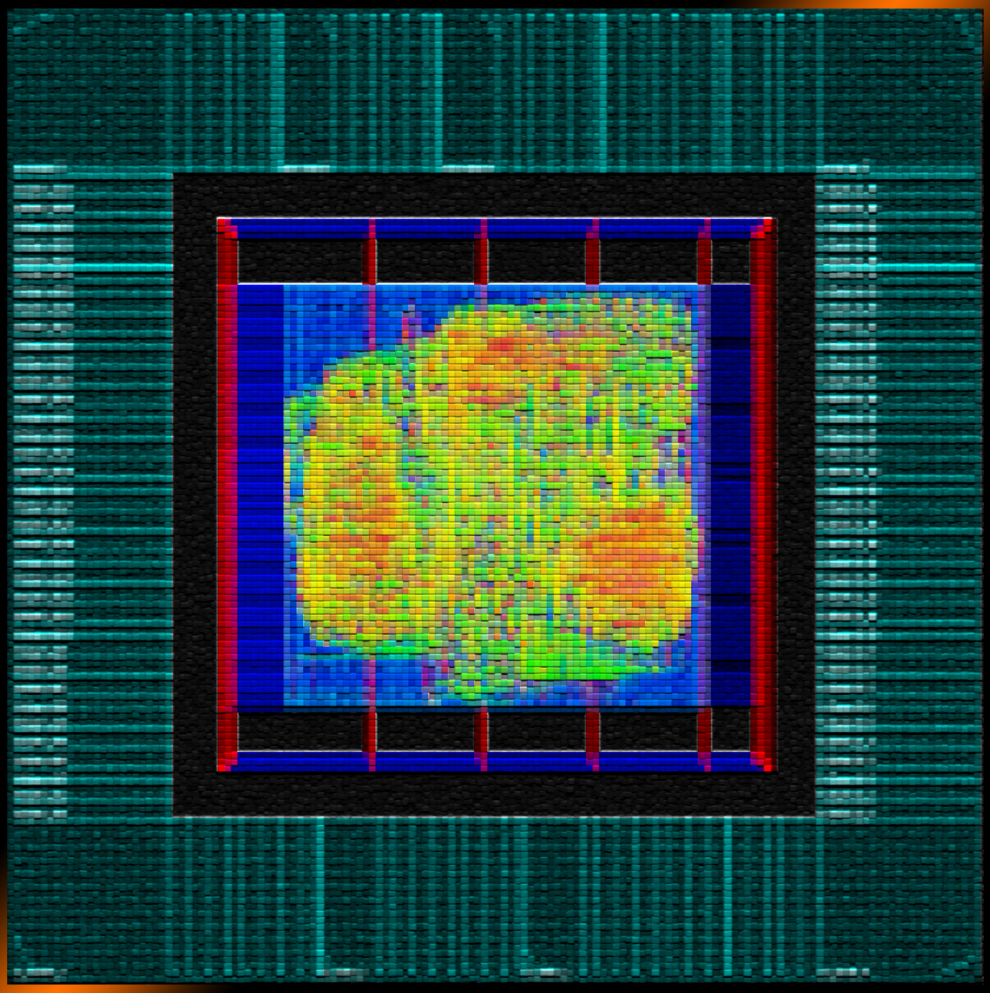
Tom M. Bruintjes



Design of a Fused Multiply-Add Floating-Point and Integer Datapath

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architectures for Embedded Systems chair

Design of a Fused Multiply-Add Floating-Point and Integer Datapath

Master's thesis by

Tom M. Bruintjes

Graduation committee:
ir. Karel H.G. Walters
dr.ir. Sabih H. Gerez
ir. Bert Molenkamp

Enschede, the Netherlands, May 16, 2011

Abstract

Traditionally floating-point and integer arithmetic have always been separated both spatially and conceptually. Even though the floating-point unit is an integral part of most contemporary microprocessors, it uses its own dedicated set of arithmetic components. Due to the high data width of floating-point numbers, these arithmetic components occupy a significant percentage of the silicon area needed for a processor. Low-cost and low-power driven processor design, which is becoming increasingly more important due to the ever growing market for battery-operated hand held devices and the need for sustainable usage of energy resources, are therefore difficult targets for floating-point arithmetic.

In this thesis we present a solution in the form of a new architecture that combines integer and floating-point arithmetic in a single datapath. Both types of arithmetic are tightly integrated by mapping functionality to the same basic hardware components (the multipliers, adders, comparators etc.). The advantage of such an approach is two-fold. Because the floating-point unit can be scheduled for integer instruction, we are able to cut-down on integer dedicated resources making floating-point units justifiable in a low-cost environment. Additionally, the hardware needed for floating-point arithmetic can be used much more efficiently, because in realistic scenarios then the amount of floating-point instructions performed is much less that a typical floating-point unit can process.

The architecture we present is tailored for a minimal silicon area and energy-efficiency. However, performance also remains an important factor. A particularly powerful architecture known as fused multiply-add (FMA) is chosen as the base for a floating-point unit with integrated integer functionality. Besides higher throughput, the added value of floating-point fused multiply-add (A×B+C) is higher accuracy, a result of the fact that only a single rounding operation is performed per instruction. From an area conservative point of view, FMA is also eligible. Instructions such as multiplication and addition/subtraction can simply be derived by using 0 and 1 for the addend (C) and multiplicand (B) respectively, hence there is no need for hardware that implements basic multiplication and addition. The architecture is further optimized for area efficiency and performance by using smart design principles like Parallel Alignment, Partial Product Multiplication, End-Around Carry Addition, Leading Zero Anticipation, Leading Zero Detection and high component re-use. The leading zero detection circuit is worth mentioning explicitly. A new approach based on earlier work [1] is presented that yields much better area (up to almost 50% reduction) for input that is not a power of two.

The resulting design is a balanced three stage pipeline with considerable integer re-use. The floating-point arithmetic is numerically compliant with IEEE-754, based on a 41-bit (8-bit exponent and 32-bit significand sign-magnitude) floating-point representation. Integer arithmetic is performed in 32-bit signed two's complement format. As a proof of concept, a VHDL structural description is implemented in STMicroelectronics 65nm technology. A performance driven implementation reaches a theoretical peak bandwidth of 2.4 GFLOPs at 1200MHz, and a low-power implementation yields a circuit that can be clocked at a maximum frequency of 500MHz. Post synthesis/place-and-route estimates of area and power consumption are provided. Comparisons with other architectures and a realistic scenario for system-on-chip (SoC) integration show that the architecture is suitable for low-cost energy-efficient hardware solutions.

Preface

From collecting rare and exotic pieces from the past to designing the next generation of chips myself, hardware has always been of great interest to me. So when I went looking for a thesis project, I knew immediately which people I had to ask. After a short discussion with my curriculum advisor and fellow hardware/VHDL enthusiast Bert Molenkamp, it was concluded that Karel Walters would most likely have interesting ideas within my area of interest. And sure, as always (I had the pleasure to work with Karel on several other occasions in the past) he had an idea that could be investigated. It was to create or modify a floating-point unit such that it can just as easily process integer data. A most unusual approach but exactly the kind of thing I was looking for. It is well known that floating-point is among the most challenging subjects in processor design. Yet, at the time Karel was explaining me his idea, I had already decided that I did not need to look any further.

As can be expected with floating-point hardware, it took me a while to complete this design. A year after starting, there is finally a correctly working datapath and this comprehensive report. I hope thatthe latter has been written thoroughly enough and that the ideas presented here will prove to be useful. The time that I have spent on the results presented in this thesis has been a year well spent. Working in the CAES group[1] is great, the brilliant discussions during the coffee break and the overall open atmosphere undoubtedly make floor 4 the best place to be during a normal working day. Thank you all CAES-people.

I would not have been able to complete all this work without the help and guidance of my committee: Karel Walter, Bert Molenkamp and Sabih Gerez. I would like to take the opportunity to express my gratitude to them. Karel, thanks for your day-to-day supervision. The time that you spent helping me understand and master the ASIC tool chain and the many discussions we had regarding complex arithmetic design principles, I highly appreciate them. Bert, of course being a VHDL specialist but also someone that pays attention to the finest details, thank you for guiding me during the final phase but also during my entire master's curriculum. Also, I would like to say that although very convenient, it can also be a little frustrating that after spending several hours on a VHDL issue someone comes up with the answer in just under a minute. Last but not least Sabih, your sharp remarks and input have helped improve my writing considerably. I especially want to convey my thanks to you for taking the time to review my work when time was pressing and there was little of it.

Tom Bruintjes
Enschede, May 2011

[1] `http://caes.ewi.utwente.nl/caes/`

Acronyms

ALU arithmetic logic unit

ASIC application specific integrated circuit

BCD binary coded decimal

CISC complex instruction set computer

CMOS complementary metal oxide semiconductor

CSA carry-save adder

DSP digital signal processing

EPIC explicitly parallel instruction computing

FA full adder

FFT fast fourier transform

FIR finite impulse response

FLOP floating-point operation

FMA fused multiply-add

FPGA field programmable gate array

GCC GNU compiler collection

GPP general purpose processor

GPSVT general purpose standard voltage threshold

GPU graphics processing unit

HDL hardware description language

IC integrated circuit

IP intellectual property

ISA instruction set architecture

LPHVT low power high voltage threshold

LSB least significant bit

LZA leading zero anticipation

LZD leading zero detection

MAC multiply-accumulate

MFU mutable function unit

MIMD multiple instruction stream, multiple data stream

MPPB massively parallel processor breadboarding

MPSoC multiprocessor system-on-chip

MSB most significant bit

NaN tot-a-number

NoC network-on-chip

PPE Power PC element

RISC reduced instruction set computer

RMS root mean square

RTL register transfer level

SIMD single instruction multiple data

SNR signal to noise ratio

SoC system-on-chip

SPE synergistic processing element

SQNR signal to quantization noise ratio

SRA shift right arithmetic

SRL shift right logical

ULP units in the last place

VLIW very long instruction word

VHDL VHSIC hardware description language

VHSIC very high speed integrated circuit

Contents

1 Introduction

The need for energy-efficient, low-cost hardware solutions has never been higher. With the ongoing growth in the average number of battery operated hand held devices per person, this is no surprise. Perhaps an even more important drive is the fact that sustainable management of energy resources is now truly becoming a pressing matter. On the other hand, the demand for more processing power is also increasing. Think of the ever improving quality in real-time 3D graphics, combining more and more functionality into a single device (e.g., smart phones) but also less evident examples such as the many embedded systems in for example TVs, cars, microwaves and washing machines. Combining high performance and energy-efficiency is only possible when algorithms and available hardware resources are analyzed up to the finest details, and then tightly coupled to each other. Currently some of the most efficient hardware solutions are achieved with heterogeneous SoCs that are (to some extent) tailored to a specific domain (e.g, digital signal processing (DSP)).

The current state-of-the art in energy efficient hardware platforms is dominated by multiprocessor system-on-chips (MPSoCs), fabricated with low-power technologies (e.g., [2]). Such architectures often comprise a fast, low-cost, power-efficient RISC processors such as the ARM [3], several 'number crunching' streaming processors and a network-on-chip (NoC) for energy-lean on-chip communication. These heterogeneous multi-core architectures are highly efficient, yet their support for floating-point operations is weak and sometimes completely lacking. This can be explained by the fact that the physical properties of a floating-point unit often conflict with the targets (area and power constraints) set out for such hardware platforms. In most cases the floating-point unit is substituted by fixed-point arithmetic or emulated by software. Both alternatives are viable from an energy and area conserving point of view, however, in terms of performance (either expressed as raw processing power or simply the kind of range and precision that is supported), they are not very satisfactory.

For fractional computations we would rather have a real floating-point unit on-board. However, how can we justify a floating-point unit in hardware solutions that are supposed to be energy efficient and area conservative? One way to look at is is that once the floating-point unit is there, we better make the best possible use of it. Preferably resulting in meaningful silicon usage close to 100% of the time. In a more realistic scenario, the usage of floating-point hardware will most likely not even reach 50%. If we could schedule the floating-point unit for the more common integer operations, the benefits would be two-fold. The needed hardware will be used more efficiently and we may be able to reduce the amount of integer specific silicon which would be beneficial both in terms of area and energy consumption.

However, there is still a clear gap between floating-point and integer arithmetic, both spatially and conceptually. Research needs to be conducted to find out how integer operation can be mapped to a floating-point unit. In this thesis the feasibility of combining floating-point and integer arithmetic into a single datapath is therefore investigated.

1.1 Motivation and Problem Statement

High performance floating-point arithmetic coincides with large amounts of silicon. This does not combine well with the targets we generally have in mind for low-cost energy-efficient hardware. However, if we could combine integer and floating-point functionality into a single datapath, the total area of a chip could be lowered by reducing the amount hardware dedicated for integer arithmetic. The usage of fully fledged floating-point units in low-power hardware solutions by sharing its datapath with integer is highly unconventional. Currently little is known about this subject.

That being said, the central problem addressed in this thesis is the definition of a (low power and silicon driven) floating-point datapath that is capable of executing integer operations with high performance.

1.2 Research Goals

The objective set out for this thesis is the exploration of combining integer and floating-point arithmetic efficiently. The work presented here focuses on the architectural aspect of designing hardware that is capable of the aforementioned. Some of the questions raised and answered by this work include:

- What floating-point and integer formats can most efficiently be combined?
- What floating-point architectures are suitable for low-cost energy efficient hardware solutions?
- How can integer operations most efficiently be mapped to floating-point hardware?

1.3 Approach

A hybrid solution between conventional integer arithmetic logic units (ALUs) and floating-point units is proposed in an attempt to conserve area and energy. To evaluate if this approach is worthwhile, a proof of concept is made using the structural hardware description language 'VHSIC hardware description language (VHDL)'. The design is implemented in a deep sub-micron (65nm) low-power technology for realistic estimates of timing, area and power consumption.

Before investigating the possibilities for the new architecture, several requirements were set up:

- Support for at least multiplication and addition of floating-point and integer operands
- The architecture should preferably be limited to two or three pipeline stages
- The design should be fully synthesizeable in a 65 or 90nm low-power process

The second and last requirement are crucial parameters for timing considerations. The first major structural design choices are driven by latency reduction, in order to achieve high performance under these constraints. However, since area and power considerations are at least as important, the later development stages are mostly focused on minimizing area and power consumption. This two-stage design approach is reflected in this thesis. The first chapters are mostly focused on performance related subjects while the later chapters deal with area and energy-efficiency.

Furthermore, we believe that although research should go beyond the state-of-the-art, usability is also a very important aspect when proposing architectural changes. It would not be the first time a new architecture is introduced only to be neglected due to incompatibilities and steep learning curves that have to be overcome in order to use new concepts to the full potential. It is therefore imperative that the transition from floating-point to integer operation (and vice versa) should be as seamless as possible. Considerable attention is given to the subject of fluent transition with the least amount of overhead.

1.4 Thesis Overview

Chapter 2 introduces the reader to the basic principles of floating-point arithmetic. Definitions and concepts used throughout the remainder of the thesis are explained here.

Chapter 3 presents a short overview of the work related to the research conducted in this thesis. The floating-point fused multiply-add datapath and re-use of floating-point hardware for integer purpose are emphasized.

Chapter 4 proposes the basic architecture for a fused multiply-add datapath that shares its functionality with integer operations. A precise specification of the instruction set architecture is provided and the data flow for an efficient datapath discussed.

Chapter 5 discusses optimizations and design principles that are applied to the basic architecture of Chapter 4. This chapter mostly focuses on latency reduction and increased throughput to obtain good performance in low-power technology realization.

Chapter 6 elaborates on the implementation details. In contrast to Chapter 5, this chapter puts more emphasis on reducing area and energy consumption for low-cost solution such as embedded systems.

Chapter 7 evaluates the physical properties (area and power) of a 65nm low-power and general purpose implementation of the new architecture. The consequences for SoC integration are explored and a comparison is made with other floating-point solutions, to obtain a rough estimate of the overhead imposed by the concepts introduced in the previous chapters.

Chapter 8 explains how a complex architecture like a floating-point unit can be tested thoroughly with the least amount of effort. In particular when floating-point formats are used that are not strictly conform to the IEEE standard, reliable points of reference are scarce. This chapter presents an elegant solution in the form of a test bench that uses the VHDL-2008 standard for floating-point functionality.

Chapter 9 concludes the thesis by summarizing the main results. The limitations of our solution are mentioned here as well as a number of improvements to the architecture and new research topics to be investigated in the future.

2 Background

2.1 Introduction

The purpose of this thesis is to design a new kind of ALU that efficiently combines floating-point and integer arithmetic/logic. Because arithmetic and logic are very different for binary floating-point and integer operands, this is not easily achieved. One should have ample understanding of computer arithmetic before undertaking such a task. In this chapter we introduce the reader to computer arithmetic and in particular floating-point arithmetic.

2.2 Number Representation

If we are to perform arithmetic on integer and floating-point numbers, we first have to know how such numbers are represented. In a purely mathematical sense, the possibilities for representing numbers are endless. To illustrate this, Table 2.1 lists a few representations of the number 640.

What differentiates most of these representations is their *base* and their *radix point*. The base of a numeral system is determined by the amount of unique symbols available for representation. Decimal numbers for example, are all base-10 numbers. Ten unique symbols are used in decimal representation: 0,1,...,9. (A subscript often indicates the base of a number, e.g., decimal 640 becomes 640_d). In the binary numeral system, only two unique symbols are used: zero and one (`0` and `1`). Since fewer unique symbols are used, the string of symbols automatically becomes longer. The radix point is the symbol

Format	**Representation**
Decimal (3 significant numbers)	640
Decimal (5 significant numbers)	640.00
Scientific Notation	0.64×10^3
Scientific Notation Normalized	6.4×10^2
Binary	`1010000000`
Binary Coded Decimal	`0110 0100 0000`
Octal	1200
Hexadecimal	280

Table 2.1: A selection of representations for the number 640

Decimal Representation	Sign-Magnitude Representation	Two's Complement Representation
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
-0	1000	0000
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100

Table 2.2: Sign-magnitude and two's complement representation

used to separate the *integer* part from the *fractional* part.

The most relevant representation in digital computers is binary. In the binary numeral system, we usually assume that strings of `0`'s and `1`'s represent natural numbers (0,1,2,...). For example, the decimal number 136 is represented in binary by the string `10001000`. However, if we also consider negative numbers, the same string could be interpreted as -8. There are several conventions to represent binary signed numbers. The most intuitive representation is *sign magnitude*. In sign magnitude representation, the most significant bit (MSB) determines the sign of the number. A `1` often means negative while `0` means positive. The remaining bits determine the magnitude (absolute value) of the number. There are two drawbacks to sign-magnitude representation. One is that for addition and subtraction the sign-bits need to be taken into account explicitly, the other is that there are two possible representations for zero (+0 and −0). The latter is not very convenient because is makes testing for zero slightly more complex and when a result is exactly zero, it is ambiguous whether it should be +0 or −0.

Because of these drawbacks, sign-magnitude is not often used for binary arithmetic. More common is the *two's complement* notation. A formal expression [4] for n-bit two's complement numbers is

$$-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

where n indicates the index of the bits [1]. For positive numbers, the term a_{n-1} is zero, so all positive numbers in two's complement are exactly the same as in sign magnitude representation. Negative numbers on the other hand, are quite different and require slightly more effort to derive. An easy procedure to obtain the two's complement form of a certain negative number is to use one's complement as an intermediate step. To find the one's complement representation of a binary number, all bits simple need to be inverted. The two's complement is then obtained by incrementation and ignoring the carry-out. Table 2.2 provides an overview of the first four positive and negative numbers in sign magnitude and two's complement notation. The biggest advantage of two's complement notation is that the logic needed to perform arithmetic on it is much simpler than for a sign magnitude representation. The drawbacks of of the unbalanced range and the additional complexity of comparing numbers are significantly outweighed by the simplicity of implementing of two's complement arithmetic.

Note that with the number representations that were mentioned so far, we are limited to integers (0,1,2,...). It is not possible to represent a fraction like 3/2 (1.5). This limitation can be overcome by redefining the binary string, such that somewhere in the string an imaginary point is present: the

[1] Assuming big-endian notation

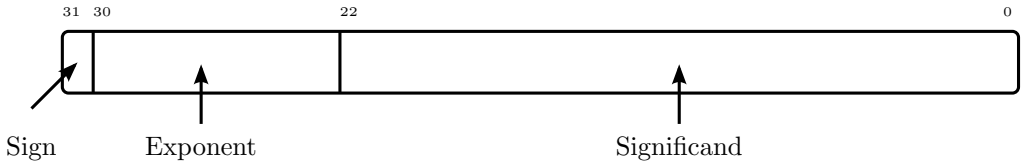


Figure 2.1: IEEE-754 single precision (32-bit) floating-point word

radix point. For binary numeral systems, this is called *fixed-point* notation. Formally a fixed-point notation is defined as a fixed number of digits to the left of a the radix point and a fixed number of digits right of the radix point. Such a representation strongly depends on the base. Representing numbers in base-10 fixed-point notation happens naturally for humans. If we want to represent 5/4, we almost automatically write down 1.25. Formally this result is derived by: $1 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} = 1 + 0.2 + 0.05 = 1.25$. For binary fixed-point, the same principle applies. The number 5/4 is represented by `1.01` $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2})$.

With fixed-point notation it is possible to represent fractions. In the example shown, the fraction had an exact decimal and binary equivalent. This is however not always the case. Often we can not find a finite, exact, representation for a number using fixed-point notation (e.g., 1/3). This is a fundamental problem for which there is no real solution, only approximations. Another problem is that the range of fixed-point numbers is severely limited. Because the number of bits before and after the radix point is fixed, it is difficult to represent very large numbers and very small numbers at the same time. The same problem is also present in the the decimal numeral system. The scientific notation is used to overcome this issue. A large number like $136,000,000,000$ becomes 1.36×10^{11} and a small number like 0.000000000136 becomes 1.36×10^{-10}. The number of symbols used for scientific notation is significantly smaller. When a similar notation is used for binary numbers, we speak of *floating-point* numbers.

2.3 Floating-Point Numbers

A floating-point number is of the form:

$$\pm \mathrm{S} \times \mathrm{B}^{\pm e}$$

where
S is called the significand (or mantissa)
B the base of the numeral system
e the exponent

Because the base of a floating-point representation is the same for every number, it does not have to be stored. Figure 2.1 shows a typical 32-bit floating-point word. This is the format used in the IEEE-754 standard for floating-point arithmetic, which will be discussed in more detail later. The leftmost bit is the sign-bit, a `0` is for positive and a `1` for negative. The next eight bits are used for the exponent. Most (modern) floating-point formats use a *biased* exponent. This means that a constant number is added to the true exponent value, such that there is no need for a representation of negative numbers. In a base-2 floating-point format, the bias of a k-bit exponent typically equals (2^{k-1}) or $(2^{k-1} - 1)$. The last segment of 23 bits is used to store the fraction. Although this part is often referred to as mantissa, Blaauw and Brooks point out in [5] that formally the mantissa is the logarithm of the significand. In this text we will refer to the fraction as the significand, which is encouraged by IEEE.

The name floating-point is derived from the fact that the radix point can be placed anywhere relative to the base. In most cases the radix point is located after the most significant bit of the significand:

$$\pm s.sss\cdots s \times 2^{\pm e}$$

To simplify floating-point operations, the significand is almost always *normalized*. A normalized number is a number whose most significant bit is not a zero. In a binary representation, this means that the first bit is always `1`. A significand like this can only represent numbers in the interval [1,2). It is therefore not necessary to store the first bit, it can be made implicit. As a consequence, the precision of the significand can be increased by one bit. On the other hand, denormalized numbers linearly fill the gap between zero and the smallest normalized number. Much smaller numbers can be represented if denormalized numbers are allowed. This allows calculations to gradually converge to zero instead of the sudden drop observed with normalized numbers.

Considering the properties mentioned above, the range of the floating-point numbers that can represented is easily determined. For example, the IEEE-754 single precision floating-point format has the following ranges:

Negative numbers:	$-(2-2^{-23})\times 2^{127}$	to	-2^{-126}
Positive numbers:	2^{-126}	to	$(2-2^{-23})\times 2^{127}$

This shows that floating-point numbers are very well capable of representing number with great precision over long ranges (unlike fixed-point). The smallest positive number follows from the smallest significand (`1.000`$\cdots$`0`) combined with the smallest exponent (-126). The largest positive number is determined by the the largest significand (`1.111`$\cdots$`0`) and the largest exponent (127). Note that the exponent ranges from -126 to 127 due to the bias 127 notation. Despite these large ranges, there are numbers that can not be represented. These number can be categorized in four regions:

1. Negative overflow: every negative number below $-(2-2^{-23})\times 2^{127}$
2. Negative underflow: every negative number above -2^{-126}
3. Positive overflow: every positive number above $(2-2^{-23})\times 2^{127}$
4. Positive underflow: every positive number below 2^{-126}

Most floating-point formats have reserved bit patterns to represent numbers from these categories. Underflow is often approximated by zero (all `0`'s) while overflow is represented by $\pm\infty$ (all `1`'s). Exceptional cases such as $\sqrt{-1}$ are symbolized by tot-a-number (NaN). Table 2.5 lists all floating-point encodings used in IEEE-754 format.

Note that there is a trade-off between precision and range. When more bits are used for the significand, the floating-point number will be more accurate. However, because only a limited amount of numbers can be represented in 32-bits, the range of numbers will decrease. The other way around results in more range but less accurate numbers. The only way to increase both accuracy and range is to use more bits. This is why a lot of floating-point units implement double (and in some cases even quadruple) precision in addition to single precision.

2.4 Floating-Point Number Representation

The floating-point example shown in the previous section is only one of many in existence. Most processor architectures used to have their own floating-point format. Manufacturers such as IBM, HP, DEC and Cray have all used proprietary floating-point formats in the past, severely limiting portability of programs depending on floating-point arithmetic. Today, almost all floating-point arithmetic is based on the IEEE-754 standard for floating-point arithmetic [6]. This standard was conceived from many years of experience. The influence of legacy floating-point arithmetic can still be found in IEEE-754. For example the formats used by IBM and Cray.

2.4.1 IBM Floating-Point Numbers

IBM has used more than one floating-point representation in the past. Their most noteworthy being the System/360 floating-point format. In this format a single precision binary floating-point number is stored in a 32-bit word. The first bit is used as a sign-bit followed by a 7-bit bias-64 (2^{k-1}) exponent and a 24-bit significand. What really sets aside the IBM format from the others, is that it uses a hexadecimal base for the exponent.

The advantage of base-16, and any large base in general, is that less *alignment* and *normalization* is required (Section 2.6). The drawback is that a larger base results in less precision.

2.4.2 Cray Floating-Point Numbers

Another influential format is the one used in the Cray-1 machines. The most notable difference between Cray and IBM formats is the increased precision and the binary base. The smallest Cray floating-point representation requires 64 bits (the same amount used by the double precision IBM representation). Floating-point numbers in Cray machines are by default twice as large as in IBM machines. The philosophy of Cray floating-point arithmetic is that with such large numbers, the occurrence of overflow and underflow is minimized. Obviously, this approach is very costly in terms of area.

Just like IBM, Cray switched to the IEEE-754 format. An interesting observation that can be made is that Cray still holds on to their original philosophy. Modern Cray computers use 64-bit floating-point operations by default. In most other architectures this corresponds to double precision where single precision is the default.

2.5 The IEEE-754 Standard for Binary Floating-Point Arithmetic

The IEEE-754 Standard for Binary Floating-Point Arithmetic [6] was introduced in 1985 with the goal to improve the portability of floating-point computations. Virtually all contemporary modern processors and co-processors support IEEE-754, making floating-point units highly compatible as opposed to the IBM and Cray formats that were discussed. The standard can be implemented in hardware or software (or a combination of both) as long as the result are guaranteed to adhere to the rules defined in [6]. The IEEE-754 standard for floating-point arithmetic defines much more than just the representation of numbers. The most important aspects are enumerated below. In the remainder of this section we will discuss the aspects of IEEE-754 standard in more detail, because a lot of work presented in this thesis deals with this floating-point format.

- Arithmetic (interchange) formats - *binary (and decimal) floating-point data*
- Operations - *operations applicable to arithmetic formats*
- Rounding - *rounding arithmetic results*
- Exceptions - *exceptional conditions occurring during arithmetic*

Arithmetic Formats

The format that IEEE has chosen consists of a signed significand and a biased exponent. The format is radix independent but only binary and decimal are officially defined in [6]. The first IEEE-754 definition included single, double and quadruple precision for the binary format, and single and double precision for the decimal format. Since 2008, half precision is also part of the standard. Custom precision formats

Precision	**Significand** (+hidden-bit)	**Exponent** (bits)	**Bias**
Binary			
half (16-bit)	11	5	15
single (32-bit)	24	8	127
double (64-bit)	53	11	1023
quadruple (128-bit)	113	15	16383
custom (k-bit, k≥128)	k - $round(4{\times}log_2(k)) + 13$ *	$round(4{\times}log_2(k)) - 13$	$2^{(k-s-1)}$-1 **
	Precision (decimal digits)	**Exponent** (decimal digits)	**Bias**
Decimal			
single (32-bit)	7	11	101
double (64-bit)	16	13	398
quadruple (128-bit)	34	17	6176
custom (k-bit, k≥32)	9×(k/32) -2	$k/16 + 9$	$3{\times}2^{k/16+3}$+s-2

* The round function rounds to the nearest integer.
** s is the significand width.

Table 2.3: Segmentation of the different formats described by IEEE-754

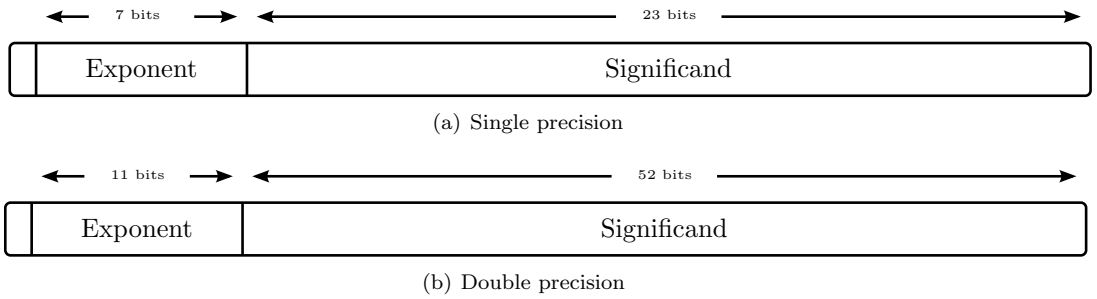
are also allowed. However, a certain ratio between exponent and significand has to be maintained. Any multiple of 32 bits can however be used. The segmentation of the different allowed floating-point words is shown in Table 2.3.

Figure 2.2(a) and 2.2(b) show the single and double precision binary floating-point numbers respectively (by far the most widely used representations). The IEEE-754 format includes an implicit (hidden) bit before the imaginary radix point. Due to *normalization*, the MSB of every floating-point number is always 1. By not explicitly storing this bit, the precision can be increased from 23 to 24 (or 52 to 53) bits. For the exponent, a $(2^{k-1}) - 1$ bias was chosen.

Operations

IEEE-754 goes further than just specifying the formats for floating-point numbers. Most arithmetic operations and rounding algorithms are also specified. This does not concern implementation details but rather the behavior. Below a short list of the most important operation is shown. We do not go into detail here.

- Arithmetic operation (e.g., add, subtract and multiply)
- Precision conversion (e.g., double to single precision)
- Scaling and quantizing



(a) Single precision

(b) Double precision

Figure 2.2: Common IEEE-754 floating-point words

Mode	Description
Round toward nearest, ties to even	Rounds toward the nearest value, if the number falls midway it is rounded to the nearest even value (LSB of `0`)
Round toward nearest, ties away from zero	Rounds to the nearest value, if the number falls midway it is rounded to the nearest larger value (for positive numbers) or smaller (for negative numbers)
Round toward 0	Rounds toward zero (i.e., truncation)
Round toward $+\infty$	Rounds toward positive infinity
Round toward $-\infty$	Rounds toward negative infinity

Table 2.4: IEEE-754 rounding modes

- Copying and manipulating signs bits
- Comparisons and ordering
- Classification and testing for exceptions
- Testing and setting flags
- Miscellaneous operations.

Rounding

It was already mentioned that most arithmetic operations do not result in a number that can be represented exactly. In such cases the result needs to be *rounded* to a number that can be represented in a given format. The IEEE-754 standard defines five rounding algorithms, listed in Table 2.4.

The most popular mode is round toward nearest, ties to even. This rounding mode generally introduces the smallest error as the result of round toward nearest is the number closest to the exact value. However, certain applications such as interval arithmetic perform better on simpler rounding mode like round toward zero. For this reason IEEE-754 includes *directed* rounding modes as well.

Exceptions

When exceptions occur, they need to be handled as described in the standard. The minimum required action taken is status bit flagging. The five exceptions covered by the standard are:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexactness.

Most of these exceptions require a unique representation. In IEEE-754 certain bit-patterns are reserved for these exceptional cases. Table 2.5 lists all possible bit patterns that can be expected in a 32-bit (single precision) floating-point result, and how they should be interpreted.

2.6 Floating-Point Arithmetic

Now that number representation has been discussed, we can focus on arithmetic. We assume that the reader is familiar with integer arithmetic, if not then [7, 4, 8] provide good starting points. Here, we

focus on basic floating-point arithmetic only.

Floating-point arithmetic is considerably more complex than integer arithmetic. We will limit our discussion to the three most basic floating-point arithmetic operations: addition/subtraction, multiplication and division. In addition, we give attention to rounding which is mandatory for most arithmetic operations. The objective is not to provide the most efficient algorithms or give an exhaustive overview of all floating-point arithmetic, but rather to show the complexity involved in computations with floating-point numbers.

2.6.1 Floating-Point Addition/Subtraction

In contrast to integer arithmetic, addition and subtraction are more complicated than multiplication and division. This is best shown by example. Suppose we want to add 2.01×10^{12} and 1.33×10^{9}.

$$\begin{array}{l} 2.01 \times 10^{12} \\ 1.33 \times 10^{9} \\ \hline \qquad\qquad + \end{array}$$

This immediately shows why floating-point addition is not straightforward. The exponents first need to be equalized before the fractions can be added. There are two possible ways to do this. The largest exponent can be decremented or the smallest exponent can be incremented. The consequence of decrementing the larger exponent is that the radix point of the fraction needs to be shifted to the right. Incrementing the smaller exponent requires shifting the radix point to the left. Assuming that a finite number of digits is used, this mean loss of precision. Left-shifting the fraction affects the MSBs of the fraction and right-shifting the LSBs. Loss of MSBs is most problematic, hence left-shifting is most undesirable. For this reason, the smaller exponent is usually incremented.

$$\begin{array}{r} 2.01 \times 10^{12} \\ 0.00133 \times 10^{12} \\ \hline \qquad\qquad + \end{array}$$

When precision is assumed to be infinite, the result of this addition is 2.01133×10^{12}. However, in a more realistic scenario, only a finite number of digits is used. If for example only three digits are used

Interpretation	Sign	Biased Exponent	Significand
Positive zero	0	0 (all 0's)	0 (all 0's)
Negative zero	1	0 (all 0's)	-0 (all 0's)
Plus infinity	0	255 (all 1's)	∞ (all 0's)
Minus infinity	1	255 (all 1's)	-∞ (all 0's)
Quiet NaN	-	255 (all 1's)	NaN (non-zero)
Signaling NaN	-	255 (all 1's)	NaN (non-zero)
Positive nonzero (normalized)	0	any number	1.any number
Negative nonzero (normalized)	1	any number	1.any number
Positive nonzero (denormalized)	0	0 (all 0's)	0.any number
Negative nonzero (denormalized)	1	0 (all 0's)	0.any number

Table 2.5: Interpretation of the IEEE-754 format

to represent the fraction in the example above, the result becomes 2.01×10^{12}. The last three bits are truncated. In this particular example the result is already normalized and needs no further processing. Often this is not the case. Assume a certain intermediate result of 0.0201×10^{10}. To normalize this number, the first non-zero digit (2) needs to occupy the first position of the fraction. This means the fraction needs to be shifter to the left twice, and the exponent incremented twice as well.

The most basic algorithm for floating-point addition and subtraction can be described by the exact same actions shown in this example. From a highly simplified point of view, the algorithm can indeed roughly be divided into three phases.

1. Aligning significands
2. Adding/Subtracting significands
3. Normalizing the result

Overflow and underflow occurrences are quite frequent: any of the three pases can result in overflow or underflow. In some cases overflows and underflows can even be triggered without actual overflow/underflow occurring. These cases have to be detected and compensated. In addition problems can arise due to zero operands being used. This illustrates that floating-point addition/subtraction is much more complex than often thought.

Using Algorithm 2.6.1, addition and subtraction is explored more thoroughly. The input of this algorithm is assumed to be formatted according to Table 2.3. For every operand N, the exponent is indicated by N_e, the significand by N_s and the sign by N_{sign}. Also, before any operation starts, the hidden-bits must first be made explicit.

The initial steps of the implementation are preparatory. Addition can be turned into subtraction by inverting the sign-bit of the subtrahend (operand B). If one of the two operands is zero, the result simply equals the other operand. In such cases, the algorithm simply halts and returns the value of the other operand. If the result is not zero, the next step is to align the exponents such that they equal.

The decimal example already showed that shifting to the right is preferred over shifting to the left because the loss involved in right-shifting is less severe than left-shifting. Alignment is achieved by repeatedly shifting the significand of the smallest number one digit to the right and incrementing the exponent until the exponents are equal. If this result in the smallest operand becoming zero, the other operand is returned.

When the exponents are equal, the significands can be added. The actual addition is the same as for integers. The result may overflow due to this addition. This can be corrected by shifting the significand to the right and incrementing the exponent. If the exponent also overflows, the result truly overflows and an exception occurs. Exponent overflow can not be corrected, hence the overflow condition must be reported and the algorithm halts.

The final step is to normalize the result. Normalization is almost the opposite of the alignment. The significand is shifted to the left until the first digit is no longer a zero. The exponent is decremented each time the significand shifts a position to the left. Due to the shifting process, underflow may occur in the exponent. Underflow can not be corrected and should be reported. If no underflow occurs, the number can be rounded and the final result returned. Rounding is postponed to the very last moment to minimize its effect on the precision of the result. Rounding itself will be discussed in Section 6.7.

2.6.2 Floating-Point Multiplication

Floating-point multiplication and division are much simpler than addition/subtraction. Let us first look at multiplication. In decimal scientific notation two numbers are multiplied by adding the exponents and multiplying the fractions. For example, 2.01×10^{12} multiplied by 1.33×10^{9} equals $(2.01 \times 1.33) \times 10^{(9+12)}$ $= 2.6733 \times 10^{21}$. The algorithm for floating-point multiplication, which is based on this principle, is shown

Algorithm 2.6.1 Floating-point addition/subtraction

Input: Normalized floating-point operands: A, B
Output: Normalized floating-point result of A-B: Result

1: **if** opcode = subtract **then**
2: B_{sign} = not(B_{sign})
3: **end if**
4: **if** A = 0 **then**
5: Result ← B
6: halt
7: **else if** B = 0 **then**
8: Result ← A
9: halt
10: **else if** $A_e \neq B_e$ **then**
11: "Designate smaller exponent operand as N1 and the other as N2"
12: **while** $N1_e \neq N2_e$ **do**
13: Increment $N1_e$
14: Right-shift $N1_s$
15: **if** $N1_s = 0$ **then**
16: Result ← N2
17: halt
18: **end if**
19: **end while**
20: **end if**
21: $Result_s \leftarrow A_s + B_s$
22: **if** $Result_s = 0$ **then**
23: Result ← 0
24: halt
25: **else if** R_s overflows **then**
26: Right-shift R_s
27: Increment R_e
28: **if** R_e overflows **then**
29: Report overflow
30: halt
31: **end if**
32: **end if**
33: **while** $Result_s$ is not normalized **do**
34: Left-shift $Result_s$
35: Decrement $Result_e$
36: **if** $Result_e$ underflows **then**
37: Report underflow
38: halt
39: **end if**
40: **end while**
41: Round Result
42: halt

in Algorithm 2.6.2. The pre-conditions for addition/subtraction also hold for multiplication. We assume that the input is of the normalized IEEE format and that the hidden-bit has been made explicit.

First the operands are checked for zero. If either of the two is zero, the result immediately becomes zero and the algorithm halts. The exponents are added in the next step. Because both exponents are biased, the bias accumulates when exponents are added. To compensate for the extra bias addition, the bias is subtracted from the resulting exponent again. After subtraction, the result is checked for overflow and underflow. If one of these exceptions is detected, this will be reported and the algorithm halts.

If the exponent is still within range, the significands can be multiplied. This multiplication is performed the same way as for integers. In sign-magnitude only the magnitudes need to be multiplied (the sign is simply the XOR of the two input signs), However, the multiplication can also be performed in two's complement notation for better performance (Chapter 5). In both cases, the product will be at least double the length of the input operands. These extra bits are dropped in the rounding stage. The multiplication is followed by the normalization and rounding steps as described for addition/subtraction.

Algorithm 2.6.2 Floating-point multiplication

Input: Normalized floating-point operands: A, B
Output: Normalized floating-point result of A×B: Result

```
1:  if A_e = 0 OR B_e = 0 then
2:    Result ← 0
3:    halt
4:  else
5:    Result_e ← A_e + B_e
6:    Result_e ← Result_e - bias
7:    if Result_e overflows then
8:      Report overflow
9:      halt
10:   else if Result_e underflows then
11:     Report underflow
12:     halt
13:   else
14:     Result_s ← A_s × B_s
15:     while Result_s is not normalized do
16:       Left-shift Result_s
17:       Decrement Result_e
18:       if Result_e underflows then
19:         Report underflow
20:         halt
21:       end if
22:     end while
23:     Round Result
24:     halt
25:   end if
26: end if
```

2.6.3 Floating-Point Division

The division algorithm (Algorithm 2.6.3) is very similar to multiplication. However, instead of adding the exponents, they are subtracted and instead of multiplying the fractions they are divided.

The first step is testing for zero again. If the divisor is zero, an error occurs (division by zero) and result is asserted to NaN. Some non-IEEE implementations may set the result to infinity instead. If the

dividend is zero then the result automatically also becomes zero. In the next step, the divisor exponent is subtracted from the dividend exponent. The bias accumulation must be compensated again.

The result is tested for underflow and overflow, and when applicable an exception is raised. If no exceptions occur, the dividend significant is divided by the divisor significand. The final steps are again normalization and rounding.

Algorithm 2.6.3 Floating-point division

Input: Normalized floating-point operands: A, B
Output: Normalized floating-point result of A/B: Result

```
1: if A_e = 0 then
2:    Result ← 0
3:    halt
4: else if B_e = 0 then
5:    Result ← NaN
6:    halt
7: else
8:    Result_e ← A_e - B_e
9:    Result_e ← Result_e - bias
10:   if Result_e overflows then
11:      Report overflow
12:      halt
13:   else if Result_e underflows then
14:      Report underflow
15:      halt
16:   else
17:      Result_s ← A_s / B_s
18:      while Result_s is not normalized do
19:         Left-shift Result_s
20:         Decrement Result_e
21:         if Result_e underflows then
22:            Report underflow
23:            halt
24:         end if
25:      end while
26:      Round Result
27:      halt
28:   end if
29: end if
```

2.6.4 Multiply-Accumulate

Multiplication and addition are sometimes combined to a single operation called multiply-accumulate. Certain applications (e.g., matrix multiplication) perform this operation so often that it is worthwhile to implement the operation in hardware. Such multiply-accumulate (MAC) units are for example often found in digital signal processors. Multiply-accumulate can also be implemented for floating-point numbers. When floating-point multiplication and addition are combined with only a single rounding operation, we speak of FMA. FMA not only offers improved performance, the precision also increases due to the elimination of a rounding operation.

2.6.5 Rounding

Because floating-point numbers have to be represented in a finite number of digits, there is only a limited amount of numbers within a certain range that can be represented. Most numbers can therefore not be represented exactly, such that rounding is required to find the closest possible representation.

As the exponent of floating-point numbers increases, so does the space between the two closest representable numbers. This means that numbers closer to zero can be represented more accurately than numbers further away from zero. For example the first number after $\texttt{1.11110}\times 2^{\mathbf{1}}$ (1.9375_d) is $\texttt{1.11111}\times 2^{\mathbf{1}}$ (1.96875_d), a difference of 0.03125_d. The difference between $\texttt{1.11110}\times 2^{\mathbf{5}}$ (62_d) and $\texttt{1.11111}\times 2^{\mathbf{5}}$ (63_d) is already 1.0_d.

The above emphasizes that inexactness is almost inherent in floating-point arithmetic. It is therefore important to have a means of measuring this error. Consider a decimal floating-point format with a precision of three digits. If the result of an arbitrary operation is 3.12×10^{-1} and the result of an infinite precise computation is 0.314159, it is common practice to identify the error as 2 units in the last place (ULP). Similarly, if the 'exact' number 0.0314159 is represented by 3.14×10^{-2}, then the error is 0.159 ULP. When floating-point format $s.ss\cdots ss \times \beta^e$ is used to represent an arbitrary number n, the the inexact error [9] is measured by:

$$\left| s.ss\cdots ss - \left(\frac{n}{\beta^e}\right) \right| \times \beta^{p-1}$$

where
- β is the base of the floating-point format
- e the exponent
- p the precision

To improve arithmetic precision, operations are often performed with more precision than the register formats provide (i.e., when 23 bits are used to store the significand, arithmetic is performed with 25-bit precision). IBM (Section 2.4.1) already introduced the *guard-bit* with their System/360. A single bit to the right of the LSB was used to store the last bit that is shifted out during alignment of two exponents. All computations performed with one addition bit of significance, produce surprisingly more accurate results.

The importance of a guard-bit can easily be demonstrated with a small example. Suppose two floating-point numbers that are close in value are to be subtracted. For example $\texttt{1.00000}\times 2^1 - \texttt{1.11111}\times 2^0$ ($2_d - 1.96875_d$).

$$\begin{array}{l} \texttt{1.00000} \times 2^1 \\ \underline{\texttt{1.11111} \times 2^0 \quad -} \end{array}$$

To subtract the smaller number from the other, it must be shifted to the right.

$$\begin{array}{l} \texttt{1.00000} \times 2^1 \\ \underline{\texttt{0.11111} \times 2^1 \quad -} \\ \texttt{0.00001} \times 2^1 \end{array}$$

One bit of significance is now lost. The loss in precision (*cancellation*) can become so large, that every digit of the result becomes meaningless. After normalization, this example results in `1.00000` $\times 2^{-4}$. Let us now compare this result with an infinitely precise computation. The 6-bit restricted example yielded 0.0625_d. If it would have been performed with infinite precision, the result would have been 2_d - 1.96875_d = 0.03125_d. The error that is introduced here is 100% (1 ULP). Now we perform the same computation with one guard-bit.

$$\begin{array}{l} \texttt{1.00000 0} \times 2^1 \\ \underline{\texttt{0.11111 1} \times 2^1} - \\ \texttt{0.00000 1} \times 2^1 \end{array}$$

The result is now `0.00000 1`$\times 2^1$, which is 0.03125_d. Notice that this is precisely the result we got from performing the subtraction with infinite precision. In this case, the guard-bit completely eliminates the inexact error. Unfortunately this is not always the case. If however, two guard bits and a *sticky-bit* are used in conjunction, results can be computed as if they were infinitely precise and then rounded [9]. The sticky-bit is called 'sticky' because once this bit becomes `1` during alignment, it keeps this value. IEEE-754 requires that operations are performed as if they are infinitely precise. Hence, the majority of IEEE-754 compatible floating-point units maintain two guard bits (a guard-bit and a *round-bit*) and compute a sticky-bit for rounding.

The additional bits must be disposed of before the result is written back to memory. This is achieved by actual rounding. We already mentioned that there are several rounding policies that can be applied for IEEE-754 compatible rounding. Based on such a policy, the intermediate result is either incremented and truncated or just truncated. Rounding routines using guard, round and stick bits, perform pattern matching to implement these IEEE-754 policies. The patterns to be found in the guard round and sticky bits can be derived from analyzing inexactness.

For round to nearest this means that if the bits to the right of the LSB of the normalized result have weight $> \frac{1}{2}$ ULP (`101` or `110`), the result is rounded up. If the bits have have weight $< \frac{1}{2}$ ULP, the result is rounded down. When the bits are exactly $\frac{1}{2}$ ULP the result must be rounded to the nearest even number. For round to zero, only truncation is applied. Note that this requires the least amount of processing. Some floating-point units only implement round to zero because this considerably simplifies the rounding stage, allowing higher clock frequencies. In Chapter 5, the implementation of the different IEEE-754 rounding algorithms is explained more thoroughly.

2.7 Summary

The basic algorithms for floating-point arithmetic have been shown in a simplified form. From this it should have become clear that although floating-point arithmetic is much more complicated than integer arithmetic, the essence is similar. The operations performed on the significand are the same as performed on integers. Combining integer and floating-point operations on a single datapath, seems therefore a promising solution for area and energy critical hardware platforms. In the next chapters we investigate how this idea can be realized efficiently (in particular for FMA).

3 Related Work

3.1 Introduction

Since the first floating-point capable computer was completed by Konrad Zuse in 1938, much has changed in the field of floating-point arithmetic. Zuse's first design, the Z1 [10], was a mechanical system based on sliding metal parts. This electrically driven machine was capable of 22-bit binary floating-point arithmetic (add, subtract, multiply, divide) at a speed of 1 Hz. Its first noteworthy successor, the Z4, was completed shortly after the discovery of digital electronic circuits. Although this meant the transition to semiconductor technology had been made, the machine still had the the dimensions of a cabinet and consumed several kilowatts of power. Due to extensive research and continuous improvements in manufacturing technology, the size of floating-point units (and digital computers in general) has been brought back to the order of square millimeters. At the same time computational performance has increased almost linearly over time.

Another remarkable observation is that over the course of time, floating-point interests have changed considerably. When digital electronic computers first appeared, most effort was put into optimizing performance and throughput of floating-point units. More recently, area and energy efficient solutions are are gaining interest. The cost and area of integrated circuits (ICs) has scaled down far enough to start considering using floating-point units for more area and energy critical applications such as an embedded system. We can see a clear trend in recent publications putting more emphasis on area reduction and minimizing energy consumption. Migration of integer functionality to floating-point units is a concept that perfectly fits this trend. Yet, the research devoted to this idea is still very limited.

In this chapter we first compare three different floating-point units to obtain ample understanding of the basic mechanisms used in contemporary floating-point units. The architectures and concepts of the UltraSparc T2, the Intel Itanium and Cell processors are explored. Their architectures and design principles serve as a base for an efficient floating-point datapath. We also shortly discuss a technique called 'dual path' adders which we decided not to use due to a tight area budget. These adders could be considered for further optimization if the area requirement is reduced. We conclude by a short overview of what has already been done to integrate floating-point and integer arithmetic in a single ALU.

3.2 The UltraSparc T2 Floating-Point Unit

The UltraSparc T2 processor is a multi-core, multi-threaded microprocessor introduced by Sun Microsystems in 2007. Because it is a modern processor that still employs a rather classical approach for floating-point arithmetic, we will shortly discuss its internal architecture to give an idea how many conventional floating-point units are put together. The UltraSparc T2 has eight cores and supports eight threads per core. Each core is equipped with one floating-point unit. This floating-point unit is fully IEEE-754 compliant and implements double and single precision floating-point operations.

A simple overview of the UltraSparc T2 floating-point architecture is shown in Figure 3.1.

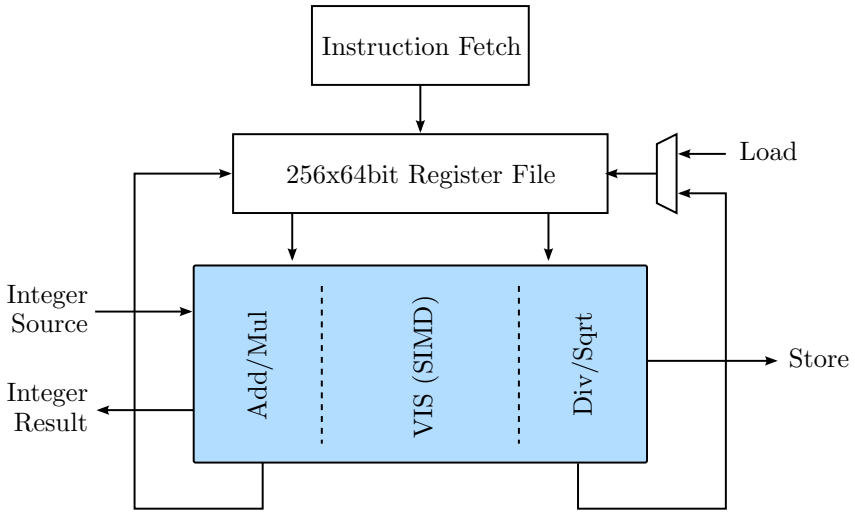


Figure 3.1: UltraSparc T2 floating-point architecture [11]

Most notable about this architecture is the clear distinction of instruction specific datapaths. Addition and multiplication can for example clearly be differentiated from division and square root. To some extend addition/subtraction and multiplication can also be seen as separate datapaths, however in the UltraSparc T2 specifically, they are considered to be merged because they share some common hardware components. The use of dedicated datapaths has been applied by computer architects for many years. A floating-point unit consisted of a datapath for addition and subtraction, a datapath for multiplication, for division and sometimes for square root and/or other instructions. More recently the FMA datapath has appeared (Section 3.3), that is slowly gaining popularity over the classical approach shown here. Despite the fact that addition and multiplication are performed on the same datapath, the UltraSparc T2 does not support multiply-accumulate operations. Some other properties that are characteristic for the UltraSparc T2 floating-point unit [11] are:

- A pipelined design that is focused on area and power reduction
- A partially merged floating-point add/subtract/multiply datapath
- Integer multiplication and division can use the floating-point datapath
- Single and double precision support in hardware
- Clock gated design for energy efficiency

Every floating-point instruction in the UltraSparc T2 is implemented in a pipelined fashion, except for division and square root. A special combinatorial datapath is used for division and square root instructions. Both instruction are non-blocking and have a fixed latency. However, when the datapath is used for integer division the latency is variable.

3.2.1 Unified Addition and Multiplication Datapath (Add/Mul)

The main floating-point execution pipeline (Add/Mul) of the UltraSparc T2 is shown in Figure 3.2. The amount of new terminology introduced can be overwhelming. In Chapter 5 the exact meaning of each individual component will become clear. The purpose of Figure 3.2 is merely to show how the pipeline stages of the UltraSparc's floating-point unit are utilized. The pipeline consists of six stages and is responsible for the execution of addition, subtraction, multiplication and non-arithmetic operations. Because the UltraSparc T2 architecture supports only one instruction issue per clock cycle, there is no need for spatially separated addition and multiplication datapaths. This property is exploited by using parts of the addition/subtraction datapath for multiplication.

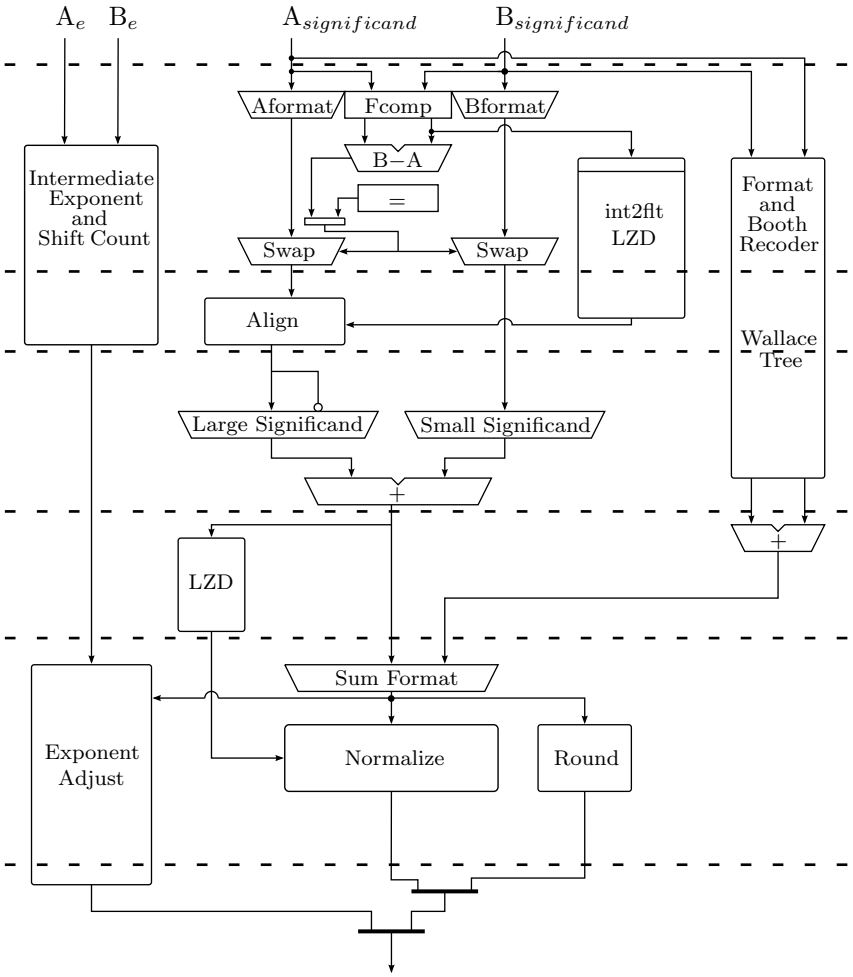


Figure 3.2: UltraSparc T2 addition/multiplication datapath [11]

Stage	Add	Multiply
1	Format input operands	
	Compare significands	Booth encoding
2	Align operands	Generate partial products
	Invert smaller operand if subtract	Reduce partial products (Wallace tree)
3	Computer intermediate result (A+B)	
4	Determine normalization shift amount	Add partial products
5	Normalize and round	
6	Format output	

Addition

The first stage of the pipeline is mainly used to format the operands such that they can be processed (e.g., normalization of denormalized input). Since the exponents must be equal before addition can start, the second stage performs alignment of the significands, such that the exponents match. The UltraSparc T2 always shifts the operand with the smallest exponent. The significands are swapped if needed. The exponent logic that computes the shift count for the shifters is divided over the first two stages. During alignment, guard bits are used to catch bits that are shifted out of range. Sticky-bit logic is used for bits that are entirely discarded. After the operand has been shifted, it is inverted for subtraction.

The actual addition/subtraction is performed in stage three. The main adder in the UltraSparc is a partitioned 64-bit adder (Section 3.2.3) that operates on the unmodified significand of the largest operand and the aligned smallest operand. In case of subtraction, the inverted significand is added. To support both single and double precision in the same datapath, an internal 64-bit representation is used for all data (i.e., single precision is extended to double precision). After subtraction the result is inverted.

Multiplication

Multiplication immediately starts in the first stage of the pipeline. The multiplier in the UltraSparc T2 performs 64x64 bit multiplication. The multiplier is implemented as a combination of carry-save adders (CSAs) in a Wallace tree organization with Booth encoding [12]. This involves two basic operations: the generation of partial products, and the reduction of the partial products. The generation process takes place in the first stage and reduction is distributed over stage two and three. Booth-4 encoding reduces the number of partial products that have to be generated. The Wallace tree configuration of CSAs accumulates the partial products to a 128-bit sum and a 128-bit carry. The sum and carry are added using a regular carry-propagate adder.

Normalization and Rounding

After multiplication/addition, the process of post-normalization and rounding begins. Post-normalization is achieved by shifting the mantissa left while decrementing the exponent until the first bit to the left of the radix point becomes `1`. In the UltraSparc T2, normalization is only required after addition or subtraction. Because the UltraSparc T2 only accepts normalized input, the result of multiplication will be either `01.--` or `10.--`. In these cases, normalization is only a matter of shifting the result one position to the left based on the value of the MSB. For subtraction or addition, the number of positions to be shifted is determined in the leading zero detection (LZD) circuit in stage four. The result is encoded in a 11-bit two's complement control signal for the normalization shifter and exponent datapath.

Non-Arithmetic

Non-artithmetic instructions are implemented on the Add/Mul datapath. The list below show the instruction that are supported and the hardware that is used for the respective operation.

- Compare exponent and significand (comparator)
- Convert from integer (shifter and adder)
- Convert to integer (shifter and adder)
- Convert double to single (adder)
- Convert single to double (adder)

3.2.2 Division and Square Root Datapath (Div/Sqrt)

The division and square root datapath implements both floating-point division and square root, as well as integer division instructions, in a non-pipelined manner. Floating-point operations have a fixed latency, 19 cycles for single precision and 33 for double precision. For integer division the latency varies between 12 and 41 cycles. Additionally, for a number of instructions results can be determined without performing actual calculations. Early completion is allowed to decrease overall latency. Given the fact that eight threads share a single datapath, variable latencies are preferred over a fixed (longest) delay. The division datapath is non-blocking with respect to the other two datapaths.

Even though floating-point division and square root have a dedicated datapath, the unified add/multiply datapath is used to calculate the resulting sign and exponent. Under normal conditions, the unified datapath is used to calculate the intermediate exponent for division or square root as if the instruction was pipelined. The result is stored until the combinatorial datapath has finished, and both results can be combined. Special cases are executed entirely on the unified add/multiply datapath. This occurs on one of the following scenarios:

Division

- Either one or both sources are NaN
- Either one or both sources are infinity
- Either one or both sources are zero
- Either one or both sources are denormalized
- Overflow occurs, this is predetermined by hardware
- Underflow occurs, also predetermined by hardware

Square root

- The input is NaN
- The input is infinity
- The input is zero
- The input is denormalized
- The input is negative

3.2.3 VIS Datapath

The VIS datapath can be used for vectorized instructions (single instruction multiple data (SIMD)). These instructions are mostly used to accelerate graphics (VIS 2.0 pixel formatting instructions). The UltraSparc T2 uses partitioned adders and multipliers to provide the means for vector instructions. Since these are outside the scope of this thesis, we will not go further into details. The VIS datapath is well documented in [11].

3.2.4 Power Management

Power reduction in the UltraSparc T2 is achieved in two ways: clock gating and reduced switching activity. Clock gating is applied to different clock domains. The first domain is the main clock signal. When no instruction is executed in the floating-point datapath, the main clock is disabled to minimize activity. The second and third domains are the add/multiply and devision datapaths. When the floating-point unit performs addition, the division datapath is disabled and vice versa. Switching activity is reduced by keeping large buses stable. For example the 64-bit data-out bus, that is held at a constant value when the floating-point unit does not have to store data.

3.3 The Intel Itanium Floating-Point Architecture

The Itanium architecture (Figure 3.3) is the result of an attempt to change the de facto standard (x86) in microprocessors. Because Moore's law can not be maintained indefinitely, reduced instruction set computer (RISC) processors, such as those based on the well-established x86 architecture, started to show their limitations well over a decade ago. As a result, Intel and HP started a joint effort in 1998 to develop a new processor architecture that could push performance beyond the limits of existing designs. Focusing on parallelism and processing as many bits and instructions per clock cycle as possible, the Itanium aims to replace RISC with very long instruction word (VLIW). Features such as a large set of 128×82-bit registers and the ability to execute multiple instructions per cycle characterize the Itanium architecture. Itanium boasts a particularly powerful floating-point architecture.

The floating-point architecture of the Itanium [13, 14] is based on three objectives. First and foremost, it was designed for high computational performance. This is accomplished by architectural features such as pipelining, instruction parallelism and a particularly large register file. Secondly, high precision was desired. Several high precision floating-point formats and very wide registers to support them provide the means for high precision floating-point arithmetic. The last objective that Itanium wants to achieve is full IEEE-754 compliance.

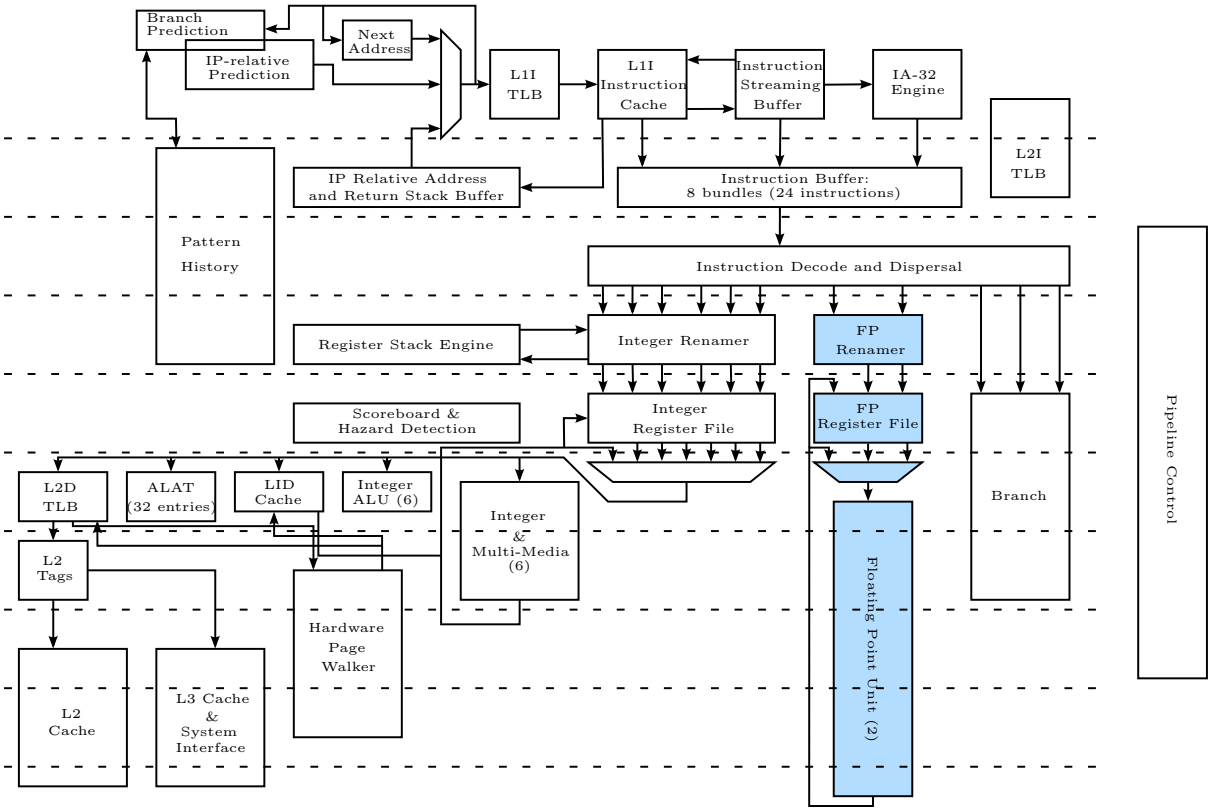


Figure 3.3: Intel Itanium architecture [4]

3.3.1 IEEE-754 Compatible Floating-Point Arithmetic

Itanium floating-point instructions are fully compliant with the IEEE-754 standard. All rounding modes have been implemented, a 64-bit floating-point status register keeps track of all five exceptions and all mandatory and recommended operations have been implemented, either in hardware or software routines.

Format	**Significand** (bits)	**Exponent** (bits)
IA-32 single precision	24	15
IA-32 double precision	53	15
IA-32 double extended precision	64	15
Full register single precision	24	17
Full register double precision	53	17
Full register double-extended precision	64	17
Single precision pair (SIMD)	24	8

Table 3.1: Intel Itanium floating-point formats

Floating-Point Representation

The Itanium architecture supports the standard IEEE-754 single and double precision formats as well as several custom floating-point formats. The IA-32 format (a 15-bit exponent and a 24, 53 or 64-bit significand), full register file format (a 17-bit exponent and a 24, 53 or 64-bit significand) that exploits the full potential of the floating-point registers and a 'single precision pair' format for SIMD instructions (an 8-bit exponent and a 24-bit significand). These formats, listed in Table 3.1, can be used freely by compilers or assembly code writers. The format used for a given computation is determined by the instruction. However, the custom formats can only be used for intermediate computations, they are not available for high level numeric programmers. Only single, double and double-extended formats can be stored in external memory.

Internally, every floating-point number is stored in 82-bit register format: a 64-bit significand and a 17-bit exponent. This is two more bits than minimally required for the highest possible precision. There are several reasons to justify the two additional bits. Among them are the fact that overflow and underflow can occur in intermediate results without the additional bits and the fact that denormalized double-extended numbers require a 17-bit exponent.

3.3.2 Itanium Floating-Point Instructions

All operations from the IEEE-754 standard have a corresponding Itanium floating-point instruction. For example:

$$\texttt{fma.[pc].[sf].f}_1 \texttt{ = f}_3 \texttt{ f}_4 \texttt{ f}_2$$

where

- `fma` is the operation (multiply-add)
- `pc` is for precision control (single, double or other precision formats)
- `sf` is for exception control (status field)
- $\texttt{f}_i$ are the operand/result registers

The operation corresponding to this particular example is $\texttt{f}_1 \leftarrow \texttt{f}_3 \times \texttt{f}_4 + \texttt{f}_2$ (multiply-accumulate or multiply-add when speaking of floating-point operations). The multiply-add instruction is one of the main features of the Itanium. The FMA architectures fuses multiply (A×B) and add (A+C) into a single operation (A×B+C). The advantage is two-fold. Firstly the multiply-add instructions introduce only one rounding error where the result of separate multiplication and addition would introduce two. Secondly performance of multiply-accumulate operations is increased. Since MAC is among the most frequently used instructions, FMA unit can be quite attractive. Increasingly more floating-point units now include a FMA datapath. Sometimes FMA entirely replaces the classic addition and multiplication datapath(s); Itanium is the prime example of this. Multiply-add is basically the only operation that has

been implemented directly in hardware. All other instruction are derived from, or implemented based on multiply-add operations. For example the add instruction:

$$\texttt{fadd.[pc].[sf].f1 = } f_3 f_2 \qquad (A{\times}1{+}C)$$

where register f_4 is replaced by a register with content $(1.0)_d$ [1], and the multiply instruction

$$\texttt{fmpy.[pc].[sf].f1 = } f_3 f_4 \qquad (A{\times}B{+}0)$$

where register f_2 is substituted by a register with content $(0.0)_d$.

Operations that are not direct derivatives of multiply-add (e.g., divide, square root, integer conversion and remainder operations) are handled by software routines that are based on multiplications and additions. Division and square root for example, use the iterative Newton-Raphson approximation method. A detailed described of these algorithms can be found in [14] and [13]. Two special instructions are available to make the first Newton-Raphson approximation. *Floating-point reciprocal approximation* (`fcrpa`), which returns an 8-bit approximation of 1/B, and *floating-point reciprocal square root approximation* (`frsqrta`) which returns an 8-bit approximation of $1/\sqrt{|A|}$. A lookup table is used to find the actual approximations. In special cases where operand A or B is zero, the instruction returns the result immediately and clears a register that serves specifically these situations. In all other cases, both division and square root operations need thirteen instructions to complete.

floating-point comparisons can be performed directly between numbers in register format, using the `fcmp` instruction. Six comparison operations have been implemented directly in hardware.

= (eq)	> (gt)
< (lt)	≥ (ge)
≤ (le)	? (unord) *

* `unord` ("unordered") is a special IEEE relation that is true if one or both operands are NaN

The inverse (*neq, nlt, nle, ngt, nge* and *ord*) of these instructions has been implemented as pseudo-operations. The remaining fourteen comparisons [6] specified in the IEEE standard can be performed as combinations of these primary twelve.

3.3.3 Exception Handling

Itanium operations can signal all IEEE-754 defined exceptions plus a number of additional Intel specific exceptions for denormal operands. Software assisted operations can also be signaled, but will not surface to user level. These exceptions are automatically dealt with by the 'Software Assistance handler'. The status bits in the floating-point status register indicate the status of operations. These bits are *sticky*, meaning that they need to be cleared explicitly after each exception occurrence.

Exceptions are often obstructions for portability because many exceptions are custom defined data structures. The Itanium has adopted the optional IEEE-754 floating-point filter that pre-processes the exception information before it is passed on to the user. This simplifies the task of exception handling because the filter abstracts the user away from interpreting and decoding all system specific information.

[1] Register `f0` is hardwired to +0.0 and `f1` to +1.0 . Both are read-only registers.

3.3.4 Architecture

Unfortunately there is no detailed information available regarding the Itanium's internal floating-point hardware organization. Most likely due to the fact that it is a commercial product. However, from Figure 3.3 we can deduce that the floating-point unit is a five stage pipelined design. Another known fact is that the multiplier and adder are fused and that invariant registers are used to implement the derivatives of the multiply-add instruction. This indicates that the multiplier or the adder are not bypassed for add and multiply instructions, so that the instruction latency for multiplication and addition is just as long as for full multiply-add. The architecture also provides the means for SIMD floating-point instructions, which means that there are multiple floating-point cores present in a single Itanium processor.

3.3.5 Failure to Success

Intel and HP intended the Itanium to set the new standard for microprocessors; first targeting the high-end market and eventually the entire industry. The Itanium is however not as successful as Intel and HP had hoped. This is partially due to the Itanium architecture being so different from the legacy x86 architecture. Most applications that predate the Itanium are not very compatible with its architecture. They would first have be be adopted which requires either very sophisticated compilers or human effort. Such compiler have proven to be notoriously hard to write and as such they do not always result in efficient machine code. Manual modifications are usually too expensive such that Itanium is often ignored.

3.4 The Vector Floating-Point Unit of the Cell Processor

Like the Itanium, Cell is also the result of combined efforts. In the year 2000, Sony, IBM and Toshiba started working on a new processor architecture. Five years later, the first generation Cell processors were commercialized. Cell [15] is a multi-core design (Figure 3.4) that consists of a single Power PC element (PPE) connected to a number of synergistic processing elements (SPEs) [16]. The ideology of the Cell processor is that existing software can run on the more established architecture of the PPE, while computational intensive applications can gradually be migrated to SPEs.

Each of the SPEs contains a floating-point unit. Mueller et al. present the floating-point unit of the first generation Cell processors in [17]. This vector based floating-point unit supports 4-way SIMD single precision instructions and 2-way SIMD double precision instructions. The Cell's floating-point architecture is based on separate single and double precision hardware, rather than a double precision floating-point unit that also supports single precision instructions. The reason for this is that the Cell's major target applications, real-time 3D graphics and multimedia streaming, demand very high single precision throughput. Since a double precision datapath does not meet these requirements, separate datapaths are used: a highly optimized single precision datapath and a more conservative double precision datapath.

3.4.1 Architecture

The floating-point unit of a SPE consists of a 128-bit frontend that provides the input, a 64-bit double precision arithmetic core and four 32-bit single precision cores. The frontend can select from multiple input sources, such as the register file or the result multiplexer. The primary set of operand registers in the frontend feed the single precision cores, while a second set of operand registers is used for both the double precision core and a 'formatter'. The formatter is used for integer multiplication, conversion

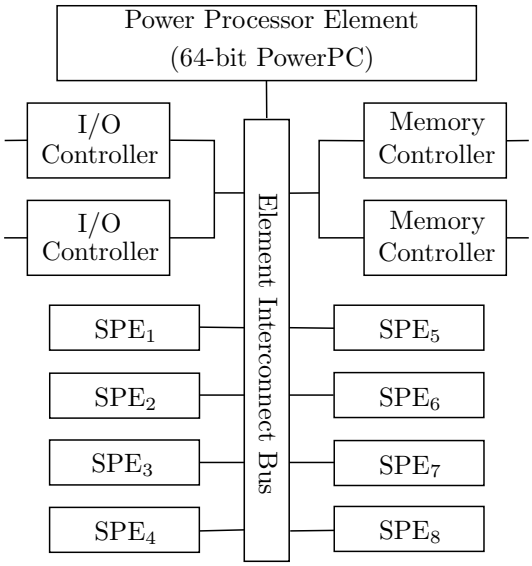


Figure 3.4: Cell processor schematic

and interpolation. Every single precision core is a six stage pipelined FMA design, comparable to the Itanium. The double precision core uses a nine stage pipeline.

3.4.2 Single Precision Floating-Point Core(s)

The class of instructions directly implemented in the single precision core is multiply-add. Division, square-root and conversion between integer and floating-point numbers are implemented in software routines, supported by hardware instructions for 1/A and $1/\sqrt{|A|}$ (the initial estimates). Further refinement of this first approximation is achieved by exploiting the FMA architecture with Newton-Raphson iterations. This shows that the floating-point unit of the Cell processor is quite similar to the one found in the Itanium. Floating-point comparisons are executed on the Cell's fixed-point unit.

Although the Cell's single precision floating-point unit is based on the IEEE-754 standard, it considerably deviates from the standard to improve performance. For example, denormalized numbers are forced to zero and operands with exponent 255_d are not treated as NaN or infinity because the meaning of NaN and infinity is not particularly useful in most of the Cell's target applications. Additionally, only the 'round to zero' rounding mode is supported. This eliminates the need for sticky-bit computations and reduces rounding to simple truncation. The speed up and area reduction of these optimizations is considerable, however they are quite radical.

3.4.3 Double Precision Floating-Point Core

Just like the single precision case, double precision is based on IEEE-754, but not fully compliant with the standard. Cell does not support double precision denormalized numbers or NaN arithmetic. However, in contrast to the single precision case, the double precision floating-point core supports all rounding modes. Therefore, a special converter provides a means of extending single precision to double precision for single precision operations that require rounding modes other than 'round to zero'.

As mentioned earlier, double precision arithmetic is implemented in a 9-stage pipeline. The first three cycles of the pipeline are reserved for the multiplier and alignment shifter. Stage four and five are used

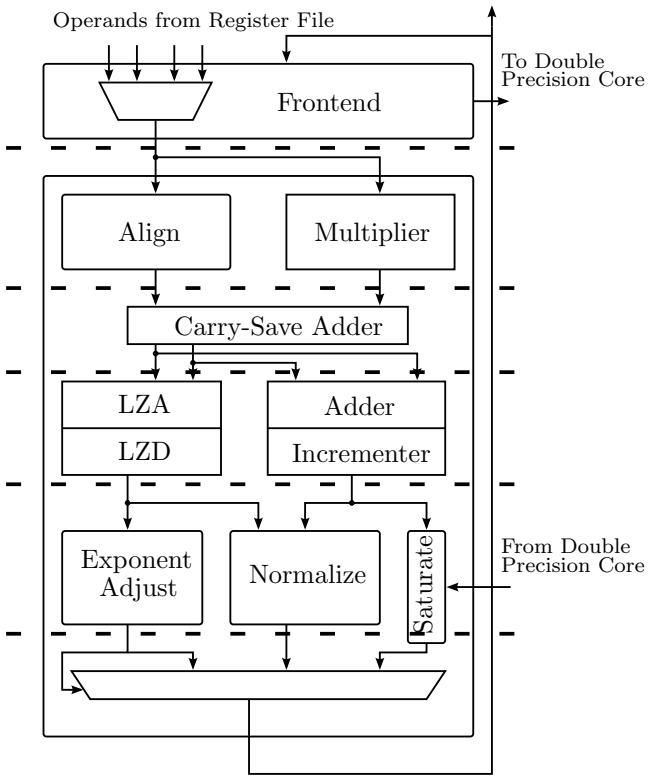


Figure 3.5: Cell single precision floating-point core

for leading zero anticipation (LZA), LZD and addition. Rounding and normalization is accomplished in the remaining four cycles.

3.4.4 Cell's Excellent Performance

The first generation of Cell processors runs at an impressive 3.2GHz, achieving a theoretical throughput of 6.4 GFLOPS per SPE, or 25.6 GFLOPS per Cell processor. This kind of performance required some very aggressive and radical optimizations. The most radical design choice being the removal of rounding. Not only does this save an entire pipeline stage, also the *sticky-bit logic* can be omitted. Another major contribution to the Cell's high performance is the lack of denormalized number support. Every denormalized number is forced to zero instead of being normalized, saving another pipeline stage. The Cell architecture even incorporates special logic to boost adder carry propagation, which demonstrates the amount of effort that was put into achieving this kind of performance. This performance excellence is however not obtained solely at architectural level. A state-of-the-art high performance manufacturing technique was used for realization of the processor and a lot transistor-level optimizations have been applied to achieved an ideal *fan-out*.

3.4.5 Power Management

Since floating-point units are present in all of the Cell's SPEs, their power consumption contributes to the overall power consumption of the chip considerably. To reduce power consumption, *clock gating* is applied. Clock gating is applied at three levels. At the highest level, the entire floating-point unit can be disabled by means of the global clock signal. At pipeline level, the stages that do not contribute

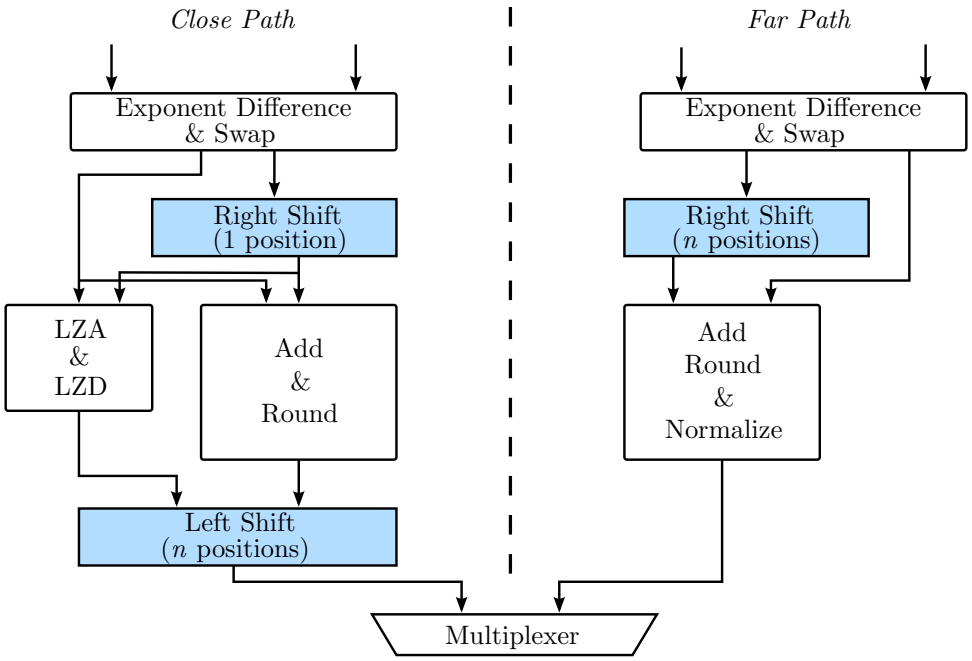


Figure 3.6: Dual path floating-point adder [21]

to the execution of a valid instructions are deactivated. The last level is based on opcode and data-depending clock gating control. Only the components within a pipeline stage that are actually needed for the operation currently executed in that stage are active (e.g., the multiplier is disabled during add operations).

3.5 Dual-Path Adders

We have seen three floating-point units that rely on different strategies. Although the Cell and Itanium are among the most progressive and optimized solutions in existence, even more exotic ideas exist. The 'dual-path' floating-point adder is an example of such concepts. The idea of a 'dual-path' adder stems from the observation that a full-length alignment shift (n-positions) and a full-length normalization shift are mutually exclusive. In other words, if alignment requires a large shift then normalization does not and vice versa. The dual-path adder, first introduced by Farmwald [18], exploits this property by splitting the datapath for addition in two separate paths. The *Close Path* that is chosen when the difference between the input exponents is zero or one, and the *Far Path* that is chosen when the difference is greater than one.

The general structure for such a dual-path significand adder is shown in Figure 3.6. Instead of having two large shifters in the critical path, there is now only one large shifter and a simpler 'shift' to move the significand one position to the left or to the right. A dual-path architecture result in a considerable amount of redundant hardware. However, since *cancellation* never happens in the Far path, leading zero detection is not needed here. The latency reduction that can be achieved by this strategy is worthwhile. Moreover, the dual-path approach has recently been investigated for application in FMA [19]. Although the performance results are promising, the penalty is that the area increases up to approximately 45% [20]. Dual-path addition is not used in the ALU we designed for this thesis. It could however be a useful addition to the architecture if higher throughput is desired and area is on a less tight budget.

3.6 Combining Integer and Floating-Point Arithmetic

The concept of mapping integer functionality to floating-point hardware is not entirely new. In the UltraSparc T2 we have seen that integer multiplication can be performed on the floating-point multiplier and division on the floating-point divider. Similarly the Cell SPEs supports some integer operations by pre-formatting the input in the frontend of the floating-point unit. In both cases, the effect that additional integer functionality has on the floating-point datapath is poorly described. Most architectures capable of performing both integer and floating-point arithmetic on a single datapath are patented and therefore closed for public. However, although being scarce, there is published work that discussed the possibilities and advantages of this concept.

The potential of combined integer/floating-point units is for example recognized by Palacharla and Smith. In [22] they conduct a feasibility research and indicate what integer operation can be migrated to a modified floating-point architecture. Their conclusion is that that approximately 40% of the common integer instructions can be offloaded to a standard floating-point unit. The instructions they find suitable for execution of floating-point hardware include addition, subtraction, shifts and other simple logical operations. Palacharla and Smith do not provide details regarding the modifications needed for augmented integer functionality on a floating-point unit. However, in response to their work, Solihin et al. present an ALU, which they call mutable function unit (MFU), that can switch between integer and floating-point functionality at runtime [23]. The floating-point adder from the MIPS R10000 processor is modified to perform integer operations. Based on the findings of Palacharla and Smith, the MFU is designed to support integer addition, shift and logic operations. In addition address generation functionality is added for memory operations (load/store). The modifications that are described are widening the significand adder and inserting switches to make the hardware reconfigurable. According to the authors, the area remains roughly the same (i.e., the overhead resulting from the modifications to support hardware configuration is neglible). The additional integer performance gain that can be achieved by the MFU (measured by simulation) is reported to be 8% to 14%.

Another reconfigurable hybrid adder, that is almost identical to the one described by Solihin, is presented in [24]. The architecture of this reconfigurable floating-point adder is shown in Figure 3.7. The circles represent programmable switches that determine if the adder is in integer or floating-point mode. The top switch disables the swap unit for integer operations, causing integer `B` to always proceed directly to the barrel shifter which is controlled by a second switch. The bottom switches select between integer and floating-point input for the adder/subtracter. Integer operands are fetched directly from registers while floating-point input first passes through the shifter and swap units.

3.7 Summary

Floating-point units have been around since the introduction of the first semiconductor microprocessors. In the classical situation floating-point arithmetic was spatially separated from integer arithmetic. Even though floating-point units are now integral parts of most contemporary microprocessors, they are still clearly segregated from the main integer ALU. Some research has already been done in the field of combining floating-point and integer arithmetic, the result are however still immature and limited. A modern processors like the Cell offers limited support for execution of integer additions on the floating-point units in the SPEs. Similarly the UltraSparc T2 only allows integer multiplication and division on the floating-point unit. For as far as we can tell, no existing architecture offers full floating-point and integer arithmetic in a single datapath.

In the next chapters a new floating-point architecture is presented that is based on the FMA architecture as seen in the Itanium and Cell processor. Unlike the architectures presented so far, this ALU is designed to get the most integer potential out of floating-point hardware. It offers the same arithmetic functionality for integer as it does for floating-point.

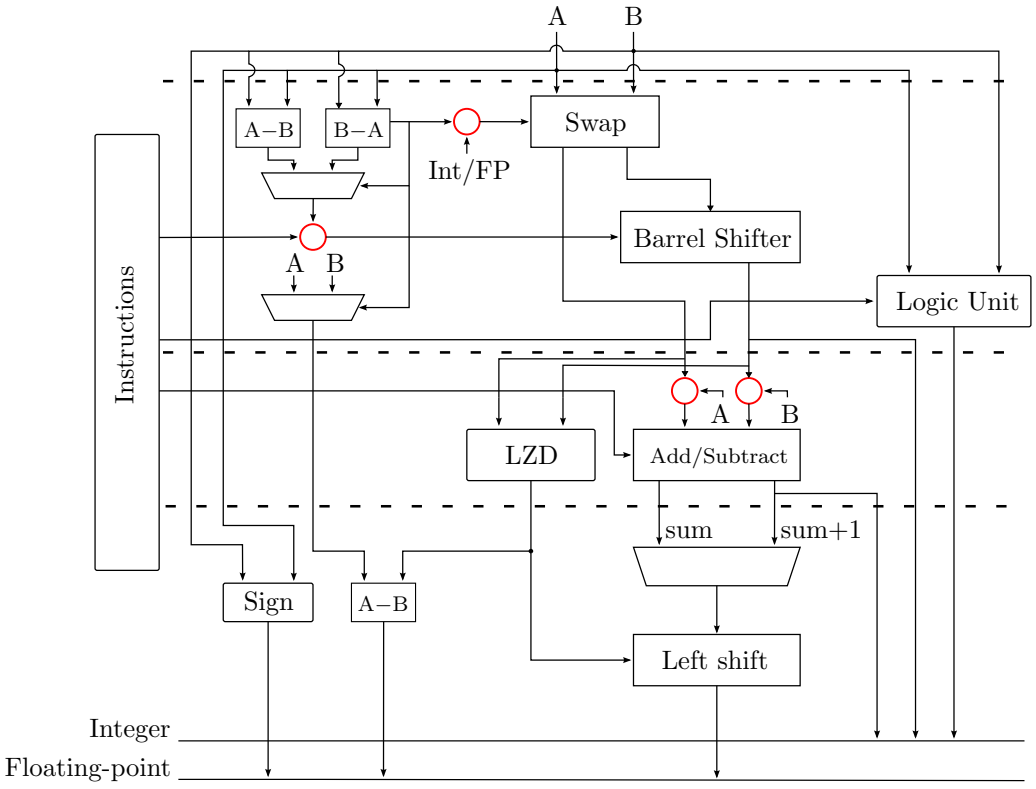


Figure 3.7: Reconfigurable floating-point integer adder [24]

4 A Fused Multiply-Add Floating-Point and Integer Architecture

4.1 Introduction

In the previous chapters we have seen that floating-point arithmetic is much more complex than integer arithmetic. This is directly reflected in the hardware requirements and as such, floating-point processors are rarely seen in low-cost (small area, low power consumption) architectures like embedded microprocessors or microcontrollers. When low-cost systems require fractional arithmetic, designers often choose to sacrifice performance or substitute floating-point with something less computationally intensive.

The most common alternative for floating-point is *fixed-point*. Like the name already suggests, fixed-point has a fixed radix point and lacks a dynamic exponent. These properties make it much less versatile than floating-point numbers, but also much less demanding. In a binary (radix-2) fixed-point notation, the exponent is always 1, hence all numbers are of the form $\langle q.f \rangle \times 2^1$ where q is the integer part and f the fractional part. Suppose we want to represent numbers in a 16-bit fixed-point format. We then have to decide how many bits are used for the integer part and how many for the fraction. The example below shows a fixed-point number with two integer bits and 14 fractional bits.

$$\texttt{01.10000100100100}\ (1.517822265625_d)$$

This example immediately shows the limitations of fixed-point notation. With only two integer bits, the range is limited to $[0, 3)$. If the format would support signed numbers, the range is limited even more. The only way to increase the range is by sacrificing precision and vice versa.

Another alternative for true floating-point is *block floating-point* [25]. Block floating-point can be considered as a hybrid form of fixed-point and floating-point. When a floating-point number becomes too small or too large, the hardware automatically scales up or down by shifting the significand and adjusting the exponent. In block floating-point arithmetic, all numbers share the same exponent at a certain moment in time (i.e., Figure 4.1: there is only one register to store the exponent and its value is used for every significand currently in memory). The programmer decides when to scale up of down the exponent. Therefore, arithmetic such as addition and multiplication is no more complex than fixed-point arithmetic, while a larger 'semi-dynamic' range can be achieved. It is not difficult to see that block floating-point still suffers from some of the shortcomings of fixed-point notation. Although the exponent offers dynamic range, only a small portion of fractional numbers can be expressed at once. Moreover, it is the programmer who has to make sure that the exponent is adjusted before the numbers run out of range, a task that is preferably handled by hardware.

A last option that can be considered is floating-point emulation in software. In Chapter 2 we saw

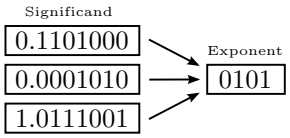


Figure 4.1: Block floating-point

that floating-point operations essentially consist of multiple smaller elementary operations (e.g., shifts, additions and multiplications). In principle these operations can all be executed on regular integer hardware. If we break down a floating-point operation, it is possible to use software routines to implement floating-point arithmetic on basic integer ALUs. For software emulation we do not have to make a trade-off between precision and range, the exact same behavior of a real floating-point unit can be achieved. However, because all elementary operations are performed sequentially, performance will be extremely low. Moreover, many basic operations that are simple to perform in hardware are difficult to implement in software (e.g., computing the sign-bit already requires many bit-masking operations just to obtain the individual sign bits of the input operands). Software implementation should therefore only be considered when floating-point operations are rarely used or when there is absolutely no other provision for fractional arithmetic.

4.2 Approach

A number of alternatives for floating-point arithmetic have been presented. The great majority of digital signal processors, embedded processors and other area/power critical devices use one of the above alternatives for fractional arithmetic. Whether it is performance or expressiveness, they all compromise something to achieve area and power efficiency. Of course they do this with good reasons, floating-point units simply require too much hardware to justify for such devices.

It is exceptionally difficult, if not impossible, to design fast floating-points unit with a small area. Yet, combining high performance and low-cost floating-point arithmetic is a major objective of the work presented here. To achieve this goal we take an entirely different approach than discussed so far. A floating-point unit is rarely used as a stand-alone processor anymore. Especially now multi-core architectures have gained immense popularity, it is much more likely to co-exist next to other, usually simpler, integer based processing cores. Instead of trying to reduce the area of the floating-point unit itself, we accept that it requires a lot of area, focus on multi-core architectures (in particular power lean MPSoCs) and try to achieve area reduction by mapping the functionality of integer cores onto the larger floating-point core so some of the integer specific hardware is no longer needed. The hardware of a floating-point unit shares great similarities with its integer counterpart. Although not trivial, it is possible to perform integer operations on floating-point hardware. If all essential integer operation can be mapped to a floating-point unit, there is less need for separate integer processors. A floating-point unit with augmented integer functionality replaces the dedicated floating-point unit and integer cores (Figure 4.2). This should make the larger area of a floating-point unit justifiable for low-cost systems.

4.2.1 Design Methodology

An integer/floating-point unit (I/FPU) was designed as part of this thesis. Although hardware design can still be performed by hand, logic synthesis is preferred, especially for larger more complex systems like a floating-point unit. Logic synthesis is a practice where a desired circuit behavior is described in a hardware description language (e.g., VHDL or Verilog) and transformed to a *netlist* of logic gates. The translation from HDL to logic gates is automated, usually backed up by sophisticated optimization

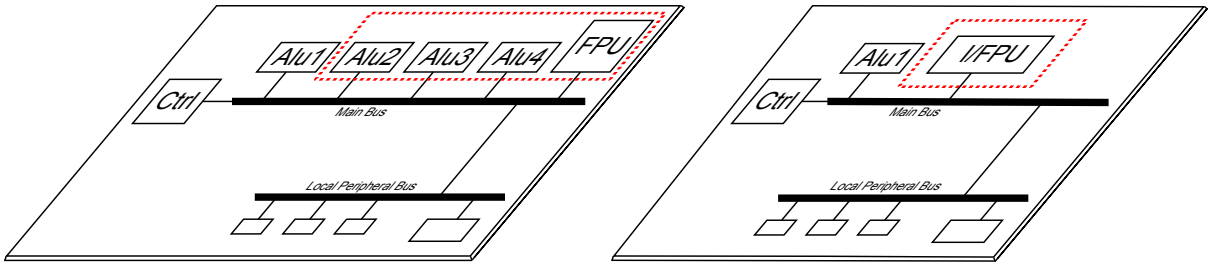


Figure 4.2: Typical multi core SoCs whith dedicated (left) and shared (right) integer/floating-point units

algorithms. This relieves the designer from the most tedious work involved in designing complex digital systems. Logic synthesis has advanced to the point where most logic synthesis results surpass human design efforts. Hence, the architecture that we will present in this thesis was designed by means of logic synthesis. This is reflected in the level of detail we present the proposed ALU. We will mainly focus on the architectural and logical levels, much less on physical aspects.

Logic synthesis is parameterized. The algorithms can optimize a design for speed, area and power consumption. The target technology may also differ. Some synthesis tools translate to field programmable gate arrays (FPGAs) (for fast prototyping) and others to application specific integrated circuits (ASICs). In this thesis we are mostly interested in ASIC technology. To be more specific, we want a power lean, area-efficient and fast ASIC device. A very effective solution to achieve energy efficiency with logic synthesis is to use low-power technology libraries. Synthesis tools require technology libraries that contain technology specific information about timing and area of the logic cells that are used to form the netlist. More details regarding low-power technology will be elaborated in Chapter 7. For now one specific aspect is of importance: low-power libraries result in slow hardware. The delay of low-power technology is averagely twice as large as a general purpose technology. This aspect greatly influences the design choices we have made.

A lot of performance is lost by using low-power technology. To regain some of this performance, architectural optimizations need to be applied to the datapath. Especially for floating-point arithmetic, a lot of effort is required to achieve high performance in low-power technology. For this reason we initially focus on high performance (Chapter 5). The resulting datapath is then analyzed and modified to support integer operations (Chapter 6).

4.3 Floating-Point Integer Arithmetic Logic Datapath

Before we start describing the microarchitecture of the ALU, two fundamental design choices need to be settled first. A number representation and a instruction set architecture (ISA) have to be chosen. As discussed in Chapter 2, numbers can be represented in various ways. This new type of ALU must support two different kinds of numbers: integers and floating-point numbers. Because the same hardware must process both kinds, their respective representations have to be matched as close to each other as possible. The ISA defines what instructions a processor facilitates.

4.3.1 Number Representation

Let us first focus on floating-point numbers. Since 1985, the most obvious choice for floating-point numbers is the format described in the IEEE-754 standard [6]. It may not always be the most efficient format for implementation, but especially in case of floating-point numbers, standardization is an

important aspect in the representation. In Chapter 2 we already discussed that the standard for floating-point arithmetic defines multiple binary and decimal floating-point formats. There are several predefined formats such as half, single, double, quadruple precision, but also custom defined precision is allowed (Table 2.3). The choice for a certain format strongly depends on the class of algorithms targeted by the architecture. At this point no such set has been defined because we would like to target as many applications as possible. Single and double precision are the most common formats implemented in hardware. Half precision is also gaining popularity for graphics processing units (GPUs), but 16 bits is very small in terms of floating-point. Double precision on the other hand is a massive 64 bit floating-point format. Implementing a double precision high performance floating-point unit is costly, especially multiplication requires a lot of hardware. At first glance single precision seems appropriate for high performance low-cost floating-point solutions. However, we should also consider that the hardware will be used for integer operations.

For integer representation, we usually stick to a word length that is a power of two. Most programming languages define integer types like *short* (16 bits), *int* (32 bits) and *long* (64 bits). Not entirely coincidentally we mentioned single precision as an appropriate candidate for floating-point number representation. It is exactly 32 bit wide which maps nicely to 32-bit integer numbers. This eliminates the need for special floating-point registers. The encoding of integer numbers is mostly done in two's complement because the hardware for arithmetic in two's complement is much simpler than for sign magnitude. However, because we propose an ALU that uses sign magnitude representation for floating-point numbers, a sign magnitude representation for integers is also a plausible alternative. Nevertheless, two's complement is chosen in favor of sign magnitude because it is such a widely adopted notation. Almost all contemporary processors use two's complement encoding for integer arithmetic. Sign magnitude would make information exchange with external devices much harder.

So far we have determined that single precision floating-point and 32 bits integers can be stored efficiently in the same memory, which makes them suitable candidates for an ALU that combines integer and floating-point processing. In terms of arithmetic however, both formats do not map so nicely. The significand of single precision floating-point is only 23 bits wide while we want to have 32-bit hardware for integer operation. There are three choices to deal with this difference: The integer numbers can be reduced from 32 bits to 23 bits; the significand of the floating-point numbers can be extended to 32 bits; we can support both 32 bit integers and 23 bits significand and end up with an irregular ALU. Irregularity is something that should be avoided. Moreover, the hardware is not used efficiently (enough) when 23-bit computations are performed on 32-bit components. On the other hand, 23-bit integer operands are unaccustomed and might discourage programmers to use the proposed platform. For these reasons the significand of the floating-point format is extended to 32 bits. The consequence is that a floating-point number can not be stored in a 32 bit register anymore. Each floating-point number will require two registers. One for the significand and another for the exponent and sign-bit. The latter is of course not very elegant[1], but what we have gained is eight bits more floating-point precision and, more importantly, the possibility to design a nice and regular datapath that can be used for both integer and floating-point arithmetic.

The proposed floating-point format is no longer entirely compliant with the IEEE-754 standard. However, this custom format has some benefits over standard single precision and for practical purposes the differences are very small. A preliminary study (Appendix A) was done to determine what the benefits of eight bits more precision mean for floating-point arithmetic. The conclusion that can be drawn from these early simulations, is that in terms of quantization noise effects, eight bits more precision are certainly beneficial for some of the most common DSP algorithms (33% more signal to quantization noise ratio (SQNR)). The biggest drawback of a 32-bit significand is the separate storage of exponent and sign-bit. However, this disadvantage also provides oppurtunities for future research. For example, compression techniques that exploit exponent sharing, similar to block floating-point but at a more fine grained level. Or flexible architectures that can easily be remanufactured to accomodate larger exponents

[1] There is no way to completely hide the mismatch because floating-point numbers have a separate exponent and integers do not.

in case more range is required.

To summarize number representation, the proposed ALU supports the following input:

41-bit Floating-Point

Width	41 bits
Significand	32 bits
Exponent	8 bits (bias-127)
Encoding	IEEE-754
Decimal Range	$-(2-2^{-32}) \times 2^{127}$ **to** -2^{-126} **and** 2^{-126} **to** $(2-2^{-32}) \times 2^{127}$

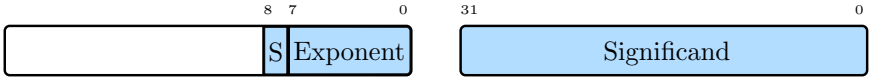


Figure 4.3: Floating-point format

32-bit Integer

Width	32 bits
Encoding	two's complement
Decimal Range	-2^{31} **to** $2^{31}-1$

Figure 4.4: Integer format

IEEE-754 Adherence

As mentioned before, the floating-point format from Figure 4.3 is not a valid IEEE-754 *interchange format*. Recall from Chapter 2 that the IEEE-754 standard for floating-point arithmetic specifies more than just the format of floating-point numbers. Also rounding modes, operations and exception handling have to meet certain standards in order to comply with IEEE-754. The intention of the research in this thesis is not to design a floating-point unit that is completely in agreement with the IEEE standard. We merely pursue an arithmetic core that saves hardware by mapping integer functionality to a floating-point datapath. The IEEE-754 standard is only a tool we use for sound floating-point arithmetic. Indeed, there are several more areas where the arithmetic of our proposed solution differers from what IEEE defines, to simplify the datapath:

- *NaN not supported*
- *Denormalized numbers not supported*
- *Round to nearest, ties away from zero not supported*
- *Only a subset of (arithmetic) operations is supported (in hardware)*[2]
- *Invalid operation, Division by zero and Inexact exceptions not supported*

IEEE-754 defines unique representations for special numbers such as infinity, zero and NaN. Because NaN only occurs during division, square root or other higher order functions, it was decided not to include explicit NaN recognition. This greatly simplifies the datapath because we do not have to worry about incompatible arithmetic. A similar argument hold for denormalized numbers. Although denormalized numbers fill the gap between zero and the smallest normalized number, the practical use for such small

[2]These do provide enough functionality to implement the remaining mandatory operations in software.

differences is hard to justify for low-cost systems because of the additional normalization hardware required. Hence, only normalized input is supported.

The latest IEEE-754 standard (2008) defines two round to nearest modes. *Round to nearest, ties to even* and *Round to nearest, ties away from zero*. The last rounding mode is not included because the additional value over Round to nearest, ties to even is hardly worthwhile the additional complexity.

The hardware only supports basic arithmetic operation such as add, subtract, multiply and multiply-add. The logical operation only include greater than, smaller than and equal to. Many more are described in the standard (e.g., conversions, scaling, testing and classification). Although these are not supported by the hardware directly, they can be added with software routines. This is not directly in violation with the standard, the functionality can still be added with software emulation. We mention it because some functionality is commonly implemented in hardware.

From the five exceptions defined by the standard (invalid operations, division by zero, overflow, underflow and inexactness), only two are supported: overflow and underflow. For basic arithmetic these are the most useful. Invalid operations are not an issue for FMA and derivatives, because NaN results do not occur and neither does division by zero. Inexactness is not signaled because almost all results are inexact. The practical use of this status information is small enough.

Despite the above mentioned deviations from IEEE-754, the proposed floating-point arithmetic is numerically sound. It results in a faster and smaller floating-point datapath that allows easier integration of integer arithmetic. The downside is that we give up some of the formality of the IEEE-754 standard. However, these shortcomings have no major impact on most practical applications.

4.3.2 Instruction Set Architecture

Formally the instruction set architecture, or instruction set, of a processor includes much more than just the set of instructions implemented by the microarchitecture. The instruction set defines native data types (e.g., floating-point, fixed-point or integer), instruction and data registers, memory architecture and addressing, interrupt handling, I/O and much more. However, in this thesis we are mostly interested in the concept of a sharing a datapath between integer and floating-point data. We therefore limit our discussion here to instruction support of the ALU.

Instruction sets can be categorized by the goal they try to achieve. Some examples of well known instruction sets are the complex instruction set computer (CISC), RISC, VLIW, the Itanium's explicitly parallel instruction computing (EPIC) and vector instruction sets such as SIMD. The goal that RISC pursues is simplicity. By implementing very simple instructions, high performance is obtained in RISC architectures. CISC and VLIW try to do the exact opposite of RISC. A CISC architecture implements more powerful instructions that can perform several simple operations (load, store and arithmetic) within one instruction. EPIC and SIMD try to exploit parallelism. Operations like complex multiplication (that involve four multiplications and two additions and a subtraction) lend themselves excellently for parallel and vectorized instruction sets. The current trend in microprocessor instruction sets is RISC. RISC is also what we propose for combined integer and floating-point arithmetic. By implementing simple instructions that perform common arithmetic/logic operations in hardware, we can achieve high performance with a relatively small area. The more complex operations that are not performed as often, can be implemented in software routines.

The most basic and common arithmetic operations are addition/subtraction and multiplication. Both are implemented natively in hardware for floating-point and integers to ensure maximum performance. Another interesting arithmetic operation is multiply accumulate (multiply-add). Many applications benefit from this powerful operation because is eliminates load and store overhead of two separate instructions. In addition, the operation improves the quality of floating-point operations because only one round-off error is made. Another common, but less often used, arithmetic operation is division.

Instruction	Input	Operation
Multiply-Add	A, B, C: Floating-point/Integer (signed)	$A \times B + C$
Multiply	A, B: Floating-point/Integer (signed)	$A \times B$
Add	A, C: Floating-point/Integer (signed)	$A + C$
Compare less than	A, B: Floating-point/Integer (signed)	$A < B$
Compare grater than	A, B: Floating-point/Integer (signed)	$A > B$
Compare equal	A, B: Floating-point/Integer (signed)	$A = B$
Shift right	A, n: Integer (signed)	$A \ll n$
Shift left	A, n: Integer (signed)	$A \gg n$

Table 4.1: Instruction set

Because a division operation can efficiently be implemented on multiply-accumulate hardware [26], it is not natively supported. The same argument holds for even more complex arithmetic such as the square root and sine/cosine functions. The only arithmetic operations that we natively support in the instruction set are multiplication, addition, subtraction and multiply-accumulate.

For logic operations the same RISC approach is taken. Only the most basic compare instructions are implemented directly in hardware. Two register values can be compared with one of the following operators: *less than* ($<$), *greater than* ($>$) or *equal* ($=$). Any other compare operation (e.g., $\leqslant$) can be derived. Bitwise operations are not implemented, except for the more common integer operations *shift left* ($\ll$) and *shift right* ($\gg$). Looking ahead to Figure 4.11 we can see that a floating-point datapath includes both a shift left for normalization and a shift right for alignment. Due to the availability of hardware we have decided to include shift left and right in the instruction set. Both shifting operations are arithmetic shifts (Figure 4.5(a) and 4.5(b)) that can shift a two's complement number n positions to the left or right. As opposed to logical shifting, an arithmetic shift right operation duplicates the sign-bit. For two's complement encoding this is much more useful than logical shifting because all numbers are sign extended. The sign-bit will not flip due to shift operations. Chapter 6 explains how arithmetic shifters can be used for normalized floating-point input as well.

(a) Shift right (b) Shift left

Figure 4.5: Arithmetic shifts ($n = 1$)

Data handling and control (branching) instruction are outside the scope of this thesis. We strictly focus on the arithmetic and logic within the ALU. An extensive instruction set specification can be found in Appendix C. Table 4.1 provides a less detailed overview of the same specification.

4.3.3 Input/Output

We are now ready to specify a datapath that implements the functionality of Table 4.1 based on the number representation of Figure 4.3 and Figure 4.4. The first thing that will be settled is I/O. Multiply-add instructions require a maximum number of three operands. The three operands can be 32-bit integer numbers or 41-bit floating-point numbers. In the previous section we already mentioned that we wanted to store the 41-bit floating-point numbers divided over two 32-bit registers such that the same local registers can accommodate both floating-point and integer data. Input and output is based on this

concept which yields a 64-bit based I/O system. The I/O consists of three 64-bit data input buses, a smaller 5-bit instruction bus for instruction selection and control, a 64-bit data output bus for results and a 3-bit status bus to indicate the status of the output (e.g., overflow and underflow).

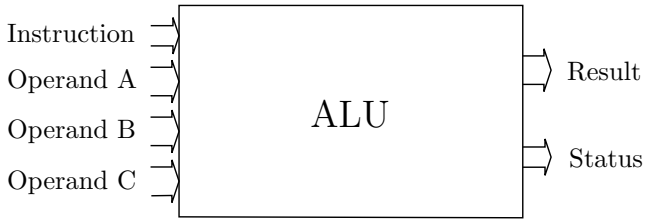


Figure 4.6: Datapath I/O model

Because formatting and requirements for I/O differ per instruction, a mapping is defined.

Bus Level Mapping

Arithmetic instructions includes addition, subtraction, multiplication and multiply accumulate. For multiplication the two primary operand inputs A and B are used. For multiply accumulate, A and B are used for the product (A×B) and C for the addend. Addition and subtraction both use input A and C for implementation convenience.

All of the floating-point and integer compares use input A and B. The shift instructions only use input C.

Bit Level Mapping

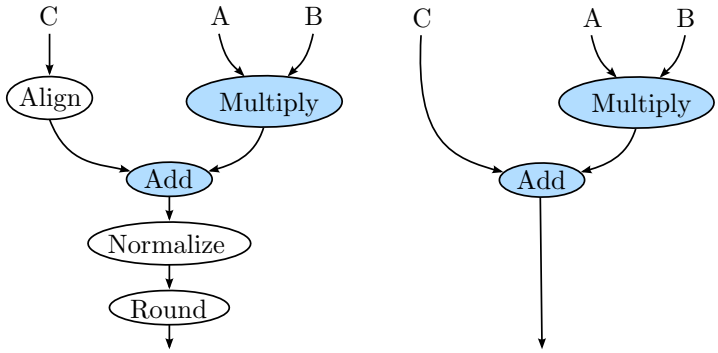
When the ALU reads data from the input, it does not read all 64 bits. For floating-point data two input registers are read per operand. The content of the significand register is mapped to the 32 LSBs of the input. The content of the exponent register to the 32 MSBs of the input. From the 32 MSBs, only nine are actually used. Bit eight is the sign-bit and bits 7..0 accommodate the biased exponent, the other bits are meaningless and will be ignored by the datapath. This mapping corresponds directly with Figure 4.3.

Integer data is read from the 32 LSBs of each input. For arithmetic and compare instructions, the MSBs are ignored. For both shift instructions, we use some of the MSB for the shift amount. The maximum shift amount can be encoded in seven bits[3], hence only the seven lower bits of the MSB are used. They are read as a 7-bit unsigned integer, the direction of the shift is determined by the instruction *opcode*.

Floating-point results are mapped to the output in a similar way as they are read from the input. The significand can be found on the 32 LSBs, the exponent and sign-bit in the MSBs. The status bits indicate whether the result overflows or underflows. If neither is the case, then a default value will be found on the status bus. For logical compares, the status bits are the only valid bits. They should indicate either true or false.

Integer results are a little less straightforward. Although integer input is limited to 32 bits, the output can be up to 64 bits. Integer arithmetic always produces a 64-bit sign extended result. Because 32-bit integer arithmetic can never overflow 64 bits, the status bits only indicate if the result exceeds the 32 bits of the first register result. If this is the case then the lower order bits are mapped to the LSBs and the higher order bits to the MSBs. The result of shift instructions maps to the LSBs and for logical compares only the status bits are valid.

[3]Integer shifts can actually be encoded with even less bits, the shifts for floating-point require seven bits (Chapter 5)



(a) Floating-point multiply-add (b) Integer multiply-accumulate

Figure 4.7: Floating-point and integer multiply-add comparison

4.3.4 Dataflow

The dataflow is entirely based on fused multiply-add. Traditional floating-point datapaths spatially separated each operation. A multiplication was performed on multiplication hardware, additions on addition/subtraction hardware and division on division hardware. Such floating-point units can actually be thought of as a collection of datapaths, one for each operation. When the first FMA unit was produced [27], it was only complementary to the primary hardware for multiplication and addition. A more recent development is that the FMA unit entirely replaces the per operation structure. Good examples of this approach are the SPE of the Cell processor [17] and the Itanium [13] architecture that were discussed in Chapter 3. Any combination of multiplication and addition/subtraction can easily be derived from FMA. Table 4.2 shows how the primary functionality is derived. By using the constants 1 and 0, multiplication, addition and subtraction are easily realized on FMA units.

Input			Effective Operation
A	B	C	A×B+C
A	B	−C	A×B−C
A	1	C	A+C
A	1	−C	A−C
A	B	0	A×B

Table 4.2: Functional derivatives of fused multiply-add

The same technique can be applied to a integer MAC unit. The difference between floating-point FMA and integer MAC units is not that large. FMA units first have to align the addend and the result needs to be normalized and rounded. As shown in Figure 4.7, MAC can easily be mapped to FMA. All it requires is bypassing some of the floating-point specific components. Of course some minor modifications have to be made to the basic components in order to support both two's complement and sign magnitude, however the basic structure provides excellent opportunities to share the hardware.

Figure 4.7(a) already provides a good impression of an actual datapath. What misses is functionality to compare two operands and functionality to check for exceptions that might occur during arithmetic. If this functionality is added, we have a datapath that meets the requirements of the instruction set we composed earlier. Figure 4.8 shows such a datapath, although still in a highly simplified manner.

In the remainder of this chapter a more precise definition of this datapath is given by looking deeper into its functionality. The majority of the depicted functional units can not be found off-the-shelf because they are too complex and target very specific goals. Their functionality will be analyzed to

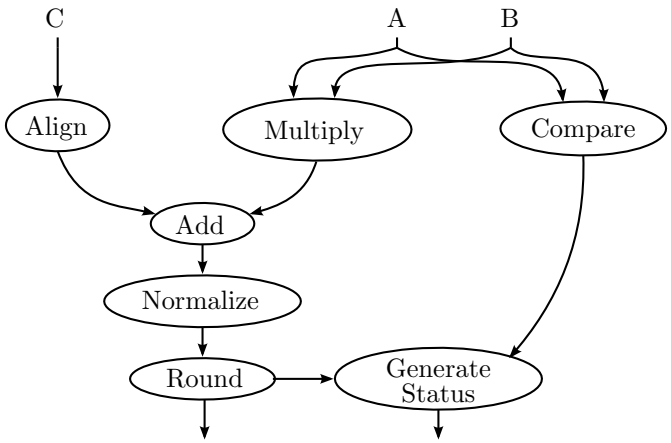


Figure 4.8: Simplified dataflow

find more elementary operations that describe the same behavior such that we can implement it with existing hardware components and find similarities between the integer and floating-point case to create opportunities to share as much hardware as possible.

Align

Alignment is a floating-point specific part of the datapath. In Chapter 2 we already discussed the alignment of two operands for addition and subtraction. For multiply-add, the C operand also needs to be aligned to the $A \times B$ product before it can be added. From Chapter 2 we know that to align the exponents, the absolute difference δ must be found by subtracting the exponents. For FMA, the exponent of A and B must first be added causing the bias to accumulate. To correct this, the bias must be subtracted again. Once δ is found, the significand of the *smallest* operand is shifted δ positions to the right and its exponent increased by δ.

Alignment needs adders, subtracters, a comparator and a logical (right) shifter. It is rather straightforward to implement the alignment, a circuit such a Figure 4.9 can be used for alignment. However, for FMA it is costly to shift the smallest operand. Before this operand can be designated, the multiplication of A and B must be completed. This results in a massive delay. There are ways to carry out alignment more efficiently for multiply-add. The basic idea is to always shift the C operand which is immediately available at the start of the operation. This involves some mathematical manipulation which we will explain in the next chapter. Here we assume that the operands can be aligned by always shifting the C operand. The hardware that is required for the exponent adjustment hardly has any purpose for integer operands. The shifter on the other hand, can be used for arithmetic right-shift instructions. This requires a small modification in the shifter because floating-point shifts should be a logical shift and integer shifts should be arithmetic shifts. The input of the shifter is extended by one bit. For integer operand the sign-bit is copied (sign extended) while for floating-point input this position is always asserted to `0`.

Multiply

Multiplication of two numbers is a well-known arithmetic operation. Together with addition, multiplication is the main resource to optimize for utilization in both integer and floating-point instructions. Many highly optimized multiplier designs exist that can be used to facilitate the multiplication of A and B. The challenge here is that multiplication of floating-point significand is unsigned (only the magnitude is multiplied) while the integer numbers are two's complement signed. In addition, the multiplication

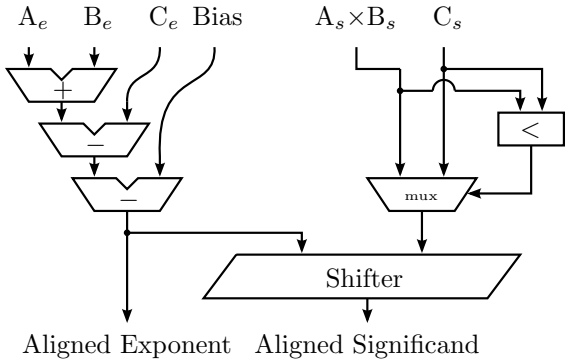


Figure 4.9: FMA operand alignment

needs to be performed fast for high performance. Fast multipliers are trees or arrays of adders. They are large components making them high priority candidates for area optimization. The next chapters will explain in detail how a multiplier can be designed to support both signed (two's complement) and unsigned multiplication with minimal area overhead and maximum performance.

Add

Addition is, just like multiplication, a very basic operation. Although we have called the operation Add so far, we actually want a component that can both add and subtract. In most ALUs, addition and subtraction is performed in two's complement notation. In two's complement, subtraction works just like addition if the carry-out is ignored. Regular adders can be used to implement both addition and subtraction. Adders can also be modified to perform addition and subtraction in two's complement on unsigned input. They conditionally complement their input (either by multiplexed inverters or XOR gates) and feed `1` to the carry-in of the adder such that the input is converted to two's complement. For FMA this is a feasible solution. However, an even more elegant solution exist that is called *end-around carry addition*. This form of addition requires only the C operand to be conditionally complemented and re-complemented. A more thorough analysis of end-around carry addition is given in Chapter 5. The carry propagation of adders is usually among the largest in arithmetic circuits. A vast amount of optimizations for latency reduction exist. In the subsequent chapter we will explain in detail how the best results for floating-point addition is obtained. In the chapter thereafter we look at the modification required to make the adder execute both integer and floating-point additions/subtractions.

Normalize

To normalize a floating-point number, its leading zeros must be removed and the exponent must be adjusted accordingly. To remove leading zeros from the significand, we can keep shifting it to the left until the MSB is a `1`. Shifting one position at a time is time consuming and highly irregular due to the number of leading zeros varying per operation. It is therefore more common to count the number of leading zeros and use that number to drive a single cycle shifter. Counting leading zeros is less trivial than it may seem. Not only do the zeros have to be detected, they also have to be encoded to a binary number. There are basically two distinct methods to count zeros, one method creates a monotonic string of zeros followed by ones and uses a special manchester carry chain adder [28], the other uses a hierarchical tree structure to directly encode the zeros to a binary number [1]. Both methods have their advantages and disadvantages as discussed by Schmookler and Nowka [29]. The datapath we propose incorporates a tree structure because they show an energy efficiency advantage over monotonic string

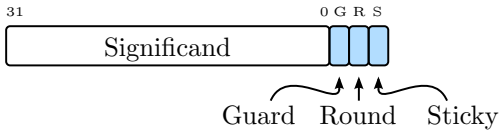


Figure 4.10: Guard, Round and Sticky-bit positioning

production. Normalization suffers from a similar problem as alignment. Before the number of zeros can be detected, the results from the adder have to be know. To improve performance, the LZD circuit can be accompanied by a LZA component. LZA (Section 5.2) predicts the number of leading zeros based on the input of the adder. The number of leading zeros can be detected based on this prediction allowing us to count the leading zeros in parallel with addition. The actual shift and corresponding exponent adjustment can be implemented on conventional hardware. Just like the alignment shifter can be used for arithmetic right-shift instructions, the normalization shifter can be used for arithmetic left shift instructions.

Round

Rounding normalized results is finding the closest representation to the exact result. Because most decimal numbers can not be represented exactly in decimal floating-point format, we have to settle for the closest possible representation. This can be either the normalized result itself (truncated to n-bits) or the incremented normalized result. The rounded result is easily obtained by 2-to-1 multiplexing the result of an incrementer (adder with carry-in of `1`) and the result directly obtained from normalization. The selection is however a more complicated process. IEEE-754 stipulates that rounding is done with infinite precision. That means the result of rounding n-bit floating-point numbers should provide exactly the same result as rounding the same floating-point number with infinite bits of precision. Proof exists that such results can be obtained by using *guard*, *round* and *sticky* bits [9], positioned as shown in Figure 4.10. The guard and round-bits are merely a two bit extension of the datapath during arithmetic. The sticky-bit is defined as the logical OR of all the bits that are lost during execution. If a `1` is shifted out of range during alignment, the sticky-bit becomes `1`, otherwise is remains `0`. Because the alignment shift is not performed bit after bit, special sticky-bit logic is required to find the logical OR of the bits lost during alignment. Once the guard, round and sticky bits are known, a rounding selection can be made relatively easy by pattern matching. The patterns are well documented, for example in [30] or [31].

Compare

The ALU must also provide the means to compare two register values of the same type (i.e., it is only possible to compare two integer or two floating-point numbers, not a combination of both). Comparing two floating-point numbers hardly shares any resemblance with comparing two integers in two's complement. This is unfortunate because we prefer to use only one comparator for area efficiency reasons. To compare two floating-point numbers in the format presented in Figure 4.3, one 8-bit unsigned comparator is needed for the exponents and and one unsigned 32-bit comparator for the significands. In addition the sign-bit has to be examined separately. Comparing two integer operands also requires a 32-bit comparator, however this one should be two's complement. Using a two's complement comparator for significands will not produce correct results for floating-point numbers because the significand's MSB does not represent a sign-bit like the MSB in two's complement. The other way around yields even more complications. We solve the problem by extending the input of the comparator by one bit like we did for the shifters. For a floating-point significand this bit will always be `0`, all integer input is sign extended.

Generate Status

As described in Section 4.3.3 the status bits of the ALU are shared between the outcome of logical compare instructions and the status of arithmetic instructions. Producing the correct status still requires a substantial amount of logic. First the status must be multiplexed between the two different instruction types, and secondly the status of the arithmetic results must be generated. Multiplexing based on instruction selection is easy and straightforward. We regard this as control which will be discussed shortly. Generating correct arithmetic status is done by checking the result for overflow and underflow. Officially IEEE-754 requires that NaN and inexact exceptions are also flagged but we do not support this for practical reasons (NaN does not occur in multiply accumulate and inexactness is true for almost any computation) and to increase performance. A number of simple (not full) comparators is used to check the results in order to generate a correct status.

Putting Things Together

Now that we have given a brief introduction to our proposed architecture and some of the basic principles and opportunities of sharing integer and floating-point arithmetic/logic, we can finally present a first actual datapath. Figure 4.11 depicts the interconnection of the most prominent components discussed so far. In the top left we see the exponent adjustment block and right shifter. This logic implements the alignment of floating-point operands and arithmetic right shifts of integers, the shifter is disabled for integer arithmetic. To the right of the shifter, sticky-bit logic detects if precision is shifted out of range during alignment by logical OR'ing all bits beyond the LSB of the shifter output. The sticky-bit is used for rounding normalized results.

A fast multiplier and adder/subtracter are used to perform the actual multiply-add operation and its derivatives: multiply, add and subtract. Both components are designed to accept two's complement signed integer input as well as unsigned floating-point significand magnitudes. The result of the adder is passed on to a left shifter for normalization. The same shifter can also be used for shifting integer operands. The shifter needs no modifications because the entire normalize/round blocks are skipped during integer arithmetic and logical left shift is exactly the same as arithmetic left shift. To detect the number of leading zeros for normalization, LZD logic is placed between the adder and shifter. The shown datapath can be sped up by including LZA logic. Figure 4.8 does not show this optimization (yet). To round the normalized result, an incrementer is included. The normalized result and the incremented normalized result always include the closest to exact representation. A section is made based on the sticky-bit, a guard-bit and a round-bit. The two latter are implemented by expanding the arithmetic datapath by two bits.

A two's complement comparator is modified by sign extending the input to provide the functionality of comparing both integer and floating-point compare instructions. The result is multiplexed with the arithmetic status that is obtained by a collection of small comparators in the final stage of the datapath. These comparators check for overflow and underflow. There is clearly still a strong resemblance between the hardware components in Figure 4.11 and the more mathematical view of the datapath in Figure 4.8. There is also still a lot of detail missing. In the upcoming chapters this datapath is further optimized and more detail is added.

4.3.5 Pipelining

The datapath of Figure 4.11 is an enormous combinatorial circuit. The critical path of this circuit runs all the way from the input registers to the output registers, passing several large component such as the multiplier, the adder, the normalization shifter and rounding logic. The *critical path* of a combinatorial circuit is the path with the longest delay. Ultimately this path determines the maximum achievable clock

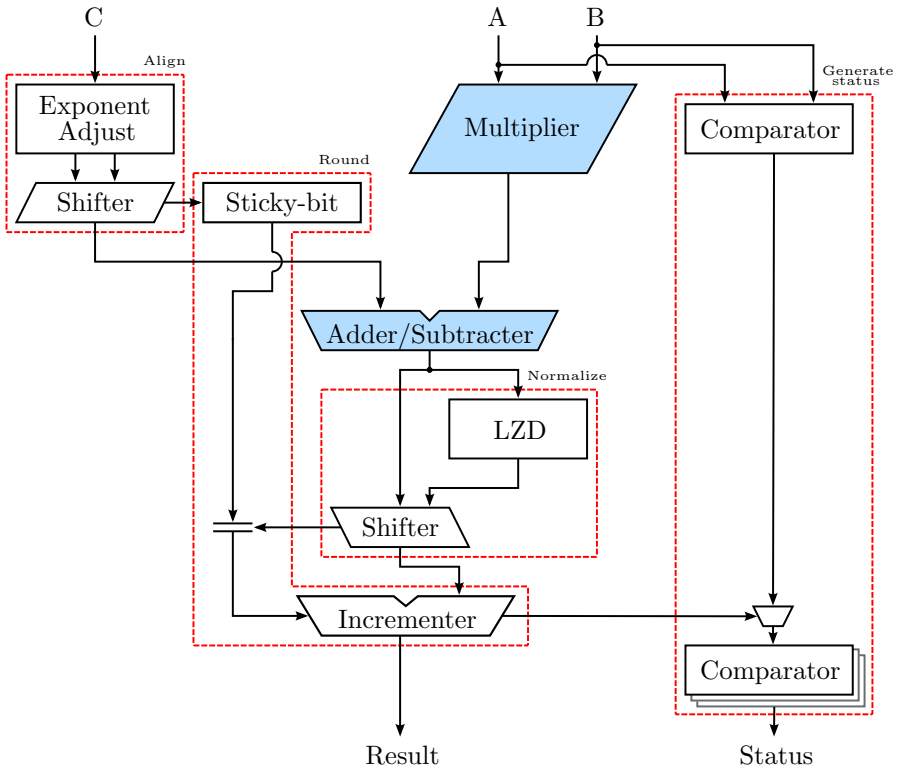


Figure 4.11: Proposed FMA floating-point and integer datapath

speed of the system. If we would synthesize this datapath, the expected performance will be rather low. The latency is simply too massive to reduce to acceptable levels by simply relying on the optimization heuristics of logic synthesis tools. One way to deal with extremely long critical paths is pipelining.

Pipelining is a technique that separates long critical paths in smaller paths by placing registers between combinatorial circuits or parts of those circuits. The smaller critical path allows faster clock speeds making the overall system run faster. We have to keep in mind though that pipelining only increases throughput and does not actually reduce the latency of the operation. Pipelining even adds more delay to the total latency by inserting flip-flops (registers) into the combinatorial circuit. However, since throughput is what we are usually interested in, pipelining is an excellent means of increasing the performance of ALUs. We will first illustrate why pipelining is beneficial for floating-point arithmetic.

Suppose we perform a number of floating-point multiply-add instructions. For such an instruction, the ALU has to take the following actions: align a third operand to the multiplication of the two primary inputs operands (we will refer to this action as `Aln`), multiply the two primary operands (`Mul`), add the aligned operand (`Add`), normalize the result (`Nrm`) and round the normalized result (`Rnd`). We can pipeline this instruction by placing register between each of these operations (i.e, the hardware blocks that implement these actions). In an unpipelined datapath all the action have to be completed before another instruction can be initiated. In a pipelined datapath on the other hand, we can already initiate the next instruction after the `Aln` step has completed. This way, major parts of the instruction latency overlap such that the *wall time* of the instruction is reduced significantly. Figure 4.12 illustrates the advantage of a pipelined versus a non-pipelined sequence of instructions

This example suggests that the more pipeline stages we create, the more throughput can be achieved and thus a faster system is obtained. In essence this is true but there are a few drawback for deeply pipelined architectures. The first major drawback is that *pipeline hazards* can occur. A pipeline hazard occurs when the n^{th} and $n\text{-}1^{th}$ stage of two consecutively issued instructions simultaneously require

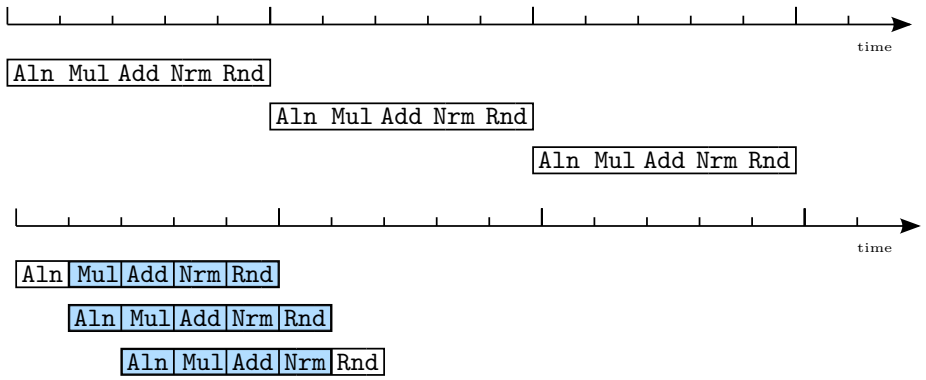


Figure 4.12: Pipelined vs. non-pipelined multiply-add instructions

the same hardware. Obviously only one instruction can be executed on a certain part of the datapath. Solving pipeline hazard can become quite complex, and when hardware is shared between pipeline stages, the chance of hazards increases as the number of stages becomes larger. A second major drawback is that scheduling instructions for a pipeline is harder than for single cycle systems. Previous work on reconfigurable processors [32] has shown that compilers for such platforms are notoriously hard to make. Pipelining the ALU will only add more the difficulty. Since we expect a strong dependency between the number of stages and the difficulty of compiling to efficient machine code, the number of pipeline stages is kept to a minimum.

Based on earlier work [17, 33] and preliminary latency estimates, the minimum number of pipeline stages for the performance we desire is set to three. For maximum effectiveness, the pipeline stages must be balanced. This means that the latency of each pipeline stage should be roughly the same. A balanced pipeline is shown in Figure 4.13. With some special techniques (Chapter 5), the latency of a fast multiplier and alignment of C is comparable to that of a large adder. Similarly, the latency of normalization and rounding can be matched to that of the multiplier and the adder.

Retiming

The balance between the pipeline stages shown in Figure 4.13 can be further fine tuned with automated *retiming* [34]. Retiming is a technique where registers are structurally relocated to improve performance without affecting functional behavior. Take for example an arbitrary pipelined circuit implemented in a non-particular fictional technology[4] as shown in Figure 4.14(a). The balance between these pipeline stages is far from optimal. The second stage is three times longer than the first stage. As mentioned before, the maximum clock speed is determined by the length of the critical path, in this case 3ns or 333MHz. After retiming (Figure 4.14(b)) the balance is better (optimal in this example), both stages are 2ns resulting in a much faster clock speed of 500MHz.

The three pipeline stages are based on estimates and are therefore not perfectly balanced. Retiming this circuit by hand is cumbersome. Fortunately, most modern synthesis tools facilitate automatic retiming for sequential circuits. The design we present in this chapter relies on automatic retiming. However, the imperfections without retiming are small. Even without retiming the presented pipelined design guarantees good performance.

[4] The delay per component highly depends on the technology used, here we just assume some convenient numbers that by no means represent actual synthesis results

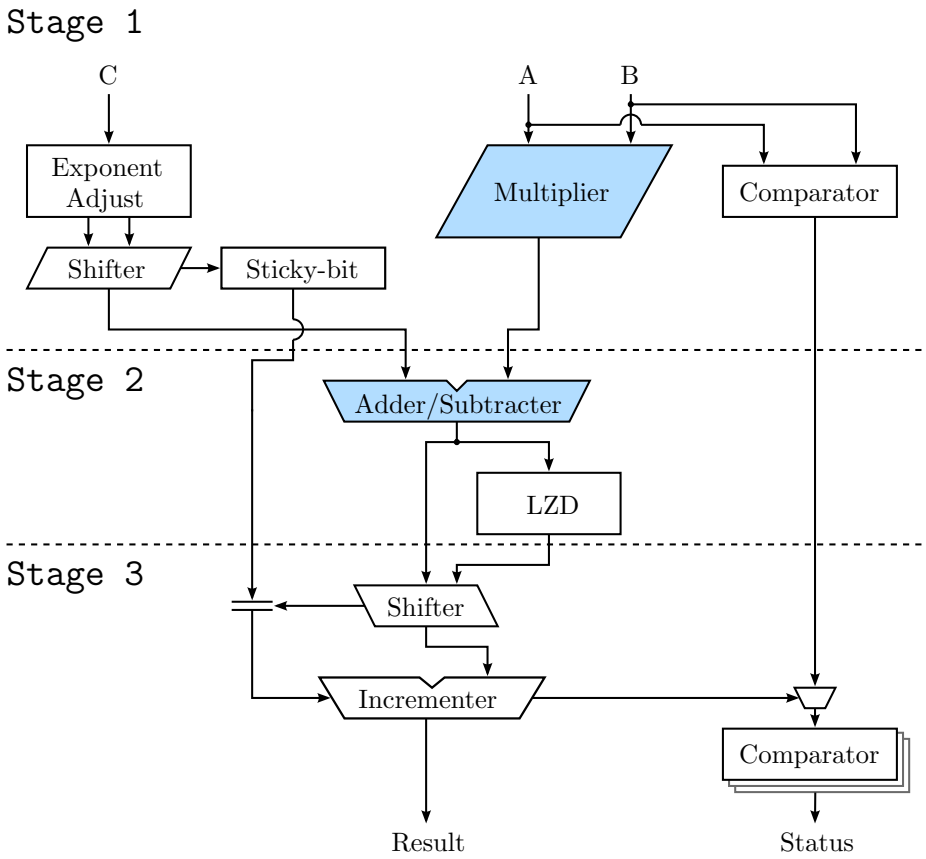


Figure 4.13: Pipelined datapath

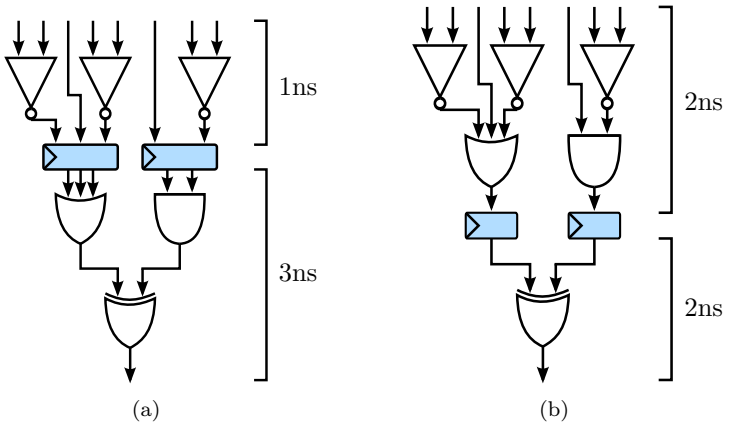


Figure 4.14: Arbitrary sequential circuit before and after retiming

4.3.6 Control and Datapath Reconfiguring

So far we have mainly focused on dataflow and not so much on the control to steer the data flow in the correct directions. Because the hardware is shared, controlling the dataflow is quite comprehensive. Almost all the major components have select signals to switch between two's complement and unsigned input. In addition, large parts of the floating-point datapath can be skipped for integer operands. In fact, the entire final stage can be skipped for all but the shift left instruction. For consistency all integer instructions complete in two clock cycles and all floating-point instructions in three cycles, despite the fact that compares and shifts could also be done in a single cycle. The output and inter-stage registers are multiplexed to provide the means of controlling the dataflow.

Configuring and reconfiguring hardware can be done in two ways. One can pre-configure all components and multiplexers to set the dataflow for a typical instruction, which is quite efficient for bursts of the same instruction. The other way is to configure the hardware during execution of the instruction for seamless transition between different instructions. This has the advantage that the datapath can quickly switch between different types of instructions without any noticeable overhead. Configuration of the ALU presented here is based on the last alternative. We believe the maximum use of a shared ALU is achieved when the overhead is the least. Pre-configuration would require at least one extra clock cycle although but it may also have a benefit for area and energy efficiency. In the first stage the opcode is translated into control signals that set up the components (multiplexer, shifter, comparators) for signed or unsigned execution. The same control signal also set up the multiplexers for the next stage. In the second stage more control, generated by the logic of the first stage, sets up the components in the current pipeline stage. This continues for the last stage until the final output is formatted and the instruction completed. The control signals flow with the data through the datapath configuring the hardware on-the-fly for correct processing.

One last issue that needs to be discussed is the pipeline hazard caused by a difference in duration of floating-point and integer instructions. The transition from integer to floating-point operation is trouble-free if we ignore the output for one cycle. Switching from floating-point to integer always results in two sources driving the same output because both instructions complete at the same time. This can only be solved by issuing a NOOP (NO OPeration) instruction between each transition from floating-point to integer operation. In practice, this means that no relevant data should be processed by the ALU in that clock cycle.

4.4 Summary

We have shown a domain independent ALU for integer and floating-point arithmetic in multi-core systems. The ALU supports full 32-bit integer arithmetic and a custom 41-bit floating-point format that strongly resembles IEEE single precision. The difference is that the significand of this floating-point format is 32-bit instead of the usual 23. This provides additional precision and a nice regular datapath that can be shared more easily between both data types. The datapath of this ALU consists solely of a FMA unit with a comparator. All arithmetic operations are derived from the FMA unit by using constants 1 and 0 for addition and multiplication respectively. The datapath has been divided in three pipeline stages for higher throughput. All floating-point instructions take three clock cycles to complete and all integer instructions two. The transition from floating-point to integer processing is seamless, from integer to floating-point causes a data hazard which is solved by stalling the pipeline with a `NOOP` instruction. The implementation of this datapath is not straightforward. Most of the components needed are not off-the-shelf and need clever and robust design principles. The next two chapters thoroughly describe the details of implementing the datapath presented in this chapter. The last chapter discusses the physical aspects of the ALU such as area and power consumption.

5 Arithmetic Design Principles

5.1 Introduction

In the previous chapter, a general outline for a FMA floating-point unit with augmented integer functionality was discussed. Such a processing core is not straightforward to implement, especially implementing floating-point arithmetic is a comprehensive task. History shows that floating-point units are notorious for being error-prone. A few missing entries in the FDIV lookup table cost Intel approximately 475 million dollars. This stipulates that one needs to be very cautious with floating-point arithmetic. Errors are easily introduced and hard to find in floating-point hardware. On the other hand, floating-point arithmetic is also known for being slow. Equivalent floating-point instructions usually need considerably more clock cycles than integer instructions. The floating-point datapath must often be deeply pipelined in order not to affect the processor's performance. In most cases architectural optimizations are a necessity, hence we have a real need for robust design techniques for fast floating-point FMA.

This chapter provides a mathematical background for implementing efficient (FMA) floating-point units. Because floating-point arithmetic has been around for a considerable amount of time, many design principles have been documented, although not always very thoroughly. Some work only covers a local optimization, in other work entire floating-point units are described. The work described here is based on local architectural optimizations. The different stages of floating-point arithmetic: exponent alignment, addition/multiplication/subtraction/division, normalization and rounding will be discussed in chronological order. Although the majority of the principles discussed is applicable to any type of floating-point arithmetic, our focus will be FMA units.

Sometimes alternatives are available for optimizing a specific step. Also the classical trade-off between area/energy efficiency and performance (latency) plays an important role in this chapter. If alternatives are important to understand the benefit of a certain design choice, they will shortly be elaborated. However, we will not discuss the details. References are available for the interested reader.

The idea behind the proposed ALU is that it is suitable for low-power (embedded) systems. For this reason it was decided to design with a low-power manufacturing technology in mind. These libraries generally mean low(er) performance. Combined with the fact that the number of pipeline stages is restricted to three, performance is seriously affected. Therefore a lot of effort was put into optimizing for low latency to gain as much performance as possible. In this chapter noticeably more attention is given to performance optimization. Area efficiency will be more prominent in Chapter 6, where we discuss the implementation details.

5.2 Alignment

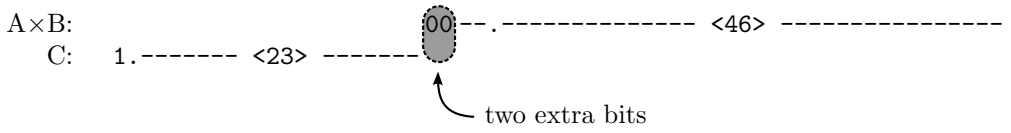
The first obstacle one finds in implementing FMA is the alignment of the product A×B and the addend C. Normally when floating-point addition is performed, the operands are compared and the smallest operand is shifted to the right to preserve as much precision as possible (see Chapter 2). As expected, this still applies to FMA, however such a straightforward approach will result in a major performance penalty. In a typical floating-point adder, both operands are available at the beginning of first cycle, whereas the A×B product in FMA usually requires at least one clock cycle before it is known. This means that before alignment can start, we would have to wait until the product is available. Once the product is known, a large compare would have to be performed (A×B is be twice as large as the original input) to determine which operand is the smallest. Only then can the operands be aligned by shifting the smallest one. Such an approach requires at least an additional pipeline stage (i.e., an additional clock cycle).

In FMA, we preferably always shift the C operand with respect to A×B because it is immediately available. However, this operand does not necessarily have to be the smallest operand. When C is larger than A×B, a left-shift is needed to align the exponents. This could result in catastrophic events where MSBs are shifted out of range and the entire computation returns incorrect answers. The solution is to treat the product as having a fixed-point and place the C operand all the way in front of the product such that the alignment can be implemented as a right-shift only.

Assume we apply this to a single precision floating-point format (23-bit significand and 8-bit exponent). After the hidden-bit has been made explicit, each significand is 24 bits wide. After multiplying A and B, we end up with a product of 48 bits. Recall that the floating-point in IEEE format is right behind the hidden-bit. If we treat the point as fixed, we first have to account for the fact that after multiplication, there will be two bits before the point.

```
   A:  1.------- <23> ---------        B:  1.------- <23> ---------
A×B:  --.--------------- <46> ----------------
```

By placing the significand of C in front of the product, we can always shift to the right because in fixed-point representation this number will always be larger than the product. All 24 bits of C must be placed at least in front of the MSB of the product. However, it is very convenient to place two more empty positions (0s) in between for several reasons.



For correct rounding, we require a *guard-bit* and a *round-bit*. Normally when we add two floating-point numbers these will be the two LSBs. However, because we only shift C as proposed, we can also place them between C and A×B. The benefit of placing the guard and round-bit here is that the significand will never overflow during addition. Because both positions are zero, a carry can never propagate any further than the first position.

The actual shift amount for alignment is determined by the absolute difference between the exponents of the two operands. For FMA this is the exponent of A×B minus the exponent of C. Because the exponents are biased in IEEE-754 format, the bias is accumulated during addition of A and B. This needs to be corrected by subtracting the bias again. This still does not produce a correct answer, because we have altered the position of the floating-point by placing C entirely in front of the product. By placing C in

front of A×B, the floating-point has moved 26 positions to the left (significand width + hidden-bit + guard-bit + round-bit). In addition we have to account for the fact that a multiplication results in two bits in front of the point as explained earlier. The result is an offset of 27 bits. By combining the offset and the bias, a new constant is obtained that needs to be subtracted from the exponent obtained by simply taking the difference between A×B and C. The shift amount is found by Equation 5.1.

$$\text{Exponent} = A_e + B_e - C_e - \text{Bias} + (\text{Significand width} + (\text{Hidden, Guard and Round-bit}) + 1) \quad (5.1)$$

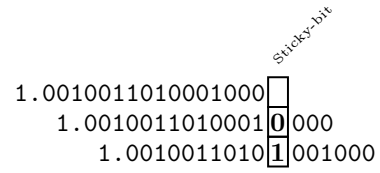
What is often neglected in literature, is how all this affects the exponent. In regular floating-point addition, the new exponents will be equal to the exponent of the operand that is not shifted. However, in FMA the exponent changes due to the imaginary fixation of the floating-point. As a starting point we can take the exponent found after adding A and B. To find the correct exponent, compensate for the bias accumulation and take the offset by placing C all the way in front of the product into account.

$$\text{Exponent} = A_e + B_e - \text{Bias} + (\text{Significand width} + (\text{Hidden, Guard and Round-bit}) + 1) \quad (5.2)$$

5.2.1 Sticky-Bit

Inexact *sticky-bit* (or just sticky-bit) calculation is closely related to the alignment phase of floating-point addition. The sticky-bit is required to guarantee correct rounding in the final stage of floating-point arithmetic (Chapter 2). The purpose of the sticky-bit is to indicate that the unrounded result is inexact. Result are inexact if meaningful bits (1s) fall outside the maximum range of the final output. In a typical FMA datapath, inexactness can occur in two places: when bits are shifted out of range during alignment of C, and when the intermediate result is scaled back to register size after normalization.

During alignment, the sticky-bit calculation can be thought of as an extra bit position behind the shifter that records all bits that pass through it while being shifted out of range.

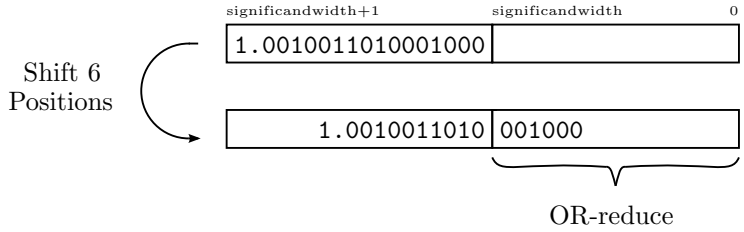


If a 1 passes through, the sticky-bit will remember this and stay 1 regardless of the other bits that pass through. This can not directly be implemented in hardware, because most shifters do not shift bit-after-bit (this is highly inefficient). The sticky-bit can however be found by OR'ing all bits beyond the LSB into a sticky-bit. All we have to know is which bits fall out of the range after shifting.

The easiest way of implementing this is extending the shifter such that bits will never really be discarded. No bits are lost and the part that is not used to further compute the multiply accumulate can be OR'ed into a sticky-bit. This is very inefficient because the already large shifter (approximately three times as wide as the significand) will be even further extended, increasing its latency tremendously. For example, the theoretical maximum shift in IEEE single precision is $127 - (-126) = 253$ positions. To ensure no bits are lost a 277-bit shifter would be required. Alignment shifting is part of the critical path in the first stage, thus we can not tolerate the additional latency. By distinguishing the worst and best cases, a more efficient solution can be found.

First we have to determine when bits will fall out of the representable range. In case of the alignment method described above, the operand that is shifted has been extended to three times the significand width plus its hidden-bit and the guard and round-bit (e.g., $3 \times 23 + 2$ for IEEE single precision). When the shift amount is more than two times the significand width plus hidden-bit, plus the guard- and round-bit (68 for IEEE single precision), bits are starting to shift out of range. This is a constant we can use to improve sticky-bit calculation.

The best case is when the shift amount is lower than the constant. No bits are lost and the sticky-bit is simply `0`. The worst case is when all input bits are shifted beyond this constant, i.e., when the shift amount is *significand width + hidden-bit* bits more. If all bits are shifted out of range, we can compute the sticky-bit by OR'ing all of the input bits together. Cases in between the best- and worst case are a bit more difficult because the input bits are partially shifted out and we have to determine which bits. A 'container' needs to catch the bits shifted out. We can do this by padding the input with zeros up to twice its original length. This input is shifted by a second (smaller) shifter that only shifts by the amount of positions equal to the original shift amount minus twice the significand width plus hidden-bit and the guard and round-bit. The bits that are shifted out are now contained in the lower (*significand width + hidden-bit*) bits of the output. By OR'ing these bits, we find the sticky-bit as shown in the example below.



The entire procedure for sticky-bit calculation is shown in Algorithm 5.2.1 again for clarification.

Algorithm 5.2.1 Sticky-bit calculation

Input: (operand) C and shift (amount)
Output: sticky

```
1: if shift ≥ (3×(significandwith + hiddenbit) + guardbit + hiddenbit) then
2:    sticky ← or_reduce(C)
3: else if shift ≥ (2×(significandwith + hiddenbit) + guardbit + hiddenbit) then
4:    sticky ← '0'
5: else
6:    newshift ← shift - (3×(significandwith + hiddenbit) + guardbit + hiddenbit)
7:    shifted ← resize(C,2×(significandwith + hiddenbit)) >> newshift
8:    sticky ← or_reduce(shifted)
9: end if
```

Because we have distinguished the best, worst and 'in-between' cases, we can ensure that bits are shifted no further than the significand width plus hidden-bit. With this property the size of the 'container' that has to catch the bits that shift out can be constrained to the length of the input. We achieve two objectives with this approach. Firstly, the required shifter(s) are smaller than one shifter that ensures no bit are lost. Secondly, we can start calculating the sticky-bit in parallel with the actual shift, resulting in a faster circuit. Both latency and area are improved.

The second time inexactness can occur is when the intermediate result is normalized and resized back to the original input size. The internal floating-point datapath is much wider that the final result. It is almost inherent that we loose precision when the internal representation is scaled back. Just like the bits that are shifted out during alignment, we can OR the bits that are discarded after normalization

into a second sticky-bit. By OR'ing this sticky-bit with the one we found during alignment, we end up with a sticky-bit that accounts for all precision that was possibly lost during multiply accumulate.

5.3 Multiplication

Multiplication is an example of the most elementary operations in computer arithmetic in general. This heavily used operation is unfortunately also very costly to implement. Fundamentally, a multiplication is a series of additions. A binary n-bit multiplication can be realized by performing a series of n shifts and additions. We have a choice to fold these over time or space. The most area-efficient solution is to perform every shift and addition on the same hardware, but this would mean that a 32-bit multiplication requires 32 cycles to complete. This is unacceptable for the purpose of our proposed ALU. The alternative is to expand over space by using n adders and shifters, which is why high performance multiplication is so expensive to implement. Because multiplication is essential for many algorithms, designing multipliers that are high-speed, low-power, and/or regular in layout have always been of substantial research interest. There are many optimizations in existence but the most popular are *Wallace Trees* and *Booth encoding*. The multiplication process consists of two steps, generating *partial products* and summing the partial products to a final product. Booth encoding [12] is aimed at reducing the number of partial products. The Wallace tree [8] focuses on summing the partial products in a very efficient way. Many floating-point units incorporate both techniques in the multiplier.

5.3.1 Booth Encoding

Booths' algorithm serves two purposes. First it enables us to multiply signed (two's complement) numbers and secondly it helps reduce the number of partial products. Although the first is not directly relevant for (IEEE) floating-point multiplication, the second property can increase the speed of the multiplier and reduce its area. To understand Booth multipliers we first have to recapitulate the basics of binary multiplication. Several important observation can be made in binary multiplication:

- In multiplications (A×B) we have a multiplier (A) and a multiplicand (B). For each digit in the in the multiplier, a *partial product* is generated.
- If the multiplier bit is `0`, the partial product is zero, otherwise the partial product is the multiplicand.
- The final product is produced by repeatedly shifting a partial product to the left and adding it to the preceding partial product.

A small example of binary multiplication is shown below

```
      101110   Multiplicand
      010011   Multiplier
------------------------------
      101110   Partial products
     101110
    000000
   000000
  101110
 00000
------------------------------
001101101010
```

Each time a `1` is encountered in the multiplier, the multiplicand is shifted and added the the previous partial product. When a `0` is encountered, we only shift the partial product. Now consider a positive multiplier with a block of `1`s surrounded by `0`s, for example `00111110`. Multiplied by an arbitrary multiplicand M we find the product by

$$\mathrm{M} \times \texttt{00111110} = \mathrm{M} \times (2^5 + 2^4 + 2^3 + 2^2 + 2^1) = \mathrm{M} \times 62$$

The same multiplication can be performed by

$$\mathrm{M} \times (2^6 - 2^1) = \mathrm{M} \times \texttt{010000(-1)0} = \mathrm{M} \times 62$$

This eliminates several consecutive 1s, reducing the number of required operations. Booth has shown that the above example can be extended to any number of blocks of 1s. Therefore it is possible to simplify multiplications with simpler operation by string manipulation. Booth formulated the algorithm as shown in Algorithm 5.3.1. $\mathcal{A}$ and $\mathcal{S}$ are two predetermined values that are conditionally added to product $\mathcal{P}$. Let m and r be the multiplicand and the multiplier and x and y the width of m and r respectively.

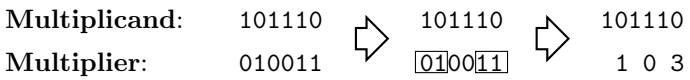
Algorithm 5.3.1 Booth multiplication

1: Determine $\mathcal{A}$ and $\mathcal{S}$, and the initial value of $\mathcal{P}$. All numbers should have length (x + y + 1).
   $\mathcal{A}$: Fill MSBs with the value of multiplicand m. Fill remaining (y + 1) bits with 0s.
   $\mathcal{S}$: Fill MSBs with the value of (-m) in two's complement and the remaining (y + 1) bits with 0s.
   $\mathcal{P}$: Fill the x MSBs with zeros and append the value of r. The LSB is 0.
2: Determine the two least significant (rightmost) bits of $\mathcal{P}$.
3: **if** 01 **then**
4: $\quad \mathcal{P}_{next} \leftarrow \mathcal{P} + \mathcal{A}$
5: **else if** 10 **then**
6: $\quad \mathcal{P}_{next} \leftarrow \mathcal{P} + \mathcal{S}$
7: **else if** 00 **then**
8: $\quad \mathcal{P}_{next} \leftarrow \mathcal{P}$
9: **else if** 11 **then**
10: $\quad \mathcal{P}_{next} \leftarrow \mathcal{P}$
11: **end if**
12: (Arithmetic) shift the obtained value one position to the right.
13: Repeat steps 2-12 , y times.
14: Drop LSB from $\mathcal{P}$ to find the product of m and r.

If the 1s in the multiplier are grouped into long blocks, Booth's algorithm performs fewer additions and subtractions than a normal multiplication algorithm. Booth encoding is however not directly applicable to high speed multipliers based on tree structures (Section 5.3.2). Tree structures are static which means all partial products have to be known a-priory while the reduction achieved by Booth encoding entirely depends on the input. The tree structure must account for the maximum number of partial products, hence no improvement is made by using Booth encoding. Booth's algorithm can only be exploited in combination with tree structures when the *radix* of the multiplication is increased.

Modified Booth Encoding

The examples shown so far were all radix-2 multiplications. In radix-2 multiplication each bit of the multiplier generates a partial products. By using radix-4 multiplication, the number of partial products can be halved at the cost of more complicated partial product generation. In radix-4 multiplication, two adjacent multiplier bits are used per partial product generation, as shown in the example below.



```
          101110
           1 0 3
-----------------
        10001010
        000000
     101110
-----------------
   001101101010
```

To make this example work, the multiplier must first evaluate 1×, 0× and 3× multiples of the multiplicand. The 1× and 0× multiples are not a problems, they can both easily be found by shifting the multiplicand. The 3× multiple (`001101101010`) on the other hand is highly irregular and requires special hardware to pre-calculate. By using Booth encoding, the 3× multiple can be re-written to (4×)-(1×). Thus Booth encoding eliminates the 3× multiple and the set of multiples (-2×, -1×, 0×, 1×, and 2×) is once again restricted to regular multiples of the multiplier that are trivial to determine.

By using an even higher radix, we run into irregularities that are not easily solved (Booth encoding does not help for 3- and 5-bit numbers). For this reason, the majority of the high speeds binary multiplier designs stick to radix-4 booth encoding. They usually implement Booth encoding by placing a *Booth recoder* before one of the inputs of the multiplier. When the input of the multiplier is n bits wide, radix-4 booth encoding is able to reduce the number of additions (and therefore the number of CSA elements in the tree) from $n+1$ to $\lceil (n+1)/2 \rceil$. Since Booth encoding can be done carry free and fully parallel for each multiple [8], the optimization not only reduce the area of the multiplier tree, but also the corresponding latency. Moreover, the multiplier supports two's complement signed numbers from which we will benefit by re-using the floating-point multiplier for integer multiplication (Chapter 6).

5.3.2 Wallace Tree

Once the partial products are determined they have to be summed to form the final product. Depending on the radix chosen for the multiplication, the number of partial products can become quite large. If we would just perform one addition after another, the latency of the multiplier will render it useless in terms of performance. To improve this, partial products are summed in a tree-like structure called a Wallace tree. This reduces the delay from n additions to $log_2(n)$ additions. However, due to *carry propagation*, the latency is still too large compared to alignment of C. A special adder that eliminates carry propagation is needed for the multiplier to match the latency of alignment such that the latency of the critical path in the first stage is minimal.

5.3.3 Carry-Save Adders

The adder is also one of the oldest and most widely used arithmetic components in digital processors. Its purpose is to add two operands A and B. In its simplest form a binary adder adds two bits. Such a combinatorial circuit is called a *half adder*. The elementary operations of a half adder are `0 + 0 = 0`, `0 + 1 = 1`, `1 + 0 = 1` and `1 + 1 = 10`. When both the augend and addend are `1`, the output consists of two bits. Because of this, the output of an adder is always represented by two bits, the *sum* and the *carry*. If n-bits operands are added, the carry of bits in position *i-1* ($i \leq n$) is added to the next higher order pair of bits *i*. This requires a combinatorial circuit that can perform addition on three bits: A, B and the carry in. Such a circuit can be constructed from two half adders combined with an OR-gate and is called a *full adder*.

To add two n-bit operands, a chain of n full adders can be used in cascade, with their carry out from the full adder connected to the carry in of the next full adder, as shown in Figure 5.2. This simple and straightforward implementation is know as a *ripple carry adder*. An obvious problem with the ripple carry adder is the latency of the propagating carry. In the worst case, the carry has to propagate all

the way from the LSB to the MSB. For a typical ripple carry adder the carry propagation is $2n+n$ gate delays. A well-know optimization for this problem is the *carry lookahead adder* [7].

In carry lookahead, the addition is separated into smaller groups of bits (often four bits per group). For each group, carry lookahead logic determines if a carry is going to propagate to the next group. Simultaneously all 1-bit adder components produce their one bit results (the carry will still propagate through each group). When the carry is going to propagate to the next group of bits, the carry lookahead logic will already have deduced this. The net result is that the carries propagate through each (4-bit) group, just as in a ripple-carry adder, but then four times as fast. Leaping from one lookahead carry unit to the next. The number of bits in a group is not an arbitrary choice. The more bits per group, the more complex carry lookahead logic becomes and the longer the latency will be. However, smaller groups means more groups have to be traversed and less acceleration is achieved. No matter how good the optimization, carry propagation will always result in high latencies. In most cases, too high for a multiplier.

The carry-save adder (CSA) is a type of adder that computes the sum of three or more binary inputs. It differs from other binary adders in that it outputs two numbers of the same dimensions as the inputs, one which is a sequence of partial sum bits and another which is a sequence of carry bits. Due to this redundant form of output, carry propagation is completely eliminated. A single bit, three bit input CSA is shown in Figure 5.1, compared to a full adder.

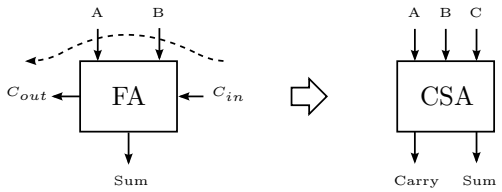


Figure 5.1: Full-adder and carry-save adder

For a single bit, the only difference between a CSA and a full adder is the way we interpret the output. However, just like a regular carry-propagate adder, we can create n-bit CSAs by connecting multiple 1-bit CSAs as shown in Figure 5.2. It clearly shows why CSA is preferred over carry propagate when the width of the operands becomes larger. While the latency of a normal adder increases linearly with the width of the input, the CSA's latency remains the same for any length.

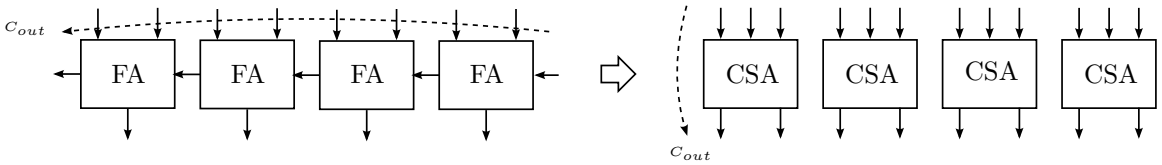
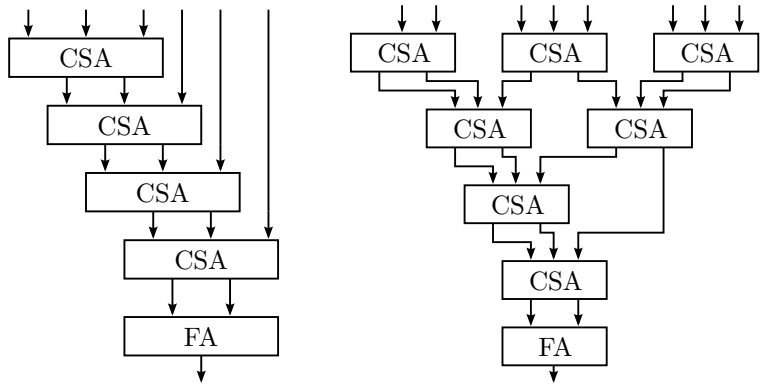


Figure 5.2: CSA carry elimination

Let us illustrate a CSA operation with a small example. Consider two 8-bit input operands, perform addition but ignore the carries:

A	01000101
B	11010111
Sum	10010010

Now perform the same addition, but only pay attention to the carries:



(a) Cascaded CSA array (b) CSA Tree

Figure 5.3

A	01000101
B	11010111
Carry	01000101

We end up with a redundant carry-save format of the addition. A sum 10010010 and the carries 01000101. What is nice about this redundant format is that if we add the sum and carries in a normal fashion (i.e., let the carry propagate), we end up with the same result as if we had performed addition with carry propagation.

Sum	10010010
Carry	01000101
Result	100011100

A	01000101
B	11010111
Result	100011100

Thus, the redundant carry-save form is converted back to the normal binary representation by adding the sum and carry with a full adder. This might sound as if the CSA is only overhead, but this is not true. The CSA is very useful when multiple additions have to be performed consecutively. In a $m \times m$ multiplication, m additions are performed. Multipliers implemented based on full adders yield a massive $m \times m$ latency due to carry propagation. If all but the last addition are carried out by CSAs, the latency of carry propagation is reduced to $m + m$. For the final addition, we can take one of the traditional fast carry propagate adders to minimize the latency of the final addition.

5.3.4 Wallace Tree Configurations

Instead of placing the CSAs in a cascaded array as shown in Figure 5.3(a), the CSAs can be placed in a tree-like structure as shown in Figure 5.3(b). The latency of the multiplier is then even further reduced to $log_2(m) + m$. A tree of CSAs is called a *Wallace Tree*. With this tree structure of CSAs, it is often possible to match the delay of alignment. Despite the fact that the array of Figure 5.3(a) is much slower than a tree structure, it can still be interesting for certain applications. An array is much more regular in structure and uses shorter wires to connect the individual components. Both properties are beneficial for low-power design.

The simple example of Figure 5.3(b) is only one of many possible configurations of the Wallace tree. The optimal configuration of a Wallace tree highly depends on the width of the input operands and the type(s) of CSAs used. Figure 5.3(b) depicts a Wallace tree constructed from solely 3:2 CSAs (3 input 2

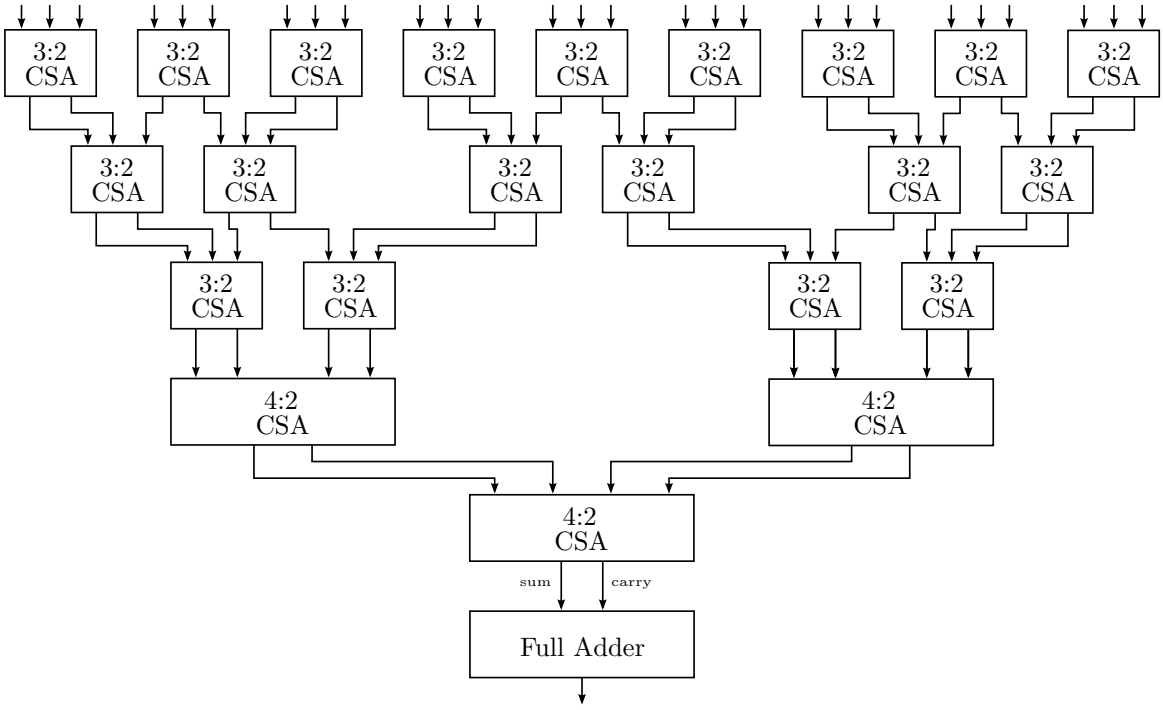


Figure 5.4: Wallace Tree

output). Every CSA in this tree reduces three partial product to two partial products. Because of this property, the CSAs are also known as 3:2 compressors. Compressors can also be made to reduce four input values, or even larger numbers. However, the higher order compressors have never really become popular. Most Wallace trees are constructed from 3:2 and 4:2 CSAs.

Figure 5.3(b) shows a Wallace tree configuration that is popular for single precision floating-point multipliers. By using 4:2 compressors in the last stages, one level of compression is eliminated which is beneficial for the latency of the multiplier. Of course the 4:2 compressors will have to be implemented efficiently and not just cascade two 3:2 CSAs [30].

5.3.5 Partial Product Multipliers

The use of CSAs in multiply arrays (trees), provides one of the most elegant and efficient solutions for multiply-accumulate. In the before-last stage of the multiplier, all partial products have been reduced to a product in carry-save format. By inserting one more CSA between the partial products from this stage and the full adder (Figure 5.5), we can almost freely 'inject' the addend into the multiply array before the cary-save format is converted by the final carry-propagate adder.

5.4 Addition

So far we have seen that C can efficiently be aligned to the product in parallel with the multiplication of A and B itself. Even adding C to the product does not require many resources or additional delay. However, we have not yet mentioned anything about negative numbers. A problem arises when the signs of A×B and C are different or when we simply want to subtract C from A×B.

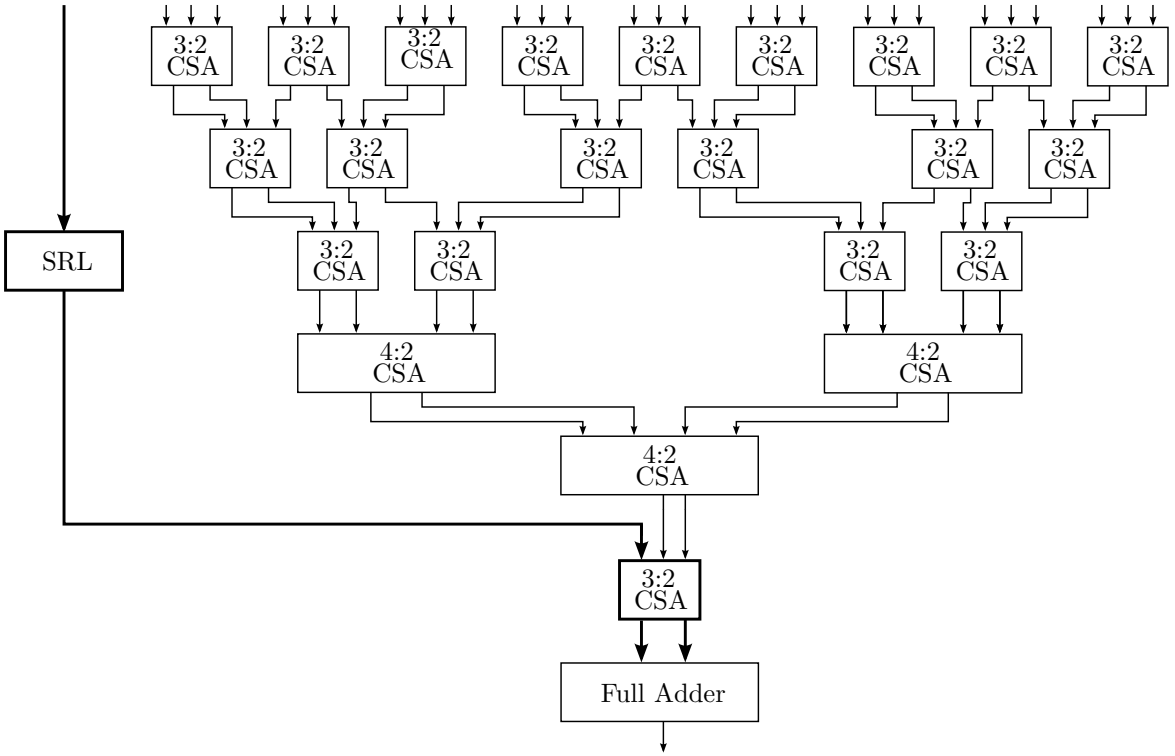


Figure 5.5: Fused multiply-add

Given that A and B are sign-magnitude numbers, negative numbers are only differentiated from positive numbers by their sign-bit. When multiplying two sing-magnitude numbers, it is sufficient to multiply the magnitudes and deal with the sign-bit separately (a single XOR-gate computes the correct sign). Addition and subtraction on the other hand, are more complicated. We can only treat the sign-bit apart from the magnitudes when both signs are equal. Eight different scenarios can be distinguished when performing signed addition/subtraction [1]:

A	+	B	
-A	+	-B	
A	+	-B	(A > B)
A	+	-B	(A < B)
A	-	B	(A > B)
A	-	B	(A < B)
-A	-	-B	(A > B)
-A	-	-B	(A < B)

When dealing with signed numbers, addition and subtraction are not fundamentally different operations. A - B is effectively the same as A + -(B). Based on this observation, the list of different scenarios to account for can be reduced considerably:

A	+	B	
-A	+	-B	
A	+	-B	(A > B)
A	+	-B	(A < B)

[1] The signs of A and B are interchangeable, these differences do not affect the operation and are not counted

This set of operations is just as powerful as the initial one while the number of different cases to account for is significantly reduced, therefore simplifying the problem at hand. The sign of a number in sign-magnitude representation can easily be changed by swapping the sign-bit, making this reduction very interesting for hardware implementations. The first two cases can be implemented by adding the magnitudes and keeping the sign-bit unchanged. However, with a normal adder, subtraction is still not possible and ideally we would like to have a FMA unit that accepts any number, positive or negative, on every input. These problematic cases will be referred to as *effective subtractions*, the other are *effective additions*.

5.4.1 End-Around Carry Addition

As discussed in Chapter 2, we can deal with addition and subtraction in a normal adder by converting the sign magnitude representation to one's or two's complement and use a regular adder to perform the effective addition or subtraction. This is however a very costly operation, especially in the case of multiply-add where three operands will have to be converted. A more elegant solution is *End-around Carry Addition*.

End-around Carry Addition requires a basic carry propagation adder that operates on two unsigned numbers of length n. The magnitudes of A and B (unsigned significands), are denoted as |A| and |B| respectively. Assuming that the operation is effective addition, the magnitude of the result is equivalent to the addition of |A| + |B|, just like expected. Effective subtraction on the other hand results in a positive number when |A| > |B| and a negative number when |A| < |B|. According to the theorems of end-around carry addition [35], if |A| > |B| we can compute the magnitude by

$$|\mathrm{R}| = |\mathrm{A}| - |\mathrm{B}| = |\mathrm{A}| + \overline{|\mathrm{B}|} + 1 \tag{5.3}$$

This basically means that |B| is inverted (converted to one's complement) before being added to |A|. After the addition, the result in incremented to find the correct magnitude. An important property of end-around carry addition is that the carry out is always `1` is this case. The proof is omitted here but can be found in [35].

If |A| < |B|, the resulting magnitude can be computed by

$$|\mathrm{R}| = |\mathrm{A}| - |\mathrm{B}| = \overline{|\mathrm{A}| + \overline{|\mathrm{B}|} + 0} \tag{5.4}$$

In this case the carry out is always `0`. Just like in the first case, the |B| operand is inverted. However, the resulting magnitude is not incremented but inverted again. From 5.3 and 5.4 we can derive a more generalized equation for effective subtraction. In 5.3, the result is incremented and the carry out of the adder is `1`. In 5.4, the result is not incremented and the carry out is `0`. A logical consequence is that we can just add the carry out of the adder to the result.

$$\Sigma = |\mathrm{A}| + \overline{|\mathrm{B}|} + \mathrm{C}_{out}$$

For effective addition we can do the same, except that |B| should not be inverted.

$$\Sigma = |\mathrm{A}| + |\mathrm{B}|^* + \mathrm{C}_{out}$$

$|\mathrm{B}|^*$ is equal to |B| for effective addition and $\overline{|\mathrm{B}|}$ for effective subtraction.

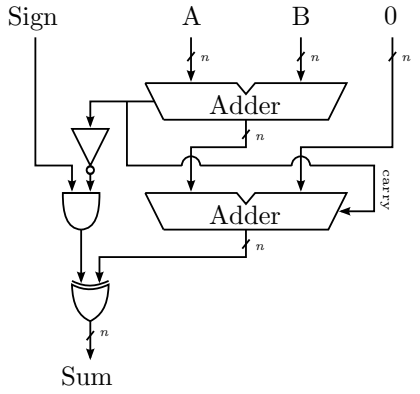


Figure 5.6: End-around carry addition

The only thing remaining is that the result must be inverted when the effective operation is subtraction and |A| < |B|. This case can be detected by combining the signs of the operands with the carry out of the adder (see also Section 5.4.2)

$$\Delta = \overline{C_{out}} \wedge \text{effective operation}$$

When Δ is `1` we invert and otherwise we leave the result unchanged. This can be done by two-to-one multiplexers but also by XOR gates. A string that is bit-wise XOR'ed with `1`s is inverted while a string that is XOR'ed with `0`s remains the same.

So, a generalized equation for end-around carry addition is

$$|\mathrm{R}| = \Sigma \oplus \Delta$$

A possible implementation of end-around carry addition is shown in Figure 5.6.

5.4.2 Sign-Bit

Knowing the effective operation and the carry out of the magnitude is sufficient to determine the sign-bit of the result (R_{sign}). At the beginning of this section a number of different cases were shown that separated effective addition from effective subtraction. For effective addition (-A + -B or A + B), the sign-bit of the result is equal to that of input A. For effective subtraction, the sign-bit may change depending on the magnitude of A and B. If |A| > |B|, the sign-bit still equals the sign of input A. When |A| < |B| the sign will change. To illustrate this, a few small examples are shown below:

Computation	Effective Operation and Magnitude	Resulting Sign
$2 + \ 1 = \ 3$	addition	positive (unchanged)
$-2 + -1 = -3$	addition	negative (unchanged)
$2 + -1 = \ 1$	subtraction (|A| > |B|)	positive (unchanged)
$-2 + \ 1 = -1$	subtraction (|A| > |B|)	negative (unchanged)
$2 + -3 = -1$	subtraction (|A| < |B|)	negative (changed)
$-2 + \ 3 = \ 1$	subtraction (|A| < |B|)	positive (changed)

To find the resulting sign-bit, the sign-bit of A×B is combined with the effective operation and the carry out of the adder. The sign of A×B is used as a base for the resulting sign. The effective operation is

straightforward to determine. First the sign-bit of the A×B product can be found by $A_{sign} \oplus B_{sign}$. Now if the sign-bit of C equals that of the product, the computation is effective addition and otherwise effective subtraction. Notice the second exclusive or relation, the effective operation is found by $((A_{sign} \oplus B_{sign}) \oplus C_{sign})$. If effective operation (`1` for effective subtraction and `0` for effective addition) is addition, the resulting sign-bit is just the sign of A×B, otherwise it may or may not be swapped depending on whether |A×B| > |B| ot not. Remember that the carry out (C_{out}) of the end-around carry adder is `1` if |A| > |B| and `0` if |A| > |B|, hence the sign-bit of the result (R_{sign}) is capture by Equation 5.5.

$$R_{sign} = (A_{sign} \oplus B_{sign}) \oplus \text{effective operation} \wedge \overline{C_{out}} \tag{5.5}$$

This equation can also be written as shown in Equation 5.6. This form can directly be translated into hardware.

$$R_{sign} = (A_{sign} \oplus B_{sign}) \oplus ((A_{sign} \oplus B_{sign}) \oplus C_{sign}) \wedge \overline{C_{out}} \tag{5.6}$$

Assuming the unique zero representation of the IEEE-754 format, both the sign-bit and the magnitude are always zero (except when round to negative infinity is used). Equation 5.6 does not always produce the correct sign-bit when the magnitude is zero. To include the unique zero representation Equation 5.7 must be used.

$$R_{sign} = (A_{sign} \oplus B_{sign}) \oplus ((A_{sign} \oplus B_{sign}) \oplus C_{sign}) \wedge \overline{C_{out}} \wedge \overline{R_{zero}} \tag{5.7}$$

This requires a check for zero at the very last stage of the floating-point computation. Since FMA is only part of the IEEE-754 standard since 2008, the sign-bit for multiply-add operations has not always settled correctly yet. One example is the newly adopted floating-point standard for VHDL-2008 [36]. More common floating-point mistakes and in particular the ones regarding the sign-bit are discussed in Appendix B.

5.5 Normalization

In the final stage, after having multiplied A with B and after having added C, the result will most likely have to be normalized and rounded. Normally these operations contribute to the critical path. We can improve the situation by using a technique called LZA. A prediction of the number of leading zeros that can be performed in parallel with addition.

5.5.1 Leading Zero Anticipation

In Chapter 2 we discussed the benefits of having normalized numbers. All normalized floating-point numbers must have a '1' as leading significand bit. In most cases, an intermediate result is not normalized. Hence, a floating-point unit must be able to normalize its results. The normalization process involves detection of *leading zeros* (discussed in 5.5.2). Leading zero detection quickly becomes part of the critical path due to the fact that it depends on the outcome of the adder, which in itself entails a considerable latency. leading zero anticipation tackles this problem by predicting the number of leading zeros in parallel with addition.

Leading zero anticipation algorithms operate on the same input (A,B) as the adder. The result is a string of indicators $\texttt{0}^k\texttt{1x}^*$ where each indicator in position i indicates if the bit on that position possibly is the leading one. Here k represents zero or more instances of `0` and x* is either zero or one followed

A	0	0	0	0	1	0	1	1	0	0	0	1	1	1	1	0	0	0
B	0	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	1	0
LZA pattern	Z	Z	Z	Z	T	T	T	T	Z	Z	T	G	G	G	G	Z	T	Z
A + B	0	0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	1	0
Leading one	-	-	-	-	X	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 5.1: Leading zero anticipation example, case I [30]

by zero or more instances of x. The first '1' in the indicators string marks the position of the leading '1' after addition. Kershaw et al. [28] first recognized that such a string of indicators could be derived from the same input as the adder. Knowles further formalized the equations in [37] by providing the truth table to set each indicator i. Different variations of LZA exist as discussed in [29], however they are all based on the same principles. LZAs make use of a propagate (T), a generate (G) and a kill (Z) function. These functions are defined as

$$\mathrm{T} = \mathrm{A} \oplus \mathrm{B} \tag{5.8}$$

$$\mathrm{G} = \mathrm{A} \wedge \mathrm{B} \quad (\mathrm{A}\ \mathrm{B}) \tag{5.9}$$

$$\mathrm{Z} = \overline{\mathrm{A}} \wedge \overline{\mathrm{B}} \quad (\overline{\mathrm{A}}\ \overline{\mathrm{B}}) \tag{5.10}$$

As the names already suggest, these functions look for carry generation, carry termination and carry propagation. They help find patterns in the input (tuples (A_{i-1},B_{i-1}), (A_i,B_i), (A_{i+1},B_{i+1}) per indicator i) that generate the appropriate indicator bit for position i.

If the adder is able to perform addition on signed numbers, the LZA algorithm becomes a bit more complicated. It should not only be able to detect leading zeros, but also leading ones (see Section 5.4.1). Several cases can be distinguished in which different patterns have to be matched to find the leading one.

Case 1 A > 0, B > 0, Result > 0
Case 2 A < 0, B < 0, Result < 0
Case 3 A and B have different signs, Result > 0
Case 4 A and B have different signs, Result < 0

Schwarz illustrates the pattern matching in [30], chapter 8. For case 1, the following pattern must precede the leading one: Z^{+}**:**$\overline{\mathrm{Z}}$ (i.e., one or more instances of Z followed by one instance of Z inverted). An example that corresponds to this case is shown in Table 5.1. In case 2, A and B are both negative numbers. This case is not relevant for FMA units because the result of A×B is always positive (only the magnitude is multiplied). However, for completeness this case is included. The following pattern can always be found in case 2, G^{+}**:**$\overline{\mathrm{G}}$. In case 3, A and B have opposite signs and the result is positive. T^{+}**:**G**:**Z**:**$\overline{\mathrm{Z}}$ is the pattern to look for here. The last case is when A and B have opposite signs and the result is negative. In this case we have to find the pattern T^{+}**:**Z**:**G^{*}**:**$\overline{\mathrm{G}}$.

All cases considered, a predictor equation can be derived that produces a string of indicators that correctly predicts for both positive and negative results. The equation has to work for all cases and all cases must exclude each other. After some boolean manipulation, Schwarz derives the following equation:

$$\mathrm{f}_i = (\overline{\mathrm{T}_{i-1}}\mathrm{Z}_i\overline{\mathrm{Z}_{i+1}}) \vee (\overline{\mathrm{T}_{i-1}}\mathrm{G}_i\overline{\mathrm{Z}_{i+1}}) \vee (\overline{\mathrm{T}_{i-1}}\mathrm{G}_i\overline{\mathrm{Z}_{i+1}}) \vee (\overline{\mathrm{T}_{i-1}}\mathrm{G}_i\overline{\mathrm{G}_{i+1}}) \qquad (i > 0) \tag{5.11}$$

However, in a more compact notation, the LZA equation can be written as:

$$f_i = \begin{cases} \overline{T_0} T_1 & i = 0 \\ T_{i-1}(G_i \overline{Z_{i+1}} \vee Z_i \overline{G_{i+1}}) \vee \overline{T_{i-1}}(Z_i \overline{Z_{i+1}} \vee G_i \overline{G_{i+1}}) & i > 0 \end{cases} \tag{5.12}$$

This equation easily translates to hardware.

Mispredictions

The predicted leading bit may be off by one position. If this is the case, then it can be proven that it is always off one position to the right of the actual leading one. This is a very regular error, which can easily be corrected during the last shift in the normalization stage. Other LZA techniques exist that produce exact results [29], however they always trade off area for correctness and performance (i.e., exact solutions require much more hardware). Such solutions are only interesting for LZA instruction in fixed-point processors, or when the adder latency entirely overlaps the latency of the exact LZA and the normalization stage is a critical path.

5.5.2 Leading Zero Detection

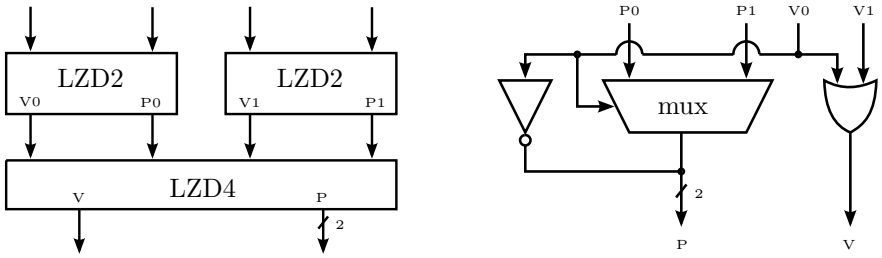
LZA as discussed so far only predicts where the leading one will be located (approximately). Whether or not LZA is incorporated in a floating-point unit, the number of leading zeros still needs to be counted. leading zero detection (LZD) algorithms are used to actually count the number of leading zeros. Usually this is done by encoding the indicator string or addition result into a binary representation of the number of leading zeros. In the known literature, LZD can be categorized into two distinct methods. One method is based on the creation of a monotonic string of zeros followed by ones. The other is based on a hierarchical tree structure.

LZD Alternatives

In monotonic string methods, the input is first preprocessed such that the input string becomes of the form 0^*1^+ (monotonic). This preprocessing is usually implemented by OR'ing and AND'ing specific parts of the input in combination with manchester carry, or look-ahead speedup techniques. Once a monotonic string is found, it is encoded using logic expressions that are restricted to AND and OR operations. Kershaw [28] provides a more detailed description of the monotonic string LZD method, we will not go into details here.

The other method for LZD is based on tree structures. A string on n inputs is first divided into $n/2$ adjacent pairs. For each pair a leading zero count is performed. At the next level the output is multiplexed into a 4-bit leading zero count and so on. This continues for $log_2(n)$ levels until a count for the entire input has been found.

The monotonic string method is faster than a tree structure if a manufacturing process is used that allows extra wide AND an OR gates. However, this method is less attractive for low-power technology [29]. If LZD is not part of the critical path (which is often the case when combined with LZA), the hierarchical tree structure is the preferred choice.



(a) Four bit LZD circuit (b) LZD4 logic

Figure 5.7

LZD by Tree Structures

Despite the fact that energy efficiency of monotonic string methods has improved [38], we believe that the tree structured LZD circuit still offers the best solution for low-cost, low-power architectures. A well-known tree structured LZD solution is presented by Oklobdzija in [1]. In his work he describes a scalable algorithm for generating LZD circuits. In the remainder of this section, his work is explained.

At the most basic level, a two bit leading zero count circuit is defined. The input for this circuit is a two bit pattern that can be either `1X`, `01`, `00` (`X` is don't care). The output also consists of two bits; one to indicate the position of the leading `1` (which is the same as the number of leading zeros) and a valid bit that indicates if there was a `1` present in the input. The truth table for this simple circuit is shown in 5.2. This logic easily translates to hardware by using multiplexers or even directly in logic gates.

Pattern	Position	Valid
1X	0	1 (yes)
01	1	1 (yes)
00	X	0 (no)

Table 5.2: Two bit LZA truth table

The 2-bit case can be extended to a four bit counter in a hierarchical way. Two 2-bit counters are connected to form a 4-bit counter, one counter for the two MSBs (left) and one counter for the MSBs (right). Let P0 indicate the position of the leading `1` in the two MSBs and P1 of the LSBs. V0 will be the valid bit for P0 and V1 for P1. The two 2-bit LZD circuits are connected with four bit LZD logic (LZD4) as shown in Figure 5.7(a). The logic for the lower LZD4 block is shown in Figure 5.7(b) and in truth-table form in Table 5.3. A four bit variant can also be implemented directly in logic gates. However, multiplexers are generally faster.

Pattern	Position (binary)	V0 Valid	Position (logical)	Valid
1011	0 (00)	1 (yes)	$\overline{V0}P0$	1 (yes)
0100	1 (01)	1 (yes)	$\overline{V0}P0$	1 (yes)
0011	2 (10)	0 (no)	$\overline{V0}P1$	1 (yes)
0001	3 (11)	0 (no)	$\overline{V0}P1$	1 (yes)
0000	XX	0 (no)	0 (no)	XX

Table 5.3: Four bit LZA truth table

Now we can also take two groups of four bits and connect them together for an an 8-bit leading zero detector. We can go further by connecting two 8-bit counters for a 16-bit counter and so on. The generalized algorithm for creating 2^n wide LZDs is shown in Algorithm 5.5.1

This algorithm is scalable for input that is a power of two. Input that is not a power of two will have to be padded with zeros. In the worst case, this means that the leading zero detector is almost twice as large as required (e.g., for input of length 33 we need a 64-bit counter). With some minor modifications, this circuit can be made more area-efficient and support any length of input that is even. The worst case can be reduced to one bit overhead. These modifications are discussed in Chapter 6.

Once the number of leading zeros is determined, the last step of normalization is trivial. The number of leading zeros is the amount of positions to shift the significand to the left and also the number to subtract from the intermediate exponent.

Handling the Error of Inexact LZA

Earlier we mentioned that LZA is not always exact. In some cases, the actual leading `1` is located one position to the right. Although solutions have been proposed for exact LZA, there is always a trade off between area/energy efficiency and guaranteed correctness. Exact solutions (e.g., [39, 38, 29] eliminate the need for correction and thus improve latency, however, the latency reduction often comes at the cost of a considerable area increase. In many cases where area is of importance, postponed correction is much more attractive because it can be implemented with considerable less hardware. The price one pays for misprediction is not very high. A two-to-one multiplexer in terms of latency, and an adder or subtracter for exponent correction; insignificant compared to the cost of exact LZA.

Correction after misprediction is easily achieved. After shifting the significand to the left by the amount resulting from counting the leading zeros in the indicator vector, a simple check on the MSB of the 'normalized' significand reveals the error. If the MSB is `1`, then the prediction was correct and no further action is required. If the MSB is `0`, then the prediction was incorrect and the significand needs to be shifted one more position to the left, and the exponent decremented. In terms of hardware this can be realized by using two exponent subtracters in parallel (one that subtracts the amount predicted by the LZA and one that subtract the same amount plus one) and selecting the result based on the MSB of the shifted significand. Another solution would be to conditionally increment the amount to subtract.

Algorithm 5.5.1 Leading zero detection

1: Form a pair of bits B_i, B_{i+1} for $0 \leq i \leq (n-1)$.
2: Determine the P and V bit for each pair.
3: **while** depth $< log_2\{\text{n}\}$ **do**
4: Determine the P_{next} and V_{next} bits from the P and V bits as follows:
5: $V_{next} = V_{left} \vee V_{right}$
6: **if** $V_{left} =$ `1` **then**
7: $P_{next} =$ `0` & P_{left} (concatenation)
8: **else if** $V_{right} =$ `1` **then**
9: $P_{next} =$ `1` & P_{right} (concatenation)
10: **else**
11: $V_{next} =$ `0`
12: **end if**
13: **end while**

5.6 Rounding

The normalized results often need to be rounded because most numbers can not be represented exactly in floating-point representation. Rounding is nothing more than choosing between the two representable numbers that are closest to the exact answer of the operation. This means the intermediate result obtained after normalization is either incremented or truncated. In principle a simple operation that requires only one adder. The real difficulty of rounding floating-point numbers is making the decision to increment or truncate. Several types of rounding exist. Always truncate, always increment or truncate in one case and increment in another. The preferred type of rounding strongly depends on the application. For this reason, the IEEE-754 standard for floating-point arithmetic defines four rounding modes.

- Round to zero
- Round to minus infinity
- Round to infinity
- Round to nearest even

To comply with the IEEE standard, all these rounding modes have to be implemented. In addition, the final result must be computed as if it was performed in infinite precision and then rounded. Algorithms for IEEE rounding have been explored extensively, for example in [40], [30] and [31]. Most of them are based on the concept of guard, round, and sticky bits as explained in Chapter 2 and at the beginning of this chapter. With only three additional bits, all floating-point arithmetic can be rounded as if it was computed with infinite precision [9]. The guard and round-bit are nothing more than a two bit extension of the datapath, i.e., all computations are performed with two extra bits of width. The sticky-bit was explained in Section 5.2.1.

Round to Nearest Even

Perhaps the mostly used rounding mode in IEEE floating-point arithmetic is round to nearest even. In this rounding model all numbers are rounded to the nearest representation, and if there is no nearest representation (if we are exactly in between), the result is rounded to the nearest even number. For this round mode, the round-bit, the guard-bit, the sticky-bit and the LSB of the normalized result must be know. A possible rounding algorithm is shown in Algorithm 5.6.1 [31].

Once the guard, round and sticky-bit are know, round to nearest even is pretty straightforward. The guard-bit is the first bit after the result's LSB. If this bit is `1`, then the discarded bits (including the guard-bit itself) have at least half the weight of the LSB (i.e., one ULP). If not, then the nearest

Algorithm 5.6.1 IEEE-754 round to nearest even

Input: guard-bit **G**, round-bit **R**, sticky-bit **S**, the normalized result truncated up to the original input width plus the hidden-bit $Significand_{normalized}$, and the least significand bit of the normalized result **LSB**

Output: $Significand_{rounded}$

1: **if** G = `0` **then**
2: $Significand_{rounded} \leftarrow Significand_{normalized}$
3: **else if** R = `1` $\vee$ S = `1` **then**
4: $Significand_{rounded} \leftarrow Significand_{normalized} + 1$
5: **else if** LSB = `0` **then**
6: $Significand_{rounded} \leftarrow Significand_{normalized}$
7: **else**
8: $Significand_{rounded} \leftarrow Significand_{normalized} + 1$
9: **end if**

representable number is always the truncated normalized result. For all other cases the round and sticky-bit have to be examined to determine if the nearest or nearest even numbers needs to be selected. If either the round or sticky-bit is `1` in combination with the guard-bit being `1`, the discarded bits have more weight than half a ULP and the result will have to be incremented.

If neither of the above cases is true, the algorithm must find the nearest even number. The algorithm by Park et al. [31], uses the LSB of the unrounded result. If neither the guard, round or sticky-bit is `1`, the result is truncated and otherwise incremented.

Round to Infinity

In a similar way round to ∞ can be described. IEEE-754 defined both round to $+\infty$ and round to $-\infty$. Algorithm 5.6.1 shows how to implement round to $+\infty$. To implement both round the $+\infty$ and $-\infty$, the sign-bit must be added as an additional parameter. For rounding to $-\infty$, the procedure shown in Algorithm 5.6.1 must be inverted. When normally we would increment, we truncate and vice versa.

Algorithm 5.6.2 IEEE-754 round to infinity

Input: guard-bit **G**, round-bit **R**, sticky-bit **S**, the normalized result truncated up to the original input width plus the hidden-bit $Significand_{normalized}$, and the least significand bit of the normalized rersult **LSB**
Output: $Significand_{rounded}$
1: **if** G = `1` $\vee$ R = `1` $\vee$ S = `1` **then**
2: $Significand_{rounded} \leftarrow Significand_{normalized} + 1$ (round)
3: **else**
4: $Significand_{rounded} \leftarrow Significand_{normalized}$ (truncate)
5: **end if**

Round to Zero

The simplest rounding mode is truncation, known as round to zero in the IEEE standard for floating-point arithmetic. This rounding mode always truncates the normalized result, regardless of the state of the guard, round and sticky-bit. Because this mode is so simple and does not require any additional hardware to implement, some high-speed floating-point units choose to support only this rounding. The SPEs in the Cell Processor (Section 3.4) are a good example of this approach.

5.7 Summary

We have shown a number of techniques that can be used to build efficient floating-point units. From efficient alignment of operands to rounding that is compliant with IEEE-754. These techniques can not blindly be applied to any floating-point datapath. For example the alignment of C as described in 5.2 results in a floating-point representation three times the size of the original input. To align this with the product of A and B, operand extension is mandatory. In other cases, such as the LZD circuit, the techniques are described in such a way that they are independent of the floating-point precision. When a supported format has been decided upon, the circuits can often be further optimized for area or speed. In the next chapter we will discuss such implementation details. We will then also discuss how to maximize the use of the floating-point hardware by re-using it for integer operations.

6 Implementation

6.1 Introduction

Chapter 4 outlined a (multiply-add) datapath for shared integer and floating-point functionality. Chapter 5 mainly focused on optimizing the datapath for high speed by presenting techniques for local improvements to the architecture. In this chapter, we finalize the design by describing how the optimizations, basic hardware blocks and control are combined into a fast and area-efficient design. A detailed block diagram of the augmented integer floating-point unit is shown in Figure 6.1.

Despite the fact that this diagram only shows dataflow and omits the necessary control, it gives a good impression of the complexity involved in designing an efficient datapath for floating-point and integer arithmetic. To avoid losing overview, the datapath is divided into seven sectors:

- Input formatting and instruction decoding
  Instruction Decoder/Input Formatter
- Alignment and exponent adjustment
  Exponent Adjustment and Shift Count, Shift Right, Sticky-bit
- Comparing operands
  Compare
- Fused Multiplication-Addition
  Partial Product Multiplier, Conditional Complement, Carry-Save Adder, Carry-Propagate Adder, Conditional Re-Complement
- Normalization
  Leading Zero Anticipation, Leading Zero Detection, Normalize
- Rounding
  Sticky-bit, Round
- Output formatting and signaling exceptions
  Output Formatter

For each sector, the (non-trivial) implementation details are discussed per hardware block. Their functionality and purpose should be clear by now. Therefore, every section only provides a minimalistic introduction to the functionality. We start at the input formatter in first stage and follow the dataflow through the datapath until the output formatter is reached. Emphasis is put on how components have to be modified to support both two's complement input for integer arithmetic and unsigned input for floating-point arithmetic. Every components is implemented such that the area is minimized without affecting the performance obtained from the optimizations described in Chapter 5

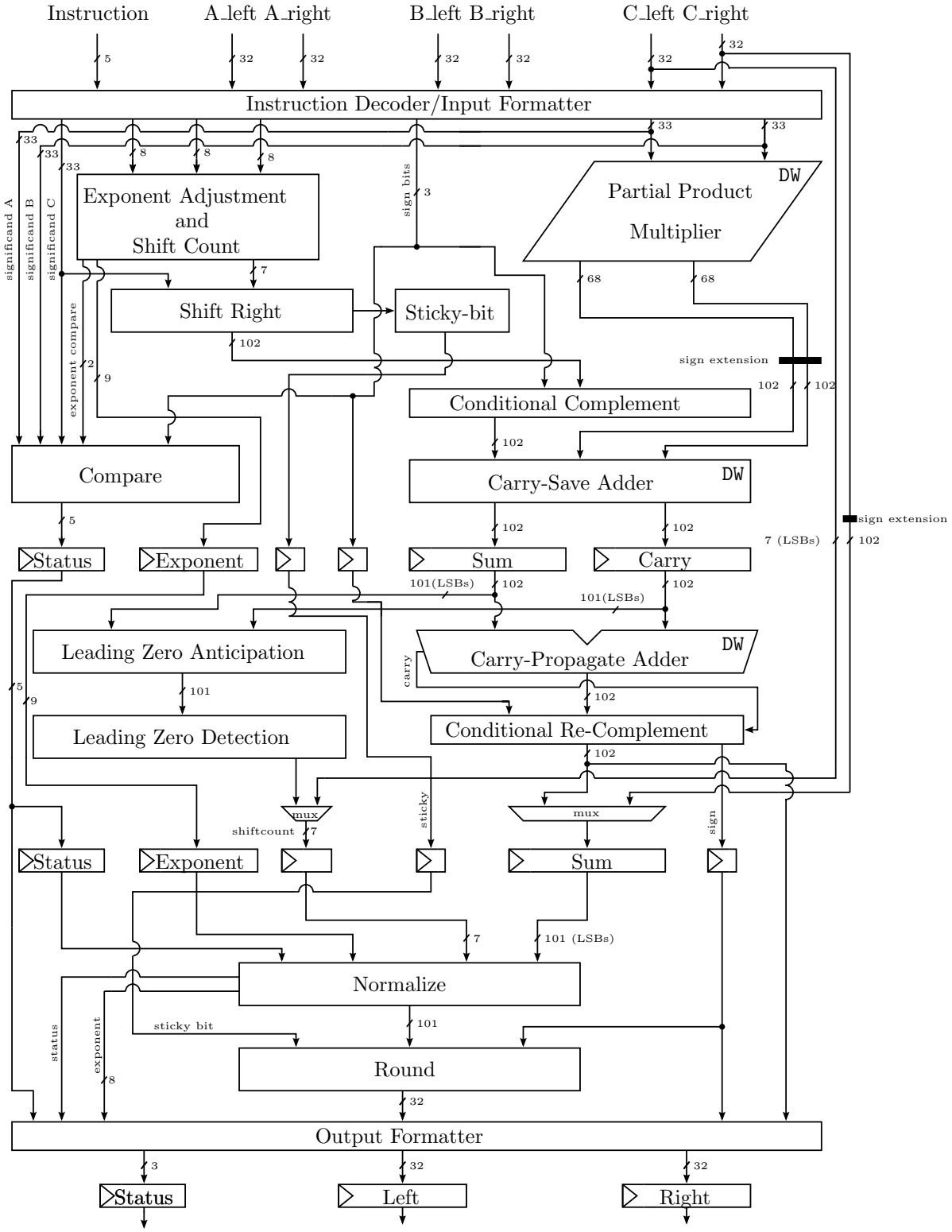


Figure 6.1: Detailed datapath overview (DW indicates Synopsis DesignWare component)

6.2 Input Formatting and Instruction Decoding

At the very start of the first stage in the ALU, all operands pass through the instruction decoder and undergo a slight transformation to accelerate processing and guarantee correct output. The instruction decoder performs three tasks:

- Decoding opcodes
- Extending operands
- Checking for zeros

6.2.1 Opcode Decoding

The ALU supports seven floating-point instructions and six integer instructions in hardware. To differentiate the instructions, a 5-bit opcode is used (one spare bit for future extension of the instruction set).

Opcode	Encoding	Type	Operation
FMAN	00001	Float	Multiply-Add round to nearest even
FMAZ	00010	Float	Multiply-Add round to zero
FMAP	00011	Float	Multiply-Add round to positive infinity
FMAM	00100	Float	Multiply-Add round to negative infinity
FLTV	00101	Float	Compare, less than
FGTV	00110	Float	Compare, greater than
FETV	00111	Float	Compare, equal to
IMAC	10000	Int	Multiply-Accumulate
ISLV	10001	Int	Arithmetic shift left
ISRV	10011	Int	Arithmetic shift right
ILTV	10101	Int	Compare, less than
IETV	10110	Int	Compare, greater than
IGTV	10100	Int	Compare, equal to

Table 6.1: ALU opcodes

As explained in the previous chapter, each instruction requires a different dataflow through the datapath. `FMAN` instructions for example, must flow through the normalization and rounding blocks while its integer counterpart `IMAC` can skip the entire last stage. To control dataflow in the ALU each opcode is decoded into control signals that (re)configure the ALU such that the dataflow corresponds to its instruction.

The opcodes are translated into four control signals that determine the initial configuration of the datapath. One to differentiate between floating-point and integer instructions, one to differentiate arithmetic from logic instructions and another to differentiate shift instructions from the rest. The last control signal is a 2-bit round mode encoding, used in the last stage to configure the floating-point rounding logic. Decoding the opcodes into control signals is straightforward to implement by multiplexers, we will therefore not discuss this in detail.

6.2.2 Operand Extending

Every IEEE floating-point number includes a *hidden-bit*. Because normalized numbers always start with a `1`, this bit does not have to be stored in memory. By assuming this bit implicitly, the precision of floating-point numbers can be extended by one additional bit, without the need for larger registers.

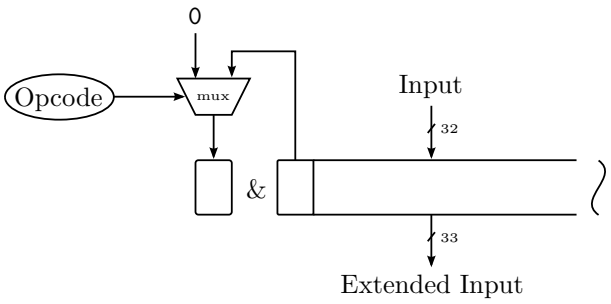


Figure 6.2: Operand extension

However, in order to perform arithmetic on such numbers, the hidden-bit first needs to be made explicit again. The ALU we present does not support denormalized numbers. Consequently, for every floating-point instruction, the instruction decoder extends all 32-bit significands to 33 bits by placing a `1` in front of the MSB (except when the operand is zero, see Section 6.2.3). For a datapath that is purely intended for floating-point arithmetic, this simply requires that the new MSB is tied to the logical 'high' value (V_{dd}). However, the same input is also used to perform integer arithmetic. A 32-bit two's complement notation is used to represent integer numbers which does not require any extension of the operand. Nevertheless the floating-point datapath is 33-bit, forcing us to extend the integer operands as well. To solve this problem, the two's complement integer operands can be extended by sign extension. Sign extension means that the sign-bit is duplicated.

Every operand that goes into the ALU is extended in the instruction decoder by concatenating (&) one additional bit to the left of the MSB. The value of this bit is determined by the opcode as depicted in Figure 6.2.

6.2.3 Checking for Zero

Zero is an exceptional case in the IEEE floating-point format. Unlike any other valid number, zero does not use a biased exponent. This causes all sorts of problems that are discussed more detailed in Appendix B. For example 0×M is expected to return 0 [6]. When no action is taken, one will find that such computations will incorrectly underflow. The problem can be explained by the fact that *bias accumulation* is compensated by the datapath (Section 5.2). When the operand is zero, there is no bias so compensating (subtracting the bias) will not produce the correct result. The exponent can even become negative, which is not allowed, hence the underflow exception will be raised.

To prevent problems like this, every operand is checked for zero when they arrive at the instruction decoder/input formatter. Based on the produced zero-check control signal, measures can be taken to prevent incorrect *zero-arithmetic*. The exponent of any floating-point number that is in accordance with IEEE-754, is zero if and only if the operand is zero. We can benefit from this property by checking only the exponents of the input. If an exponent is zero then the operand is zero, otherwise the input would be invalid (denormalized numbers are not supported). Consequently, the zero checks require only three 8-bit comparators. Moreover, the comparators only have to check for `0`s which is inexpensive to implement in hardware. For each operand there is a separate zero check control signal.

Note: some of these problems may be resolved by converting all exponents to two's complement. This is however not very area-efficient and adds more delay than a comparator.

6.3 Alignment Shift and Exponent Adjustment

After passing through the instruction decoder/operand formatter block, operand C is aligned with the product while operands A and B are multiplied to form the product itself (Figure 6.3). Alignment of two floating-point numbers consists of two actions: adjusting the exponent of one operand to match the exponent of the other, and shifting the significand of that same operand to match the new exponent. Adjustment of the exponents is often not discussed in publications related to floating-point arithmetic. This is mostly because it is assumed to be trivial. In a purely mathematically sense this is true, viz., multiplication is just addition of the exponents and for floating-point addition/subtraction the exponent simply equals the exponent of the operand that is not shifted. In binary floating-point arithmetic however, we have to deal with overflow, underflow and non-regular (e.g., zero and infinity) input. Especially in the case of FMA, exponent adjustment is anything but trivial.

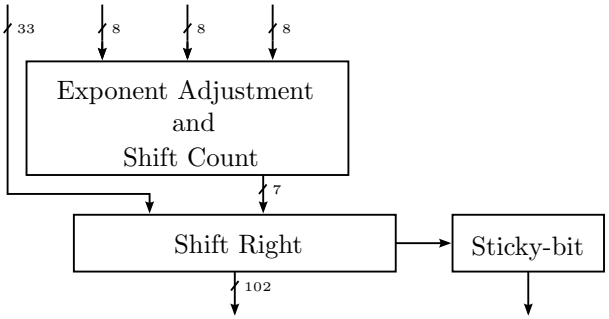


Figure 6.3: Operand alignment sector

6.3.1 Exponent Adjustment

The IEEE-754 floating-point interchange formats specify that the exponent is biased. This choice was made for historical reasons because comparing biased numbers by hand is a lot easier and quicker than comparing for example two's complement numbers. Unfortunately biased notation entails a few very unfortunate adverse properties for arithmetic implementation. Some of these properties include the accumulation of the bias during addition and the difficulty of underflow detection. A solution that is often implicitly assumed is to convert the exponent to a signed representation. In a signed representation (usually two's complement) exponents can be added and subtracted without having to worry about bias accumulation and underflow is easier to detect because the sign-bit inverts. However, we have chosen not to convert the exponent to a signed representation for two reasons:

- Conversion would add additional delay to the critical path of the first stage. All exponents would have to be converted before the actual computation starts, making it very difficult to overlap the conversion delay with other processing delays.
- If we assume two's complement representation, conversion would require a converter consisting of an inverter a subtracter and an incrementer. In terms of area and energy efficiency, this is costly.

The solution we propose is to keep using the biased notation and extend the exponent by one bit. This means we still have to compensate for bias accumulation, but at least this can be done in parallel with the significand multiplication. Because we extended the exponent by one bit, overflow is easily detected. When after addition, either the MSB is `1` or all the other bits are `1`, the intermediate exponent has exceeded the maximum representable number (Table 2.3). Both cases are easily detectable. The MSB OR'ed with the AND-reduced LSBs provides a 1-bit overflow control signal (`1` meaning overflow).

In FMA, overflow handling is a little more complicated than multiplication or addition separately. During multiply-add operations a third operand is added or subtracted. This could very well mean that although the intermediate result of A×B overflows, the final result of $A \times B - C$ is within the representable range of the floating-point unit. By using one more bit in the exponent, we can continue normally without checking for overflow, because two 8-bit biased exponents can not overflow in a 9-bit representation. Besides the addition/subtraction of C, normalization could also prevent the final result from overflowing. This even holds for non-FMA floating-point units. Only when the exponent still exceeds the maximum after normalization will the result truly overflow, hence overflow detection is postponed until after normalization. It should even be postponed until after rounding because the result may still overflow during round-off (Section 6.7).

Underflow is even more difficult to detect than overflow. A floating-point number underflows when its *biased* exponent is smaller than or equal to zero. Assuming that none of the exponents is initially zero, the intermediate exponent may underflow during compensation for bias accumulation. For example, multiplication of two floating-point numbers with the minimal exponent of -126 (`00000001` in binary bias-127 notation) yields -125 (`00000010`). This result now includes the bias twice, so we have to subtract the bias to find the actual exponent. Obviously 127 is much larger than the exponent itself so the result will underflow. Detection of underflow in the result itself is very difficult. It is much easier to predict if the result will underflow based on the intermediate result (before subtracting the bias). If this result is smaller than or equal to the bias (or as explained shortly, the bias minus a constant offset for FMA), we have detected underflow. The only requirement is a 9-bit comparator. The cost of this comparator can be justified by sharing it with floating-point compare instructions. These instructions require an exponent comparator regardless of the exponent representation that is used.

Unlike overflow, normalization can not recover the result from underflowing. However, in FMA the addition of C could mean the final result does not underflow. If a floating-point computation underflows, the result is usually represented by zero [6]. Thus adding C to the product results in C again, unless C itself is also zero. Therefore based on the zero-check control signals retrieved from the instruction decoder, either the C operand is forwarded to the next stage or the result really underflows (when C is zero) and the status bits are asserted to the underflow status.

Now that the overflow and underflow detection is settled, we can focus on the dataflow of exponent adjustment itself. The main part of the exponent aligner is depicted in Figure 6.4.

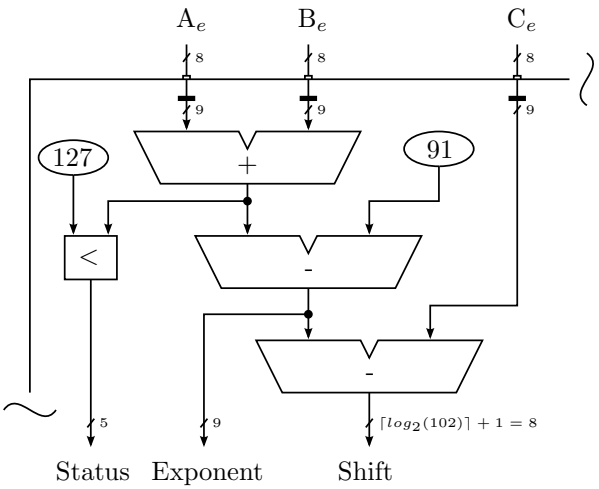


Figure 6.4: Exponent alignment and shift count

This flow is based on Equation 5.2. The equation states that after adding the exponents of A and B, the bias must be subtracted and an implementation specific offset must be added. The bias and the offset are both constant so Equation 5.2 can be simplified to one addition and one subtraction. The bias for

an 8-bit exponent is 127 and the offset for a significand of 32 bits with a guard and round-bit is 36 (1 hidden-bit + 32 significand bits + 1 guard-bit + 1 Round-bit + 1 multiply compensation):

$$\begin{aligned} \text{Exponent} &= A_e + B_e - 127 + 36 \\ &= A_e + B_e - 91 \end{aligned} \tag{6.1}$$

Equation 6.1 is implemented by adding the exponent of A and B in a 9-bit adder and subtracting the constant in a 9-bit subtracter as shown in Figure 6.4. The result is also used to drive the shifter that aligns the significands.

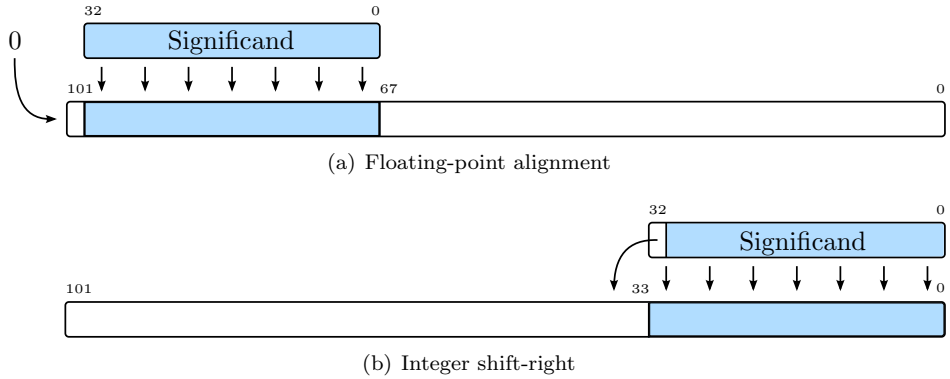
6.3.2 Significand Shift

Section 5.2 explained how C can efficiently be aligned with the product by placing C entirely in front of A×B. Equation 5.1 was derived to find a right-shift count for C. This equation, that actually describes the absolute difference between the exponents of A×B and C, only slightly differers from the one used to find the exponent. By subtracting C_e we find the shift count needed to drive a shifter that aligns the significands. Because the exponent of A×B is already known at this point, the required shift count is straightforward to implement. The exponent of C is however not a constant, so another subtracter is needed, as shown in Figure 6.4.

Integer Re-use

The alignment optimization discussed in Section 5.2 minimally requires a 101-bit shifter (3×33+2). This shifter needs to be a logical shifter to ensure that `0`s are shifted in from the left. However, to save area, we want to use the same shifter for integer shift-right instructions. To be able to shift two's complement integer input in a meaningful way, an arithmetic shifter (Figure 4.5(a)) is required. Arithmetic shifting is not suitable for floating-point alignment because the MSB of the normalized input is `1`. To facilitate both integer and floating-point shift, an arithmetic shifter is used but it is extended to 102 bits. For floating-point alignment the MSB is asserted `0` with the significand placed immediately to the right as shown in Figure 6.5(a). The arithmetic shifter now shifts in zeros as desired for floating-point alignment. The integer operands can of course be sign extended as described earlier.

Integer input is mapped differently to the shifter to aid datapath regularity. In Chapter 4 we agreed to set the duration of every integer instruction to two clock cycles. Because the shifter produces a result



(a) Floating-point alignment

(b) Integer shift-right

Figure 6.5: Input mapping to right shifter

every clock cycle, an additional register and control would be needed to temporarily store the outcome of the shift instruction and route it to the output of the ALU. For area-efficiency reasons, the output of the shifter is instead routed through the adder. The outcome is effectively the same if one or both inputs to the multiplier are zero.

$$(\mathrm{A} \gg n) = 0 \times 0 + (\mathrm{A} \gg n)$$

Theoretically the shift count can be 253 for bias-127 exponents. With such large shifts the C operand will have no effect on the product of A and B. The shifter itself can only shift 102 positions, so a 7-bit ($\lceil log_2(102) \rceil$) shift count is sufficient to drive the shifter. However eight bits (256 positions) are used to represent the shift amount, because this is convenient for sticky-bit calculation (Section 5.2.1). Any amount that is larger than 102 will be 'saturated' by the shifter. The shifter will shift out C entirely so it does not affect the result. A negative shift (detected by another comparator not shown in Figure 6.4) indicates if A×B is too small to affect C. In this case we overwrite the shift count with zero. C remains unaffected and will flow through the datapath to the output of the ALU.

Facilitating Compare Instructions

We already shortly mentioned that the comparator used for underflow detection is also used to compare exponents for floating-point compare instructions. The input of the comparator is multiplexed between A_e and C_e, *and* B_e and 127 (the bias) respectively. Both multiplexers are controlled by a 1-bit signal, obtained by the instruction decoder, that differentiates arithmetic from logic operations. Once the multiplexers are configured for logical compare operations, the output of the comparator indicates if the exponent of A is *smaller than* the exponent of B (`1` if true and `0` if false). For a full comparator, *greater than* and *equal to* are also mandatory. If one of the two is known then the third can be derived. In hardware the latter requires less area and is therefore implemented. The exponent adjustment block produces two 2-bit encoding for further comparison in the Compare block. The MSB encodes the logic value for 'less than' (`1` meaning true) and the LSB encodes 'equals' (`1` meaning true).

6.3.3 Sticky-Bit

The *primary sticky-bit* (inexactness caused by alignment) is computed by special sticky-bit logic. Algorithm 5.2.1 was derived for efficient OR reduction of the bits shifted out during alignment. The actual implementation does not differ from the algorithm. A 66-bit shifter is driven by a subtracter that subtracts an implementation specific constant. In this case 68; which is two times the extended significand width plus the guard and round bits. The shifter output is OR reduced by a tree of OR-gates.

6.4 Comparing Operands

Comparing floating-point operands is very different from comparing integers. For a full floating-point comparison, the exponents, significands and sign bits have to be evaluated individually. For two's complement integer compare, the sign and magnitude have to be compared together. Comparing in two's complement notation is more difficult than sign magnitude. As mentioned before, to support the full range of logical compares (i.e., 'less than', 'greater than' and 'equal to'), only two out of three have to be implemented. The third can be derived from the other two. Since 'less than' and 'equal to' require the least area, these are the compares that are physically present in the compare block.

Let us first focus on floating-point less than (`FLTV(A,B)`). Obviously when A is negative and B positive, A is smaller than B. There is however one exception:

IEEE-754 states that "*Comparisons shall ignore the sign of zero (so +0 = -0)*"

Hence, if the zero check control signals from the input formatter indicate that A and B are zero, the outcome is forced to equal (false from a 'less than' viewpoint). If both signs are equal, the exponent is the next parameter that determines which of the two operands is smaller. At the end of the previous section, we explained that the alignment block already provides the means for comparing exponents. If the two exponents are equal, then the significand determines the outcome of the operation.

For a 'less' than compare predicate we need at least a sign-bit comparator, an 8-bit exponent comparator ($<$) and a 32-bit significand comparator ($<$). For 'equal to', similar comparators are needed although they are much simpler because equality comparators ($=$) only have to compare the input bit-for-bit. An efficient implementation for full comparison is shown in Figure 6.6. Instead of comparing everything at the same time, a more hierarchical approach is taken. First the signs, exponents and significands are compared separately and subsequently encoded in a single bit true/false value. Custom pattern matching logic then evaluates the obtained values against the opcode to determine the correct outcome. A design like this yields better synthesis result than flattened comparison because most tools are not able to recognize that only one set of comparators is needed to support full floating-point compare functionality ($<$, $=$ or $>$).

Integer Re-use

To compare two integers, we require a 32-bit two's complement comparators. The floating-point significand comparator is 32-bit (we do not have to compare the hidden-bit), which makes it a candidate for hardware re-use. This comparator is however not directly compatible with two's complement input. Although configurable comparators exist, tools will not be able to recognize the need for them without explicitly pointing this out. As with the alignment shifter, the input is therefore sign extended for integers and `0`-extended for floating-point operands. A common two's complement comparator can then be used to compare both types of input. The pattern matching logic accounts for the fact that integer compare depends solely on the result of this single comparator.

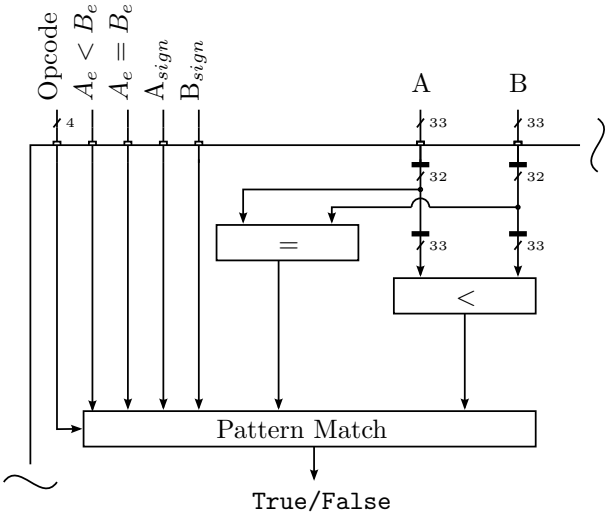


Figure 6.6: Comparator

6.5 Fused Multiplication-Addition

At the heart of the ALU lies the multiply-add structure. By the multiply-add structure mean the partial product multiplier, the carry-save adder and the carry-propagate adder from Figure 6.1. Combined, these components provide the basic means for multiplication and addition. The conditional complementer and re-complementer also have to be included in order to support subtraction. In the upcoming sections we will focus on this core functionality of the datapath.

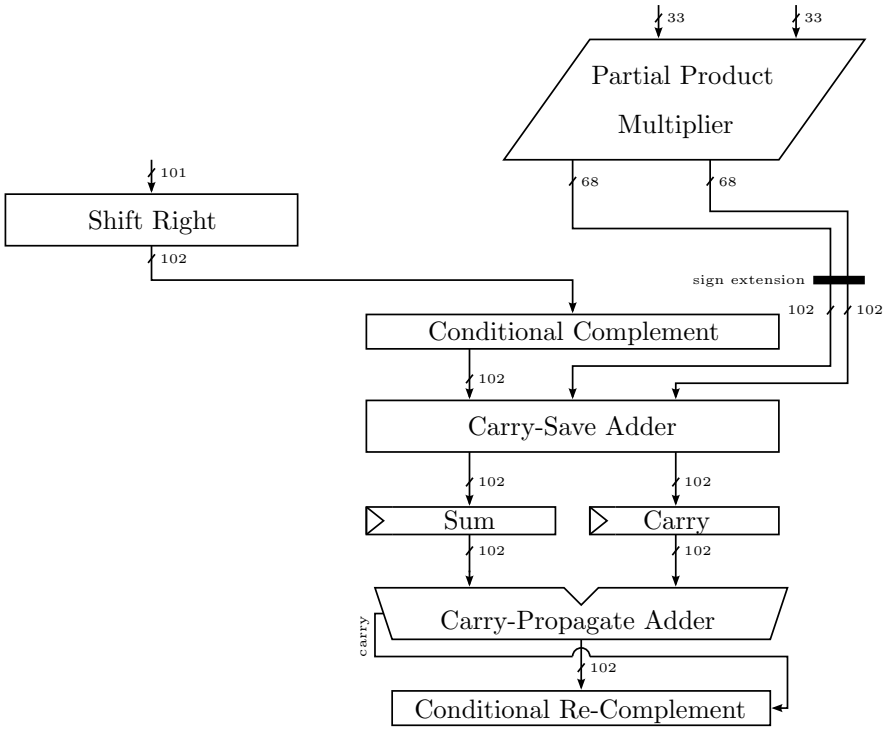


Figure 6.7: Core multiply-add functionality

6.5.1 Conditionally Complementing

After being aligned, the 102-bit addend C may need to be complemented as discussed in Section 5.4.1. C only needs to be complemented in case of effective subtraction. Fortunately, the effective operation is very easy to determine. Effective subtraction only occurs when the signs of two operands differ. Therefore, $((A_{sign}$ XOR $B_{sign})$ XOR $C_{sign})$, indicates whether or not the effective operation is a subtraction. There is one exception: if any of the operands is zero, the addend should never be complemented. If A or B is zero, the C operand is simply forwarded to the output (i.e., A×0+C=C or 0×B+C=C). If C itself is zero, it should not influence the result (A×B+0=A*B), hence C should never be complemented if any of the operands is zero. The effective operation is therefore combined with the logical OR of the zero detect signals from the instruction decoder. Based on the evaluation of the resulting control signal, the addend C is inverted to obtain its complement.

The (complemented) addend is also extended by one bit. Because the partial product obtained from the multiplier is always signed due to Booth encoding, a false carry out may occur. By explicitly including the sign-bit, some of these false carry out cases are eliminated. Although including the sign-bit is not sufficient to solve the entire problem, we mention it here because the entire datapath needs to be arranged for 102-bit wide data.

I/O Signal	Width	Direction	Function
A	n	Input	Multiplier
B	n	Input	Multiplicand
tc	1	Input	Two's Complement Control 0 = unsigned 1 = signed
PP_l	$n+2$	Output	Partial product
PP_r	$n+2$	Output	Partial product

Table 6.2: Partial product multiplier parameters

6.5.2 Partial Product Multiplier

In Chapter 5 we showed that the for FMA architectures, the most efficient multiplier is a partial product multiplier based on modified radix-4 Booth encoding and a Wallace tree structure. Despite the availability of sophisticated tools, designing such a multiplier is a complex matter. Even non-Booth multipliers based on trees or arrays require substantial design effort. For this reason, many tools support operation inferencing. When a certain operation such as addition or multiplication is encountered in the RTL description, the compiler consults technology independent libraries that contain parameterized building blocks like multipliers and adders. Not only does inferencing reduce development time, the result is often much more optimized than the result obtained from manual design efforts.

Some tools may be able to recognize multiply-add operations and deduce a structure based on a partial product multiplier as described in Section 5.4.1. However, because the 102-bit aligned C operand does not match the 66-bit multiplication product of A×B, most tools are not capable of recognizing the exact design we presented in Chapter 5. And even if they did, the multiply-add structure would be either signed or unsigned, not a combination of both. Inferencing multiplication and addition separately will also not produce the desired result. Without proper guidelines, the compiler may deduce a fast Wallace tree multiplier, but it will also include the final stage carry propagate adder. This is unwanted since inserting an additional CSA *before* the final carry propagate addition (Figure 5.5), significantly contributes to the performance of the FMA datapath. Although this means we can not rely on operation inferencing, a complete manual design of the multiply-add logic is not a viable solution either, considering the required development time. Fortunately, most compilers allow explicit instantiating building block from the before mentioned libraries. So does Synopsis DesignCompiler [41], the compiler that is used for the realization (Chapter 7) of the ALU we describe here. The Synopsis DesignWare library[1] contains over 140 re-usable datapath IP components including a partial product multiplier (`DW02_multp`). The most relevant parameters of this partial product multiplier are listed in Table 6.2. For the specific design of this thesis, a partial product multiplier with n=33 is instantiated.

The block diagram corresponding to the instantiated partial product multiplier is shown in Figure 6.8. As can be seen, it consists of a Booth recoder to generate the partial products, a Wallace tree to reduce the partial products and additional routing logic to position the partial product for the initial level of CSAs. Exactly the kind of combination required for fast FMA. Because `DW02_multp` is fully parameterized, we benefit from both the convenience of design automation and the flexibility of manual design. The tooling can for example find the best combination of CSAs for an efficient Wallace tree, based on the input length n. At the same time the partial products are directly accessible, giving us full control of the data flow after the Wallace tree (i.e., an additional CSA can be inserted for fast multiply-add). The Booth recoder is controlled by the `tc` signal. With this signal we can configure the multiplier for floating-point arithmetic or integer arithmetic. With `tc = 1`, the Booth recoder scans the input as two's complement data, with `tc = 0` as an unsigned number. It should be noted that in case of multiplication by constant (e.g., A×1, for the FMA derivative A+C), the constant should be mapped to port B for optimal results.

[1] http://www.synopsys.com/IP/SOCInfrastructureIP/DesignWare/Pages/default.aspx

Regardless of the two's complement control signal, the output is always signed due to Booth encoding. This has consequences for re-complementation (carry-out after addition), which we will discuss at the end of this chapter. Due to Booth encoding, the output is also two bits wider than the minimum (66 bits) that we would expect from a multiplier. However, when the outcome is positive (which is always the case for floating-point arithmetic), the two MSBs can be ignored according to the DesignWare specification. After adding the partial products, both positions will be `0` which is ideal because we need them to be zero for the guard and round-bit positions (Section 5.2). For the integer case we can also ignore the MSBs due to sign extension.

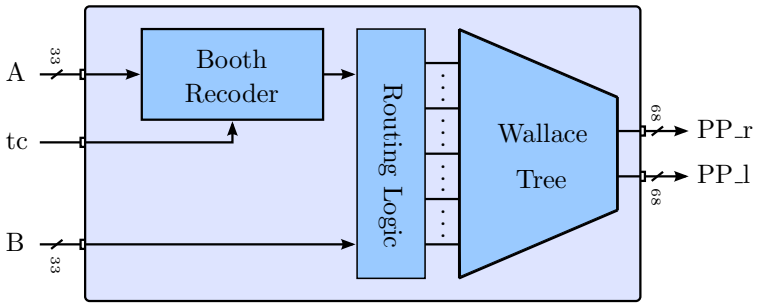


Figure 6.8: Synopsys DesignWare partial product multiplier (DW02_multp)

The Synopsys DesignWare partial product multiplier perfectly suits our needs. However, this component makes the design highly vendor dependent. Since DesignWare is Synopsys proprietary, it can not be used in combination with other tooling. A partial product multiplier is a very specific component that not all tools may have available. In principle the Synopsys tools should cover all the needs for realization. Synplicity can be used for FPGA prototyping (Section 7.2) and DesignCompiler for ASIC migration. However, in the event that the Synopsys tools can not be used, the partial product multiplier will have to be substituted with a similar building block or (re)designed by hand. We will not discuss in length all the details of manually designing efficient partial product multipliers. The basic design principles have been explained in Chapter 5 and Schwarz also covers a lot of detail in [30].

There is however one point of interest that is often a problem with combining the partial product and the addend prior to reducing them to one final product. Due to sign extension of the partial products, there is a chance of erroneous carry out propagation that needs to be ignored. As discussed in Section 5.4.1, the carry out of the final adder is used to determine if the result needs to be re-complemented. The carry out of an effective subtraction with a positive sum must be detected and separated from the other carries. Schwarz describes a method that maintains an additional bit in the CSA tree. However, since we have no control over this structure and the data sheet of `DW02_multp` does not mention anything regarding sign extension carry out suppression, additional custom control is used to detect erroneous carry out bits of the adder.

6.5.3 Carry-Save Adder

The partial products and the aligned and conditionally complemented addend are compressed with a single 3:2 CSA, before converting the redundant carry save format with the final adder of the multiplier tree. Although such a CSAs is easily designed by hand, the CSA from the DesignWare library (`DW01_csa`) was used to further reduce development time. Of course this CSA is also fully parameterized like the partial product multiplier. All its parameters are listed in Table 6.3.

To use this CSA properly, the carry-in should be fixed to `0` and the carry-out ignored. It does not matter how the partial product and addend are connected to the input, however for naming convention

I/O Signal	Width	Direction	Function
A	n	Input	e.g. left partial product
B	n	Input	e.g. right partial product
C	n	Input	e.g. addend
ci	1	Input	Carry-in
Sum	n	Output	Sum output data
Carry	n	Output	Carry output data
co	1	Output	Carry-out

Table 6.3: Carry-save adder parameters

the addend is connected to port C. As shown in Figure 6.7, the Carry and Sum obtained from 3:2 compression are first stored in a local register before they are further processed by the final adder.

Note that manually designing a n-bit 3:2 CSA can be done by using n full adder elements in parallel, without connecting the carry out and carry-in of the elements.

6.5.4 Final Adder

The final adder for end-around carry addition should be a fast carry propagate adder. Moreover, end-around carry addition requires that the final addition is incremented when the effective operation is subtraction, and the magnitude of the product is smaller than that of the addend (Equation 5.3). The most cost effective solution for incrementing is asserting the carry-in of the adder to `1`. Unfortunately this is not possible here because the increment directly depends on the carry-out of the adder. We are forced to use two adders, one to compute the sum and one to find the incremented sum. Because the increment is conditional, a *compound adder* is commonly found in the final stage of end-around carry addition. Compound adders produce both the sum and sum+1. The correct result is selected with a two-to-one multiplexer driven by the carry-out of the adder. The advantage of compound adders is that they work in parallel, which means less delay. Alternatively, two adders can be used sequentially by connecting the carry-out of the first adder to the carry-in of the second adder. This solution was already shown in Figure 5.6 and has an area advantage over compound adders because it does not require the multiplexer.

The final addition implementation is shown in Figure 6.9. Post-synthesis results (Chapter 7) show that two 102-bit adders working sequentially are not the critical path. Since sequential adders provide a marginal area improvement, this solution was chosen in favor of the compound adder. By connecting one of the inputs of the second adder to an internal fixed constant of value zero, it becomes the functional equivalent of an incrementer. As indicated in Figure 6.9, the first adder element is another instantiated DesignWare component (`DW01_add`). The reason for this is that they provide easy access to the carry-in and out, in contrast to HDL operand inferencing.

Overflow

The carry-out of the second adder can be ignored. Due to the positioning of the guard and round-bit, as described in the beginning of Chapter 5, the significand will never overflow. Both positions are initially `0`, stopping any carry from propagating further than the round-bit position as depicted in Figure 6.10.

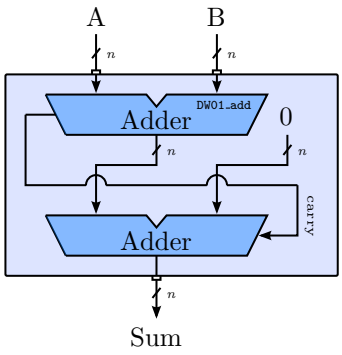


Figure 6.9: End-around carry adder

6.5.5 Integer-reuse

Most floating-point hardware re-use is achieved in the core of the datapath, that was just described. For integer multiply-accumulate, the partial product multiplier, the CSA and the final carry propagate adder are used to obtain the exact same functionality that the datapath also provides for floating-point numbers. With `tc`=1, the partial product multiplier reads the input as two 33-bit two's complement numbers and produces two 68-bit signed partial products. The alignment block takes into account that integer arithmetic input should not be shifted or complemented, such that it can immediately be added to the partial products, after being sign extended. The CSA compresses the addend C and partial products into a redundant carry-save format that is converted to a single two's complement result by the carry propagate adder.

The overhead needed to map integer multiply-accumulate to fused multiply-add is minimal due to the configurable partial product multiplier and flexibility of two's complement notation. The only modification that is needed sign extension of the addend and partial products, instead padding them with zeros. The shift amount is fixed to zero and the complementers disabled such that the dataflow remains regular. This requires only one additional control signal that differentiates integer arithmetic from floating-point arithmetic (obtained by the instruction decoder). Because two's complement arithmetic does not differentiate between addition and subtraction, the final adder will simply perform both operation without the need of complementers.

Integer Arithmetic Status Control

Although not explicitly shown in Figure 6.1, an additional control block evaluates integer arithmetic results for single (32-bit) register overflow. Because integer instructions will never result in actual over-

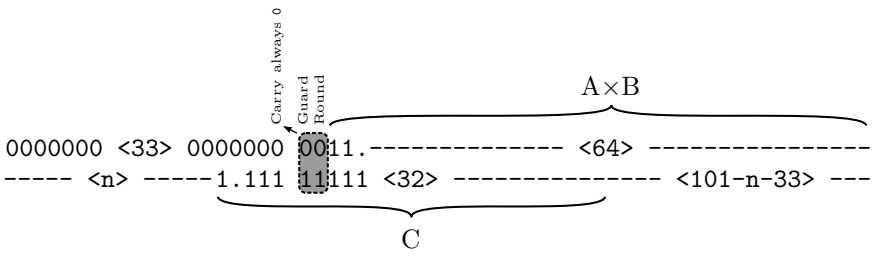


Figure 6.10: Significand overflow elimination by guard and round-bit

flow, the status bits are used to indicate if the result can be stored in a single register. It is easily shown that two 32-bit two's complement integers will never overflow. Considering that maximum representable number is $2^{31}-1$, the highest possible arithmetic result is $2^{64}-2^{32}+2^{31}$ $((2^{31}-1)\times(2^{31}-1)+(2^{31}-1))$. This still falls well within the 64-bit range $(2^{63}-1)$, which also holds for the most negative result. It is therefore more useful to indicate if the results overflow in a single register (32 bits), because this can reduce memory usage.

Due to sign extension, detecting single register overflow is very easy to deduce. If the 32 MSBs all have the same value, then the result will also fit in a single register. To implement this detection mechanism, we AND-reduce and OR-reduce the 32 MSBs of the adder outcome. If this results in either `1` or `0` respectively, then the result will fit a single register.

6.5.6 Conditional Re-complement

The final step in end-around carry addition for floating-point operands is conditionally re-complementing the sum. Section 5.4.1 already hinted that the complement of the adder sum can be found by using XOR gates. If an arbitrary string of bits is XOR'ed with `0`, the outcome will be exactly the same as the input. By XOR'ing with `1`, every bit is inverted.

By connecting every bit position of the sum to an XOR gate, inversion of the sum can easily be controlled with the other XOR input. We could also use inverters and multiplexers which may have a latency advantage. However, as mentioned before, the second stage is not critical for the overall performance so in terms of performance the difference will not be noticed. Because the difference in area is also negligible, the XOR approach was chosen as it is closest to the definition of end-around carry addition by Vassiliadis et al. [35]

The effective operation and the carry-out of the final adder control the inversion. The logical value of $(((A_{sign}$ XOR $B_{sign})$ XOR $C_{sign}) \wedge \overline{C_{out}})$ is fed to the secondary input of the XOR gates. It is easily shown that this results in correct inversion if we look back at Equation 5.4. If the carry-out is `0` and the addend was complemented for effective subtraction, the result must be re-complemented.

Erroneous Re-complementation

In some cases the datapath as described so far may detect the need for re-complementing when this is actually not required. As mentioned earlier, a faulty carry-out of the final adder can cause such an event. The sign-bit is therefore included in the addition explicitly [33] such that the carry will never propagate all the way through for effective addition (i.e., both sign-bits `0`).

Other cases where faulty re-complementation might occur is zero-arithmetic. Earlier we described that in case A or B is zero, or when C is zero, the other operand is simply forwarded to the output. Unless, the complementer is aware of this, it will still complement its input if the conditions are met (effective subtraction with a carry-out of `0`). Two measures are taken to make the complementer aware of such situations.

- Include control that disables the re-complementer when no complement was performed
- Include control that disables the re-complementer when A or B is zero, or when C is zero (and either A or B or both are not)

6.6 Normalize

The sum obtained from the multiply-add core may have to be normalized in case of floating-point arithmetic. To speed up this process, it is divided over two pipeline stages. In the second stage the LZA and LZD circuits determine the number of leading zeros by means of prediction. In the last stage a shifter actually performs the left shift and the according exponent adjustment based on the prediction of the second stage. Together the leading zero anticipator, leading zero detector and shifter form the Normalize sector of the datapath (Figure 6.11). Because LZA can deviate one position from the actual number of leading zeros, a correction may have to be performed before rounding the result.

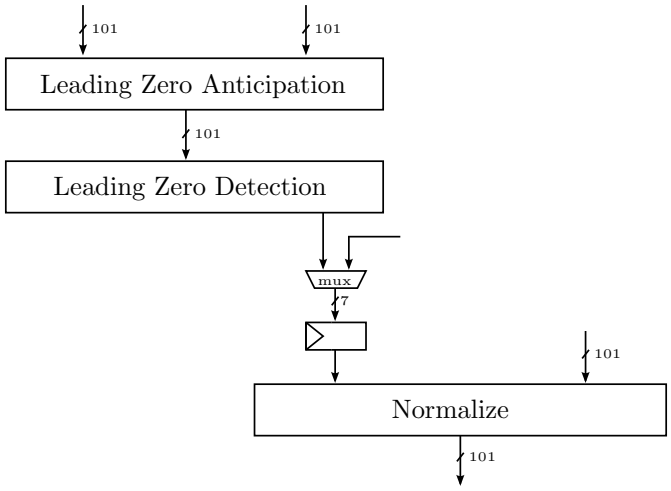


Figure 6.11: Normalization sector

6.6.1 Leading Zero Anticipation

LZA is a technique to predict the number of leading zeros based on the input of the adder. It improves overall performance by taking normalization off the critical path. In Section 5.5.1, a boolean relation was derived that forms a string of indicators (f) from the adder input (A and B). These indicators (f_i) predict if position i of the result after addition will be `1` or `0`. For convenience the equation is repeated here.

$$f_i = \begin{cases} \overline{\mathrm{T_0}}\mathrm{T_1} & (\mathrm{i} = 0) \\ \mathrm{T_{i-1}}(\mathrm{G}_i\overline{\mathrm{Z_{i+1}}} \vee \mathrm{Z}_i\overline{\mathrm{G_{i+1}}}) \vee \overline{\mathrm{T_{i-1}}}(\mathrm{Z_i}\overline{\mathrm{Z_{i+1}}} \vee \mathrm{G_i}\overline{\mathrm{G_{i+1}}}) & (\mathrm{i} > 0) \end{cases}$$

where

$$T_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$
$$Z_i = \overline{A_i}\ \overline{B_i}$$

Based on the prediction of f, the number of leading zeros can be counted in parallel with addition, which allows us to reduce the delay of normalization.

The input for the LZA logic is the same input as for the adder. However, the sign-bit is explicitly included for end-around carry addition. To perform correct LZA, the sign-bit must not be included, hence the

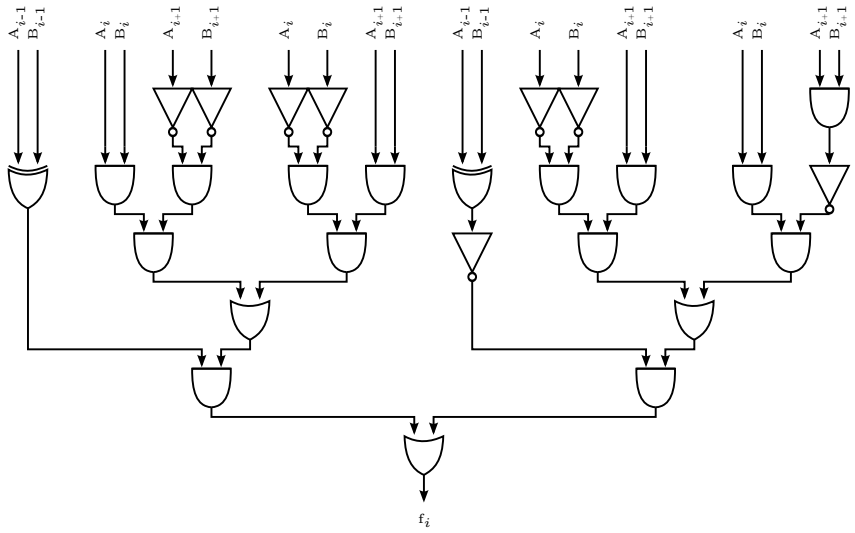


Figure 6.12: Leading zero anticipation logic

MSBs of A and B are removed by the leading zero anticipator. We already mentioned that the logic for LZA is quite straightforward to implement after the logic relation for each indicator has been found. It consists solely of inverters, AND, OR and XOR gates. One aspect that needs extra attention is that the first and last indicator positions are different from the rest. Because the boolean equation for indicator position i is defined based on input positions $i-1$, i and $i+1$, the first and last positions are defined differently. For position $i = 0$, the indicator is defined as $\overline{T_0}T_1$. The least significant indicator should be `0`.

LZA logic is comparable to an adder in terms of area. The latency however, is significantly smaller. Figure 6.12 depicts the (unoptimized) logic for one positional leading zero anticipation. Even without any optimization, this circuit is only six logic gates deep. For comparison, the carry of the adder propagates through 101 full adder elements that are each three logic gates deep.

6.6.2 Leading Zero Detection

Leading zero detection is based on the work of Oklobdzija [1], slightly modified for area efficiency. Algorithm 5.5.1 showed how a hierarchical tree structure can be created that encodes the number of leading zeros into a binary number. This algorithm scales well for powers of two. Without any overhead, a 4-bit leading zero detector is created from two 2-bit leading zero detectors, an 8-bit leading zero detector from two 4-bit leading zero detectors and so on. After addition of A×B and C, the intermediate floating-point significand is 101 bits (2×33+2) wide. The closest power of two is $2^7 = 128$. Instantiating a 128-bit LZD leads to area overhead, hence the algorithm from Oklobdzija was modified for a more area-efficient solution.

A 101-bit leading zero detector is the most desirable solution, however 101 is not a multiple of two (the smallest possible leading zero detector described by Algorithm 5.5.1) making it very hard to create such a tree structure efficiently. The closest alternative is 102 bits. The area overhead of a 102-bit LZD is minimal and can be made fairly easy by combining smaller LZD circuits. We use 64, 32, 4 and 2-bit LZDs to obtain a 102-bit version. Individually these counters are all powers of two that can be generated according to Algorithm 5.5.1. First the 64 and 32-bit trees are combined to form a 96-bit circuit. Then the 4 and 2-bit LZDs are combined to obtain a 6-bit counter. The final step is to combine the resulting 96-bit and 6-bit LZD circuits into a 102-bit counting tree. This last step requires modifications to the algorithm from Oklobdzija, that can best be explained by a small example.

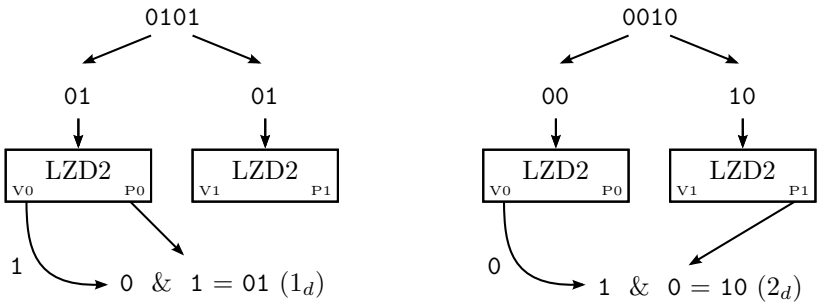


Figure 6.13: 4-bit LZD example

Consider the combination of two 2-bit LZDs (Figure 5.7(a)). Oklobdzija states that if the valid bit of left LZD (V_0) is 1, a 0 is concatenated to the positional bits of the right LZD (P_1). Otherwise, if the valid bit of the right LZD is 0, a 1 is concatenated to the positional bits of the left LZD. Figure 6.13 illustrates that this works flawlessly.

Now consider the combination of a 6-bit LZD and a 2-bit LZD. If the same principle is applied, the zero count is not always correct. In Figure 6.14 we can see that if the valid bit of the left LZD is 1, detection is still accurate. However, when the valid bit is 0, the result is no longer what we expect. This can be explained by the fact that a 6-bit counter is not a power of two. It holds in general that combinations of LZDs that are not both powers of two, Oklobdzija's algorithm does not produce correct results.

A solution can be found by analyzing what exactly goes wrong. If two $n/2$ LZD circuits are connected to perform a n-bit zero count, the result of the left LZD is the number of leading zeros in the upper $n/2$ bits, while the right LZD counts the zeros in the lower $n/2$ bits. When the left count is valid, the right count does not matter, hence a 0 is concatenated with the left result. If the left count is not valid and the right count is, we know that all the bits that precede the lower $n/2$ bits are zeros. If the left LZD is a power of two, a 1 on position $(n/2)+1$ equals the number of upper bits and therefore also the number of zeros. A concatenation of 1 with the positional bits from the right LZD still produces a correct result. However, when the left LZD is not a power of two, a 1 on on position $(n/2)+1$ does not equal the number of zeros. For example, the 6-bit LZD shown in Figure 6.14 uses three bits to represent the leading zero count. A 1 on position $(3)+1$, represents eight which is more than the maximum of six.

What Oklobdzija's algorithm actually does is adding both counts in a very clever way. Unfortunately this also restricts it to certain lengths. A slight modification is applied to remove this restriction. Instead of concatenation, an adder is used to combine the results of both LZD circuits when the number of zeros is larger than the left leading zero detector can count. First the result of the smaller counter is extended to match the output of the larger counter. The results are then added but the outcome will equal the number of leading zeros minus one. The left leading zero detector only counts up to $(n/2)-1$ positions, if the number is larger then the invalid bit is flagged. This means that the result from the left leading

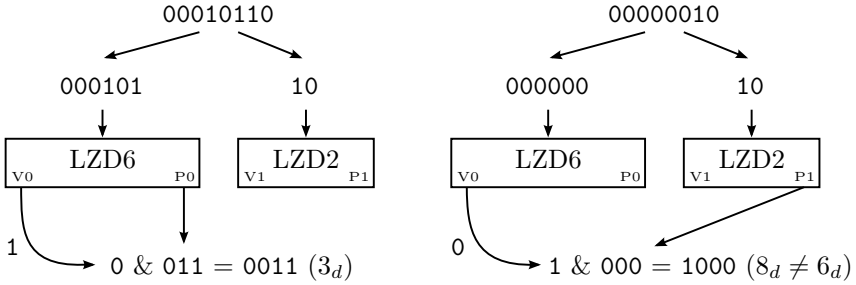


Figure 6.14: 6-bit LZD example

zero detector must be incremented for accurate detection. This can be implemented by a common adder with a carry in of 1. Hence, to combine the 96 and 6-bit LZDs, a 7-bit adder is used with the carry in fixed to 1. A multiplexer, driven by the valid bit of the left leading zero detector selects between the cases where the the leading zero count from the left LZD should be used of the output of the adder. Figure 6.15 shows that this solution works for the 6-bit example that initially failed.

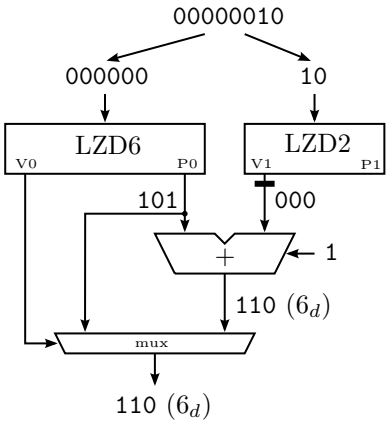


Figure 6.15: Modified leading zero detection

6.6.3 Shift Left and Exponent Adjustment

The LZA and LZD circuits produce a binary encoded leading zero count. This is merely a preparation for the actual normalization though, as the significand still needs to be shifted until the MSB is 1. For normalization a left shifter is needed. Of course the intention is to share this shifter between floating-point normalization and integer shift left instructions. It turns out that this is simpler for a left shifter than for a right shifter. There is no difference between an arithmetic shift-left and a logical shift-left. In both cases zeros are shifted in from the right. A common 101-bit shifter suffices for both purposes.

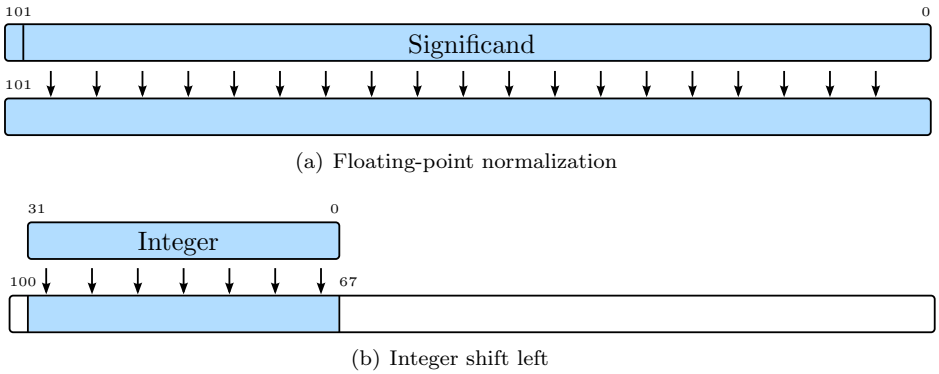
To match the two clock cycle delay of integer arithmetic, the input for the shifter is stored for one clock cycle, before it is sent to the shifter. To save one register, the input of the inter-stage summation register is multiplexed instead of the input of the shifter itself. The main purpose of this register is to store the intermediate significand of A×B+C. Because this result still includes the sign-bit, the MSB is removed before the left-shifter normalizes the significand. Since the MSBs from the shifter proceed for further processing, the integer input should also be mapped to the MSBs for datapath regularity. However, this also means that the input should be corrected for the loss of the MSB, as shown in Figure 6.16(b).[2]

6.7 Rounding

The rounding unit supports four out of five IEEE rounding modes. “Round to nearest ties away from zero” was omitted due to the strong resemblance with “round to nearest ties to even” which is used much more often. Selection between rounding modes is made based on the the opcodes from Table 6.1. When the opcodes are interpreted by the instruction decoder, a two bit round mode control signal is generated. These encodings can be found in Table 6.4.

At the time the intermediate normalized floating-point result arrives at the rounding block, it is still 101 bits wide. Before it is being rounded, we truncate it to 35 bits, a 33 bits significand plus a guard and

[2]For ISLV the input is not extended



(a) Floating-point normalization

(b) Integer shift left

Figure 6.16: Input mapping to left shifter

round-bit as shown in Figure 4.10. The discarded bits are logically OR'ed into a secondary sticky-bit which is then combined with the sticky-bit we found during alignment. The sticky-bit is paired with the round and guard bits to form a triple that serves as a bit pattern for rounding selection (Figure 6.17). Rounding algorithms 5.6.1, 5.6.2 and round to zero are implemented exactly as they are described in Chapter 5.

The hardware requirement for actual rounding are modest. A 33-bit incrementer, several small comparators for pattern matching and the control logic to select the rounding mode. The major penalty of rounding comes from sticky-bit calculation. Both the primary and secondary sticky-bit calculation require a 66-bit OR-gate tree to reduce the discarded bits to a single sticky-bit.

Similarly to the normalization hardware, rounding has no influence on integer arithmetic as the entire third stage is bypassed for this type of instruction. This also means that none of the rounding hardware is re-used for integer purposes.

6.7.1 Overflow

Because rounding involves incrementing the significand, an overflow can occur. At the begin of this chapter it was already mentioned that overflow detection is best postponed until after rounding. This has not yet been implemented but a solution is presented in Section 9.3.

Mode	Encoding	Description
`ZERO`	`00`	Rounds to zero (i.e., truncates the result)
`NEAREST`	`01`	Rounds to nearest, ties to even
`POSINFINITY`	`10`	Rounds to $+\infty$
`NEGINFINITY`	`11`	Rounds to $-\infty$

Table 6.4: Round modes

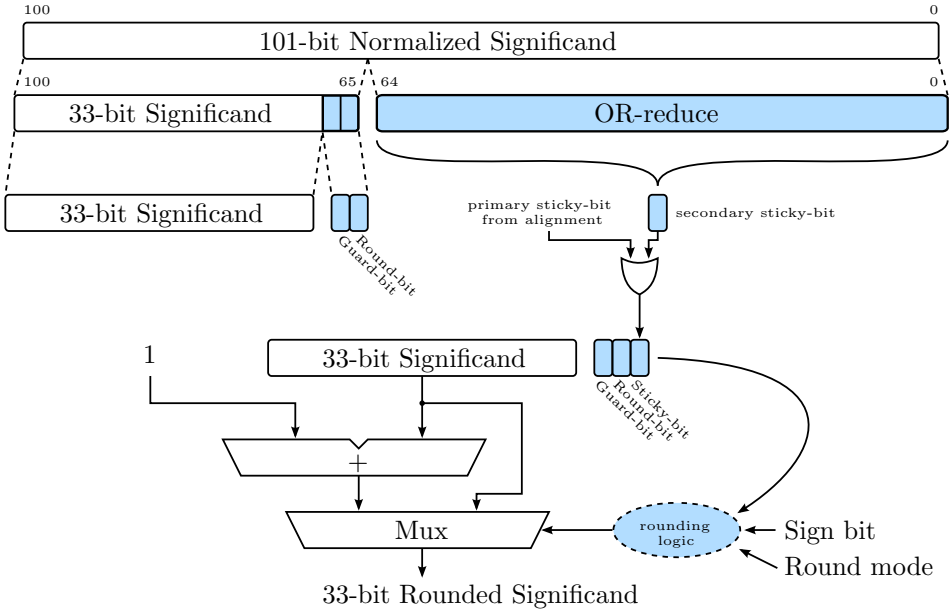


Figure 6.17: IEEE-754 rounding for FMA

6.8 Output Formatting and Exceptions

The output formatter can be found at the very end of the pipeline. It serves three purposes.

- Routing data to the correct output
- Performing the final checks for overflow/underflow, and flag exceptions
- Correcting the sign-bit

6.8.1 Routing

Once the data has been processed, it is prepared for register storage by routing it to the data-out bus in a specific way. The output formatter has access to two 32-bit data buses (referred to as Left and Right) and a 3-bit status signal (Figure 4.6). The output of of the ALU is formatted to match its input, although for integer numbers this is not always possible. The floating-point results are all normalized, have a 32-bit significand, an 8-bit exponent and a sign-bit. The integer output is two's complement and in principle occupies twice the original storage amount of the input (i.e., 64 bits).

Floating-Point

When the formatter receives floating-point numbers, they are segmented into a 33-bit significand, a 9-bit exponent and a sign-bit. This data is mapped to the output as depicted in Figure 6.18. The MSB of the significand is dropped (i.e., becomes a hidden-bit) and the significand is routed to the right data bus. The exponent is also resized by dropping the MSB. The sign-bit is concatenated to the left of the exponent and the combination is mapped to the LSBs of the left data bus. The remaining positions are forced to `0`.

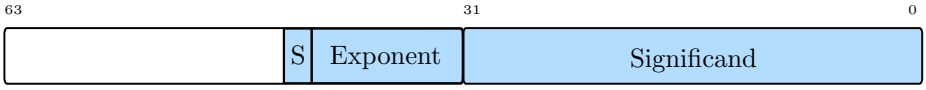


Figure 6.18: Floating-point output formatting

Integer

Because integer words can be up to twice as long as the original input, the result is divided over two registers. The upper bits (HI) are mapped to the left data bus and the lower bits (LO) to the right bus.

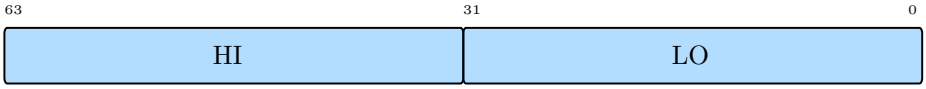


Figure 6.19: Integer output formatting

6.8.2 Status Control

The ALU differentiates six conditions of two different types, arithmetic and logic status. A 3-bit encoding is used to indicate the status of the ALU.

Status	Encoding	Applies to	Description
`TRUE`	`111`	Floating-point/Integer Logic	Outcome is true
`FALSE`	`000`	Floating-point/Integer Logic	Outcome is false
`DEFAULT`	`100`	Floating-point/Integer Arithmetic	No overflow/underflow
`OVERFLOW`	`001`	Floating-point Arithmetic	Overflow occurred
`UNDERFLOW`	`010`	Floating-point Arithmetic	Underflow occurred
`EXCEEDS32B`	`101`	Integer Arithmetic	Result larger than 32 bits

Table 6.5: Status bits description

Logical compare instructions either result in true or false. Since no exceptions can occur, the status bits themselves are used to indicate what the outcome of the respective comparison is. During floating-point arithmetic, two exceptions are flagged by the ALU: overflow and underflow. If neither is the case, a default arithmetic value is asserted to the status bits. For integer arithmetic the rules are slightly different. Since overflow and underflow can not truly occur (Section 6.5.5), the status bits indicate if the result can be represented in 32 bits (Table 6.5).

For reasons that where explained in Section 5.2, overflow should be detected at the very end of arithmetic instructions. Since the operand formatter is the final stage, a number of comparators must be place here to provide the means for detection[3]. Overflow is easy to detect due to the 9-bit exponent extension. If the exponent value is either `1--------` or `011111111`, the exponent has exceeded the maximum and we encountered an overflow. Besides signaling this exception with the status bits, the output is also overwritten with the IEEE-754 infinity representation (Table 2.3), as dictated by the IEEE-754 standard. Underflow is detected with the help of LZD. In IEEE-754 format, the same pattern is used to represent both zero and underflow. Simply checking for zero in the output formatter is therefore not sufficient to detect underflow. The leading zero anticipator described in Section 5.5.1 can help detect underflow by inspecting its valid bit. If the valid bit is `0`, we know that the significand consists of solely zeros. LZD also provides a leading zero count that equals the shift amount for normalization. If this shift amount

[3]The current datapath actually checks for overflow and underflow after normalization (Section 6.7.1), it is better to postpone until after rounding but this has yet to be implemented.

is larger than the exponent, the result will underflow and the status bits can be asserted to encode the underflow exception.

The underflow detection described above does not work for the special case where all input is zero. Regardless of the sign-bit, if the magnitude of all input is zero, underflow will be detected. This must be corrected by the output formatter. The input formatter already checks for zero input, hence those control signals can simply be evaluated by the output formatter to see if the status bit should be overwritten with the default value.

6.8.3 Sign-Bit

The sign-bit as calculated by Equation 5.7 is correct except for arithmetic with zero and infinity. The way the sign-bit has mathematically been defined in IEEE-754 does not correspond to this equation. A more elaborate discussion of zero-arithmetic can be found in Appendix B. The IEEE-754 standard surprisingly does not explicitly discuss the sign-bit for arithmetic with infinity. Infinity arithmetic is therefore based on the results obtained from x86 processors with an IEEE-754 compatible floating-point unit. This basically comes down to the exclusive or of the sign bits A_{sign} and B_{sign}. If the XOR is `1`, then the sign-bit is `1`, otherwise it is `0`.

IEEE-754 states the following about the sign-bit for zero-arithmetic:

"*When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be +0 in all rounding-direction attributes except roundTowardNegative; under that attribute, the sign of an exact zero sum (or difference) shall be -0. However, x + x = x - (-x) retains the same sign as x even when x is zero.*

When (a×b) + c is exactly zero, the sign of fusedMultiplyAdd(a, b, c) shall be determined by the rules above for a sum of operands. When the exact result of (a×b) + c is non-zero yet the result of fusedMultiplyAdd is zero because of rounding, the zero result takes the sign of the exact result."

To adhere to these rules, we check if the result is exactly zero, differentiation from underflow as discussed earlier. If the result appears to be zero, we must force the sign-bit to `0`, except when the rounding mode is `NEGINFINITY` or in the exact cases of (-0)×0-0 and 0×(-0)-0.

6.9 Summary

A complete design has been established. With relatively small overhead, large parts of the (originally floating-point) datapath can be re-used for integer arithmetic. The most drastic measure is the additional control needed to configure the hardware for signed (two's complement) or unsigned operating mode. The effects are most notable in terms of performance. The area overhead is minimal since the basic building blocks only need very small modification to support both signed and unsigned input. In many cases sign/`0`-extension is sufficient to make the synthesis tools understand that the same hardware is used for a different purpose. In Appendix D we provide an additional overview of the datapath usage per instruction(type), by means of illustrations.

Although this design shows the feasibility of combining floating-point and integer arithmetic in a single datapath, we have yet to determine the actual effectiveness of such a shared datapath. In the next chapter we therefore look at the realization of this datapath. Implementations in different technologies will be evaluated and compared with other (classical) approaches to decide if augmented integer floating-point units are worthwhile. We also look at the consequences of integrating the presented ALU in a state-of-the-art MPSoC.

7 Realization

7.1 Introduction

Tight area budget and the need for energy efficiency lead to the design of a new kind of ALU that combines floating-point and integer arithmetic/logic in a single datapath. Chapters 4, 5 and 6 showed how such a datapath can be devised. However, the physical properties such as area, clock frequency and energy consumption have yet to be presented. Because an actual implementation is subjective to fewer assumptions than any kind of model, it provides by far the most accurate and reliable readings of physical properties. Hence, in this chapter we discuss the findings of an implementation in 65nm process technology.

A standard netlist to GDSII (the *de facto* IC layout file system) design flow was used, based on Synopsys DesignCompiler [41] and Cadence Encounter [42]. With the current design, the tool chain is restricted to Synopsys tools due to the use of the aforementioned DesignWare components (see Chapter 6). This drawback is less of an inconvenience than it may seem at first glance. The synthesis tools by Synopsys are amongst the most widely used and are well supported by the semiconductor industry. In addition the DesignWare components are highly optimized. Manual designs of these components are unlikely to match the quality, speed and area efficiency of the DesignWare equivalents. In order to make the design completely vendor independent, a partial product multiplier and carry-save adder will have to be developed.

The hardware has not actually been manufactured. It should therefore be noted that although the numbers shown in this chapter are fairly accurate, they should still be regarded as estimates. Especially the quantities related to energy consumption should be used carefully. They are based on post place-and-route results from Cadence Encounter, which still showed some minor load capacitance problems.

7.2 FPGA Prototyping

Before hardware is manufactured, it is wise to evaluate it in an FPGA prototype first. Although not being the main purpose of this thesis, FPGA synthesis has been performed to ensure that the current design is compatible with FPGA prototyping. The ALU has been synthesized targeting a Xilinx Virtex 5 LXT (XC5VLX330T-2FFG1738) FPGA, using Synopsys Synplify D-2010.03-SP1 in conjunction with DesignWare foundation C2009.06-SP5. A summary of the resource utilization is shown in Table 7.1. The critical path is located in the third pipeline stage, running from the summation register through the normalizer, rounder and formatting to the left output of the ALU. The critical path limits the clock

Cell	Usage
FDE	3
FDR	513
GND	1132
LDCP	2
MUXCY	10
MUXCY_L	364
MUXF7	4
VCC	1129
XORCY	329
LUT1	122
LUT2	1428
LUT3	2509
LUT4	259
LUT5	542
LUT6	1105
LUT6_2	12
IBUF	129
IBUFG	1
OBUF	67
BUFG	1
I/O Register bits	0
Register bits not including I/Os	516 (0%)
Latch bits not including I/Os	2 (0%)
Global Clock Buffers	1 of 32 (3%)
Total LUTs	5977 (2%)

Table 7.1: Virtex-5 (XC5VLX330T-2FFG1738) resource utilization

frequency to 105MHz. An interesting observation is that the imbalance between the timing critical stage (i.e., stage three) stage following (the first stage) is very small. When retiming is disabled there is a 0.658ns slack observable. This indicates that the design is by itself already fairly well balanced.

7.3 ASIC Implementation

Once simulations and FPGA prototyping show that a design operates properly, the final steps before tape-out[1] can be initiated. In this section we describe the evaluation of ASIC migration of the ALU. ASIC design can roughly be divided in two steps: synthesis and place-and-route. During synthesis the hierarchical structural description of the architecture is translated into a flat netlist of standard cells found in technology libraries. The place-and-route (layout) phase consists of placing the synthesized components on a die, inserting a power-grid, synthesizing a clock tree and routing the interconnect wires. Although post place-and-route results are in principle the most accurate, in this case post-synthesis results are regarded to be more reliable. Because there are still some issues related to routing density and load capacitance, the post-place and route figures should only be interpreted in the order of magnitude.

[1] Final GDSII files were originally placed on magnetic tapes. This moment was fittingly called tape out.

ASIC Libraries

The technology libraries chosen for implementation are:

- CORE65LPHVT: STMicroelectronics 65nm low power high voltage threshold (LPHVT)
- CORE65GPSVT: STMicroelectronics 65nm general purpose standard voltage threshold (GPSVT)

CORE65GPSVT synthesis is used to asses the maximum clock frequency that can be achieved with the current design. From the technology libraries at our disposal, this general purpose complementary metal oxide semiconductor (CMOS) process delivers the highest performance. For energy-efficient applications the CORE65LPHVT library is more interesting. This low power high voltage threshold library is optimized for reduced energy consumption. The price that is paid for this reduction, is increased area (roughly 30% increase) and limited performance (the maximum achievable clock frequency is aproximately half of what can be reached with GPSVT implementations).

Procedures

The quality of results obtained from synthesis and place-and-route strongly depends on the tools and setting used during the process. We will therefore discuss the tool chain more thoroughly before we present the results. Synthesis was done in Synopsys DesignCompiler (C-2009.06-SP5), using DesignWare foundation (C-2009.06-SP5), the ultra_compile command and the following settings:

- Minimal area target (using the area high effort parameter)
- Disabled ungrouping for detailed area analysis
  (due the architectural structure of the design, ungrouping does not yield improved area or latency)
- Target clock speeds of 200/550(500)/1350(1200)MHz
  (550/1350MHz are the maximum achievable for low power and high performance respectively, before place-and-route)
- Clock gate insertion enabled
- Full scan chain insertion (for testability)

Post-synthesis simulations (Chapter 8) have been performed using the back-annotated netlists (i.e., with timing constraints) obtained from synthesis, to verify that the synthesis results are reliable. After synthesis, the low-power designs have undergone place-and-route in order to extract accurate power figures. Cadence Encounter (09.12-s159_1) was used to place-and-route the design with the placeDesign and nanoRoute commands. A basic script provided by STMicroelectronics was run, modified with the following (key) settings/constraints:

- Floorplan design aimed at 70% core utilisation
- Power net based on four power rails, evenly distributed over the die
- Automatic clock tree synthesis
- Fanout maximized at 20 (ST 65nm LPHVT technology determined)
- Transitions maximized at 0.700 ns (ST 65nm LPHVT technology determined)
- Capacitances maximized at 0.070 pF (ST 65nm LPHVT technology determined)
- Timing and power driven placement and routing
- High timing driven efforts
- Clock gate aware routing

Now that a precise definition of the procedures used for realization is given (such that the results are reproducible), we can discuss the results.

7.3.1 Timing

The maximum clock frequency for the ALU implemented in LPHVT technology is, according to the synthesizer, 550MHz. In a GPSVT process, the design can be synthesized at 1350MHz without running into timing issues. At first this does not seem impressive at all. Especially compared to the Cell's floating-point unit which runs well over 3GHz. However, if things are put in perspective, the overall picture becomes more positive. The ALU discussed in this thesis is based on three pipeline stages where the Cell processor uses six stages. In addition the Cell architecture does not apply rounding and is primarily optimized for speed. In terms of speed, the architecture we present is on par with the floating-point unit of the UltraSparc T2 processor. The latter is clocked at 1200MHz by default and can be boosted to 1600MHz. This processor also uses a much deeper pipeline and is not based on FMA architecture, such that multiply-add performance will be lower. On the other hand, the UltraSparc T2 datapath does support normalized and double precision numbers.

The synthesis estimates appear to be too optimistic. After place-and-route, the upper bound of the clock frequency for the the low-power implementation seems to be 500MHz. However we believe that the place-and-route result are rather poor due to lacking knowledge in this field on our part. The post place-and-route results shown in this thesis are based on a very basic script that STMicroelectronics provided, extended with a number of additional constraints. We believe the results obtained with this script are too pessimistic. For area and timing, post place-and-route result are therefore not particularly more meaningful than synthesis results. Hence, we will take the post-synthesis result with a penalty of 10% latency overhead (based on work of others) due to place-and-route. 200MHz and 500MHz low-power implementations are used to estimate the needed area. The GPSVT implementation is expected to reach 1200MHz after place-and-route.

The critical path location depends on the constraints used during synthesis. In the timing driven implementations (i.e., 500MHz LPHVT and 1200MHz GPSVT), the critical path is located in the first stage. Starting at the instruction selection input, it ripples through the formatter, the partial product multiplier and the CSA to end in the inter-stage register that stores the carry-save sum. Synthesis in lower clock speeds, which is interesting for low-power designs, result in a different critical path. When the LPHVT design is synthesized for a target clock speed of 200MHz, the critical path is located in the third stage. This path starts at the normalization shifter input and runs through the normalizer, rounder and output formatter. Table 7.2 shows detailed timing analysis for 200, 500 and 1200MHz implementations (excluding setup times). Figure 7.1 illustrates how the critical paths run through the datapath.

The fact that the critical path changes depending on the clock constraint is a good indication for the balance between the pipeline stages. The difference between the first and last stage are minimal, otherwise

Process	**Location**	**Arrival** (ns)	**Departure** (ns)
LPHVT - 200MHz			
	Normalize	0.00	3.14
	Round	3.14	4.71
	Output Formatter	4.71	4.86
LPHVT - 500MHz			
	Instruction Decoder/Input Formatter	0.00	0.30
	Partial Product Multiplier	0.30	1.67
	Carry-Save Adder	1.67	1.87
GPSVT - 1200MHz			
	Instruction Decoder/Input Formatter	0.00	0.16
	Partial Product Multiplier	0.16	0.68
	Carry-Save Adder	0.68	0.77

Table 7.2: Critical path analysis

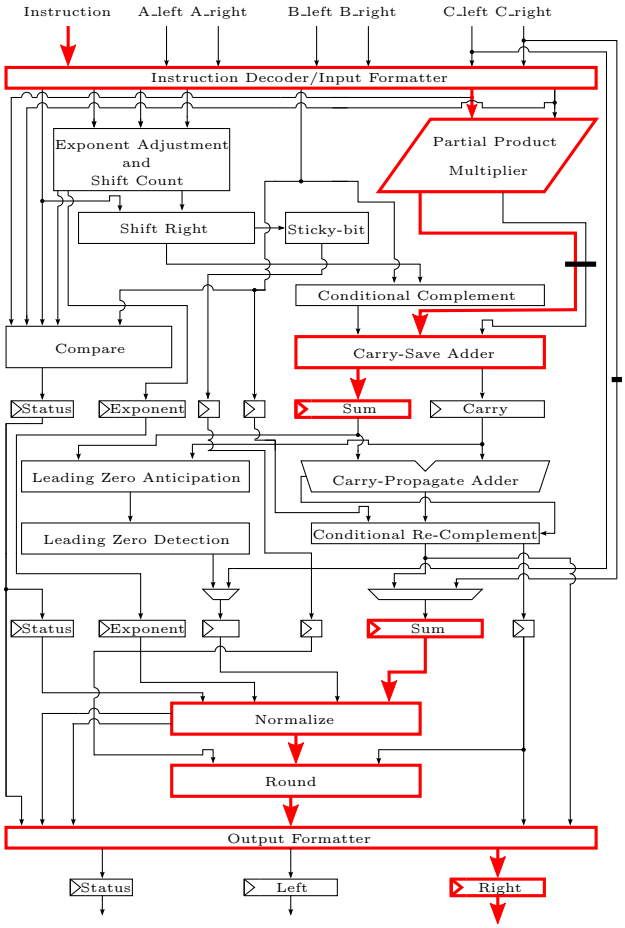


Figure 7.1: Critical paths

the location of the critical path would not depend on the implementation efforts of the tools. A good balance in the architecture of the pipeline is advantageous because it makes the design less depending on tools that support retiming. Despite the well balanced architecture, the low-power design can not be synthesized at 550MHz when retiming is disabled. A slack of -0.2ns is reported which indicates that about 60MHz is gained by retiming. In the GPSVT process, our design can still be realized at 1350MHz with retiming disabled.

It is difficult to determine how much the overhead caused by integrating integer operations into the floating-point datapath affects the maximum achievable clock frequency. The architecture has been designed with integer functionality in mind from the start, hence the integer functionality can not simply be disabled. Due to the fact that the dataflow has been designed as regularly as possible (Chapter 6), we think the difference is hardly noticeable. However, in order to obtain accurate numbers, a complete redesign is needed. The overhead created by irregularities in IEEE-754 arithmetic is much worse and easier to identify. Stage one and three (both critical paths) are seriously affected. The input and output formatters are solely needed to compensate for zero-arithmetic and exceptional sign bits. Early designs that did not account for the irregularities in the IEEE-754 sign-bit and zero-arithmetic could run as fast as 700Mhz in LPHVT technology.

7.3.2 Area

In the worst case scenarios (highest clock frequency, scan chain insertion and clock gating enabled), the area of the architecture is approximately 0.04mm^2 for both the LPHVT and GPSVT libraries. Although this number is based on synthesis estimates and therefore does not include filler cells and routing overhead, it is still a quite meaningful number. Post place-and-route results including all overhead (clock tree, power net, routing wires, buffers and filler cells) show a unexpected area increase of over 100% in LPHVT. Table 7.3 provides an overview of the timing and area differences between the GPSVT and LPHVT libraries. Parameters we are currently unable to determine are marked with '-'. In addition to the uncertainty in the post place-and-route results (Section 7.3), an ALU is never implemented in isolation. These place-and-route numbers are therefore not particularly more meaningful in this case. Hence, the area discussion is based on synthesis estimates that only include the total standard cell area.

For three interesting implementations, the area is evaluated more thoroughly. Two designs based on the highest possible (estimated) clock frequencies of 500 and 1200MHz in LPHVT and GPSVT respectively, and a 200MHz implementation because this is a frequency likely to be found in current low-power hardware platforms [2]. Tables 7.4, 7.5 and 7.6 show the hierarchical area distribution for the above mentioned implementations. The components shown in first column roughly match the ones from Figure 6.1. The second column is the absolute area expressed in square micrometers, the third column the area expressed in percentage of the total area. Note that the percentages shown do not add up to 100. The difference is overhead not worth mentioning in detail. The last column shows which DesignWare components are instantiated, either by hand or by operator inference. The area of the DesignWare component is also shown as the percentage of the total area of the specific component.

Figure 7.2 shows the averages of the percentages per implementation of the major components plotted in a pie chart for convenience. At almost one third of the total area, the multiplier is by far the largest component in the datapath. As expected this component is the constraining factor in the design, both in terms of latency and area. Parallel alignment and normalization also require a substantial amount of silicon area. Mostly due to parallel alignment, the registers for pipelining are among the components with the largest area . However, the area/performance trade-off is worth the extra silicon as we will show shortly. Based on the exact numbers shown in the tables, we can conclude that the area overhead in this design has been kept to almost a bare minimum. The components that require the most area are already highly optimized. Also the overhead caused by integrating integer functionality is very small. Although very accurate numbers are hard to derive because of the fact that the datapath has been designed with integer functionality in mind from the start, the numbers shown in this section indicate that the area required for augmented integer functionality is at most 3 to 4% (determined by parts of the Input and Output Formatters and the area unacounted for by Table 7.4, 7.5 and 7.6).

Property	LPHVT	GPSVT
Theoretical Maximum Clock Frequency Estimated by Synthesis	550MHz	1350MHz
Estimated Maximum Clock Frequency after Place-and-Route	500MHz	1200MHz
200MHz Synthesis Estimated Area	0.03mm^2	0.03mm^2
500MHz Synthesis Estimated Area	0.04mm^2	0.03mm^2
1200MHz Synthesis Estimated Area	-	0.04mm^2
200MHz Place and Route Core Area	0.09^{2}mm^2	-
500MHz Place and Route Core Area	0.11^{2}mm^2	-

Table 7.3: ST 65nm technology library overview

Component(s)	Area (μm^2)	Percentage (%)	DesignWare Instances
Overflow Control	97.7	0.3	-
Sticky-bit Generation	275.1	0.7	-
Output Formatter	421.7	1.1	-
Comparator	537.6	1.4	`DW_cmp` (50%)
Instruction Decode & Input Format	578.7	1.5	-
Exponent Adjustment	707.7	1.9	-
Round	922.4	2.4	`DW01_inc` (46%)
Leading Zero Detection	962.0	2.5	-
Carry Save Adder	1099.2	2.9	`DW01_csa` (100%)
EAC & Recomplement	1864.7	4.9	`DW01_inc` (57%)
Adder	2126.2	5.6	`DW01_add` (100%)
Leading Zero Anticipation	2327.5	6.2	-
Alignment Shift & Complement	3143.9	8.3	-
Non-Combinatorial (Registers)	4533.8	12.0	-
Normalizer	5341.9	14.1	-
Partial Product Multiplier	11535.6	30.6	`DW02_multp` (100%)
Total	**37758.8**	**100.0**	-

Table 7.4: Hierarchical area distribution - 65nm GPSVT implementation at 1200MHz

Component(s)	Area (μm^2)	Percentage (%)	DesignWare Instances
Overflow Control	93.6	0.2	-
Sticky-bit Generation	278.2	0.7	-
Output Formatter	507.0	1.2	-
Comparator	537.2	1.3	`DW_cmp` (54%)
Instruction Decode & Input Format	728.0	1.8	-
Exponent Adjustment	787.3	1.9	-
Leading Zero Detection	1014.5	2.5	-
Round	1091.5	2.7	`DW01_inc` (37%)
Carry Save Adder	1187.2	2.9	`DW01_csa` (100%)
EAC & Recomplement	1961.4	4.8	`DW01_inc` (56%)
Adder	2250.5	5.5	`DW01_add` (100%)
Leading Zero Anticipation	2487.7	6.1	-
Alignment Shift & Complement	3478.3	8.5	-
Non-Combinatorial (Registers)	4668.5	11.4	-
Normalizer	6329.0	15.5	-
Partial Product Multiplier	12248.0	30.1	`DW02_multp` (100%)
Total	**40730.6**	**100.0**	-

Table 7.5: Hierarchical area distribution - 65nm LPHVT implementation at 500MHz

Component(s)	Area (μm^2)	Percentage (%)	DesignWare Instances
Overflow Control	92.0	0.3	-
Sticky-bit Generation	276.6	0.8	-
Output Formatter	324.5	1.0	-
Exponent Adjustment	420.2	1.3	-
Instruction Decode & Input Format	440.4	1.3	-
Comparator	523.6	1.6	`DW_cmp` (50%)
Round	639.6	2.0	`DW01_inc` (45%)
Leading Zero Detection	916.7	2.8	-
Carry Save Adder	950.6	2.9	`DW01_csa` (100%)
EAC & Recomplement	1707.6	5.2	`DW01_inc` (54%)
Adder	1841.8	5.6	`DW01_add` (100%)
Leading Zero Anticipation	2369.1	7.2	-
Alignment Shift & Complement	2723.2	8.3	-
Normalizer	4272.3	13.1	-
Non-Combinatorial (Registers)	4213.5	14.7	-
Partial Product Multiplier	9747.9	29.8	`DW02_multp` (100%)
Total	**28519.9**	**100.0**	-

Table 7.6: Hierarchical area distribution - 65nm LPHVT implementation at 200MHz

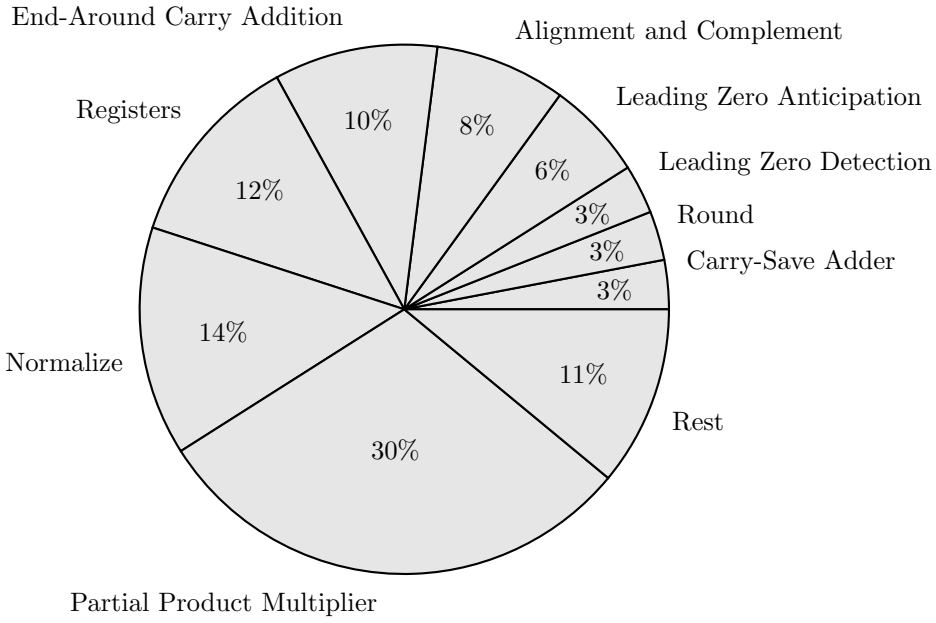


Figure 7.2: Area distribution of ALU components

Gate Count

The absolute physical area of a hardware design is highly depending on the technology used for implementation. Because technology is constantly improving, a comparison between two designs based on their absolute total area is unlikely to be fair. To obtain a more technology independent measure for comparison, the *gate count* is used. The gate count is obtained by dividing the total circuit area by the gate area of a two input NAND gate (NAND2). In this way, a relatively fair comparison can be made between the area of two different design that do not necessarily have to be implemented using the same technology.

The NAND2 area of the two 65nm ST technology libraries used for ASIC realization of our proposed architecture are listed below:

- CORE65LPHVT : 3.12 μm^2 (`HS65_LH_NAND2AX7`)
- CORE65GPSVT : 2.08 μm^2 (`HS65_GS_NAND2X7`)

Using these dimensions, we can express the area of our design in gate numbers:

Implementation	Absolute Area (μm^2)	Gate Count
GPSVT - 1200MHz	37758.8	18153
GPSVT - 500MHz	32475.0	15613
LPHVT - 500MHz	40730.6	13054
LPHVT - 200MHz	28519.9	9141

These result are not surprising. When more speed is desired synthesis tools will use more gates. For comparison the gate count of a 500MHz GPSVT implementation is also included. This clearly indicates that low-power design requires much more area to meet the same timing constraints.

7.3.3 Power Consumption and Energy-Efficiency

Because the purpose of the architecture partially is to provide low-cost floating-point functionality in energy-efficient hardware environments, the ALU itself should also be energy-efficient. Power analysis has been performed to obtain insight into the power consumption and energy efficiency of the hardware. A lot of power is consumed by the wires that interconnect the different components inside the ALU. An accurate power model should include wire load, which can only only be obtained by placing and routing the design. All the data discussed in this section is therefore based on post place-and-route results. Despite the fact that place-and-route results are not completely reliable, power estimates are are much more accurate after place-and-route. They should however still be interpreted in orders of magnitude only.

Energy consumption of a circuit can be expressed by:

$$E = \frac{1}{2} C {V_{dd}}^2 \tag{7.1}$$

Where V_{dd} is the supply voltage and C the total load capacitance of the design. Since the supply voltage is squared, lowering V_{dd} is usually quite effective in reducing energy consumption. The supply of 65nm ST technology is relatively low at 1.2V. The implementations are therefore by itself already quite energy-lean. At architectural level, energy consumption can be minimized by lowering C. There are several ways to do this, one way is to apply clock-gating. Clock-gating is what was chosen to improve energy-efficiency in this design. The effect of clock-gating on the energy consumption of our ALU is discussed at the end of this section.

Switching	200MHz	500MHz
20%	6.683mW	8.834mW
50%	13.62mW	18.17mW
80%	17.96mW	25.99mW

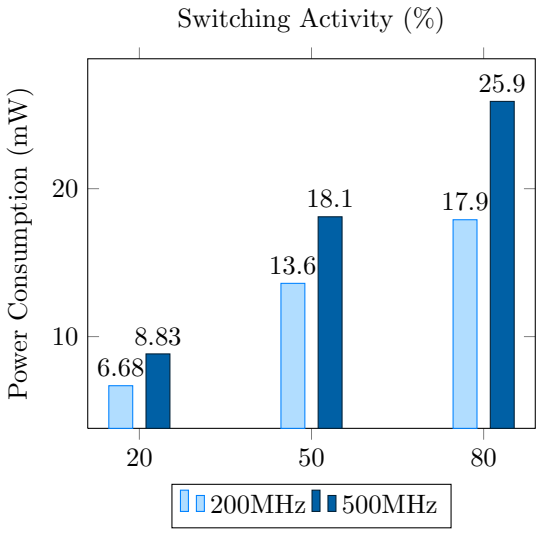


Figure 7.3: ALU power consumption estimates

Total power consumption of a chip, or any arbitrary piece of hardware, can be divided into static power and dynamic power. Currents flow through the transistors even when they are turned off. This is what causes static power. Static power can be considered as wasted energy as it does not contribute in any way to the system's functionality. In modern process technology (65nm and beyond), almost half of the energy consumption can be attributed to static power. Power dissipated by activity of the system (i.e., the system performing some function) is called the dynamic power. Within dynamic power, two subcategories can be distinguished: switching power and internal power. Switching power is caused by charging and discharging the load capacitance of the cells in a design. The total load capacitance is the summation of the interconnect capacitance and the the capacitance, hence it is better to base energy consumption on post place-and-route results power estimates.

The LPHVT library is very efficient at reducing static power. Since we are using deep-submicron technology, the use of low-power CMOS is almost mandatory for energy-efficiency (viz., static power contribution). Power measurements are therefore only performed for the low-power implementations [2]. The worst case scenario is assumed during power analysis. This means simulations are performed for 1.1V supply and 125°C environment temperature. Because power consumption scales with the clock frequency, estimates for 200 and 500MHz (with 0.258 and 0.113ns slack respectively) are compared. Usually the switching activity is based on actual data (e.g., an FFT computation). However, since such data is not readily available and because we only test an ALU and not a complete processor with memory, primary input switching activity is based on pre-set scalars (20% - low, 50% - average, 80% - high switching activity). Figure 7.3 shows the power estimates distiled from the power reports produced by Cadence Encounter (09.12-s159_1). Since energy en power are directly related (power is defined as energy divided by time), these are a good indication of the energy consumption of the circuit.

Power consumption strongly depends on the clock frequency. We therefore normalize it with respect to the frequency. The average power consumption is then defined as:

20% Activity: 51.08 μW/MHz
50% Activity: 104.4 μW/MHz
80% Activity: 141.7 μW/MHz

[2] A comparison between general purpose and low-power implementations could not been performed withing the time frame reserved for power analysis

Clock	Internal	Switching	Leakage
200MHz	5.045mW	8.461mW	0.1147mW
500MHz	7.136mW	10.83mW	0.2074mW

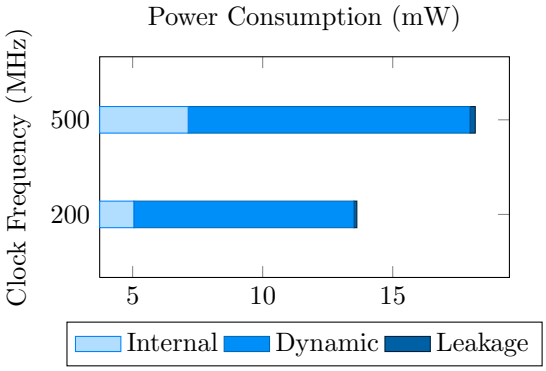


Figure 7.4: Detailed ALU power consumption estimates (50% activity)

Even more detailed power information is shown in Figure 7.4. In this figure the consumption of internal power, switching power and leakage power is distinguished for 50% input switching activity. This data confirms that most power is lost due to dynamic power. As mentioned earlier, without clock gating, the power caused by switching activity would have been even higher.

Clock-Gating

Clock gating is a techniques that inserts clock enable gates in the design. When certain part of the system are not required, the clock is disabled. By disabling the clock, these part become static thus lowering the dynamic power consumption.

The architecture has been synthesized and routed for 200MHz with clock gating disabled to show effective clock gating is. As can be seen in Table 7.7, clock gating reduces power consumption by almost 50%. We think this high percentage can be explained by the fact that the synthesis tool reckognises that the pipeline registers can be clock gated. By clock gating the registers, large parts of the datapath become idle because the input does not change.

7.4 Comparison

The data shown above provides a good indication of the physical properties of the new architecture when it is implemented. To give more meaning to the bare numbers, we will compare them to the properties of other architectures. A comparison between two implemented architectures is not trivial, especially in this case because the functionality (e.g., the non-IEEE floating-point format) and its implementation are relatively unconventional. Currently there are no equivalent datapaths to compare with. We therefore compare our solution with another implementation that approximates the functionality of our ALU, based on VHDL built-in floating-point support. In addition we compare the solution we propose to a more conventional IEEE-754 compatible floating-point unit: the GRFPU. GRFPU is what Aeroflex Gaisler has available for LEON-based SoCs.

Switching Activity	Without Clock Gating	With Clock Gating
20%	11.83mW	6.683mW
50%	22.53mW	13.62mW
80%	33.07mW	17.96mW

Table 7.7: Power consumption with and without clock gating

7.4.1 VHDL-2008 Standard Implementation

First a comparison between the newly found architecture and the VHDL standard interpretation of floating-point is made. Since the introduction of VHDL-2008, floating-point (arithmetic) has been defined as part of the VHDL-1076 standard. The behavior of this floating-point functionality has directly been derived from the fixed and floating-point packages originally written by David W. Bishop [36] (Section 8.1). These packages, based on VHDL-93 syntax, can be compiled and synthesized. Moreover, according to the FAQ [36], the synthesized results are quite efficient:

"Q: What synthesis results can I expect?
A: Under the hood, all of the fixed and floating-point functions call functions from the numeric_std package. Much work was done to make sure that these algorithms would be as fast as possible."

Synthesizing the desired functionality from this package is therefore a quick and convenient method to add floating-point support to any given architecture. Especially when research and development is on low budget or time-to-market is short, this flexible but instant solution is a likely choice.

For a fair and meaningful comparison the following components have been synthesized:

- Floating-point FMA unit (only support for 'round to nearest' and sticky-bit is disabled)
- Floating-point comparator (less than)
- Floating-point comparator (greater than)
- Floating-point comparator (equal)
- (Integer MAC unit)
- (Integer comparator (less than))
- (Integer comparator (greater than))
- (Integer comparator (equal))
- (Integer shifter (left))
- (Integer shifter (right))

To match the functionality of the ALU described in this thesis as close a possible, integer functionality has been included. It should be noted that synthesis tools are not smart enough to derive resource sharing between the floating-point and integer hardware like described in this thesis. The comparison is therefore of great interest, it clearly shows the benefits of combining integer and floating-point. There are some restrictions to the FMA hardware generated from Bishop's packages. Firstly, there is no support for multiple rounding modes. Per instance of the `mac` routine that implements FMA, only one rounding mode can be specified. Synthesis tools are unable to derive that two instances with different parameters share the majority of their needed hardware. Implementing all rounding modes would therefore make the comparison unfair. Secondly the sticky-bit had to be disabled due to version incompatibilities [3]. However, if these two minor differences are ignored, the functionality is equivalent, including the non-IEEE floating-point format.

The results of synthesis can be found in Table 7.8. The last column shows the absolute area and the area relatively to the implementations shown in the previous section (we compare the maximum achievable clock frequency realizations and a design at 200MHz). The architecture by Bishop does not use parallel alignment or LZA. This can immediately be noticed in terms of latency. Using three pipeline stages in combination with retiming and the exact same settings as described in Section 7.3, the low-power design could not be realized for clock frequencies over 360 MHz (2.75ns period constraint). Also in terms of area, parallel alignments appears to be beneficial when high clock frequencies are desired. Even though the components are 50% wider than for non-paralelized alignment, the synthesizer manages to produce an implementation that requires an equal amount of silicon area for a much faster datapath. The area overhead of this implementation compared to our implementation ranges from 20 to 44%. Considering

[3] The synthesizeable package has been written for and tested on DesignCompiler C-2006.06-SP4, version C-2009.06-SP5 was used for this comparison and appears to be incompatible.

Process		Area (μm^2)
LPHVT - 200MHz		
	Floating-Point	30961.3
	Floating-Point & Integer	41163.7 (∼144%)
LPHVT - 360MHz		
	Floating-Point	35178.2
	Floating-Point & Integer	52049.9 (∼128%)
GPSVT - 925MHz		
	Floating-Point	31946.7
	Floating-Point & Integer	50769.6 (∼120%)

Table 7.8: Area VHDL-2008 implementation

that the maximum clock frequencies for realization of of Bishop's implementation are lower, the 44% area overhead indication by the 200MHz implementation is likely the most realistic one.

To give an impression of the area that can be saved purely by integrating integer functionality in a floating-point datapath, we have also synthesized a datapath that is based on solely the floating-point hardware. The reduction achieved is certainly worthwhile. At a clock frequency of 200MHz, the area of an FMA unit with integer functionality integrated is 12644 μm^2 smaller than a design based on separate FMA and MAC units. This comes down to a reduction of approximately 30%.

7.4.2 Aeroflex Gaisler GRFPU

For SoCs or MPSoC based on the openly available 32-bit LEON processor by Gaisler Aeroflex, the GRFPU is the most natural choice for accelerated floating-point support. This floating-point unit is based of the Sparc V8 architecture which is also the base of the UltraSparc T2 floating-point unit described in Section 3.2. Hence, the architectures are very similar (most notably being the unpipelined hardware support for division and square). The Product information sheet mentions the following features:

- IEEE-754 compliant, supporting all rounding modes and exceptions
- Operations: add, subtract, multiply, divide, square-root, convert, compare, move, abs, negate
- Data formats: single and double precision (32- and 64-bit floats)
- Fully pipelined, 4 clock cycles latency for all operations except divide and square-root
- Non-blocking parallel execution of divide or square-root operations with other operations
- Supports all SPARC V8 floating-point instructions

The IEEE-754 mandatory instructions can be characterized as follows:

Instructions	Throughput	Latency	Description
FADD(S/D), FSUB(S/D), FMUL(S/D)	1	4	Add, subtract, multiply
FITO(S/D), F(S/D)TOI, FSTO(S/D)	1	4	Convert float/integer
FCMP(S/D), FCMPE(S/D)	1	4	Compare
FDIV(S/D)	16/17	16/17	Divide (single/double)
FSQRT(S/D)	24/25	24/25	Square-root (single/double)

Before looking at the physical properties of the GRFPU, it is good to determine the architectural differences. First of all it should be noted that this floating-point unit support double precision operands. It also uses one additional pipeline stage for the instructions we support natively in our datapath. However, it does not implement fused multiply-add. Furthermore it is interesting to see that division and square root require 16/17 and 24/25 clock cycles to complete respectively. This gives us confidence that a software implementation of these instructions is feasible. Looking at the Itanium FMA-based

implementation of these instructions (13 clock ticks before completion each), it should not be a problem to match the performance of the GRFPU.

According to Gaisler this floating-point unit can be produced at 400 MHz (400 MFLOPS) on a typical 130nm standard cell process, using approximately 80 kgates. In the worst case scenario (1200MHz), the new type of ALU we propose does not even come near that amount. Less than a fourth of the number of gates is needed while the GRFPU is almost certainly going to be outperformed based on the primary instructions (addition, subtraction and multiplication). Yet, it should not be forgotten that the GRFPU offers more precision and a slightly more elaborate instruction set.

7.5 Realistic SoC Integration Scenario

Earlier it was mentioned that an ALU is almost never stand-alone. Usually is is part of a processor or a system-on-chip. To conclude our discussion of realization related aspects, a realistic scenario of integrating the ALU in a modern state-of-the art MPSoC is investigated. The massively parallel processor breadboarding (MPPB) [2] architecture is chosen as an example platform. MPPB is part of ongoing research that focuses on the shortcoming of current DSP solutions for space exploration. The demand for digital signal processing from on-board applications in earth observation, science and telecommunication is continuously growing while the only European radiation hard DSP processor TSC21020F is old technology. The performance requirements of many applications often lead to ASIC development for single purpose in order to meet the requirements. FPGAs which provide more flexibility are not yet able to deliver the needed performance. MPPB fills the gap by providing a future-proof architecture that is both flexible and delivers high performance.

The most basic MPPB architecture (shown in Figure 7.5) consists of many different subsystems as is the case for most SoCs. The major components are the Xentium cores, the LEON processor, the NoC, the ADC/DAC, the memories and the AMBA bus system. Due to the presence of multiple Xentium cores, MPPB delivers unmatched processing power. The Xentium is a integer/fixed-point DSP core with a versatile instruction set which makes it a very (energy-)efficient processor. In this example, one of the Xentium cores is the candidate to be replaced by a combined floating-point/integer ALU. Despite the fact that the Xentium integer instruction set is much richer that the one we natively support in our ALU, the added floating-point support could be well worth the sacrifice. Besides, the most basic and most used integer instructions are supported.

The most important characteristics of the Xentium core are shown in Table 7.9. For good measurement: a single Xentium core consists of ten smaller functional units. Two of these units can perform multiplication and six can do addition and subtraction. These units operate on 32 or 40-bit wide data that can be loaded from and stored to local memories of the same size (i.e., 40-bit registers). Note that this is almost the amount needed for the floating-point format proposed for the solution shown in this thesis.

The numbers in Table 7.9 are based on 65nm low-power and general purpose implementations using exactly the same procedures as described in Section 7.3. A comparison is therefore easily made. Recall that the area of our ALU is approximately 0.04mm^2 for a performance driven implementation at 500MHz, and slightly less than 0.03mm^2 at 200MHz in low-power technology. Since the Xentiums run at 200MHz,

Property	CORE65LPHVT	CORE65GPSVT
Maximum Clock Frequency	220MHz	444MHz
Total Area (200MHz)	0.823mm^2	0.715mm^2
Combinatorial Area	0.363mm^2	0.252mm^2
Area Memories	0.305mm^2	0.317mm^2

Table 7.9: Xentium post-synthesis results

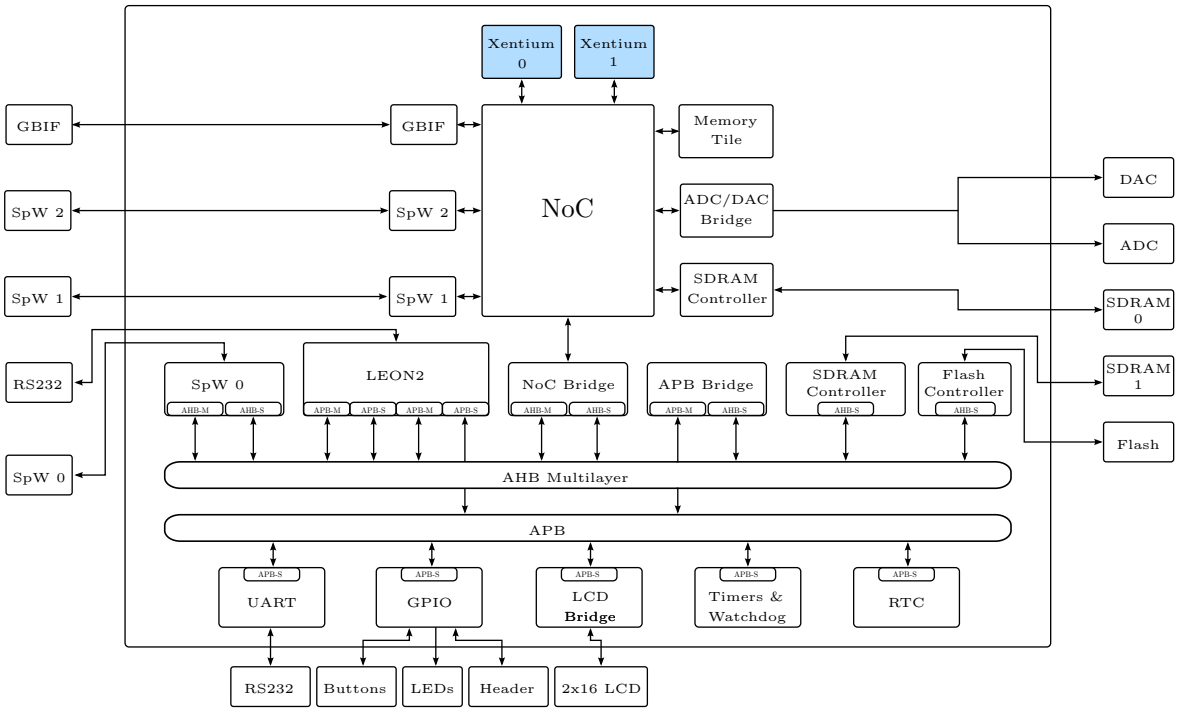


Figure 7.5: MPPB platform architecture

it should be no problem to drive the ALU at the same frequency as the Xentiums. At 0.363mm^2 (Xentium memories excluded), more than ten floating-point integer cores will fit in the area of a single Xentium. This indicates that also in terms of area, our ALU is hardly a bottleneck. Moreover, based on these number we believe it should even be possible to make the floating-point/integer an integral part of the Xentium. Unfortunately we can not say much about how the energy consumption of the ALU relates to for example a Xentium. At the time there are no meaningful power estimates (i.e., based on input switching activity) available for the MPPB components. We also have not yet computed actual energy consumption for some functions (e.g., finite impulse response (FIR) or fast fourier transform (FFT)), a good comparison can therefore not yet be made.

7.6 Summary

This chapter outlined a 65nm implementation approach, discussed physical properties and compared them with other architectures. The ALU can be implemented in low-power CMOS process at a maximum clock frequency of approximately 500MHz with an area of roughly 0.04mm^2. At 500MHz, the ALU consumes only 104.4μW/HMz when clock gating is used. With clock gating enabled, power consumption is reduced roughly 50%. A 200MHz implementation shows even more promising results for low-cost and energy-efficient hardware solutions. The area for 200MHz is only 0.03mm^2 and power consumption as low as 51.08μW/HMz. The highest clock frequency that can be reached with general purpose implementations is 1200MHz. With such a high speed circuit, a theoretical bandwidth of 2.4GFLOPs can be reached.

When the architecture is compared to less advanced architectures that do not combine integer and floating-point arithmetic, we see that by combining integer and floating-point arithmetic, about 20 to 40% area overhead is eliminated. Also the many architectural principles described in Chapter 5 and 6 yield a very fast datapath. The VHDL standard implementation of floating-point multiply-add, which

is already optimized for speed, can not be clocked higher than 360MHz while in theory our architecture could be clocked at 550MHz (500MHz is more realistic). The architecture also matches the performance of classical floating-point units such as the Sparc V8 architecture implemented by the GRFPU from Gaisler. Higher speed and less area are facts, however it should be mentioned that our instruction set is slightly less extensive and the GRFPU fully support double precision which requires a much larger multiplier. In terms of energy-efficiency, we can not yet say much. However, initial power estimates show satisfying results in the order of miliwats, which is within the low-power domain.

8 Verification

Without being properly tested, complex hardware components like ALUs, are likely to contain errors. Any error has to be detected and repaired before tape-out. A small and barely noticeable malfunction in the P5 Pentium floating-point unit cost Intel approximately $475.000.000. Hence, to verify the functional correctness of the ALU that we presented here, a flexible regression test has been used. Because a proper test for the floating-point arithmetic that we propose is not trivial, we shortly discuss the test in this chapter.

8.1 Test Bench

As usual with hardware designs, a testbench environment is placed around the ALU to test its behavior. The test setup is depicted in Figure 8.1. Instead of pre-determining what output is expected from the ALU, its results are compared against a reference design of which we know the result to be correct.

A *sequencer* reads instructions (opcodes and data) from a text-file (`instructions.bin`) and converts them to binary data. This binary data (the test set) is stored in a small register file. After reading all data, the sequencer dispatches the opcodes and instructions to both the implementation and reference ALU simultaneously. After the data has been processed by both ALUs, it is bit-wise compared in the *comparator*. Because the behavioral description of the reference design completes any instruction in just one clock cycle, a two cycle delay (buffer) is placed between the output of the reference design and the comparator. If a mismatch occurs, an error has been found and is reported. If the entire test set has been processed and no mismatches occurred, the test passes. Although this does not prove that the ALU is error free, it shows that the ALU behaves as expected and ensures the absence of errors anticipated by the test set.

A big advantage of this test setup is that no manual testing is required. Whenever a new feature or modification is implemented, all previous tests can easily be repeated to see if the modification has affected any of the earlier functionality (i.e., regression testing). Another advantage is the flexibility of the test set. The data can be generated in many different ways (Section 8.2) and moreover, the implementation design can easily be replaced by the post synthesis and post place-and-route netlists for testing with timing constraints and performing accurate power analysis.

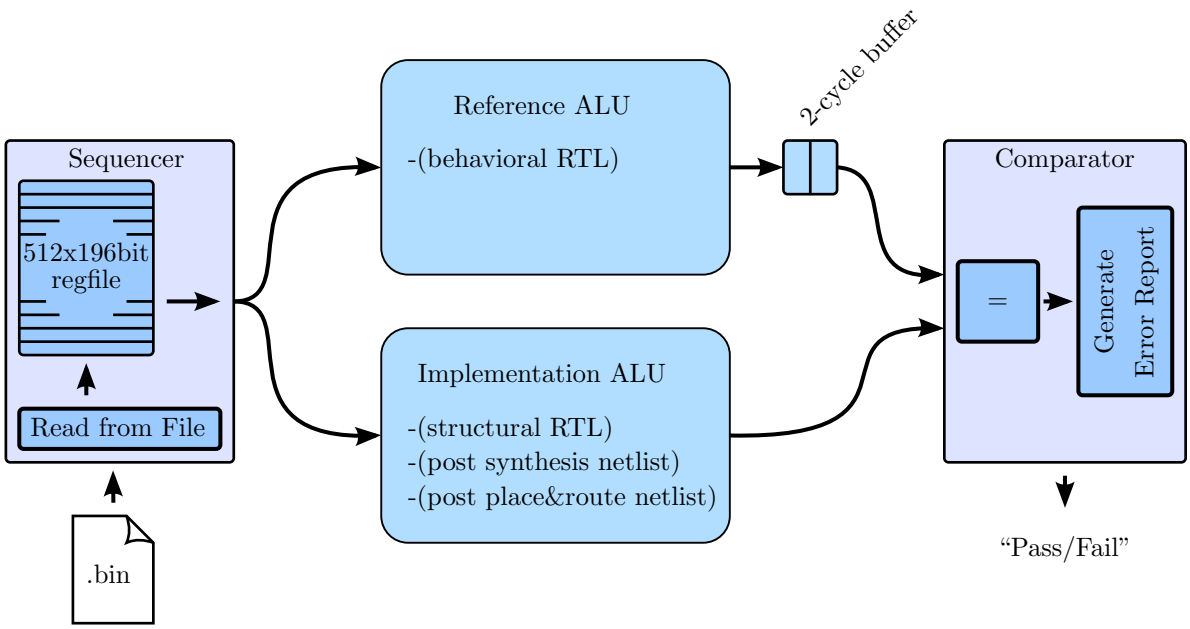


Figure 8.1: Testbench

8.1.1 Reference ALU

Finding a reference for floating-point arithmetic as proposed in Chapter 4 is difficult because this format can not be categorized as a valid IEEE representation. Indeed, most floating-point implementations (hardware or software) only support IEEE single, double and double-extended formats. A solution can be found in the VHDL-2008 floating-point support library. This library, originally introduced by David W. Bishop [36] to bridge the gap between VHDL-93 and VHDL-2008, provides floating-point arithmetic functionality compliant with IEEE-754. A floating-point format of any length (exponent and significand length being independent) can be defined. Arithmetic and logic can be performed on the defined format, according to the rules of IEEE-754:

VHDL example of floating-point instantiation

```
1 variable f_a : float(8 downto -32); -- 8-bit exponent, 32-bit significand
2 variable f_b : float(8 downto -32);
3 variable f_c : float(8 downto -32);
4 variable f_r : float(8 downto -32);
```

The package supports multiply-add (`mac`) and can be fine-tuned per operation. In the example below, denormalized number support has been disabled and three guard bits are used (i.e., one guard-bit, one round-bit and a sticky-bit). The round style has been set to 'Round to nearest, ties to even' and exception handling has been enabled:

VHDL example of floating-point fused multiply-add

```
1 f_r := mac(l => f_a, r => f_b, c => f_c, -- a*b+c
2   denormalize => false,                  -- denormalized numbers forced to 0
3   check_error => true,                   -- exception handling enabled
4   round_style => round_nearest,          -- round result to nearest
5   guard => 3);                           -- 1 guard-bit, 1 round-bit 1 sticky-bit
```

Besides arithmetic operation, logic operations such as the 'less than' compare are also available, as shown below:

VHDL example of floating-point compare

```
1 if(lt(l => f_a, r => f_b, denormalize => false, check_error => false)) then -- a  < b
2   status_reg_next <= STATUS_LOGIC_TRUE;
3 else                                                                      -- a !< b
4   status_reg_next <= STATUS_LOGIC_FALSE;
5 end if;
```

Recently this package has been standardized (i.e., accepted by the board of commissioners of the VHDL-1076 standard) and adopted in the `ieee` library. Even though the package has been approved, some issues still reside as we found out during our tests. The observed problems are discussed in Appendix B. Despite the small problems encountered, we believe that the VHDL-2008 floating-point library provides a viable solution for reference purposes. To verify the functionality of the package itself, single and double precision instantiations of it have been compared against a GCC floating-point multiply-accumulate implementation (`fmaf` for single precision and `fma` for double precision), running on Intel and AMD x86 processors.

8.2 Test Set

Test sets are collections of input data that are used to verify devices. Because the detection of errors heavily relies on a test set, it is important to apply a constructive test set to the device under verification. The test set for the test bench of Figure 8.1 is read from file (`instructions.bin`). This offers great flexibility, because as long as the data is formatted correctly, the data in this file can be generated by whichever means is practical. The test set that was used in this particular case, separates instructions (197 bits) by newlines. Each instruction begins with a 5-bit opcode (Table 6.1) followed by three floating-point or integer operands (Figure 4.3 and 4.4). When the instruction only requires two operands, the remaining input must be filled with zeros.

A practical means of generating floating-point data for this test bench is to use the floating-point package from Bishop. The `to_float` instruction is capable of converting the VHDL real datatype to any given floating-point format. The example below shows the creation of a binary floating-point word representing 5.20 with an 8-bit exponent and 32-bit significand:

VHDL example of floating-point test data generation

```
1 a := to_slv((to_float(5.20, 8, 32))); -- convert 5.2 to float and then to binary
```

In this format the operands can easily be written to file.

The test that was applied to verify our design primarily focuses on corner cases. This means it mostly tests operations such as arithmetic including zeros and infinity and overflow/underflow detection, but also the sign-bit is tested extensively. Although this test covers a large number of the potential problems, further testing is needed as discussed in Section 9.3.

9 Conclusion

9.1 Introduction

This chapter presents an overview of our findings, an evaluation of these findings, some recommendations for improvement and the final conclusion of this thesis. The primary goal of this thesis was to investigate the possibilities for increasing floating-point hardware usage by incorporating integer functionality in the arithmetic logic datapath, such that the floating-point unit can be scheduled for integer operations. During the search for such an architecture, that also needed to be both energy-efficient and low-cost as well as high performance, many (sometimes unforeseen) interesting and useful discoveries were made. Before a conclusion is given, we first summarize our most important findings and pinpoint the successes and shortcomings of the architecture presented in this thesis.

9.2 Summary

In Chapter 1 we set out three research topics for this thesis. The first one was *"What floating-point and integer formats can most efficiently be combined?"* After comparing several different floating-point and integer storage formats in Chapter 2, it should be clear by now that IEEE-defined floating-point formats are almost obligatory. Similarly, two's complement notation is the standard for representing signed integer operands. For a low-cost floating-point and integer solution, IEEE-754 single precision and 32-bit two's complement integer is a good combination (double precision would be too much of a bottleneck). However, because only 23 bits of the floating-point numbers are used for the significand, the arithmetic hardware for single precision floating-point is not sufficient for 32-bit integer operands. The floating-point format we propose instead, is an extended single precision based format. Eight additional bits in the significand are used (i.e., a sign-bit, an 8-bit exponent and a 32-bit significand) in order to make efficient use of the the hardware that is needed for 32-bit integer arithmetic. The result is a regular datapath (Chapter 4) that supports both common 32-bit integer input (two's complement) and a floating-point format that strongly resembles single precision format, but uses eight more fractional bits. A major drawback is that the floating-point format is no longer entirely IEEE-754 compatible (Chapter 4). In addition, storage will be either very inefficient (exponent and sign-bit have to be store separately from the significand if standard 32-bit registers are used) or custom unconventional memories are needed. On the other hand, the eight additional bits offer more precision (Appendix A) that partially make up for the absence of double precision support.

The second topic states: *"What floating-point architectures are suitable for low-cost energy efficient hardware solutions?"*. From the many architectures in existence (Chapter 3), we believe the FMA

(A×B+C) architecture is currently one of the more attractive ones. Multiply-accumulate (or multiply-add) instructions are very common and greatly benefit from the FMA architecture. Not only is FMA faster, it also yields more accurate results because only a single round operation is performed. Single multiply or add/subtract operations can easily be derived from multiply-add (A×1+C and A×B+0). A FMA unit is however not a floating-point unit. Division, square root, integer/floating-point conversion and comparison instructions are mandatory for IEEE-754 compliance. Division and square root are particularly expensive to implement in hardware. These instructions are only sparsely needed which made us decide not to support them in hardware. Chapter 3 gave a short overview of the Itanium processor, which implements both division and square root in software, requiring only 11 instructions and additional support in the form of a (hardware) lookup table. Comparisons are more common and require much less resources. These operations can be implemented natively in the architecture. Note that for full comparison (>, = and <), only two out of three primary compares have to be implemented, the third can be derived. Conversions are omitted entirely since we see little need for such functionality.

The last en most important question is: *"How can integer operation most efficiently be mapped to floating-point hardware?"* This question can be answered by looking deeply into the floating-point datapath and distill the basic arithmetic that is used within. For the primary floating-point instructions directly supported by FMA (i.e., multiply-add, addition, multiplication and subtraction), the most prominent components (also the components suitable to process integer data) needed are a high-speed multiplier, an adder/subtracter, a right shifter for alignment and a left shifter for normalization. Multiply-accumulate, multiplication, addition and subtraction, as well as shift left and right (integer) instructions can be mapped to these components. Sharing these components between two's complement integer and sign-magnitude floating-point can be as trivial as sign extension (Chapter 6) but sometimes also requires sophisticated architectural exploits, such as modified Booth encoding (Chapter 5). In most cases the components themselves do not need any modification at all. A lot of integer functionality can be obtained from a floating-point datapath by cleverly routing the dataflow (Chapter 6).

Besides research topics, also a number of requirements were formulated. Firstly it was stated that the new architecture should at least support multiplication and addition of both floating-point and integer operands. The FMA architecture we propose meets this requirement and even supersedes it. Floating-point fused multiply-add is more accurate and faster than separate multiply and add instructions. The FMA implementation we presented can also perform subtraction on the adder, by means of a technique called End-Around Carry Addition (Chapter 5). The entire multiply-add core functionality (the multiplier, adder and additional hardware to support subtraction) is re-used for integer MAC. Integer multiplication and addition/subtraction can be derived in a similar fashion as for floating-point (i.e., by using literals 1 and 0).

The second requirement was that the pipeline would be limited to two or three stages. This is a severely restrictive factor considering the last requirement is that the design should be fully synthesizeable in a 65 or 90nm low-power process. The combination of these two requirements has defined the architecture as has been presented in this thesis. Low-power technology and long combinatorial paths such as a floating-point datapath do not combine easily. It is not difficult to see why. For a full multiply-add operation, two 32-bit significands first need to be multiplied. The third significand then needs to be shifted so that the exponents match. Only then can the product and aligned addend be added. Once addition has been performed, the result needs to be normalized by shifting the significand to the left until the MSB is `1`. This requires that the number of leading zeros is determined even after being normalized, the result still needs to be rounded. We have not even mentioned the many exceptions and irregularities the datapath has to account for. Often a pipeline of six or more stages is used to implement all this functionality with a reasonable latency/throughput.

For short pipelines, multiplication and alignment is a serious bottleneck. Because normally we always shift the smallest operand for a floating-point additions, this means that for multiply-add we would have to wait until the multiplication has completed. Two or three stages are needed to prevent this massive delay of lowering the maximum clock frequency to unacceptable levels. By always aligning the addend (C operand), we can parallelize alignment with multiplication such that parts of the delay overlap. To

be able to align C not knowing if this is the smallest operand, it needs to be rewritten such that its fraction is entirely located to the left of the radix point of the product. This requires that the datapath is extended, the shifter and adder become 50% larger to support parallel alignment. Despite the increased area, this optimization pays out (also in terms of area for high speed implementations) as we have shown in Chapter 7. In addition to paralel alignment, the addition can be optimized to further reduce latency. Fast multipliers consist of many CSAs that are combined into two partial products in carry-save format. The sum and carry have to be added in a normal fashion to obtain the actual product. If such a multiplier is used, FMA can be implemented elegantly by inserting another CSA that compresses the aligned added into the carry and sum before the final addition is performed. Only one carry-propagation is needed then, a significant reduction considering adder-carry propagations are among the longest delays in ALUs. Another crucial optimization is LZA. By predicting where the leading zero will be based on the input of the final adder, another stage can be eliminated from the datapath. LZA is accompanied by LZD, which actually transforms the prediction into a binary number that can be used to shift the summation for normalization. A new, more area-efficient, way for LZD has been found and presented. With this new approach up to 50% area can be saved for LZD applied to input that is not a power of two. When optimizations such as the ones described above are applied, a three stage pipeline can be found that is reasonably balanced. Registers are placed after multiplication and between addition and normalization (Figure 4.13). The best results are obtained if retiming is enabled during synthesis. However, also without retiming the three stage pipeline reaches high clock frequencies (500MHz in low-power and 1200MHz in general purpose technology).

9.2.1 Contributions

In retrospect, several important contributions have been made by answering the questions raised at the begin of this thesis:

First and foremost the definition of a new architecture for combined integer and floating-point arithmetic and logic. A fully functional ALU design is presented in detail. This architecture is suitable for energy-efficient low-cost hardware platforms and provides high (multiply-add) performance. To prove the feasibility of this concept, the architecture has been implemented into a netlist suitable for IC manufacturing. Detailed and accurate timing, area and power consumption estimates are provided.

Secondly, several contributions are made in the form of improvements in area efficiency of various hardware components required for the proposed architecture. One of the more notable optimizations is a modification for area optimizing the energy-efficient leading zero detect circuit by [1]. With the new approach, area savings up to almost 50% can be achieved.

9.3 Evaluation and Recommendations for Improvement

Chapter 7 showed that the architecture that was developed as part of this thesis can be realized with attractive physical properties. The result is high performance as well as low-cost and energy-efficient when manufactured in low-power technology. However, there is still room for further improvement. More importantly, the current design still needs minor modifications for numerical correctness.

9.3.1 Resolving Known Issues and Further Testing

Several small issues still reside in the architecture presented so far. The most critical problem is overflow due to rounding. Because the overflow check is performed after normalization, the increment of rounding is not taken into account. In some cases this increment may cause overflow in the significand. To account for this overflow, the overflow check should be performed after rounding which requires only a minor

change to the architecture. An elegant solution would be to increase the length of the incrementer by one bit. If the MSB of the result is 1 after rounding, we simply take the 32 MSBs including the overflow-bit and increment the exponent. The overflow check performed to the exponent field, as described in Section 6.8.2, still applies. If the exponent overflows then the result overflows.

Additionally some special cases of zero-arithmetic produce a faulty sign-bit. To be more precise, when the product of A×B is -0 and C is also -0, the sign-bit of the result should be minus (1). The current implementation produces a positive sign-bit for this case. An easy solution would be to add another exception for this specific occasion. However, maybe a better solution exists that accounts for all exceptions in a more general fashion. It would be worthwhile to search for a more elegant way of generating the correct sign-bit.

Aside from these two known issues described above, it may be possible that more problems, which are not found by the current test set (Chapter 8), are still hidden within the design. Currently only corner cases and an arbitrary set of regular input is used to test the design. Smarter and more profound testing is needed to obtain better insight in the functional correctness of the design. Well-known tests that we plan to use for further validation are for example 'Testfloat' [43] which is based on the high quality software IEEE-754 implementation 'SoftFloat', and the 'IEEE 754 Compliance Checker (IeeeCC754)' [44] which is even accompanied by a publication [45].

9.3.2 Scalability and Portability

The structural VHDL description is parameterized. This means that the architecture can easily be realized for other floating-point/integer combinations as well (e.g., double precision and 64-bit integer). However, the new approach for LZD has not yet been parameterized. Currently the best combination of smaller LZD circuits has been designed by hand for an efficient 102-bit LZD. The new algorithm should be formulated as a uniform algorithm such that a general case is described.

Currently the design still relies on the DesignWare library for the partial product multiplier and the CSA. The CSA should be easy to re-produce in a more vendor independent way. The partial product multiplier unfortunately is not. It will be difficult to match the efficiency of DesignWare (especially in Design Compiler Synthesis) because these components are the result of years of accumulated experience. Yet in the future it may be needed to manually re-design the DesignWare components, to make entirely vendor independent version of the architecture's HDL description.

9.3.3 Extending the Instruction Set

Although the ALU provides the basic set of instructions needed for most practical applications, it does not yet provide the full range as found in many more mature floating-point architectures. Furthermore, there are still opportunities to exploit the hardware for integer purposes even further than has been done.

Software Routines

As discussed in earlier chapters, some basic instructions such as division and square root have deliberately not been implemented in hardware for area efficiency. The area required for a hardware implementation of these instructions would dwarf the current design [46], eliminating all area minimizing efforts presented in this thesis. It was also mentioned that these instructions can efficiently be implemented as software routines due to properties of FMA (e.g., single instruction multiply-add for accumulation of iterations and single rounding for high precision). Iterative approximation algorithms have been proven to produce results that are equivalent to infinitely precise computations rounded to a finite number of bits [47]. The Itanium architecture, discussed in Section 3.3, has demonstrated that software implementation of

division and square root is a feasible solution. The best known algorithm of software approximation for division and square root is the Newton Raphson method [13]. However, more recently Goldschmidt's algorithm has also been investigated and shows promising results [47]. It should be noted that in order to implement these software routines, lookup tables are required for initial estimates [13]. These lookup tables are usually implemented in hardware. Therefore, hardware is still needed though considerably less than for an actual hardware implementation of both operations.

Although operations like division and square root should be implemented first, more functionality needs to be added for full IEEE-754 compliance. Conversion from and to integer is required and several simpler transformations such as sign-bit swapping and producing absolute values. Furthermore, the differences for integer and floating-point approximations of division and square root should be investigated. Neither division or square root have been implemented for integer operations.

Despite the formal need for these operations, one could ask if they are really needed before implementing the routines. In many cases division, and square root to an even larger extend, is not often needed. Digital filters and Fourier transforms can perfectly be implemented on solely a FMA unit.

Hardware Extensions

Several new features could be added to the datapath. A simple modification that could be valuable is to provide an actual single precision floating-point integerchange format. Although being more precise, the floating-point format we proposed in Chapter 4 could be a disadvantage for external communication (i.e., when the receiver strictly expects 32-bit single precision). By applying the round algorithms described in Section 5.6 in a more flexible way, we could produce actual single precision numbers. The most critical resources are already available. Using multiplexers we could simply apply rounding to the 23 normalized MSBs instead of 32 bits. The downside of this feature is that rounding is located in the last pipeline stage. Currently this is already the critical path which means the modification would affect the performance.

Another interesting feature would be SIMD-like (vector) instruction support. By partitioning the major components of the datapath shown in Figure 6.1, multiple integer instructions can be performed in a single clock cycle. In the current design, many components already have the capacity to perform multiple smaller integer operations. For example the adder, which is 102 bits wide. By partitioning the adder as shown in Figure 9.1, two 32-bit integer additions can be performed in a single clock cycle. Even the output of the ALU is ready for vectorized hardware utilization. In the case of addition, only correct partitioning and overflow detection need to be added. Both are relatively easy and cheap to implement. Other instructions can also be vectorized, although more effort is required. The right shifter, used for alignment, is 101 bits wide. By mapping the input in a similar way as done with the adder, two shifts can be performed at the same time, given that the shift amount does not differ. The left shifter (normalization) would have to be made wider in order to support two 32-bit input operands. Multiplication and multiply-accumulate are the most difficult to implement. Currently the multiplier can only perform one 32×32 multiplication, a second multiplier would be needed for vector support. However, multiple passes could be used to mimic a SIMD architecture. Even though this would not yield increased performance, a full vector-instruction set could be provided.

9.3.4 Reducing Area and Increasing Performance

Due to the nature of the architecture itself and the many fine grained optimizations that have already been applied, reducing the area of the datapath is difficult without taking radical steps. That being said, for hardware that is on a really tight area budget, the Cell approach to rounding may be useful (i.e., truncate every result after normalization). Round to zero requires absolutely no resources and can therefore still reduce the total area of our design. However, Table 7.5 shows that removing the rounding

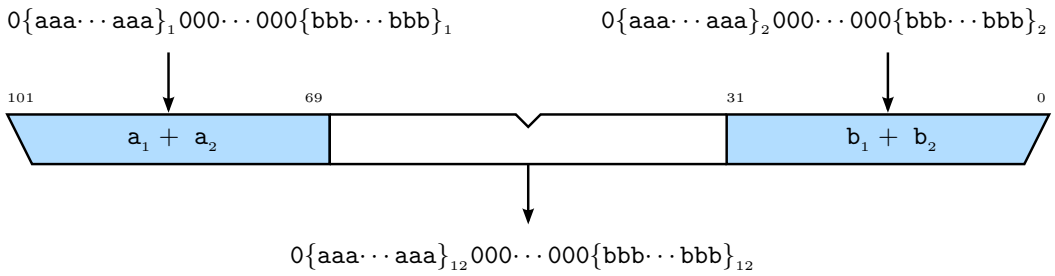


Figure 9.1: Vectorized (SIMD) addition

logic, including the sticky-bit generation unit, is only a marginal improvement in terms of area. A mere 3.4% of the total area can be removed. In terms of latency, the removal of IEEE rounding may be more useful, since the round logic is part of the critical path (Section 7.3.1). Appendix A discusses the effect of eight addition precision bits. It would be interesting to know if this additional precision could replace the 'round to nearest even' round mode (effectively elimination the need for a round block). The research of Appendix A would have to be extended by performing the same computations on the new architecture.

To increase performance, the dual path adder that was described in Section 3.5 might be a good start.

9.3.5 Reducing Dynamic Power Consumption

Power analysis shows that the ALU is already quite energy efficient. Especially with the ST 65nm LPHVT library, power consumption estimates look very promising. Using low power technology is an effective way to reduce the overall power consumption of a chip. By using a low-power library, static power dissipation and current leakage are minimized. Dynamic power dissipation is also a major factor in energy-efficiency. Currently only automatic clock gating (Section 7.3.3) is applied to reduce power dissipation caused by charging and discharging of load capacitances (i.e., switching from logical high to low and vice versa). Although clock gating is a major improvement, power consumption can be reduced even further. A technique called power gating could be used. Appendix D shows that although the utilization of the datapath is very high, there are still large static parts during each instruction. Switching these parts off entirely may yield better results than clock gating. In addition, we could apply architectural changes such that dynamic switching is eliminated in non-critical parts of the datapath. Figure D.3 illustrates that only a very small part of the datapath is actually used. By keeping all input of the components not used completely stable, no power is dissipated due to charging or discharging.

9.3.6 Other Research Directions

Aside from the hardware changes and instruction set extensions proposed above, other research directions may be worthwhile to investigate. The first thing that comes to mind is the inefficient storage of the floating exponents. Only nine out of 32 bits (eight bits for the exponent and one for the sign-bit) are used when local floating-point and integer registers are shared. Even if the inefficiency can be solved elegantly, there is still the problem of matching exponents with significands. A straightforward solution would be to always store the exponent and significand adjacently. However, because integers require only a single register, segmentation of the register file will quickly become an issue. Algorithms such as heap compaction could be borrowed from the field of compiler design.

Another direction is to address the scheduling problems that appear when integer instructions are interleaved with floating-point instructions. A transition from integer to floating-point mode requires flushing the pipeline. Although the pipeline in not very deep, each transition means a penalty of one clock cycle.

To keep the number of transitions to a minimum, instructions can be buffered and executed later if scheduling allows it. However, this brings many problems (e.g., starvation) that operating systems also have to deal with. In [22] compiler techniques are described that are aware of these problems. The authors indicate that the modifications that need to be applied to established compilers to support this kind of hardware are manageable. However, their findings are merely a first step in the right direction. Much more research is needed to get the most potential out of the hardware.

9.4 Conclusion

With the design and implementation of a new architecture for floating-point and integer arithmetic and logic, the objective of this thesis has been fulfilled. Namely, to find an architecture that is suitable for low-cost floating-point support in energy-efficient hardware platforms. An important lesson that can be learned from this work is that the success of such an architecture can not be credited to one or a few techniques. Only if every aspect is thoroughly analyzed and optimized to the fine details, can high-performance be combined with low-power and small area. The 65nm implementation has shown that the architecture is feasible and is well within the bounds of justification for energy-efficient and low-cost hardware solutions.

A typical 65nm low-power implementation at 200MHz has an area of 0.03mm^2 and consumes about 104.4μW/MHz based on an average switching activity of 50%. The theoretical floating-point bandwith of the design is then 400MFLOPs due to the powerful fused multiply-add architecture. If low-power is of less concern, the performance can be streched to 2.4GFLOPs with a 1200MHz implementation in general purpose technology. Because clock gating can be applied very effectively in this architecture, it is also highly energy-efficient. After automatic clock gate insertion, the energy consumption is reduced to about 50%.

Finally, the use of the floating-point hardware has been increased by incorporating integer functionality in the datapath, by re-using the basic floating-point arithmetic blocks. Full integer MAC can be implemented as well as shift left and right, without using additional (noteworthy) hardware. The overhead of combining integer with floating-point arithmetic and logic in a single datapath is minimal. In terms of area the additional resources can be neglected, less than 2% ot the total area is spent on multiplexers and hardware configuring to support integer instructions. Also in terms of latency, the consequences can hardly be noticed. Based on the findings in this theses, it can be concluded that integer and floating-point can indeed be combined efficiently and that floating-point is feasible for low-cost and low-power hardware platforms.

A

Quantization Effects

The precision of an arbitrary number represented in floating-point notation is directly correlated to the numbers of bits used in the significand. To increase precision, one simply increases the number of bits in the significand. Many processors with floating-point support, offer two levels of precision: IEEE-754 single precision (23+1 bits significand) and double precision (52+1 bits significand). The format proposed in Chapter 4 can be categorized in between single and double precision. It is basically single precision format with eight additional bits in the significand. To obtain some insight in how the additional bits affect precision, we look at *quantization effects*.

A.1 Quantization

Quantization is the process of converting continuous values (infinitely precise) to a finite set of discrete values. This introduces an error in the values (due to rounding or truncation) which is referred to as the quantization error. It does not require a lot of imagination to see that more bits result in smaller quantization errors. Of course quantization also occurs when the precision of (already) digital numbers is lowered, hence the quantization errors provide a good base for investigation of the advantages of additional bits in the significand. In many scientific and engineering fields (e.g., DSP), the signal to noise ratio (SNR) is an often-used measure to quantify how much a signal has been corrupted by noise. SNR is defined as the power ratio between a meaningful signal and the unwanted background noise:

$$SNR = \frac{P_{signal}}{P_{noise}} = \left(\frac{A_{signal}}{A_{noise}}\right)^2 \tag{A.1}$$

where A is the root mean square (RMS) amplitude. SNRs are often expressed using the logarithmic decibel scale, such that:

$$SNR = 10 \cdot log_{10}\left(\frac{A_{signal}}{A_{noise}}\right)^2 = 20 \cdot log_{10}\left(\frac{A_{signal}}{A_{noise}}\right) \tag{A.2}$$

When the corruption is a result of quantization effects, this ratio is called the signal to quantization noise ratio (SQNR). The advantage of having more precision bits will be shown by comparing SQNRs. We distinguish two different types of quantization noise. The noise introduced by quantization itself, and the additional noise resulting from operations that are performed in finite precision (a direct result of quantization). Figure A.1 shows how these ratios are found.

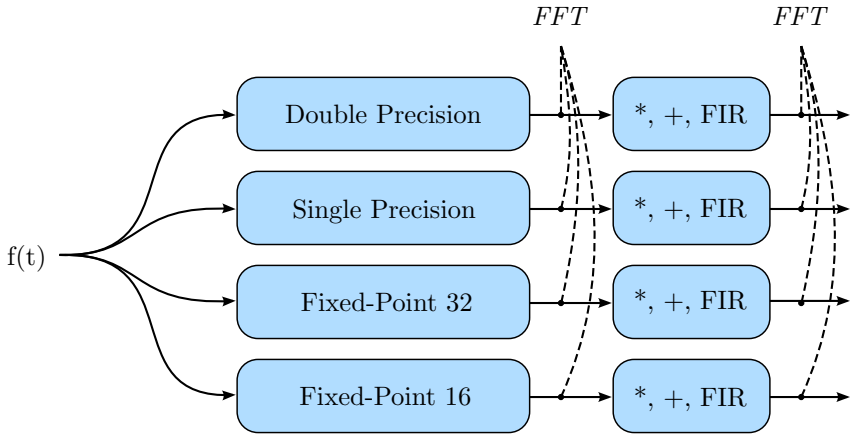


Figure A.1: Signal to quantization noise ratio flow

First an input signal *f(t)* (or any arbitrary sequence of numbers) is quantized. For practical reasons, a double precision representation of a continuous signal is used as the reference (uncorrupted) signal [1]. Several different levels of quantization are applied to the reference signal: single precision floating-point, 32-bit fixed-point and 16-bit fixed-point. Unfortunately, no tools are available that easily provide the means to perform floating-point computation in formats other than specified in IEEE-754. Therefore we have chosen to mimic our increased precision format by a fixed-point representation with 32 fractional bits. The input is chosen such that the advantage of the dynamic range, floating-point numbers have over fixed-point numbers, is minimal. The input is normalized between 0.0 and 1.0 (Figure A.2)). The 16-bit fixed-point representation is included because this format is commonly found in embedded (signal) processors. After quantization, a 1024 point FFT is performed. Analyzing the spectral output of a FFT to determine SQNR is not only common practice, it also shows a *noise floor* that can be used to quickly get an impression of the quantization error/noise. The input signal is a perfect sine wave that fits exactly in a 1024 point FFT window. This ensures that effects such as spectral leakage will be minimal. Before we look any further into the flow of Figure A.1, we first discuss the spectral output (of the first FFT) after purely quantizing the signal.

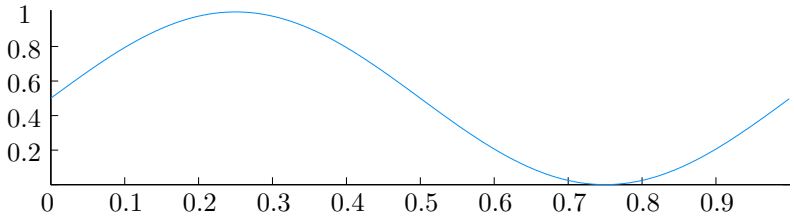


Figure A.2: Normalized input signal *f(t)*

In Figure A.2 we see the normalized input signal. The fixed-point representations are signed with a 1-bit integer part and 16 or 32-bit fraction. Therefore, the input signal must be normalized between 0.0 and 1.0. If an infinitely precise FFT is applied to the non-quantized input, we would see a spectral output consisting of only a large spike at the frequency of the sine, and an infinitely low noise floor. Figure A.3 shows the FFT output of the different levels of quantization of the sine signal. By default the FFT algorithm produces results for the inverse frequencies of a signal, resulting in a second spike (all the data to the right of the center in Figure A.3). We have chosen to plot this raw FFT output here. The different levels of noise resulting from quantization can clearly be distinguished by the level of the noise floors. The FFT output has been normalized and plotted on logarithmic scale, such that the peaks

[1] For this investigation we used Matlab R2010b [48], which only supports floating-point numbers up to double precision. This means double precision can not be included in our analysis.

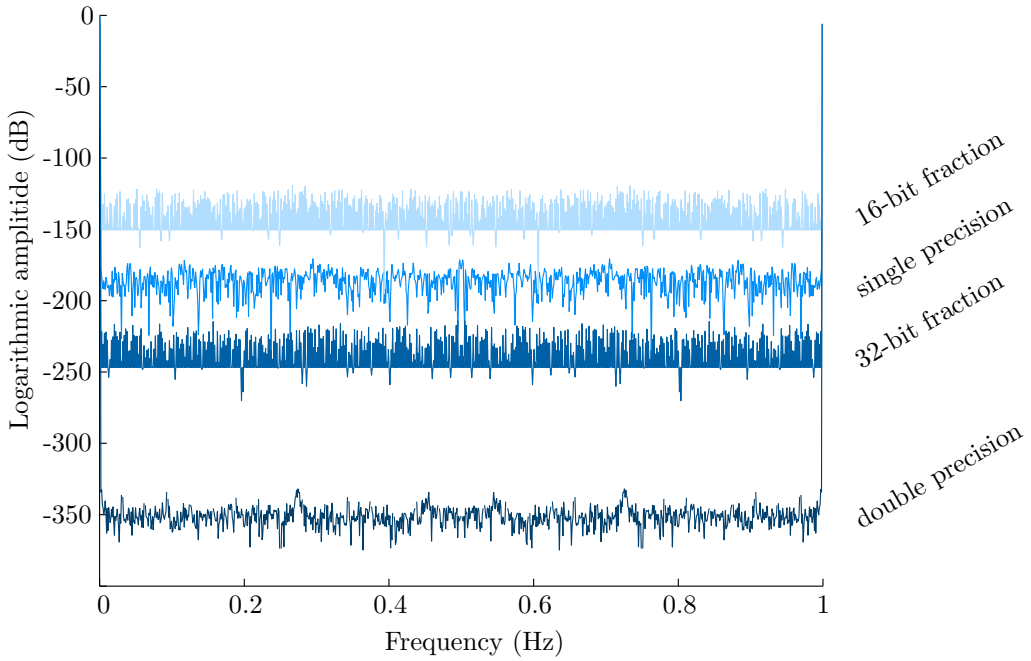


Figure A.3: FFT output of quantized sine signal

land exactly on the 0dB level. In that case, a lower noise floor means less quantization effects and thus a more precise representation.

When we apply Equation A.2 to the FFT output of Figure A.3, we find the corresponding SQNR. The measured SQNRs can be found in the first column of Table A.1. In most literature on analog to digital conversion, one often finds the following formula to approximate the SNR of an arbitrary fixed-point format with Q bits in the fraction [49]:

$$SQNR_{fixed} = 6.02 \cdot Q + 1.761dB \tag{A.3}$$

Less known is the similar approximation for floating-point numbers [50]:

$$SQNR_{float} = 6.02 \cdot Q + 7.44dB \tag{A.4}$$

The SQNRs as predicted by these relations have been listed in Table A.1 as well. As expected, these theoretical numbers are very close to the result we found from our experiment.

From these finding we can conclude that using eight additional significand bits will give us approximately 50dB more SQNR, which is approximately 33% more than standard single precision floating-point.

Precision	**Measured SQNR** (dB)	**Predicted SQNR** (dB)
Single Precision	153.426	151.92
Fixed-Point 32-bit	199.028	200.421
Fixed-Point 16-bit	103.380	104.101

Table A.1: Measured and predicted SQNR

Normally this would also cost 33% more bits (8/24 bits), however, since the hardware is already 32 bits wide to support 32-bit integer operations, the additional precision is free. Table A.1 also clearly shows that the SNR increases linearly with the number of bits used in the fraction. The SNR of 16-bit fixed-point is almost half of that of the 32-bit fixed-point representation. This tells us that there is a very simple relation between the number of bit and the maximum SNR that can be achieved. To double the SNR one doubles the number of fractional bits (not taking the bounady cases such as $Q < 2$ into account).

A.2 Operations

Now that we have shown the effects of quantization we can proceed to the analysis of operations. Unfortunately there is no straightforward and fair way to compare fixed-point and floating-point operations. Fixed-point precision degrades much faster than floating-point due to the lack of an exponent. In many cases a floating-point number can be adjusted by shifting the significand to the left and decreasing the exponent accordingly. With fixed-point operations, the least significand bit are inherently shifted out of the representable range. Therefore, substituting 32-bit fixed-point for 32-bit significand floating-point numbers, is less meaningful than initially thought. The following result should therefore be considered only as really rough estimates.

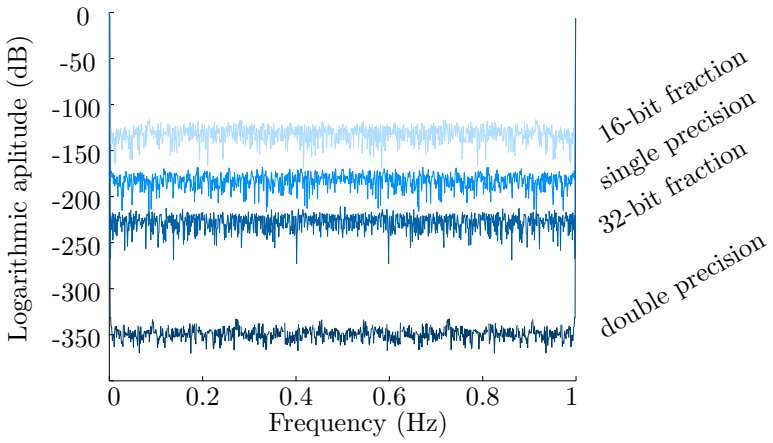
A.2.1 Multiplication

The two main operations supported by the ALU, addition and multiplication, are tested. First we repeatedly multiply the input signal with a number of constants between 0.0 and 1.0. These constants are representable exactly in binary fractions such that the precision of the constants does not influence the outcome. Each multiplication introduces additional noise, which means that if we keep accumulation multiplications the noise will constantly increase. The severity of this effect greatly depends on the precision of the format used. When the operations are finished, we take the FFT of the output and compare it to the other levels of precision. We can also determine the absolute difference in noise before and after the operation to see what the quantization effect of a certain operation is.

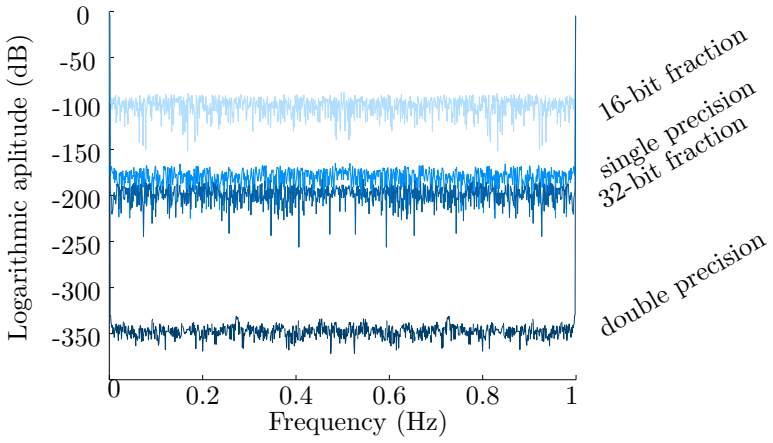
The spectral output of the 1024 point FFTs after multiplication(s) have been visualized in Figure A.4. After one multiplications the effects can hardly be noticed (Figure A.4(a)). After four multiplications, we can see that the noise floor of single and double precision floating-point remains the same while the noise floor of the fixed-point format has moved up noticeably. After even more multiplications (Figure A.4(c)), we can see that the precision of a 32 bits fraction in fixed-point is already at the level of single precision floating-point, which has only 24 effective bits. This can be explained by the fact that in fixed-point multiplication with constants smaller than 1 basically means shifting precision bits out of the fixed-point range. The floating-point formats have hardly changed because of the flexibility provided by the exponent. The actual SQNRs (shown in Table A.2) confirms this observation.

A.2.2 Addition

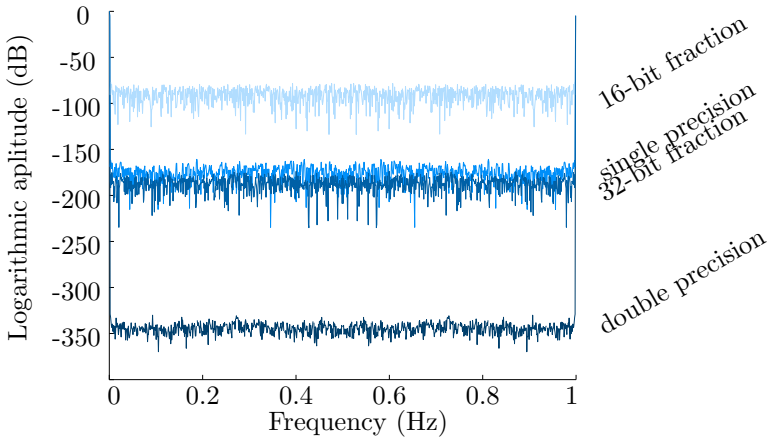
The effect of addition are considerably less than for multiplication. This is not difficult to see because multiplication is nothing more than repeated addition. To test additions we subtract (actual addition would cause clipping in the signal because the fixed-point notation can not exceed 1.0) a number of small constants. After the 'additions' are completed, we use the FFT as the input for our analysis again. The results can be found in Figure A.5. For addition the SQNR of fixed-point is much better than for multiplication. Even after eight additions, the differences are minimal. In addition, when we look at the actual SQNRs in Table A.2, the advantages of floating-point an not as evident as for multiplication.



(a) After one multiplication

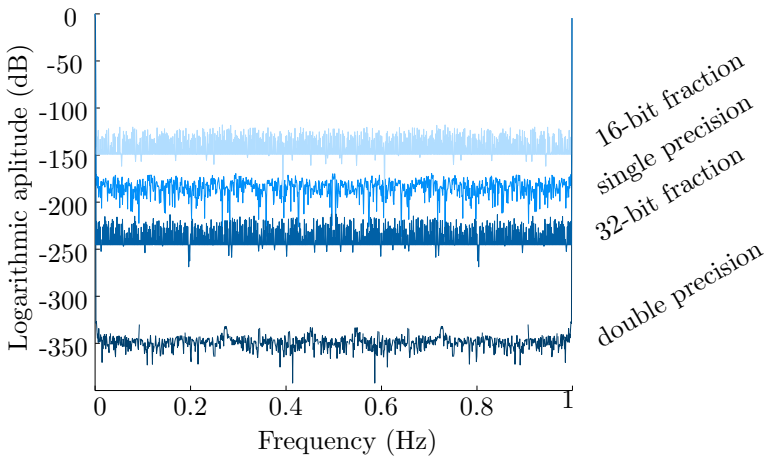


(b) After four multiplications

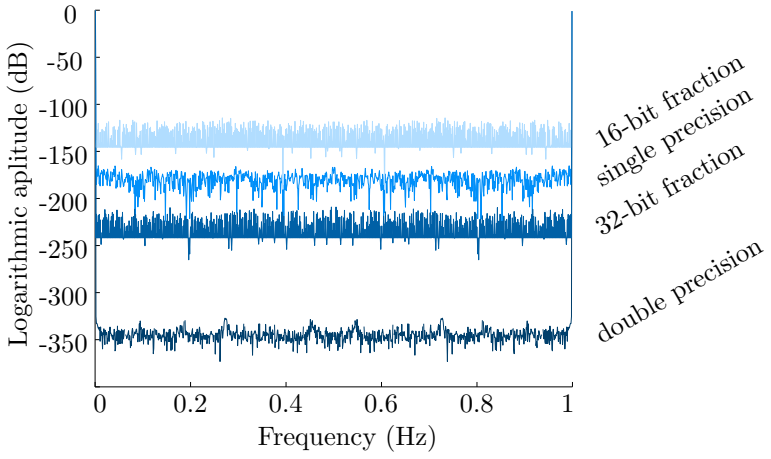


(c) After eight multiplications

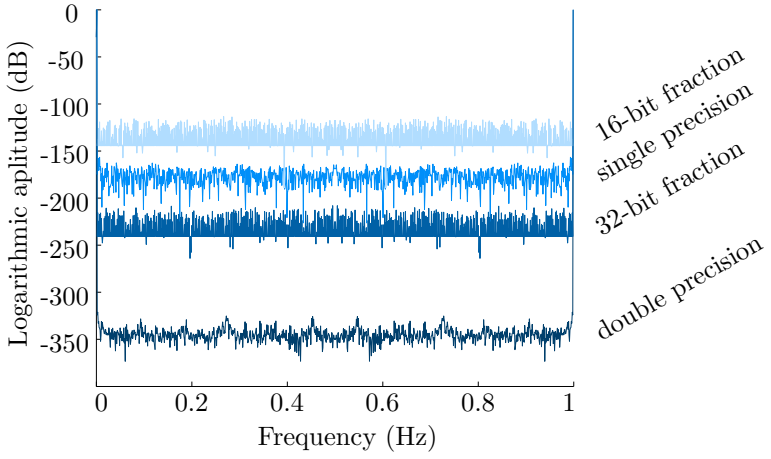
Figure A.4: 1024-FFT noise floors after multiplication



(a) After one addition



(b) After four additions



(c) After eight additions

Figure A.5: 1024-FFT noise floors after addition

Single Multiplication	
Single Precision	149.5985
32-bit Fraction	194.6003
16-bit Fraction	98.2395
Four Accumulated Multiplications	
Single Precision	147.9739
32-bit Fraction	166.4462
16-bit Fraction	70.0052
Eight Accumulated Multiplications	
Single Precision	143.9675
32-bit Fraction	156.3405
16-bit Fraction	59.6658

Single Addition	
Single Precision	151.4651
32-bit Fraction	197.0935
16-bit fraction	100.9864
Four Accumulated Additions	
Single Precision	149.9449
32-bit Fraction	194.8466
16-bit Fraction	99.9449
Eight Accumulated Additions	
single Precision	144.3636
32-bit Fraction	194.0638
16-bit Fraction	95.6171

Table A.2: Signal to quantization noise ratio (in dB) after multiplication and addition

An even larger number (32) of multiplications and addition also produces interesting results. Single precision floating-point representation loses some SQNR but remains roughly the same. Fixed-point on the other hand suffers greatly from larger numbers of accumulated operation. After 32 multiplications, 16-bit fixed-point precision has degraded to the point where a division by zero occurs, and 32-bit fixed-point SQNR has been reduced to 39.3637dB. Single precision floating-point on the other hand results in a very acceptable 137.1697dB. Surprisingly, addition performs even worse in some cases. Both 16-bit and 32-bit fixed-point have a SQNR of roughly 12.7dB. A big difference with the 147.1921dB of single precision floating-point.

So far we have only looked at the differences in SQNRs between the different formats. As mentioned earlier, we can also compare SQNR before and after the operations to get an idea of what the impact of different operations is on the precision of a computation. Table A.3 shows the additional noise added to a computation for different operations and different numbers of operations.

	1 Multiplication	2 Multiplications	8 Multiplications
Single Precision	4.7586	5.4526	7.9877
Fixed-Point 32-bit	5.4367	32.5818	63.0748
Fixed-Point 32-bit	6.5756	33.3755	63.5824
	1 Addition	**2 Additions**	**8 Additions**
Single Precision	0.7386	3.4816	7.9520
Fixed-Point 32-bit	0.7380	4.1814	8.6797
Fixed-Point 32-bit	0.7378	4.1226	9.0059

Table A.3: Added noise (in dB) per operation

To summarize our findings so far, we conclude the following:

- A 32 bits significand does indeed provide advantages over standard single precision, approximately 33% more SQNR.
- The additional SQNR of extra fractional bits is linearly related to the number of bits.
- Floating-point offers significantly more precision over fixed-point during multiplication, even if the number of fractional bits used is smaller.
- In case of addition, fixed-point can be just as precise as floating-point.
- If applications are multiplication intensive, floating-point is favored over fixed-point.
- When operations accumulate, floating-point is favored over fixed-point.

Hence, extending the significand to match the 32-bit input is worthwhile, and using floating-point for DSP rather than fixed-point can certainly be an improvement.

A.3 Practical Applications

We have shown what the advantages are of additional significand bits. But what does all this mean for practical applications such as a digital filter? Filters can be used to remove noise from a signal. In the case of a FIR filter [49] this is accomplished by a series of multiplications and additions. The generalized structure of a n-th order FIR filter is shown in Figure A.6. Every output element is a weighted average of input samples. Averaging is accomplished by multiplication with constants $(b_0, b_1 \cdots b_n)$. For every sample output of a n-th order FIR filter, n+1 multiplications and n additions are involved. However, the multiplications are not accumulated in contrast to our earlier tests.

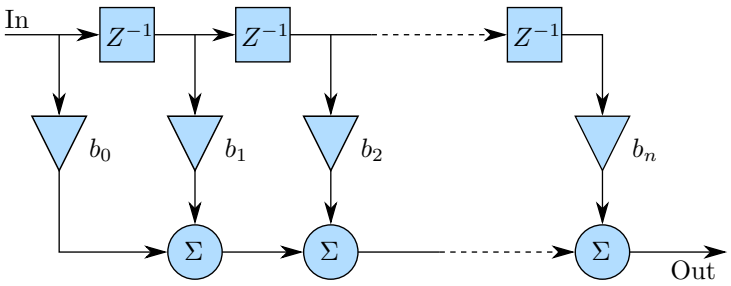
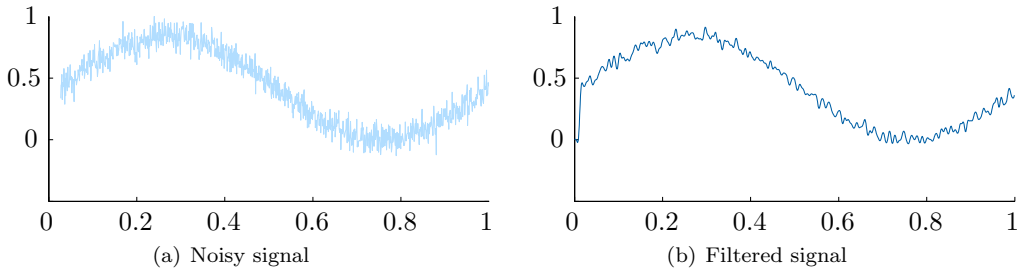


Figure A.6: Regular structure n-th order FIR filter

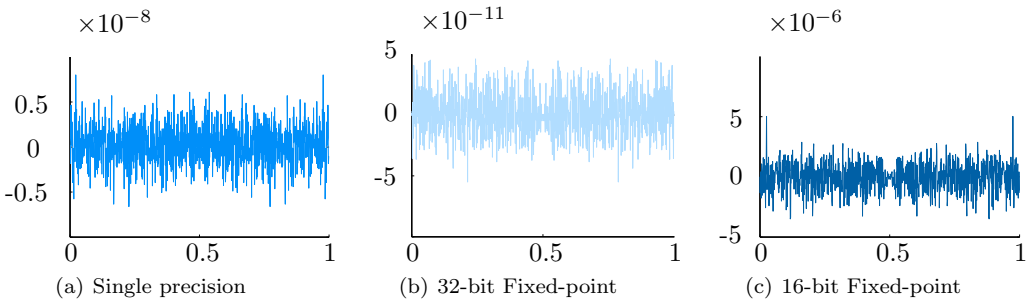
To go from a noisy signal like Figure A.7(a) to a cleaner signal as shown in Figure A.7(b), we have used a 24^{th} order *low pass* FIR filter.



(a) Noisy signal (b) Filtered signal

Figure A.7: Noisy signal before and after filtering

This filter can of course be implemented with different precision. We use the double precision implementation as the ideal again. The absolute difference between the different formats and double precision (the quantization error) has been plotted in Figure A.8.



(a) Single precision (b) 32-bit Fixed-point (c) 16-bit Fixed-point

Figure A.8: Absolute quantization error in 24^{th} order FIR filter

Precision	SQNR 24^{th} order filter	SQNR 127^{th} order filter
Single Precision	143.7527	138.3766
Fixed-Point 32-bit	90.0202	84.3353
Fixed-Point 16-bit	186.4696	180.6767

Table A.4: Post-filter SQNRs

What we observe is that the absolute errors differ several orders of magnitude. A second observation is that the error is almost entirely determined by the number of fractional bits. The floating-point format hardly provides an advantage over fixed-point. This can be explained by the fact that a FIR filter is mainly based on additions. Of course, there are n multiplications involved, but these do not accumulate. This confirms our earlier finding that, when computations are based on additions and not so much on multiplication, floating-point does not provide much advantage over a fixed-point format. The number of fractional bits is a more important parameter then. In this small example, the errors can be neglected compared to the noise that is still present in the signal. However, in certain applications FIR filters are used with orders in the range of several thousands. A floating-point format with an extended significand may then show its superiority. The SQNRs after 24^{th} and 128^{th} order FIR filters are listed in Table A.4 for indication.

Common Mistakes in Floating-Point Arithmetic

Designing floating-point hardware is notorious for being difficult and error-prone. The many subtleties related to overflow/underflow, other exceptions and rounding, make that errors are sometimes difficult to reveal. On several occasions floating-point units have been commercialized only to find out after production that the hardware sometimes still produced incorrect answers. Think not only of the infamous Intel FDIV bug, also Cray and IBM have shipped faulty floating-point units. In this section we present a small selection of concepts that are easily misinterpreted and often go wrong, based on experience from designing the FMA unit presented in this thesis.

B.1 IEEE-754 Floating-Point Arithmetic and Zero

Whenever arithmetic is performed on one or more operands that are zero (*zero-arithmetic*), problems quickly start showing up. In the IEEE-754 floating-point format, zero considerably differs from other numbers due to the fact that there are two distinct representations for zero ($+0$ and -0), and the fact that it does not use a biased exponent like all other valid numbers. Both properties entail different kinds problems which are discussed next.

B.1.1 Unbiased Exponents

Any combination of operands that does not include zero, will accumulate the bias while adding exponents for multiplication. A correction has to be performed which subtracts the bias from the resulting exponent. However, when either one or both operands are zero, such a correction is not needed. If performed anyway, the exponent is likely to underflow, and definitely does not result in correct output. This is just one of many examples where the actions taken for non-zero-arithmetic do no apply to zero-arithmetic. Also complementing for subtraction, normalization and rounding have to be bypassed. Due to these irregularities, it is very difficult to design zero-arithmetic as an integral part of a floating-point datapath. It is often more convenient to treat zero as an exception.

A robust solution, also implemented in the architecture described in chapters 4, 5 and 6, is to detect if the operands are zero in the initial stage. Based on these finding one can control the data flow for zero-arithmetic, bypassing certain components and performing corrections if needed. If one of the two product operands (A,B) is zero, C is unaltered and bypasses the alignment shifter. To keep the datapath as regular as possible and to match the delay of operations that do not involve zeros, C is still routed through the adder which computes 0+C. Normalization and rounding are disabled such that the output

is exactly the same as the input (C). If operand C itself is zero, the inter-stage register that stores the shifted addend is overwritten with zeros. The adder than computes A×B+0.

B.1.2 Sign Bits

For arithmetic as well as logic operations, the sign-bit in IEEE-754 (multiply-add) is problematic. Especially in the case of zero, because of the ambiguity related to the presence of both +0 and -0. When the result of an arithmetic operation is exactly zero, it is not immediately clear whether it should be +0 or -0. For this reason IEEE-754 exactly defines how the sign-bit for zero should be dealt with. These definitions are mathematically sound, however they are not very convenient from a hardware engineer's point of view (Section 6.8.3). When the result of an arithmetic operation is exactly zero, IEEE-754 stipulates that the sign should be positive unless 'round to $-\infty$' is used, when the result becomes zero due to rounding or when both signs are the same (i.e., $0+0$ and $0-(-0)$). Under the 'round to $-\infty$' attribute the sign shall be negative. The sign of results becoming zero after rounding must maintain the sign of the unrounded result. When both operands have the same sign, the resulting sign-bit retains the same value as the input operands.

These rules are exceptions to an exception which makes sign handling confusing, especially for FMA which has only been standardized since the 2008 update of IEEE-754. A few examples (corresponding to the exceptions just discussed) of cases that are most the likely to go wrong have been listed below:

$$\begin{aligned} -5 \times 2 + 10 &= +0 \neq -0 \\ 0 \times 0 + (-0) &= +0 \neq -0 \\ 0 \times (-0) + (-0) &= -0 \neq +0 \\ roundTowardNegative(-5 \times 2 + 10) &= -0 \neq +0 \end{aligned}$$

To compute every sign-bit according to the standard requires a considerable amount of control. For the ALU presented in this thesis, a choice was made to adjust the sign-bit computed by Equation 5.7 for every result that is zero in the final stage.

To summarize the above, when one is interested in implementing IEEE-754 floating-point arithmetic, special care should be taken when dealing with zero-arithmetic. Especially the sign-bit requires attention in the following cases:

- If the result is exactly zero, the sign should by default be positive
- If the result is zero and both *addition* operands where zero and had the same sign, the resulting sign should retain the value of the input operands
- If the result is zero and the round using the round to negative infinity attribute, the sign-bit should be negative

In case of logic operations, the sign for zero operand must be ignored completely. This means that $+0 = -0$, no exceptions. Although this rule is very simple, it may not directly be intuitive.

Note: The floating-point reference implementation by Bishop [36] suffers from the sign-bit problems illustrated above.

B.2 Rounding and Sticky-Bit

The majority of IEEE-754 compatible rounding procedures is based on the guard, round, sticky-bit principle (Section 2.6.5). Although obtaining these bits is usually not a problem (Section 5.2.1), their

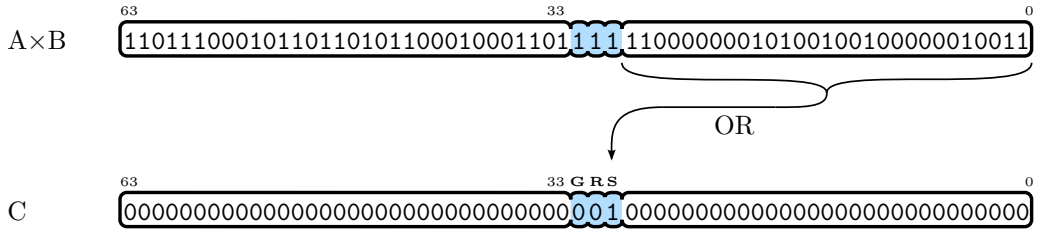


Figure B.1: Wrong sticky-bit usage

usage can be confusing. In particular the use of the sticky-bit in FMA is not always clear. A common misconception is to use the sticky-bit as an additional guard-bit. The sticky-bit can not simply be concatenated to the C operand as we do with the guard and round-bits. With a small example we will illustrate why this may cause problems.

Suppose two arbitrary 32-bit floating-point numbers have been multiplied (A×B) as shown in Figure B.1. Of the 64-bit product, only 32 bits can be stored in memory. Chapter 5 explained that all bits 'behind' the sticky-bit position have to be discarded and are therefore logically OR'ed into the sticky-bit to indicate inexactness [1]. Now suppose we want to add zero using 'round to zero' mode. We would expect to find the 32 MSBs of the product.

$$\begin{array}{l} \texttt{11011100010110110101100010001101} \\ \texttt{00000000000000000000000000000000} \\ \hline \texttt{11011100010110110101100010001101} + \end{array}$$

Instead, we find the incremented 32 MSB of the product. By inserting the sticky-bit into the addend before addition, a carry-in alters the product:

$$\begin{array}{l} \texttt{11011100010110110101100010001101}\ \mathbf{111} \\ \texttt{00000000000000000000000000000000}\ \mathbf{001} \\ \hline \texttt{11011100010110110101100010001}\mathbf{10} + \end{array}$$

A similar situations can occur when a very small operand is added to a large product. The sticky-bit calculated during alignment will cause a carry-in and produces an incorrect result. Hence, the sticky-bit can not simply be used as a guard-bit.

The question that rises now is how the sticky-bit should be used then. The purpose of the sticky-bit is essentially to indicate that the result of a certain operation is no longer exact. Whenever the sticky-bit is `1`, we know that meaningful bits (tailing `0`'s do not contribute to the precision of a floating-point number) have been shifted out of range, and that the result in infinite precision would be different from the actual result. The sticky-bit should only be used as an indicator for these situation and not to perform actual arithmetic with. In the example below we perform the addition with the guard and round bits only.

$$\begin{array}{l} \texttt{11011100010110110101100010001101 11} \\ \texttt{00000000000000000000000000000000 00} \\ \hline \texttt{11011100010110110101100010001101 11} + \end{array}$$

[1] Sticky-bit calculations in FMA are two-fold: during alignment a sticky-bit is calculated and another one during normalization/rounding. They are combined into a single sticky-bit for rounding.

After completing addition, the sticky-bit is concatenated to the final product used for rounding.

```
11011100010110110101100010001101 111
```

The three LSB (`111`) should then be used as a pattern to determines whether we need to round up or down according one of the IEEE-754 rounding modes (Algorithm 5.6.1 or Algorithm 5.6.2).

Note: The floating-point reference implementation by Bishop [36] suffers from the sticky-bit problems illustrated above.

B.3 Fused Multiply-Add and Overflow

Overflow and underflow control for FMA is unconventional, which may also lead to problems. Whenever multiplication or addition/subtraction are performed separately and overflow or underflow is detected, the result simply overflows/underflows[2]. In FMA the situation is different because a second operation is performed on the intermediate result. If the result of A×B overflows, the subtraction of C could result in a number that is perfectly representable in the supported range. Similarly, underflow of A×B can be undone by adding C that is not zero. The addition of C could however also result in overflow/underflow. In addition normalization and rounding can affect the underflow/overflow status of operations.

From the above it should be clear that detection of overflow/underflow needs to be performed multiple times for multiply-add instructions. The most important message we would like to express here, is that the final checks should performed after rounding. However, due to bypasses for zero-arithmetic and data flow control, one should also keep track of the status of intermediate results.

One additional problem related to the specific implementation of underflow detection presented in Section 6.7.1, is that 0×0+0 is detected as underflow. In this implementation, the valid bit of the leading zero detector partially indicates if a result underflows. Of course LZD performed on 0×0+0 will assert the valid bit to `0`, triggering the underflow exception. A simple solution is to overwrite the status when all operands are zero. These control signal are available from the operand formatter in the first pipeline stage, hence a simple correction can be performed in the final stage when all operands appear to be zero.

[2] In certain cases the shift for normalization may be enough to recover from overflow

C

Instructionset Specification

For each instruction implemented in hardware, a specification is provided, consisting of:

- Required input
- Returned output
- Operation(s) performed (expected behavior)
- Round direction (only for floating-point arithmetic)
- Encoding of status bits (exception flags or compare results)
- Associated opcode
- Time to completion

The specifications are based on the following syntax:

$A	Operand A in register format (formatting depends on instruction (Section 4.3.1))
$A.lsbs	Only the least significand bits of operand A
$A.msbs	Only the most significand bits of operand A
$Res	Result in register format (formatting depends on instruction (Section 4.3.1))
HI	32 high order bits (MSBs)
LO	32 low order bits (LSBs)
Status	3-bit status pattern encoding (exception flags or compare instruction output)

Floating-point Arithmetic

`FMAN` - **Floating-point Multiply-Add**	
Description	Multiplies two floating-point operands and adds (or subtracts, depending on the sign) a third operand in a single instruction. Advantages include better precision and faster execution of multiply-add operations.
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, $B and $C
Output	One floating-point result (sign-bit, 8-bit exponent and 32-bit significand): $Res
Operation	$Res = $A × $B + $C
Round direction(s)	Nearest, ties to even
Status	`100` - default `001` - overflow `010` - underflow
Opcode	`00001`
Latency	3 clock cycles

FMAZ - Floating-Point Multiply-Add	
Round direction(s)	Zero
Opcode	00010
Other fields similar to FMAN	

FMAP - Floating-Point Multiply-Add	
Round direction(s)	Positive infinity
Opcode	00011
Other fields similar to FMAN	

FMAM - Floating-Point Multiply-Add	
Round direction(s)	Negative infinity
Opcode	00100
Other fields similar to FMAN	

Floating-point multiplication, addition and subtraction do not have a unique opcode. They are however directly supported by the hardware. Literals 1 and 0 are to be used for the multiplicand and the addend respectively in order to perform addition/subtraction and multiplication. For different rounding modes, the appropriate multiply-add instruction can be selected.

FMUL - Floating-Point Multiplication	
Description	Multiplies two floating-point operands. Implemented as FMA derivative with the addend hardwired to 0
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, $B and 0
Output	One floating-point result (32-bit significand, 8-bit exponent and sign-bit: $Res
Operation	$Res = $A × $B + 0 (= $A × $B)
Round direction(s)	Nearest/Zero/Positive or Negative infinity
Status	100 - default 001 - overflow 010 - underflow
Opcode	N/A (use FMA)
Latency	3 clock cycles

FADD - Floating-Point Addition/Subtraction	
Description	Adds (or subtracts, depending on the signs) two floating-point operands. Implemented as FMA derivative with the multiplicand hardwired to 1
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, 1 and $C
Output	One floating-point result (sign-bit, 8-bit exponent and 32-bit significand): $Res
Operation	$Res = $A × 1 + $C (= $A + $C)
Round direction(s)	Nearest/Zero/Positive or Negative infinity
Status	100 - default 001 - overflow 010 - underflow
Opcode	N/A (use FMA)
Latency	3 clock cycles

Floating-Point Compare

FLTV - Floating-Point Compare 'Less Than Value'	
Description	Compares two floating-point operands and determines if one operand is smaller than the other.
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, $B and 0
Output	N/A (only status bits valid)
Operation	Status = $A < $C
Status	111 - true 000 - false
Opcode	00101
Latency	3 clock cycles

FGTV - Floating-Point Compare 'Greater Than Value'	
Description	Compares two floating-point operands and determines if one operand is smaller than the other.
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, $B and 0
Output	N/A (only status bits valid)
Operation	Status = $A > $C
Status	111 - true 000 - false
Opcode	00110
Latency	3 clock cycles

FLTV - Floating-Point Compare 'Equal to Value'	
Description	Compares two floating-point operands and determines if they are equal.
Input	Three floating-point operands (sign-bit, 8-bit exponent and 32-bit significand): $A, $B and 0
Output	N/A (only status bits valid)
Operation	Status = $A = $C
Status	111 - true 000 - false
Opcode	00111
Latency	3 clock cycles

Integer Arithmetic

`IMAC` - **Integer Multiply-Accumulate**	
Description	Multiplies two integer operands and adds a third in a single instruction.
Input	Three integer operands (3 × 32-bit signed integers): $A, $B and $C.
Output	One 64-bit integer result divided over two register format words: HI (upper 32 bits) and LO (lower 32 bits).
Operation	{HI,LO} = $A × $B + $C
Status	`100` - default, result requires 32 bits or less for representation (only LO) `101` - result requires more than 32 bits for representation (HI and LO)
Opcode	`10000`
Latency	2 clock cycles

Integer multiplication, addition and subtraction do not have unique opcodes. They to perform such operations, the `IMAC` instruction should be invoked using literals 1 and 0 for the multiplicand and the addend respectively.

`IMUL` - **Integer Multiplication**	
Description	Multiplies two integer operands.
Input	Three integer operands (3 × 32-bit signed integers): $A, $B and 0.
Output	One 64-bit integer result divided over two register format words: HI (upper 32 bits) and LO (lower 32 bits).
Operation	{HI,LO} = $A × $B + 0 (= $A × $B)
Status	`100` - default,, result requires 32 bits or less for representation (only LO) `101` - result requires more than 32 bits for representation (HI and LO)
Opcode	N/A (use `IMAC`)
Latency	2 clock cycles

`IADD` - **Integer Addition/Subtraction**	
Description	Adds or subtracts (depending on the signs of the operands) two integer operands.
Input	Three integer operands (3 × 32-bit signed integers): $A, 0 and $C.
Output	One 64-bit integer result divided over two register format words: HI (upper 32 bits) and LO (lower 32 bits).
Operation	{HI,LO} = $A × 0 + $C (= $A + $C)
Status	`100` - default, result requires 32 bits or less for representation (only LO) `101` - result requires more than 32 bits for representation (HI and LO)
Opcode	N/A (use `IMAC`)
Latency	2 clock cycles

Integer Shift

ISLV - Integer Shift Left Value	
Description	Left shifts integer operand by specified amount.
Input	Two integer operands, one 32-bit signed and one 7-bit unsigned number: $A.msbs, $B.lsbs
Output	One 32-bit signed integer operand
Operation	N/A (only status bits valid)
Status	100 - default
Opcode	10001
Latency	2 clock cycles

ISRV - Integer Shift Right Value	
Description	Left shifts integer operand by specified amount.
Input	Two integer operands, one 32-bit signed and one 7-bit unsigned number: $A.msbs, $B.lsbs (all other input must be zero)
Output	One 32-bit signed integer operand
Operation	LO = $A.msbs >> $C.lsb
Status	100 - default
Opcode	10011
Latency	2 clock cycles

Integer Compare

ILTV - Integer Less Than Value	
Description	Compares two integer operands and determines if one operand is smaller than the other.
Input	Two integer operands (2 × 32-bit signed integers): $A, $B
Output	N/A (only status bits valid)
Operation	Status = $A < $C
Status	111 - true 000 - false
Opcode	10100
Latency	2 clock cycles

IGTV - Integer Greater Than Value	
Description	Compares two integer operands and determines if one operand is bigger than the other.
Input	Two integer operands (2 × 32-bit signed integers): $A, $B
Output	N/A (only status bits valid)
Operation	Status = $A > $C
Status	111 - true
	000 - false
Opcode	10101
Latency	2 clock cycles

IETV - Integer Equal To Value	
Description	Compares two integer operands and determines if they are equal.
Input	Two integer operands (2 × 32-bit signed integers): $A, $B
Output	N/A (only status bits valid)
Operation	Status = $A = $C
Status	111 - true 000 - false
Opcode	10110
Latency	2 clock cycles

D

Dataflow and Datapath Usage

The following illustrations provide a visualization of the data flow per instruction. Figure 6.1, showing a complete overview of the datapath, has been redrawn here (Figure D.1) for convenience. All other figures show the same datapath with the parts that are being used for a specific instruction highlighted. Every component drawn with thickened lines is required for the respective operation.

Although these images can also be used to quickly get a rough idea of what hardware is needed for a certain instruction and the (non-control type) overhead resulting from integer integration, they should not be used as a measure for physical properties such as area usage. See Chapter 7 for details regarding physical properties such as area and energy consumption. The conceptual hardware blocks shown here are not drawn proportionally.

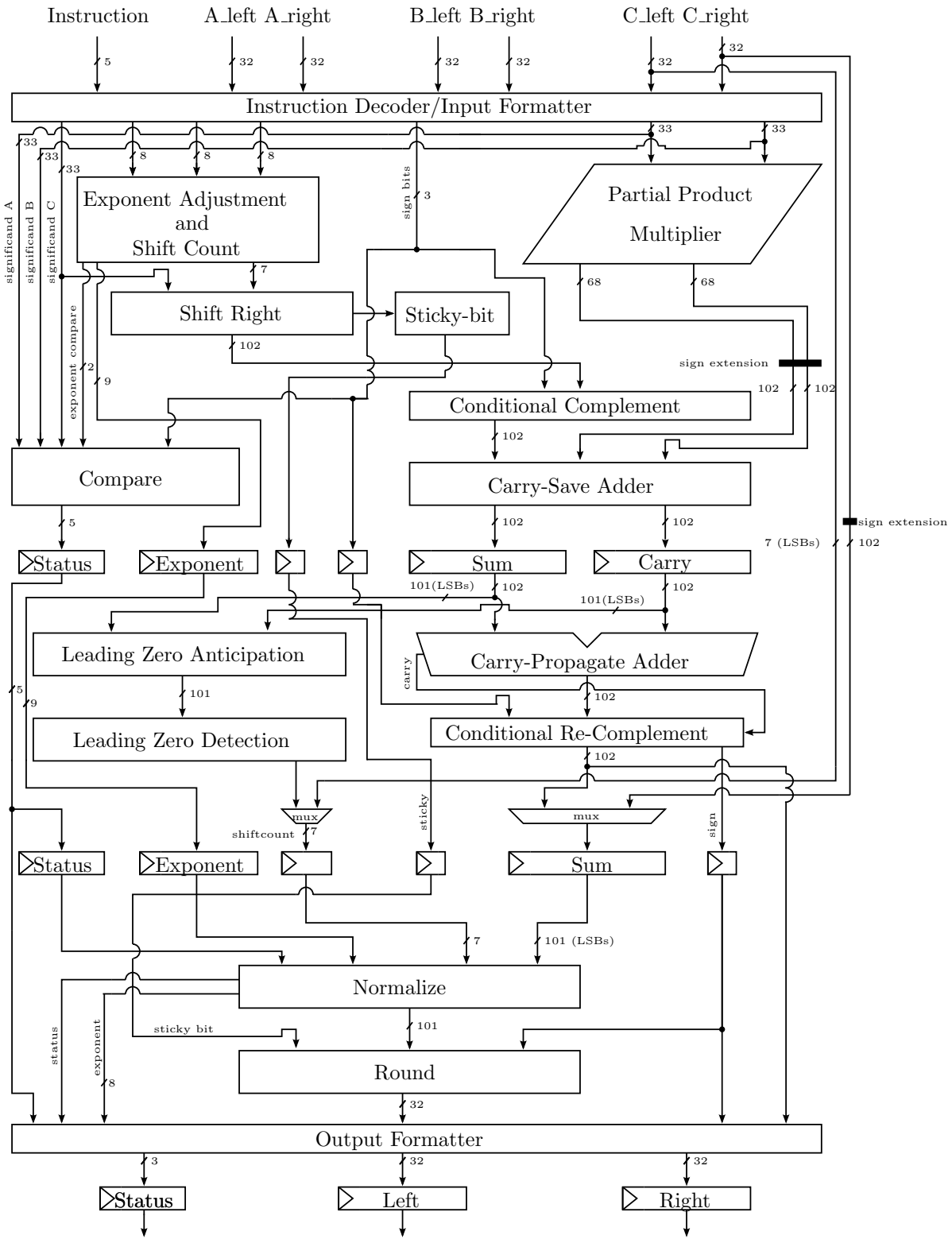


Figure D.1: ALU datapath

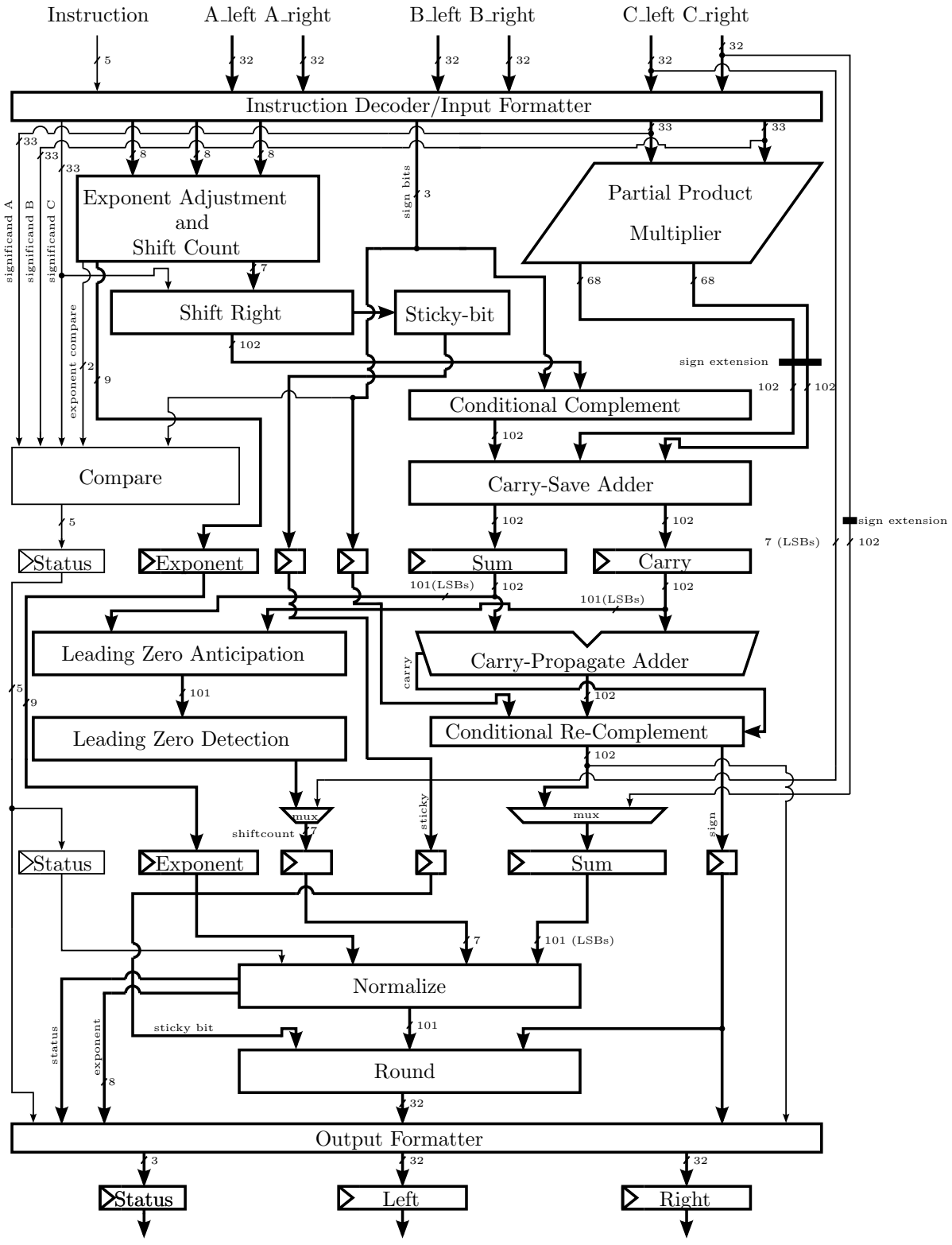


Figure D.2: Floating-point arithmetic dataflow

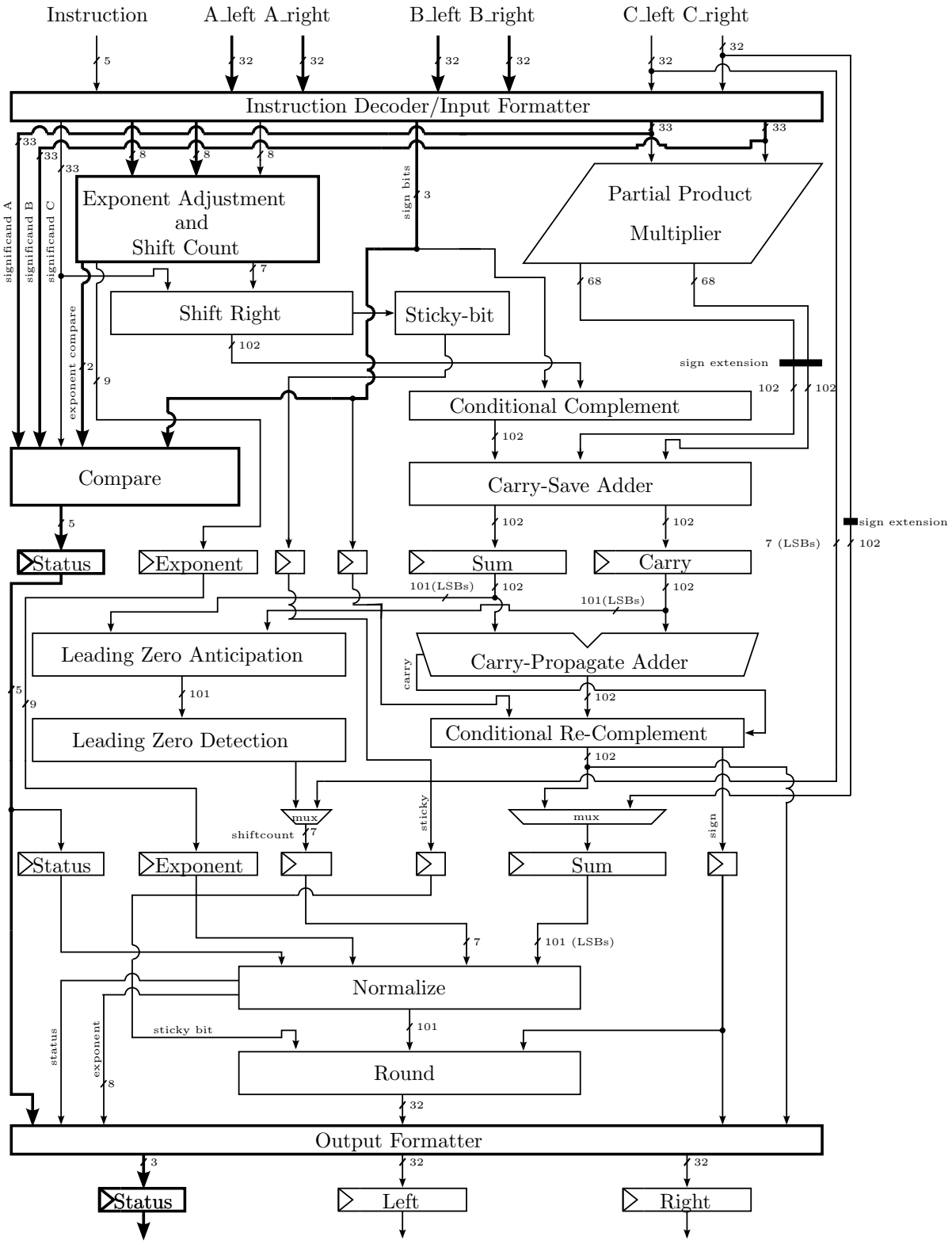


Figure D.3: Floating-point compare dataflow

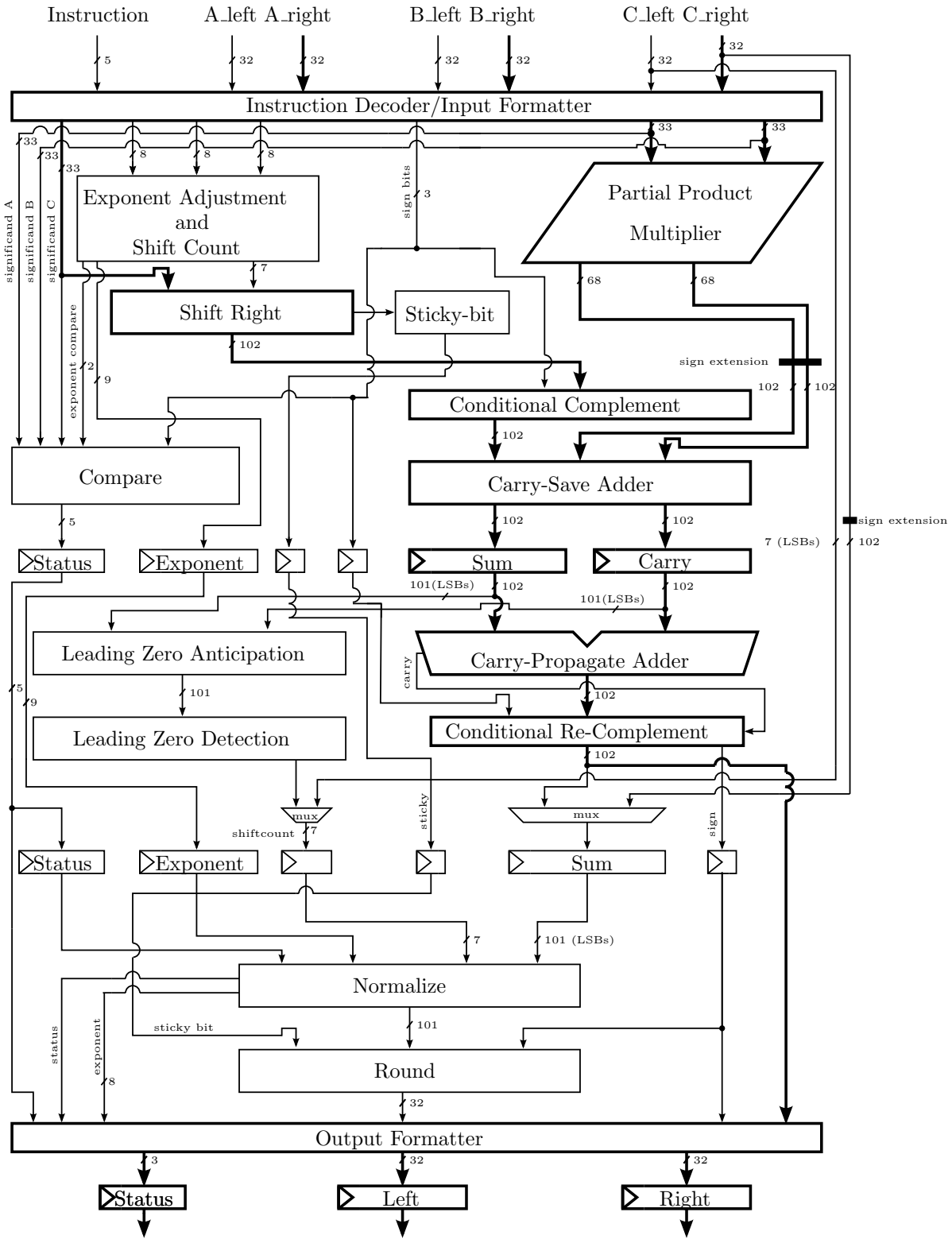


Figure D.4: Integer arithmetic dataflow

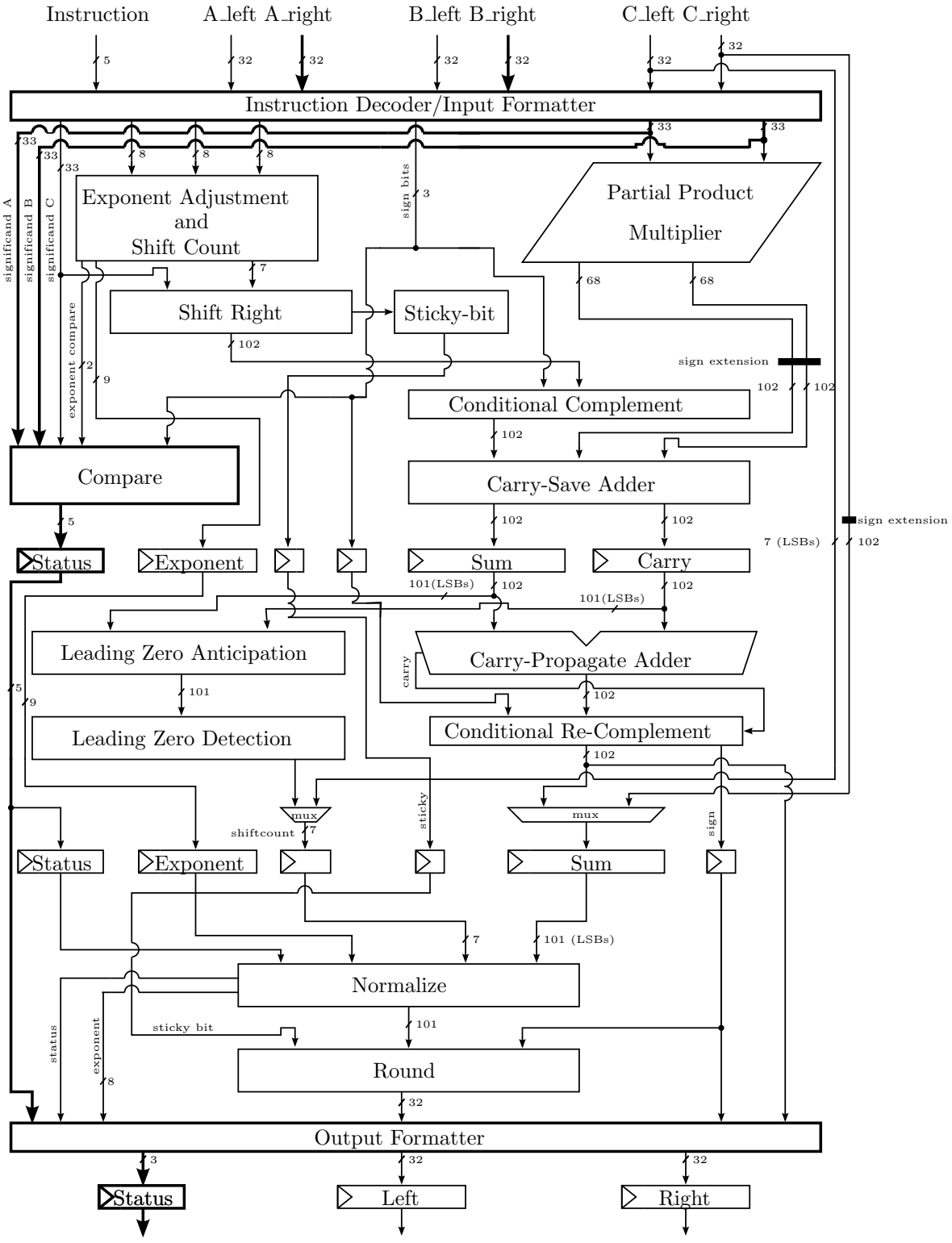


Figure D.5: Integer compare dataflow

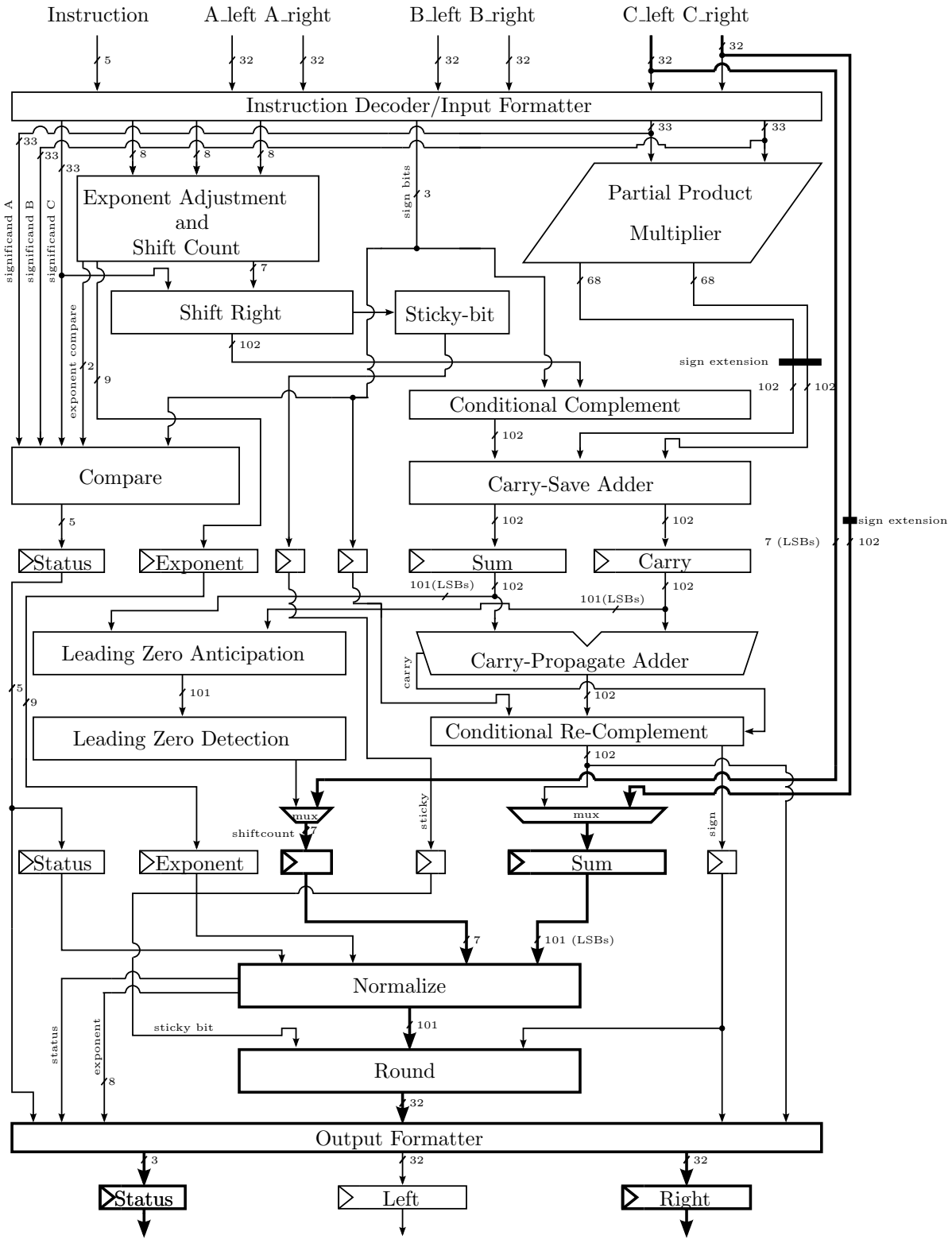


Figure D.6: Shift left dataflow

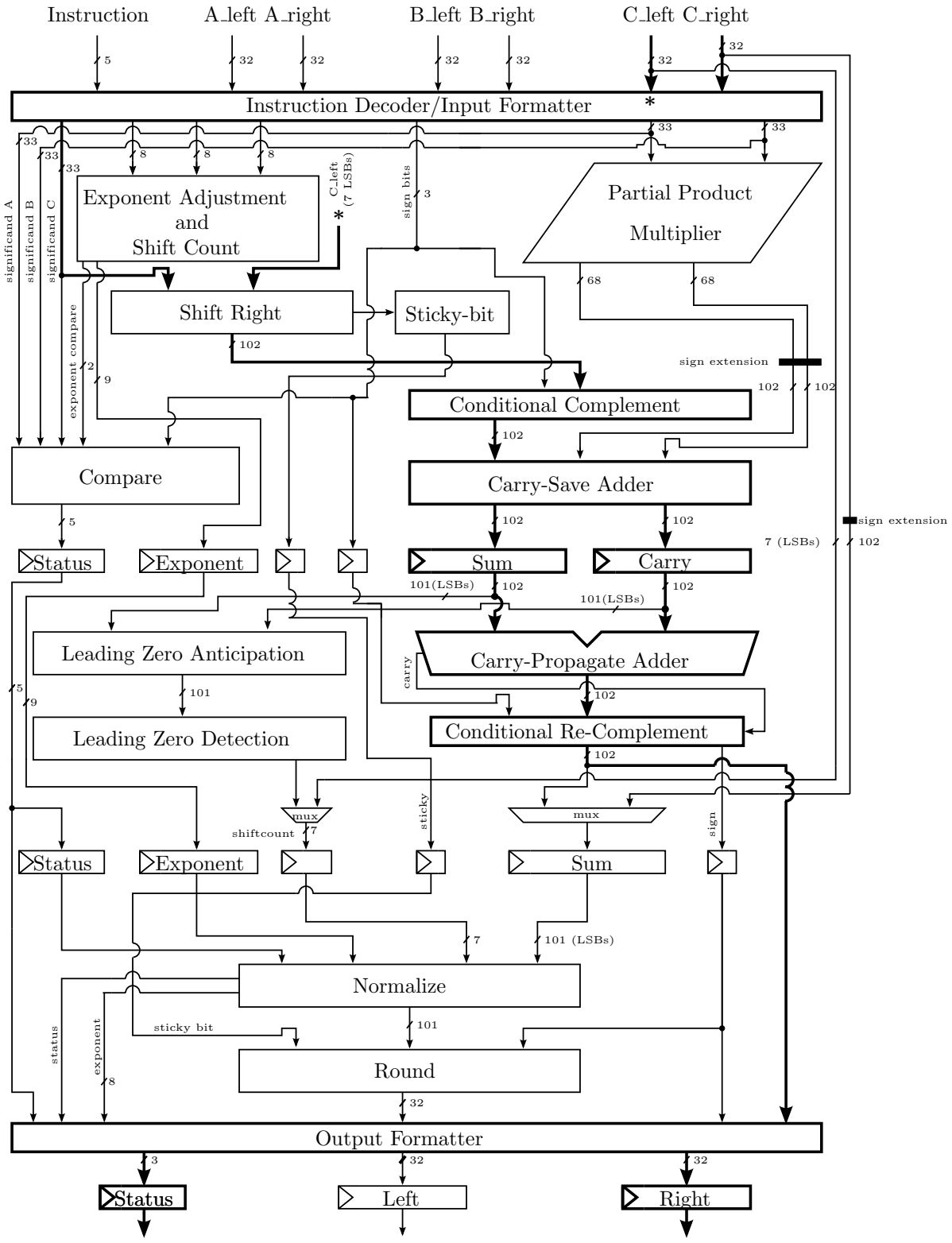


Figure D.7: Shift right dataflow

Bibliography

[1] V. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 124 –128, Mar. 1994.

[2] K.H.G. Walters, S.H. Gerez, G.J.M. Smit, S. Baillou, G.K. Rauwerda, and R. Trautner, "Multicore soc for on-board payload signal processing." **accepted for** *Adaptive Hardware and Systems 2011*, June 2011.

[3] D. Seal, *ARM Architecture Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2000.

[4] W. Stallings, *Computer Organization & Architecture*. Upper Saddle River, NJ, USA: Prentice Hall Press, eighth ed., 2009.

[5] G. A. Blaauw and J. Brooks F. P., *Computer Architecure, Concepts and Evolution*. Addison-Wesley, first ed., feb. 1997.

[6] "IEEE Task P754", *IEEE 754-2008, Standard for Floating-Point Arithmetic*. pub-IEEE-STD, aug. 2008.

[7] M. M. Mano and C. Kime, *Logic and Computer Design Fundamentals*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2004.

[8] B. Parhami, *Computer arithmetic: algorithms and hardware designs*. Oxford, UK: Oxford University Press, 2000.

[9] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5–48, March 1991.

[10] R. Rojas, "Konrad zuse's legacy: the architecture of the z1 and z3," *Annals of the History of Computing, IEEE*, vol. 19, pp. 5 –16, april 1997.

[11] S. Microsystems", "Opensparc t2 core microarchitecture specification," tech. rep., Sun Microsystems Inc., December 2007.

[12] A.D.Booth, "A Signed Multiplication Technique (Part 2)," *J. Mech. & Appl. Math*, vol. 4, pp. 236–240, 1951.

[13] M. Cornea, J. Harrison, and P. T. P. Tang, "Intel itanium floating-point architecture," *Workshop on Computer Architecture Education*, 2003.

[14] M. Cornea-Hasegan and B. Norin, "Ia-64 floating-point operations and the ieee standard for binary floating-point arithmetic," *Intel Technology journal Q4*, 1999.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589 –604, july 2005.

[16] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *Micro, IEEE*, vol. 26, pp. 10 –24, march-april 2006.

[17] S. Mueller, C. Jacobi, H.-J. Oh, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong, "The vector floating-point unit in a synergistic processor element of a cell processor," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 59 – 67, 27-29 2005.

[18] P. M. Farmwald, *On the design of high performance digital arithmetic units.* PhD thesis, Stanford University, Stanford, CA, USA, 1981.

[19] J. Bruguera and T. Lang, "Floating-point fused multiply-add: reduced latency for floating-point addition," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 42 – 51, june 2005.

[20] S. A. Jain, "Low-power single-precision ieee floating-point unit," Master's thesis, Massachusetts Institute of Technology, 2003.

[21] A. Amaricai, M. Vladutiu, L. Prodan, M. Udrescu, and O. Boncalo, "Exploiting parallelism in double path adders' structure for increased throughput of floating point addition," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pp. 132 –137, 29-31 2007.

[22] S. Palacharla and J. E. Smith, "Decoupling integer execution in superscalar processors," in *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 285–290, IEEE Computer Society Press, 1995.

[23] Y. Solihin, K. W. Cameron, Y. Luo, D. Lavenier, and M. Gokhale, "Dynamically mutable functional unit in superscalar processors," tech. rep., University of Illinois, 2007.

[24] D. Lavenier, Y. Solihin, and K. Cameron, "Integer/floating-point reconfigurable alu," *Proceedings of the 6th Symposium on New Machine Architectures*, 1999.

[25] A. Oppenheim, "Realization of digital filters using block-floating-point arithmetic," *Audio and Electroacoustics, IEEE Transactions on*, vol. 18, pp. 130 – 136, June 1970.

[26] A. Robison, "N-bit unsigned division via n-bit multiply-add," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 131 – 139, june 2005.

[27] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the ibm risc system/6000 floating-point execution unit," *IBM J. Res. Dev.*, vol. 34, pp. 59–70, January 1990.

[28] W. Hayes, R. Kershaw, L. Bays, J. Boddie, E. Fields, R. Freyman, C. Garen, J. Hartung, J. Klinikowski, C. Miller, K. Mondal, H. Moscovitz, Y. Rotblum, W. Stocker, J. Tow, and L. Tran, "A 32-bit vlsi digital signal processor," *Solid-State Circuits, IEEE Journal of*, vol. 20, pp. 998 – 1004, Oct. 1985.

[29] M. Schmookler and K. Nowka, "Leading zero anticipation and detection-a comparison of methods," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 7 –12, 2001.

[30] E. M. Schwarz, "Binary floating-point unit design," in *High-Performance Energy-Efficient Microprocessor Design* (A. Chandrakasan, V. G. Oklobdzija, and R. K. Krishnamurthy, eds.), Series on Integrated Circuits and Systems, pp. 189–208, Springer US, 2006.

[31] P. Woo-Chan, L. Shi-Wha, K. Oh-Young, H. Tack-Don, and K. Shin-Dug, "Floating point adder/-subtractor performing ieee rounding and addition/subtraction in parallel," *IEICE transactions on information and systems*, vol. 79, no. 4, pp. 297–305, 1996.

[32] P. M. Heysters, *Coarse-Grained Reconfigurable Processors - Flexibility meets Efficiency.* PhD thesis, Univ. of Twente, Enschede, September 2004.

[33] R. Jessani and M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units," *Computers, IEEE Transactions on*, vol. 47, pp. 927 –937, Sept. 1998.

[34] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 23–36, 1981.

[35] S. Vassiliadis, D. Lemon, and M. Putrino, "S/370 sign-magnitude floating-point adder," *Solid-State Circuits, IEEE Journal of*, vol. 24, pp. 1062 –1070, Aug. 1989.

[36] D. W. Bishop, "`http://www.vhdl.org/fphdl`," May 2011.

[37] S. C. Knowles, "Arithmetic processor design for the t9000 transputer," in *Advanced Signal Processing Algorithms, Architectures, and Implementations II* (F. T. Luk, ed.), vol. 1566, pp. 230–243, SPIE, 1991.

[38] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, "Low-power leading-zero counting and anticipation logic for high-speed floating point units," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, pp. 837 –850, july 2008.

[39] G. Zhang, Z. Qi, and W. Hu, "A novel design of leading zero anticipation circuit with parallel error detection," in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pp. 676 – 679 Vol. 1, May 2005.

[40] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for ieee floating-point multiplication," *Computers, IEEE Transactions on*, vol. 49, pp. 638 –650, jul 2000.

[41] Synopsys, "`http://www.synopsys.com`," May 2011.

[42] Cadence, "`http://www.cadence.com`," May 2011.

[43] J. Hauser, "`http://www.jhauser.us/arithmetic/TestFloat.html`," May 2011.

[44] B. Verdonk, A. Cuyt, and D. Verschaeren, "`http://cant.ua.ac.be/old/ieeecc754.html`," May 2011.

[45] B. Verdonk, A. Cuyt, and D. Verschaeren, "A precision- and range-independent tool for testing floating-point arithmetic i: basic operations, square root and remainder," *ACM TOMS*, vol. 27, no. 1, pp. 92–118, 2001.

[46] C. Shuang-yan, W. Dong-hui, Z. Tie-jun, and H. Chao-huan, "Design and implementation of a 64/32-bit floating-point division, reciprocal, square root, and inverse square root unit," in *Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on*, pp. 1976 –1979, 2006.

[47] P. Markstein, "Software division and square root using goldschmidt's algorithms," in *In 6th Conference on Real Numbers and Computers*, pp. 146–157, 2004.

[48] Mathworks, "`http://www.mathworks.com`," May 2011.

[49] J. G. Proakis and D. K. Manolakis, *Digital Signal Processing (4th Edition)*. Prentice Hall, 4 ed., 2006.

[50] R. Boite, H. Xian-Liang, and J. Renard, "A comparison of fixed-point and floating-point realization of digital filter," in *Electrotechnics, 1988. Conference Proceedings on Area Communication, EUROCON 88., 8th European Conference on*, pp. 142–145, jun 1988.