DESIGN AND IMPLEMENTATION OF A FLOATING POINT UNIT FOR RIGEL, A MASSIVELY PARALLEL ACCELERATOR

BY

WOJCIECH JAN TRUTY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Associate Professor Sanjay Patel

ABSTRACT

Scientific applications rely heavily on floating point data types. Floating point operations are complex and require complicated hardware that is both area and power intensive. The emergence of massively parallel architectures like Rigel creates new challenges and poses new questions with respect to floating point support. The massively parallel aspect of Rigel places great emphasis on area efficient, low power designs. At the same time, Rigel is a general purpose accelerator and must provide high performance for a wide class of applications. This thesis presents an analysis of various floating point unit (FPU) components with respect to Rigel, and attempts to present a candidate design of an FPU that balances performance, area, and power and is suitable for massively parallel architectures like Rigel.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ASIC	Application specific integrated circuit
CPU	Central processing unit
FMACC	Fused multiply and accumulate
FMADD	Fused multiply and add
FPU	Floating point unit
GPU	Graphics processing unit
IEEE	The Institute of Electrical and Electronics Engineers, Inc.
LSB	Least significant bit
MIMD	Multiple instruction, multiple data
MSB	Most significant bit
RNE	Round to Nearest, Ties to Even
SPMD	Single program, multiple data

CHAPTER 1

INTRODUCTION

Ever since Jack Kilby built the first integrated circuit, scientists and engineers have been pushing hard to fit as many components as possible on silicon wafers, and to operate them as fast as possible. Until recently, obtaining higher performance from digital logic was synonymous with increasing the operating frequency. However, overall performance of any hardware element is not solely a function of the operating frequency. The architecture of the hardware plays a crucial role in how "fast" it will operate. The architecture becomes even more critical as frequency scaling slows down due to increasing power and heat concerns.

In a processor, performance is affected by the number of cores, amount of cache memory, instruction issue bandwidth, and the operations that the processor supports. Some processors are designed for general purpose, every day tasks and are architected to minimize latency ($\frac{seconds}{instruction}$). Others are designed to perform exceptionally well at specific tasks and maximize throughput ($\frac{instructions}{second}$). The latter are known as accelerators and are used in applications such graphics, audio, video, and signal processing. Accelerators are architected to take advantage of the characteristics of their target domain.

A new class of accelerators has emerged. These *compute accelerators* are designed for a broader set of applications and include general purpose GPUs [1], Cell [2], and Larrabee [3]. They attain high throughput by executing applications on multiple processing units in parallel. Applications targeted for compute accelerators involve a significant number of operations on floating point data types. Floating point computation is complex and presents challenges to hardware designers who are trying to balance high performance with low area and power requirements. As the number of parallel processing units increases, floating point unit (FPU) architectures need to adapt to stricter area and power budgets while continuing to meet the high throughput demands of accelerators.

1.1 Notation

Throughout this thesis it is necessary to refer to individual bits, and bit ranges of multi-bit signals. A notation similar to that used in Verilog is used throughout this thesis. For an n-bit signal x, the LSB refers to the right-most bit, and is denoted as $x[0]$. The MSB refers to the left-most bit and is denoted as $x[n-1]$. To indicate that a portion of signal x is accessed, the notation $x[a:b]$ is used, where a indicates the starting bit of the range, b is the ending bit in the range, and a and b satisfy the condition $a > b$. For example, for an 8-bit signal sum, the notation $sum[7:5]$ indicates that the most significant three bits are accessed.

1.2 Thesis Organization

This thesis is organized in the following way. The rest of this chapter is devoted to describing Rigel and providing motivation for the investigation. Chapter 2 describes the floating point system as well as algorithms behind floating point operations. Chapter 3 covers implementation of each FPU component including an initial investigation using DesignWare blocks. It also presents a candidate design. Finally, Chapter 4 evaluates the performance of the candidate design using a Rigel benchmark suite.

1.3 Rigel: A Massively Parallel General Purpose Accelerator

1.3.1 Introduction

Rigel [4] is a MIMD compute accelerator targeted toward task oriented applications in the areas of visual computing, signal processing, and computational science. The architectural objective of Rigel is to provide high compute throughput with a minimum per-core area while still supporting a SPMD parallel model. Compared to existing accelerators which contain domain-specific hardware, specialized memories, and restrictive programming models, Rigel is more flexible and provides a more straightforward target for a broader set of appli-

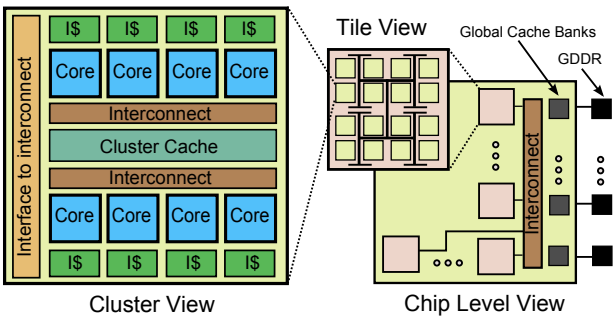


Figure 1.1: Diagram of the Rigel processor.

cations. Rigel's low-level programming interface adopts a single global address space model where parallel work is expressed in a task-centric, bulk-synchronized manner using minimal hardware support [5]. A block diagram of Rigel is shown in Figure 1.1.

The fundamental processing element of Rigel is an area-optimized dual-issue fine-grained in-order processing core. The cores execute a 32-bit RISC-like instruction set with 32 general-purpose registers. Cores are organized as *clusters* of eight cores attached to a shared write-back data cache called the *cluster cache*. The cores, cluster cache, core-to-cluster-cache interconnect and the cluster-to-global interconnect logic make a single Rigel cluster. Clusters are connected and grouped logically into a *tile*. Clusters within a tile share resources on a tree-structured interconnect. Tiles are distributed across the chip and are attached to *global cache* banks via a multi-stage crossbar interconnect. The global caches provide buffering for high-bandwidth memory controllers and are the point of coherence for memory.

1.3.2 Core Design

As mentioned previously, the Rigel core is a dual-issue in-order processing core. The pipeline is divided into several stages: a Fetch stage, a Decode stage, a four stage Execution phase, and a Writeback stage. Figure 1.2 illustrates the core pipeline.

In the Fetch stage, two instructions are loaded from the instruction cache. In the Decode stage, both instructions access the four-ported general purpose register file, and are dispatched to their respective execution units. The scheduler can dual-issue up to two instructions as long as no dependencies or structural hazards exist, and both instructions

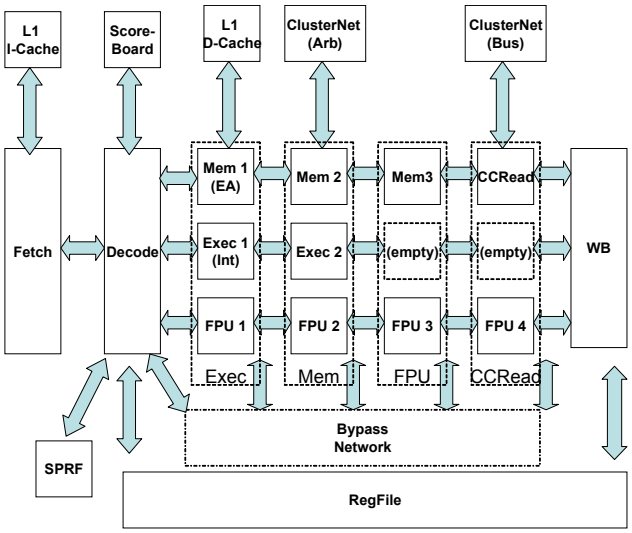


Figure 1.2: Diagram of the Rigel core pipeline.

belong to different execution pipes. The Execution pipeline is divided into three separate pipes: Integer/ALU, Floating Point, and Memory. To achieve high throughput, all three pipes are fully bypassed.

1.4 Motivation

Previous works [6], [7], [8], [9] provide a thorough investigation and description of FPU designs. However, the emergence of highly parallel processing units creates new challenges not previously seen or considered. In the past, higher performance could be obtained by increasing the operating frequency or by utilizing faster yet larger hardware designs. In a massively parallel processor like Rigel, area and power concerns make it necessary to investigate other options for increasing performance. The design of the core is simplified by not supporting traps or exceptions in hardware.

Additionally, performance of an accelerator like Rigel is measured in a different domain than performance of a general purpose CPU. Whereas the architecture of the latter is designed to minimize latency ($\frac{seconds}{instruction}$), an accelerator is designed for the highest throughput ($\frac{instructions}{second}$). At the chip level, Rigel achieves high throughput by utilizing over 1000 cores. At the core level, high performance is achieved by dual-issuing instructions and bypassing

results. The goal of this thesis is to develop a clear picture of an FPU design which balances performance, area, and power. Although the design is intended for Rigel, the results obtained in the analysis should serve as a guide for any massively parallel system.

CHAPTER 2

FLOATING POINT REPRESENTATION

2.1 Introduction

In an ideal world, all data would be an integer. It could then be represented in the binary format with ease. However, scientific, digital signal processing (DSP), and many other general purpose applications must handle inputs with fractional components. This poses a question: how does one represent a fractional number in binary, a system that is designed to work with whole values from the beginning? One solution is to use fixed binary point representation where some bits represent the whole part of the number and some represent the fractional part. One drawback to this approach, however, is that a highly precise fixed point representation has a limited range since a set number of bits is used to represent the decimal portion. Conversely, a fixed point representation with large range has low precision because fewer bits are used to represent the fractional component.

To address this issue, one needs a format which can represent a large range of values by varying the precision. Generally, numbers closer to zero need to be represented with higher precision, while large numbers are not expected to be as precise. For example, if one considers buying a $1,000,000 house, the cent amount is insignificant. But when deciding whether to buy gasoline at $2.00 per gallon versus $2.99 a gallon, the exact price makes a difference. Such representation is called floating point representation since the radix point "floats" and its location is not static as it is with fixed point representation. One form of floating point representation which is broadly used in industry follows the IEEE 754 Standard for Floating-Point Arithmetic [10].

2.2 IEEE 754 Standard

The IEEE standard defines a method of representing fractional values in binary, and it outlines numerous operations on floating point operands. Furthermore, the standard sets rules on how to handle abnormalities, as well as irregular operands and results, such as square roots of negative numbers. The standard has been updated and revised several times, and its latest major revision was in 2008.

2.2.1 Representation

As shown in Figure 2.1, a floating point number conforming to the IEEE standard is made up of a sign (S), an exponent (E), and a mantissa (M) field. The value represented by any given IEEE Floating Point number may be obtaining by using Equation (2.1).

$$value = (-1)^S * 2^{E-bias} * M \tag{2.1}$$

The IEEE standard defines several levels of precision: binary32 (single precision), binary64 (double precision), and binary128 (extended precision). The bit length of each field varies based on the precision of the floating point number and is outlined in Table 2.1.

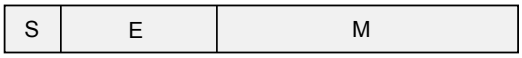


Figure 2.1: IEEE floating point representation.

The sign bit S is needed because IEEE Floating Point numbers are stored in a sign-magnitude format. S is zero for positive numbers, and one for negative numbers. The mantissa M is normalized such that it satisfies the condition $1.0_2 \leq M < 10.0_2$. Because of

Table 2.1: IEEE 754 Floating Point Bit-Lengths

Format	Sign Bit	Exponent	Mantissa	Total Size
binary32	1	8	23	32
binary64	1	11	52	64
binary128	1	15	112	128

this property, the MSB of the mantissa M is always implied to be one, so it is not stored as part of the floating point number. However, with this assumption in place a problem exists: M cannot hold values in the range $[0, 1.0)$. As outlined by Figure 2.2, this creates a large discontinuity between the number zero and the next representable value in a region where high precision is expected. To address this issue, the IEEE 754 standard defines an exception to the rule for numbers whose exponent value is zero. For such numbers the implied MSB of the mantissa is zero. The numbers whose exponent E is zero and mantissa M is non-zero are known as denormalized numbers. They allow IEEE Floating Point numbers to have high precision for the whole region near the number zero. Their inclusion in the standard is controversial because, as will be noted in further sections, they add more complexity to IEEE Floating Point computations.

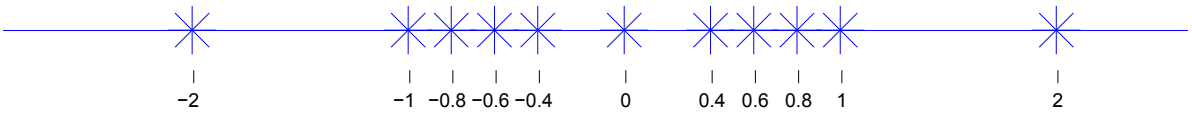


Figure 2.2: The effects of the implied one notation in the mantissa on number representability. Numbers very close to zero cannot be represented.

The exponent E is obtained by adding an offset to the 2's complement representation of the number of right shifts needed to normalize the mantissa. For an N-bit exponent, a bias value $2^{N-1} - 1$ is used as an offset. Biasing the exponent gives the IEEE Floating Point numbers a monotonically increasing property which enables the use of unsigned comparators to compare two IEEE Floating Point numbers [11]. Unsigned comparators are more advantageous than 2's complement comparators because of their smaller hardware size and reduced complexity.

IEEE Floating Point numbers may also express special values. Not-a-number (NaN) is used to identify a result of an illegal operation, such as $0/0$ or $\sqrt{-1}$. Section 2.2.4 describes NaNs and illegal operations in more detail. The values $\pm\infty$ are used to represent numbers which are out of range. To represent a special value, the N-bit exponent E field is set to the value $2^N - 1$. The mantissa field is set to zero to represent an out of range value. To represent a NaN, the mantissa field is set to a non-zero value.

2.2.2 Operations

The IEEE 754 standard defines several required operations on floating point numbers, and suggests some operations which may be implemented, but are not required by the standard. Among the required are the common operations such as multiplication, addition, division, square root, and conversion between integer and IEEE Floating Point data types.

2.2.3 Rounding

Many times a floating point operation produces a result which cannot be precisely represented using IEEE Floating Point numbers. Multiplication of two N-bit numbers, for example, may produce a product which requires $2N$ bits to represent. Only N bits may be kept in the final product, however, and a decision must be made whether or not to round the result based on the truncated bits.

The IEEE 754 standard defines five rounding modes which specify how to generate the final result. In the Round to Nearest, Ties Break to Even (RNE) mode, the result is rounded to the nearest representable number, with ties rounding to an even number. In the Round Away from Zero mode, the result is rounded to the nearest representable number, with ties rounding to the number of greater magnitude. In the Round to Zero mode the extra bits of the result are discarded, therefore rounding to a number with smaller magnitude. This method is also known as Truncation. The Round to Positive Infinity mode specifies that the final answer must be no smaller than the exact result. Finally, the Round to Negative Infinity mode specifies that the rounded result must be no greater than the infinitely precise result. The standard also specifies that a way to select any rounding mode during execution must be provided. By default, the Round to Nearest Even mode is selected. As an example, Table 2.2 shows the effects of each rounding mode on several values, assuming that the end result must be rounded to a whole number.

A common method of preserving precision without having to perform computations on arbitrarily wide operands is to use three status bits: guard, sticky, and round. The guard bit is the most significant bit that was shifted out and will not be a part of the final answer. The round bit is the second most significant bit that was shifted out. Finally, the sticky bit

Table 2.2: Effects of Rounding Modes on Results

Exact Value	To Nearest Even	Away from Zero	To Zero	To $+\infty$	To $-\infty$
-3.5	-4	-4	-3	-3	-4
-1.12	-1	-1	-1	-1	-2
1.5	2	2	1	2	1
2.5	2	3	2	3	2
2.51	3	3	2	3	2

is the result of ORing all the other shifted out bits. These three bits are all that is needed to generate a properly rounded result.

2.2.4 Handling Exceptions

The IEEE 754 standard specifies how to handle cases when it is not possible to generate a correct result. There are several times when this occurs. One is when operands have invalid values. This occurs when any of the operands are NaN, when multiplying zero by $\pm\infty$, when the divisor is zero, or the input into the square root function is negative, for example. Second is when valid operands generate an invalid result. This occurs when multiplication of two values generates a result which is too large to represent in a given format, for example.

The IEEE 754 standard does not specify how the system should behave after detecting an exception. However, it states that a method for signaling when exceptions occur needs to be provided. There are five flags which are used for this: Invalid Operation, Division by Zero, Overflow, Underflow, and Inexact.

As mentioned previously, the standard provides the NaN values to represent results of invalid operations. There are two types of NaNs: quiet NaN, and signaling NaN. Quiet NaNs are used to represent results when the Invalid Operation flag is set. With some exceptions, operations on quiet NaNs do not generate any exception flags. The result of operations on quiet NaNs is a quiet NaN whose bit pattern is the same as one of the NaN inputs. Signaling NaNs are usually used to represent uninitialized variables, and operations on signaling NaNs raise the Invalid Operation flag.

2.3 Floating Point Operations

This section describes the most common operations required by the IEEE standard in detail, and explains the algorithms for implementing them.

2.3.1 Multiplication

Multiplication of two floating point values follows basic algebraic concepts [12]. A number x may be rewritten as shown in Equation (2.2), where x_n is a normalized mantissa of x, $base$ is the number base (10 for decimal and two for binary), and exp is the number of shifts the radix point was shifted to the left to normalize x.

$$x_n \times base^{exp} \tag{2.2}$$

From Equation (2.1), it is clear that any IEEE Standard Floating Point number may be written in this manner, and the format provides all the components directly. Given this notation, the product of two numbers x and y may be obtained by the following procedure:

$$\begin{aligned} product &= x * y \\ &= x_n \times base^{exp_1} * y_n \times base^{exp_2} \\ &= x_n * y_n \times base^{exp_1 + exp_2} \end{aligned}$$

This splits the multiplication process into two parallel data paths. The first calculates the sum of the exponents, while the second calculates the product of the two mantissas. Because both data paths operate on standard integer values, they may be implemented using conventional hardware methods. A more detailed discussion with regards to hardware implementation may be found in Section 3.4.

When dealing with IEEE Floating Point numbers, the multiplication process involves several additional steps which are outlined in Figure 2.3. In the Unpack Operands (UO) stage, the mantissa and exponent fields of each operand need to be evaluated in order to correctly generate the implied MSB of the mantissa. To reiterate, the MSB is implied to be

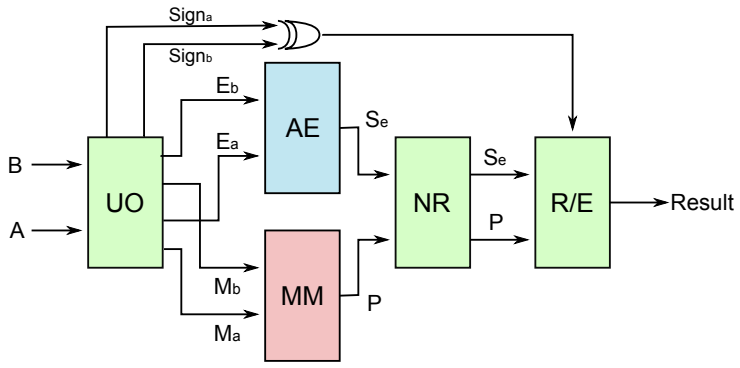


Figure 2.3: Data flow diagram for the IEEE 754 floating point multiply operation.

one for all cases except when the exponent is zero. If the exponent is zero and the mantissa is greater than zero, the exponent is set to one since this indicates that the operand is a denormalized number.

After unpacking, the 2n-bit product (P) is generated in the Multiply Mantissas (MM) stage by multiplying the two n-bit mantissas. In parallel, the exponent sum (S_e) is obtained in the Add Exponents (AE) stage by adding the biased exponents together and subtracting the exponent bias. The latter is needed because without the subtraction the sum would be $S_e = (exp_a + bias) + (exp_b + bias) = 2 * bias + exp_a + exp_b$, which would not be correct. The sign of the product is determined by XORing the sign bits of the two operands.

The Normalize Result (NR) stage follows next. Here, P is normalized to obtain a mantissa which conforms to the IEEE Floating Point format. Remembering basic algebra rules of multiplying two decimal numbers, if the first number contains a digits before and b digits after the radix point, and the second number contains o digits before and p digits after the radix point, the product of the two numbers will contain up to $a + b$ digits before and $b + p$ digits after the radix point. An n-bit mantissa contains one bit before and $n - 1$ bits after the binary point. Therefore, the product of two n-bit mantissas will contain $2n$ bits. Bits $P[2n - 3 : 0]$ are fractional bits, while $P[2n - 1 : 2n - 2]$ represent the whole number. If the MSB of P is one, the binary point is shifted one position to the left by right-shifting P and incrementing S_e. If $P > 0$ and $S_e < 1$, P needs to be right-shifted, and for each right shift S_e needs to be incremented until either $P = 0$ or $S_e = 1$. This occurs when the product is

a denormalized number. If denormalized operands are present, it is necessary to locate the leading one in P. This is done by left-shifting P and decrementing S_e for every shift until $S_e = 1$ or $P[2n-2] = 1$, whichever happens first. If, after shifting, $P[2n-2]$ is zero and $P > 0$, the result is a denormalized number. After normalization, bits $P[2n-2 : n-1]$ become the candidate mantissa.

Denormalized values introduce additional complexity. Therefore, some implementations which do not fully conform to the IEEE 754 standard flush all denormalized operands and results to zero. This simplifies the normalization hardware greatly, because at most only one right shift may have to be performed (which can be implemented using multiplexers). And in all cases where $S_e < 0$, the result is set to zero and treated as an underflow.

The next phase is the Rounding/Exception Handling (R/E) stage. Here the result is rounded following the guidelines described in Section 2.2.3. After rounding it may be necessary to re-normalize the mantissa, and to modify the exponent of the result accordingly. If the rounded result is a denormalized number, the exponent is set to zero. Additionally, the result is checked for validity as described in Section 2.2.4 so that the proper Exception Flags may be set. For example, if the product is a denormalized number, the Underflow Flag needs to be set, and the exponent of the result needs to be set to zero. On the other hand, if one of the operands was zero and the other ∞, then the Invalid Flag needs to be raised and the result is changed to a quiet NaN.

2.3.2 Addition

Addition is a conceptually simple operation. Humans can perform this task without much difficulty, even on floating point numbers. Computers, on the other hand, have a much more difficult time summing floating point values. Floating point addition requires multiple dependent steps which must be executed serially, making it a long latency task.

For the IEEE Floating Point operands the basic floating point addition algorithm can be broken down into the following steps [13]:

1. **Operand Unpacking:** The signs, exponents, and mantissas of both operands are separated. The appropriate MSB is appended to each mantissa. Normally, the MSB is

one for a non-zero operand. However, in the case of denormalized numbers, the MSB is set to zero and the exponent is set to one. The effective operation is determined by evaluating the sign bits as well as the opcode of the instruction. If the instruction is an addition and the signs of the operands differ, or if the instruction is a subtraction and the signs of the operands are the same, then the effective operation is subtraction. Otherwise the effective operation is addition. The mantissas are also converted from sign magnitude form to 2's complement representation based on their respective signs.

2. **Operand Swapping:** The exponents are compared to determine which exponent is bigger (E_{max}) and which one is smaller (E_{min}). The operands are swapped so that M_{max} is the mantissa of the operand with the bigger exponent, and M_{min} becomes the mantissa of the operand whose exponent is E_{min}. This is done so that the exponent difference (calculated in the next step) is always positive.

3. **Operand Alignment:** Here, the operand whose exponent is E_{min} needs to be aligned so that its exponent is equal to E_{max}. This is done by first computing the difference of the two exponents: $d = E_{max} - E_{min}$. Next, the mantissa M_{min} is right-shifted by d places. The tentative exponent of the sum is E_{max}.

4. **Mantissa Addition:** The two mantissas are added together. Due to swapping, the resulting operation becomes $M_{max} + M_{min}$ in the case of addition, and $M_{max} - M_{min}$ in the case of subtraction. If the result is negative, a 2's complement operation needs to be performed to convert it to sign magnitude format. This involves a negation and addition.

5. **Normalization:** The sum needs to be normalized so that the MSB is one, or in the case of denormalized numbers the exponent is one. This involves finding the most significant one and then left-shifting the sum. For each left shift E_{max} is decremented by one.

6. **Rounding:** The sum must be rounded correctly (see Section 2.2.3). This involves an optional incrementation of the sum.

7. **Renormalization:** It may be necessary to normalize the rounded sum by right-shifting it, and incrementing E_{max}.

8. **Exception Detection/Final Result Generation:** Based on the inputs and the final value of the result, the exception flags are generated. Additionally, if any of the operands were NaNs or infinities, the appropriate result is generated. The final sum is generated based on the input operands, the effective operation, rounded result, and any exceptions which were detected.

Figure 2.4(a) illustrates the above steps. Reducing the latency of addition can only happen if the number of these serial steps is reduced. There are several key observations which aid in optimizing addition:

1. It is possible to reduce the number of 2's complements to one. When the effective operation is subtraction, by swapping the operands so that the smaller number is subtracted from the larger number, 2's complement no longer needs to be performed on both operands in step 1 and on the result in step 4. Instead, only M_{min} is negated (1's complement) and a carry is added in step 4 to perform the 2's complement on the smaller number.

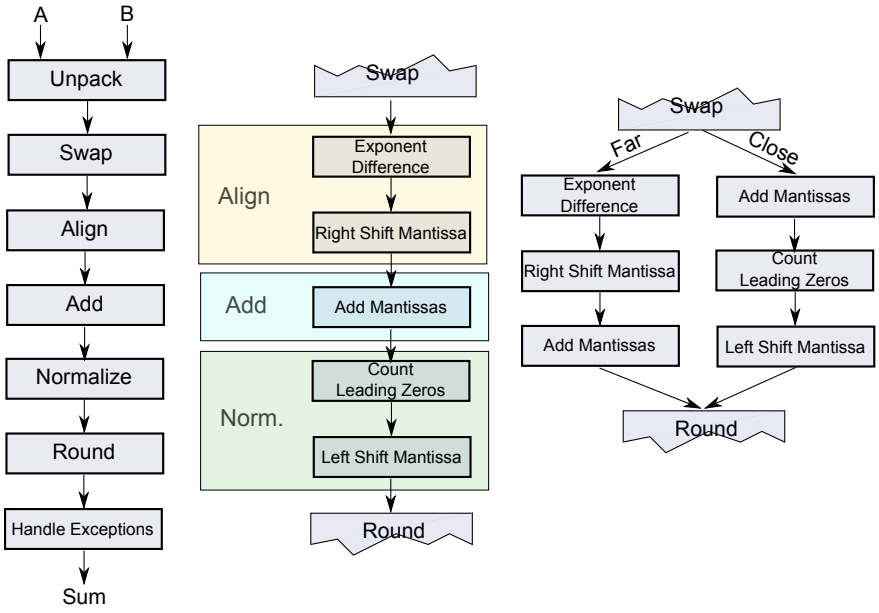
2. As shown in Figure 2.4(b) the Operand Alignment step involves a large right shift, and the Normalization step involves a large left shift. These shifts are mutually exclusive [9]. The initial left shift is needed only when the operands are very different in magnitudes; more concretely, when the exponent difference d is greater than one. The right shift during Normalization is needed only when subtraction causes most of the most significant bits of the result to be zero, known as "massive cancellation." It happens only when the magnitudes of the two operands are similar; more concretely, d is less than or equal to one. Therefore, it is possible to split the addition path into two parallel paths: a CLOSE path for $d \leq 1$ and the FAR path for $d > 1$. Figure 2.4(c) illustrates the modified data path.

3. It is possible to parallelize a portion of the Normalization step with the Mantissa Addition step. In particular, the leading zero counter can be replaced with a leading

zero anticipator which operates in parallel with the addition process [14].

4. Not all operands require that every step be executed to generate the proper result. Therefore, it is possible to output the sum earlier for some values [15].

5. It is also possible to speed up the rounding step by pre-computing all of the possible results using a specialized adder [16].



(a) (b) (c)

Figure 2.4: An illustration of the steps involved in the basic IEEE Floating Point addition process (a). The Align, Add, and Normalization stages are shown in greater detail (b) to show how they can be optimized using the dual path implementation (c).

It is important to remember that the IEEE standard dictates that the sum must be either exact or precisely rounded if it cannot be represented exactly. It may seem, then, that due to the arbitrarily long right shifting during alignment the addition must be done on infinitely wide operands. However, that is not the case. For an n-bit mantissa, it is possible to retain all the necessary information to generate a precisely rounded result by summing two $(n + 1)$-bit operands. In the Alignment step both mantissas are first appended with a zero at the LSB. This is done so that the proper value is returned during subtraction due to 2's complement. M_{min} is then right-shifted d places (where d is the exponent difference), but the values of the bits which were shifted out are noted. After the alignment shift is

completed, the most significant bit that was shifted out becomes the guard bit, and the result of ORing the other shifted out bits is known as the sticky bit. The guard and sticky bits are used in the rounding process to determine whether the number should be rounded up or down [12]. After alignment, the $(n+1)$-bit M_{max} and M_{min} are added or subtracted together (depending on the effective operation). This is best illustrated in Figure 2.5.

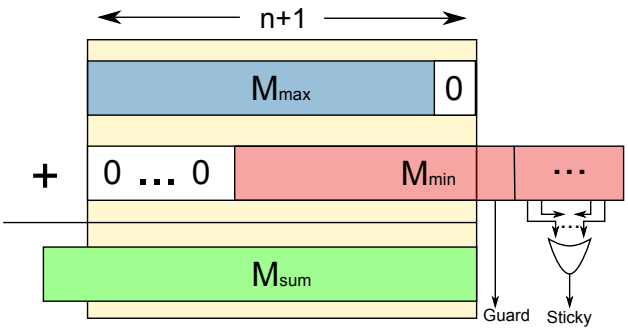


Figure 2.5: An illustration of the alignment process.

2.3.3 Fused Multiply and Add

Multiply and add operations are among the most common floating point operations [8]. In many cases the result of a multiply operation is subsequently added to some other number. Often several products are added together, or accumulated, into a running sum. The dot product takes the form $x = a_0 * b_0 + a_1 * b_1 + ... + a_i * b_i$ and is one of the most obvious examples of multiplication and accumulation. In pseudo code, the dot product can be expressed as a series of multiply and add pairs as shown in Figure 2.6(a).

```
mul temp0,a0,b0
mul x,a1,b1
add x,x,temp0
mul temp0,a1,b1
add x, x, temp0
.....
mul temp0, ai,bi
add x, x, temp0
```

(a)

```
mul x,a0,b0
fmadd x,a1,b1,x
fmadd x,a2,b2,x
...
fmadd x,ai,bi,x
```

(b)

Figure 2.6: Pseudo code for calculating the dot product using separate multiply and add operations (a), and using fused multiply-add operations (b).

Because accumulating the products is an extremely common task in scientific computations [17], computer architects combined the separate multiplication and addition operations into a single fused multiply-add operation (FMADD). With the FMADD operation in place, the dot product can be streamlined into the code shown in Figure 2.6(b).

Originally, the IEEE 754 standard did not support a fused multiply-add operation on floating point numbers, so each manufacturer which chose to support it was free to implement this functionality in their own way. This opened doors to inconsistencies since some implementations rounded the intermediate product, while others added the exact product to the sum before rounding the final answer. As of its 2008 revision, the IEEE 754 standard supports a floating point FMADD operation, and dictates the exact way it should be implemented and executed. In the code in Figure 2.6(a), each time a multiplication is performed the product is rounded. When an add operation is executed, the sum is rounded as well. Therefore, there are two rounding steps for each pair of multiply and add operations. This results in decreased precision and introduces greater rounding errors for long streams of computations. The IEEE 754 standard FMADD operation, on the other hand, only rounds the final sum and not the intermediate product. The data path for an FMADD implementation conforming to the IEEE standard is shown in Figure 2.7.

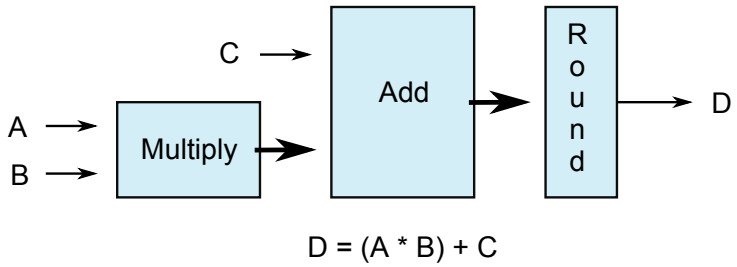


Figure 2.7: Fused multiply-add data flow.

Besides the increased precision, the fused multiply and add operation provides a greater benefit in terms of reduced code size and an increased instruction issue bandwidth. As can be seen by looking at the two dot product implementations in Figure 2.6, each FMADD operation replaces two separate instructions yielding a speedup of up to two. Section 4.2 contains an in depth analysis of the performance benefits of the FMADD operation.

2.3.4 Comparison

The ability to compare two floating point numbers is invaluable because it gives programmers the freedom to use floating point data types in control structures such as loops or if/else statements. One way to compare two floating point numbers when a hardware comparator is not available, is to write a software algorithm which uses an integer comparator. However, this method may require several cycles before the result of the comparison is ready. This is not an ideal solution for an accelerator because any conditional statements depending on the result of the floating point comparison will stall. Therefore, a fast hardware floating point comparator is needed.

The method for comparing two floating point numbers is straightforward and does not require complicated hardware. Because IEEE Floating Point numbers are stored in a sign-magnitude, monotonically increasing format, an unsigned integer comparator with some additional control logic is all that is needed to produce the correct result [11]. When comparing two positive values, the result of the comparator is directly fed to the output. However, when either of the numbers is negative, the results of the comparator's "less than" and "greater than" outputs are inverted before being passed to the final output. Additionally, one must consider special cases such as the existence of positive and negative zeros, as well as infinities and NaNs. In the case of zeros, their signs are ignored and two zero values are treated as equal. In the case of infinities, the unsigned comparator produces the correct result except for the "equal to" case. This comparison raises the invalid flag, as does any comparison on a NaN. A schematic of the complete floating point comparator is shown in Figure 2.8.

2.3.5 Division and Square Root

Division of floating point values and the square root function are complex, and therefore long latency, operations. Many implementations have been proposed for division and square root algorithms, and they are evaluated by Oberman [7]. The most common implementations are based on iterative digit recurrence algorithms. These offer smaller and less complex hardware at a cost of longer latency and low throughput. To obtain shorter latency, look-up tables may be used. Another method to lower the latency of division is to first compute the

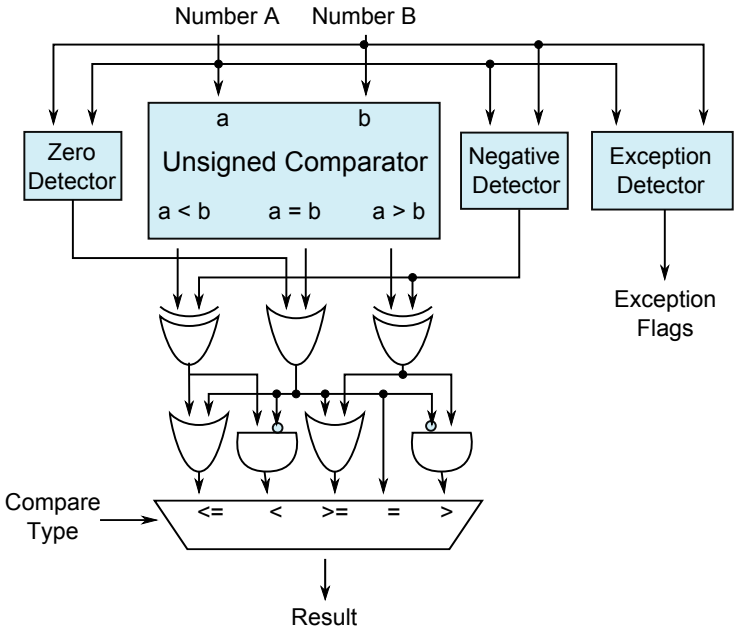


Figure 2.8: IEEE floating point comparator.

reciprocal of the divisor. The quotient is the product of the dividend and the reciprocal. The principal characteristics of the square root algorithm are similar to division [7].

2.3.6 Absolute Value

Absolute value is a very simple operation on IEEE Floating Point numbers. Since the values are stored in sign-magnitude format, all that is needed to compute the absolute value of a floating point number is to change the sign bit to zero. Additionally, it is important to check for exceptions.

CHAPTER 3

FPU DESIGN AND IMPLEMENTATION

3.1 Methods

Rigel is implemented using the Synopsys ASIC flow targeted for a commercial 40 nm process at 1.2 GHz. All synthesis was performed with an operating voltage set to 0.9 V with nominal conditions and targeted for minimum area. Initial design investigation (see Section 3.2.2) used DesignWare components synthesized using Design Compiler with automatic repipelining turned on. Most of the FPU was implemented using the Arithmatica CellMath suite using the SAVI Language. The SAVI implementation was synthesized using Arithmatica with ultra high effort and automatic repipelining turned on. The resulting netlist was integrated into the Rigel pipeline by performing an ultra high effort incremental compilation with automatic repipelining in Design Compiler's topographical mode. The design was verified using the Arithmatica generated bit accurate C-model of the design. The outputs from the model were compared to the outputs of a separate C program.

3.2 Design Space Exploration

3.2.1 Targeted Applications

Rigel is a massively parallel general purpose accelerator. It is designed for computationally intensive applications in the area of visual computing, signal processing, and scientific computation. Tasks such as vector scaling, dot products, and vector addition are very common in the targeted applications. As indicated by Oberman [8], multiplication and addition are the most common floating point operations performed by such applications; therefore, it is

critical to optimize the adder and multiplier for best performance. Additionally operations such as division, square root, absolute value, and conversion between the integer and floating point format need to be supported either by using software emulation or, ideally, directly in the hardware.

3.2.2 DesignWare Study

The Rigel group uses a Synopsys based design flow with access to Synopsys DesignWare Intellectual Property (IP). DesignWare is a collection of pre-designed logic blocks licensed by Synopsys. Of particular interest are the floating point units, such as an adder/subtracter, multiplier, square root unit, and dividers. All of these units are parametrized and offer features such as IEEE compliant rounding modes, optional denormalized number support, and various precisions. It is possible to use DesignWare blocks to implement a fully functional FPU. However, given the fact that the group also has access to Arithmatica, the decision was made to implement the whole FPU in Arithmatica.

However, before attempting to design a full FPU in Arithmatica, it was important to determine what the base parameters should be. DesignWare floating point adder/subtracter (DW_fp_addsub), multiplier (DW_fp_mult), square root (DW_fp_sqrt), and reciprocal (DW_fp_recip) blocks were characterized to obtain the expected area, power, and latency estimates for the overall design. Synthesis was performed at various frequencies: 750 MHz, 1.0 GHz, 1.2 GHz, 1.5 GHz. For all these frequencies, single and double precision were investigated. For all these, all DesignWare supported rounding modes [18] were varied. Denormalized number support was turned on and off. Finally, latency was varied from one to four stages. Since the DesignWare blocks in question contain no sequential logic, registers were added at the outputs and automatic repipelining was turned on in Design Complier.

DW_fp_addsub

Table 3.1 shows a summary of results from characterizing the DesignWare adder/subtracter. At 750 MHz, a single and double precision single-stage adder/subtracter is possible. At 1.0 GHz and 1.2 GHz at least two stages are needed for either precision. At 1.5 GHz, at least

two stages are needed for single precision; however, double precision requires at least three stages. Figure 3.1 shows the estimated area of a single precision adder/subtracter for various latencies and frequencies with no denormalized number support with RNE rounding mode.

The area savings obtained by not supporting double precision are illustrated in Figure 3.2 for each configuration of frequency, latency, rounding, and denormalized number support; the figure shows how much smaller (in percent) the area is when the precision is changed from double to single. On average, a double precision adder/subtracter requires about twice as much area as a single precision adder/subtracter with identical parameters. The shaded area shows the region within half a standard deviation away from the mean. The large deviations from the mean are a consequence of the synthesis process when automatic repipelining is turned on.

Figure 3.3 illustrates how much smaller (in percent) the area of an adder/subtracter which flushes denormalized values to zero is than a fully compliant single precision IEEE Floating Point adder/subtracter. On average, the area of an adder/subtracter which does not support denormalized values is between 10 and 23 percent smaller than that of a fully compliant adder/subtracter. The large variance is once again a fault of the synthesizing with automatic repipelining.

Figure 3.4 shows the area savings (in percent) obtained by truncating the result instead of implementing the RNE mode. An additional 5 to 11 percent more area is required if RNE mode is implemented. This is consistent with the fact that although rounding requires an extra adder and additional control logic, these do not require a lot more extra area.

Table 3.1: DW_fp_addsub Characterization Summary

Frequency	Precision	Stages Required	Power Consumption (mW)
750 MHz	Single	1	1.5
750 MHz	Double	1	3.1
1.0 GHz	Single	2	2.4
1.0 GHz	Double	2	4.0
1.2 GHz	Single	2	3.2
1.2 GHz	Double	2	5.6
1.5 GHz	Single	2	3.7
1.5 GHz	Double	3	7.1

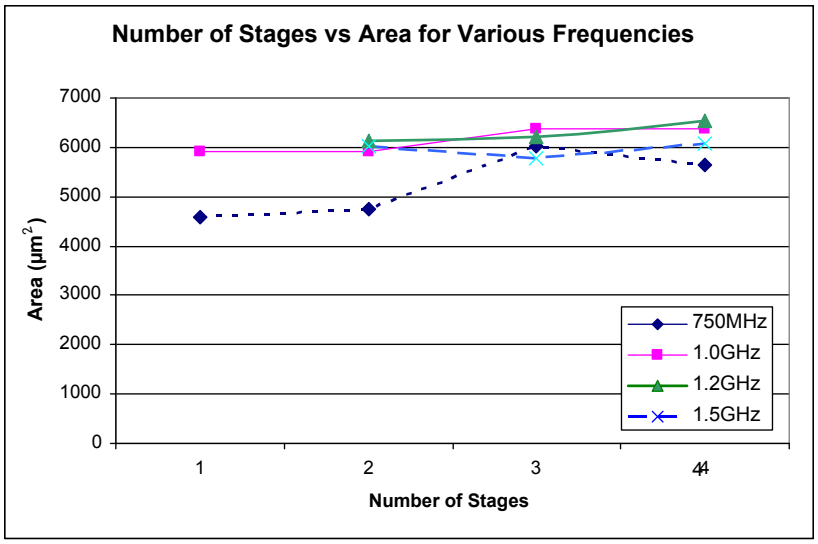


Figure 3.1: The estimated area of a single precision adder/subtracter with Round to Nearest Even mode and no denormalized number support for varying latencies and frequencies.

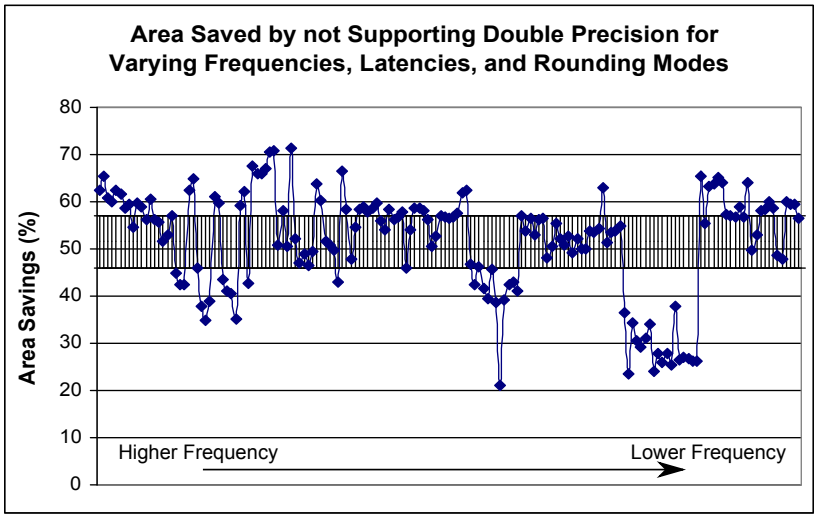


Figure 3.2: Area savings (in percent) in an adder/subtracter obtained from not supporting double precision for various frequencies, latencies, and rounding modes. The shaded region indicates area within half a standard deviation.

DW_fp_mult

Table 3.2 shows a summary of results from characterizing the DesignWare multiplier. At 750 MHz, a single stage, single and double precision multiplier is possible. At 1.0 GHz and 1.2 GHz at least two stages are needed for either precision. At 1.5 GHz, at least two

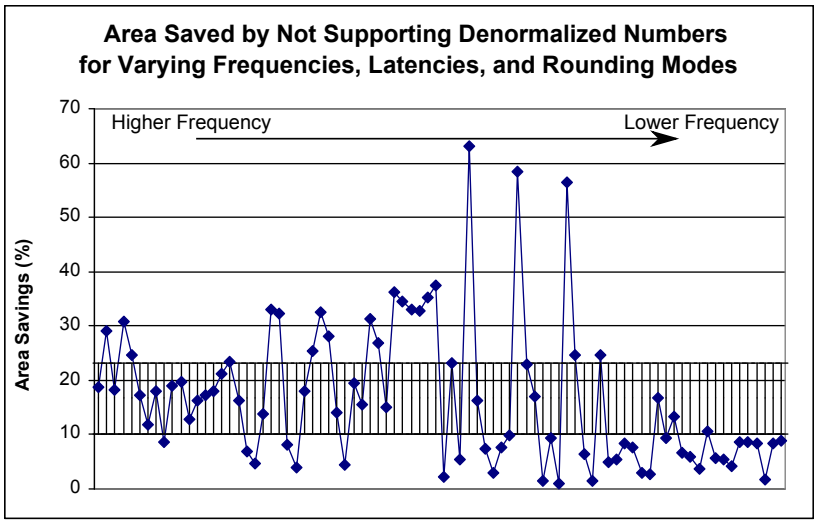


Figure 3.3: The area saved (in percent) by flushing all denormalized operands and results to zero in an adder/subtracter instead of supporting them according to the IEEE 754 Standard. The shaded region indicates area within half a standard deviation.

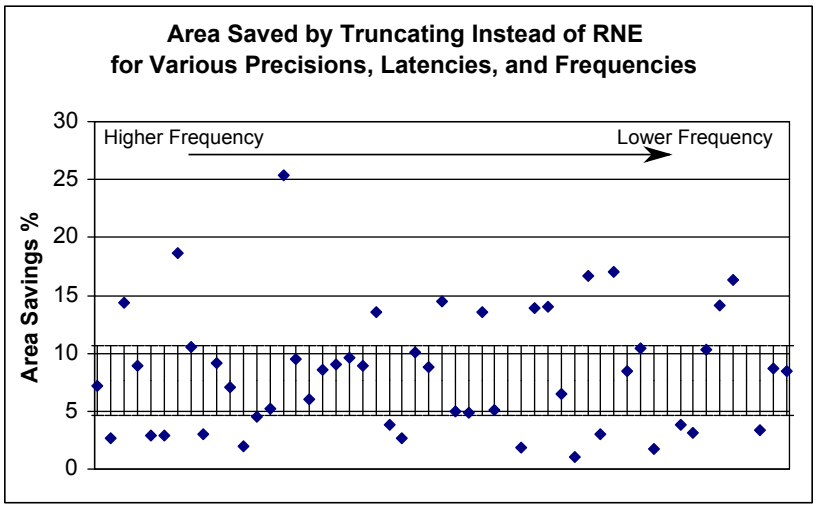


Figure 3.4: The area saved (in percent) by truncating the results of an adder/subtracter instead of rounding them to nearest even. The shaded region indicates area within half a standard deviation.

stages are needed for single precision; however, double precision requires at least three stages. Figure 3.5 shows the estimated area of a single precision multiplier for various latencies and frequencies. Denormalized number support was turned off and Round to Nearest Even mode was selected.

The area saved by not supporting double precision is illustrated in Figure 3.6. For every

Table 3.2: DW_fp_mult Characterization Summary

Frequency	Precision	Stages Required	Power Consumption (mW)
750 MHz	Single	1	1.9
750 MHz	Double	1	10.3
1.0 GHz	Single	2	4.0
1.0 GHz	Double	2	15.1
1.2 GHz	Single	2	5.3
1.2 GHz	Double	2	23.8
1.5 GHz	Single	2	6.3
1.5 GHz	Double	3	31.7

configuration of frequency, latency, rounding, and denormalized number support, the figure shows how much smaller (in percent) the area is when the precision is changed from double to single. On average, a single precision multiplier requires about 75 percent less area than a double precision multiplier with identical parameters. The shaded area shows the region within half a standard deviation away from the mean. The large deviations from the mean are a consequence of the synthesis process when automatic repipelining is turned on.

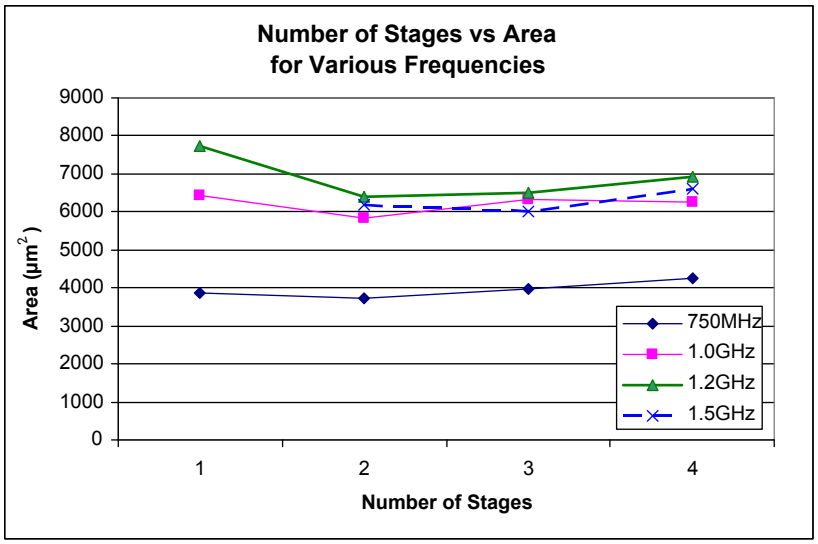


Figure 3.5: The estimated area of a single precision multiplier with Round to Nearest Even mode and no denormalized number support for varying latencies and frequencies.

Figure 3.7 illustrates how much smaller (in percent) the area of a multiplier which flushes denormalized values to zero is than that of a fully compliant single precision IEEE Floating Point multiplier. The area saved by flushing all denormalized operands and results to zero

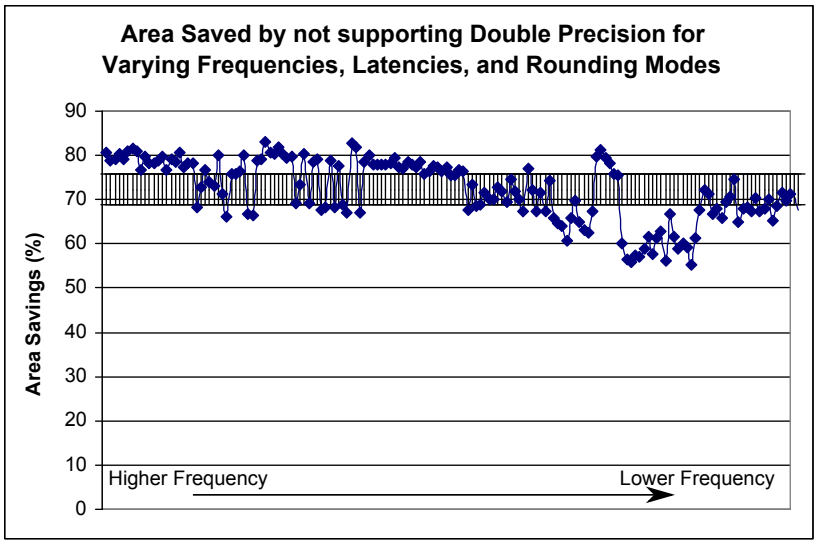


Figure 3.6: Area savings (in percent) in a multiplier obtained from not supporting double precision for various frequencies, latencies, and rounding modes. The shaded region indicates area within half a standard deviation.

is between 10 and 23 percent.

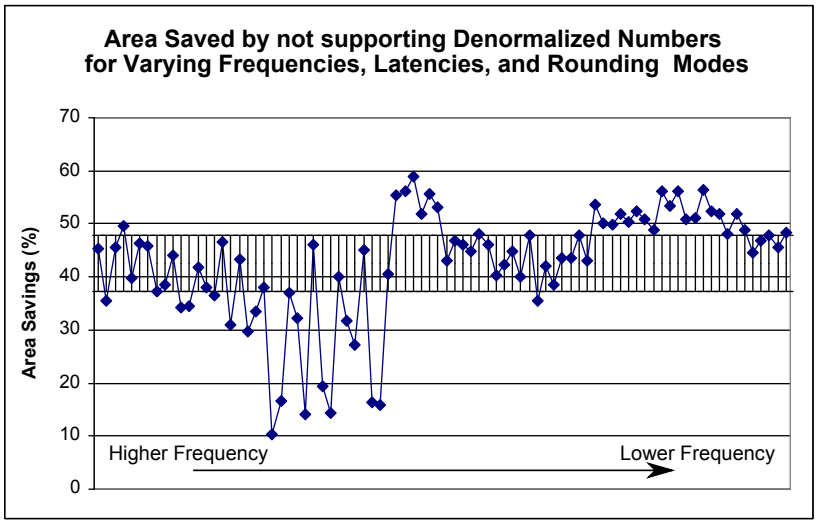


Figure 3.7: The area savings from not supporting denormalized numbers in a multiplier. The shaded region indicates area within half a standard deviation.

Figure 3.8 shows the area savings (in percent) associated with truncating the product instead of implementing the RNE mode. Between 4 and 9 percent of area is saved. As was the case with the adder/subtracter, these savings are consistent with the fact that rounding requires an extra adder and additional control logic, neither of which takes up a significant

portion of the area.

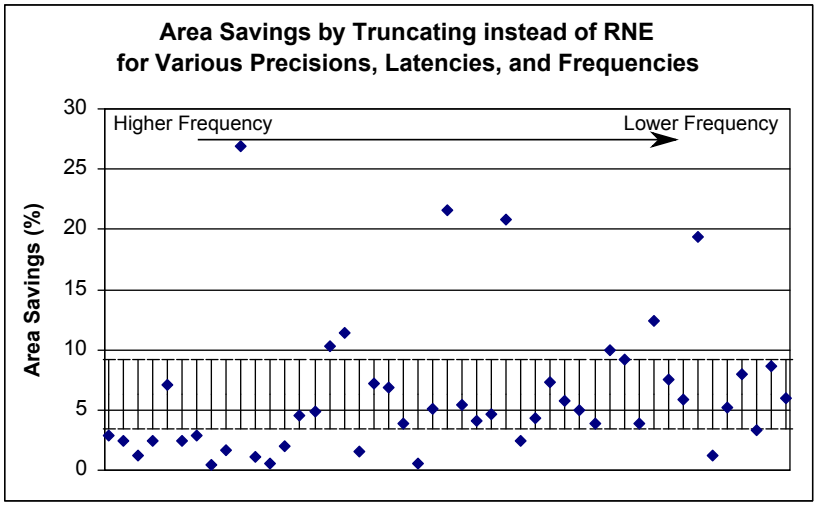


Figure 3.8: The area saved (in percent) by truncating the results of a multiplier instead of rounding them to nearest even. The shaded region indicates area within half a standard deviation.

DW_fp_sqrt

Table 3.3 shows a summary of results from characterizing the DesignWare Square Root unit. At 750 MHz, at least three stages are required for single precision. At 1.0 GHz, four stages are needed for single precision. At 1.2 GHz, at least five stages are needed; however, if Truncate mode is selected a four stage square root unit is feasible. At 1.5 GHz, six stages are required for RNE mode, while five are required if Truncation is used. Figure 3.9 shows the estimated area of a single precision square root unit for various latencies and frequencies. Denormalized number support was turned off and RNE mode was selected.

Table 3.3: DW_fp_sqrt Characterization Summary

Frequency	Precision	Stages Required	Power Consumption (mW)
750 MHz	Single	3	4.5
1.0 GHz	Single	4	6.3
1.2 GHz	Single	5 (4 with Truncation)	9.1
1.5 GHz	Single	6 (5 with Truncation)	13.5

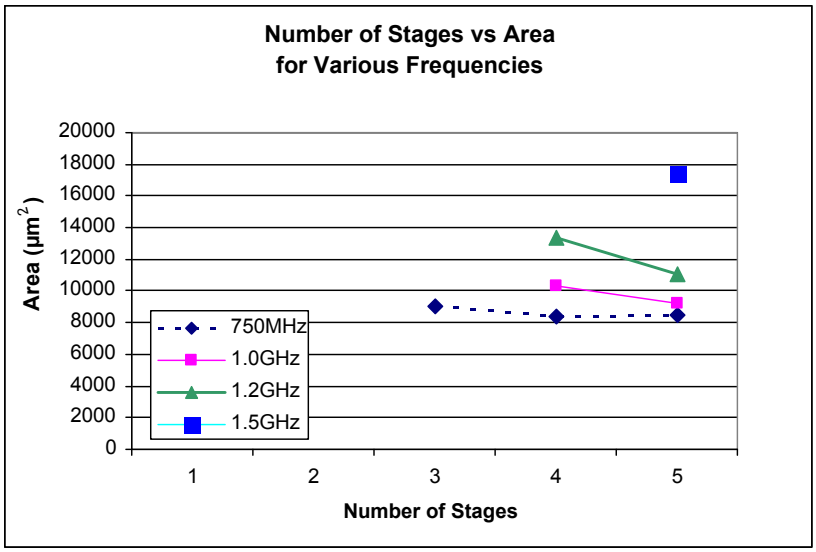


Figure 3.9: The estimated area of a single precision square root unit with RNE mode selected and no denormalized number support for varying latencies and frequencies.

DW_fp_recip

Table 3.4 shows a summary of results from characterizing the DesignWare Reciprocal unit. At 750 MHz, at least two stages are required for single precision. At 1.0 GHz and 1.2 GHz, at least three stages are needed. Although timing was met with two stages at 1.0 GHz, the area was about 60 percent smaller when latency was set to three stages. At 1.5 GHz at least four stages are required. Figure 3.10 shows the estimated area of a single precision reciprocal unit for various latencies and frequencies. Denormalized number support was turned off and RNE mode was selected.

Table 3.4: DW_fp_recip Characterization Summary

Frequency	Precision	Stages Required	Power Consumption (mW)
750 MHz	Single	2	5.3
1.0 GHz	Single	3	6.1
1.2 GHz	Single	3	9.6
1.5 GHz	Single	4	14.8

As far as rounding is concerned, implementing the RNE mode requires about 4 to 6 percent more area than Truncation. At high frequencies, the choice of a rounding mode becomes more critical because several short latency configurations which met timing with truncation did not meet timing when RNE mode was selected. One interesting feature of this block

is the ability to turn on "Faithful Rounding." This introduces a small error into the result, but it saves about 35 percent in area and improves the ability to meet timing with fewer stages at high frequencies. For example, with Faithful Rounding turned on, a three-stage reciprocal unit meets timing at 1.5 GHz.

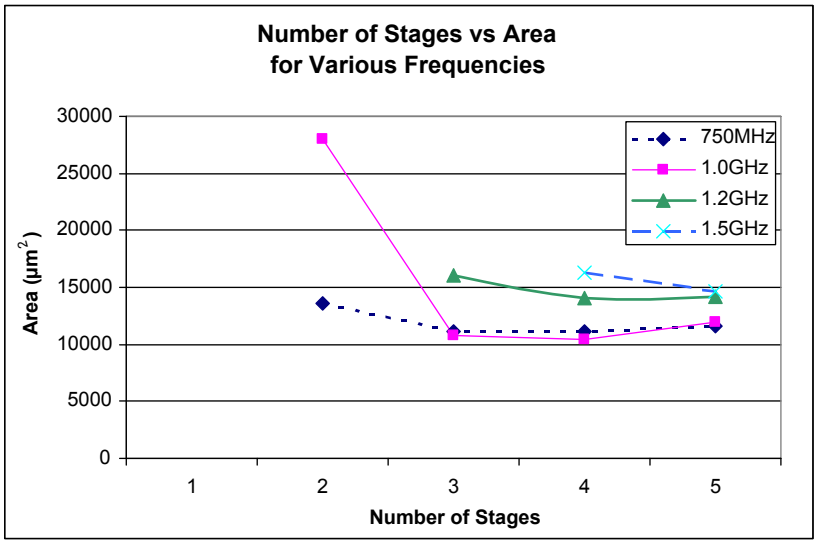


Figure 3.10: The estimated area of a single precision reciprocal unit with Round to Nearest Even mode selected and no denormalized number support for varying latencies and frequencies.

3.3 Adder Implementation

The DesignWare study on an adder/subtracter (see Section 3.2.2) provided a direction for implementing a custom adder/subtracter in Arithmatica. Not only was a design which fit into the targeted four stages possible, but an adder/subtracter with a two-cycle back-to-back latency was not out of the question. The results showed that the additional area required to support double precision floating point numbers as well as denormalized values was too much for Rigel. Therefore, the goal was to implement a single precision adder/subtracter with no denormalized number support and two to three cycle latency.

As mentioned in Section 2.3.2, floating point addition is a multi-step process and several designs have been proposed to reduce its latency. Most of the time, however, the reduced latency comes at a cost of more area. This is due to either duplicating hardware units

when computing several sums in parallel [16] [9], or using hardware such as a leading zero anticipator (LZA) to predict the outcome of operations [14]. Because of the massively parallel nature of Rigel, area is of the utmost concern. At the same time, being an accelerator, Rigel favors higher throughput and places less emphasis on shorter latency.

Two designs were implemented. The first one was the dual path design which incorporates the CLOSE and FAR paths [9] and it is illustrated in Figure 3.11. The second design used the single path approach and is shown in Figure 3.12. The dual path design was investigated in case the single path design did not meet timing at lower latencies. In both designs the operands were swapped such that only one 2's complement had to be performed in case of subtraction (refer to Section 2.3.2). For both designs, the area impact of RNE was investigated. Additionally, denormalized values were flushed to zero. Automatic repipelining was used in Arithmatica; therefore, the adder was implemented as one combinational block with movable registers at the outputs.

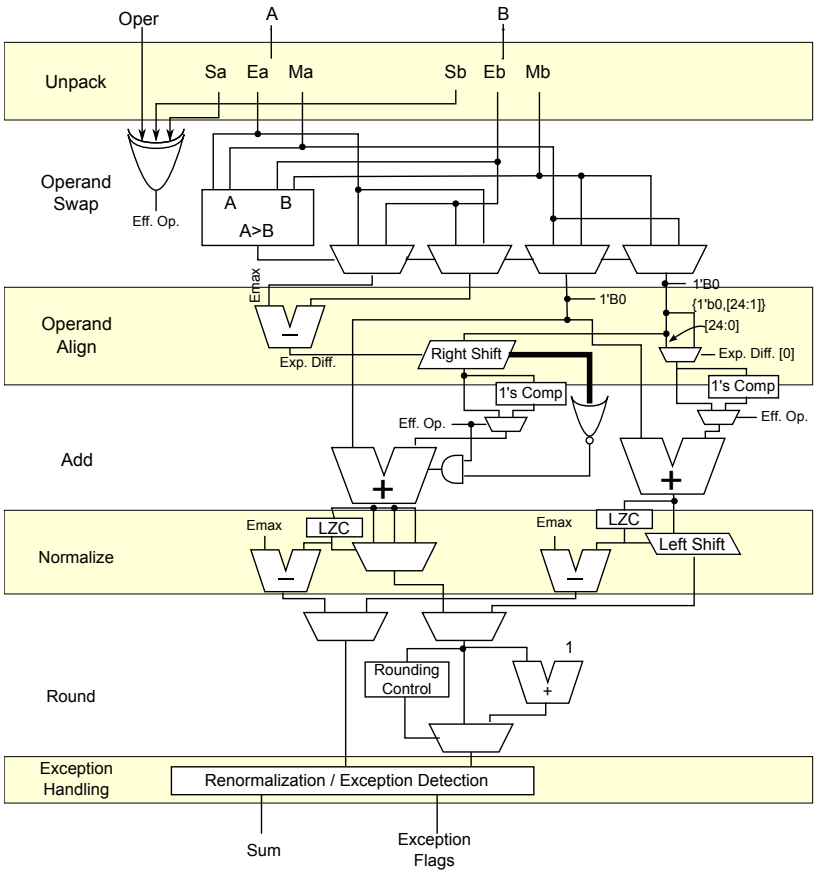


Figure 3.11: Diagram of the dual path adder implemented in Arithmatica.

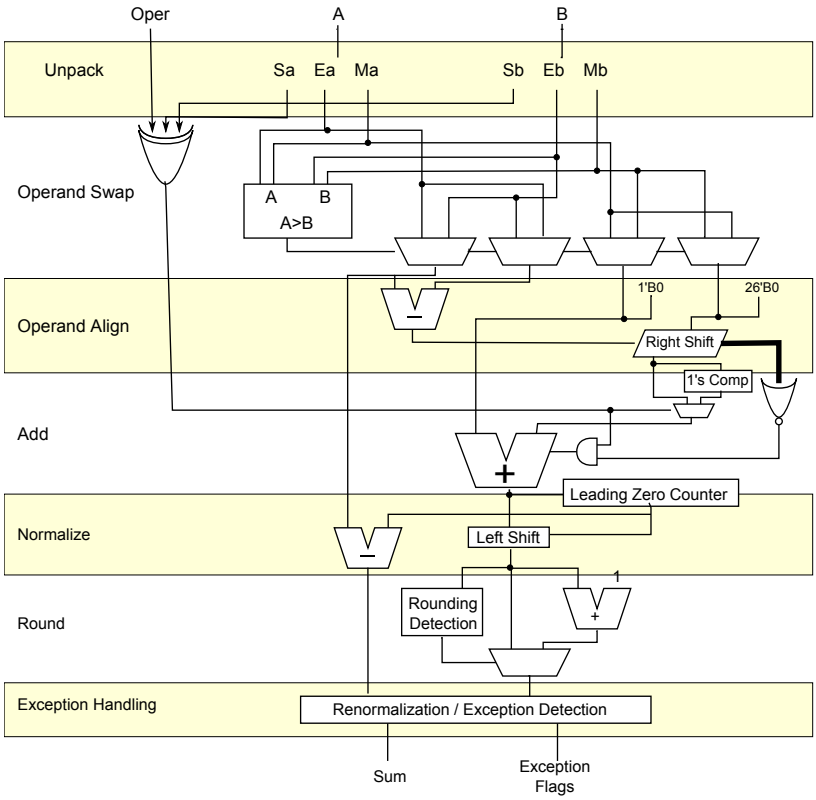
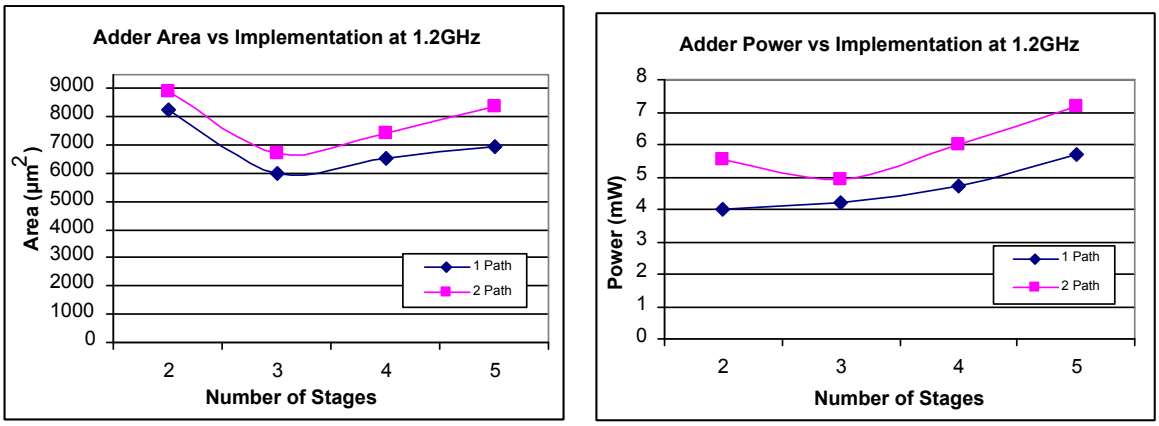


Figure 3.12: Diagram of the single path adder implemented in Arithmatica.

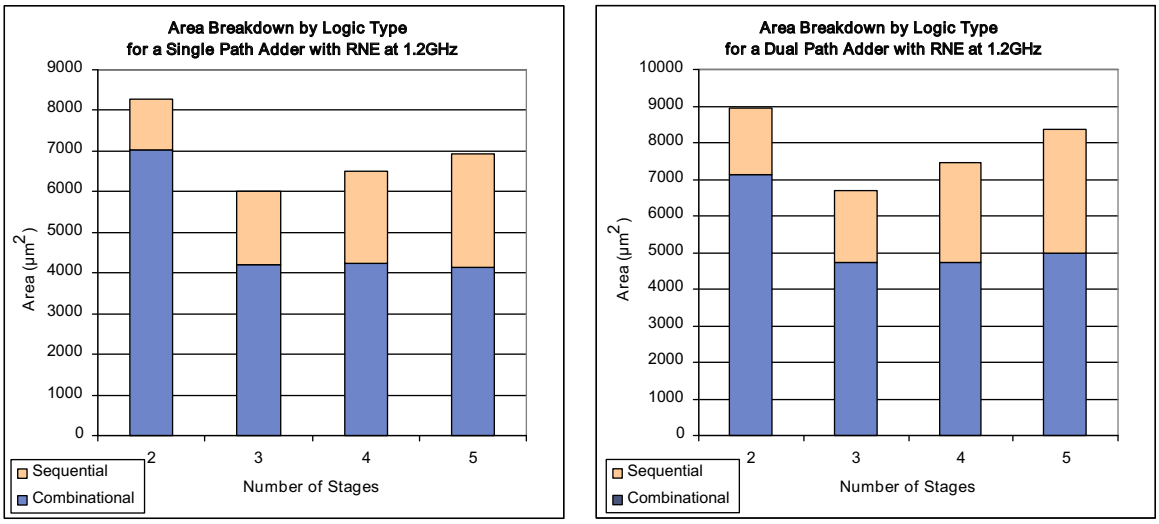
Both designs meet timing at 1.2 GHz. Figure 3.13 shows the area and power consumption of the dual path and the single path implementations at varying latencies with RNE rounding mode. At low latency the difference in areas between the two implementations is only 5 percent. This is because the single path design was upsized significantly to meet timing at the short latency. However, at higher latencies the difference is greater. At longer latencies the dual path design is not needed, and the second path takes up unnecessary space. Additionally, the dual path adder consumes between 20 and 30 percent more power. These results make the single path implementation a clear winner at 1.2 GHz.

Although a two stage adder/subtracter meets timing, a three stage design looks more ideal in terms of area and power. The combinational area reaches a minimum of around 4,100 μm^2 at three stages. As the number of stages increases past three, the combinational area stays constant, but the sequential area increases because more latches are added to increase the number of stages. The breakdown of area by combinational logic and sequential logic for the single path and dual path implementations is shown in Figure 3.14.



(a) (b)

Figure 3.13: The areas and power consumption of the single precision adder/subtracter implemented in Arithmatica using the single path and dual path methods at 1.2 GHz.



(a) (b)

Figure 3.14: The area breakdown by logic type for the Arithmatica generated single and dual path adder implementations at 1.2 GHz.

The area and power results obtained from comparing rounding modes in the single path implementation are shown in Figure 3.15. The savings by truncating the sum instead of rounding to nearest even are between 10 and 13 percent. The difference in power consumption for the two rounding modes is insignificant and is lost in the synthesis noise.

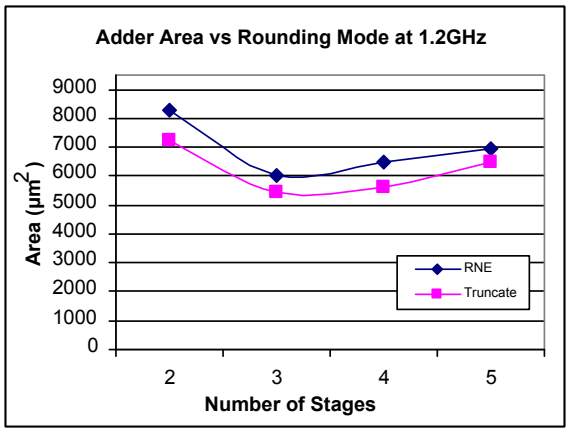


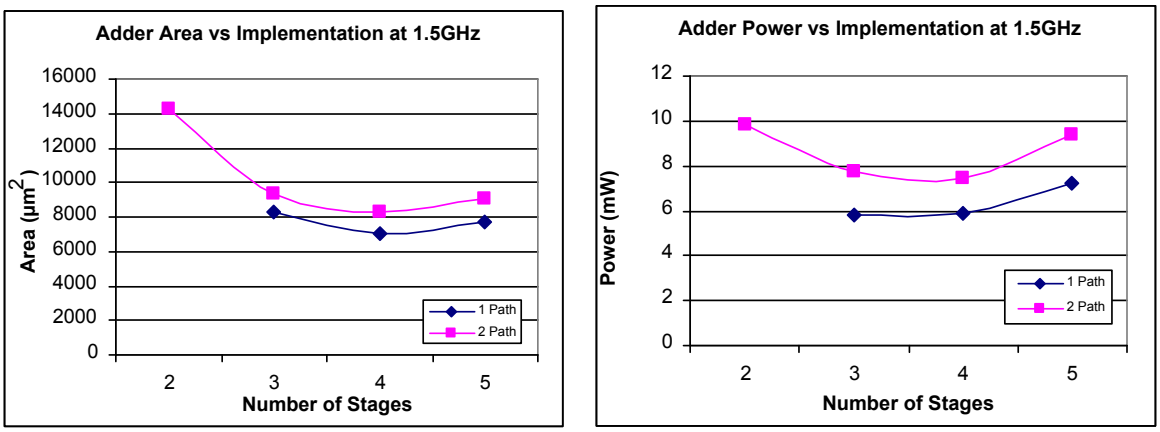
Figure 3.15: A comparison of areas of the Arithmatica-generated single path adder/subtracter for two rounding modes at 1.2 GHz.

Figure 3.16(a) shows the area comparison for the dual and single path adders at 1.5 GHz. At this frequency the dual path design meets timing with two stages while the single path design requires at lest three stages. For both designs, four stages yield the smallest area. The higher frequency caused the area to increase by around 10 percent for a pipe with more stages, and by 30 percent for an adder with shorter latency. The power consumption increased by 25 percent for longer latencies and 40 percent for the shorter pipe.

3.4 Multiplier Implementation

As with the adder/subtracter, the results obtained in the DesignWare study show that double precision and denormalized number support in a multiplier are too expensive to implement on an area constrained system like Rigel. The study also shows that the target latency is around two to three stages.

The diagram of the multiplier implementation is shown in Figure 3.17. As described in Section 2.3.1, the datapath consists of two parallel sections, one which adds the exponents



(a) (b)

Figure 3.16: The areas and power consumption of the single precision adder/subtracter implemented in Arithmatica using the single path and dual path methods at 1.5 GHz.

and one which multiplies the mantissas. The exponent addition portion utilizes a simple 8 bit unsigned adder.

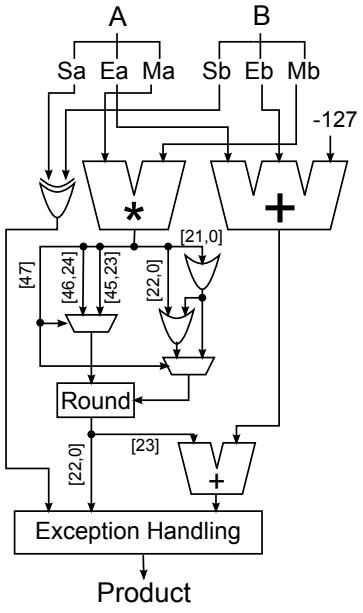
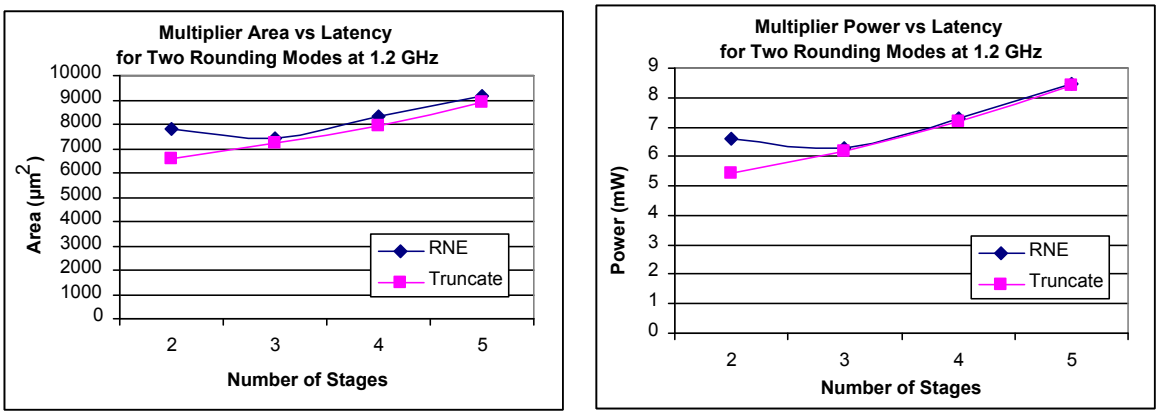


Figure 3.17: Diagram of the floating point multiplier implemented in Arithmatica.

The mantissa multiplier may be implemented using various multiplier architectures. Multiplication in hardware follows a process similar to that of doing multiplication by hand. When two numbers A and B are multiplied together, A is multiplied by each digit in B to generate several partial products. Afterwards, these partial products are added together

to generate the final result [12]. Various techniques were developed to generate the partial products [19] and reduce the amount of summations [20], [21] in order to decrease the latency and area of the hardware. The speed and size of the floating point multiplier unit depends greatly on the performance of the mantissa multiplier. Arithmatica offers several Booth encoded multiplier architectures [12] and was configured to automatically choose the most efficient design.

The multiplier design was simplified by flushing denormalized values to zero. Figure 3.18 shows the areas and estimated power consumption of the synthesized multiplier with RNE mode and with Truncation. As the number of stages increases past three, the areas of both implementations rise as well. This is mostly due to the increased sequential area from adding more latches, as shown in Figure 3.19. The combinational area increases slightly as well, due to fanout of the logic to the increased number of latches and buffers needed to meet hold constraints. A similar trend has been observed in the DesignWare study.

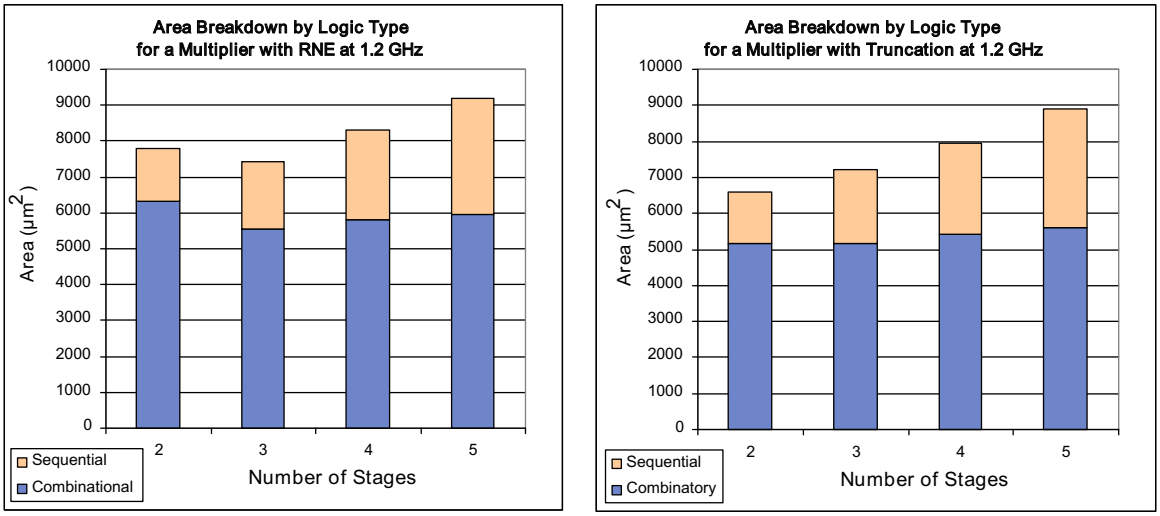


(a) (b)

Figure 3.18: The areas and power consumption of the single precision multiplier implemented in Arithmatica using RNE and Truncate rounding modes at 1.2 GHz.

With two stages, Truncation offers about 15 percent in area savings over RNE mode. At this low latency the timing budget is tight and the additional step associated with RNE has a significant impact on the ability to meet timing; therefore, hardware is upsized. However, with three stages and more, the savings are reduced to about 3 percent. Although the combinational area savings from truncation are about 7 percent at the higher latencies, the sequential logic generated by automatic repipelining takes away from these savings. Although

a two stage multiplier meets timing with both rounding modes, a three stage design offers minimum area for a multiplier with RNE mode.



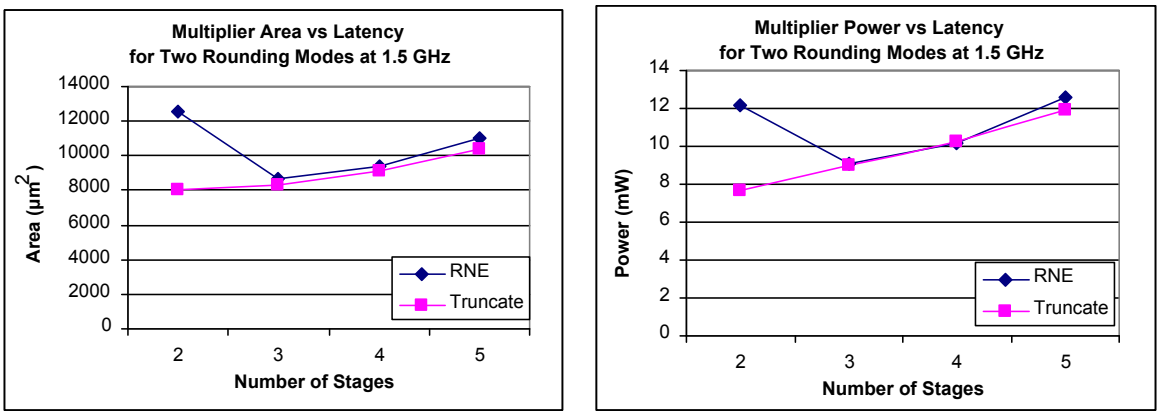
(a) (b)

Figure 3.19: The area breakdown by logic type for the Arithmatica generated multiplier implementations at 1.2 GHz.

Figure 3.20 shows the area and power consumption for the multipliers with different rounding modes at 1.5 GHz. With two stage latency, Truncation offers significant area savings over RNE. However, as the timing is relaxed due to a longer pipeline, the difference between RNE and Truncation becomes insignificant. As was the case at 1.2 GHz, a three stage latency offers the smallest area at 1.5 GHz. The area increased about 15 percent due to the higher frequency, and the power consumption increased by about 46 percent due to the increase in frequency.

3.5 Fused Multiply and Add Implementation

As described in Section 2.3.3 the FMADD operation fuses the multiply and add instructions into one operation. One of the most commonly used cases for the FMADD operation is to accumulate several products together, such as in the dot product. An FMADD unit design which offers high throughput is critical. There are several approaches to achieving high throughput. One is to make the latency of the FMADD very short, so that dependent



(a) (b)

Figure 3.20: The areas and power consumption of the single precision multiplier implemented in Arithmatica using RNE and Truncate rounding modes at 1.5 GHz.

operations can issue back-to-back. The second approach is to split up the single multiply and accumulate chain into several independent accumulations. In the case of a dot product, for example, instead of accumulating all products into the same sum, it is possible to accumulate some products into one sum and accumulate the other products into other sums. At the end of the process all that is needed is to add all of these sums together to generate the final answer.

The latter approach hides the latency of individual FMADD operations so that it is possible to achieve a throughput of one instruction per clock cycle even if the latency of an FMADD operation is longer than one cycle. The drawback to this approach, however, is that it requires that multiple registers be available to store several concurrently running sums. The drawback to the first approach is that it may not always be possible to implement a one cycle FMADD unit at a given frequency.

The IEEE 754 standard [10] specifies that rounding may only be performed at the end of the addition step. This means that the exact product needs to be added. So, for n-bit mantissas, a $2n$-bit product will be generated. Therefore, as mentioned in Section 2.3.2, the adder will need to sum two $2n + 1$-bit operands. This not only increases the size of the adder implemented in Section 3.3, but may increase its latency as well.

The overall back-to-back latency of the FMADD depends mostly on the latency of the adder, since the result of accumulation can be bypassed directly to the adder input in

parallel with multiplication. Several approaches have been proposed to reduce the FMADD latency. Nielsen et al. [22] propose a design based on a redundant number representation [23] with back-to-back latency of as little as two cycles. Other approaches rely on the adder optimizations mentioned in Section 2.3.2. The FMADD unit implemented in Arithmatica utilizes the single path adder design implemented in Section 3.3.

Another issue that needs to be considered is whether the accumulator resides in the general purpose register file or whether a separate accumulator file exists for this purpose. Both implementations have some advantages and disadvantages, and both impact the instruction set architecture (ISA) and microarchitecture of the processor.

If the accumulator resides in the general purpose register file, the *fused accumulator file design*, then other instructions can access its value directly, which is a major advantage. The *separate accumulator file design*, on the other hand, requires that data be moved from the accumulator file to the general purpose register file and vice versa. This requires the ISA to have specific instructions which perform this task. Additionally one must decide which instructions besides the FMADD, if any, have direct access to the accumulator file. The compiler must be aware of all these specifications so that it can generate proper code.

A disadvantage of the fused accumulator file design is that register space may be limited and there may not be enough registers to allocate a portion to be used as accumulators without evicting some values. Additionally, since the FMADD operation reads three operands at a time, in a superscalar design like Rigel register port conflicts may arise. One solution is to increase the number of register file ports, but this is not always an ideal solution since it increases the register file area and access latency. If no additional ports are added, it may not be possible to issue multiple instructions at a time if the non-FMADD instruction reads more than one operand.

On the other hand, with the separate accumulator file design it is possible to dual-issue an FMADD instruction with all other instructions since there are no port conflicts. Additionally, a separate accumulator file offers extra register space, which is advantageous in applications that are register starved.

The FMADD unit implemented in Arithmatica was designed independent of where the accumulator resides. Figure 3.21 shows a top level diagram of the implementation. To

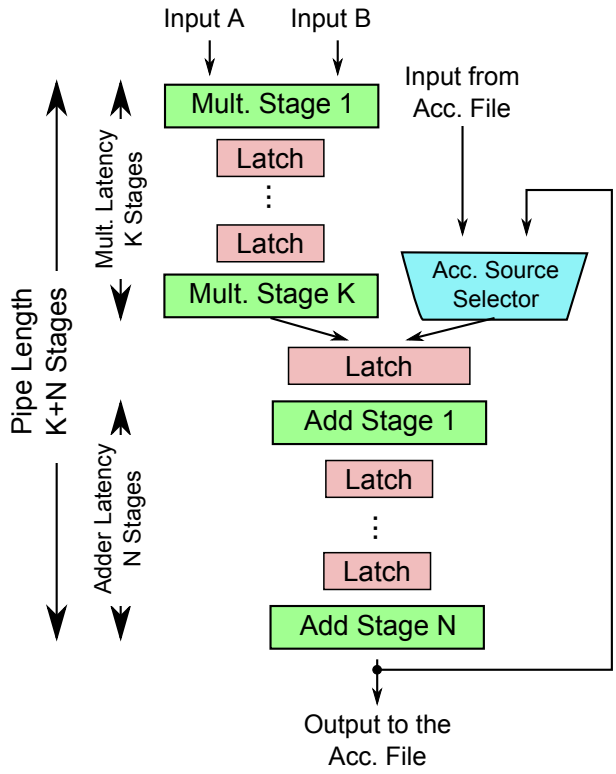


Figure 3.21: A top level diagram of the FMADD implementation. If the overall pipeline has $k + n$ stages, then the multiplier has a latency of k cycles, and the adder has the latency of n cycles. Since the accumulator result is bypassed directly from the adder output, this FMADD implementation yields a n cycle back-to-back latency.

decrease latency, the accumulator input is either bypassed directly from the adder output or obtained from the accumulator file. As mentioned before, if the accumulator result is bypassed, the back-to-back latency of the FMADD operation becomes the function of the adder latency. It is important to note that the exact latency of the multiplication phase is not known since Arithmatica repipelined the data path to meet timing, and thus some multiplication logic may have been fused into the addition phase.

Two overall pipeline latencies were modeled: four and six stages. This allowed me to investigate two and three cycle back-to-back latencies using a four stage pipeline, and four and five cycle latencies using the six stage pipeline. In the implementation, denormalized numbers were not supported and were flushed to zero. Additionally, RNE and Truncate modes were separately implemented to investigate their impact on area and power.

The areas, including a breakdown by logic type, and the power consumptions of the

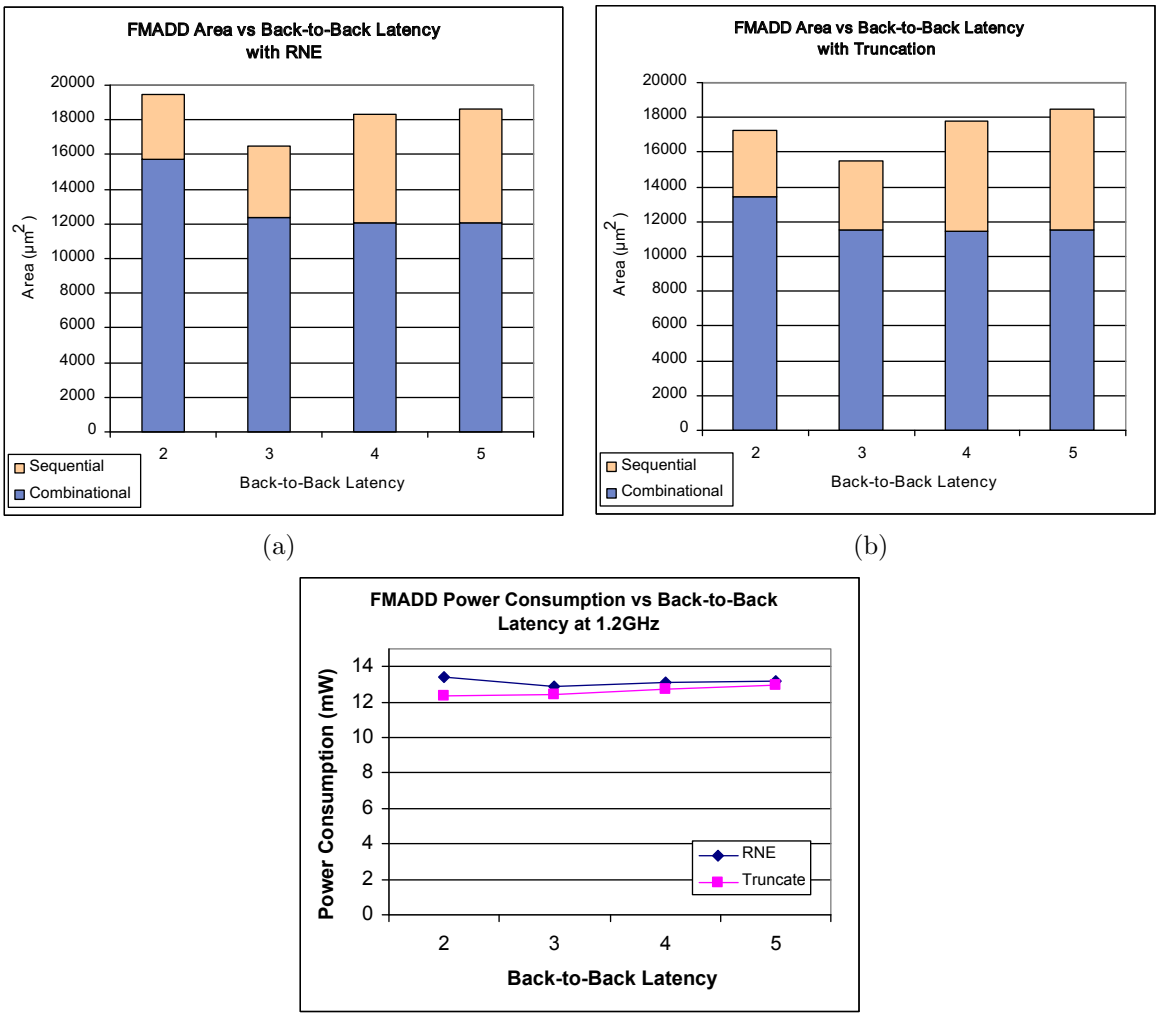
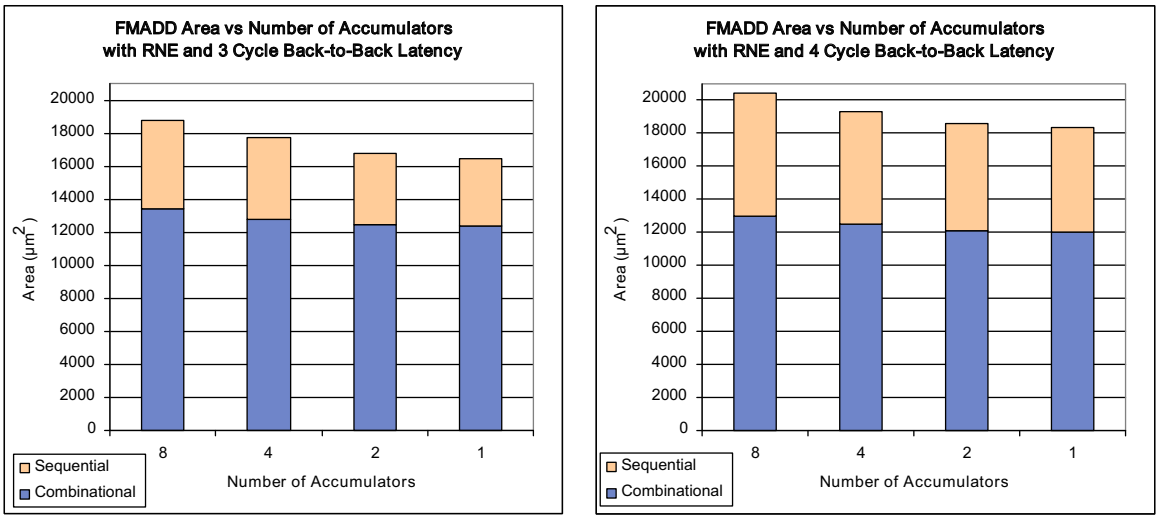


Figure 3.22: Area breakdown by logic type and power consumption for the FMADD unit for two rounding modes with varying back-to-back latencies at 1.2 GHz.

FMADD unit for varying back-to-back latencies and separate rounding modes are shown in Figure 3.22. For RNE the combinational area reaches a minimum with the back-to-back latency of four. The combinational area for Truncation reaches a minimum with three cycle back-to-back latency. The sequential area increases due to the increased number of stages. Consequently, the overall area rises for back-to-back latencies greater than 3 for both rounding modes. Truncation offers about 7 percent in area savings at low latencies; however, at latencies higher than four the area savings are insignificant and are lost in the noise.

To investigate the area and power impact of the separate accumulator file design, a variable



(a) (b)

Figure 3.23: The area breakdown by logic type for the FMADD unit at 1.2 GHz with varying number of accumulators and back-to-back latencies.

sized accumulator file was implemented. The addressability of each accumulator entry is 32 bits. Figure 3.23 shows how the area is impacted by varying the number of accumulators for an FMADD unit with constant back-to-back latency. Figure 3.23(a) shows the area for an FMADD unit with a three cycle back-to-back latency, while Figure 3.23(b) shows a four cycle back-to-back implementation. The sequential area increases as expected due to the additional flip-flops used as accumulators. The combinational area increases slightly due to additional logic such as multiplexors in the accumulator file as well as address comparators at the accumulator source selector.

As mentioned previously, high throughput is very important when considering a design for Rigel. Since it was not possible to reduce latency below two to three cycles, the only way to obtain the high throughput is by providing enough accumulators so that independent instructions can be issued every cycle. Figure 3.24 shows the breakdown of areas by logic type for various back-to-back latencies. However, for each latency, the number of accumulators was set to an amount which would guarantee a throughput of one. Therefore, for two cycle latency, two accumulators were synthesized. For, three and four cycle latency, four accumulators were synthesized. Finally, for five cycle latency, eight accumulators were synthesized.

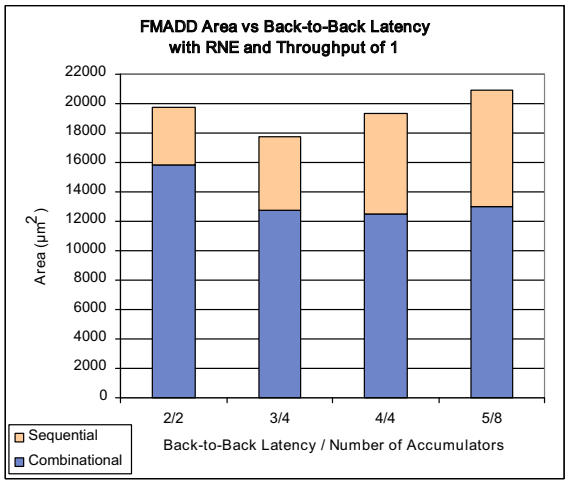


Figure 3.24: The area breakdown by logic type of the FMADD unit at 1.2 GHz with a guaranteed throughput of one for varying back-to-back latencies.

Although the FMADD unit outputs an IEEE Floating Point single precision value and meets timing with two and three cycle back-to-back latency, a different implementation was investigated in case the original does not meet timing after placement and routing. In this implementation additional three status bits, which act as pre-decode flags, are output with each result to indicate whether the result is zero, NaN, or infinity. These bits reduce the latency of the unpacking phase of the adder. The impact on the overall area was insignificant because the combinational area saved from not having to implement an extra comparator was lost in the extra sequential area needed to store the flags.

3.6 Comparator

The comparator was implemented by a method very similar to that outlined in Section 2.3.4. However, in order to save area the comparator only generated the less than (LT) and equal (EQ) flags. The greater than (GT) flag was generated by NORing the LT and EQ flags. At 1.2 GHz the comparator latency is one; therefore, its result may be bypassed to the integer stage in order to resolve conditional branches. The area occupied by the comparator is about 1,640 μm^2.

3.7 Format Conversion

IEEE Floating Point to signed integer (f2i) and signed integer to IEEE Floating Point (f2i) units were also implemented in order to provide a method for fast conversion between the two formats. In order to perform an f2i conversion, the number needs to be unpacked to decode the sign bit (S), exponent (E), and the mantissa (M). Additionally, the proper MSB of M must be generated and eight 0s need to be appended to M to make it a 32-bit number. Next, M must be right-shifted d places, where $d = 158 - Exp$. If $d \leq 0$ the magnitude of the number is greater than 2^{31} and cannot be represented using the integer format. Similarly, if $d \geq 32$ the magnitude of the number is between zero and one. In f2i conversion, Truncation is implemented. Therefore, a floating point value less than one will result in an integer value of zero. For the cases where $d \leq 0$ or if the input is a NaN, the value zero is also output and an invalid operation exception is raised. Additionally, based on the sign bit, the shifted value may need to be converted to 2's complement form of representation. Figure 3.25 shows the diagram of the f2i unit.

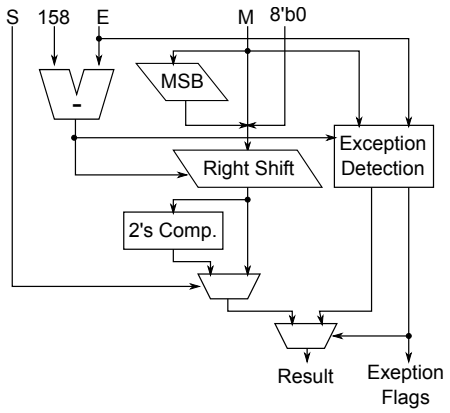


Figure 3.25: IEEE Floating Point to Integer Unit implementation diagram.

Integer to floating point conversion is a similar but reverse process. First the integer value must be converted to sign magnitude form. Next, the most significant one needs to be located so that the mantissa M may be normalized by left-shifting it d places, where d is the number of leading zeros. Then, the least significant seven bits are truncated to normalize the mantissa to 24 bits. The RNE mode is implemented; therefore, the mantissa needs to be properly added. Next, the exponent needs to be generated. This is done by subtracting

d from 158. If the rounding step denormalized the mantissa, the exponent is incremented. Finally, the sign bit will be generated based on the MSB of the input. Figure 3.26 shows the diagram for the unit.

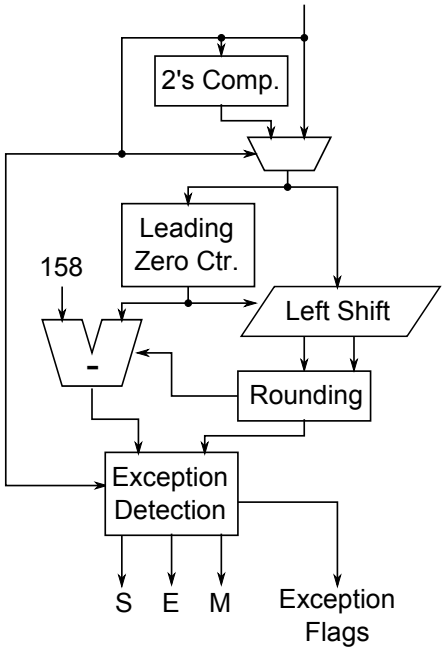


Figure 3.26: Integer to IEEE Floating Point Unit implementation diagram.

Both, f2i and i2f units meet timing with one cycle latency. The Format Conversion unit has an area of 3,510 μm^2; however, its area is reduced to about 3,200 μm^2 when the latency of the i2f unit is increased. Since i2f is not as common an operation, the latency of the i2f unit was set to be the same as the latency of the adder and multiplier to reduce pipeline complexity.

3.8 Implementation Conclusions

The results obtained from synthesizing the individual units give a clearer picture as to the implementation constraints and area and power specifications of the whole FPU. A three cycle latency for the multiplier and adder/subtracter offers the best area/latency trade off. Additionally, a four stage FMADD unit with three cycle back-to-back latency has the smallest footprint. The absolute value, comparator, and f2i units have a latency of one cycle,

and their results can be bypassed for immediate use. Additionally, the i2f unit should have the same latency as the multiplier and adder/subtracter to simplify the datapath.

The diagram of the FPU is shown in Figure 3.27. When synthesized with four accumulators, the area occupied by the FPU is 23,782 μm^2. When synthesized without the FMADD unit, the FPU takes up around 13,100 μm^2. The final area is less than the sum of the individual units because logic reuse was utilized by Arithmatica. Additionally, the individual units include many shared pipeline structures. To add square root and division capabilities it is possible to include the DesignWare Square Root and Reciprocal units. However this was not done as part of this thesis. Additionally, the output from the exception flags needs to be stored in a status register. Since Rigel does not support exceptions or traps in hardware, an instruction must be provided that reads the contents of the exception status register.

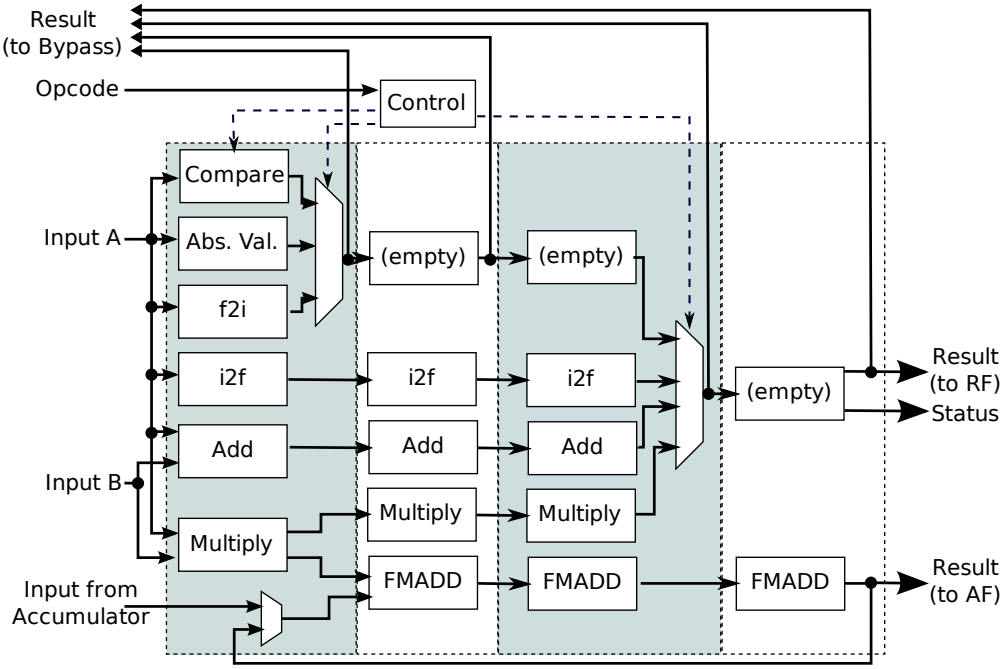


Figure 3.27: An architectural diagram of the proposed FPU design for Rigel.

In the future, it may be beneficial to consider shorter latency designs. Although units with two cycle latency were shown to have significantly larger area than longer latency designs, implementation of fine-grained multithreading may offset these costs due to simplified scheduling and thread swapping logic. Additionally, the investigations in this thesis used the 0.9 V operating voltage which is well below the operating conditions of commercially

available products. A higher operating voltage may yield more attractive area and latency results, although at a cost of increased power consumption.

CHAPTER 4

PERFORMANCE EVALUATION

4.1 Methods

Performance evaluation for the design was done on a cycle accurate simulator of the Rigel architecture. The simulator models all levels of Rigel, from the two-wide core pipeline up to global cache. The fused multiply and add functionality was added to the simulator. Both accumulator file designs were implemented: the separate accumulator file (FMACC), and the fused accumulator file (FMADD) designs. The fused accumulator file design properly modeled register file port contention with four available ports. The scheduler did not dual-issue instructions if a pair of instructions required more than four ports to execute. All benchmarks were simulated on a single cluster. The cache parameters were set as follows: 2 kB 2-way set associative Level 1 data cache with 32 words per line, and one cycle access latency. Level 2 cache was a 65 kB 8-way set associative with access latency of two cycles.

The Rigel Benchmark Suite is composed of several kernels parallelized using the Rigel Task Model (RTM): A 512x512 blocked dense-matrix multiply (DMM); Sobel edge detection filter (Sobel); A 13x13 Gaussian filter (Convolve); Scaled vector addition (SVA); and a medical image reconstruction kernel (MRI) [24]. For a detailed description of these benchmarks refer to [4] and [5].

All of the benchmarks are written in C with RTM extensions. They are compiled using the LLVM compiler into Rigel assembly. A GNU Binutils toolset is used to assemble the code into Rigel executable binary. Because the compiler does not support the FMACC implementation, the assembly was hand optimized to implement it properly. Performance is based on the execution length of the critical loops (in cycles).

This chapter presents the results of performance analysis using the Rigel benchmarks.

Initial analysis was performed to develop a picture of the theoretical performance improvements offered by the fused multiply and add instruction. The results of the analysis are presented in the next section, and the results from the benchmarks are presented in the following sections.

4.2 Fused Multiply and Add Performance Analysis

As mentioned in Section 2.3.3, an important benefit of the operation is that it increases the instruction issue bandwidth since a single operation may be used to perform the function of two separate operations. However, the resulting speedup depends greatly on the algorithm of the code in question, operand availability due to register allocation and cache misses, as well as other system factors. This section evaluates the performance benefits of the FMADD operation for several common scenarios on two machines. The first machine models an architecture capable of issuing up to one instruction per cycle. The second is a superscalar machine capable of issuing up to two instructions from different functional units per cycle, and is used to model the Rigel architecture.

The most basic multiply and add pair calculates $X = A + B * C$. The separate multiply and add implementation of this computation (Figure 4.1(a)) will take at least two cycles to execute on either machine, while the FMADD implementation (Figure 4.1(b)) will take at least one cycle to execute on either machine, thus yielding a maximum speedup of two.

```
mul d, b, c
add x, d, a
```

(a)

```
fmadd x, b, c, a
```

(b)

Figure 4.1: The most basic multiply and add pair using separate multiply and add operations (a), and using a single fused multiply-add operation (b)

However, this implementation assumes that the operands A, B, and C may be accessed directly by the operations. Usually operands must first be placed in registers by loading them from memory. Figure 4.2(a) shows a more common implementation which assumes A is already stored in a register, while B and C must be loaded from memory. Assuming an ideal system where the loads take one cycle each, the multiply and add version would take

at least four cycles on either machine, while the FMADD implementation shown in Figure 4.2(b) would take three cycles. For both machines, if the latency of the multiply, add, and FMADD instructions is one, the speedup decreases to 1.33. Under non-deal conditions, if B or C are not in the cache for example, the speedup is lower.

```
load b, location_b
load c, location_c
mul d, b, c
add x, d, a
```

(a)

```
load b, location_b
load c, location_c
fmadd x, b, c, a
```

(b)

Figure 4.2: The more commonly used case for a multiply and add function using separate multiply and add instructions (a), and using a single fused multiply-add instruction (b). Unlike the implementation in Figure 4.2, the operands must first be loaded from memory here.

On many occasions, the number of elements which need to be multiplied and added is significantly greater. When multiplying two $N \times N$ matrices, for example, each element in the resulting matrix is computed using a dot product of two vectors with N elements each. The computation is of the form $X = A_1 * B_1 + A_2 * B_2 + ... + A_N * B_N$ and requires $2N$ loads, N multiplies and $N - 1$ additions. Therefore, the total number of operations for computing the dot product of two N-element vectors using the separate multiply and add approach is $4N - 1$. An FMADD implementation requires $3N$ operations. For the single issue machine, the speedup approaches an asymptotic maximum of 1.33 as the number of elements in the vectors increases. Figure 4.3 illustrates this relationship.

For a dual issue machine like Rigel, the analysis becomes more complex. The separate multiply and add code can be organized as shown in Figure 4.4(a) requiring $3N$ cycles of execution. As shown in Figure 4.4(b), the FMADD implementation can be organized in such a way that the latency is $2.5N$ for even N, and $2.5N + 0.5$ for odd N. Therefore, the speedup is 1.2 for even number of elements, and approaches an asymptotic maximum of 1.2 as N approaches infinity for odd number of elements.

As can be seen from Figure 4.3, the speedup from using the FMADD operation is nowhere near the expected two. This is because for each multiply and add pair, there are two load instructions that exist in both implementations. However, if the algorithm is modified such

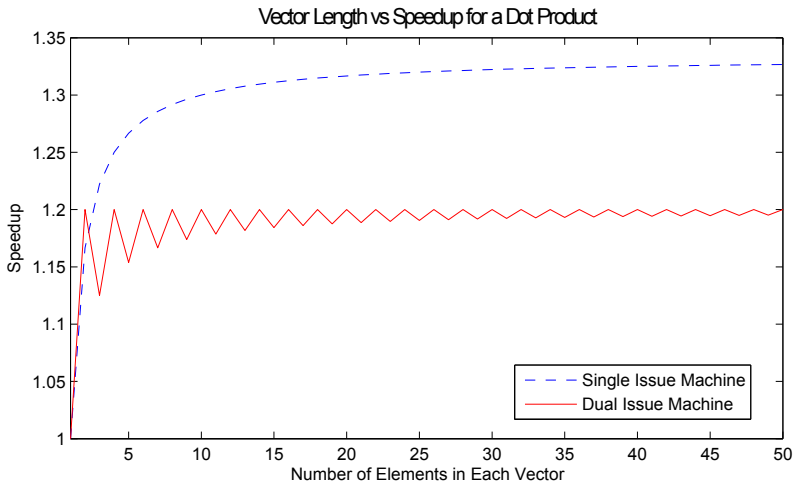


Figure 4.3: Vector length vs. speedup for the dot product code. A comparison between the speedup that FMADD offers on a single issue machine and a dual issue machine.

	SI Cycle	DI Cycle
load a1, location_a1	1	1
load b1, location_b1	2	2
mul temp, a1, b1	3	3
load a2, location_a2	4	3
load b2, location_b2	5	4
mul x, a2, b2	6	5
add x, x, temp	7	6
load a3, location_a3	8	6
load b3, location_b3	9	7
mul temp, a3, b3	10	8
add x, x, temp	11	9
load a4, location_a4	11	9
...	...	...
load bN, location_bN	4N-3	3N-1
mul temp, aN, bN	4N-2	3N-1
add x, x, temp	4N-1	3N

(a)

	SI Cycle	DI Cycle
load a1, location_a1	1	1
load b1, location_b1	2	2
mul x, a1, b1	3	3
load a2, location_a2	4	3
load b2, location_b2	5	4
fmadd x, a2, b2, x	6	5
load a3, location_a3	7	6
load b3, location_b3	8	7
fmadd x, a3, b3, x	9	8
load a4, location_a4	10	8
load b4, location_b4	11	9
fmadd x, a4, b4, x	12	10
...	...	...
load bN, location_bN	3N-1	2.5N-1 (+0.5)*
fmadd x, aN, bN, x	3N	2.5N (+0.5)*

(b)

Figure 4.4: Example code which implements the function: $X = A_1 * B_1 + A_2 * B_2 + ... + A_N * B_N$, implemented using the separate multiply and add instructions (a) and the FMADD instruction (b). The cycle number when each instruction is expected to execute is shown for the single issue machine (SI Cycle), and the dual issue machine (DI Cycle). * For odd N.

that the ratio between the number of loads and multiply-and-add operations is reduced, the speedup will increase. It is possible to do so in situations where multiple operations share

the same operands. For example, when multiplying two matrices, instead of calculating the resulting matrix one element at a time, one can perform several dot products in parallel as shown in Equation (4.1). This way, some values are reused, thus requiring fewer load operations.

$$\begin{aligned} X_0 = &\quad A_0 * B_{00} + A_1 * B_{01} + A_2 * B_{02} + ... + A_N * B_{0N} \\ X_1 = &\quad A_0 * B_{10} + A_1 * B_{11} + A_2 * B_{12} + ... + A_N * B_{1N} \\ &\qquad\qquad ... \\ X_i = &\quad A_0 * B_{i0} + A_1 * B_{i1} + A_2 * B_{i2} + ... + A_N * B_{iN} \end{aligned} \tag{4.1}$$

If this approach is taken, the expected speedup may be computed as follows. Given N elements in each vector, each dot product requires N multiplications, $N-1$ additions, and N load operations. Additionally, N operands need to be loaded only once since they will be reused by all of the dot products. Therefore, given i parallel dot products, the separate multiply and add implementation requires iN multiplications, $i(N-1)$ additions, and $iN+N$ loads, or $(i+1)N + 2iN - i$ operations. On the other hand, the FMADD implementation requires $(i+1)N + iN$ operations. The expected speedup for various values of i and N on a single issue machine is illustrated in Figure 4.5.

For a dual issue machine the analysis is once again more complex because code organization has a big influence on performance. As shown in Figure 4.6(a) for i parallel dot products of N-element vectors, where $i > 1$, the separate multiply and add implementation can be written in such a way that all load operations (except the first three) are overlapped with the multiply and add instructions. In that case the total number of cycles required for the computation is $3 + iN + i(N-1) = 2Ni - i + 3$ cycles. The FMADD implementation, as shown in Figure 4.6(b), requires $N(i+1)$ load operations and iN FMADD operations. Since all but one FMADD instruction can be overlapped with load instructions, the minimum number of cycles this computation requires is $N(i+1)+1$. The speedup for various values of i and N is illustrated in Figure 4.7.

It is important to note that the aforementioned speedup is only possible under ideal

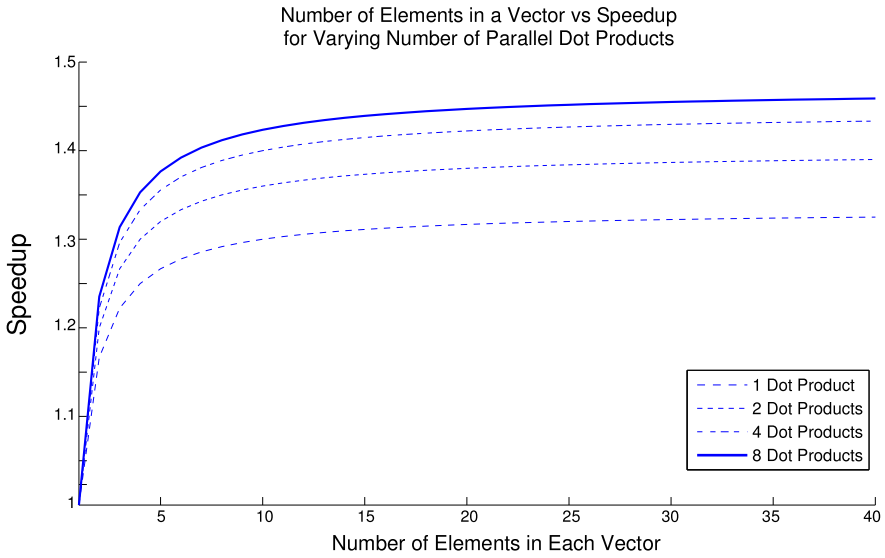


Figure 4.5: A comparison between the speedup that FMADD offers on a single issue machine for computing several dot products in parallel.

conditions - where there are no stalls due to cache misses, for example. Additionally, the code organization shown in Figure 4.6(b) is not always feasible due to register pressure. Under non-ideal conditions, the expected speedup is lower.

4.3 Dense Matrix Multiply

Matrix multiplication can be defined as a series of dot products. To compute matrix C, which is the product of two matrices A and B, for each element C_{yx} in matrix C, a dot product of Row_y in matrix A with Column_x in matrix B is computed. This is illustrated in Figure 4.8(a). Blocked matrix multiplication follows a similar concept, except that instead of computing one dot product on a whole row and column, different processing units calculate a dot product on a portion of a row and column. The individual dot products are then added together to generate the final result as shown in Figure 4.8(b).

DMM attains peak performance by exploiting data locality within the blocks. As shown in Figure 4.9(a), only enough elements of Row_y in matrix A are allocated to fill a cache line. This results in a $1 \times j$ block of cached elements, where j is the cache line size. As each element i in Column_x of matrix B is accessed, j elements of Row_i in matrix B also get cached.

```
                          DI
                         Cycle
load a0, location_a0      1
load b0, location_b0      2

load c0, location_c0      3
nop                       3

load a1, location_a1      4
mul t0, a0, b0            4

load b1, location_b1      5
mul t1, a0, c0            5

load c1, location_c1      6
mul x, a1, b1             6

load a2, location_a2      7
mul y, a1, c1             7

load b2, location_b2      8
add x, t0, x              8

load c2, location_c2      9
add y, t1, y              9

load a3, location_a3     10
mul t0, a2, b2           10

load b3, location_b3     11
mul t1, a2, c2           11
      ...                ...
mul t0, aN, bN        2iN-i+2
mul t1, aN, cN        2iN-i+2

add x, x, t0          2iN-i+3
add y, y, t1          2iN-i+3
```

(a)

```
                          DI
                         Cycle
load a0, location_a0      1
load b0, location_b0      2

load c0, location_c0      3
mul x0, a0, b0            3

load a1, location_a1      4
mul x1, a0, c0            4

load b1, location_b1      5
nop                       5

load c1, location_c1      6
fmadd x0, a1, b1, x0      6

load a2, location_a2      7
fmadd x0, a1, c1, x1      7

load b2, location_b2      8
nop                       8

load c2, location_c2      9
fmadd x0, a2, b2, x0      9

load a3, location_a3     10
fmadd x1, a2, c2, x1     10

load b3, location_b3     11
nop                      11
      ...                ...
load cN, location_ci   N(i+1)
fmadd x0, aN, bN, x0   N(i+1)

fmadd x1, aN, cN, x1 N(i+1)+1
```

(b)

Figure 4.6: Implementation of the parallel dot product in Equation (4.1) implemented using the separate multiply and add instructions (a) and the FMADD instruction (b). The cycle number when each instruction is expected to execute is shown for the dual-issue machine (DI Cycle).

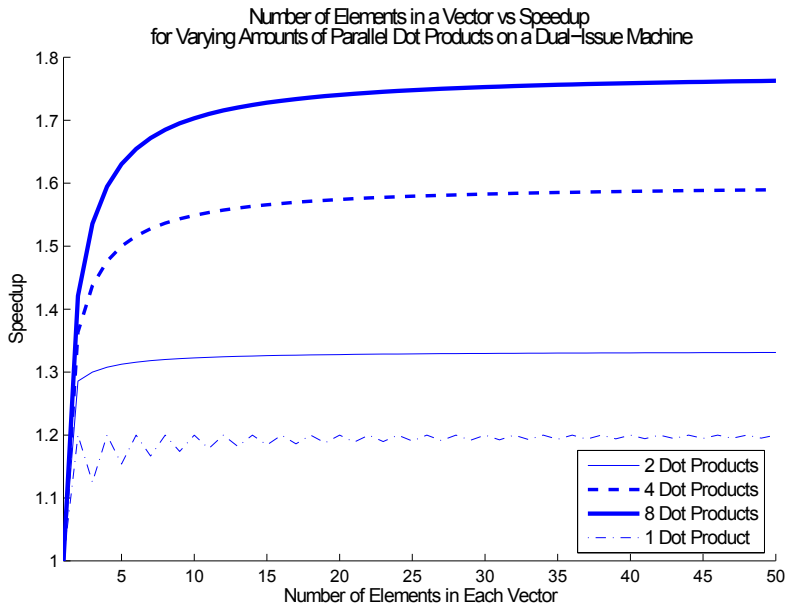


Figure 4.7: A comparison between the speedup that FMADD offers on a dual issue machine for computing several dot products in parallel.

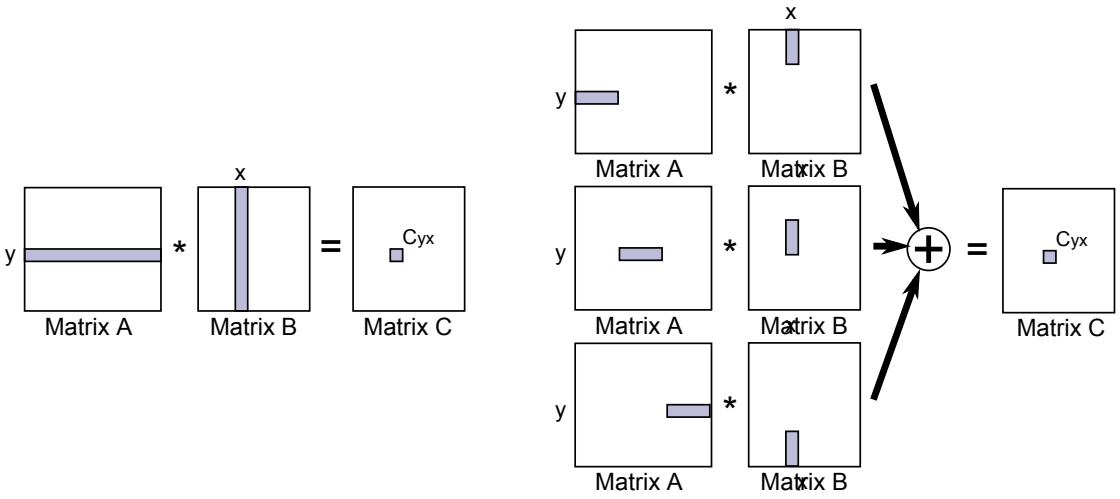


Figure 4.8: An illustration of basic matrix multiplication (a) and blocked matrix multiplication (b).

This generates a $j \times j$ block of cached matrix B elements. Therefore, the partial result of element C_{yx} is a dot product of j elements of Row_y in A and j elements of Column_x in B. Since Column_{x+1} through Column_{x+j-1} of B are cached (assuming no evictions occured), on the next iteration a partial result for element $C_{y,x+1}$ will be computed. After $C_{y,x+j-1}$ is computed, j elements in Row_{y+1} of A are cached, and the process repeats to calculate $C_{y+1,x}$ through $C_{y+1,x+j-1}$.

Instead of calculating the result of one element in C at a time, it is possible to parallelize DMM to calculate $C_{y,x}$ through $C_{y,x+k}$ simultaneously, where k is some integer such that $0 \leq k < j$. However, this requires a large number of free registers to hold all the pointers and values necessary. Given a cache line size of j elements and k parallel dot products, $j+2$ address pointers are needed, $k+1$ data registers are needed to hold the multiplicands, and k data registers are needed to hold the partial dot products. This makes DMM a register limited application. The parallelised matrix multiplication approach is illustrated in Figure 4.9(b).

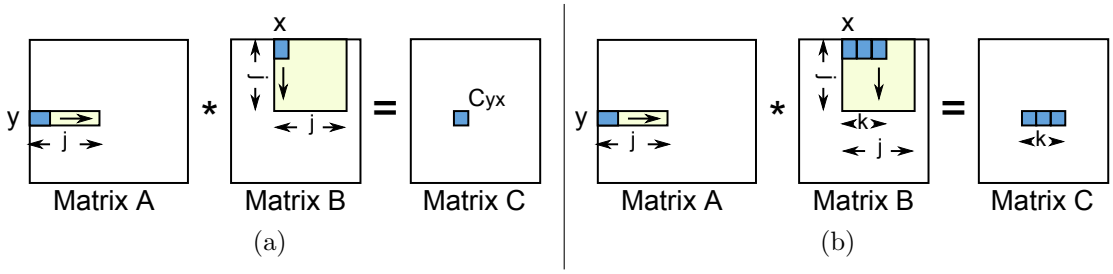
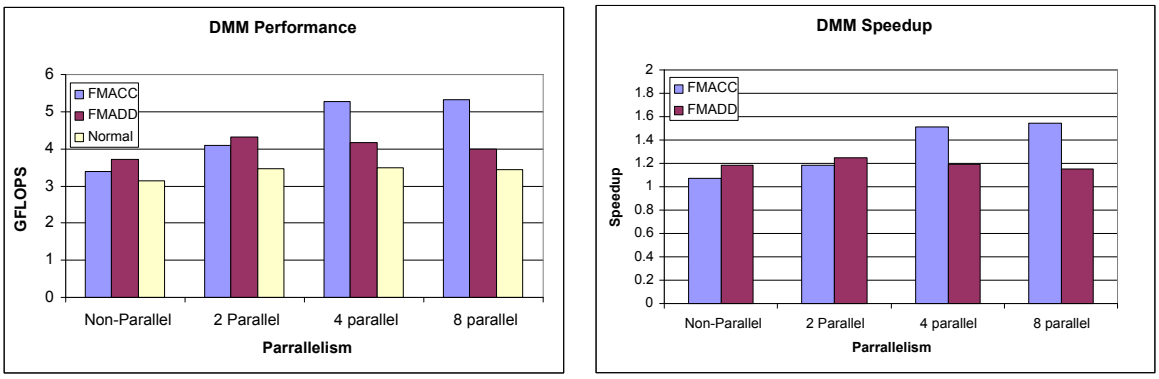


Figure 4.9: An illustration of data locality for a non-parallel implementation (a) and how parallelism exploits this locality (b).

The results from the benchmark for various levels of parallelism are shown in Figure 4.10(a). For the non-parallel implementation, the speedup from the FMADD is around 1.19 while the speedup from FMACC is 1.08. The lower FMACC performance is due to the extra moves required between the general purpose register file and accumulator file. Additionally, the code offers enough room to align the instructions such that no register port conflicts exist in the FMADD implementation.

As the number of parallel dot products increases, so does the speedup from FMACC. As shown in Section 4.2 this is the expected trend. However, the speedup from the FMADD



(a) (b)

Figure 4.10: The performance (a) and speedup (b) for various parallelized versions of the DMM benchmark obtained from the use of the fused multiply and add instruction for the separate accumulator file (FMACC) and fused accumulator file (FMADD) design.

implementation peaks at around 1.24 and then decreases as parallelism increases past two. As shown in the performance graph, the standard and FMADD implementation performance decreases as parallelism increases. This is a consequence of register starvation. From earlier analysis, given eight parallel dot products and a cache line size of eight words, at least 25 registers are needed just to hold all the required information inside the critical loop. This is in addition to registers required to store other constants such as the stack pointer, loop iterators, etc. Since the FMADD implementation uses the general purpose register file, some register values get evicted and pushed onto the stack. Additionally, without extra free registers it is difficult, if not impossible, to schedule the code such that load latencies are hidden.

Referring back to the Expected Speedup analysis from Section 2.3.3, the speedup with the non-parallel and 2-parallel implementations should have been slightly higher than what was obtained in the real benchmark. This occured for two reasons. First, the analysis was done under ideal conditions: the latency of each load was assumed to be one. However, in reality cache misses cause the latency of some loads to be significantly higher. And second, the analysis did not take into account all the code necessary for loop execution and calculating the locations of the pointers. When a perfect cache is modeled, the speedup increases by three to four percent.

Table 4.1 compares the number of lines of code in the critical loops for each implementation to show that the resulting performance is comparable to the expected performance.

Table 4.1: Number of Lines in the Critical Loop of DMM for Varying Parallelism and Implementation

	Non-Parallel	2-Parallel	4-Parallel	8-Parallel
MUL/ADD	39	65	120	262
FMADD	34	51	94	172
FMACC	37	55	96	180

The results from DMM show that there is an added benefit from having extra registers. Additionally, having a unified register and accumulator file improves performance, without having a negative effect on the ability to dual-issue instructions since port conflicts can be avoided with proper code optimization.

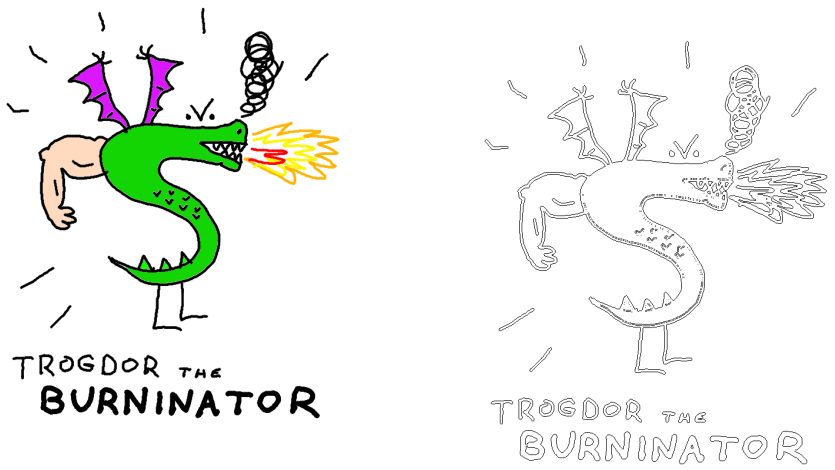
4.4 Sobel Edge Detection

The Sobel benchmark convolves an image with two 3x3 sized filters. Thus, every pixel in the resulting image is calculated using two 9-element dot products. Figure 4.11 shows the effect of the Sobel filter on an image of Trogdor [25].

Figure 4.12(a) illustrates the process of convolution using a single filter; Sobel performs two of these per pixel. As was the case with DMM, the way to maximize performance is to exploit data locality and parallelism. Sobel is easily parallelizable and allows for a lot of data sharing. Figure 4.12(b) shows how to parallelize the Sobel benchmark and shows the implicit data sharing. Instead of solving one resulting pixel at a time, it is possible to perform several convolutions in parallel to obtain values of several resulting pixels at once.

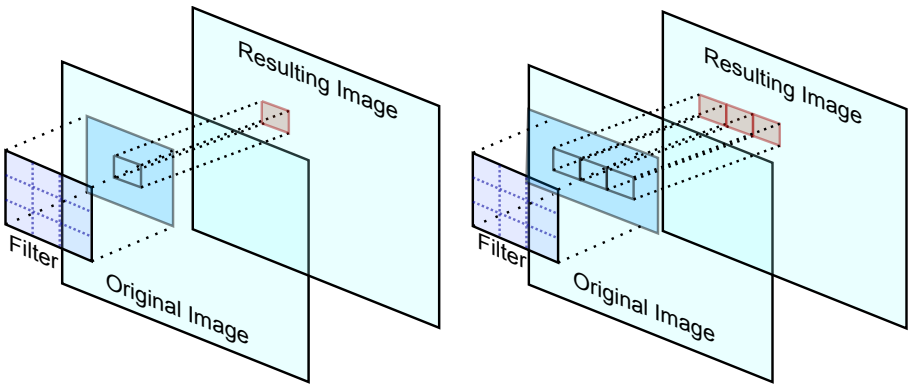
The performance comparisons for the various levels of parallelism are shown in 4.13(a). The speedup obtained from the FMADD and FMACC implementations is almost the same. For the FMACC, the moves between the register file and accumulator file are hidden between other operations. Also, unlike DMM which requires a move to and from the accumulator file on every iteration, these moves are only required every nine iterations in Sobel.

The speedup peaks at around 1.2 with eight parallel dot products. Modeling perfect



(a) (b)

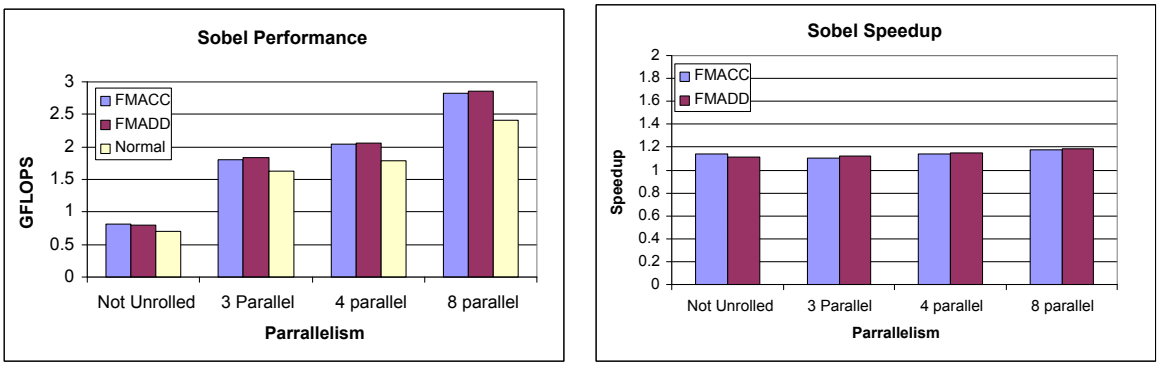
Figure 4.11: An illustration of the edges detected using the Sobel filter on an image of Trogdor [25].



(a) (b)

Figure 4.12: An illustration of convolving a single filter with an image one pixel at a time (a), and several pixels in parallel (b).

caches did not increase the speedup noticeably. This is due to the fact that Sobel shares data accross computations and is therefore not memory limited. The lower than expected speedup is caused by the small number of elements in the 3x3 filter. Sobel does not offer a lot of computational room for a bigger speedup because the ratio of computational instructions to loop control instructions is small. Table 4.2 shows the resulting number of lines of code in the critical loops for each implementation. The increased lines of code for the FMACC implementation is caused by the transfer instructions.



(a) (b)

Figure 4.13: The performance (a) and speedup (b) for various levels of parallelism in Sobel obtained from the use of the fused multiply and add instruction for the separate accumulator file (FMACC) and fused accumulator file (FMADD) design.

Table 4.2: Number of Lines in the Critical Loop of Sobel for Varying Parallelism and Implementation

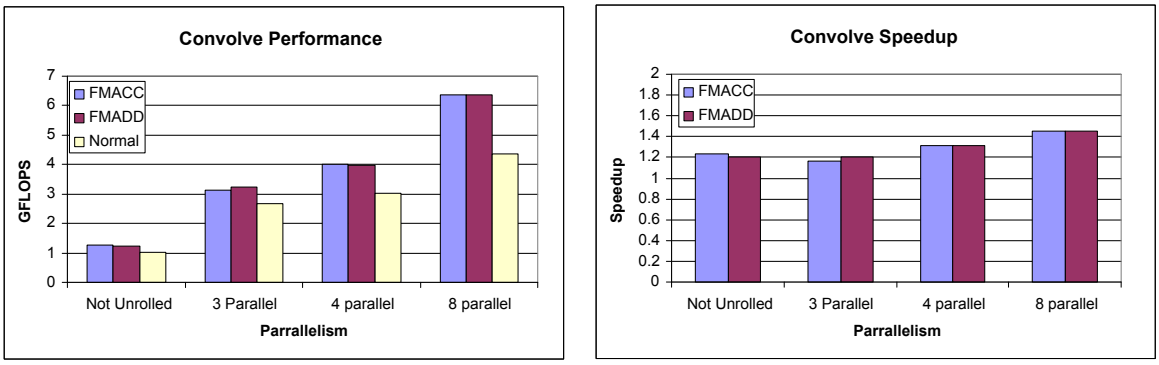
	Non-Parallel	3-Parallel	4-Parallel	8-Parallel
MUL/ADD	32	44	50	74
FMADD	30	38	42	58
FMACC	34	50	58	90

4.5 Convolution

Sobel offered much more data parallelism than DMM, but it did not offer a long enough computational capacity to take advantage of this data parallelism. The Convolve benchmark convolves a 13x13 filter with another 512x24 element matrix. Such filters can be used

to blur an image, for example. Due to the filter size, Convolve offers significantly more computations within the critical loop which, combined with data sharing, offers significant speedup potential. It is parallelized in the same manner as Sobel.

Performance results are shown in Figure 4.14(a) and the resulting lengths of the critical loop are shown in Table 4.3. Like in Sobel, moves between the general purpose register file and the accumulator file in the FMACC implementation are few and can be hidden between other operations. The speedup with eight parallel dot products is around 1.45. Like with Sobel, perfect caches did not increase the speedup. It is important to note that a higher speedup may be possible if the inner loop is unrolled even further.



(a) (b)

Figure 4.14: The performance (a) and speedup (b) for various parallelized versions of the Convolve benchmark obtained from the use of the fused multiply and add instruction for the separate accumulator file (FMACC) and fused accumulator file (FMADD) design.

Table 4.3: Number of Lines in the Critical Loop of Convolve for Varying Parallelism and Implementation

	Non-Parallel	3-Parallel	4-Parallel	8-Parallel
MUL/ADD	15	21	24	36
FMADD	14	18	20	28
FMACC	16	24	28	44

4.6 Scaled Vector Addition

SVA is a benchmark which multiplies each element of two vectors by some factor and then adds the corresponding scaled elements together. In other words, for two vectors A and B, and scaling factors F_0 and F_1, each element i in the resulting vector C would be obtained by performing: $F_0A_i + F_1B_i$. To parallelize SVA, it is possible to unroll the inner loop to find several results in parallel. However, this benchmark does not have any data sharing; therefore, it strides through memory and becomes limited by the lack of data locality. FMACC does not offer any speedup since the moves between the general purpose register file and the accumulator file outweigh any benefit from fusing multiplication with addition. FMADD offers a speedup of around 1.15 when the loop is unrolled by a factor of four, which is a consequence of being memory limited and having no data sharing. When perfect caches are modeled, the speedup increases to around 1.19.

4.7 MRI

The inner loop of MRI is dominated by sine and cosine calculations. These occur through function calls. When only the code within the inner loop of MRI is rewritten using FMADD and FMACC implementations, the speedup is insignificant. However, when the sine and cosine functions are recompiled with FMADD, the speedup is about 1.12. Because the compiler does not support the FMACC implementation, no analysis was done on implementing this functionality within the sine and cosine functions. However, the speedup from the FMACC implementation is expected to be about the same, or slightly lower due to moving data between two register files.

The small speedup is not surprising. The computations within the inner loop do not follow a "dot-product" like pattern which can take advantage of fusing the multiply and add instructions. Unrolling the inner loop in an attempt to exploit parallelism is not advantageous because MRI is register limited, and unrolling by a factor of two causes registers to be evicted to the stack.

4.8 Conclusion

Fusing multiplication with addition has the potential to improve performance. However, due to the general purpose nature of Rigel, the benefits are limited. Even though certain applications see a significantly improved performance, this improvement is nowhere near the ideal speedup. The cost of memory accesses and register file size places limits on performance. Although extending the register space with a separate accumulator file helps in some cases, usually the cost of moving data between two register files hides the benefit of fusing multiplication with addition. For most applications, fusing multiplication with addition will offer some performance improvement.

For an accelerator like Rigel, the metric used to determine the overall benefit of the fused multiply and add unit is FLOPS/mm^2. An FMADD unit adds about 12 percent in core area, and offers between 5 percent and 50 percent performance improvement. Therefore, a fused multiply and add unit offers higher performance per area. Although the separate accumulator file boosts performance by increasing register space, already present compiler support for the fused design makes the latter more attractive. Additionally, increasing the size of the general purpose register file can offset the lack of a separate accumulator file.

REFERENCES

[1] NVIDIA, "NVIDIA GeForce 8800 GPU architecture overview," Tech. Rep. TB-02787-001-v01, November 2006.

[2] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, 2006, pp. 1–8.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 18:1–18:15, August 2008.

[4] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, "Rigel: An architecture and scalable programming interface for a 1000-core accelerator," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 140–151, 2009.

[5] D. R. Johnson, J. H. Kelm, N. C. Crago, M. R. Johnson, W. Tuohy, W. Truty, S. Kofsky, S. S. Lumetta, W.-M. W. Hwu, M. I. Frank, and S. J. Patel, "Rigel: A scalable architecture for 1000+ core accelerators," presented at 2009 Symposium on Application Accelerators in High Performance Computing, Urbana, IL, USA, 2009.

[6] S. Oberman, H. Al-Twaijry, and M. Flynn, "The SNAP project: Design of floating point arithmetic units," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, July 1997, pp. 156–165.

[7] S. Obermann and M. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, August 1997.

[8] S. F. Oberman, "Design issues in high performance floating point arithmetic units," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1997.

[9] P. M. Farmwald, "On the design of high performance digital arithmetic units," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1981.

[10] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, 2008.

[11] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, MA, USA: Morgan Kaufmann, 2010.

[12] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford, UK: Oxford University Press, 2000.

[13] J. Hennessy and D. Patterson, *Computer Architecture - A Quantitative Approach*, 3rd ed., D. Penrose, Ed. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[14] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Leading-zero anticipatory logic for high-speed floating point addition," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 8, pp. 1157–1164, August 1996.

[15] S. F. Oberman and M. J. Flynn, "A variable latency pipelined floating-point adder," Stanford, CA, USA, Tech. Rep. CSL-TR-96-68, 1996.

[16] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced Latency IEEE Floating-Point Standard Adder Architectures," in *ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, 1999, pp. 35–42.

[17] J.-M. Muller, "Some functions computable with a fused-mac," in *ARITH '05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, 2005, pp. 52–58.

[18] *Datapath - Floating Point Overview*, Synopsys, Inc., March 2010. [Online]. Available: https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/fp_overview2.pdf

[19] O. Macsorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, January 1961.

[20] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, February 1964.

[21] K.-Y. Khoo, Z. Yu, and A. N. Willson, Jr., "Improved-booth encoding for low-power multipliers," in *ISCAS '99: Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, vol. 1, July 1999, pp. 62–65.

[22] A. M. Nielsen, D. W. Matula, C. Lyu, and G. Even, "An IEEE compliant floating-point adder that conforms with the Pipelined Packet-Forwarding paradigm," *IEEE Transactions on Computers*, vol. 49, pp. 33–47, 2000.

[23] E. Normale, S. Lyon, M. Daumas, M. Daumas, D. Matula, and D. Matula, "Recoders for partial compression and rounding," Ecole Normale Superieure de Lyon, LIP, Tech. Rep. 97-01, 1997.

[24] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, B. P. Sutton, and Z. P. Liang, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1307–1318, 2008.

[25] H*R web site. [Online]. Available: http://www.homestarrunner.com