed architectures.

Many reconfigurable computing systems are based on one or more FPGAs connected to a number of memory banks. All designs presented in this project are implemented on the Wildstar reconfigurable computing engine from Annapolis Micro Systems. Figure 1.2 shows the structure of this board.

Some of the main features of the Wildstar board are:

- 3 Xilinx VIRTEX XCV1000 FPGAs,
- total of 3 million system gates,
- 40 Mbytes of SRAM,
- 1.6 Gbytes/sec I/O bandwidth,
- 6.4 Gbytes/sec memory bandwidth,
- processing clock rates up to 100MHz.

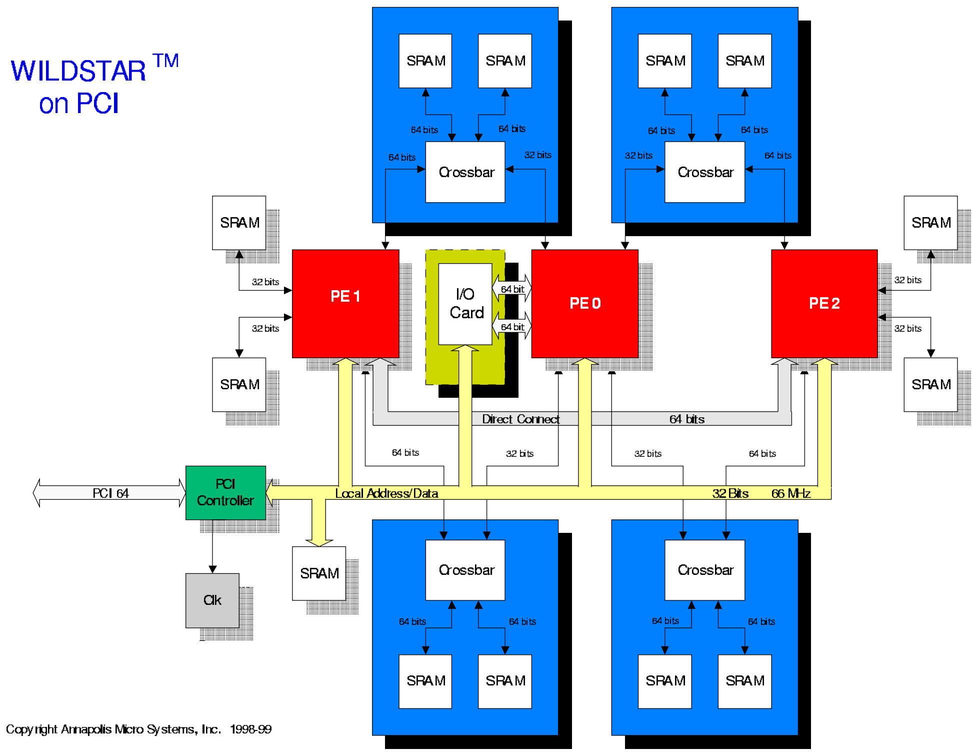

Figure 1.2: Structure of the Wildstar reconfigurable computing engine

1.4 Motivation for This Work

Accelerating Algorithms

One of the main applications of reconfigurable hardware is to accelerate algorithms implemented on general purpose processors. Candidate algorithms are initially implemented purely in software that runs on a general purpose processor. Acceleration of the algorithm is achieved by partitioning of the algorithm into portions to be implemented in software on a general purpose processor and portions to be implemented in reconfigurable hardware.

Typically, the parts of the algorithm that are assigned to software are serial or procedural in nature, while the highly parallel, computational parts of the algorithm get implemented in hardware. Custom datapaths are created in reconfigurable hardware to achieve desired functionality. Communication with the general purpose processor is done through memory banks and/or register tables in reconfigurable hardware, both of which are accessible by the custom hardware and the general purpose processor.

Typical speedups are in the range of 10-100 times for algorithms implemented in custom hardware compared to software implementation on the general purpose processor. Highly regular and parallel applications such as DSP (filters, media processing, FFT), network controllers and similar applications are prime candidates for this form of acceleration.

Control Over Bitwidths

Custom datapaths that are designed in reconfigurable hardware often use either fixed-point or floating-point arithmetic. In either case optimal signal bitwidths throughout the custom datapath are application-specific and depend on the values they carry. Having datapath bitwidths that are too wide for the variables they represent is a waste of resources on the FPGA, especially so in operating elements that are highly sensitive to bitwidth, such as multipliers. Conversely, datapath bitwidths that are too narrow for the variables they represent result in erroneous outputs and in some cases malfunction of the overall circuit.

Hence, it is important to have fine-grained control over datapath bitwidths throughout the design process. This means that all logic elements that constitute the datapath need to be parameterized by their bitwidth. In other words, when a designer puts together a custom datapath, (s)he specifies the exact bitwidth of each element.

Parameterized Elements

Bitwidth flexibility is easy to achieve in fixed-point arithmetic and logic elements for various functions are available to the designer in any bitwidth. In floating-point arithmetic, however, the presence of three distinct fields with complex interactions makes it difficult to represent logic functions for every possible bitwidth. It is the aim of this work to produce a library of parameterized hardware modules for floating-point arithmetic.

To completely describe any floating-point format, two values suffice: width of the exponent field (*exp_bits*) and width of the fraction (or mantissa) field (*man_bits*). The sign field always has a bitwidth of 1. Hence, the total bitwidth of any floating-point format is given by $1 + exp\_bits + man\_bits$. These parameters allow for the creation of a wide range of different modules through a single VHDL description. Values of the parameters help resolve the description at compile time and ensure synthesis of the correct amount of logic needed to perform the function of the module for the given format.

1.5 Related Work

One of the earliest investigations into using FPGAs to implement floating-point arithmetic was done by Fagin et al. [4], who in 1994 showed that implementing IEEE single precision operators was possible, but also impractical on then current FPGA technology. The circuits designed by the authors were an adder and a multiplier and both had full implementation of all four rounding modes specified by the IEEE 754 standard. Area was the critical constraint, with the authors reporting that no device in existence could contain a single precision multiplier circuit. Therefore, the authors propose adopting smaller, custom formats which may be more appropriate to FPGA architectures than the full IEEE formats.

This line of thought was expanded on by the significant work of Shirazi et al. [17] who suggested application-specific formats for image and DSP algorithms in widths of 16 (1-6-9) and 18 (1-7-10) bits, as opposed to the full 32 (1-8-23) bits in the IEEE standard. Modules for addition/subtraction, multiplication and division were presented, though no work was done on implementing rounding or error-handling.

Another significant work came from Louca et al.[11] in which the authors, building on the work of Shirazi and others, abstract the normalization operation away from the actual arithmetic operators, in an effort to conserve area. Only IEEE single precision addition and multiplication modules were implemented, but neither contained circuitry for normalizing the output values. Rather, a separate normalization unit served several operator modules by alternating between them in time. No rounding capability was implemented by the authors, due to area constraints.

In an effort to expand the capabilities of existing architectures, Ligon et al. [8] presented IEEE single precision adder and multiplier circuits on the then newly available Xilinx 4000 series FPGAs. Both circuits supported rounding to nearest, but did not use a separate normalizing unit. Similar work by Stamoulis et al. [18] presented IEEE single precision adder/subtractor, multiplier and division circuits. However, the authors do not present any rounding capability, as the intended application (3D computer graphics) does not require this functionality. Again, the work does not make use of a separate normalization unit.

Two works by Sahin et al.[16][15] present adder, subtractor, multiplier and accumulator circuits, but again only in IEEE single precision format. Also, rounding capability is not implemented. The authors introduce the concept of providing pipeline synchronization signals. Namely, all floating-point modules presented had a *ready* signal on each input and a *done* signal on the output, which helps synchronize the flow of data through pipelines made by the modules. Essentially, when the *ready* signal is high, the module processes the input data, which is then valid. Upon completion, the module makes the output values available on its data outputs and signals their validity with a *done* signal going high. The authors present additional modules, including multiply-accumulate (MAC), in their second publication, with no other major changes.

Recent work by Dido et al.[2] discusses optimization of datapaths, especially in image and video processing applications such as high definition television (HDTV). This datapath optimization is achieved by providing flexible floating-point formats that are optimal for every signal in the datapath. However, their work presents a non-general floating point format that is primarily used to represent scaled integer values, because this suits the particular application. Hence, the format cannot represent any of the IEEE formats. Hardware modules presented by these authors are format converters, similar to those shown in this work, as well as an inversion module, all without support for rounding or error handling. Their format contains no sign bit or bias of the exponent.

The floating-point formats in our work are a generalized superset of all these formats. It includes all the IEEE formats as particular instances of exponent and mantissa bitwidths, as well as the flexible floating-point format presented by Dido et al.[2] and the two formats by Shirazi et al.[17]. Also, we abstract normalization as well as rounding functionality into a separate unit with a choice of rounding to zero and rounding to nearest. In this way, as Dido et al.[2] explained, we make use of non-normalized values between arithmetic operators in pipelines, which in no way compromises the results. Normalization is then executed after an arbitrarily long chain of operators and not after every operation. All our modules are equipped with *ready* and *done* pipeline synchronization signals, as well as limited error handling capabilities which are propagated through the pipeline with the results.

1.6 Conclusion

This chapter presented an introduction to fixed and floating-point arithmetic, reconfigurable hardware used to implement all the modules in the library and motivation for the research presented here, as well as a survey of related work. The next chapter will introduce all the hardware modules that make up the library.

Chapter 2

Hardware Modules

In this chapter, the hardware modules which were designed as part of this work are presented, including their functions, structures and use. Low level modules, which are used as building blocks for top level modules that constitute the library, are described in Section 2.1. Library modules used for floating-point format control are presented in Section 2.2. Floating-point arithmetic operator modules are described in Section 2.3. Library modules for conversion between fixed and floating-point representations of values are described Section 2.4. Following these discussions of individual modules, Section 2.5 presents an example of assembly of library modules to build a complete IEEE single precision adder. Testing of the hardware modules is described in Section 2.6, while the results of synthesis experiments conducted on operator modules are presented in Section 2.7.

A complete list of all the hardware modules presented in this chapter is found in Appendix A, along with the VHDL entity for each. Names, functions and latencies (in clock cycles) of all modules that represent the parameterized library for floating-point arithmetic are shown in Table 2.1. All VHDL entity definitions in Appendix A present

Table 2.1: Floating-point hardware modules and their latency in clock cycles

Module	Function	Latency
`denorm`	Introduction of implied integer digit	0
`rnd_norm`	Normalizing and rounding	2
`fp_add`	Addition	4
`fp_sub`	Subtraction	4
`fp_mul`	Multiplication	3
`fix2float`	Unsigned fixed-point to floating-point conversion	4
	Signed fixed-point to floating-point conversion	5
`float2fix`	Floating-point to unsigned fixed-point conversion	4
	Floating-point to signed fixed-point conversion	5

`generic` and `port` declarations. The former contain parameters for the circuit, while the latter contains inputs and outputs. Parameters are compile-time variables that determine the amount and structure of logic needed to implement the circuit. Inputs and outputs are run-time variables of the circuit.

2.1 Low Level Hardware Modules

The hardware modules that constitute the parameterized library for floating-point arithmetic rely on the existence of some basic logic functions, such as multiplexers and fixed point adders. These basic building blocks are described in this section.

These building blocks that perform functions commonly encountered in logic design all share one characteristic - parameterization. All of them have parameterized functionality, so that they can be used to build higher level modules that are themselves parameterized. Details of parameterization of each building block are described in the following sections.

Fixed-Point Adder and Fixed-Point Subtractor

Many floating-point algorithms require the implementation of fixed-point addition as part of the overall algorithm. Examples are rounding, floating-point addition and conversion between fixed and floating-point formats. A dedicated module `parameterized_adder` has been written to provide this functionality. It is parameterized by the width of the input and, consequently, output signals. Carry input and carry output signals are also provided.

Another frequent use of this operation is fixed-point subtraction. Module `parameterized_subtractor` provides this functionality, parameterized by the width of the input and output signals.

Fixed-Point Multiplier

Fixed-point multiplication is needed by floating-point algorithms - in particular, floating-point multiplication. Hence, the module `parameterized_multiplier` was designed, parameterized by the width of the input signals. The output signal, which is the product of the input values, has twice the bitwidth of the input signals.

Variable Shifter

Through several floating-point algorithms, there exists a need for shifting signals, left or right, by a variable number of bits. Such functionality is usually implemented through variable shifter circuits, which accept two inputs: one for the signal to be shifted and the other to indicate the distance through which the first is to be shifted. The module `parameterized_variable_shifter` is parameterized by the width of the input, the width of the controlling signal and the direction of shifting (left or right).

Delay Block

Module `delay_block` has been developed to aid in the pipeline assembly for all the modules in this work. Its function is to delay any signal by a given number of clock cycles between its input and output. In this way, the designer can synchronize signals to produce correctly pipelined modules. Two parameters control the synthesis of the `delay_block` module: bitwidth of the input and output signals and the number of cycles the input is delayed.

Absolute Value

When handling signed fixed-point signals, it is sometimes necessary to derive their absolute value, e.g. in converting from signed fixed-point representation to the floating-point representation of a number. Thus, the module `parameterized_absolute_value` has been developed to accept a signed fixed-point value at its input and produce the absolute value of the input signal on its output. This module is parameterized by the width of its input and output signals.

Priority Encoder

Prior to many variable shifting operations, it is necessary to determine the position of the most significant bit that has value '1' in a given signal. Examples of high level algorithms that require this functionality are normalizing and converting from fixed-point to floating-point representations of a number.

Thus, module `parameterized_priority_encoder` has been developed, taking a signal to be examined on its input and producing the value, in unsigned fixed-point representation, of the index of the most significant '1' in the input signal. The module is parameterized by the width of the input signal, as well as the width of the output signal.

Multiplexer

One of the most frequently used low level logic functions throughout this work is multiplexing. Virtually every high level module uses multiplexing at some step in its algorithm by making one or more instances of the module `parameterized_mux`. This module is parameterized by the width of its two inputs and thus the width of its one output, while the select input has the bitwidth of 1.

Comparator

Another highly reused low-level logic function is comparison of unsigned fixed-point numbers. Higher level modules like floating-point addition and normalization use the module `parameterized_comparator`. This module is parameterized by the width of the fixed-point signals that are to be compared, and produces three boolean outputs, exactly one of which is always high, while the others are low. These outputs indicate whether input A is larger, equal to or smaller than input B.

2.2 Format Control Hardware Modules

Denormalization

Every floating-point value can be represented in many ways:

$$-24.97 \times 10^{12} = -2.497 \times 10^{13} = -0.2497 \times 10^{14} \ldots$$

The *normalized* form of a floating-point value is defined in the same way as in standard scientific notation: it is the form in which exactly one non-zero digit forms the integer part of the mantissa. In the given example, this is the expression in the middle.

When the binary number system is used, the only non-zero digit is '1' and, hence, the integer part of normalized floating-point numbers is '1'. Since this is redundant information, only the fractional part of the number is stored. The integer part of the value is then referred to as the *implied '1'*. While this provides efficiency in storage, the implied '1' is necessary to carry out arithmetic operations on the number and must be re-introduced, which is the function of the `denorm` module.

However, by convention formally described in the IEEE standard 754, the value zero is represented by all zero exponent and mantissa fields, with the sign bit as either '0' or '1' (± 0). The full-mantissa representation of the zero value is composed of an all-zero fractional part and a 0 (not 1) for the integer part. This is the only exceptional circumstance that can arise.

Hence, the `denorm` module inserts the implied '1' into the representation, between the mantissa and exponent fields, unless the value being processed is zero. In the latter case, it will insert a '0'. Implementation of this functionality is a simple n-input OR gate, taking as its Boolean inputs all the bits of the exponent and mantissa fields of the floating-point signal. The output of the gate is then inserted into the floating-point signal, between the exponent and the mantissa fields. Due to its relatively simple function, this module is purely combinational and is not pipelined, but is parameterized by the bitwidths of the exponent and mantissa fields.

Rounding and Normalizing

Subsequent to all arithmetic operations performed on the floating-point value, it is necessary to return it to the standard, normalized form, before presenting the result for storage or further processing by external units. This is the function of the `round_norm` module, whose structure is shown in Figure 2.1. There are two parts to

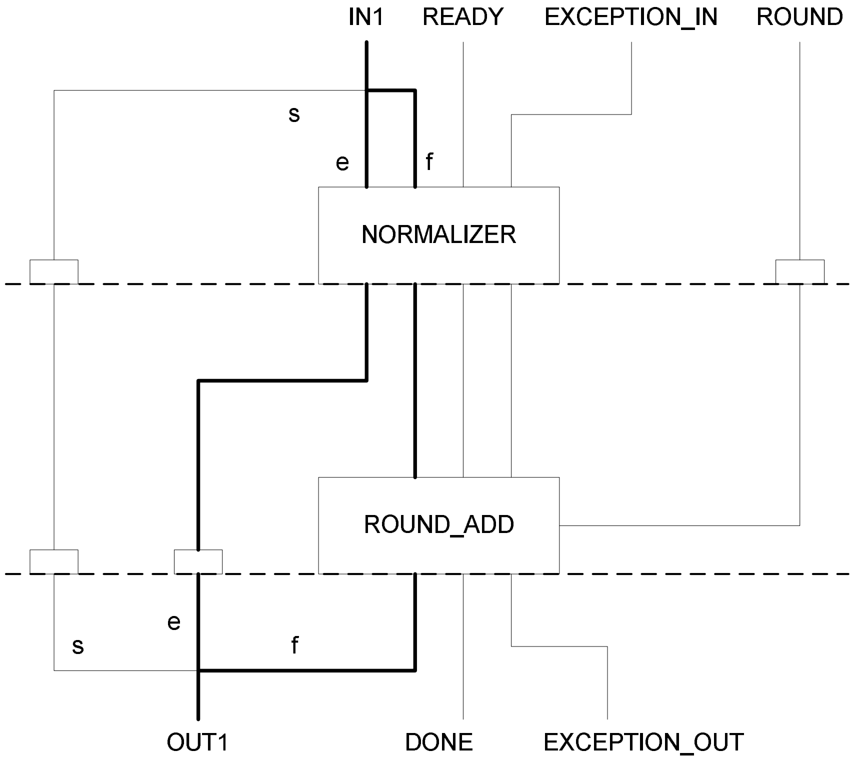

Figure 2.1: Rounding and normalizing

the functioning of the `round_norm` module: normalizing (submodule `normalizer`) and rounding (submodule `round_add`). Normalizing a floating-point value refers to shifting left its mantissa until its MSB is '1', while decrementing the exponent for every bit the mantissa was shifted. In this way, the floating-point value has been brought into its *normalized* form.

During processing, mantissa bitwidth will normally increase, as in the introduction of guard digits during addition, for example. Hence, to return the post-processing floating-point value into the correct format, its mantissa bitwidth must be reduced to its original size. However, reduction of bitwidth introduces the need for rounding, because the fraction value needs to be converted into a less precise representation. The IEEE standard specifies four rounding modes:

- round to zero,
- round to nearest,
- round to positive infinity ($+\infty$), and
- round to negative infinity ($-\infty$).

Out of these four, the first two modes of rounding are implemented in the `round_norm` module, with rounding to nearest being the default, as it is in the IEEE standard 754.

Rounding to zero is the simplest form of rounding to implement, because it is represented by simple truncation. For both positive and negative floating-point values, removal of low-order bits from the fractional part of the mantissa can only reduce the overall magnitude of the number, thereby bringing it closer to zero.

However, rounding to zero introduces significant rounding error in some instances, which can be avoided by rounding to nearest. As can be seen in Figure 2.2, where the bitwidth of the mantissa is reduced by two bits, rounding to zero assigns number labelled A with value 100011 to rounded value 1000, although it is much closer to rounded value 1001.

Rounding to nearest tackles this problem and minimizes rounding error by assigning every value to its closest rounded value. Thus, A gets assigned to 1001, not 1000. An issue of equidistant numbers arises in rounding to nearest: if a number is equally distant to two rounded values, which will be chosen? In our implementation, the equidistant value is assigned to the higher rounded value, because this simplifies the hardware implementation. This is illustrated in Figure 2.2 by number labelled B,

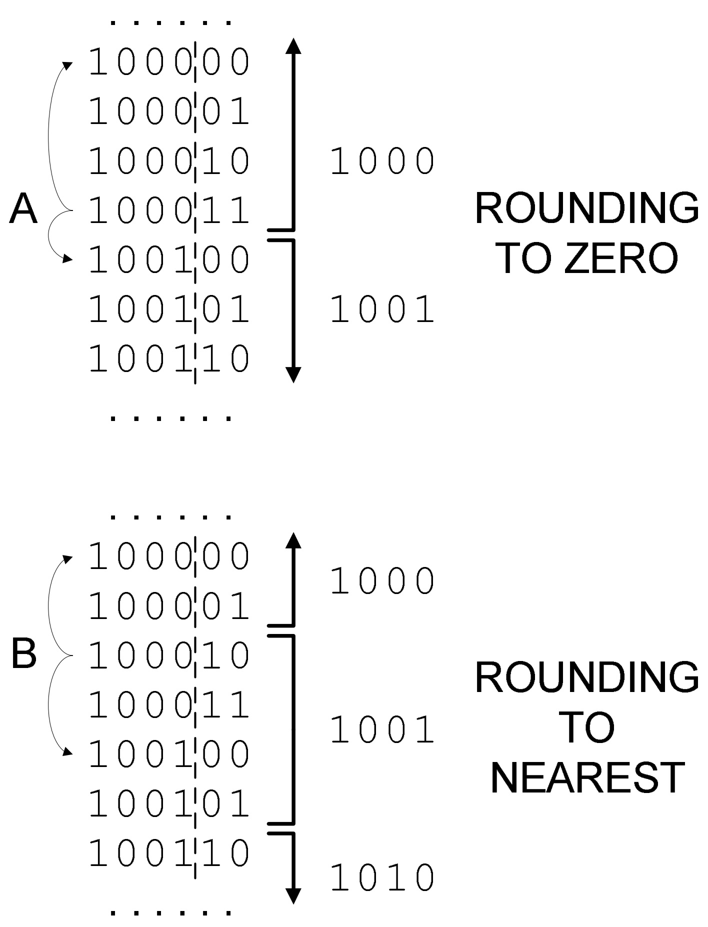

Figure 2.2: Rounding to zero and nearest

which is equally distant to rounded value 1000 and 1001, but is assigned to the higher one.

Both forms of rounding are implemented in the `rnd_norm` module dynamically through a `round` signal. When this signal has value '0', it indicates rounding to zero and the rounding addition is disabled, thus just truncating the inputs to the correct fraction bitwidth. Inversely, a high `round` signal indicates rounding to nearest, where rounding addition is enabled and inputs are rounded to the nearest rounded value.

The final function of the `rnd_norm` module is to remove the integer part of the mantissa field, or in other words remove the "implied '1'", thus returning the floating-point value completely into the original format.

Module `rnd_norm` is parameterized by three values: the width of the exponent field, the width of the mantissa field on the input and the width of the mantissa field on the output. The difference between the last two parameters indicates the number of bits of the mantissa to be removed during rounding.

2.3 Arithmetic Operator Hardware Modules

Addition

Addition is one of the most computationally complex operations in floating-point arithmetic. The algorithm of the addition operation for floating-point numbers is composed of four steps:

- ensure that the operand with larger magnitude is on input 1 (`swap`),
- align the mantissas (`shift_adjust`),
- add or subtract the mantissas (`add_sub`), and
- shift the result mantissa right by one bit and increment the exponent if addition overflow occurred (`correction`).

Each of the four steps of the algorithm is implemented in a dedicated module, shown above in brackets. The four sub-modules are assembled into the overall `fp_add` module as shown in Figure 2.3.

The `swap` submodule compares the exponent and mantissa fields of the input variables. Based on these two comparisons, the two floating-point inputs are multiplexed to the outputs. If the exponent field of input A is larger, or the exponent fields are

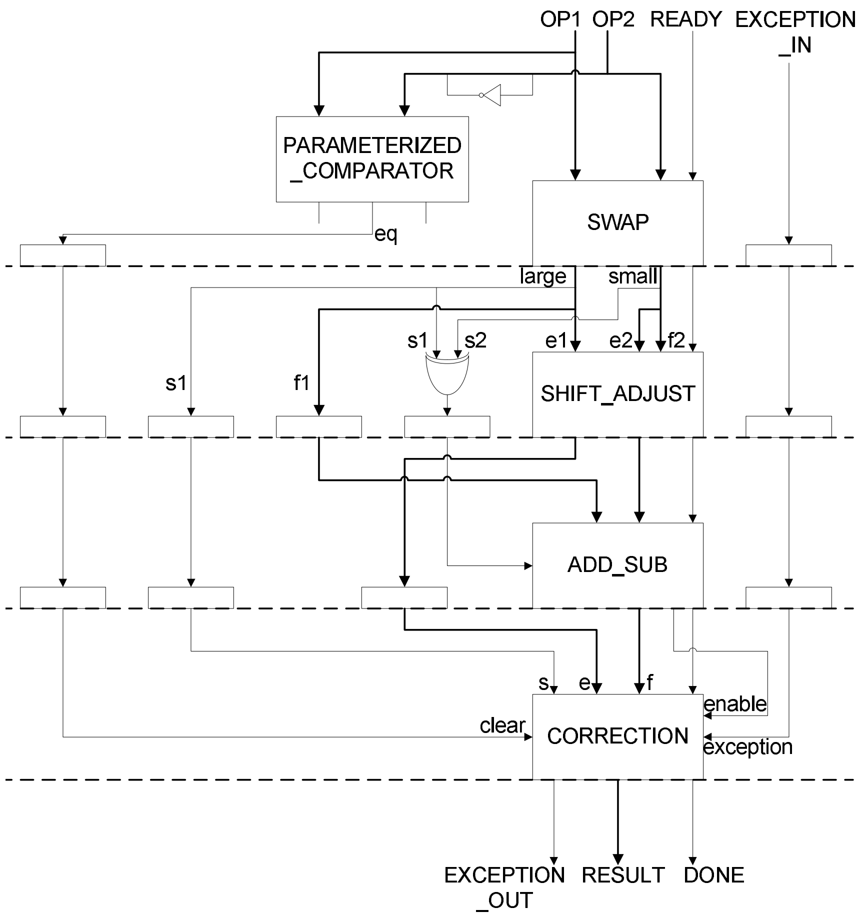

Figure 2.3: Floating-point addition

equal and the mantissa of input A is larger, input A is multiplexed to output `large` and input B to output `small`. Otherwise, the reverse mapping of inputs to outputs occurs.

Submodule `shift_adjust` is responsible for aligning the mantissas of the larger and smaller operands. It achieves this by shifting the smaller mantissa to the right as many bits as is the difference between the exponents. Another function of this module is to introduce the guard bit into the smaller operand's mantissa. Guard bits are introduced in the addition algorithm to provide rounding to nearest for the result.

Hence, the mantissa of the sum is one bit wider than the mantissas of the inputs. Expansion of the mantissa fields happens during aligning of the mantissas, so that the extra information the guard bit carries can be saved when the smaller operand's mantissa is shifted right and some least significant bits may be lost. The guard bit is introduced into the larger operand's mantissa to the right of the least significant bit and always has value '0'.

Once the mantissas are aligned, it is necessary to either add or subtract them, depending on the signs of the two operands. If the signs are the same, the addition operation is constructive and the mantissas are added. If the signs are opposite, however, the addition operation is destructive and the mantissas are subtracted. Submodule `add_sub` will perform this variable operation under the control of the `op` input, which is fed with the XOR of the input sign bits.

Outputs of the overall addition algorithm are controlled by the `correction` module. If an exception is indicated on the input, the exception is propagated to the output and the result output is set to all zeros. Otherwise, if the input values are detected to be of the same magnitude, but opposite sign, an exception is not generated on the output, but the result output is still blanked out, to indicate zero value, as $A + (-A) = 0$. Otherwise, if an overflow in the addition of the mantissas was detected, the result mantissa is shifted to the right by one bit, truncating the least significant bit, and the most significant bit is filled with '1'. Also, the exponent field is incremented by 1, to reflect the shift in the mantissa. These operations correct for the overflow in the addition of the mantissas. Finally, the floating-point value assembled from the sign, exponent and mantissa fields is presented on the output.

Module `fp_add` is parameterized by the width of the exponent and mantissa fields of the floating-point format it operates on.

Subtraction

The subtraction operation is similar to the addition operation, as

$$A - B = A + (-B)$$

Thus, we use a slightly modified addition module to perform subtraction. This is especially helped by the sign-magnitude form of the floating-point format. To invert a floating-point value, all that needs to be done is to invert the sign bit (most significant bit, MSB, of the floating-point signal).

There is only one, minor structural difference between the addition and subtraction modules: the inverter on the MSB of the second operand is not used on the input to the `parameterized_comparator` module (see Figure 2.3), but on the input to the `swap` module. That way, we invert the sign of the second operand to achieve subtraction. Also, the comparator now monitors input values equal in both sign and magnitude, since $A - A = 0$. Because the inverter is only moved from one location to another, module `fp_sub` occupies the exact same area as the `fp_add` module.

Similarly to the `fp_add` module, the `fp_sub` module is also parameterized by the width of the exponent and mantissa fields of the floating-point format it operates on.

Multiplication

Unlike fixed-point arithmetic, in floating-point arithmetic, multiplication is a relatively straight-forward operation compared to addition. This is again due to the sign-magnitude nature of the floating-point format, because

$$((-1)^{s_1} \times m_1 \times 2^{e_1}) \times ((-1)^{s_2} \times m_2 \times 2^{e_2}) = (-1)^{s_1 \oplus s_2} \times (m_1 \times m_2) \times 2^{(e_1+e_2)}$$

From the above, it can be concluded that the three fields of the floating-point format do not interact during multiplication and can thus be processed at the same time, in parallel. The sign of the product is given as the exclusive OR (XOR) of the input value signs. Mantissa of the product is calculated by fixed-point multiplication of the input value mantissas, while the exponents of the input values are added to give the exponent of the product.

The only further complication of the floating-point multiplication algorithm is the fact that the exponent fields are biased. When two biased exponent fields are added, the result contains the bias twice, one of which must be subtracted. If, using IEEE standard 754 notation, E is an unbiased exponent and e is a biased exponent, it stands that:

$$\begin{aligned}
&e_1 + e_2 = \\
&\quad (E_1 + BIAS) + (E_2 + BIAS) = \\
&\quad (E_1 + E_2) + 2 \times BIAS = \\
&\quad E_p + 2 \times BIAS = \\
&\quad e_p + BIAS
\end{aligned}$$

The structure of the floating-point multiplier is given in Figure 2.4. The `fp_mul` module is parameterized by the bitwidths of the exponent and mantissa fields of the floating-point format it processes. The bitwidth of the product is $1 + exp\_bits + (2 \times man\_bits)$. The mantissa field has twice the bitwidth of the input mantissas because it is their fixed-point product.

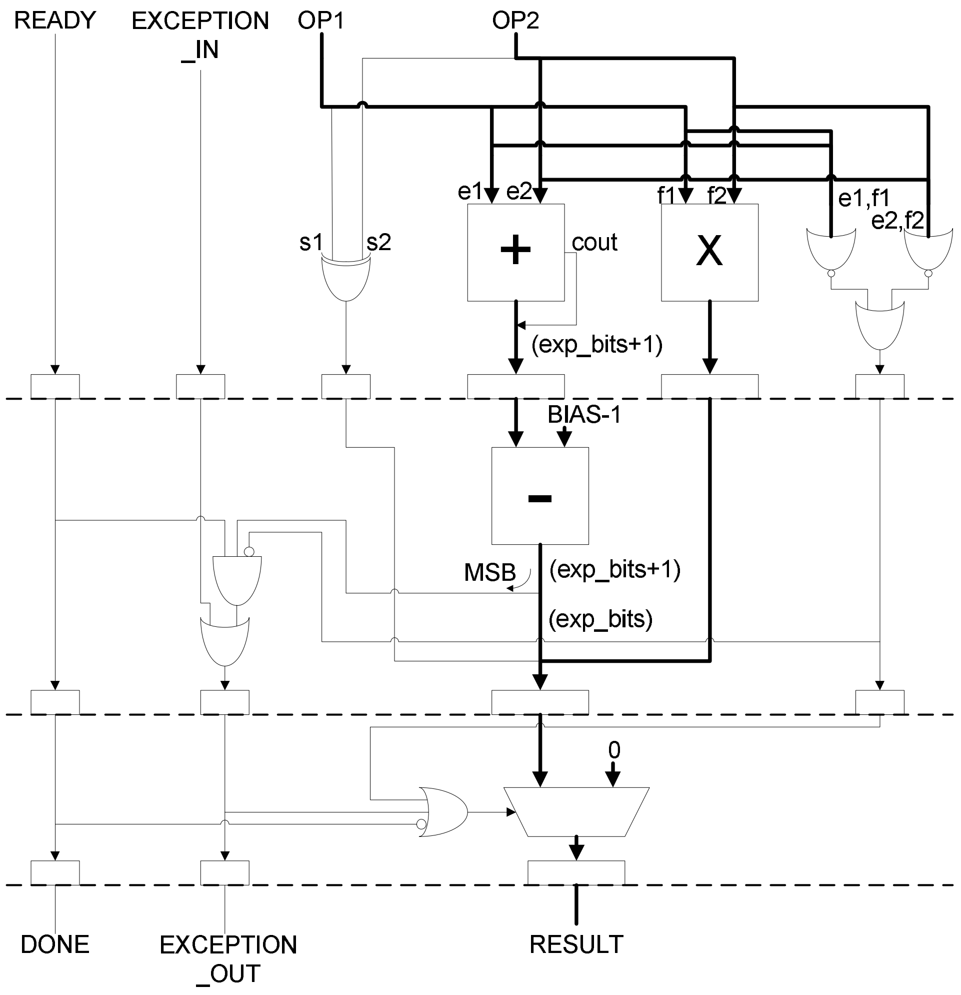

Figure 2.4: Floating-point multiplication

2.4 Format Conversion Hardware Modules

Custom hardware architectures have the ability to perform some sections of the algorithm in fixed-point arithmetic and others in floating-point arithmetic, depending on the optimal representation of each variable in the algorithm. It is the goal of our library to provide all the hardware modules the designer needs to build such hybrid fixed and floating-point architectures. Hence, some of the most important modules are those that convert between fixed and floating-point representations of variables.

Conversion From Fixed-Point To Floating-Point

Module `fix2float` was designed to convert a given value from fixed to floating-point representation. Thus, its input is a fixed-point value and its output is the corresponding floating-point representation. Since fixed-point values can be in the unsigned or signed (two's complement) form, two versions of the `fix2float` module have been developed. The structure of the unsigned version is shown in Figure 2.5, while the structure of the signed version is shown in Figure 2.6. The signed version is more

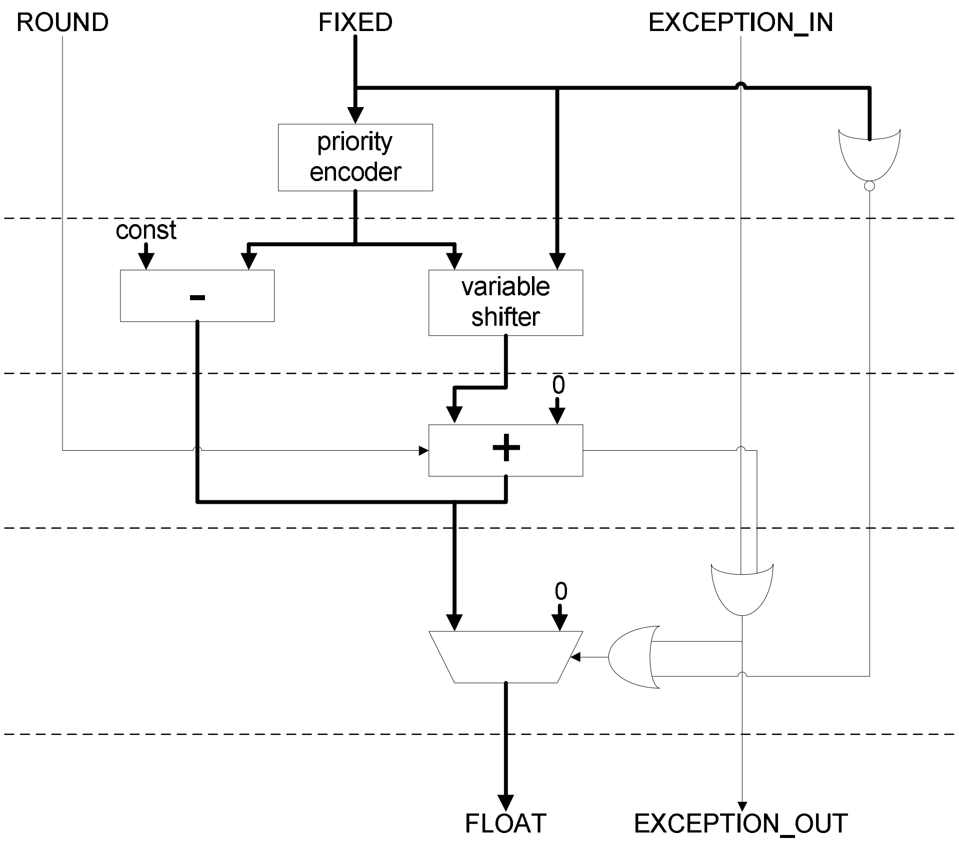

Figure 2.5: Conversion from unsigned fixed-point to floating-point representation

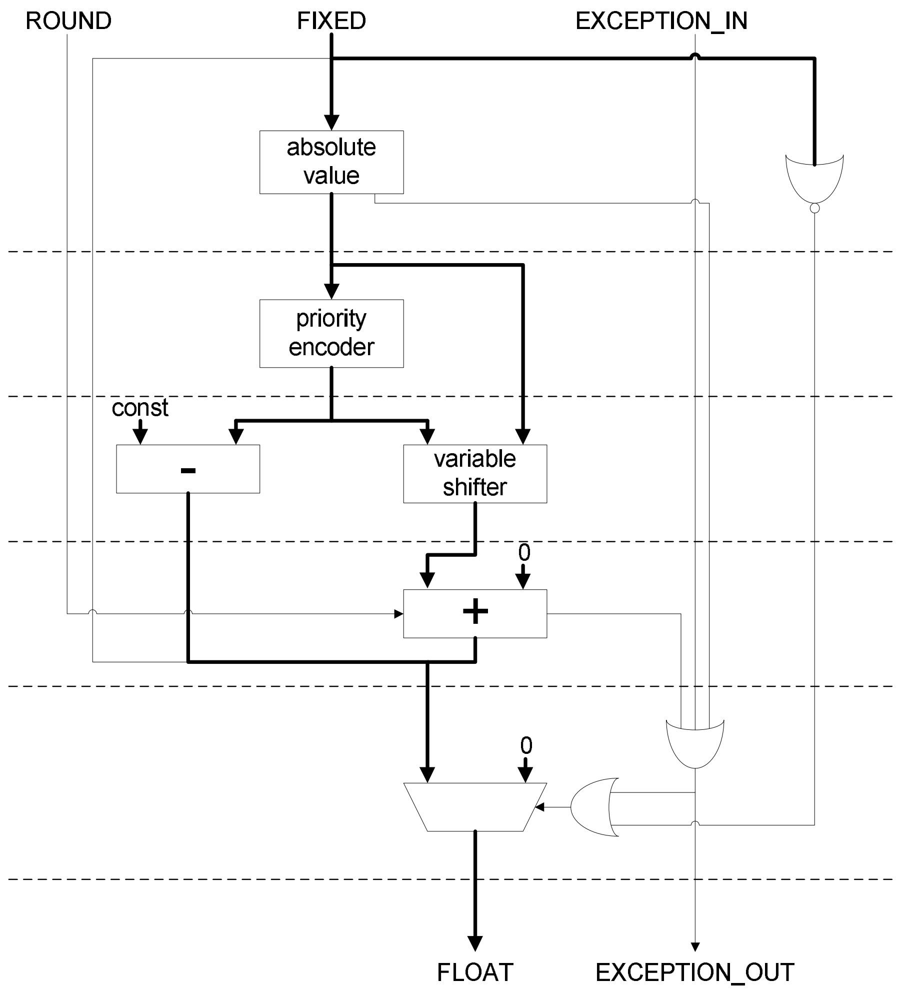

Figure 2.6: Conversion from signed fixed-point to floating-point representation

complex due to handling of the two's complement representations of the input and hence has a longer latency of 5 clock cycles, as opposed to 4 clock cycles for the unsigned version (see Table 2.1).

To determine the sign-magnitude form of the resulting floating-point representation, the absolute value of the input fixed-point number must first be obtained. In the conversion from signed fixed-point numbers, it may be necessary to derive the two's complement of the input signal, while in the case of unsigned fixed-point numbers, no operation is necessary, as only non-negative values can be represented. This added operation results in the difference in latencies of the signed and unsigned module versions.

The mantissa of the final result is produced by shifting left the absolute value of the input until its MSB is '1', while the exponent is derived from format constants and the number of shifts made to the mantissa. For example:

$$\begin{aligned}
&01001011_2 = 75 = \\
&\quad = 01001011_2 \times 2^0 = 10010110_2 \times 2^{-1} = 1.0010110_2 \times 2^{7-1} \\
&\quad = 1.0010110_2 \times 2^6 = 1.171875 \times 64 = 75 \\
&\quad \Rightarrow f = 0010110_2 \\
&\quad \Rightarrow e = 6 + BIAS
\end{aligned}$$

The value of the exponent field depends on the normalizing shift of the mantissa, $shift$, the bitwidth of the fixed-point input, fix_bits, and the bias value, $BIAS$. Its final form is

$$e = E + BIAS = ((fix\_bits - 1) - shift) + BIAS = (fix\_bits + BIAS - 1) - shift$$

The absolute value of the input is fed into a priority encoder, to determine the $shift$ value. This constitutes the first clock cycle of the unsigned architecture and the second cycle of the signed one. Once the value of the normalizing shift is known, the exponent field is calculated by performing the subtraction $(fix\_bits + BIAS - 1) - shift$. The value of the signal `const` in Figures 2.5 and 2.6 is $fix\_bits + BIAS - 1$.

Once the value of the normalizing shift is known, the mantissa is produced by shifting left the absolute value of the input and the exponent is calculated through subtraction. These operations happen in parallel, in the second clock cycle of the unsigned architecture and third of the signed one.

After the shifting operation, the width of the mantissa field is that of the fixed-point input and may need to be reduced to the width specified by the floating-point format that is to appear on the output. This reduction in bitwidth calls for rounding, in a similar fashion as discussed in Section 2.2. Rounding to zero or nearest are both available through input `round` and happen in clock cycle three in the unsigned architecture and four in the signed one.

The final clock cycle of both architectures is dedicated to determining the outputs of the circuit. The floating-point output is either the calculated value or all zeros. The latter option is multiplexed to the output in case of an exception being received at the input or encountered during processing, or a zero fixed-point input, which requires an all-zero floating-point output. Otherwise, the floating-point value calculated by the module is presented on the output.

Both versions of the `fix2float` module are parameterized by three values: the width of the fixed-point input, the width of the exponent field and the width of the mantissa field of the floating-point output.

Conversion From Floating-Point To Fixed-Point

Module `float2fix` implements the inverse function to that of the `fix2float` module: conversion from the floating-point representation of a value to its fixed-point representation.

As before, two versions of the `float2fix` module exist: one for converting to signed and the other to unsigned fixed-point representation of the input floating-point value. The structure of the hardware for conversion to the unsigned fixed-point representation is shown in Figure 2.7, while Figure 2.8 shows the signed version. Due

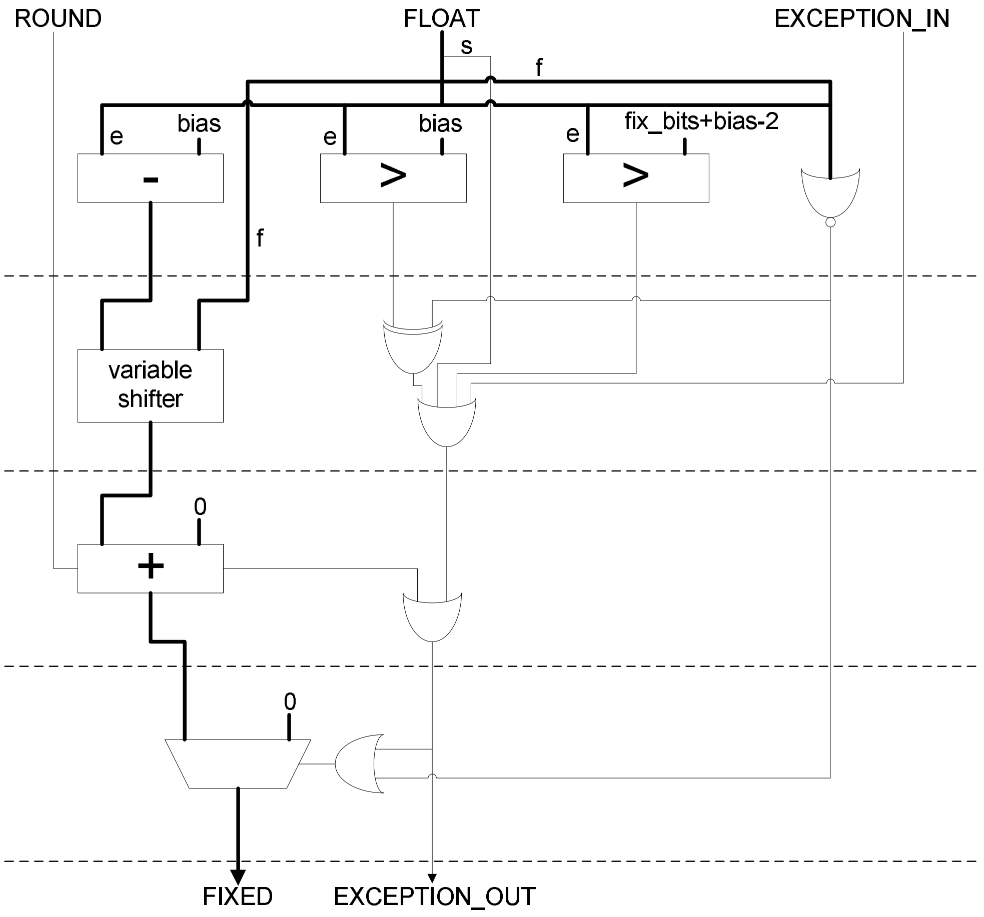

Figure 2.7: Conversion from floating-point to unsigned fixed-point representation

to the added complexity of handling two's complement representations of the output value, the signed version has a latency of 5 clock cycles, while the unsigned version has a latency of 4 clock cycles.

The functioning of the `float2fix` module can easily produce exceptions because, in general, floating-point formats have a wider range than fixed-point formats. For

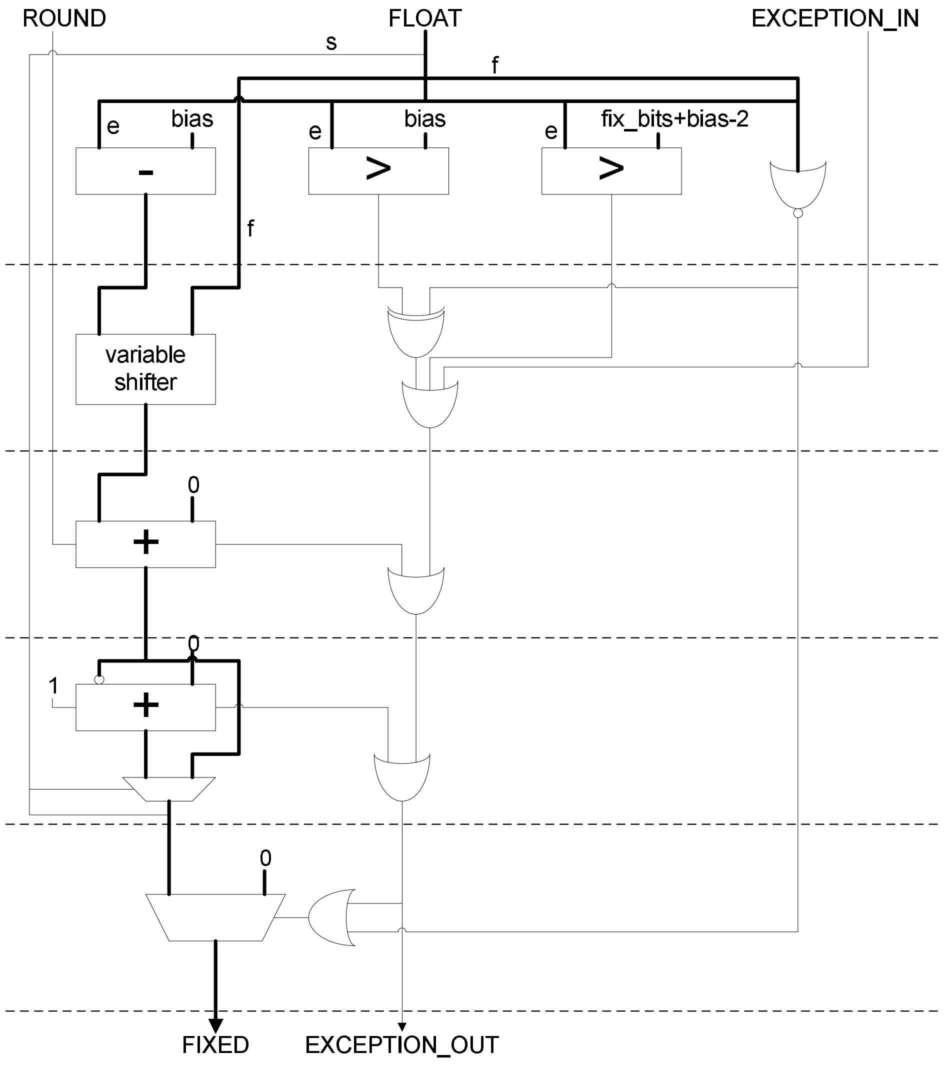

Figure 2.8: Conversion from floating-point to signed fixed-point representation

instance, all floating-point values that have negative exponents (magnitude less than 1) cannot be represented in integer fixed-point formats by values other than 0 or 1. Also, all floating-point values that exceed the largest representable value in the target fixed-point format produce an exception. In the unsigned version, another exception is caused by negative floating-point values appearing on the input, which can by definition not be represented in unsigned fixed-point format. These exceptions are trapped in the first clock cycle of both the signed and the unsigned architecture.

Also in this clock cycle, the shift required to produce the fixed-point output from the mantissa value is calculated using the exponent field. This shift is simply the unbiased value of the exponent. For example:

$$\begin{aligned} & 1.01011_2 \times 2^6 = \\ & \quad = 1.34375 \times 64 = 86 \\ & \quad = 1010110_2 \times 2^0 = 86 \end{aligned}$$

The shift required is calculated by subtracting the bias value from the exponent field of the input. In the second clock cycle of both versions of the `float2fix` module, the absolute value of the fixed-point representation is obtained by shifting left the mantissa field of the input. In parallel with this, the exception signals obtained in the first clock cycle are combined into one exception signal.

Because the fixed-point format on the output may specify a smaller bitwidth than the mantissa field of the input floating-point format, some least significant bits of the absolute value of the fixed-point representation, obtained by shifting the mantissa field, may need to be truncated. This truncation calls for rounding functionality, implemented in the third clock cycle of both the signed and the unsigned architecture.

The prepared absolute value of the fixed-point representation, rounded to the required bitwidth, is ready for output in the unsigned version, while in the signed version, it may need to undergo a two's complement operation before being placed on the output. It is because of this extra step that the signed version of the module has the longer latency of 5 clock cycles. The two's complement of the absolute value is found by inverting all the bits and adding 1. The sign bit of the input floating-point value is used to select the correct form (positive or two's complement) of the fixed-point value, before it is passed to the next stage.

The final stage of both the signed and the unsigned architectures is the output stage, where the computed fixed-point representation is placed on the output, unless the input was zero or an exception was encountered during operation or received at the input, in which case the output is set to all zeros.

Module `float2fix` is parameterized by the bitwidths of the exponent and mantissa fields of the input floating-point signal, as well as the bitwidth of the fixed-point output.

2.5 Implementing an IEEE Single Precision Adder

This section describes assembly of several hardware modules in the library into one module that performs a complete floating-point operation. As an example, we present assembly of an IEEE single precision adder. The IEEE single precision floating-point format consists of 1 sign bit, 8 exponent bits and 23 mantissa bits. To implement an IEEE single precision adder, the designer would use three parameterized modules: `denorm`, `fp_add` and `round_norm`. The first module would be instantiated twice, to introduce the implied integer digit into both input operands. The `fp_add` module accepts the two prepared operands and produces their sum. The mantissa field of the sum signal is 25 bits wide: one extra bit for the integer digit introduced by the `denorm` module and one extra bit for the guard bit during addition. Module `round_norm` accepts the sum signal and reformats it back into IEEE single precision format. Assembly of the overall IEEE single precision adder is shown in Figure 2.9.

The IEEE single precision adder module, assembled as shown above and mapped

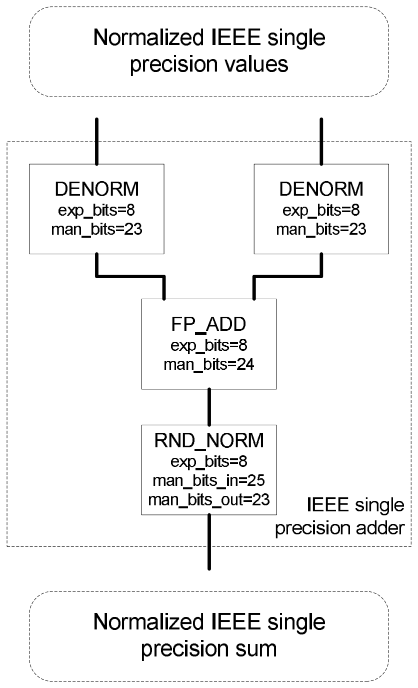

Figure 2.9: Assembly of an IEEE single precision adder

to an XCV1000 processing element of the Annapolis Micro Systems Wildstar engine takes up 305 slices, or just under 2.5% of the chip. Its VHDL description demonstrates instantiation of parameterized library modules and is given in Appendix B.

2.6 Testing

All the hardware modules described in this chapter have been tested both in simulation and in hardware. The purpose of the two testing stages was to ensure the correct operation of the VHDL description of each module. A set of input vectors was developed for each module to test its operation with a range of inputs. Parameterization of each module was also tested to ensure correct operation of the module at various instances in the design space.

The simulator used to test the VHDL descriptions was Mentor Graphics ModelSim (see Section 1.3). Iteration between modification of the VHDL description and analysis in the simulator continued until the correct operation of the module was achieved for all the test vectors.

The second testing stage was done in hardware. The VHDL description, shown to operate correctly in the simulator, was synthesized and loaded on the Wildstar board. The same set of test vectors used in simulation was applied to the hardware implementation. Testing results from hardware were compared to expected result values. If they did not match, the VHDL description was further modified to achieve the correct operation.

An example of a test vector used to test the IEEE single precision adder circuit in Section 2.5 is given below.

Operand 1	**Operand 2**	**Sum**
41BA3C57	4349C776	43610F01

$41BA3C57_{16} =$

$= 0\ 10000011\ 01110100011110001010111_2$

$= +\ 1.45496642 \times 2^4$

$= 23.27946281$

$4349C776_{16} =$

$= 0\ 10000110\ 10010011100011101110110_2$

$= +\ 1.57639956 \times 2^7$

$= 201.77914429$

$$
\begin{aligned}
Sum &= \\
&= 225.05860710 \\
&= +\,1.75827036 \times 2^7 \\
&= 0\;10000110\;11000010000111100000001_2 \\
&= 43610F01_{16}
\end{aligned}
$$

2.7 Results

This section presents results of synthesis experiments conducted on the floating-point operator modules *fp_add*, *fp_sub* and *fp_mul* of Section 2.3. The aims of the experiments are to:

- determine the area of the above modules in several floating-point formats,
- examine the relationship between the area and total bitwidth of the format, and
- estimate the number of modules that can realistically be used on a single FPGA.

The experiments were conducted by synthesizing the modules for specific floating-point formats on the Annapolis Micro Systems Wildstar reconfigurable computing engine (see Section 1.3). Table 2.2 shows results of the synthesis experiments on floating-point operator modules. The quantities for the area of each instance are expressed in slices of the Xilinx XCV1000 FPGA. Results for the `fp_add` module in Table 2.2 also represent the `fp_sub` module, which has the same amount of logic.

Floating-point formats used in the experiments were chosen to represent the range of realistic floating-point formats from 8 to 32 bits in total bitwidth and include the IEEE single precision format (E1 in Table 2.2).

Table 2.2: Operator synthesis results

Format	**Bitwidth**			**Area**		**Per IC**	
	total	exponent	fraction	fp_add	fp_mul	fp_add	fp_mul
A0	8	2	5	39	46	236	200
A1	8	3	4	39	51	236	180
A2	8	4	3	32	36	288	256
B0	12	3	8	84	127	109	72
B1	12	4	7	80	140	115	65
B2	12	5	6	81	108	113	85
C0	16	4	11	121	208	76	44
C1	16	5	10	141	178	65	51
C2	16	6	9	113	150	81	61
D0	24	6	17	221	421	41	21
D1	24	8	15	216	431	42	21
D2	24	10	13	217	275	42	33
E0	32	5	26	328	766	28	12
E1	32	8	23	291	674	31	13
E2	32	11	20	284	536	32	17

The number of operator cores per processing element, shown in the two rightmost columns, is based on a Xilinx XCV1000 FPGA, with a total of 12288 slices, with 85% area utilization. A realistic design cannot utilize all the resources on the FPGA because of routing overhead; a practical maximum is estimated at about 85%. Also included is an overhead allowance of approximately 1200 slices for necessary circuitry other than the operators themselves. This allowance may represent memory read and/or write circuitry, state machines, register tables and similar circuits which are required by most designs. Thus, results shown for the number of modules per IC correspond to realistic designs using the operator cores.

The results in Table 2.2 show growth in area with increasing total bitwidth, for all modules. This growth is represented graphically in Figure 2.10.

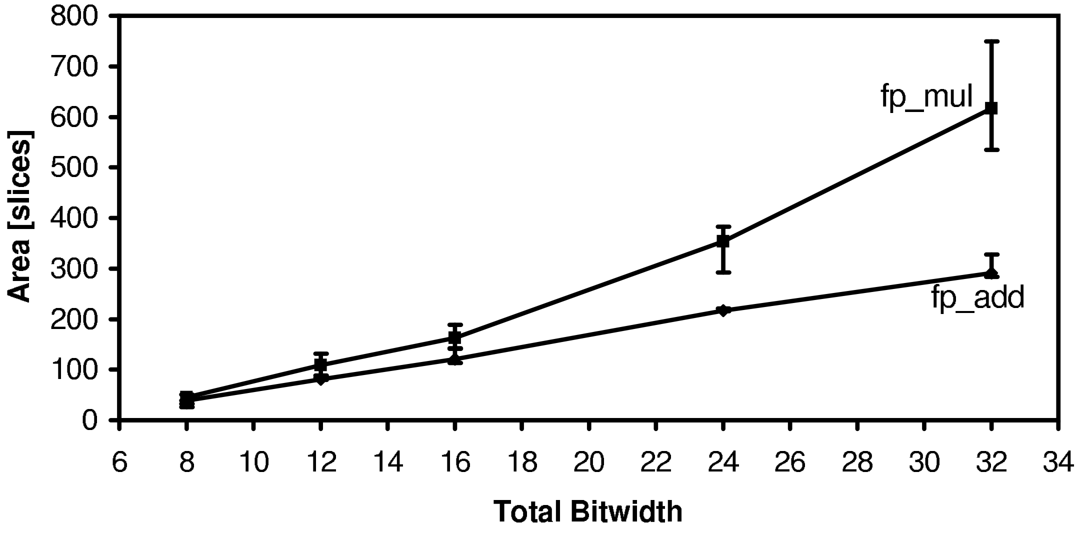

Figure 2.10: Growth of area with increasing bitwidth

The inverse effect of the growth in size with bitwidth is the reduction in the number of cores that can realistically fit onto an XCV1000 processing element. This is shown graphically in Figure 2.11.

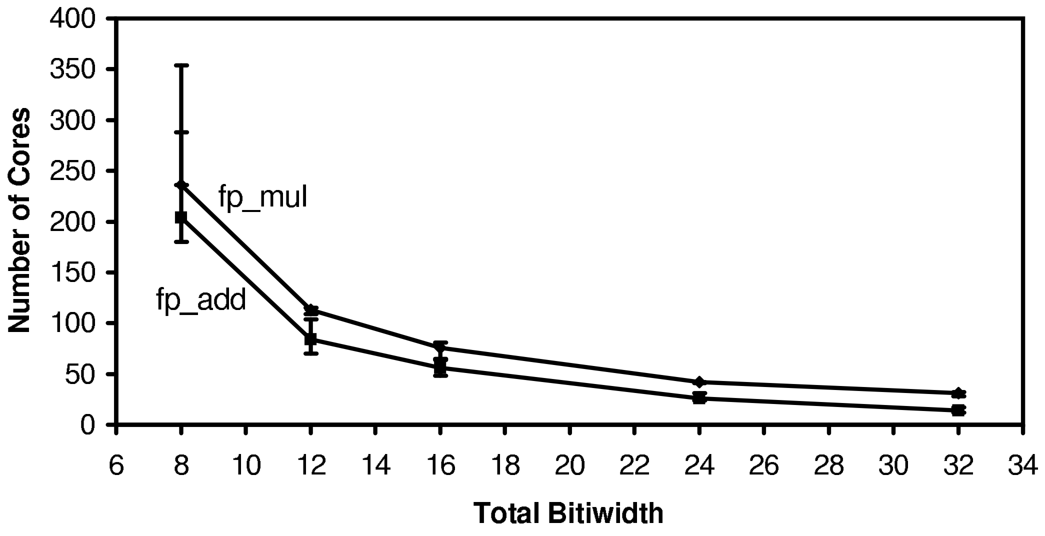

Figure 2.11: Reduction in number of cores per processing element

2.8 Conclusion

In this chapter, all the hardware modules that constitute the parameterized library of hardware components have been presented. Some of the building block modules that are used to construct the top level modules of the library were presented. Construction of complete floating-point operations from library modules, testing and results of synthesis experiments on operator modules were also discussed. To demonstrate functionality of all these modules and give an example of assembling the library modules to make a fully custom floating-point datapath, the next chapter introduces an example application: the K-means clustering algorithm for multispectral and hyperspectral images.

Chapter 3

An Application: K-means Clustering

The hardware modules described in Chapter 2 lend themselves to the creation of finely customized hardware implementations of algorithms. They give the designer full freedom to implement various sections of the algorithm in the most suitable arithmetic representation, be it fixed or floating-point. Also, bitwidths of all the signals in the circuit, whether in fixed or floating point representation, can be optimized to the precision required by the values the signal carries.

When using floating-point arithmetic, the designer using the library has full control to trade off between range and precision. Because all the modules in the library are fully parameterized, the boundary between the exponent and fraction fields for the same total bitwidth is flexible. With a wider exponent field, the designer provides larger range to the signal, while sacrificing precision. Similarly, to increase the precision of a signal at the cost of reduced range, the designer chooses a narrower exponent and wider fraction field.

Another benefit of using the library is that the input data format does not define the way the data is processed; its representation can be changed if required. Similarly, the results of processing can be stored in any format that is needed for further processing or presentation. Hence, the library of parameterized modules presented here provides the finest-grain control possible over signal format and bitwidth, as well as an opportunity to implement hybrid fixed and floating-point designs.

To demonstrate the above concepts and illustrate the use of the library, an application was developed, designed and implemented. The application is a hybrid implementation of the K-means clustering algorithm for satellite image processing and is described in this chapter.

The K-means algorithm is highly suitable as an example application in this thesis because it demonstrates the ability of the library modules to form hybrid fixed and floating-point circuits. Also, a purely fixed-point implementation of the K-means algorithm exists and can be used to assess the quality of the hybrid implementation. Construction of the hybrid implementation makes use of most modules in the library (with the exception of the multiplication operator) and shows the ability of the modules to form fully pipelined circuits. Also, the construction of the hybrid implementation shows that correct assembly of library modules achieves appropriate arithmetic operation with real data. The hybrid implementation of the K-means algorithm exploits the ability of the library modules to construct fully custom floating-point formats and demonstrates processing of data in a format different from the one it is stored in.

3.1 K-means Clustering Algorithm and Structure

The K-means clustering algorithm is commonly used for segmentation of multi-dimensional data. This chapter describes two implementations of this algorithm, applied to multispectral/hyperspectral satellite images, both using reconfigurable hardware. The first implementation by M. Estlick and M. Leeser [3] is based on a purely fixed-point datapath. The second implementation was developed as part of the work presented in this thesis and uses a hybrid fixed and floating-point datapath. Both implementations are designed to operate on 10-channel multispectral data, with 12 bits per channel; this data is segmented into 8 clusters.

General properties that pertain equally to both implementations are described in Section 3.1, details particular to the purely fixed-point implementation are contained in Section 3.2 and details particular to the hybrid implementation are given in Section 3.3.

Algorithm

K-means is an unsupervised clustering algorithm that operates by assigning multi-dimensional vectors to one of K clusters. The aim of the algorithm is to minimize variance of all vectors within each cluster. The algorithm is iterative: after each vector is assigned to one of the clusters, the positions of cluster centers are re-calculated and the vectors are assigned again. Pseudo-code describing the K-means algorithm is given below.

```
kmeans(image)
{
  //initialize cluster centers
  centers=initialize_cluster_centers();
  //main loop; until terminating condition is met
  while(done!=TRUE)
  {
    //go through all pixels
    for(pixel=0;pixel<N;pixel++)
    {
      //initialize min_distance to maximum value it can hold
      min_distance=infinity;
      //go through all the clusters
      for(cluster=0;cluster<K;cluster++)
      {
        //calculate the distance between pixel and cluster center
        distance=vector_distance(image[pixel],centers[cluster]);
        //is it shortest so far?
        if(distance<min_distance)
        {
          //assign pixel to cluster
          cluster_image[pixel]=cluster;
          //keep minimum distance
          min_distance=distance;
        }
      }
    }
    //go through all clusters
    for(cluster=0;cluster<K;cluster++)
    {
      //initialize accumulators
      accumulators[cluster]=0;
      //initialize counters
      counters[cluster=0;
    }
    //go through all the pixels
    for(pixel=0;pixel<N;pixel++)
    {
      //accumulate pixel value
      accumulator(cluster_image[pixel])+=image[pixel];
      //increment counter
      counter(cluster_image[pixel])++;
    }
    //go through all the clusters
    for(cluster=0;cluster<K;cluster++)
      //re-calculate cluster centers
      centers[cluster]=accumulator[cluster]/counter[cluster];
    //evaluate terminating condition; are we done?
    done=terminating_condition();
  }
  //return clustered image and final positions of cluster centers
  return cluster_image,centers;
}
```

The given pseudo-code processes a total of `N` pixels, assigning them to one of `K` clusters. Variable `centers` contains `K` multidimensional vectors which represent the center points for each cluster. The while loop iterates until a terminating condition is met. The distance to each cluster center is calculated for each pixel in the input image. The minimum distance is kept and the pixel is assigned to the cluster that corresponds to the minimum distance. Once all the pixels have been assigned to one of the clusters, accumulators and counters are cleared. Each pixel in the image is then accumulated into the accumulator that corresponds to the cluster it was assigned to. The corresponding counter is incremented. Once all accumulations have been performed, cluster centers are re-calculated to be the average value of all the pixels assigned to the particular cluster. The terminating condition is evaluated and the process is repeated. The algorithm returns the clustered image, as well as the final values of all the cluster centers.

In both the purely fixed-point and the hybrid implementation the algorithm is partitioned between the host and the reconfigurable computing engine. In both implementations, initialization and re-calculation of cluster centers happens on the host, while cluster assignments of the pixels and accumulations happen in reconfigurable hardware. The interaction between the host and reconfigurable hardware is illustrated in Figure 3.1.

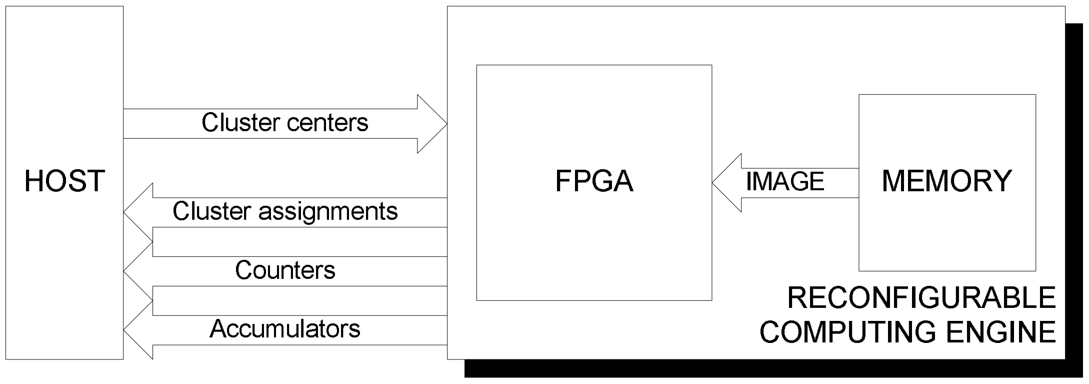

Figure 3.1: Interaction between the host and the reconfigurable hardware

Applied to multispectral and hyperspectral image processing, the reconfigurable hardware partition of the K-means clustering algorithm consists of assigning each pixel in the input image to one of the K clusters, as well as accumulating the values of all pixels assigned to each cluster. Hence, inputs to the circuit are all pixels in the image and the positions of all K clusters, while its outputs are a cluster assignment for each pixel, accumulated values of all pixels assigned to each cluster and counters of the number of pixels accumulated for each cluster.

Pixels are assigned to the nearest cluster, which is achieved by comparing Manhattan, or 1-norm, distances of the pixel to each cluster center. The pixel is thus assigned to the cluster corresponding to the shortest distance. Hence, the first stage of the datapath concentrates on determining the Manhattan distance of the pixel to each cluster center, while the second compares those K distances to find the minimal one, yielding the cluster assignment.

There is an accumulator associated with each cluster, or a set of K accumulators. Also, there is a counter associated with each cluster, or a set of K counters. Once the cluster assignment for a given pixel is made, it is used to reference the corresponding accumulator and counter. The value of the pixel is added to the accumulator, while the counter is incremented to reflect the accumulation.

The above set of operations is performed on every pixel in the image serially. The pixels are streamed from on-board memory, until every pixel is processed.

Design Structure

The overall K-means clustering circuit is composed of two functional units and two shift-register units. The first functional unit is a datapath, taking at its input the pixel value and the locations of all cluster centers and outputting the cluster assignment of the pixel. The second functional unit accumulates pixel values according to the cluster assignment given by the datapath circuit. Both shift-register units are used to delay signals from memory to the accumulator circuit, to synchronize their arrival with the cluster assignment which is delayed due to the latency of the datapath unit. The first delay pipe is used to pass pixel values from memory to the accumulator circuit, in parallel with their processing in the datapath unit. The second delay pipe passes the signal from memory indicating that pixel data is valid, which is used to enable the accumulator circuit. An illustration of the structure of the K-means circuit is shown in Figure 3.2.

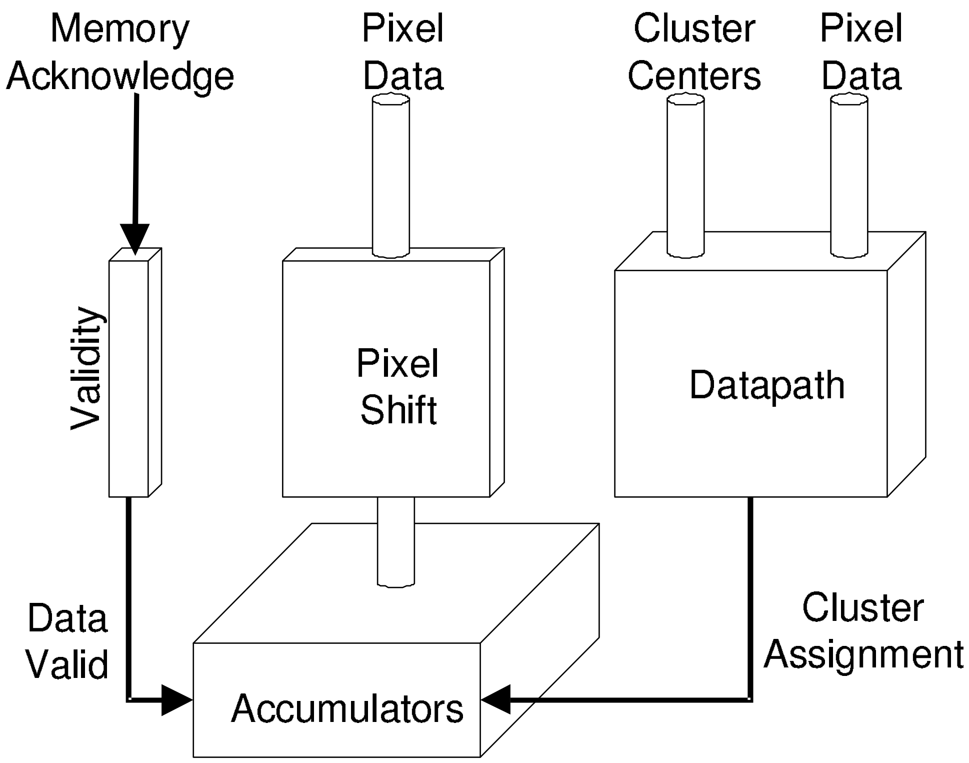

Figure 3.2: Structure of the K-means circuit

The validity shift pipeline is a shift register in both implementations. The pixel shift pipeline is a set of shift registers, transferring pixel values from memory to the accumulator circuit, in both implementations. The accumulator unit operates in fixed-point arithmetic in both implementations, because of its area requirements. The purely fixed-point implementation has a datapath unit that operates only in fixed-point arithmetic, while the hybrid implementation has a datapath unit that operates both in fixed and floating-point arithmetic.

Datapath Unit

The datapath unit determines the cluster assignment for each pixel. It arrives at this decision by computing the distance of the pixel to each of the cluster centers, in Manhattan or 1-norm in both of our implementations, and finding the minimal one of those distances. The pixel is assigned to the cluster corresponding to the minimal distance. Hence, the first stage of the datapath circuit calculates 8 distances, while the second stage of the circuit compares those 8 distances to extract the minimal one.

Distance calculations of the datapath circuit are done in 10-dimensional space, because the algorithm operates on 10-channel data. To perform this distance calculation, each dimension (or channel) of the pixel value is subtracted from the corresponding dimension (or channel) in the cluster center, followed by an absolute value operation on all ten differences, leading into an addition tree that sums all the absolute values, resulting in the value of the distance. Thus, calculating distance involves three operations: subtraction, absolute value and addition. Once all 8 distances are known, they are compared to find the minimal one. The four operations that make up the datapath unit are illustrated in Figure 3.3.

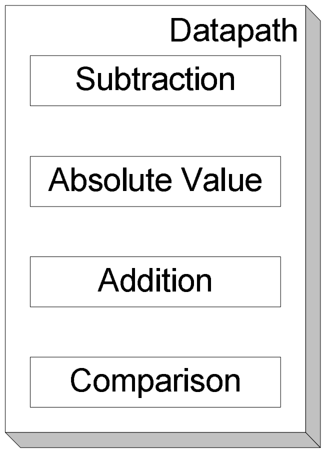

Figure 3.3: Structure of the datapath unit

3.2 Fixed-point Implementation

The implementation of the K-means clustering algorithm by M. Estlick and M. Leeser [3] is the first of the two implementations considered in this chapter. Its inclusion is based on the fact that it represents a purely fixed-point implementation that does not utilize the library of parameterized hardware components of Chapter 2, but is a correct implementation of the algorithm and can be used to compare against. Thus, it is used here as a reference implementation and a basis for the building of the hybrid implementation described in Section 3.3.

In this implementation, all four units that make up the K-means circuit operate in fixed-point format. The datapath circuit is built from fixed-point subtraction, absolute value, addition and comparison operations. All the inputs to the circuit are assumed in fixed-point format and the outputs are provided in fixed-point format. This suits the application because both pixel and cluster center data is available in 12-bit unsigned fixed-point format and the result is expected in 3-bit unsigned fixed-point format (to represent the cluster assignment to one of 8 clusters).

Because of the relatively small size of the fixed-point datapath circuit, all 8 distance calculations are performed in parallel and a comparison tree is used to derive the minimal distance. This architecture is thus capable of processing a new pixel every clock cycle. Processing results (segmented images) of this design are available and were used as a basis for comparison with the results of the hybrid design.

3.3 Hybrid Implementation

The hybrid fixed and floating-point implementation of the K-means algorithm was created to demonstrate the use of the hardware modules presented in Chapter 2. In particular, it demonstrates the creation of hybrid circuits and the required sectioning of the implementation.

Three out of the four blocks that make up the K-means implementation circuit (see Figure 3.2) still operate in fixed-point format - pixel delay pipe, validity delay pipe and the accumulator circuit. In fact, there is no change to those three circuits from the purely fixed-point implementation. The datapath circuit on the other hand is now divided into a floating-point section and a fixed-point section. Figure 3.4 represents the sectioning of the structure of the datapath circuit.

An inherent advantage of the floating-point format is the relative simplicity of the absolute value operation. Due to the sign-magnitude nature of floating-point representation, the absolute value is found by simply forcing the sign bit of the value to zero (indicating a positive number). Thus, in the datapath circuit, only the operations of subtraction and addition are implemented with hardware modules from the parameterized library, while the absolute value operation is done by signal manipulation.

The floating-point format used in the floating-point section of the datapath was chosen to have range equal to or greater than that of the unsigned fixed-point format.

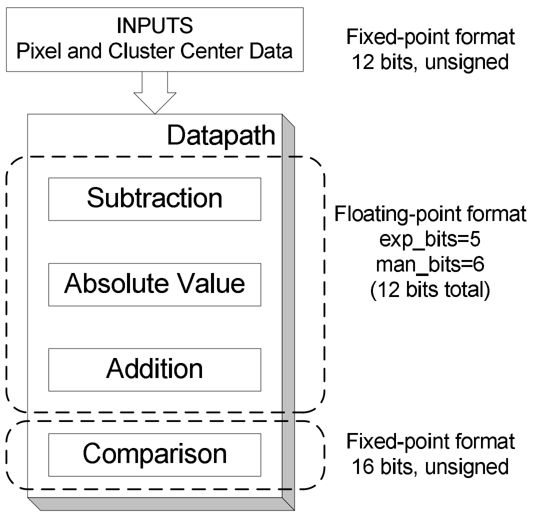

Figure 3.4: Structure of the hybrid datapath unit

Hence, the floating-point format needs to be able to represent all values between 0 and $2^{12} - 1$, or $[0, 4095]$. To maintain total bitwidth of 12 bits, floating-point format with 5 exponent bits and 6 mantissa bits (1-5-6) was chosen, with range calculated as follows. The largest positive number that can be represented is:

$$\begin{aligned}&0\,11111\,111111_2 = \\ &\quad = (1 + \frac{63}{64}) \times 2^{(31-15)} = \\ &\quad = 1.984375 \times 2^{16} = 130048\end{aligned}$$

Therefore, the range is [-130048,130048].

Converters from fixed to floating-point representation are placed at the input to the subtraction stage of the datapath circuit, to convert the 12-bit unsigned pixel and cluster center data into the 1-5-6 floating-point format for processing. Similarly, floating to fixed-point converters are placed between addition and comparison stages of the datapath circuit to convert the distance measurement from the 1-5-6 floating-point format in which it is computed, to the 16-bit unsigned fixed-point format expected by the comparison operation. The distance signal in the datapath circuit reaches its maximum value when pixel and cluster centers are maximally apart in every channel. In other words, the maximum value of the distance signal for 10 channels is

$$(FFF_{16} - 000_{16}) \times 10 = 9FF6_{16} = 1001\,1111\,1111\,0110_2$$

Thus, 16 bits are required to represent the distance signal. Structural composition of the floating-point section of the datapath circuit is shown in Figure 3.5.

The distance calculation in the datapath circuit takes up more area resources when implemented in floating-point format than it does in the purely fixed-point implementation. Because of this fact, it is not possible to fit multiple floating-point distance calculations into one design that will be implemented on an XCV1000 processing element. Instead, only one distance is calculated and the comparison of distances is performed serially. Because the distance to each of the eight clusters is to be computed for every pixel, a new pixel value is read every eight clock cycles. This serialization, made necessary by the increased area of the distance calculation, affects the structure of the overall K-means implementation in the comparison stage of the datapath circuit and in the rate of memory reads. The throughput of the purely fixed-point datapath circuit is 1 clock cycle, while the throughput of the hybrid datapath circuit is 8 clock cycles.

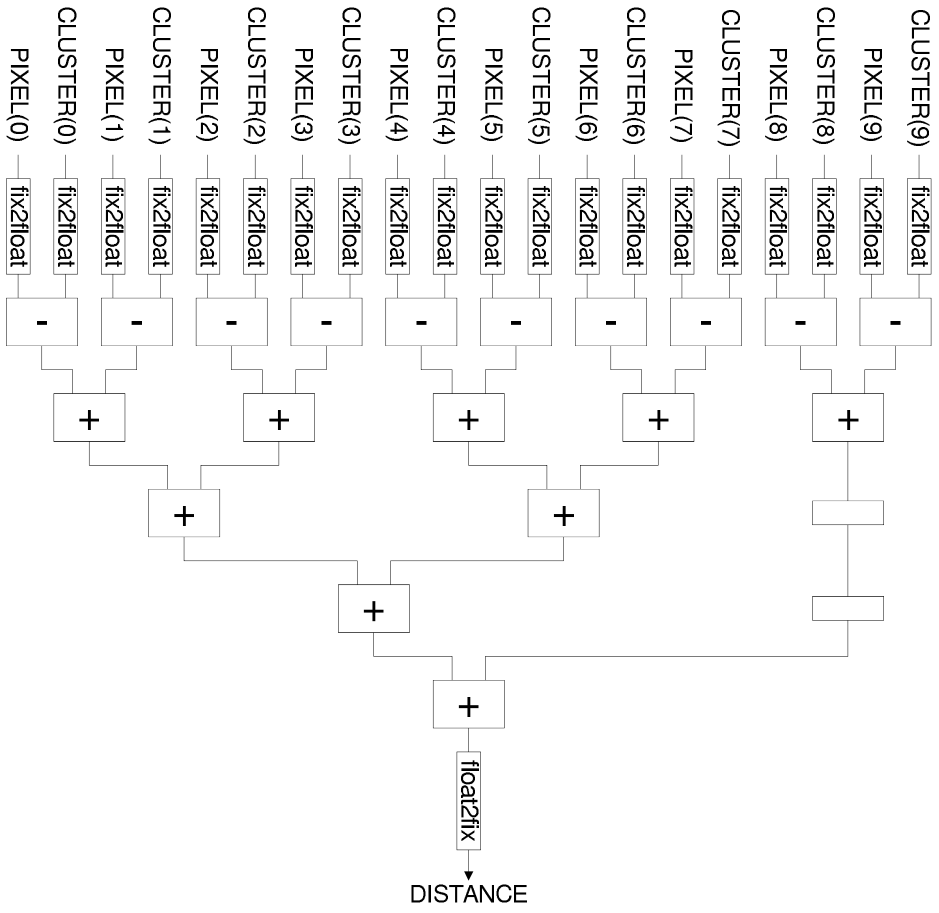

Figure 3.5: Floating-point distance calculation in the datapath circuit

3.4 Testing

The hybrid implementation of the K-means clustering algorithm was tested in both simulation and hardware to ensure it operates correctly. Testing methods used were similar to those described in Section 2.6. The test vectors were a set of cluster centers and a set of pixel values. The VHDL description of the hybrid implementation was modified and simulated until it performed correctly for all test vectors. Following this, the hybrid implementation was synthesized and the same test vectors were applied to confirm that the hardware circuit also operates correctly. An example of a test vector applied to the hybrid implementation is given below.

Cluster	**Channel**									
	0	1	2	3	4	5	6	7	8	9
0	000	000	000	000	000	000	000	000	000	000
1	111	111	111	111	111	111	111	111	111	111
2	222	222	222	222	222	222	222	222	222	222
3	333	333	333	333	333	333	333	333	333	333
4	444	444	444	444	444	444	444	444	444	444
5	555	555	555	555	555	555	555	555	555	555
6	666	666	666	666	666	666	666	666	666	666
7	777	777	777	777	777	777	777	777	777	777

Pixel	**Channel**									
	0	1	2	3	4	5	6	7	8	9
0	000	000	000	000	000	000	000	000	000	000
1	555	555	555	555	555	555	555	555	555	555
2	220	221	222	223	224	225	226	227	228	229

Pixel	**Cluster Distance**								**Assignment**
	0	1	2	3	4	5	6	7	
0	**0**	AAA	1554	1FFE	2AA8	3552	3FFC	4AA6	0
1	3552	2AA8	1FFE	1554	AAA	**0**	AAA	1554	5
2	156D	AC3	**1F**	A91	153B	1FE5	2A8F	3539	2

Once correct operation of the circuit was achieved, the synthesis and operation results presented in the following section were obtained.

Table 3.1: K-means implementations synthesis results

Property	**Fixed-point**	**Hybrid**
Area (in slices)	9420	10883
Percent area used	76%	88%
Minimum period	15.679ns	19.829ns
Maximum frequency	63.78MHz	50.43MHz
Throughput	1 cycle	8 cycles

3.5 Results

This section describes synthesis results of both implementations of the K-means clustering algorithm. Also, processing results, which are clustered images, for both implementations are presented.

Synthesis Results

Both the purely fixed-point and the hybrid implementations of the K-means clustering algorithm are synthesized for the Wildstar reconfigurable computing engine, using the same software tools (see Section 1.3). A comparative summary of synthesis results for the two implementations is shown in Table 3.1. The hybrid implementation is both larger and slower than the purely fixed-point one because of the increased size of the distance calculation section of the datapath circuit.

The hybrid implementation has a potential advantage over the purely fixed-point implementation if the distance metric in both circuits is to be changed from Manhattan to Euclidean. The latter distance metric is the standard metric used in the K-means clustering algorithm, including its software implementation. It produces better clustering results, with 20-40% less total variance in classification than Manhattan distance metric [3]. The purely fixed-point implementation uses the Manhattan distance metric at the cost of inferior performance because of the prohibitive area cost of implementing Euclidean distance calculations.

Converting the purely fixed-point implementation to use the Euclidean distance metric would require 80 additional 12-bit fixed-point multipliers (52 slices each) and approximately double the cost of the addition stage. The total cost of this conversion would be about 4500 slices. The cost of converting the hybrid implementation would be 10 additional 12-bit (1-5-6) floating-point multipliers (108 slices each), putting the total cost at about 1100 slices. Therefore, upgrading the hybrid implementation to use the Euclidean distance metric would require significantly less area than upgrading the purely fixed-point implementation. On the limited hardware resources used in this work, neither of the applications could be upgraded to use the Euclidean distance metric.

Processing Results

The output of both K-means clustering algorithm implementations is a clustered image representing the multispectral or hyperspectral input image. To compare processing results of the purely fixed-point and the hybrid implementations, both designs were given the same input image with 10 channels and 12 bits of data per channel, and clustered images with 8 clusters were collected on their outputs. Figure 3.6 shows the two clustered images, with the output of the hybrid implementation on the top and the output of the purely fixed-point implementation on the bottom.

The clustered images are pseudo-colored, meaning that each cluster in each image is assigned a random color from a set of K (in this case 8) colors. In other words, the actual color used to represent the cluster carries no information. Also, the same circuit produces a different clustering for every run of the algorithm. This is due to the sensitivity of the algorithm to the initialization, which is different in every run. Therefore, the two clustered images in Figure 3.6 are not identical. As can be expected from random color assignment and variation in clustering, they differ in coloring and cluster assignment of a number of pixels. However, in essence, both images demonstrate good quality K-means clustering, because similar pixels are assigned to the same cluster.

3.6 Conclusions

The hybrid implementation of the K-means algorithm showed that the library modules of Chapter 2 can be used to form fully pipelined circuits that perform a chain of arithmetic operations correctly in a fully custom floating-point format. This was further used to construct a hybrid fixed and floating-point circuit which illustrated the use of most of the modules in the library. Comparison of the new, hybrid implementation to the existing, purely fixed-point one showed that good quality K-means clustering is achieved.

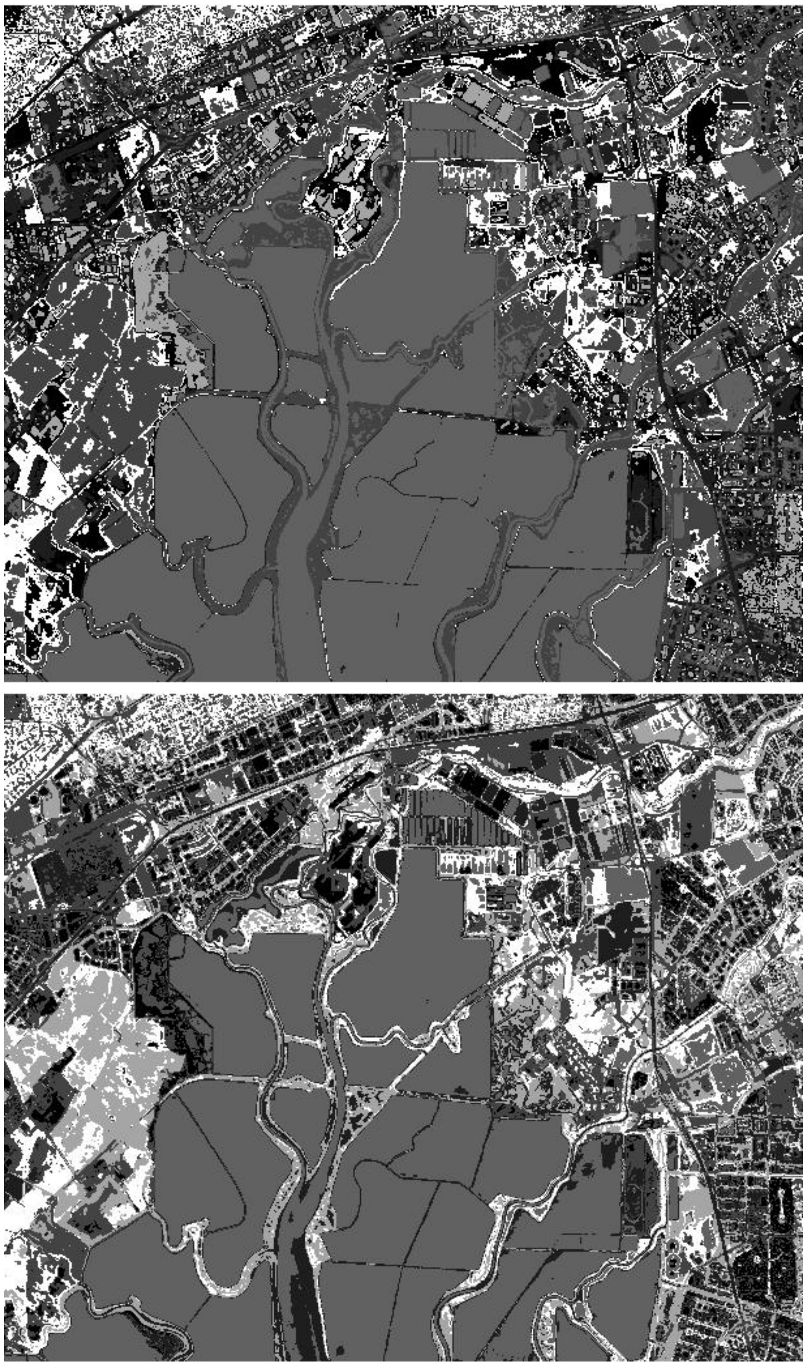

Figure 3.6: Clustered image: hybrid (top) and purely fixed point (bottom) - note that both images are pseudo-colored

Chapter 4

Conclusions

The library of parameterized hardware modules for floating-point arithmetic has been created and includes modules for format control, arithmetic operators and conversion to and from any fixed-point format. All the modules are parameterized to operate on any floating-point format, with rounding to zero or nearest. Limited exception handling is implemented in all the modules, with the ability to propagate an error through a pipeline of modules. Ready and done synchronization signals are provided in all modules, to aid in the creation of pipelines.

The library can be used to implement finely tuned datapaths, in both fixed and floating-point arithmetic, to the exact bitwidths, ranges and precisions required by the signals in the algorithm. Also, library modules for format conversion enable creation of hybrid fixed and floating-point designs. These benefits of the library, as well as ease of use of the modules in creation of custom pipelines, have been demonstrated through the hybrid implementation of the K-means clustering algorithm.

The hybrid version is different from the purely fixed-point one in the format of the distance calculation, which is performed in floating-point. The hybrid version is also more serial, due to its increased area. The results of the algorithm are unchanged with these alterations, indicating suitability of the modules to perform the given functionality in the algorithm.

Synthesis results indicate that a realistic design on a Xilinx XCV1000 FPGA may include up to 31 addition or 13 multiplication operators, complete with denormalizing, rounding and normalizing functionality each, for the IEEE single precision format. Similarly, a useful custom floating-point format, with 5 exponent and 6 mantissa bits for example, may provide the designer with up to 113 addition or 85 multiplication modules, all also complete with full format handling functionalities, on the same FPGA.

Synthesis experiments on the K-means clustering algorithm implementations indicate that the hybrid implementation is larger and slower than the purely fixed-point implementation. Although the hybrid design is thus inferior, processing results indicate that it implements the algorithm equivalently to the fixed-point version. Hence, it serves its purpose of showing that the library can be used to correctly implement hybrid designs of complex algorithms.

4.1 Future Work

With the completion of the library and demonstration of its use come further challenges. Clearly, the library is a resource that can be used to design highly optimized circuits in custom floating-point formats or even hybrid implementations. Hence, future work on this research may include implementations of algorithms that would particularly benefit from the use of the library. These may include algorithms that are highly parallel and have signal values that have a high enough range to require floating-point representation, yet are tolerant enough to accommodate its lower precision. Also, the range and precision required will ideally necessitate bitwidths significantly lower than those of the IEEE formats, so that higher parallelism may be achieved.

The most interesting possibility for future research lies in automation of the design process using the library. Currently, algorithms that would benefit from implementation by the modules in the library need to be manually analyzed and optimized for the best fixed or floating-point format for each signal in the design. Automation of this process would allow the designer to go from a formal description of the algorithm directly to an optimized implementation using fully custom formats.

Research on bitwidth analysis [20, 19] indicates possibilities for area and power savings in bitwidth reduction and use of custom formats. However, these approaches do not have the required implementation support to enable the complete transition from algorithm to hardware. The library of hardware modules presented in this thesis can bridge this gap and provide a complete solution.

Bibliography

[1] IEEE Standards Board and ANSI. IEEE Standard for Binary Floating-Point Arithmetic, 1985. IEEE Std 754-1985.

[2] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs. In *International Symposium on Field-Programmable Gate Arrays*, pages 50–55. ACM, ACM Press, February 2002.

[3] M. Estlick, M. Leeser, J. Theiler, and J. Szymanski. Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *International Symposium on Field-Programmable Gate Arrays*, pages 103–110. ACM, February 2001.

[4] B. Fagin and C. Renard. Field Programmable Gate Arrays and Floating Point Arithmetic. *IEEE Transactions on VLSI Systems*, 2(3), September 1994.

[5] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1), March 1991.

[6] J. P. Hayes. *Computer Architecture and Organization.* McGraw Hill, Second edition, 1988.

[7] J. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kauffmann Publishers, Second edition, 1996.

[8] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[9] Annapolis Micro Systems Inc. Floating-Point Math Library. Technical Data Sheet Doc # 12763-0000 Rev 1.7.

[10] I. Koren. *Computer Arithmetic Algorithms*. Prentice Hall International, 1993.

[11] L. Louca, T. A. Cook, and W. H. Johnson. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116, April 1996.

[12] Z. Luo and M. Martonosi. Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware With Delayed Addition Techniques. In *IEEE Transactions on Computers*, volume 49, pages 208–218, March 2000.

[13] A. R. Omondi. *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*. Prentice Hall International, 1994.

[14] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10)*, pages 132–144, Grenoble, France, 1991. IEEE Computer Society Press, Los Alamitos, CA.

[15] I. Sahin and C. S. Gloster. Floating-Point Modules Targeted for Use with RC Compilation Tools. http://www4.ncsu.edu:8030/˜isahin/papers/DACPaper.pdf, (last visited January 2002), 2001.

[16] I. Sahin, C. S. Gloster, and C. Doss. Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems. In *2000 MAPLD International Conference*, 2000.

[17] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995.

[18] I. Stamoulis, M. White, and P. F. Lister. Pipelined Floating-Point Arithmetic Optimized for FPGA Architectures. In *9th International Workshop on Field Programmable Logic and Applications*, volume 1673 of *LNCS*, pages 365–370, August-September 1999.

[19] D. Stefanovic and M. Martonosi. On availability of bit-narrow operations in general-purpose applications. In *The 10th International Conference on Field Programmable Logic and Applications (FPL 2000)*, Villach, Austria, August 2000.

[20] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver, Canada, June 2000.

[21] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on VLSI Systems*, 8(3):273–286, June 2000.

[22] R. Yates. Fixed-point arithmetic: An introduction. Digital Sound Labs, March

2001. http://personal.bellsouth.net/lig/y/a/yatesc/fp.pdf (last visited January 2002).

Appendix A

VHDL Entities

Module:

`parameterized_adder`

Entity:

```
entity parameterized_adder is
  generic
  (
    bits        : integer           :=  0
  );
  port
  (
    --inputs
    A     : in    std_logic_vector(bits-1 downto 0);
    B     : in    std_logic_vector(bits-1 downto 0);
    CIN   : in    std_logic;
    --outputs
    S     : out   std_logic_vector(bits-1 downto 0) :=  (others=>'0');
    COUT  : out   std_logic                         :=  '0'
  );
end parameterized_adder;
```

Module:

`parameterized_subtractor`

Entity:

```
entity parameterized_subtractor is
  generic
  (
    bits        : integer           :=  0
  );
  port
  (
    --inputs
    A     : in    std_logic_vector(bits-1 downto 0);
    B     : in    std_logic_vector(bits-1 downto 0);
    --outputs
    O     : out   std_logic_vector(bits-1 downto 0) :=  (others=>'0')
  );
end parameterized_subtractor;
```

Module:

`parameterized_multiplier`

Entity:

```
entity parameterized_multiplier is
  generic
  (
    bits        : integer           :=  0
  );
  port
  (
    --inputs
    A     : in    std_logic_vector(bits-1 downto 0);
    B     : in    std_logic_vector(bits-1 downto 0);
    --outputs
    S     : out   std_logic_vector((2*bits)-1 downto 0) :=  (others=>'0')
  );
end parameterized_multiplier;
```

Module:

`parameterized_variable_shifter`

Entity:

```
entity parameterized_variable_shifter is
  generic
  (
    bits          : integer   :=  0;
    select_bits   : integer   :=  0;
    direction     : std_logic :=  '0' --0=right,1=left
  );
  port
  (
    --inputs
    I             : in    std_logic_vector(bits-1 downto 0);
    S             : in    std_logic_vector(select_bits-1 downto 0);
    CLEAR         : in    std_logic;
    --outputs
    O             : out   std_logic_vector(bits-1 downto 0)
  );
end parameterized_variable_shifter;
```

Module:

`delay_block`

Entity:

```
entity delay_block is
  generic
  (
    bits  : integer :=  0;
    delay : integer :=  0
  );
  port
  (
    --inputs
    A         : in  std_logic_vector(bits-1 downto 0);
    CLK       : in  std_logic;
    --outputs
    A_DELAYED : out std_logic_vector(bits-1 downto 0) :=  (others=>'0')
  );
end delay_block;
```

Module:

`parameterized_absolute_value`

Entity:

```
entity parameterized_absolute_value is
  generic
  (
    bits  :       integer                              :=  0
  );
  port
  (
    --inputs
    IN1   : in    std_logic_vector(bits-1 downto 0);
    --outputs
    EXC   : out   std_logic                            :=  '0';
    OUT1  : out   std_logic_vector(bits-1 downto 0)    :=  (others=>'0')
  );
end parameterized_absolute_value;
```

Module:

`parameterized_priority_encoder`

Entity:

```
entity parameterized_priority_encoder is
  generic
  (
    man_bits    : integer :=  0;
    shift_bits  : integer :=  0
  );
  port
  (
    --inputs
    MAN_IN        : in    std_logic_vector(man_bits-1 downto 0);
    --outputs
    SHIFT         : out   std_logic_vector(shift_bits-1 downto 0) :=  (others=>'0');
    EXCEPTION_OUT : out std_logic                                 :=  '0'
  );
end parameterized_priority_encoder;
```

Module:

`parameterized_mux`

Entity:

```
entity parameterized_mux is
  generic
  (
    bits        : integer           :=  0
  );
  port
  (
    --inputs
    A     : in    std_logic_vector(bits-1 downto 0);
    B     : in    std_logic_vector(bits-1 downto 0);
    S     : in    std_logic;
    --outputs
    O     : out   std_logic_vector(bits-1 downto 0) := (others=>'0')
  );
end parameterized_mux;
```

Module:

`parameterized_comparator`

Entity:

```
entity parameterized_comparator is
  generic
  (
    bits        : integer           :=  0
  );
  port
  (
    --inputs
    A           : in    std_logic_vector(bits-1 downto 0);
    B           : in    std_logic_vector(bits-1 downto 0);
    --outputs
    A_GT_B      : out   std_logic   :=  '0';
    A_EQ_B      : out   std_logic   :=  '0';
    A_LT_B      : out   std_logic   :=  '0'
  );
end parameterized_comparator;
```

Module:

`denorm`

Entity:

```
entity denorm is
  generic
  (
    exp_bits    : integer :=  0;
    man_bits    : integer :=  0
  );
  port
  (
    --inputs
    IN1           : in    std_logic_vector(exp_bits+man_bits downto 0);
    READY         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    --outputs
    OUT1          : out   std_logic_vector(exp_bits+man_bits+1 downto 0)  :=  (others=>'0');
    DONE          : out   std_logic                                       :=  '0';
    EXCEPTION_OUT : out   std_logic                                       :=  '0'
  );
end denorm;
```

Module:

`rnd_norm`

Entity:

```
entity rnd_norm is
  generic
  (
    exp_bits      : integer :=  0;
    man_bits_in   : integer :=  0;
    man_bits_out  : integer :=  0
  );
  port
  (
    --inputs
    IN1           : in    std_logic_vector((exp_bits+man_bits_in) downto 0);
    READY         : in    std_logic;
    CLK           : in    std_logic;
    ROUND         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    --outputs
    OUT1          : out   std_logic_vector((exp_bits+man_bits_out) downto 0)  :=  (others=>'0');
    DONE          : out   std_logic                                           :=  '0';
    EXCEPTION_OUT : out   std_logic                                           :=  '0'
  );
end rnd_norm;
```

Module:

`fp_add`

Entity:

```
entity fp_add is
  generic
  (
    exp_bits      : integer :=  0;
    man_bits      : integer :=  0
  );
  port
  (
    --inputs
    OP1           : in    std_logic_vector(man_bits+exp_bits downto 0);
    OP2           : in    std_logic_vector(man_bits+exp_bits downto 0);
    READY         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    CLK           : in    std_logic;
    --outputs
    RESULT        : out   std_logic_vector(man_bits+exp_bits+1 downto 0)  :=  (others=>'0');
    EXCEPTION_OUT : out   std_logic                                       :=  '0';
    DONE          : out   std_logic                                       :=  '0'
  );
end fp_add;
```

Module:

`fp_sub`

Entity:

```
entity fp_sub is
  generic
  (
    exp_bits      : integer :=  0;
    man_bits      : integer :=  0
  );
  port
  (
    --inputs
    OP1           : in    std_logic_vector(man_bits+exp_bits downto 0);
    OP2           : in    std_logic_vector(man_bits+exp_bits downto 0);
    READY         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    CLK           : in    std_logic;
    --outputs
    RESULT        : out   std_logic_vector(man_bits+exp_bits+1 downto 0)  :=  (others=>'0');
    EXCEPTION_OUT : out   std_logic                                       :=  '0';
    DONE          : out   std_logic                                       :=  '0'
  );
end fp_sub;
```

Module:

`fp_mul`

Entity:

```
entity fp_mul is
  generic
  (
    exp_bits      : integer :=  0;
    man_bits      : integer :=  0
  );
  port
  (
    --inputs
    OP1           : in    std_logic_vector(exp_bits+man_bits downto 0);
    OP2           : in    std_logic_vector(exp_bits+man_bits downto 0);
    READY         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    CLK           : in    std_logic;
    --outputs
    RESULT        : out   std_logic_vector(exp_bits+(2*man_bits) downto 0)  :=  (others=>'0');
    EXCEPTION_OUT : out   std_logic                                         :=  '0';
    DONE          : out   std_logic                                         :=  '0'
  );
end entity;
```

Module:

`fix2float`

Entity:

```
entity fix2float is
  generic
  (
    fix_bits  : integer :=  0;
    exp_bits  : integer :=  0;
    man_bits  : integer :=  0
  );
  port
  (
    --inputs
    FIXED         : in  std_logic_vector(fix_bits-1 downto 0);
    ROUND         : in  std_logic;
    EXCEPTION_IN  : in  std_logic;
    CLK           : in  std_logic;
    READY         : in  std_logic;
    --outputs
    FLOAT         : out std_logic_vector(exp_bits+man_bits downto 0)  :=  (others=>'0');
    EXCEPTION_OUT : out std_logic                                     :=  '0';
    DONE          : out std_logic                                     :=  '0'
  );
end fix2float;
```

Module:

`float2fix`

Entity:

```
entity float2fix is
  generic
  (
    fix_bits  : integer :=  0;
    exp_bits  : integer :=  0;
    man_bits  : integer :=  0
  );
  port
  (
    --inputs
    FLOAT         : in  std_logic_vector(exp_bits+man_bits downto 0);
    ROUND         : in  std_logic;
    EXCEPTION_IN  : in  std_logic;
    CLK           : in  std_logic;
    READY         : in  std_logic;
    --outputs
    FIXED         : out std_logic_vector(fix_bits-1 downto 0)         :=  (others=>'0');
    EXCEPTION_OUT : out std_logic                                     :=  '0';
    DONE          : out std_logic                                     :=  '0'
  );
end float2fix;
```

Appendix B

VHDL Description of the IEEE Single Precision Adder

```
--=========================================================--
--                       LIBRARIES                         --
--=========================================================--
-- IEEE Libraries --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
-- float
library PEX_Lib;
use PEX_Lib.float_pkg.all;
-----------------------------------------------------------
--            IEEE Single Precision Adder               --
-----------------------------------------------------------
entity single_precision_adder is
  port
  (
    --inputs
    IN1           : in    std_logic_vector(31 downto 0);
    IN2           : in    std_logic_vector(31 downto 0);
    READY         : in    std_logic;
    EXCEPTION_IN  : in    std_logic;
    ROUND         : in    std_logic;
    CLK           : in    std_logic;
    --outputs
    OUT1          : out   std_logic_vector(31 downto 0) :=  (others=>'0');
    EXCEPTION_OUT : out   std_logic                     :=  '0';
    DONE          : out   std_logic                     :=  '0'
  );
end single_precision_adder;
-----------------------------------------------------------
--            IEEE Single Precision Adder               --
-----------------------------------------------------------
architecture single_precision_adder_arch of single_precision_adder is
  signal  rd1       : std_logic                     :=  '0';
  signal  rd2       : std_logic                     :=  '0';
  signal  rd3       : std_logic                     :=  '0';
  signal  rd4       : std_logic                     :=  '0';
  signal  exc1      : std_logic                     :=  '0';
  signal  exc2      : std_logic                      :=  '0';
  signal  exc3      : std_logic                      :=  '0';
  signal  exc4      : std_logic                      :=  '0';
  signal  rnd1      : std_logic                      :=  '0';
  signal  rnd2      : std_logic                      :=  '0';
  signal  rnd3      : std_logic                      :=  '0';
  signal  rnd4      : std_logic                      :=  '0';
  signal  op1       : std_logic_vector(32 downto 0) :=  (others=>'0');
  signal  op2       : std_logic_vector(32 downto 0) :=  (others=>'0');
  signal  sum       : std_logic_vector(33 downto 0) :=  (others=>'0');
begin
  --instances of components
  denorm1: denorm
    generic map
    (
      exp_bits      =>  8,
      man_bits      =>  23
    )
    port map
    (
      --inputs
      IN1           =>  IN1,
      READY         =>  READY,
      EXCEPTION_IN  =>  EXCEPTION_IN,
      --outputs
      OUT1          =>  op1,
      DONE          =>  rd1,
      EXCEPTION_OUT =>  exc1
    );
  denorm2: denorm
    generic map
    (
      exp_bits      =>  8,
      man_bits      =>  23
    )
    port map
    (
      --inputs
      IN1           =>  IN2,
      READY         =>  READY,
      EXCEPTION_IN  =>  EXCEPTION_IN,
      --outputs
      OUT1          =>  op2,
      DONE          =>  rd2,
      EXCEPTION_OUT =>  exc2
    );
  adder: fp_add
    generic map
    (
      exp_bits      =>  8,
      man_bits      =>  24
    )
    port map
    (
      --inputs
      OP1           =>  op1,
      OP2           =>  op2,
      READY         =>  rd3,
      EXCEPTION_IN  =>  exc3,
      CLK           =>  CLK,
      --outputs
      RESULT        =>  sum,
      EXCEPTION_OUT =>  exc4,
      DONE          =>  rd4
    );
  rnd_norm1: rnd_norm
    generic map
    (
      exp_bits      =>  8,
      man_bits_in   =>  25,
      man_bits_out  =>  23
    )
    port map
    (
      --inputs
      IN1           =>  sum,
      READY         =>  rd4,
      CLK           =>  CLK,
      ROUND         =>  rnd4,
      EXCEPTION_IN  =>  exc4,
      --outputs
      OUT1          =>  OUT1,
      DONE          =>  DONE,
      EXCEPTION_OUT =>  EXCEPTION_OUT
    );
  rd3   <=  rd1   AND rd2;
  exc3  <=  exc1  OR  exc2;
  main: process (CLK)
  begin
    if(rising_edge(CLK)) then
      rnd4  <=  rnd3;
      rnd3  <=  rnd2;
      rnd2  <=  rnd1;
      rnd1  <=  ROUND;
    end if;--CLK
  end process;--main
end single_precision_adder_arch;--end of architecture
```