The Dissertation Committee for Eric Charles Quinnell certifies that this is the approved version of the following dissertation:

Floating-Point Fused Multiply-Add Architectures

Committee:

_____

Earl E. Swartzlander, Jr., Supervisor

_____

Jacob Abraham

_____

Tony Ambler

_____

Jason Arbaugh

_____

Adnan Aziz

Floating-Point Fused Multiply-Add Architectures

by

Eric Charles Quinnell, B.S.E.E.; M.S.E.E.

Dissertation

Presented to the Faculty of the Graduate School of

the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2007

For my wife
Eres mi vida, mi alma, y mi corazón.

Acknowledgements

This work on the design and implementation of the new floating-point fused multiply-add architectures would not be possible without the knowledge, expertise, and support of the following people:

First and foremost

Leslie K. Quinnell, my wife – for her unwavering patience, understanding, and support throughout the lifetime of this project.

Supervisors

Dr. Earl E. Swartzlander, Jr., The University of Texas at Austin – for his wisdom and unparalleled knowledge in the field of computer arithmetic, as well as for single-handedly convincing a student to pursue a wild new idea.

Carl Lemonds, Advanced Micro Devices – for his vast experience and expertise in the field of x86 floating-point design, as well as his uncanny ability to plainly identify the benefits and flaws of any idea, without which these new architectures would never have been conceived.

The Committee

Dr. Earl E. Swartzlander, Jr.
Dr. Jacob Abraham
Dr. Tony Ambler
Dr. Adnan Aziz
Dr. Jason Arbaugh

Advanced Micro Devices

Carl Lemonds
Dimitri Tan
Albert Danysh
Derek Urbaniak

Legal

Rick Friedman, The University of Texas at Austin
Brian Spross, Advanced Micro Devices
Antony Ng, Dillon & Yudell

Family and Friends

Leslie K. Quinnell
Don and Patricia MacIver
Charlie and Denise Quinnell
Chris and Janet King
Brig General & Mrs. Philip J. Erdle USAF (Ret)
Mrs. Marianne Quinnell
and countless others…

Floating-Point Fused Multiply-Add Architectures

Publication No. _____

Eric Charles Quinnell, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Earl E. Swartzlander, Jr.

This dissertation presents the results of the research, design, and implementations of several new architectures for floating-point fused multiplier-adders used in the x87 units of microprocessors. These new architectures have been designed to provide solutions to the implementation problems found in modern-day fused multiply-add units.

The new three-path fused multiply-add architecture shows a 12% reduction in latency and a 15% reduction in power as compared to a classic fused multiplier-adder. The new bridge fused multiply-add architecture presents a design capable of full performance floating-point addition and floating-point multiplication instructions while still providing the functionality and performance gain of a classic fused multiplier-adder.

Each new architecture presented as well as a collection of modern floating-point arithmetic units that are used for comparison have been designed and implemented using the Advanced Micro Devices (AMD) 65 nanometer silicon on insulator transistor technology and circuit design toolset. All designs use the AMD 'Barcelona' native quad-core standard-cell library as an architectural building block to create and contrast the new architectures in a cutting-edge and realistic industrial technology.

Table of Contents

List of Figures

List of Tables

Chapter 1

An Introduction to the Floating-Point Fused Multiply-Add Unit

This chapter presents a brief introduction to the floating-point fused multiply-add arithmetic unit, its recent spike in interest due to 3D graphics and multimedia demands, and the problems found in its architectural implementation. The chapter finalizes with a short overview of this dissertation's research.

1.1 Introduction

This dissertation presents the results of the research, design, and implementation of several new architectures for floating-point fused multiplier-adders used in the x87 units of microprocessors. These new architectures have been designed to provide solutions to the implementation problems found in modern-day fused multiply-add units, simultaneously increasing their performance and decreasing their power consumption.

Each new architecture, as well as a collection of modern floating-point arithmetic units used as reference designs for comparison, have been designed and implemented using the Advanced Micro Devices (AMD) 65 nanometer silicon on insulator transistor technology and circuit design toolset. All designs use the AMD 'Barcelona' native quad-core standard-cell library as an architectural building block to create and contrast the new architectures in a cutting-edge and realistic industrial technology.

This chapter presents an introduction to the floating-point fused multiply-add architecture, a brief discussion of its implementation benefits and problems, and a

description of the recent spike in its academic and industrial use. The chapter finishes with an overview of the dissertation.

1.2 The Floating-Point Fused Multiply-Add Unit

In 1990, IBM unveiled implementation of a floating-point fused multiply-add arithmetic execution unit on the RISC System 6000 (IBM RS/6000) chip [1], [2]. IBM recognized that several advanced applications, specifically those with dot products, routinely performed a floating-point multiplication, A x B, immediately followed by a floating-point addition, $(A \times B)_{result} + C$, *ad infinitum*. To increase these applications' performances, IBM design engineers created a new unit that merged a floating-point addition and floating-point multiplication into a single hardware block—the floating-point fused multiplier-adder. This floating-point arithmetic unit, seen in Figure 1.2.1, executes the equation (A x B) + C in a single instruction.

The floating-point fused multiply-add unit has several advantages in a floating-point unit design. Not only can a fused multiplier-adder improve the performance of an application that recursively executes a multiplication followed by an addition, but the unit may entirely replace an x87 co-processor's floating-point adder and floating-point multiplier.

A fused multiplier-adder may emulate a floating-point adder and floating-point multiplier by inserting fixed constants into its data path. A floating-point addition is executed by replacing the equation operand B with 1.0, forming the equation (A x 1.0) + C. Likewise, a floating-point multiplication is executed by replacing operand C with 0.0, forming the equation (A x B) + 0.0. This simple injection of constants allows a floating-point fused multiplier-adder to be built as the stand-alone, all-purpose execution unit inside a floating-point co-processor.

However, such advantages do not come without a cost. Although an application may experience increased performance when a program requires multiplications followed by additions, others that require single-instruction additions or single-instruction

multiplications without the cross-over experience a significant decrease in performance. A fused multiply-add unit may be able to emulate a floating-point adder or floating-point multiplier, but the block's additional hardware imposes extra latency on the stand-alone instructions as compared to their original units.

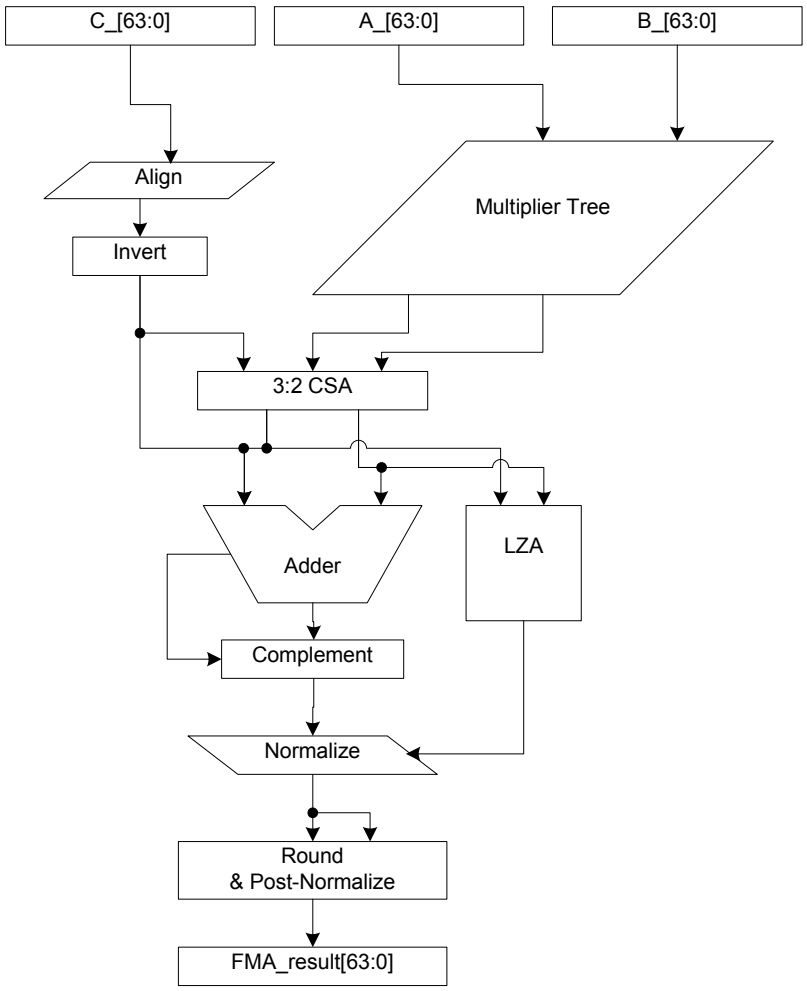

Figure 1.2.1 Simple block diagram of a floating-point fused multiplier-adder

Single instruction latencies are not the only disadvantage to a floating-point fused multiplier-adder. The unit's internal components require bit-widths and interconnectivities commonly more than double that of components found in floating-point adders and floating-point multipliers. With the increasing presence of the parasitic constraints found in designs with massive interconnectivity [3] - [5], the fused multiply-

add architecture is quickly becoming not only a design with difficult timing goals, but also one with heavy power consumption.

The pros and cons of the fused multiplier-adder are well known, and the list of disadvantages has historically driven industry to avoid the unit's use in x87 designs. However, modern-day applications have grown in complexity, requiring a noticeably increased use of the fused multiply-add equation (A x B) + C.

For instance, the fused multiply-add is now used in applications for DSP and graphics processing [6], [7], FFTs [8], FIR filters [6], division [9], and argument reductions [10]. To accommodate this increased use of the fused multiply-add instruction, several commercial processors have implemented embedded fused multiply-add units in their silicon designs. These chips include designs by IBM [1], [11]-[13], HP [14], [15], MIPS [16], ARM [6], and Intel [17], [18].

With the continued demand for 3D graphics, multimedia applications, and new advanced processing algorithms, not to mention the IEEE's consideration of including the fused multiply-add into the 754p standard [19], the performance benefits of the fused multiply-add unit is beginning to out-weigh its drawbacks. Even though the fused multiply-add architecture has troublesome latencies, high power consumption, and a performance degradation with single-instruction execution, it may be fully expected that more and more x87 designs will find floating-point fused multiply-add units in their silicon.

1.3 Overview of the Dissertation

Chapter 2 presents the complete history, advancement, and academic design suggestions of the floating-point fused multiply-add architecture presented by published literature. The chapter begins with the IEEE-754 standard [20] followed by an original description of the IBM RS/6000 unit and finishes with the most recent fused multiply-add publication.

Chapter 3 provides the details of how a circuit is designed and implemented in the AMD 65nm silicon on insulator technology and toolset environment. Following the toolset description, the chapter also presents the results of a custom-implemented floating-point component library which was used by each design in this dissertation.

Chapter 4 presents the design and implementation results of three standard floating-point units created as a reference for comparison. The designs include a floating-point adder, floating-point multiplier, and a modern implementation of a floating-point classic fused multiplier-adder.

Chapter 5 presents the design and implementation of a new floating-point three-path fused multiply-add architecture created to simultaneously increase the performance and reduce the power consumption of a fused multiply-add instruction. The results of the implementation are directly compared to the results from the floating-point classic fused multiply-add unit from Chapter 4.

Chapter 6 presents the design and implementation of a new floating-point bridge fused multiply-add architecture created to allow full-performance executions of single floating-point instructions while still providing the performance benefits of a fused multiply-add architecture. The results of this implementation are directly compared to the results of the floating-point adder, floating-point multiplier, and floating-point classic fused multiplier-adder from Chapter 4.

Chapter 7 summarizes the design results and highlights the benefits and disadvantages of the two new floating-point fused multiply-add architectures. This chapter concludes the dissertation.

Chapter 2

Previous Work on the Floating-Point Fused Multiply-Add Architecture

This chapter provides a description of previous significant works on the floating-point fused multiply-add architecture, including an overview of the original IBM RS/6000.

2.1 Introduction

Several works for the reduction of latency or power consumption in floating-point fused multiply-adders have been published since IBM's original papers on the RS/6000 [1], [2]. This chapter presents the invention and proposed advancements or implementations of the fused multiply-add unit in chronological order. These publications come from both industry circuit implementations and academic architecture proposals.

While not all suggestions for improved fused multiplier-adders have actually been implemented, much of the already completed research provides insight to the variety of complications found in the original architecture. Each paper listed in this section after the original provides a different approach to the design of floating-point fused multiply-add units in an attempt to deviate from the industry-wide acceptance of IBM RS/6000 style architectures. The research presented is fully comprehensive and presents all major advancements to the fused multiply-add architecture to date.

The chapter begins with a brief summary of the IEEE-754 standard [20], followed by the research papers on floating-point fused multiply-add units. Papers that do not present changes to the original fused multiply-add architecture or those that are only industry implementation reports are not described in this chapter.

2.2 The IEEE-754 Floating-Point Standard

The field of floating-point computer arithmetic is a sub-section of computer engineering that concentrates on the development and execution of complex mathematics in modern-day microprocessors. The family of floating-point chips and co-processors are the units in a microprocessor that execute advanced applications such as 3D graphics, multimedia, signal processing, Fourier transforms, and just about every variety of scientific, engineering, and entertainment solutions that require complex mathematics.

The floating-point notation is the electronic world's binary equivalent form of scientific notation for decimal values. A binary floating-point number may represent a vast range of real numbers, including values approaching the infinitely large to those approaching the infinitely small—all in a compact and finite bit-width. In a floating-point number, a selection of binary bits representing an integer fraction hold a numerical position in the real number plane based on another selection of binary bits representing an exponent. The rules that define and govern this floating-point format are enumerated in the Institute of Electrical and Electronics Engineers (IEEE) specification document reference number 754, known as the IEEE-754 floating-point standard [20].

The IEEE-754 standard sets down specific rules and formats for any electronic processing system that uses floating-point arithmetic. The 754 begins by defining precision and exponent parameters for its description of floating-point formats as follows:

p = the number of significant bits (precision)
E_{max} = the maximum exponent
E_{min} = the minimum exponent

It then specifies all binary numeric floating-point numbers to be represented by bits in the following three fields:

1) 1-bit sign s
2) Biased exponent e = E+bias
3) Fraction f = $\cdot b_1 b_2 \ldots b_{p-1}$

where

s = 0 or 1
E = any integer between Emin and Emax, inclusive
b_i = 0 or 1

These bits represent an actual number by listing them in the following form:

$$\text{Number} = (-1)^s \times 2^e \times (1.b_1 b_2 ... b_{p-1})$$

The standard goes on to list the parameter values and their respective names, all listed in Table 2.2.1. For example, a "single precision" floating point format uses 24 bits to represent an implicit one and 23 fraction bits (i.e., 1.*fraction*), 8 bits to represent exponents from the range 2^{+127} to 2^{-126} (approximately 1.7 x 10^{38} and 1.18 x 10^{-38} respectively) and one bit for the sign (i.e., 0 for positive or 1 for negative). The standard lists single, double, and extended precisions as floating-point data type options, each respectively increasing in numerical range and precision.

Table 2.2.1 The IEEE-754 table of formats [20]

	Format			
Parameter	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
E_{max}	127	≥ +1023	1023	≥ +16383
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent *bias*	127	unspecified	1023	unspecified
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

In the standard's data type definition of "single precision," the 23 fraction, 8 exponent, and 1 sign bit may be stored in a single 32-bit register or memory location. The implicit '1' from the fraction is just that, and does not need to be included in the register. "Double precision" is stored in a 64-bit register or memory location. Figure 2.2.1 shows the bit partitioning of the stored binary words.

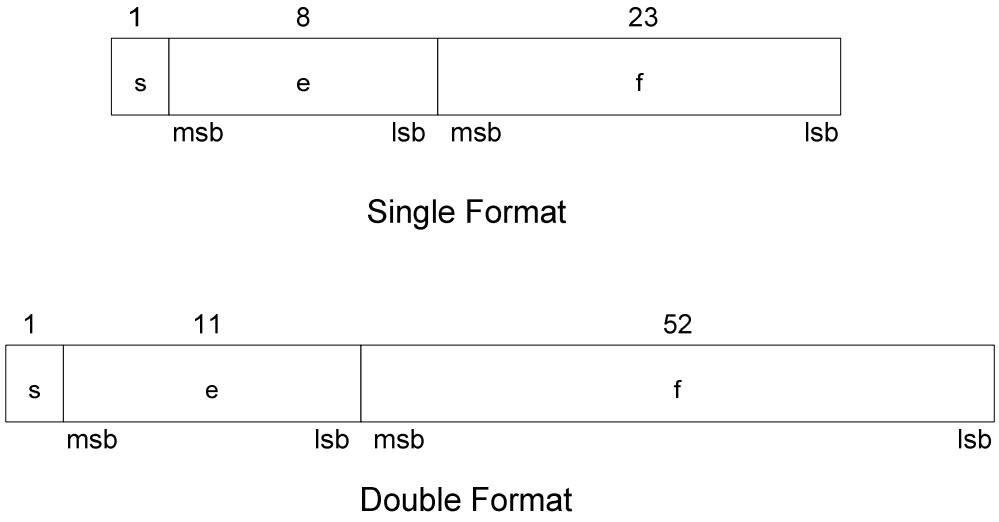

Figure 2.2.1 The IEEE-754 single and double precision floating-point data types [20]

After the enumeration of formats and precisions, the IEEE 754 standard lists a set of requirements for rounding methods. In a floating-point arithmetic calculation, (i.e., a floating-point addition) the result, since it is a fractional representation, may have bits that are too small to include in the format specified. However, leaving them out will alter the correct answer, creating error.

To consider calculation errors, the 754 standard requires any compliant application to support four rounding methods: round to nearest even, round toward plus infinity, round toward minus infinity, and round toward zero. These methods give application programmers the power to determine what kind of rounding error correction is best for their design. In hardware designs conforming to this requirement, generally the rounding stage is at the end of an execution block.

Following the section on rounding, the 754 standard enumerates the required mathematical functions that must be supported in a floating-point machine. The list requires the implementation of the following operations: add; subtract; multiply; divide; square root; remainder; round to integer in floating-point format; convert between

floating-point formats; convert between floating-point and integer; convert binary to decimal; and compare. In the recent meetings of the IEEE-754 committee, currently known as the IEEE 754r, the inclusion of the fused multiply-add function, z = a + b x c, is being considered as an addition to the floating-point operation standard [19].

The formats, rounding, and operation requirements of the IEEE-754 standard listed here are important for the complete understanding of the project described in this and following chapters. All designs, simulations, implementations, as well as most previous publications are in IEEE-754 double precision, 64-bit format and conform to the rounding methods described.

2.3 The IBM RISC System/6000 Fused Multiplier-Adder

In January of 1990, a paper by Robert K. Montoye, Erdem Hokenek, and S. L. Runyon was published in the *IBM Journal of Research and Development* on the design of the RISC System/6000 floating-point execution unit [1]. The journal publication was followed by an *IEEE Journal of Solid-State Circuits* paper published in October 1990 [2]. These papers included details on the invention of the multiply-add fused unit, or what later became more commonly known as the same name rearranged: the fused multiply-add unit, or the multiply-accumulate chained (MAC) unit.

The original paper [1] states that "the most common use of floating-point multiplication is for 'dot-product' operations" and that "a single unit which forms the multiply-accumulation operation D = (A x B) + C would produce a significant reduction in internal busing requirements."

What the paper states is that the equation D = (A x B) + C was at the time a highly-used group of operations in the floating-point unit. The problem with the operation is that a floating-point multiplication and round must first take place before the floating-point addition and round is performed. If a single unit were produced that performs the whole

operation in one block, several benefits could be realized. The original block diagram describing the combination is shown in Figure 2.3.1.

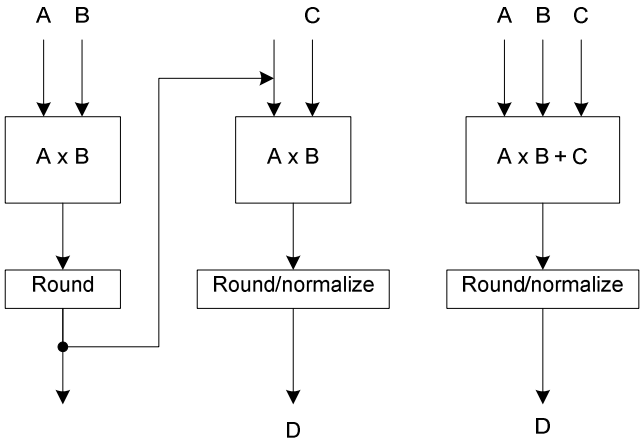

Figure 2.3.1 Block diagram showing the combination of add and multiply (right, redrawn) [1]

The papers go on to describe additional benefits of combining the floating-point adder and floating-point multiplier into a single functional unit. First, the latency for a multiply-add fused mathematical operation is reduced significantly by having an addition combined with a multiplication in hardware. Second, the precision of the final result is increased, since the operands only go through a single rounding stage. Third, there is a decrease in the number of required input/output ports to the register file and their controlling sub-units. Finally, a reduced area of both the floating-point adder and floating-point multiplier may be realized since the adder is only wired to the output connections of the multiplier.

Reference [1] follows these enumerated points with a basic description of the architecture required for a fused multiply-add unit. Since three operands are used in the fused multiply-add operation, the floating-point data must be handled in a range of normalization different than that found in the standard floating-point adder or floating-point multiplier.

In multiplication, the product word size is double the bit-width of the operands. In a floating-point addition, the addition operand (i.e., the addend) is shifted to align the implicit binary point so that the operands are properly aligned when they are added. In an extreme case of addition, the addend's binary point may line up at any position with respect to the product. To cover all of the addition and multiplication cases without losing precision, a fused multiply-add operation must be a full three times the size of the original operator bit widths to correctly normalize the data. Figure 2.3.2 shows the required alignment data-range.

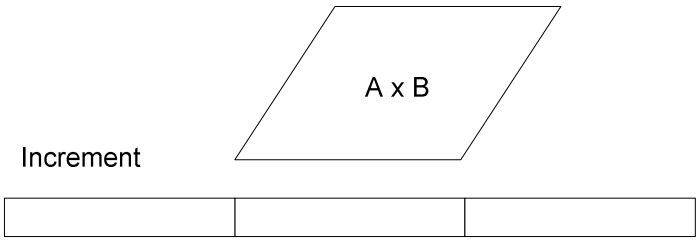

Figure 2.3.2 Alignment range for the 3rd operand in a multiply-add fused operation (redrawn) [1]

Reference [2] describes this normalization problem as one of the tradeoffs required in a fused multiply-add unit. The large bit range requires a very large shifter which can add a significant latency to the operation. The solution presented is the pre-alignment of the addend in parallel with the multiplication.

The second challenge in combining a floating-point multiplication with a floating-point addition comes again from the large precision range. To perform an addition in double precision format, a fused multiply-add unit requires a 161-bit adder. This creates a very difficult timing arc in implementation.

For the adder's case of massive cancellation, which is a case when lots of leading '0's result from a subtraction of nearly identical operands, the leading '1' must be found and the data normalized. To do this, a leading-zero detector (LZD) is included in the fused multiply-add unit and runs in parallel with the massive adder. The LZD is built to find the

first ‘1’ position in the result before the addition is completed, allowing a pre-calculated normalization control to immediately shift the output data from the adder.

Paper [2] presents the original fused multiply-add architecture as built on the IBM RS/6000 shown in Figure 2.3.3. The IBM RS/6000 was implemented in 1μm CMOS silicon technology at IBM.

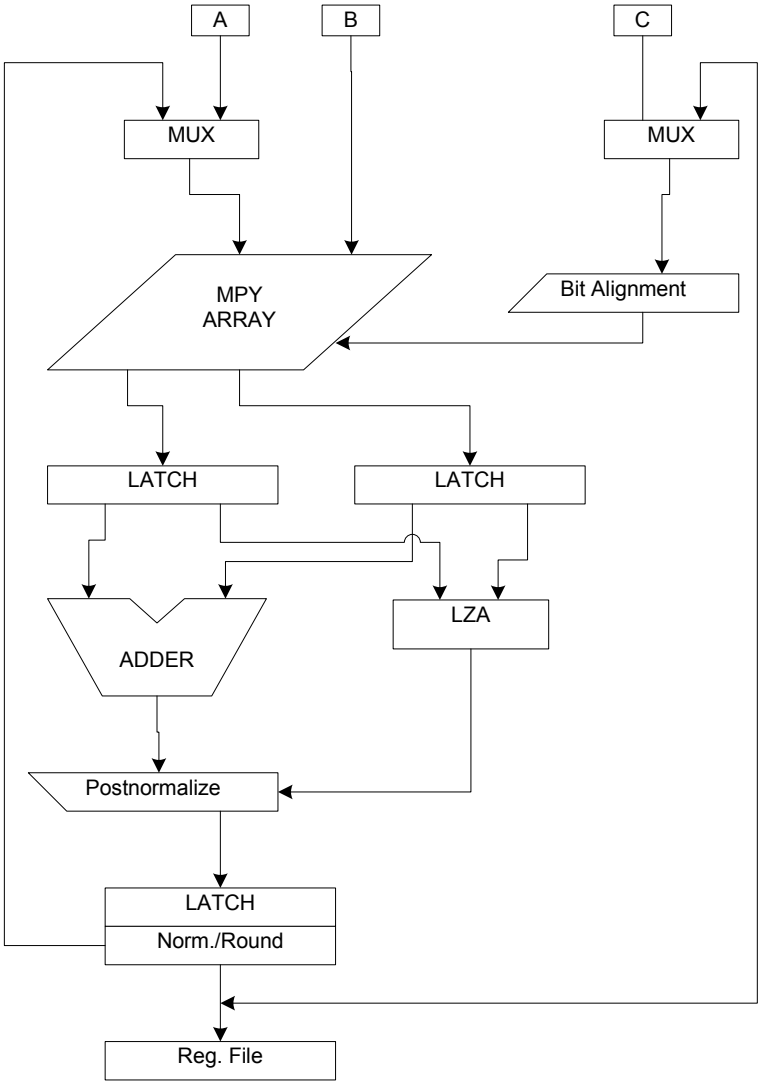

Figure 2.3.3 Original fused multiply-add unit (redrawn) [2]

2.4 The PowerPC 603e and Dual-Pass Fused Multiplier-Adder

The first papers of note after the RS/6000 on fused multiply-add units are [12] and [13]. Both are on the same subject and contribute the same ideas, just in different detail.

Paper [12] is on the implementation of the PowerPC 603e microprocessor. This paper provides three main contributions for the fused multiply-add architecture: first, the paper provides far better detail and descriptions of the original IBM RS/6000 architecture with slight improvements; second, it uses a dual-pass iterative technique in the multiplier to reduce the area and power consumption of the overall fused multiply-add unit; third, the adder/complement stage uses an iterative dual-pass end around carry (EAC) adder that reduces the overall adder size by replacing bit ranges with incrementors. The PowerPC 603 fused multiply-add architecture is shown in Figure 2.4.1.

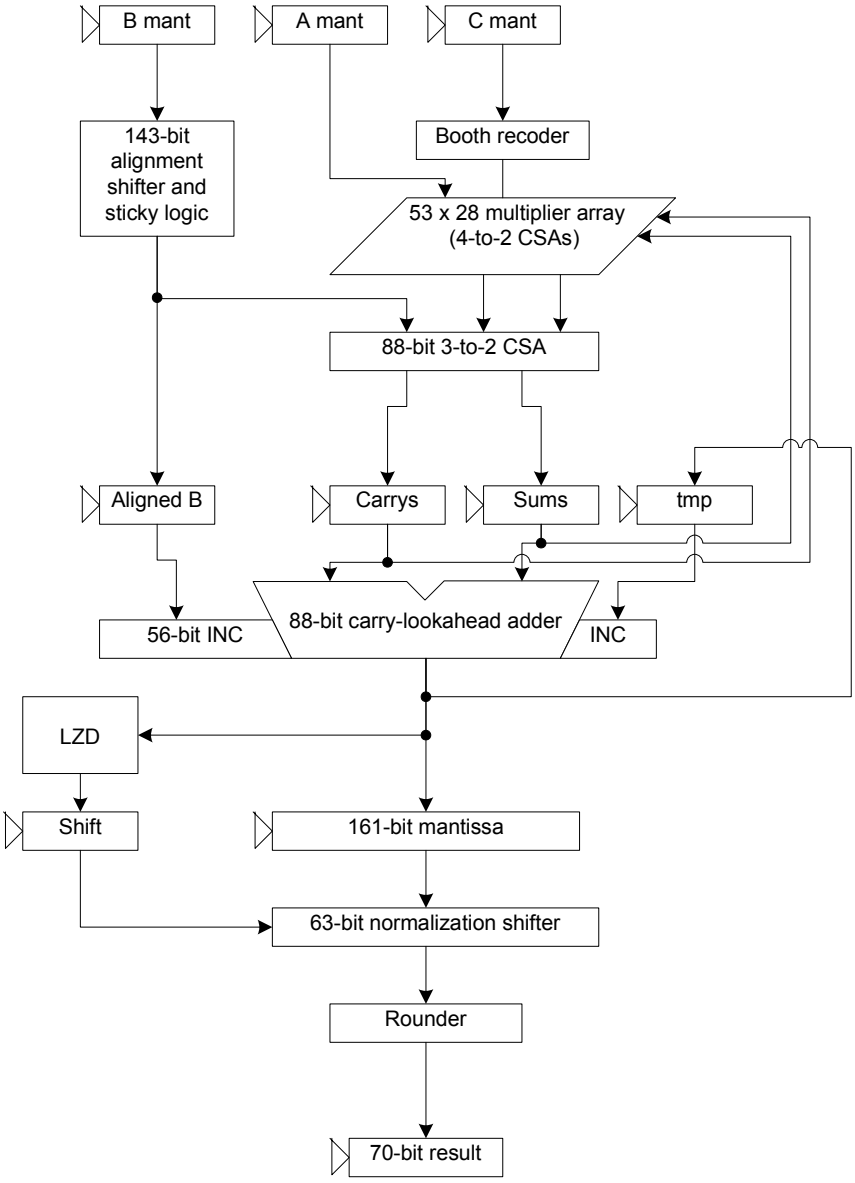

Figure 2.4.1 Block diagram of the PowerPC 603e fused multiply-add unit (redrawn) [12]

The fused multiply-add architecture presented is in essence the same as the IBM RS/6000. First, a large addend aligner that runs in parallel to the multiplier array meets up with the product data in a 3:2 carry-save adder (CSA). The operands are added together in a 161-bit adder while an LZD calculates the shift amount for the normalization shifter. The result is then rounded and passed out of the block.

Although the general architecture is the same as that of the IBM RS/6000, the PowerPC 603e uses far less area and power in exchange for additional cycles. Specifically, each double-precision fused multiply-add instruction must pass through the multiplier block twice before a correct carry save product is calculated.

The details of the iterative components are described thoroughly in [13]. These components are specifically designed to accelerate the performance of single-precision data types. The iterative multiplier scheme reduces the logic in the CSA critical path by a full half. While this makes the tree too small for double-precision numbers, a single-precision instruction is able to produce a product in one multiplier cycle.

To implement this scheme, both the adder and multiplier have the ability to hold for a second cycle during a double-precision fused multiply-add instruction. The iterative multiplier, shown in Figure 2.4.2, adds together partial products like any standard CSA tree for the first cycle. For the second cycle, the partial product result from the first iteration is fed back into the tree and combined with the new partial products.

The dual-pass EAC adder, shown in Figure 2.4.3, performs an addition in every cycle regardless of the data type. In the first pass of a double-precision calculation, a selection of the lower product bits are added together and fed-back to the addition stage. Since many of the low-end partial products are complete in the first iteration, they do not need to be re-combined with the higher-order product from the multiplier second pass. Instead, the bits are passed to an incrementer for the case of EAC 2's complementation.

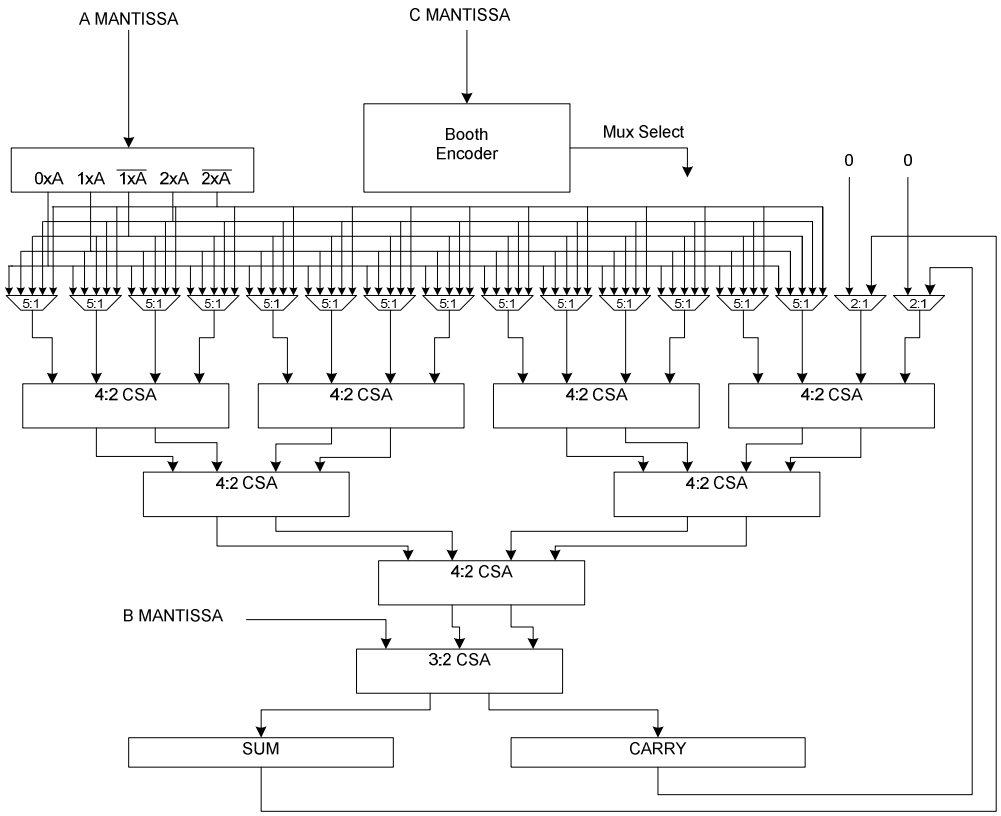

Figure 2.4.2 Iterative Booth radix-4 multiplier CSA tree [13]

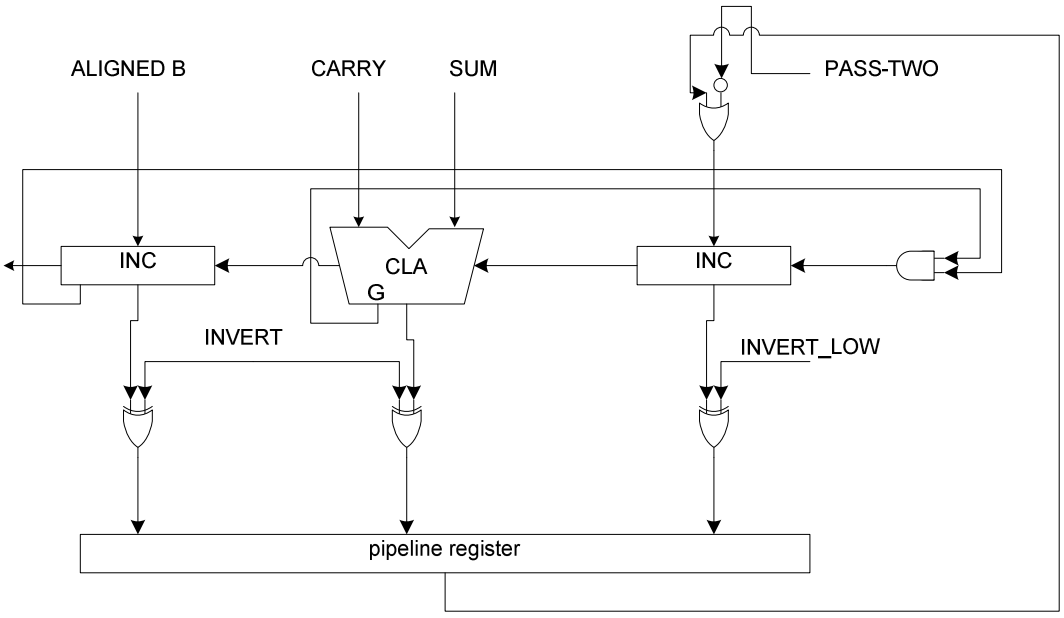

Figure 2.4.3 Dual-pass, iterative EAC adder [13]

In any fused multiply-add case, the high 55-bits of the 161-bit adder input range may only come from the addend and never the multiplier product. Since the product result has a fixed internal bit position, the addend must align with respect to the product. If the addend is much greater than the product, only the top 55-bits of the 161-bit adder result are required. In this case, the product is only used in carry propagation and rounding, so the top 55-bits only come from the addend. If the product is much larger than the addend, the top 55-bits of the 161-bit internal range are all '0's due to the fixed position of the product, so the 161-bit addition result discards the top 55-bits and normalizes the product instead.

Since the design only requires carry propagation in the high 55-bits of the 161-bit adder range, an incrementer is used instead of an adder. The carry-out from this high-end incrementer is passed back to the carry-in of the low-end incrementer, completing the EAC scheme. In total, the 161-bit CPA from the RS/6000 fused multiply-add architecture is reduced in the 603e to an 88-bit EAC CPA with incrementers on each side.

To finalize, the iterative, dual-pass fused multiply-add architecture provides lower single-precision latency, as well as a large reduction in area and power by its creative implementation of the adder and multiplier array. The cost of these gains comes from the taxation of double-precision operations with an extra cycle. These data types can therefore only get half the throughput of single-precision instruction vectors.

The floating-point fused multiply-add architecture described in [12] and [13] has been physically implemented on the IBM PowerPC 603e floating-point unit in 0.5µm CMOS silicon technology.

2.5 The Pseudo-Fused Multiplier-Adder

Naini, Dhablania, James, and Das Sarma presented a paper in 2001 on the implementation of the HAL SPARC64 [21]. The paper is not specifically about a fused multiply-add unit, but does provide a very interesting idea on an architectural arrangement named the "pseudo-fused multiply-add."

In the implementation of the HAL SPARC64, the FPU architecture provides support for a fused multiply-add instruction via two pseudo-fused multiply-add instructions:

- unfused-floating-point multiply-add (uFMADD)
- unfused-floating-point multiply-subtract (uFMSUB)

The SPARC chip itself has two floating-point execution pipelines that can calculate up to two independent fused multiply-add instructions. The pipelines each include a standard floating-point adder (floating-point adder) and floating-point multiplier (floating-point multiplier) with pseudo-fused multiply-add bus handling. This pseudo-fused multiply-add handler is simply a forwarding bus that takes the result from a pipeline's floating-point multiplier and sends it directly to the pipeline's floating-point adder, bypassing the register file. Although bypass buses are now common place in modern FPUs, the bus presented is specifically for pseudo-fused multiply-add instructions.

The pseudo-fused multiply-add does not combine the hardware of the floating-point multiplier and adder. Instead, each floating-point multiplication performs rounding on the data before forwarding the result to the adder on the reserved fused multiply-add bus. The floating-point adder unit uses a third operand from the register file and adds it to the forwarded result. The final pseudo-fused multiply-add is rounded and sent back to the FPU register file. The dual pipe SPARC FPU architecture is shown in Figure 2.5.1 in its original form.

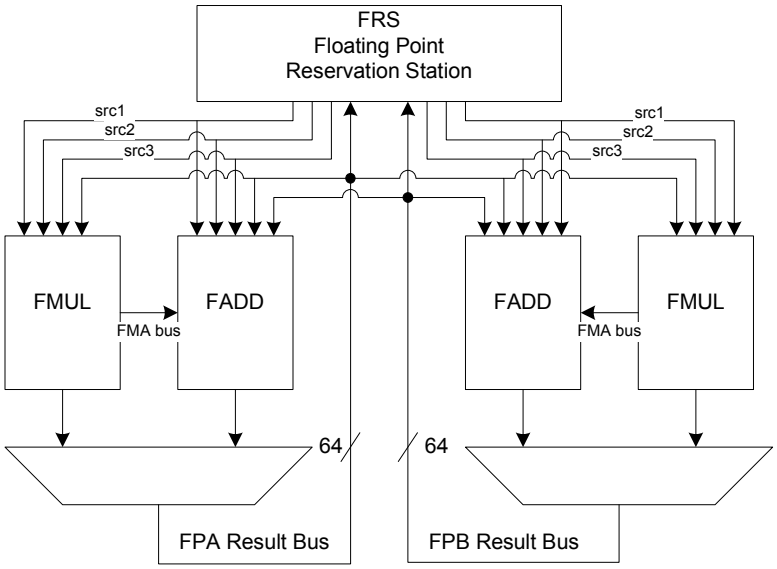

Figure 2.5.1 Dual unit floating-point unit with pseudo-fused multiply-add forwarding buses [21]

The results presented for the SPARC FPU show a latency of 3 cycles for a floating-point addition or floating-point multiplication, and a latency of 4 cycles for a pseudo-fused multiply-add instruction. The HAL SPARC64 has been implemented on 0.15μm CMOS silicon technology.

2.6 Reduced Power Fused Multiplier-Adders

Later in 2001, Pillai, Shah, A. J. Al-Khalili, and D. Al-Khalili presented a paper that compares the IBM RS/6000 architecture with a proposed architecture specifically designed for power reduction [22]. The general philosophy of the architecture is to provide two parallel computation paths (as well as a bypass for floating-point multipliers) that process under different data range assumptions. Early in the pipeline, as soon as the correct path is known via the exponent difference, the other path pipeline is gated and the inputs hold the previous state, saving power.

Figure 2.6.1 (redrawn for clarity) shows the paper's proposed architecture—the Concordia fused multiplier-adder. The Concordia architecture uses alignment blocks before the multiplier array in a move to pre-shift the operands into alignment. This allows

the resulting product terms to be immediately forwarded to two separate paths, each of which may be turned off via pre-calculation. The chosen path which matches the aligned data range of the operands goes on to complete the fused multiply-add instruction. As an added feature, a third partial bypass path is allowed for floating-point multiplier single instructions to have reduced latency.

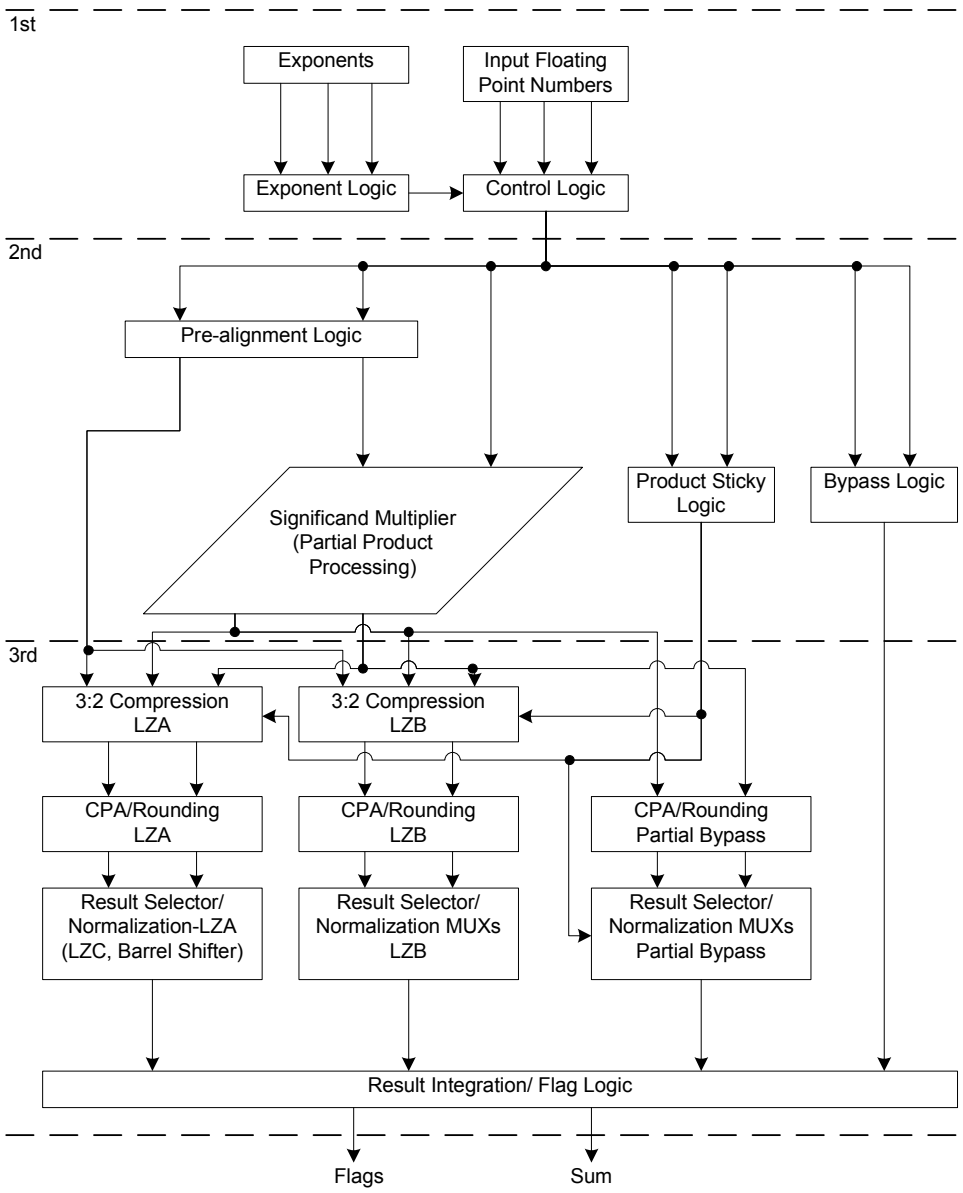

Figure 2.6.1 Concordia fused multiplier-adder (redrawn) [22]

A complication of the Concordia architecture comes from the alignment before the multiplier array. An operand aligned before the multiplier widens the multiplier tree input range, requiring either a larger variable multiplier tree or a loss of precision by parsing lower bits for power savings and latency reduction. The paper's description and arguments for the acceptance of small ulp errors in digital signal processing applications suggest that the Concordia fused multiplier-adder uses the in-accurate multiplier implementation option in pursuit of lower power consumption and latencies.

The paper finalizes by presenting a 44% reduction in power consumption and a 9% latency reduction in the Concordia architecture as compared to the IBM RS/6000 design (re-built on the same technology for comparison). The architecture was synthesized on both 0.35 μm CMOS silicon technology as well as a FPGA and simulated with digital signal processing application data to produce the results.

2.7 A Fused Multiplier-Adder with Reduced Latency

The greatest deviation from the original IBM RS/6000 architecture comes from a paper by T. Lang and J.D. Bruguera on a reduced latency fused multiplier-adder [23]. This proposal claims to achieve a significant increase in fused multiply-add unit performance by the combination of the addition and rounding stage into one block. Although the add/round stage is a widely used component in modern floating-point adder and floating-point multiplier architectures as seen in [24] – [27], its use in a fused multiplier-adder proves to be more difficult.

Lang and Bruguera describe that in order to combine the addition and rounding stages in a fused multiply-add unit, the add/round block must follow the leading-zero anticipator (LZA) normalization stage. Much like a floating-point adder in cases of massive cancellation, the location of the floating-point itself must be determined before any rounding is performed. If the addition and rounding occur simultaneously, then the required compound adder must logically follow the normalization.

The reduced latency fused multiply-add architecture is shown in Figure 2.7.1. In this design, the aligned addend combines with the multiplier product much in the same way as in the IBM RS/6000. However, immediately after the CSA, the data enter two complementary half adder (HA) paths. Sign detection logic determines the correct inversion, selects the correct HA result, and passes the data to the normalization stage.

The correctly inverted data stall at the normalization stage and waits for LZA shift control. In this architecture, the LZA itself is on the critical path. To reduce the time between the multiplier output and the first normalization shift, the authors design a special LZA encoder that produces the control signals on an accelerated path. These signals exit the LZA one bit at a time as they are calculated, as opposed to a standard encoder which selects all the outputs from a parallel multiplexer simultaneously. As each control exits the block, it drives its respective multiplexer normalization.

When the data exit the normalization stage, it is split between a 51-bit compound adder and a 108-bit carry/sticky block. The carry/sticky block creates and passes the rounding information bits to rounding control, which then selects the correct augmented adder output. The data are post-normalized, and the fused multiply-add is complete.

The paper claims an estimated 15-20% reduction in latency as compared to a standard fused multiply-add [23]. This result is calculated theoretically, and the actual architecture has yet to be implemented in either a synthesized or a custom CMOS silicon design.

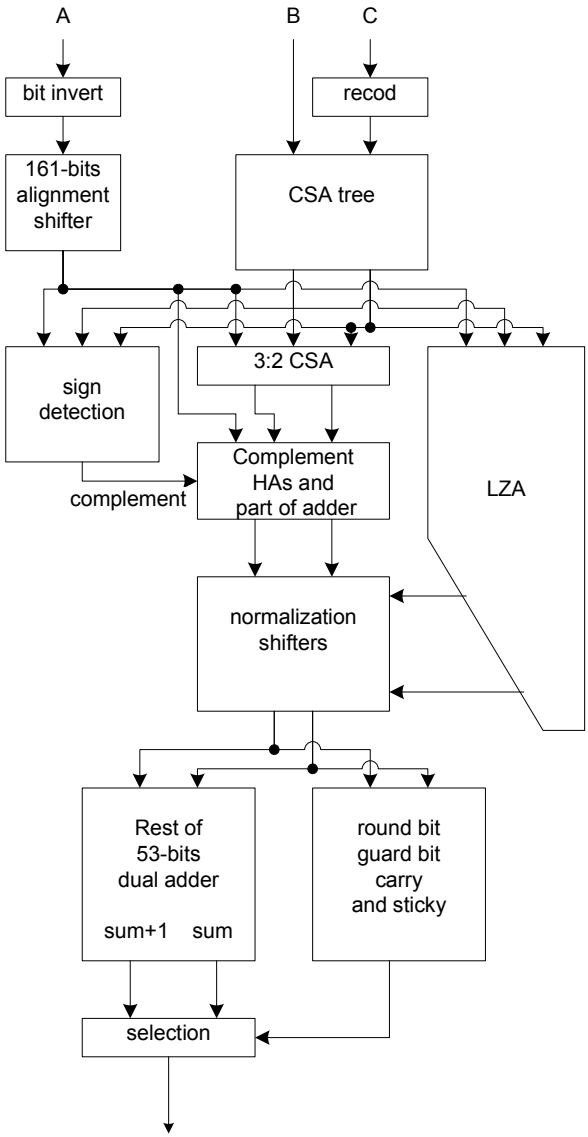

Figure 2.7.1 Lang/Bruguera combined addition/rounding stage fused multiply-add (redrawn) [23]

2.8 Multiple Path Fused Multiplier-Adder

Peter-Michael Seidel wrote a paper in 2003 that proposes a multiple path fused multiply-add architecture to selectively execute on different data ranges for increased performance [28]. Much like the architecture of a common dual-path floating-point adder, the

proposed architecture uses pre-determined data range assumptions that perform different operations on parallel hardware.

Seidel specifically suggests that a fused multiply-add may be split up into 5 distinct cases, all based on the difference in the exponents ($\delta = [A_{exp} + B_{exp}] - C_{exp} + BIAS$):

1. $\delta \leq -54$, where the addend is far greater than the multiplication product. The product only affects the post-normalization, depending on rounding mode.
2. $-54 < \delta \leq -3$, where the addend is greater than the product. The product operands are aligned and added.
3. $-2 \leq \delta \leq 1$, where the product and addend may cause massive cancellation during a subtraction. This case is handled like the close path in a dual-path adder.
4. $2 \leq \delta < 53$, where the product dominates the upper digits of the result. The addend is aligned and added.
5. $53 \leq \delta$, where the product term is much greater than the addend. The addend only affects rounding.

Each fused multiply-add range case listed is seen in Figure 2.8.1. The bit descriptions are all for IEEE double-precision operands.

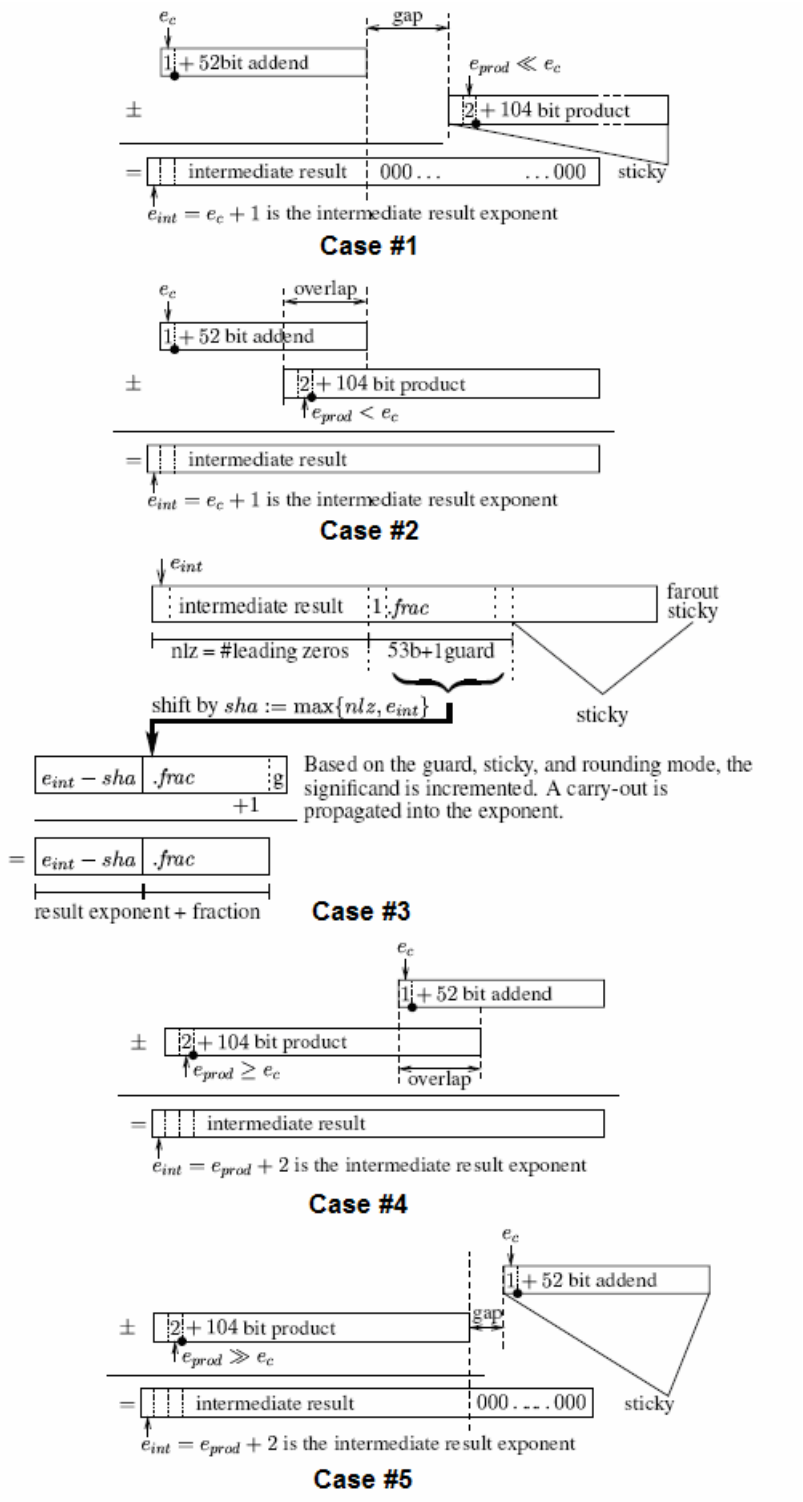

Figure 2.8.1 The fused multiply-add 5 data range possibilities [29]

The paper goes on to suggest an implementation for a fused multiply-add unit that considers the ranges of the five cases as shown in Figure 2.8.2. The implementation uses two parallel hardware paths—one for the far exponent differences and one for the close exponent difference. Much like [22], the far path uses two aligner blocks to selectively shift operands based on the specific data range. For range case 1 and 2, one of the multiplier operands is shifted before entering the multiplier tree. For case 4 and 5, the addend is aligned to the position of the multiplier product.

The hardware suggestion for the implementation of the close exponent difference case performs an alignment on the addend to match the significand product. The data are passed to a combined add and round stage that processes in parallel to a LZA block. The add/round result is complemented if necessary, and normalized by the LZA.
The multiple path fused multiply-add paper claims around a 30% gain in performance as compared to a IBM RS/6000 architecture [28]. These performance gains are estimated based on theoretical calculations.

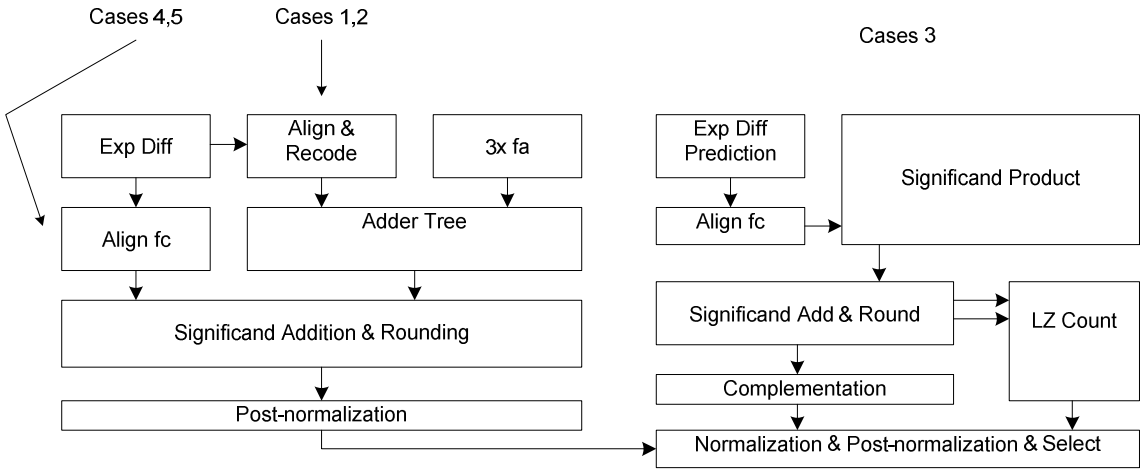

Figure 2.8.2 Suggested implementation for a 5-case fused multiply-add (redrawn) [28]

2.9 3-Input LZA for Fused Multiplier-Adders

In 2005, Xiao-Lu presented a paper [30] for the specific improvement of the critical path found in the Lang and Bruguera fused multiply-add architecture [23]. Specifically, the paper presents a new algorithm for accelerating the LZA stage in the fused multiply-add, since the LZA block is the critical path in Lang and Bruguera's scheme.

Modern architectures design LZA blocks to predict the leading '1' in a massive cancellation subtraction based on the derivation of a set of equations [31]. These leading one's prediction (LOP) equations, as seen in Figure 2.9.1, pass to an encoder which generates normalizing signals correct to within one digit. These equations (f_i) are generated on the assumption that the predicted result consists of two operands.

$$t_i = a_i \otimes b_i$$
$$g_i = a_i \cdot b_i$$
$$z_i = \overline{a_i} \cdot \overline{b_i}$$

$$f_i = t_{i-1} \cdot (g_i \cdot \overline{z_{i+1}} + z_i \cdot \overline{g_{i+1}}) + \overline{t_{i-1}} \cdot (z_i \cdot \overline{z_{i+1}} + g_i \cdot \overline{g_{i+1}})$$

Figure 2.9.1 A two-input LZA algorithm [30]

The Lang/Bruguera fused multiply-add architecture is unique, as it provides three inputs to the LZA block. In the original Lang/Bruguera paper, the three inputs are combined with a 3:2 CSA before entering the LOP unit. The Xiao-Lu paper presents new equations, shown in Figure 2.9.2, that allow this three input string to predict the leading '1'. A three-input LOP removes the requirement for a 3:2 CSA and therefore decreases the number of logic stages in a LZA. A comparison of two- and three-input LZAs is shown in Figure 2.9.3.

$$s_i = \overline{a_i} \cdot \overline{b_i} \cdot c_i$$
$$e_i = \overline{a_i} \cdot \overline{b_i} \cdot \overline{c_i} + (a_i \oplus b_i) \cdot c_i$$
$$t_i = a_i \cdot b_i \cdot \overline{c_i}$$
$$w_i = a_i \oplus b_i \oplus c_i$$
$$x_i = a_i \cdot b_i + (a_i + b_i) \cdot \overline{c_i}$$

$$f_i(pos) = e_i \cdot t_{i+1} + w_i \cdot e_{i+1} + w_{i+1} \cdot x_{i+2}$$
$$f_i(neg) = s_i \cdot \overline{x_{i+1}} + \overline{w_i} \cdot t_{i+1}$$

Figure 2.9.2 A three-input LZA algorithm [30]

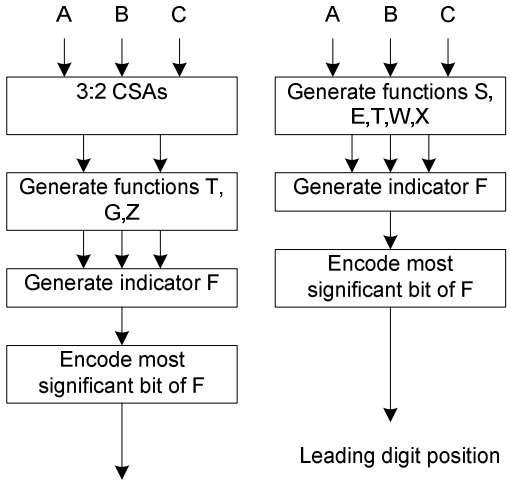

Figure 2.9.3 A comparison of two- and three-input LZA algorithms [30]

Paper [30] claims the three-input LZA scheme shows a 17% reduction in latency and 20% reduction in required area as compared to a two-input scheme. The results come from a Synopsis 0.13 μm synthesis. The reduction in the LZA latency directly improves the critical path delay for a Lang/Bruguera fused multiply-add architecture.

2.10 A Fused Multiplier-Adder with Floating-Point Adder Bypass

The final paper included in this section is a second paper by Lang and Bruguera [32]. The paper describes a fused multiply-add architecture that enhances the functionality of their original proposal for a reduced-latency fused multiply-add unit. While in their original

paper [23] the reduced-latency fused multiplier-adder is designed to accelerate the performance of a fused multiply-add unit, the architecture shares the same disadvantage as the IBM RS/6000 design—the fused multiply-add architecture increases the latency of stand-alone floating-point additions.

The new Lang/Bruguera architecture is designed to allow a floating-point addition instruction to bypass the blocks in the fused multiply-add unit that add to its single-instruction latency. In the first Lang/Bruguera fused multiply-add unit, a floating-point adder instruction has to use the constant '1.0' (A x '1.0' + C) to propagate a multiplier input through the CSA tree, producing an addition operation. Meanwhile, the addend is sent through an aligner unit too large for a floating-point adder range, adding unnecessary latency to the data.

As shown in Figure 2.10.1, the new architecture uses selection multiplexers after the multiplier stage to choose different operands based on the instruction input. Mimicking common floating-point adder designs, an additional "far" path is added to the Lang/Bruguera fused multiply-add scheme for cases where the floating-point addend must still be aligned by a significant amount. This path is processed in parallel to the "close" path, which is a slight deviation from the first Lang/Bruguera fused multiply-add scheme. For floating-point adder data with close exponents, the large LZA and normalization hardware already found in the fused multiply-add path handles the operations correctly. Both paths are merged after normalization and passed to the add/round stage.

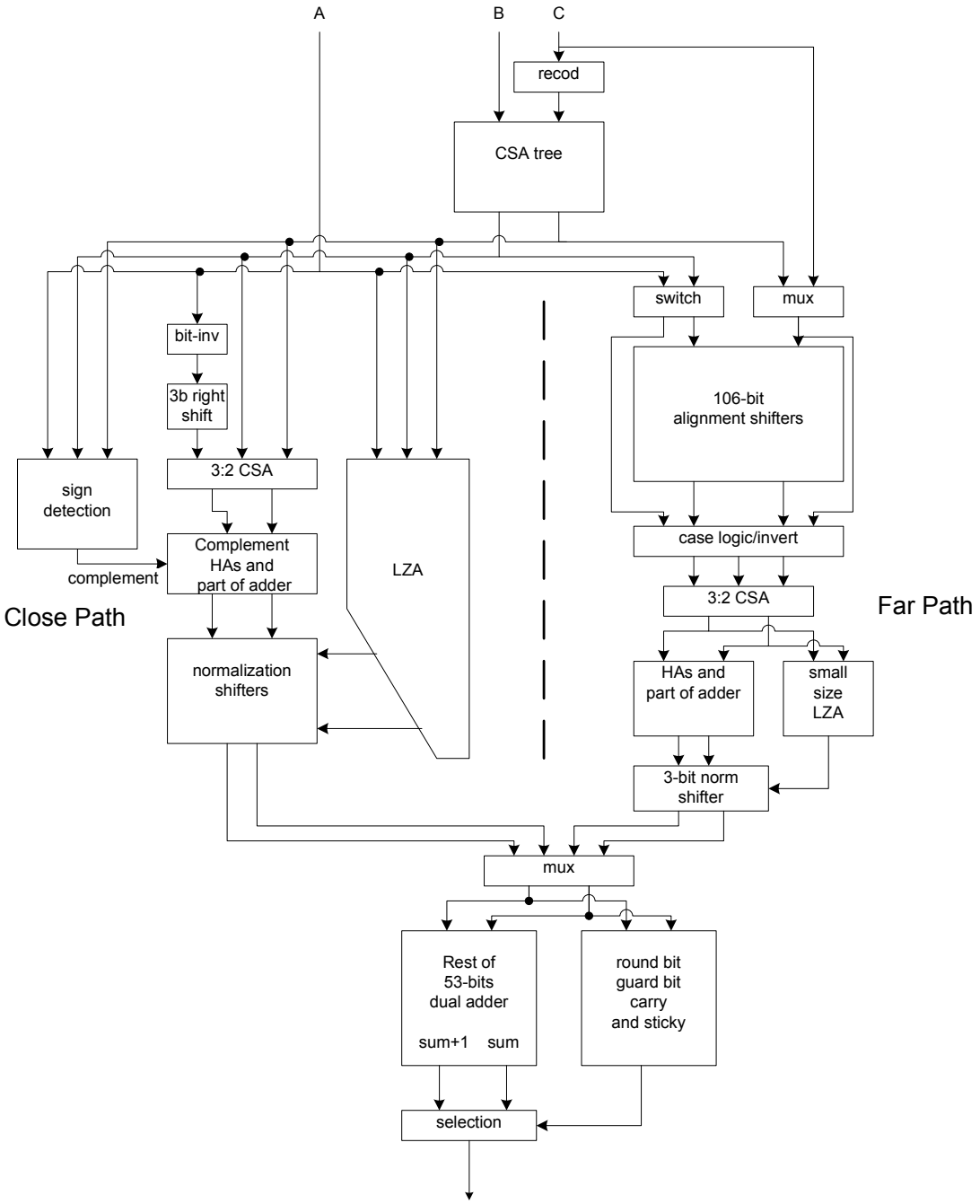

Figure 2.10.1 Lang/Bruguera fused multiply-add with floating-point adder capabilities (redrawn) [32]

However, the new architecture makes changes to how a fused multiply-add instruction is processed. In the design, the multiplier and aligner data from the head of the unit pass to both the far and close paths. The fused multiply-add data in the far path are assumed to

have a large exponent difference, so the use of a dual path parallel inversion is not required. Instead, only a single operand is needed for inversion during a subtraction, and the data may pass to a smaller size LZA. For the fused multiply-add close path, the fused multiply-add scheme follows the algorithm originally provided by Lang and Bruguera with the addition of a 3-bit aligner used in floating-point adder cases. Like the floating-point adder operation, both paths are merged after each normalization. The fused multiply-add data are added, rounded, and post-normalized, completing the instruction.

Much like the original Lang and Bruguera paper, this paper concludes by claiming a 40% acceleration of floating-point adder instructions as compared to an IBM RS/6000 fused multiply-add unit handling the same [32]. Additionally, the fused multiplier-adder provides a 10% reduction in latency compared to the IBM RS/6000. This result is calculated theoretically, and the actual architecture has yet to be implemented in either a synthesized or a custom CMOS silicon design.

The fused multiply-add unit latency reduction is lower than the original Lang and Bruguera improvements due to additional logic stages supporting a single-instruction floating-point addition. The results were calculated by theoretical delay analysis.

2.11 A Comparison of Literature

Table 2.11.1 shows a comparison of the various floating-point fused multiply-add architectures presented in this chapter against the original IBM RS/6000. Each design is compared against the IBM RS/6000 in the categories of latency reduction, power reduction, implementation, numerical accuracy, and whether the unit is capable of a maximum-performance single-instruction execution of a floating-point adder or floating-point multiplier.

Table 2.11.1 Comparison of proposed fused multiply-add architectures

Design	Latency vs RS/6000	Power vs RS/6000	Implemented or Theoretical	Numerically Correct?	Max-performance FPM?	Max-performance FPA?
IBM RS/6000 [1],[2]	N/A	N/A	Implemented	Yes	No	No
IBM PowerPC 604e [12],[13]	Faster SP, Slower DP	½ size Mul tree	Implemented	Yes	No	No
HAL SPARC64 (pseudo FMA) [21]	Slower	N/A	Implemented	Rounded Twice	Yes	Yes
Concordia FMA [22]	-9%	-44%	Implemented	No	Yes	No
Lang/Bruguera [23]	-(15-20%)	N/A	Theoretical	Yes	No	No
Seidel Multi-Path [28]	-30%	N/A	Theoretical	unclear	No	No
Xiao-Lu LZA improvement of Lang/Bruguera [30]	-(15-20%) - (0.17 x LZA)	N/A	LZA Implemented, FMA Theoretical	Yes	No	No
Lang/Bruguera w/ FPA bypass [32]	-10%	N/A	Theoretical	Yes	No	No[†]

[†] 40% faster floating-point add performance as compared to a classic FMA execution of the same

Chapter 3

Methods and Components using AMD 65nm SOI

This chapter begins by detailing the implementation methods and tools used to create a circuit in the AMD 65nm silicon on insulator design flow. Following that, the chapter lists the architectures and implementations of shared floating-point arithmetic components used in several of the final designs.

3.1 Introduction

This chapter provides a detailed description of the methods and components used to design, implement, and test the floating-point fused multiply-add circuits presented in this dissertation. The designs have been implemented using the AMD 65nm silicon on insulator (SOI) transistor models and implementation design flow.

The AMD 65nm SOI circuit design flow used is also known as the AMD "axe" flow. The axe flow is a collection of industry tools and software linked together with AMD transistor libraries and databases organized in such a fashion that the progression of a circuit from RTL to GDSII "flows" through the necessary CAD tools in a logical order. This implementation flow, as well as the RTL tools and compilers that were used for designing the behavioral models, are described in detail in the first half of this chapter.

The second half of this chapter includes detailed descriptions of the floating-point components and libraries built specifically for this dissertation's floating-point fused multiply-add designs. The components, ranging from multiplier arrays and adders to barrel shifters and sticky trees, are shared in a common floating-point library that has been created to keep the components used by the designs consistent in their

implementation. These macro components are all original designs and have not been downloaded from any AMD IP database.

At AMD, the CAD tools, manufacturing models, and standard cell libraries are in a volatile consistently evolving developmental state. All of the models and tools are subjected to frequent, rapid and drastic fundamental changes to meet the demands of whatever AMD project is currently under development. Since this dissertation is intended to compare high-level architectural changes alone, a specific "snapshot" of the standard cells and tools for 65nm SOI development was taken on July 30th of 2006. This flow snapshot uses the most up to date models and libraries as of that specific date, and has ignored any further changes since then to keep the implementations consistent from origin to completion. It should be noted that the axe flow used for this dissertation is now an outdated and retired CAD system at AMD.

3.2.1 Design and Implementation Method Overview

A wide variety of CAD tools are used at AMD to bring a design from concept on paper to GDSII mask data. These tools include both in-house CAD developments as well as externally written design software. Depending on the technology and goals of the design, this arrangement of tools varies in functionality and what models it considers. For the specific fused multiply-add designs considered here, this section describes in detail each step used to take the fused multiply-add architectural concepts to "front-end" design completion.

When considering the design flow at a high-level, the toolsets may be split into three major categories. The first category of CAD systems is the register transfer level (RTL) Verilog code used to describe the architecture at a purely digital level. At this highest-level behavioral model, the circuit is designed and tested for digitally functional correctness, using virtual logic analyzers to debug and adjust the inputs and outputs of the block. The RTL models are also passed into a set of verification procedures during

development that use custom test benches and already proven legacy vectors to ensure the formal verification of the digital model.

The second level of design, commonly called the “front-end” design, is the translation of the digital RTL into a transistor-level description. The transistor-level model is coupled with the manufacturing models to create and simulate the circuit in an analog environment, producing accurate simulation estimates of timing, power, and area. The front-end is considered complete when the model has acceptable results based on a true floor plan and pessimistic Steiner routing parasitic calculations.

The final level of design, called the “back-end” design, is the fine-tuning and physical routing of the circuit. This level uses pre-routing, auto-routers, and routing editors to physically add and adjust the interconnection of the circuit in a model that may “tape-out” to the GDSII manufacturing mask standard. The circuit’s final transistor model is coupled with the routing model and the circuit undergoes a series of fine-grain electrical tests, including electromagnetic simulations, IR calculations, local heating, noise, and a full chip-level analog timing simulation. A model that passes all of the back-end checks, as well as provides acceptable electrical power, timing, and area results, is ready for GDSII tape-out. Any errors or unacceptable results require design iterations at either the back-end, front-end, or RTL levels, depending on the errors and their severity.

This section includes descriptions of the RTL and front-end design methods used in this dissertation’s fused multiplier-adders. The back-end design was not included in the implementations, as the intended results are only for architectural comparisons and not for immediate industrial tape-outs. A design in the back-end of the flow requires an enormous amount of resources, effort, and time to fine-tune the circuit and prepare it for manufacturing.

The decision to keep all designs in the front-end of the flow was made early in the project, concluding that the results produced by the back-end design, such as electromagnetic reports, local heating, timing results that are equal to or slightly better than the Steiner front-end estimations, and mask layers of the metal interconnects would provide little additional useful information for an academic architectural comparison. Additionally, the back-end design has more focus on connecting up various front-end blocks and fixing bad route netlists than evaluating architecture, so a line was drawn and the fused multiplier-adder designs concluded at a transistor-level with a floor plan and Steiner routing parasitics.

3.2.2 High-Level Design – Verilog RTL

The first step in the design of the fused multiply-add units was the translation of the block diagram to an RTL description via the Verilog2K high-level design language (HDL). Each design has been coded in Verilog2K RTL, compiled by Synposys Chronologic VCS compiler tools, and debugged using the Novas Debussy logic analysis software. Verification has been performed by a collection of test vectors and test benches which include comprehensive corner cases, as well as built-in Verilog behavioral checks within the RTL models themselves to ensure a more formal level of functional verification. A description, screenshots, and examples of each RTL toolset are described in the following sub-sections.

3.2.2.1 Verilog 2K HDL and the VCS Compiler

The Verilog 2K RTL HDL language is a syntax coding standard most recently updated in 2005 [33]. This language is written so that every line of code is executed simultaneously, simulating electrical components with multiple inputs and outputs that execute on multiple signals at the same time. The code may be written in any text-based editor compliant with the user's operating system, and must be compiled with a tool conforming to the Verilog 2K standard.

An example of the Verilog HDL syntax used for the fused multiply-add designs is provided in the RTL design of a Booth multiplexer. In this example, Verilog code is written using the UNIX-based XEMACS program to create the Booth multiplexer in a format that is accepted by the debugging and simulation software.

The Booth multiplexer seen in Figure 3.2.2.1 is a block used in a radix-4 multiplier tree that accepts inputs from both a Booth encoder block and an un-processed multiplier operand. The Booth encoder block outputs signals from the radix-4 Booth encoding of the other multiplier operand input that determine if the partial product bits created for a specific position in the multiplier tree require an inversion or shifting, i.e., if the operand is multiplied by {-2,-1,0,1,2}. The behavioral code of this architectural block is realized in the Verilog code seen in Figure 3.2.2.2.

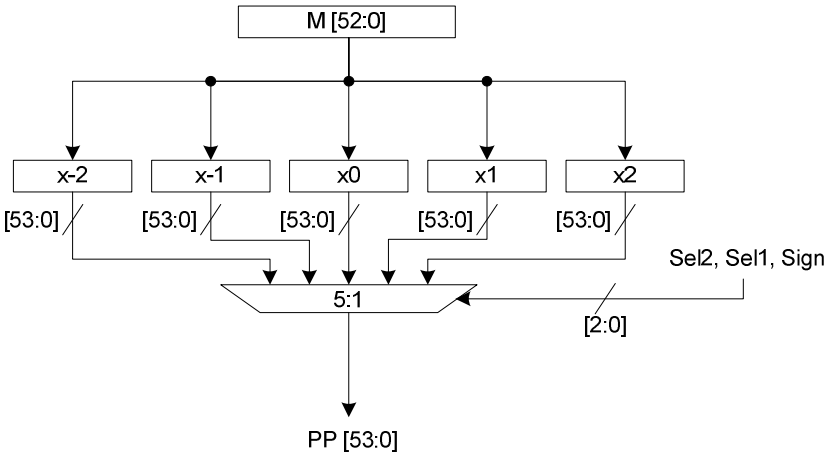

Figure 3.2.2.1 Radix-4 Booth multiplexer

```
// ================================================================
// (#1) Booth Mux
// ================================================================
module fma_lib_boothmux
            (
            M,
            Sel2,
            Sel1,
            Sign,

            PP,
            );
 // =====
 // Inputs
 // =====
  input  Sel1 ;
  input  Sel2 ;
  input  Sign;
  input [52:0] M;

 // ======
 // Outputs
 // ======
 output reg [53:0] PP;

 // =========
 // Booth Mux
 // =========
  reg [53:0]         PP_Shift;

  always @* begin
    PP_Shift[53:0] = ({54{Sel1}}  & {1'b0, M[52:0]}) | ({54{Sel2}} & {M[52:0], 1'b0});
    PP[53:0] = PP_Shift[53:0] ^ {54{Sign}};
  end

endmodule
```

Figure 3.2.2.2 Verilog code for a radix-4 Booth multiplexer

After a Verilog block like the Booth multiplexer is combined with all the blocks required for a design, the total model is not yet ready for compilation and simulation in a debug tool. A functional block or collection of blocks may be syntactically correct according to the Verilog standard, but without an input/output file that produces vectors as stimuli, a debugger will provide no useful information on the design.

Figure 3.2.2.3 shows the syntax of an input/output file that creates two test vectors for the FMA_Classic collection of Verilog modules. The vectors are latched to virtual registers in an initialization statement, and time increments are described by numerical statements following a '#' sign (in this case, a 100ps increment from the initial state to the following state). After the test vectors are described, the file makes a call to the top level of the fused multiply-add model that connects vectors to the various inputs.

When a Verilog collection has been coded along with an input/output file for stimulus, the code needs to be compiled and ported to a format that can be read by a debugging tool. For the fused multiply-add designs, the Synposys Chronologic VCS compiler is used to collect all the Verilog files and combine them into a single object file database. If the code compiles without errors, the object code may be simulated and prepared for debugging.

To follow with the same example of the input/output file, a UNIX terminal output of the VCS compilation and simulation tools for the fused multiply-add Classic test is shown in Figure 3.2.2.4. The first command is the call to the VCS compiler, and the second './simv' command is the tool that passes the vectors through the Verilog models and dumps the outputs to a database for the debugger. If both tools complete successfully, the models may be viewed in the virtual logic analyzer.

```
initial begin

  // ============================================
  // Initial Design Tests
  // ============================================

  A = 64'h4030_0000_0000_0000;
  B = 64'h4030_0000_0000_0000;
  C = 64'h4060_0000_0000_0000;
  fp_op = 1'b0;
  RNE_sel = 1'b0;
  RPI_sel = 1'b0;
  RMI_sel = 1'b0;
  $fsdbDumpvars;

  #100

  A = 64'h4030_ABCD_1234_4445;
  B = 64'h402C_4321_ABCD_AAAF;
  C = 64'h4080_0000_0000_0000;

  $fsdbDumpvars;

end

 wire [63:0] FMA_round_result;
 wire [63:0] FMA_unround_result;
 wire        inexact;

 FMA_Classic UU_FMA_Classic
  (
   .A(A[63:0]),
   .B(B[63:0]),
   .C(C[63:0]),

   .RNE_sel(RNE_sel),
   .RPI_sel(RPI_sel),
   .RMI_sel(RMI_sel),

   .fp_op(fp_op),

   .FMA_round_result(FMA_round_result[63:0]),
   .FMA_unround_result(FMA_unround_result[63:0]),
   .inexact(inexact)

   );
```

Figure 3.2.2.3 A Verilog input/output stimulus file

```
pcslw126:/proj/bobcat/user/equinnel/FMA/FMA_Classic --> vcs +v2k +vcsd -P
$DEBUSSY/share/PLI/vcsd_latest/LINUX/vcsd.tab $DEBUSSY/share/PLI/vcsd_latest/LINUX/pli.a FMA_Classic_test2.v
                Chronologic VCS (TM)
        Version X-2005.06-SP1-16 -- Sun Mar 18 16:45:36 2007
            Copyright (c) 1991-2005 by Synopsys Inc.
                ALL RIGHTS RESERVED

This program is proprietary and confidential information of Synopsys Inc.
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

***** Warning: ACC/CLI capabilities have been enabled for the entire design.
     For faster performance enable module specific capability in pli.tab file
Parsing design file 'FMA_Classic_test2.v'
Parsing included file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_Mul.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_exp.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_aligner.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_add.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_lza.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_normalizer52.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_normalizer109.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_complement.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_rnd.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_sign.v'.
Back to file 'FMA_Classic_top.v'.
Parsing included file 'FMA_Classic_lib.v'.
Back to file 'FMA_Classic_top.v'.
Back to file 'FMA_Classic_test2.v'.
Top Level Modules:
       FMA_Classic_test
TimeScale is 1 ps / 1 ps
Starting vcs inline pass...
6 modules and 0 UDP read.
        However, due to incremental compilation, only 1 module needs to be compi led.
recompiling module FMA_Classic_test because:
        This module or some inlined child module(s) has/have been modified.
if [ -x ../simv ]; then chmod -x ../simv; fi
g++  -o ../simv  5NrI_d.o 5NrIB_d.o 1u9E_1_d.o MmII_1_d.o ynMm_1_d.o m0nb_1_d.o ToFd_1_d.o KHnE_1_d.o SIM_l.o
/tool/cbar/apps/sim/vcs-2005.06-SP1-16/redhat30/lib/libvirsim.a /tool/cbar/apps/sim/debussy-
6.1v1p1/share/PLI/vcsd_latest/LINUX/ pli.a /tool/cbar/apps/sim/vcs-2005.06-SP1-16/redhat30/lib/libvcsnew.so
/tool/cbar/apps/sim/vcs-2005.06-SP1-16/redhat30/lib/ctype-stubs_32.a -ldl -lm  -l c -ldl
../simv up to date
CPU time: 3.120 seconds to compile + 12.440 seconds to link
[2]  - Done                    xemacs FMA_Classic_test2.v
pcslw126:/proj/bobcat/user/equinnel/FMA/FMA_Classic --> ./simv
Chronologic VCS simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version X-2005.06-SP1-16; Runtime version X-2005.06-SP1-16;  Mar 18 16: 46 2007
Novas FSDB Dumper for VCS2005.06-DKI, Release 6.1v2_RD (Linux) 01/19/2006
Copyright (C) 1996 - 2006 by Novas Software, Inc.
*Novas* Create FSDB file 'verilog.fsdb'
*Novas* Begin dumping the top modules, layer(0).
*Novas* End dumping the top modules.
```

Figure 3.2.2.4 UNIX output of VCS compile and simulation

3.2.2.2 Novas Debussy Debugger

After a selection of Verilog HDL has been compiled and simulated in the VCS software tools, the data are ready for viewing via Novas Debussy debugging GUI. Although the VCS CAD package allows for a debugging method via a text-based output, the models used in this dissertation are so large that such a debugging interface is excessively tedious. The Novas GUI debugger provides an easier visual analysis to debug and verify the Verilog RTL code.

A screenshot of the Debussy debugger tool is shown in Figure 3.2.2.5. The debugger has an array of options and colors that allow analysis from overall views of the Verilog system to specific bit-items of individual blocks. The tool itself is directly connected to the VCS compiler software, so that changes to the Verilog may be compiled and directly updated on the debugging screen. This live updating functionality between all the RTL tools allows for an efficient environment to develop and test RTL Verilog code capable of immediate digital testing feedback.

The verification of the behavioral Verilog models is conducted two-fold. First, the input/output Verilog files previously described allow for user-generated test cases to be input and tested, with outputs seen on the Debussy viewer. However, since user-generated test cases cannot cover all the internal cases of an architecture, behavioral statements have been embedded within the Verilog code to verify the functionality of various blocks.

As an example, Figure 3.2.2.6 shows the behavioral code for the output of a multiplier tree. The Verilog multiplier design uses a collection of blocks to produce a product in sum/carry format. To verify that the sum/carry vectors are correct, the multiplier outputs are combined with a behavioral adder and compared bit-by-bit in Debussy, shown in the screenshot in Figure 3.2.2.7, to a single-line behavioral multiplier statement using the same precision.

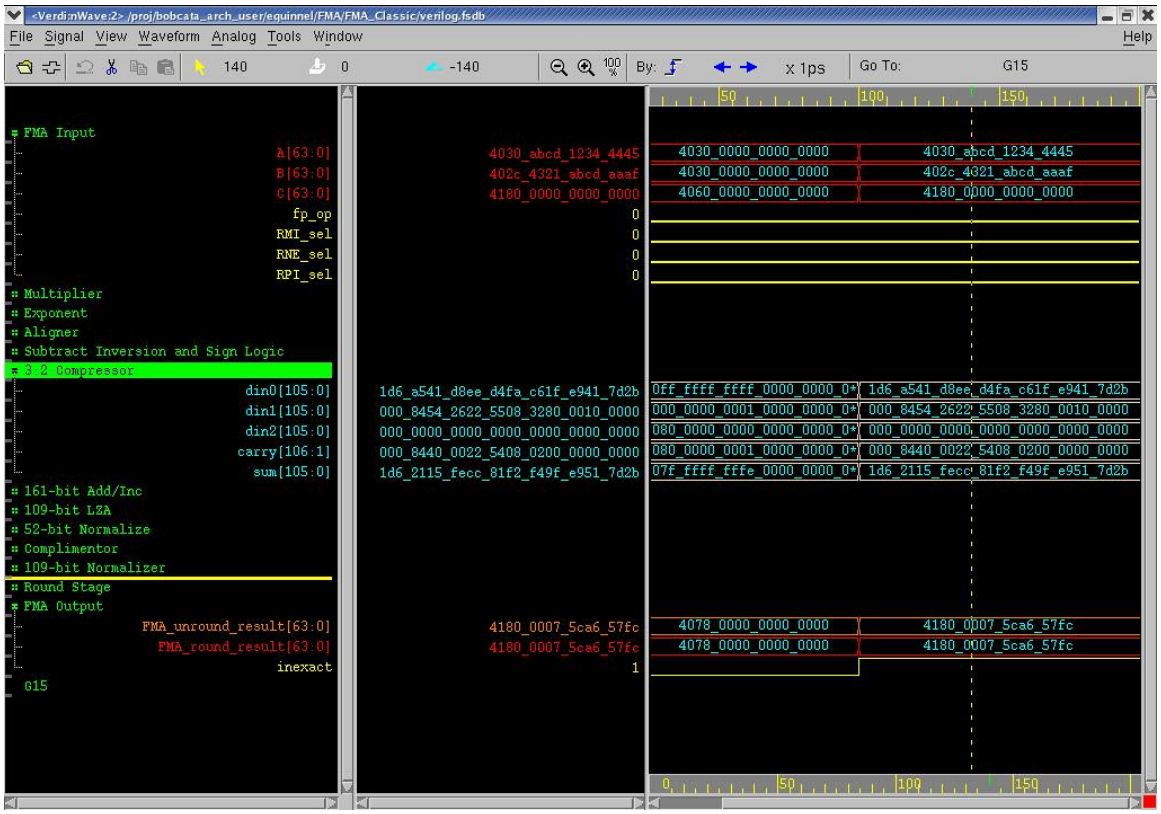

Figure 3.2.2.5 Novas Debussy debugger

```
// ============================================================
// (#7) Check results
// ============================================================

reg [105:0]                Result_verilog;
reg [105:0]                Result_behavior;

always @ * begin
  Result_verilog[105:0] = Mul_mantissa_sum[105:0] + {Mul_mantissa_carry[105:1],1'b0};
  Result_behavior[105:0] = A_mantissa[52:0] * B_mantissa[52:0];
end
```

Figure 3.2.2.6 Verilog behavioral checkpoint code

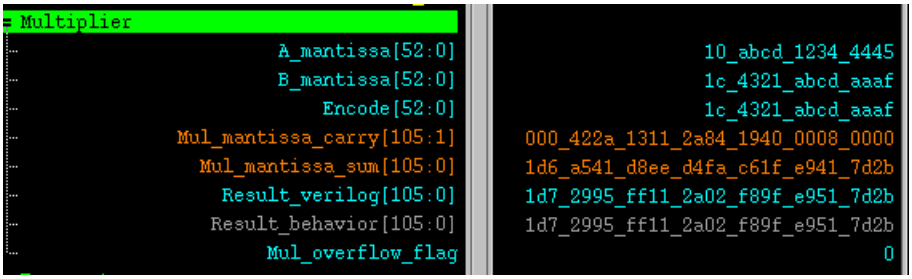

Figure 3.2.2.7 Debussy behavioral checkpoint screenshot

Any Verilog design in the fused multiply-add design process had to pass every user- and randomly-generated test vector with both the test bench block outputs and internal behavioral checkpoints before exiting the RTL stage. While all these checks do not replace formal verification, the behavioral and vector verification tests conducted has been as comprehensive as possible.

True circuit verification at the RTL level involves a series of test benches and checks against units already in silicon, all conducted by a full team of verification engineers. Such staffing was not available for these designs, so all tests were performed iteratively until the new units passed all available behavioral and vector cases.

3.2.3 Front-End Implementation – The AMD AXE Flow

The AMD "axe" flow is the implementation CAD toolset used to take a design from Verilog RTL to GDSII mask-layer tape-out data. The flow itself is split into two halves in a front-end and back-end partition. The front-end of the flow deals with what the system calls "system-level modules," (SLMs), and the back-end uses higher-level blocks (which are essentially a collection of SLMs) called "route-level modules," (RLMs), and "top-level modules" (TLMs).

A completed front-end design in the axe flow consists of a SLM block database model containing all the information on a full-level transistor- and gate-level floor plan, Steiner routing parasitics, flattened internal format and SPICE netlists, parasitic timing runs, and SLM power simulations. The transistor-level SLMs are connected to a variety of

manufacturing models that vary in process corners and threshold voltages. A completed back-end design consists of RLMs and a TLM that have passed every possible electrical, timing, and other analysis checkpoint in the flow. The complete TLM is the database of the GDSII mask layers.

As mentioned in the introduction, the fused multiply-add designs are all at front-end completion, meaning full SLM models that pass the flow's checkpoints. The back-end RLMs and TLMs were not created as the models do not provide much more valuable information for an architectural comparison than a complete SLM can already provide. Back-end models concentrate on stitching up SLMs, auto-routing, and fixing dirty netlists, and such information is not essential for the comparisons at hand.

The following sub-sections provide the steps and details of the front-end AMD axe design flow. While all the details of each step are too numerous to list here, each step in the SLM flow is described with a brief overview, example, and screenshot to provide a basic understanding of a circuit's transformation from RTL Verilog code to a transistor- and gate-level electrical models.

3.2.3.1 Gate Level Verilog using the 'Barcelona' library

The first step in the AMD axe flow is a circuit designer's translation of architectural RTL code into its "gate-level" equivalent using a standard cell library. The gate-level description is a Verilog equivalent to the RTL as described by small Verilog modules that make up basic digital logic components, such as inverters, NAND gates, MUXes, and so on. These gate-level Verilog modules are each individually linked to a Cadence-based schematic, layout, and technology library that describes the component at an electrical and manufacturing level.

The standard cell library used for a circuit's implementation is both process and project dependent. At AMD, entire teams of engineers are employed to build and layout these building block components. Each team is geared to design components that fulfill the

needs and performance targets as requested by the project's implementation engineers. Additionally, any engineer may custom-design a library component, perform layout and characterization runs, and enter the block into the standard cell project database. This custom block design procedure is not covered here.

The library database chosen to implement the fused multiply-add designs comes from the AMD 'Barcelona' project, which is a 65nm silicon on insulator (SOI) design of an x86 native quad-core processor. When the fused multiply-add design began, the 'Barcelona' library was the most cutting-edge and comprehensive library available, so it was a natural choice to implement a design involving brand new floating-point architectures. Additionally, the AMD quad-core is already at the silicon level, which largely verifies the accuracy and functionality of the library simulation models.

An example of the transition of a block from architecture, to RTL, to gate-level Verilog is provided showing the construction of a 3-bit aligner block. A 3-bit aligner takes an input string and shifts the data to the right between 0- and 3-bit positions based on the 2-bit aligner control input. The architectural diagram and Verilog code are shown in Figure 3.2.3.1 and Figure 3.2.3.2, respectively.

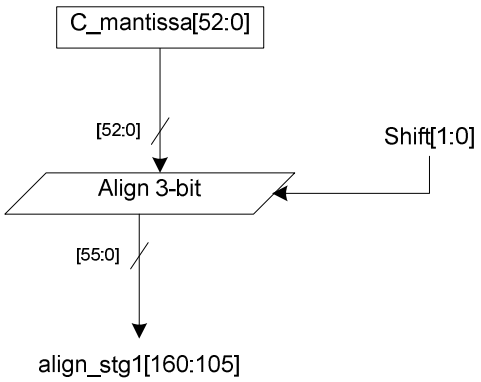

Figure 3.2.3.1 3-bit aligner architecture

```
// ================================================
// (#2)  Align 0, 1, 2, 3   ctl bits [1:0]
// ================================================

reg [160:105] align_stg1;

always @ * begin

  case(1'b1)
       shift_1 & ~shift_3 : align_stg1[160:105] = {1'b0, C_mantissa[52:0], 2'b0};
       shift_2 & ~shift_3 : align_stg1[160:105] = {2'b0, C_mantissa[52:0], 1'b0};
       shift_3            : align_stg1[160:105] = {3'b0, C_mantissa[52:0]};
       default            : align_stg1[160:105] = {C_mantissa[52:0], 3'b0};
  endcase // case(1'b1)

end
```

Figure 3.2.3.2 3-bit aligner Verilog RTL

The gate-level equivalent of the 3-bit aligner must first consider what components are needed to logically create the architectural block – in this case, a series of 4:1 MUXes. When basic components are decided, a circuit designer then must consider what loads the cells drive, as well as whether any control signals need a fan-out to drive components that perform work on a string of inputs. Since the example 3-bit aligner works on an incoming 53-bit string, a total of 56 MUXes must be used – one per bit alignment possibility. The incoming control signals are probably too weak to individually drive 56 different 4:1 MUXes, so the controls are buffered with inverter trees before input selection. The gate-level schematic and gate-level Verilog are shown in Figure 3.2.3.3 and Figure 3.2.3.4.

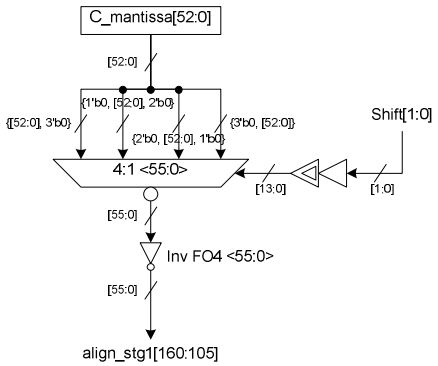

Figure 3.2.3.1 Gate-level schematic of a 3-bit aligner

```
// ================================================
// (#2)  Align 0, 1, 2, 3   ctl bits [1:0]
//
//  Trees must drive 56 mux inputs. 3-stages for
//  fanout with inverted drivers
// ================================================

        wire      shift_1_X;
        wire [1:0] shift_1_P;
        wire [6:0] shift_1_PX;

        inx4   UU_shift_1_X         ( .A(exp_difference_aligner_ctl[0]),          .Z(shift_1_X)       );
        inx3_5 UU_shift_1_P[0]    ( .A(shift_1_X),                              .Z(shift_1_P[0])    );
        inx4_5 UU_shift_1_P[1]    ( .A(shift_1_X),                              .Z(shift_1_P[1])    );
        inx4   UU_shift_1_PX[6:0] ( .A(shift_1_P[1/4,0/3]),                     .Z(shift_1_PX[6:0]) );


        wire      shift_2_X;
        wire [1:0] shift_2_P;
        wire [6:0] shift_2_PX;

        inx4   UU_shift_2_X         ( .A(exp_difference_aligner_ctl[1]),          .Z(shift_2_X)       );
        inx3_5 UU_shift_2_P[0]    ( .A(shift_2_X),                              .Z(shift_2_P[0])    );
        inx4_5 UU_shift_2_P[1]    ( .A(shift_2_X),                              .Z(shift_2_P[1])    );
        inx4   UU_shift_2_PX[6:0] ( .A(shift_2_P[1/4,0/3]),                     .Z(shift_2_PX[6:0]) );


        wire [160:105] align_stg1_X;

        imux4ux1 UU_align_stg1_X[160:105]  (      .D0({3'b0, C_mantissa_P[52:0]}),
                                                  .D1({2'b0, C_mantissa_P[52:0], 1'b0}),
                                                  .D2({1'b0, C_mantissa_P[52:0], 2'b0}),
                                                  .D3({C_mantissa_P[52:0], 3'b0}),

                                                  .S0(shift_1_PX[6:0/8]),
                                                  .S1(shift_2_PX[6:0/8]),

                                                  .Z(align_stg1_X[160:105]) );

        wire [160:105] align_stg1;

        inx4 UU_align_stg1[160:105] ( .A(align_stg1_X[160:105]),    .Z(align_stg1[160:105]) );
```

Figure 3.2.3.2 Gate-level Verilog of a 3-bit aligner

3.2.3.2 Flattening the Netlist – axe -flat

After creating a gate-level Verilog description of the RTL, the first step in the axe flow is to flatten the netlist. Gate-level Verilog is commonly written in hierarchical format, allowing the engineer to organize the design in logical and readable code. However, the axe flow requires gate-level data to be in several single file descriptions, including a placement file, standard cell, and transistor level models. The code is "flattened" into a single description and translated into these formats. Additionally, like any code, the Verilog must be checked by a compiler and be free from errors in order to translate the data correctly. The step "axe –flat" removes the gate-level hierarchy, compiles the code, and translates data into the required data formats and model descriptions.

After the code is flattened and compiled, axe –flat step runs a series of connectivity tests to verify that the cells and transistors are interconnected in a way providing a clean netlist. For example, the step checks to make sure that every input and output of every block has a driver and a receiver. If a cell does not drive anything, the gate-level Verilog should explicitly state "UNUSED" in the output net to meet AMD flow requirements. Another example of the net check is the use of buffers before MUXes with pass-gate transistors. Classic noise analysis and transmission studies always single out the problems of a pass-gate transistor that is driven from a far away or weak source. The axe –flat step requires that pass-gate transistors are always buffered locally.

A sample output from a UNIX prompt running the axe –flat step is shown in Figure 3.2.3.5 and continued in Figure 3.2.3.6.

```
pcslw126:/proj/bt/users/equinnel/FMA_axe/FMA_Classic --> axe -flat

** START FormalVer,clear_flags 1.48 03/18/2007 21:14:56 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Axe run dir: /proj/bt_users/equinnel/FMA_axe/FMA_Classic

** START Flat 1.59 03/18/2007 21:14:58 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
chdir gate
Running xnlw on all the top-level schematics:
Successfully added (FMA_Classic_Mul( macrolib) to /proj/bt_users/equinnel/FMA_axe/FMA_Classic//gate/StopChunkList.txt
Successfully created FMA_Classic.sourcelist.xnlw
```

Figure 3.2.3.3 UNIX output of axe -flat (part 1)

```
Running v92udb on tmp.9019.v9.FMA_Classic.vlist (FMA_Classic_Mul.v9 FMA_Classic_aligner.v9 FMA_Classic_exp.v9
FMA_Classic_lib.v9 FMA_Classic_sign.v9 FMA_Classic_add.v9 FMA_Classic_lza.v9 FMA_Classic_normalizer52.v9
FMA_Classic_complement.v9 FMA_Classic_normalizer109.v9 FMA_Classic_rnd.v9 FMA_Classic_top.v9 FMA_Classic.v9).
Running design->subdesign pin consistency check
Archiving sources
Fixing up $ and # characters in instance and net names
Checking tsize vs sfx for consistency
Tying off unconnected scan pins
Tying off macro repeater pins
Hooking up cell power/ground pins to default power/ground net
Flattening design
Running unique sizer
Fixing up udb unused nets
No DC Object Waivers file exists...
Running gater topology check...
Designcheck Revision Information
Revision: 1.16
Date: 2006/03/02 16:43:28
Author: franks

  Passed gater topology check
Running names check...
Designcheck Revision Information
Revision: 1.23
Date: 2005/12/14 22:30:32
Author: carlob
  Passed names check.
Running clock gater expansion
Estimated area usage: 42.67%
** FINISH Flat 1.59 03/18/2007 21:15:49 equinnel pcslw126 Elapsed 00:00:51

** START DesignCheck,flat_slm 1.72 03/18/2007 21:15:55 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Now Running Check(s) : flat_slm
No DC Object Waivers file exists...
Running Check(s): driver cellsize illegalcell flop_clock tristateconx unbufmux cgnames cgnetnames clkgtr clkconx
DC#2    Rule Check driver                    =>Passed<=    Severity =>SAFE<=
DC#25   Rule Check cellsize                  =>Passed<=    Severity =>SAFE<=
DC#30   Rule Check illegalcell               =>Passed<=    Severity =>SAFE<=
DC#19   Rule Check flop_clock                =>Passed<=    Severity =>SAFE<=
DC#33   Rule Check tristateconx              =>Passed<=    Severity =>SAFE<=
DC#16   Rule Check unbufmux                  =>Passed<=    Severity =>SAFE<=
DC#21   Rule Check cgnames                   =>Passed<=    Severity =>SAFE<=
DC#22   Rule Check cgnetnames                =>Passed<=    Severity =>SAFE<=
DC#18   Rule Check clkgtr                    =>Passed<=    Severity =>SAFE<=
DC#5    Rule Check clkconx                   =>Passed<=    Severity =>SAFE<=

Designchecks Passed. See results files in verif/designcheck for details
driver is the check to be run
cellsize is the check to be run
illegalcell is the check to be run
flop_clock is the check to be run
tristateconx is the check to be run
unbufmux is the check to be run
cgnames is the check to be run
cgnetnames is the check to be run
clkgtr is the check to be run
clkconx is the check to be run
** FINISH DesignCheck,flat_slm 1.72 03/18/2007 21:16:15 equinnel pcslw126 Elapsed 00:00:20
```

Figure 3.2.3.4 UNIX output of axe -flat (part 2)

3.2.3.3 Translating for Verification – axe -u2v

The "axe –u2v" step is a simple and quick format conversion from a database created by the axe –flat step into a single-file gate-level Verilog file used by the formal verification software. This step also creates single-file Verilog capable of translating into a SPICE netlist for power simulations. A UNIX output of the axe –u2v step is shown in Figure 3.2.3.7.

```
pcslw126:/proj/bt/users/equinnel/FMA_axe/FMA_Classic --> axe -u2v

** INVOCATION by equinnel: -u2v : /proj/bt_users/equinnel/FMA_axe/FMA_Classic/

** START Udb2Verilog 1.23 03/18/2007 21:39:27 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Generating gate/derived/FMA_Classic.gate.v
Generating gate/derived/FMA_Classic.flat.v
Generating gate/derived/FMA_Classic.flatnomove.v
** FINISH Udb2Verilog 1.23 03/18/2007 21:39:44 equinnel pcslw126 Elapsed 00:00:17
```

Figure 3.2.3.5 UNIX output of axe -u2v

3.2.3.4 Equivalency Checking – axe -formal

When a gate-level Verilog module is coded using an AMD standard-cell library, the most important step in the SLM front-end flow is to verify that the transistor and gate-level description matches the RTL exactly. This step requires absolute matching for every possible input combination, so the verification must be the highest-level possible. To meet this equivalence requirement on a formal level, the axe flow uses the command "axe –formal" to call a process that collects the various Verilog files, RTL, and any constraints required, and passes them to the Cadence/Verplex logical equivalence checker (LEC) software tool.

The Cadence/Verplex LEC CAD suite acts both as a UNIX tool that can process quickly with the axe flow as well as a slower GUI debugger should an error occur in the RTL vs. gate-level Verilog test. The GUI debugger is called on command and has a variety of features ranging from a gate-level schematic viewer and generated test vectors that induce a failure. An example of the UNIX output of the LEC check tool with a dirty

netlist is shown in Figure 3.2.3.8. The same dirty list is shown as on the GUI vector generation screen in Figure 3.2.3.9 and the debugging GUI screen in Figure 3.2.3.10.

```
Contents of Ladner_73.parsed.stats:
Adjusted Statistics, originals in /proj/bt_users/equinnel/axe/bt_fpa/verif/verplex/rtl2gate/Ladner_73.verplex.log:
================================================================================
                Compare Result        Golden       Revised
--------------------------------------------------------------------------------
Primary inputs                        148          148
  Mapped                              148          148
    Extra                             0            0
    Not-mapped                        0            0

No Tri-state (Z) key points

Primary outputs                       75           75
  Mapped                              75           75
    Equivalent                74
    Inverted-equivalent       0
    Non-equivalent            1                          ERROR !!!!!
    Abort                     0
  Unmapped                            0            0
    Extra                             0            0
    Not-mapped                        0            0

No Black-box key points

No Cut key points

State key points                      0            0
  Mapped                              0            0
    Equivalent                0
    Inverted-equivalent       0
    Abort                     0
    Non-equivalent            0
  Unmapped                            0            0
    Extra                             0            0
    Unreachable                       0            0
    Not-mapped                        0            0
================================================================================
Duplicate checks (removed from above)
================================================================================
Compared points       PO       DFF      DLAT      Total
--------------------------------------------------------------------------------
Equivalent            0        0        0         0
Inverted-equivalent   0        0        0         0
Non-equivalent        0        0        0         0
================================================================================
Gaters removed Golden     =    0
Gaters removed Revised    =    0
Duplicates removed Golden =    0 PO,    0 State points
Duplicates removed Revised =   0 PO,    0 State points


ERROR !!!!!  LEC DIRTY -- NOT SETTING FLAG  ERROR !!!!!
Non-equivalent Primary outputs
```

Figure 3.2.3.6 UNIX output of axe –formal

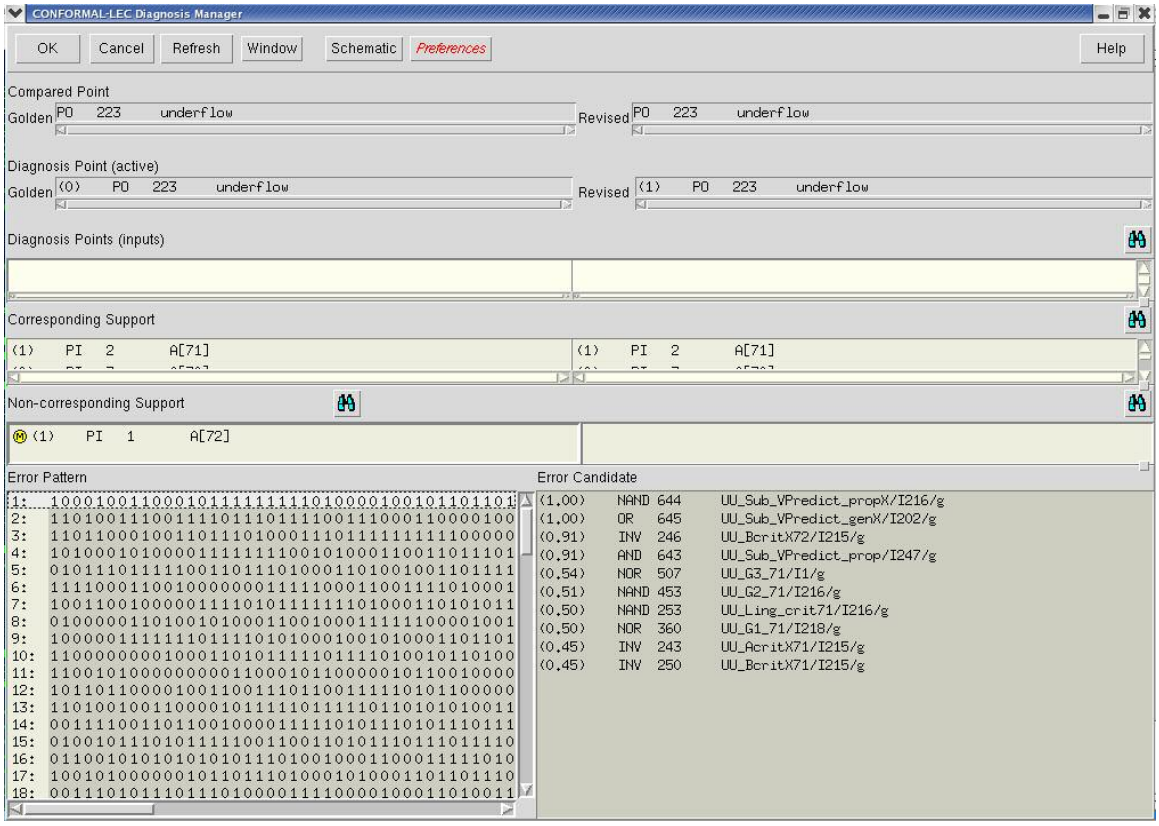

Figure 3.2.3.7 LEC error vector screen

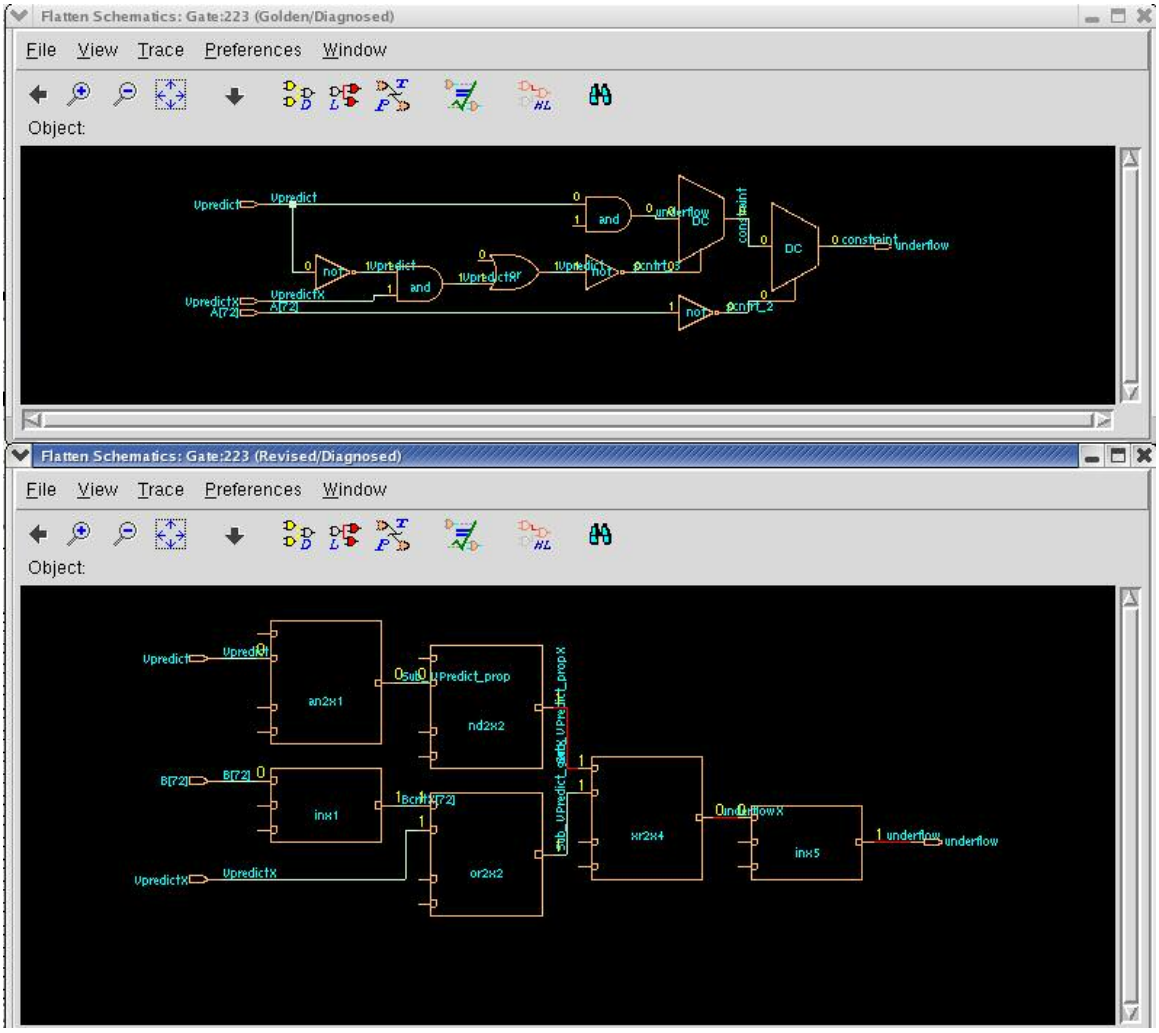

Figure 3.2.3.8 LEC schematic debugger

3.2.3.5 Floorplan Layout – axe -place and axe -vp

After the equivalency checking of a gate-level implementation, the design is ready to be floorplanned in the in-house placing toolset. This placement tool is very advanced and has a huge number of features, including an in-house syntax for floorplanning capable of live updating right as code is written, on-demand connectivity information, and interfacing with timing tool results. The entire CAD placement system is called the “px” placer tool, and is used throughout the front-end and back-end axe flow.

To initialize the placer tool, the “axe –place” command is used to setup a bounding box, identify clock rows, and place the cells that have been written in the in-house placer code. This step also identifies any floorplanning violations, including cells outside the boundary, unconnected design for test (DFT) scan chains, and unplaced cells. A UNIX output of the axe –place routine with unstitched scan-chains is shown in Figure 3.2.3.11 and continued in Figure 3.2.3.12.

```
pcslw126:/proj/bt/users/equinnel/FMA_axe/FMA_Classic --> axe -place

** INVOCATION by equinnel: -place : /proj/bt_users/equinnel/FMA_axe/FMA_Classic/

** START Place 1.256 03/18/2007 22:43:10 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Copying flat design to place
Updating boundary information from fplan.data
WARNING(Place): Can't find FMA_Classic in global fplan file
Annotating placement information
Postplacing clockgaters.pp
Archiving sources
Running placement checks

Placement check summary:
  multiple instantiations: 0
  overplaced instances:    0
** FINISH Place 1.256 03/18/2007 22:43:21 equinnel pcslw126 Elapsed 00:00:11

** START DesignCheck,place_slm 1.72 03/18/2007 22:43:22 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Now Running Check(s) : place_slm
No DC Object Waivers file exists...
Running Check(s): grid gater_to_load_dist clockrow clkcell_placement
grid is the check to be run
DC#14   Rule Check grid                          =>Passed<=    Severity =>SAFE<=
DC#22   Rule Check gater_to_load_dist            =>Passed<=    Severity =>SAFE<=
gater_to_load_dist is the check to be run
clockrow is the check to be run
DC#20   Rule Check clockrow                      =>Passed<=    Severity =>SAFE<=
DC#24   Rule Check clkcell_placement             =>Passed<=    Severity =>SAFE<=
```

Figure 3.2.3.9 UNIX output of axe -place (part 1)

```
Designchecks Passed. See results files in verif/designcheck for details
clkcell_placement is the check to be run

Placement check summary:
  grid failures: 0
  gater_to_load_dist failures: 0
  clockrow failures: 0
  clkcell_placement failures: 0
** FINISH DesignCheck,place_slm 1.72 03/18/2007 22:43:36 equinnel pcslw126 Elapsed 00:00:14

** START ScanStitch 1.62 03/18/2007 22:43:39 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Removing AUTOBUFFERS
Stitching using scan files from place/scan
AUTOBUFFERING TURNED OFF
Saving hard order files to place/scan/derived
** FINISH ScanStitch 1.62 03/18/2007 22:43:49 equinnel pcslw126 Elapsed 00:00:10

** START SpareRepeaters 1.9 03/18/2007 22:43:49 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
** FINISH SpareRepeaters 1.9 03/18/2007 22:43:49 equinnel pcslw126 Elapsed 00:00:00

** START DesignCheck,scan_slm 1.72 03/18/2007 22:43:49 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Now Running Check(s) : scan_slm
No DC Object Waivers file exists...
Running Check(s): overlaps bounds unplaced scan
DC#13   Rule Check overlaps                    =>Passed<=     Severity =>SAFE<=
overlaps is the check to be run
bounds is the check to be run
DC#12   Rule Check bounds                      =>Passed<=     Severity =>SAFE<=
DC#11   Rule Check unplaced                    =>Passed<=     Severity =>SAFE<=
DC#37   Rule Check scan                        =>Failed<=     Severity =>WARNING<=

One or more Designchecks Failed. See results files in verif/designcheck for details
ERROR: check(s) failed.  See summary verif/designcheck/dc.summary.
unplaced is the check to be run
scan is the check to be run

Placement check summary:
  overlaps failures: 0
  bounds failures: 0
  unplaced failures: 0
  scan failures: 652
** FINISH DesignCheck,scan_slm 1.72 03/18/2007 22:44:03 equinnel pcslw126 Elapsed 00:00:14

** START PlaceScanSummary 1.2 03/18/2007 22:44:03 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Placement check summary:
  multiple instantiations: 0
  overplaced instances:    0
  grid failures: 0
  gater_to_load_dist failures: 0
  clockrow failures: 0
  clkcell_placement failures: 0
Post ScanStitch Placement check summary:
  overlaps failures: 0
  bounds failures: 0
  unplaced failures: 0
  scan failures: 652
** FINISH PlaceScanSummary 1.2 03/18/2007 22:44:03 equinnel pcslw126 Elapsed 00:00:00
```

Figure 3.2.3.10 UNIX output of axe -place (part 2)

When the axe flow initializes the floorplan boundary and creates a file with cell placement information, the in-house floorplanning syntax files may then be coded to start placement visible in what is known as the "VP" floorplan GUI tool. The command "axe –vp" starts the VP GUI and links the in-house placement files for a live update as code is written.

To provide an example of the VP GUI and the px placer syntax, Figure 3.2.3.13 shows the placement code for the sum block of a 109-bit adder. The syntax itself is simple, including information such as color, relative or absolute row placement, and how strings of cells are unrolled via stacking, stacks with skips, or interleaving with other cells.

```
######################
# Sum Block
######################

module SumBlock {

color thistle

row 0
$name_SD_54 skip=1
$name_SH_54 skip=1
$name_SI_54 skip=1

row 1
$name_SA_[53:0] skip=1
$name_SG_[53:0] skip=1
$name_SF_[53:0] skip=1

row 2
$name_SD_[53:0] skip=1
$name_SC_[53:0] skip=1
$name_SB_[53:0] skip=1
$name_SE_[53:0] skip=1
$name_SH_[53:0] skip=1
$name_SI_[53:0] skip=1

row 109
$name_LOA
$name_LOB

}

UU module=SumBlock row=$bottom
```

Figure 3.2.3.11 PX placement code for an adder sum block

The VP GUI interface tool shows the gate-level cell placements as instructed by the px file code. The VP tool masks the internal layouts of the standard cells for ease of viewing and shows their various interconnects using either color-coded flylines or Steiner route estimations. The viewer is also capable of hierarchical placement, allowing the user to place modules and use different levels of files to organize the code into a more readable format. A zoomed-in screenshot of the VP GUI tool displaying a piece of the example sum block code is shown in Figure 3.2.3.14 and Figure 3.2.3.15. The first figure shows a cell's input and output flylines to other cells, while the second figure shows the same cell with a Steiner routed output net.

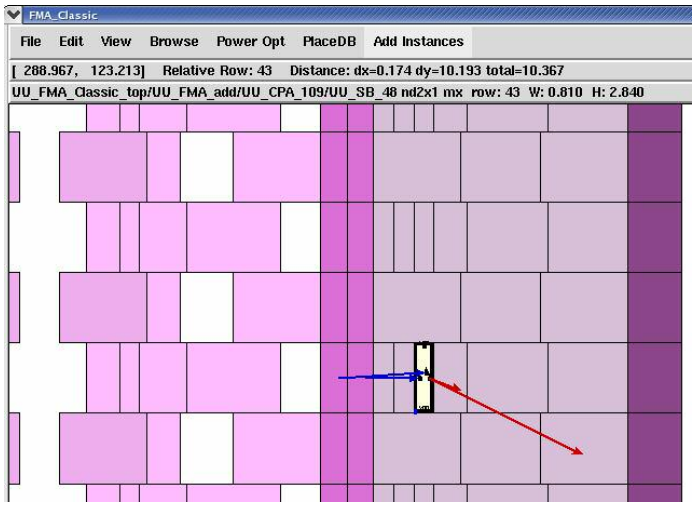

Figure 3.2.3.12 VP output of a cell with I/O flyline interconnects

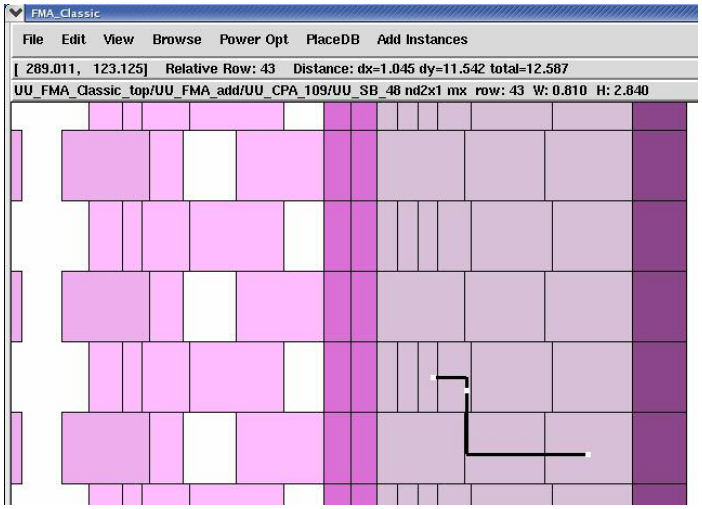

Figure 3.2.3.13 VP output of a cell with a Steiner output interconnect

As previously mentioned, the VP GUI tool is also capable of interfacing with the timing files produced by the axe timing steps. This interface allows for a visual identification of the circuit's critical paths, as well as any other timing paths the user wishes to single-out and visualize. This tool, like the interconnection feature when highlighting individual cells, is capable of showing the flylines or Steiner routes of the path. Additionally, the incremental latencies of each cell stage are optionally listed under their respective units when the timing interface is activated. Figure 3.2.3.17 and Figure 3.2.3.18 show selected screenshots of both the flyline and Steiner route schemes when activated with a timing tool critical path.

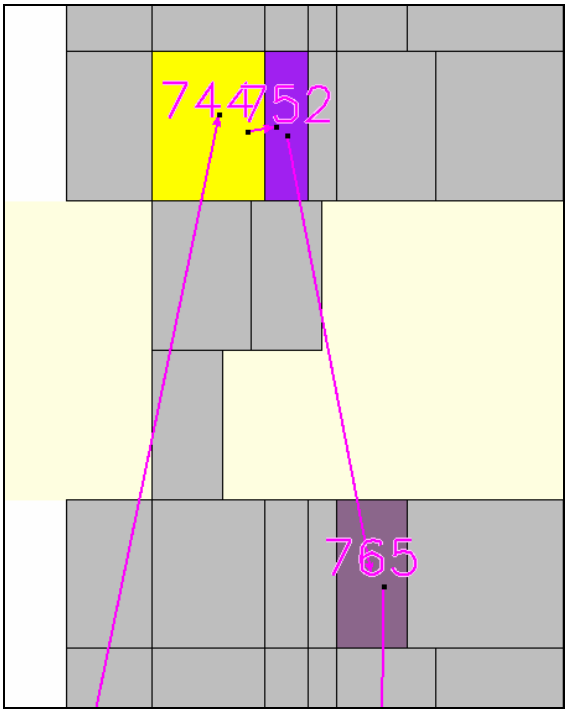

Figure 3.2.3.14 VP timing interface with flylines

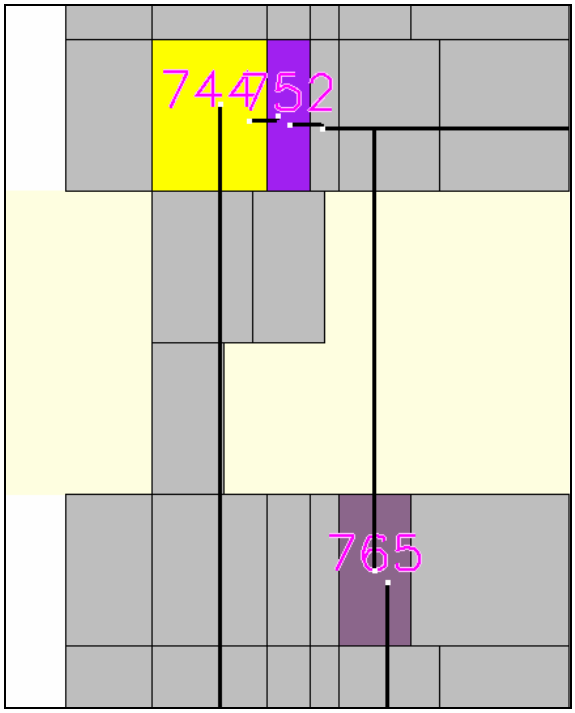

Figure 3.2.3.15 VP timing interface with Steiner routes

3.2.3.6 Placement-Based Estimated Timing – axe -espftime

When a placement is complete and all cells described in the gate-level Verilog occupy a space in the floorplan, the implementation is ready for placement-based estimated timing. This step, called by the command "axe –espftime", uses the Synopsys Primetime timing software along with a variety of internal algorithms and models to provide a very accurate and overly-pessimistic timing report. Aside from the power estimation, this step represents the final command in the front-end SLM design flow. A block that does not meet timing requirements in this step will in most cases not be accepted into the RLM flow. Timing requirements must be met in this timing model, and designs that do not meet the required specifications are re-iterated either in the SLM design or RTL model.

The placement-based timing model is considered pessimistic due to the way routing parasitics are calculated. Each net is routed with a Steiner estimate, or a Manhattan-style route designed for the shortest path on the lowest-level metal layer possible. While RLM designs may increase in actual routing distance, typically critical paths for long flylines are routed manually at the highest metal layer for minimal interconnect resistance, making the Steiner estimate, for the most part, pessimistic.

The axe command itself, like many of the steps in the flow, includes a long list of features. The Primetime reports from the timing run are parsed and processed with internal scripts to provide information on edgerates, effective fan-outs (EFOs) from cells (or the combined load of input cap and routing cap seen by a cell output), and even a recommended re-sizing script for cells identified having transistors with too little or to much physical width for current sourcing/sinking.

An example of the timing tool is shown in Figure 3.2.3.18 showing the UNIX output of the axe –espftime command executed on a full fused multiply-add Classic placement and gate-level model. Following, Figure 3.2.3.19 shows a segment from the Primetime parsed timing report, including EFO calculations, timing increments, and routing latency

increments. Figure 3.2.3.20 displays a report of suggested re-sizing of cells based on EFO calculation. Finally, Figure 3.2.3.21 shows a report of edgerates for various nets.

```
** START Timing,espf,setup,0,0 1.115 02/28/2007 20:53:21 equinnel pcslw126 Linux 2.4.21-47.ELsmp i686
Running: RM Old PT Session...
Running: Netlist Generation...
Running: Timing Analysis...
Running: Post Process...
Running: Saving Session SynDbs...
Timing Analysis completed.
** FINISH Timing,espf,setup,0,0 1.115 02/28/2007 21:02:33 equinnel pcslw126 Elapsed 00:09:12
```

Figure 3.2.3.16 UNIX output of axe –espftime

```
# Path 1: C_X[56]:R FMA_round_result_X[6]:R cycles=1
# Prev Slack: 262   Next Slack: 299   Total Delay: 1224
# collapsed 207 similar paths.

 Path  Incr   Dir  Fanout        Tran   TotC  Ceff   GatC  Instance or Net (arc)
-------------------------------------------------------------------------------
...
#
 261    7    F  inx4         10    ES   68%    13%  UU_FMA_Classic_top/UU_FMA_align/UU_shift_1_X (A -> Z)  EFO=2.057
 261    0    F  2            10    18    18     17  UU_FMA_Classic_top/UU_FMA_align/shift_1_X
 272   10    R  inx4_5       14    ES  100%    96%  UU_FMA_Classic_top/UU_FMA_align/UU_shift_1_P1 (A -> Z) EFO=3.686
 274    2    R  4            14    43    36     34  UU_FMA_Classic_top/UU_FMA_align/shift_1_P[1]
 284   10    F  inx4         14    ES   85%    80%  UU_FMA_Classic_top/UU_FMA_align/UU_shift_1_PX3 (A -> Z)  EFO=4.091
 285    1    F  8            14    40    36     32  UU_FMA_Classic_top/UU_FMA_align/shift_1_PX[3]
 #
```

Figure 3.2.3.17 A segment from a parsed Primetime report

```
###############################################################################
# Gates with greater than desired EFO.
###############################################################################
UU_FMA_add/UU_CPA_109/UU_G2_8     Z  (oai21x4)  EFO= 5.948  worstslack= -894  EFO_range=( 4.77, 2.77)
UU_FMA_add/UU_CPA_109/UU_P1_8     Z  (oai22x4)  EFO= 5.940  worstslack= -894  EFO_range=( 4.77, 2.77)
UU_FMA_exp/UU_exp_diff_sum4       Z  (inx2)  EFO= 5.076  worstslack= -894  EFO_range=( 4.77, 2.77)
UU_FMA_exp/UU_exp_cout_X          Z  (inx9)  EFO= 4.842  worstslack= -894  EFO_range=( 4.77, 2.77)
```

Figure 3.2.3.18 A segment from a re-sizing script

```
UU_FMA_Classic_top/UU_FMA_fmul/UU_PP_25/Sel2_X[0]      35      rise      inx6
UU_FMA_Classic_top/UU_FMA_fmul/PP_7[56]                32      fall      inx7
UU_FMA_Classic_top/UU_FMA_fmul/UU_PP_20/Sign_X[0]      32      rise      inx8
UU_FMA_Classic_top/UU_FMA_align/shift_BIG              32      rise      nd3x1
UU_FMA_Classic_top/UU_FMA_fmul/PP_11[56]               31      rise      inx7
```

Figure 3.2.3.19 A segment from a edgerate report

3.2.3.7 Power Estimation – HSim with axe-extracted SPICE netlist

The final simulation in the front-end design flow is estimated power consumption via SPICE netlists simulated in Synopsys HSim software. This step is not directly integrated into the axe flow and requires a small amount of user-generated files and controls. However, the base SPICE netlist comes directly from the axe –u2v Verilog single-file derivation as well as the standard-cell library extraction of SPICE models. Additionally, the Steiner routing parasitics are imported from the axe –espftime step and included in the total power simulation.

The first part of generating the files required for a HSim run is to convert the axe –u2v Verilog file and axe –espftime route parasitics into a single SPICE netlist via an internal script called "v2spi". Much like the RTL debugging simulations, a HSim run is meaningless without an input stimulus file, so the second step of the power simulation is to create a series of randomly generated inputs that is compatible with the SPICE netlist via an internal Python script. Finally, a HSpice technology file and process corner must be included in the set of files, a fixed frequency and temperature selected, and all the various HSim options changed to meet the requirements of the run.

When all files are prepared, the HSim program is executed and results are ready for viewing in a Spice Explorer waveform viewer GUI after the long simulation is finished. Simulations for the fused multiply-add designs averaged between 4 and 10 hours per 20 vector inputs running over simulation periods up to 30ns on AMD server farms. Figure 3.2.3.22 shows the UNIX output of such a HSim execution.

When the results of the simulation are viewed in the Spice Explorer, as shown in Figure 3.2.3.23, power is calculated by manual calculation and observation. Specifically, the integrated total current is observed over the various random-input clock periods, and the maximum total current found in a single cycle is selected (among other simulation runs as

well), normalized according to frequency, and multiplied by the simulation voltage. Specifically, the equation used for max power is as follows:

$$P_{\max} = \max\{\int_{t_0}^{t_1} i(Vss)dt\}\frac{1}{T_{period}}V_{DD} \qquad (1)$$

```
Synopsys Inc.
HSIMplus Linux 2.4.21 Version Z-2006.06-SP2-ENG3 - 194302162007
Tracking No - HSIMplus 2007.07.6
Copyright (C) 1998 - 2007. All rights reserved.

Simulation started on Sun Mar 11 15:47:28 2007
…
Subckt Defined/Used          : 146/79
Subckt with parameterized elem : 0
Subckt instantiated with param : 0
Maximum Circuit Level        : 2
Circuit Statistics

CAP          Elements        : 427375
GCAP         Elements        : 21604
VSRC_VS      Elements        : 197
VSRC_DC      Elements        : 2
SOI          Elements        : 177338

Total # of Elements          : 604912
Total # of Nodes             : 83453

DC initialization completes after 1000 iterations

End of operating point solution, CPU time used: 302.81 sec
Memory usage Physical: 103 MB, Virtual: 136 MB

Simulation Statistics

Comparison Errors            : 0
Accepted Time Steps          : 18757
Repeated Time Steps          : 31
Minimum  Time Steps          : 17843
MOS evaluations              : 1904964001

Simulation Parameters
Circuit Temperature          : 100
Transient Time               : 3e-08

End of transient, CPU time used: 20592.77 sec

End of circuit analysis, CPU time used: 20907.77 sec
```

Figure 3.2.3.20 UNIX output of a HSim power simulation

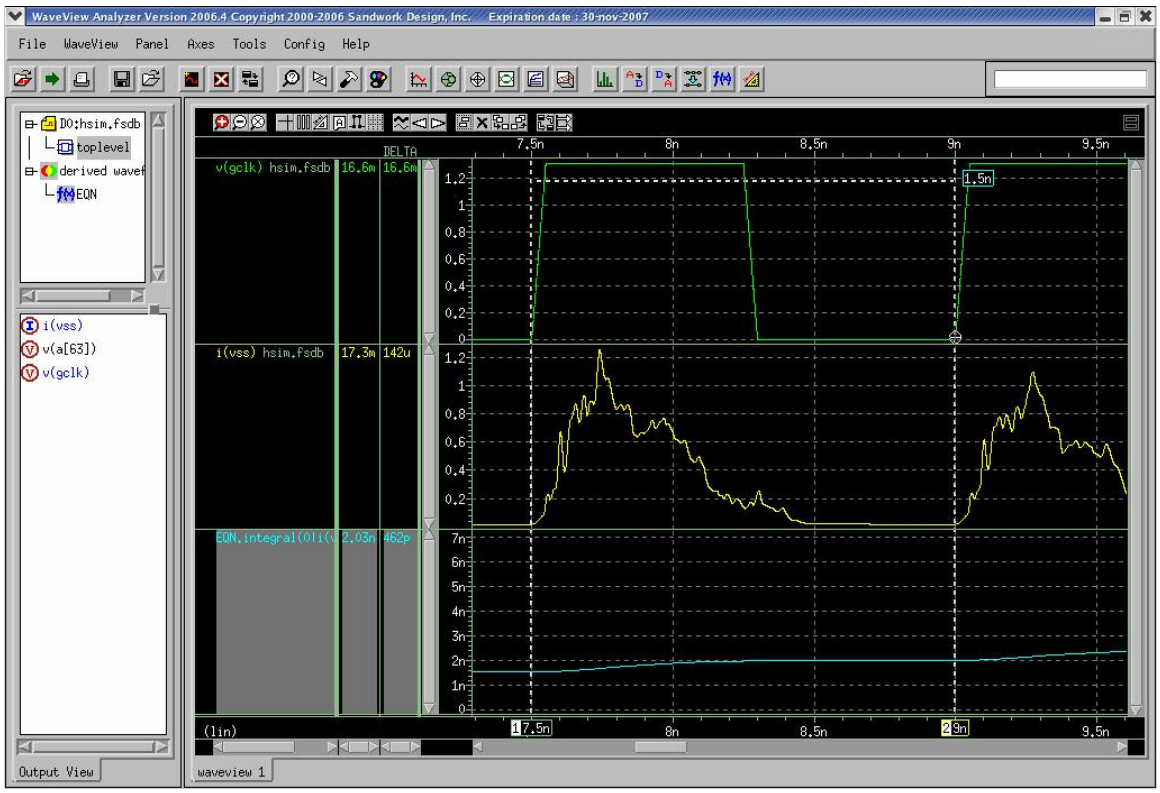

Figure 3.2.3.21 Spice Explorer power simulation screenshot

3.3 Floating-Point Components

In any floating-point arithmetic unit, the architecture itself is in essence a combination of smaller arithmetic components arranged in a way conducive to the functionality and performance desired from the overall design. While an architecture spends a great deal of focus on how the various components are arranged, a critical part of the design is how the components themselves are designed. A poor selection of internal component designs will guarantee a poor resulting architecture, no matter its organization.

When comparing architectures on their organizational merit alone, it is crucial to keep the internal components identical in design and execution. A new architecture may claim better results in performance, area, or power savings, but if this architecture is built with higher-performance blocks than the original, the true results are inconclusive. Architectures with different builds of similar units will always raise the question as to

whether the new architecture presented is better or whether the unit just had better execution.

For this reason, the components used in all the fused multiply-add designs have been built with identical component architectures and implementations into a pseudo-common floating-point library. If the bit-width needs of any single component requirement did not match a design in the collection of components, the implementation would simply expand the unit while keeping the design consistent. Naturally, various fine-tuning is required when a component is placed and drives actual loads which are heavily dependent on the architecture, but the basic philosophy has been to keep internal pieces as consistent as possible—so that results of the fused multiply-add simulations come from an architectural comparison, and not changes in components.

This section lists the major component designs and implementations used in the fused multiply-add designs. Each component is described with an overview of its architecture, screenshots of an implemented floorplan, and identification of its global use in various architectures. While undoubtedly some of the design selections may later be questioned, inciting a discussion that a different design would boost the performance of the fused multiply-add units, such disagreements do not affect the final results. As long as the components are consistent from design to design, the relative comparisons should be sound.

This does not mean however, for example, that all adders selected for the designs are ripple-carry architectures, nor does it mean that there was no thought put into the building of components. Each unit was built to be the highest-performance unit possible with all variables considered, including complexity and the time required to implement and debug. Additionally, every component used has been built only if it has actually been implemented in an industrial design. Highly abstract and theoretical versions of the same

components were not selected, as the goals of the fused multiply-add designs seek to narrow the number of original design comparisons.

3.3.1 Radix-4 53-bit x 27-bit Multiplier Tree

The first common component developed for the fused multiply-add designs is the largest of the floating-point library—a radix-4 53-bit x 27-bit double-precision multiplier tree. The design was chosen to be radix-4 due to the area and power savings as compared to a radix-2 multiplier, as well as the simplicity in design as compared to a radix-8 or higher, which requires the complex execution of an operand multiplied by ±3.

To begin the construction of the multiplier, one of the input operands must be radix-4 Booth encoded. This multiplier's Booth encoding, shown in Table 3.3.1, allows the number of partial products required to be cut in half. A Booth encoding does this by doubling the range of each bit in the multiplier – allowing a digit to represent a number anywhere in the range {-2,-1,0,1,2}. This set of numbers is very convenient, as the bit-level multiplication of each number in this set requires at most a 1-bit shift, inversion, or NAND masking when used in a partial product array.

Table 3.3.1 Radix-4 Booth encoding for a multiplier tree

Input[2:0]	Booth Value	Sel2	Sel1	Sign
000	0	0	0	0
001	+1	0	1	0
010	+1	0	1	0
011	+2	1	0	0
100	-2	1	0	1
101	-1	0	1	1
110	-1	0	1	1
111	0	0	0	1[‡]

[‡] This encoding represents "negative" zero. The actual sign of this encoding should have a sign bit equal to '0', but leaving it a '1' allows for easy correction via "hot-ones" and "sign-encoding" within the multiplier tree.

A multiplier tree is created by Booth encoding one of the multiplier input operands into 27 unique encodings. The untouched 53-bit operand is sent to a Booth multiplexer, where it is multiplied by any number in the number range {-2, -1, 0, 1, 2} according to the incoming Booth encoding, shown in Figure 3.3.1. As a result of each Booth encoding, these operations create 27 uniquely multiplied strings, known as the "partial products", which span the numerical range exactly double of the inputs. An array of multiplier partial products is shown in Figure 3.3.2.

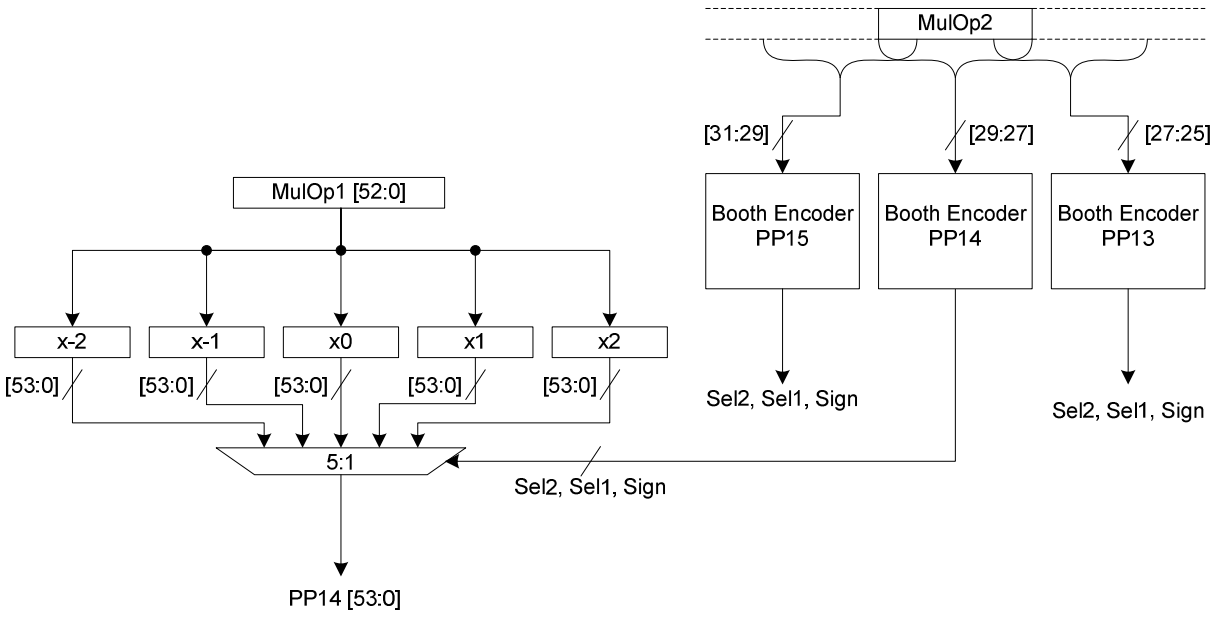

Figure 3.3.1 Booth encoded digit passed to a Booth multiplexer

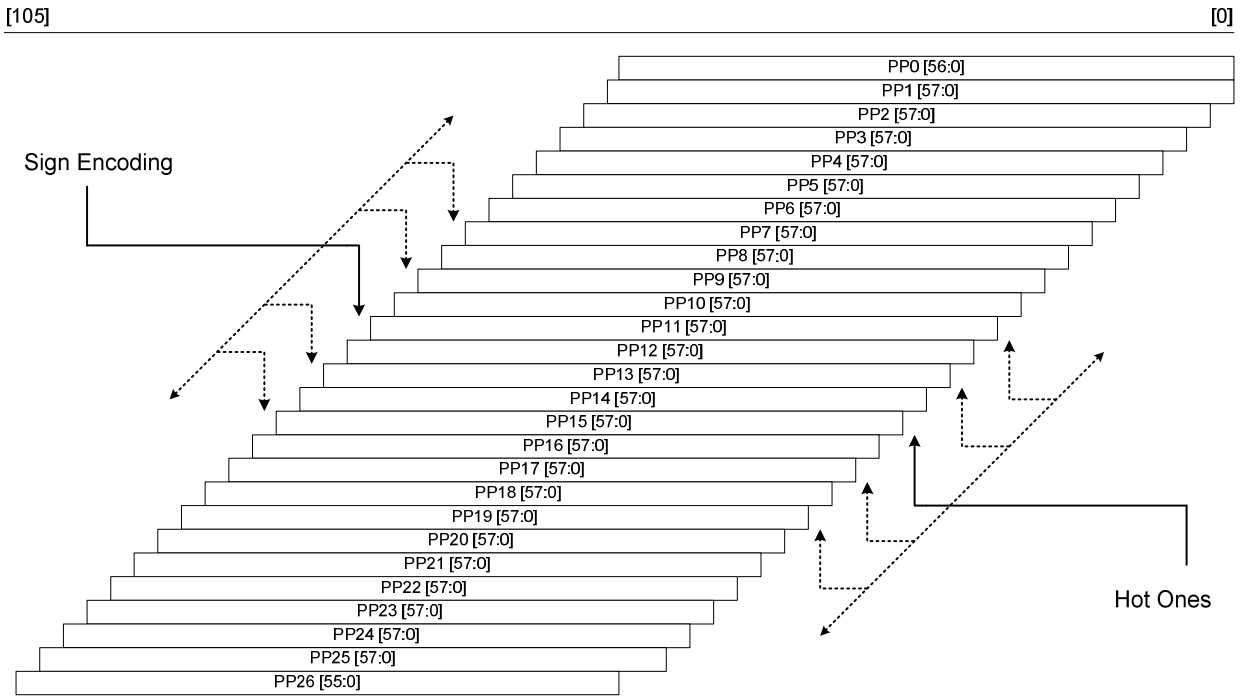

Figure 3.3.2 Multiplier 27-term partial product array

One of the unique problems presented by a radix-4 multiplier tree with a "negative zero" option, as seen in the encoding of '111' in Table 3.3.1 is the problem of correct 2's complementation. If the partial product array is left untouched with this encoding, any negative numbers will produce an incorrect result. To correct this error, the concepts of "hot ones" and "sign encoding" are introduced at the extremities of each partial product in the array.

If a partial product has a Sign bit set to '1', the result is an inversion produced by a Booth multiplexer. To make the product inversion a correct 2's complement, the following partial product term introduces a hot '1' into the LSB position of the preceding term. Additionally, sign bits are appended to the MSB of each partial product term based on the Sign bit of that term itself, allowing a propagation or cancellation of correct carries during compression. Every partial product in Figure 3.3.2 may include a hot one or sign

encoding, save for the first and last products which are special cases. Figure 3.3.3 shows the Verilog code describing a partial product with hot ones and sign encoding.

```
fma_lib_boothmux UU_PP_8 (
                        .M(A_mantissa[52:0]),
                        .PP(PP_8[55:2]),
                        .Sel2(Sel2[8]),
                        .Sel1(Sel1[8]),
                        .Sign(Sign[8])
                        );

assign PP_8[57:56] = {1'b1, ~Sign[8]};   //sign encode
assign PP_8[1:0] = {1'b0, Sign[7]};      //hot one
```

Figure 3.3.3 "Hot one" and "sign encoding" of a partial product

After all the Booth multiplexer selections and 2's complement corrections, the partial products begin compression using large arrays of 4 input, 2 output carry-save adders (4:2 CSA). At each bit position, a 4:2 CSA takes up to 4-bits of partial product inputs, produces a sum on the same bit line, and passes a carry 1-bit position higher. This compression allows each CSA stage to exactly half the number of partial products. For 27 terms, 4 stages are required to produce a product in carry-save, or a carry vector and sum vector that need only be added for a complete multiply.

Although multiplier compression is commonly performed with a 3:2 CSA, the 4:2 CSA was selected specifically for the fused multiply-add design multiplier compression, as a custom circuit 4:2 CSA standard cell has been included in the 'Barcelona' library. This 4:2 CSA compression scheme, as well as the Booth encoding and Booth multiplexer partial product generation all combined as a multiplier is shown in Figure 3.3.4. Finally, the floorplan of the floating-point multiplier tree is seen in Figure 3.3.5 with a color legend in Table 3.3.2.

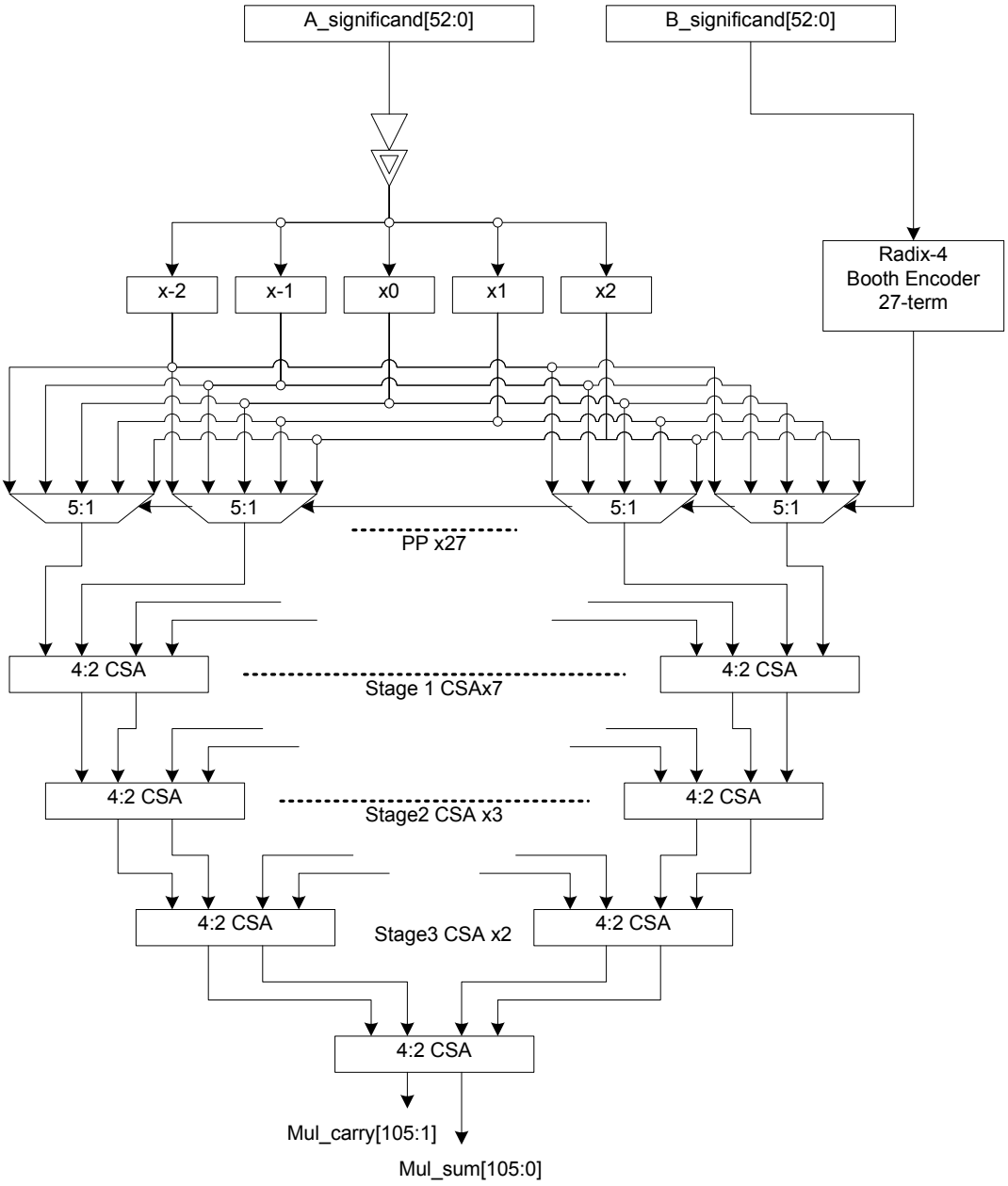

Figure 3.3.4 Floating-point radix-4 multiplier tree

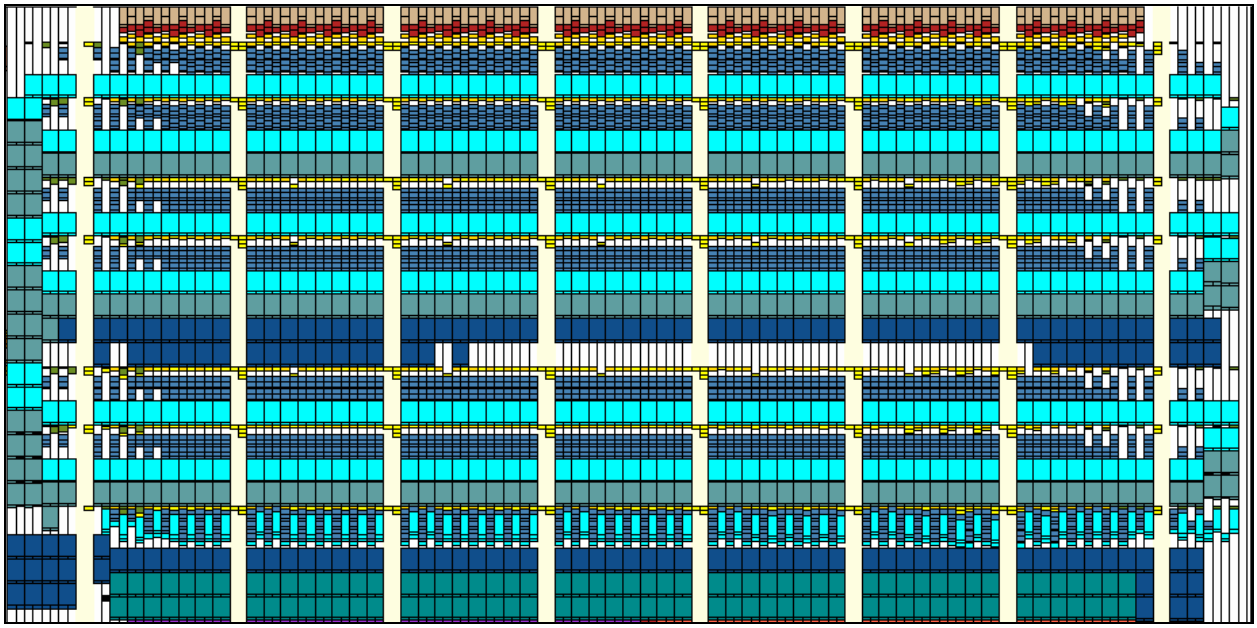

Figure 3.3.5 Multiplier tree floorplan

Table 3.3.2 Multiplier color legend

Color/Shape	Component
Tan/Brown	Booth Encoder
Yellow/Gold	Buffering
Dark Blue (small cell)	Booth MUX
Cyan	4:2 CSA Stage 1
Blue	4:2 CSA Stage 2
Dark Blue (big cell)	4:2 CSA Stage 3
Blue-Green	4:2 CSA Stage 4
Large Horizontal Splits	Clock Rows

3.3.2 Kogge-Stone Adders, Incrementers, and Carry Trees

In any floating-point arithmetic architecture, an array of adders of varying bit-width is always required for numerous functions. For example, a floating-point adder requires not only the main adders in the add/round stage, but also needs smaller adders and incrementers to correctly calculate the exponent in parallel. Floating-point multipliers

require a carry tree, or an adder that calculates only the MSB output, in addition to several other adder structures for exponent, rounding, etc. Since there is such a high adder count for all floating-point arithmetic operations, the adder architecture selected for the units is of great importance.

A popular class of adders used in industrial design today is the “parallel-prefix” family [34]. A parallel-prefix adder is a Ling factored carry-look-ahead style architecture that uses basic AOI/OAI and NAND/NOR components in a one-way parallel and uniform structure. Among the most widely known of the prefix adders is the Kogge-Stone architecture. This architecture, shown in Figure 3.3.6, is a uniform and exponentially decaying tree of propagate/generate (PG) terms that determine if a partitioned number of bits, referred to as the “sparseness” of the tree, see an incoming carry propagation to increment locally added bits. The final PG term arrives as a multiplexer selection signal representing an incoming carry, and the correct sum bits are selected as outputs.

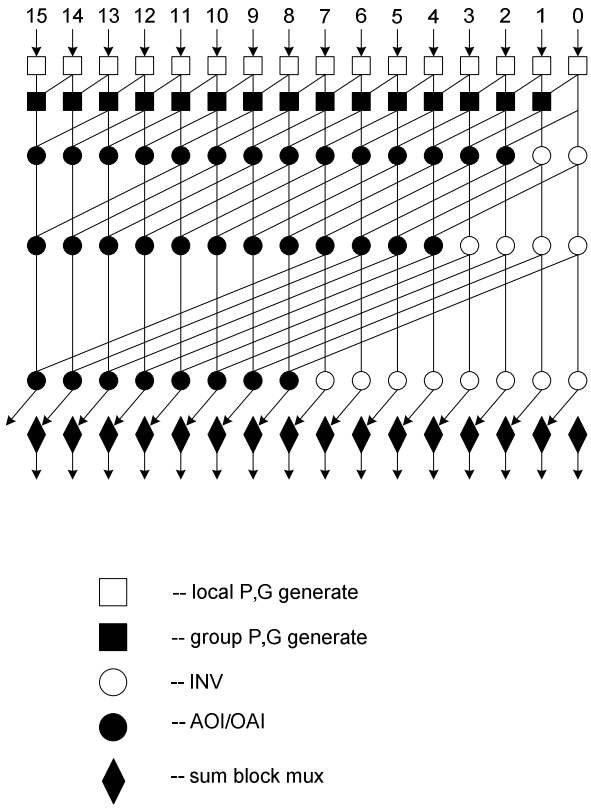

Figure 3.3.6 Kogge-Stone prefix adder and its components [34]

A prefix structure like the Kogge-Stone does not apply only to adders. Incrementers, carry trees, and compound adders may all use the same architecture with different nodes. A Kogge-Stone incrementer uses the exact design from Figure 3.3.6, except that all nodes are only propagate terms. A Kogge-Stone carry tree again uses the same architecture, but only the final left-most term PG term and the inputs that feed it is required (term 15 in the figure). Finally, a Kogge-Stone compound adder uses the architecture, but replaces all white circle nodes with full propagate terms (NAND/NOR), and replicates the sum multiplexer with an augmented sum multiplexer (sum+1) that is selected by the total carry propagate term.

In the fused multiply-add designs, the Kogge-Stone prefix architecture is used for all adders, incrementers, carry trees, and compound adders. The specific design was selected due to its low logic stage count, uniform layout, and ease of design when expanding to multiple bit-widths for multiple functions. While the Kogge-Stone may not be the best selection for every single possible function in every type of floating-point architecture, its consistency allows for a uniform and controlled design conducive to architectural comparisons.

The largest implementation of a Kogge-Stone adder in the fused multiply-add designs is a 109-bit adder in a 1-bit/1-row configuration found in the fused multiply-add Classic architecture, while the smallest is a 13-bit adder used in exponent logic over all designs. All adder designs follow a “snake” style implementation, where a single bit path starts in a row and weaves back and forth from cell to cell, creating a tight-fitted interlacing of cells. Additionally, all adders are designed with the PG Tree spanning the entire top of an adder, with the partitioned sum block cells in parallel at the bottom.

Figure 3.3.7 and Figure 3.3.8 show the 109-bit Kogge-Stone sparse-2 adder in floorplan and block format. Figure 3.3.9 shows the floorplan of a more-compact 52-bit incrementer (The adders shown in this dissertation all use purple as a color base). Following, a 13-bit

adder is shown in Figure 3.3.10 (All adders in the exponent paths use orange as the color base).

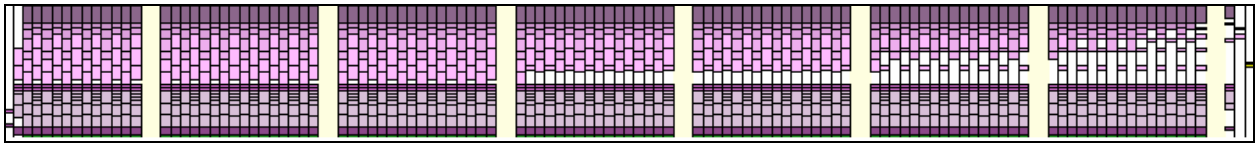

Figure 3.3.7 Kogge-Stone 109-bit adder

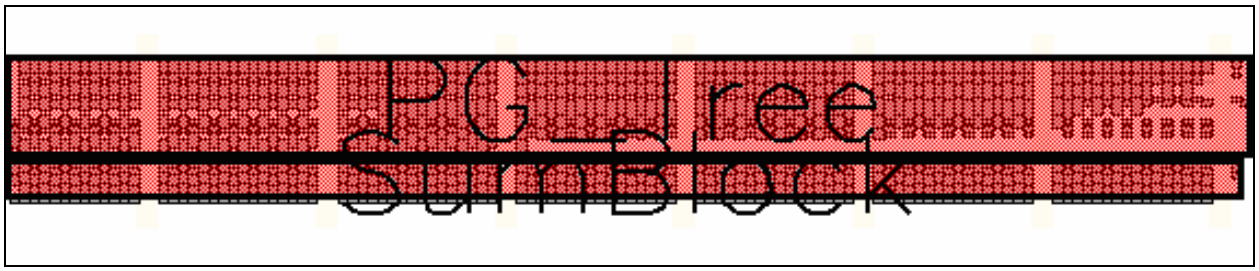

Figure 3.3.8 Block view of the 109-bit adder

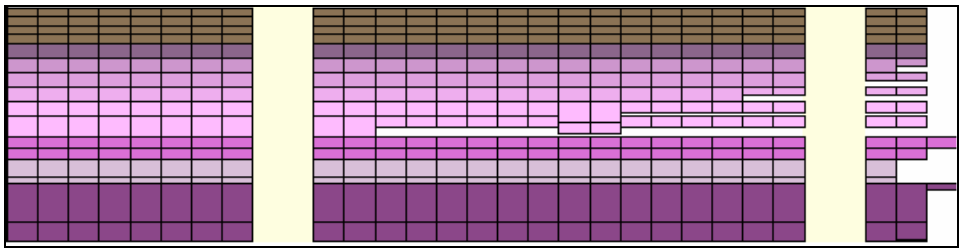

Figure 3.3.9 Kogge-Stone 52-bit incrementer

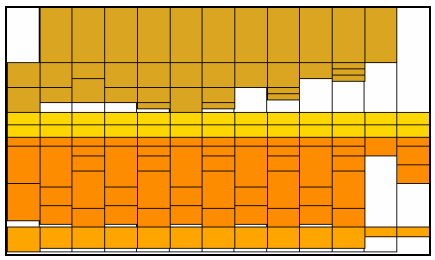

Figure 3.3.10 Kogge-Stone 13-bit adder

3.3.3 Leading-Zero Anticipators (LZA)

A major component in floating-point add and fused multiply-add architectures is the leading-zero anticipator (LZA). This block, commonly used in parallel with an adder, takes two input strings and uses a set of logical equations, cited in Chapter 2.9, to predict the bit position of the leading '1' after a subtraction that causes massive cancellation. When the position is found, the result is encoded with a priority encoder block, and output to the normalization stage.

A small example is provided in Figure 3.3.11 to display a case when a floating-point subtraction operation would use a leading-zero anticipator. In the figure, two 9-bit operands that require a subtraction operation are passed to both the adder and the LZA block. The adder begins its operation, while in parallel the leading one's prediction (LOP) block identifies the foremost '1' as located in the 2^1 position. This bit position is passed to the priority encoder, which encodes the 8-bit input string (the place before the decimal is ignored) into a 3-bit selection control. The priority encoder, in this case, sends a '111' to the normalizer, and the result is shifted left by 7 positions, just as the result from the adder completes. The shift control then updates the exponent logic with the normalization amount.

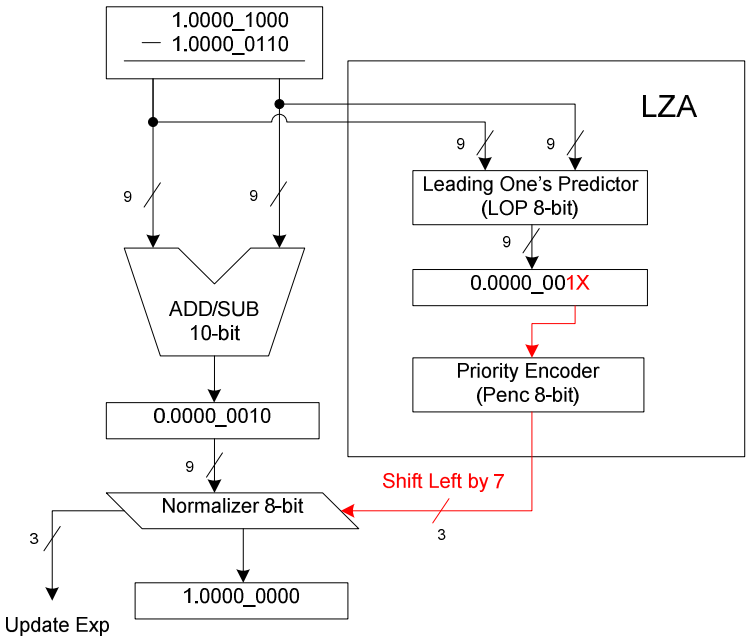

Figure 3.3.11 LZA 9-bit floating-point example

As described by the example, a leading zero anticipator is split into two functional blocks—the leading one's predictor and the priority encoder. The leading one's predictor is a simple block that takes two adder input operands and creates logic terms using the LZA equations coded in Verilog as shown in Figure 3.3.12. The implementation of the LOP terms is elementary, only requiring basic NAND, NOR and XOR cell combinations to generate the prediction terms.

```
// ============================================
// (#1) Leading one's prediction generation
// ============================================

reg [107:0]       LOP_T;
reg [107:0]       LOP_G;
reg [107:0]       LOP_Z;

reg [108:1]       LOP;


always @ * begin

  LOP_T[107:0] = OpA[107:0] ^ OpB[107:0];
  LOP_G[107:0] = OpA[107:0] & OpB[107:0];
  LOP_Z[107:0] = ~(OpA[107:0] | OpB[107:0]);

  LOP[108:1] = { 1'b0,
                 (  LOP_T[107:2] & LOP_G[106:1] & ~LOP_Z[105:0] ) |
                 ( ~LOP_T[107:2] & LOP_Z[106:1] & ~LOP_Z[105:0] ) |
                 (  LOP_T[107:2] & LOP_Z[106:1] & ~LOP_G[105:0] ) |
                 ( ~LOP_T[107:2] & LOP_G[106:1] & ~LOP_G[105:0] ),
                 ~LOP_T[0] };

  //    LOP[0] = 1'b1;

end // always @ *
```

Figure 3.3.12 Leading one's prediction (LOP) equations in Verilog

The second half of the LZA block takes the output vector generated by the LOP block and priority encodes it into control signals based on the first ‘1’ found in the string. The logic making up the priority encoder block is neither uniform nor simplistic, and is one of the more complex pieces to design and implement in a floating-point unit.

The basic building block of a priority encoder used in the fused multiply-add designs is a 4-bit priority encoder cell. As shown in the following equations, encoding a four bit cell will create a 2-bit position vector, named X, as well as a ‘1’s detection signal, named Y. If the LZA desired is only 4-bits, the X variable would be the shift count, and the Y variable could go unused. However, the Y variable serves an important function when expanding the bit count of an LZA.

$$X[1] = \overline{(LOP[3] + LOP[2])} \quad (2)$$

$$X[0] = (\overline{LOP[3]} \cdot LOP[2]) + (\overline{LOP[3]} \cdot \overline{LOP[1]}) \quad (3)$$

$$Y = LOP[3] + LOP[2] + LOP[1] + LOP[0] \quad (4)$$

When priority encoders need to exceed 4-bits using these equations, the best expansion of bit-size is in multiples of 4. As shown in Figure 3.3.13, five total 4-bit priority encoders may be organized to encode a 16-bit LOP vector. Four of the 4-bit encoders are used directly to encode the LOP vector, and the fifth is used to encode the Y signals from each other 4-bit block. The resulting X vector output is the correct high-order bits of the 16-bit encoding, as well as the multiplexer select signals for the low-order bits.

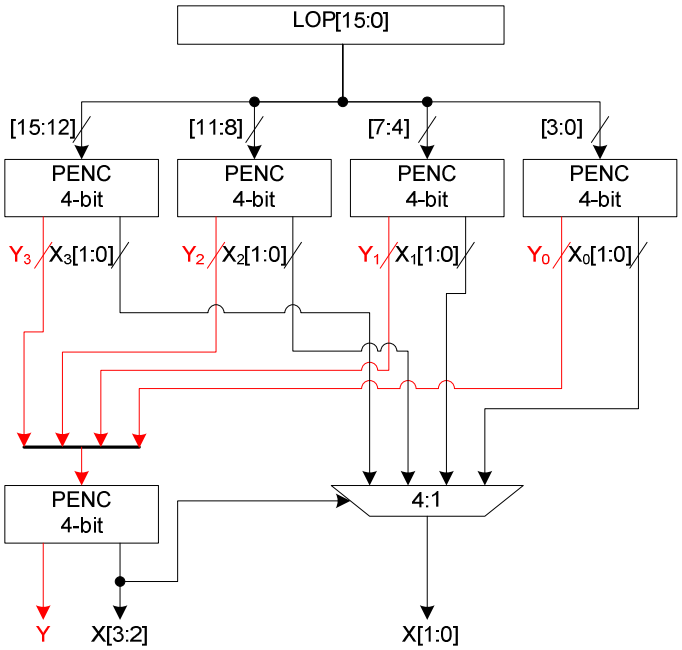

Figure 3.3.13 Priority encoder 16-bit

The next size priority encoder block is 64-bits. As shown in Figure 3.3.14, the 64-bit encoder has exactly the same architecture as a 16-bit encoder, only with larger buses and components. In the case when an encoder less than 64-bits be desired, the lowest order

16-bit priority encoder either need not exist, or be fed a string of ‘0’s to emulate no data in the undesired bit positions.

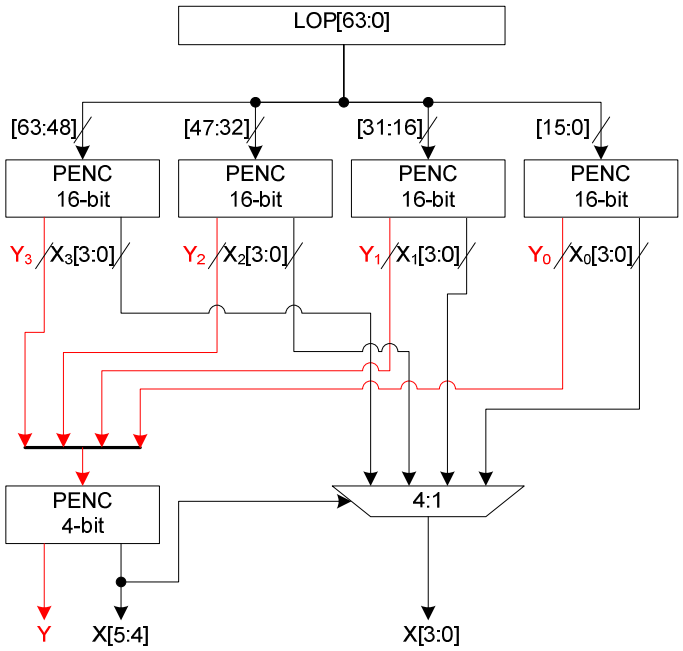

Figure 3.3.14 Priority encoder 64-bit

A floorplanned screenshot and block level of an implemented 57-bit LZA block is shown in Figure 3.3.15 and Figure 3.3.16 (Green is used throughout the dissertation to denote LZA blocks). As shown in the figures, the leading one’s prediction cells are very uniform and capable of high densities, as the components required for each bit in the datapath need exactly the same amount of standard cells. However, the priority encoder, seen in the bottom half of the unit, has unusual sizing requirements and cell counts, and can be at best organized in a pseudo-random spread of gates. While a better organization of the priority cell placements is possible, the selected floorplan has more to do with the requirements of critical path flylines and localized placements than area saving efforts.

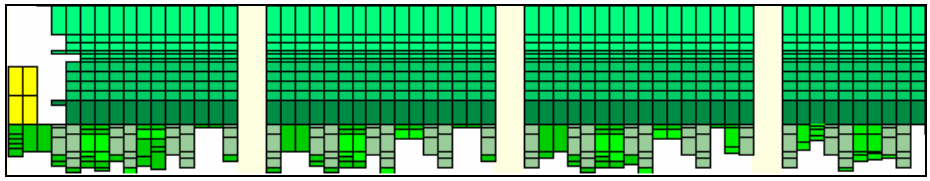

Figure 3.3.15 LZA 57-bit floorplan

Figure 3.3.16 LZA 57-bit blocks

3.3.4 Miscellaneous Components

While there are several other miscellaneous components that make up the building blocks of the fused multiply-add designs, their design and implementation is on a lower-order of complexity than the multiplier, adders/incrementers, and leading-zero anticipators presented in this chapter. The components already described may be built in a variety of architectures and methods, so there is importance to identifying the arithmetic algorithms used to build an original design for architectural comparison.

The remaining components not described include in the Chapter are items such as shifters, sticky trees (large OR trees), buffer fan outs, basic logic operations, and random control logic. Each of these components, along with others not mentioned, does provide a necessary and important contribution to the execution of the implemented circuits. These components are also not without cost in terms of area, delay, and power consumption. However, there are no complex methods to the remaining components and a better understanding of their use and presence will come from the understanding of the architectural design at a higher level.

Chapter 4

References for Comparison: A Floating-Point Adder, a Floating-Point Multiplier, and a Classic Fused Multiplier-Adder

This chapter provides the design and implementation details of a floating-point adder, floating-point multiplier, and classic floating-point fused multiply-adder created using the AMD 65nm silicon on insulator circuit design flow. These units provide the base implementation references against which the new fused multiply-add architectures are compared.

4.1 Introduction

In order to make a fair and relevant proposal of any new architectural circuit design, a comparison of performance must be made between the new architecture and an architecture that is already established. In some cases, no previous architecture exists. Regardless, a new design must provide a solution or improvement to the status quo.

The first step in the fused multiply-add design process was to establish a basis for comparison by designing, implementing, and observing modern arithmetic units. Specifically, a floating-point adder, floating-point multiplier, and floating-point fused multiply-adder have been created and implemented using high-performance architectures. Furthermore, to keep all implementations consistent in design and method, each base unit has been developed in the same environment and technology as the proposed designs.

The following sections of this chapter present the design and implementation details of a double-precision floating-point adder, multiplier, and fused multiply-adder using the AMD 65nm silicon on insulator technology library and circuit design flow. These units

have been created specifically to provide a basis for comparison in performance, area, and power consumption against those realized by the proposed designs.

4.2 Double-Precision Floating-Point Adder

The floating-point adder is one of the most fundamental units used in floating-point co-processors. The unit is designed to take two input floating-point operands and perform an addition or subtraction. The result is rounded according to the IEEE-754 specifications and passed out of the block. In some designs, more common in the x86 market, the floating-point adder also produces an un-rounded result so that the control units may detect special cases, denormals, exceptions, and various other data for trap handling.

A modern day floating-point adder is nearly always designed using the Farmwald dual-path architecture [35]. This general scheme, which has been selected as the floating-point adder design here, splits the addition datapath into two separate parallel cases. As identified by the literature, the hardware required to handle data for operands with large exponent values is different than that for exponents that are very close. Floating-point adders now build two paths to handle these different data ranges, commonly known as the “far path” and the “close path”.

In cases with large exponent differences, or what is called the far path, the operands must align their floating-points via a large shifter before addition or subtraction may occur. In cases with matching exponents, or the close path, a subtraction of very similar numbers may result in massive cancellation, where the resulting number may be much smaller than the original data range and must be normalized.

Figure 4.2.1 shows the top-view architecture of the dual-path floating-point adder that has been designed and implemented as a reference design. This architecture uses the Farmwald split, employing a far path and close path in parallel. When the correct path is chosen by the exponent logic, the selected path sends its data to the combined add/round stage, and both an un-rounded and rounded result are produced.

The following sub-sections provide the details of each block used in this floating-point adder design. Following these descriptions are the complete results of the floating-point adder implementation.

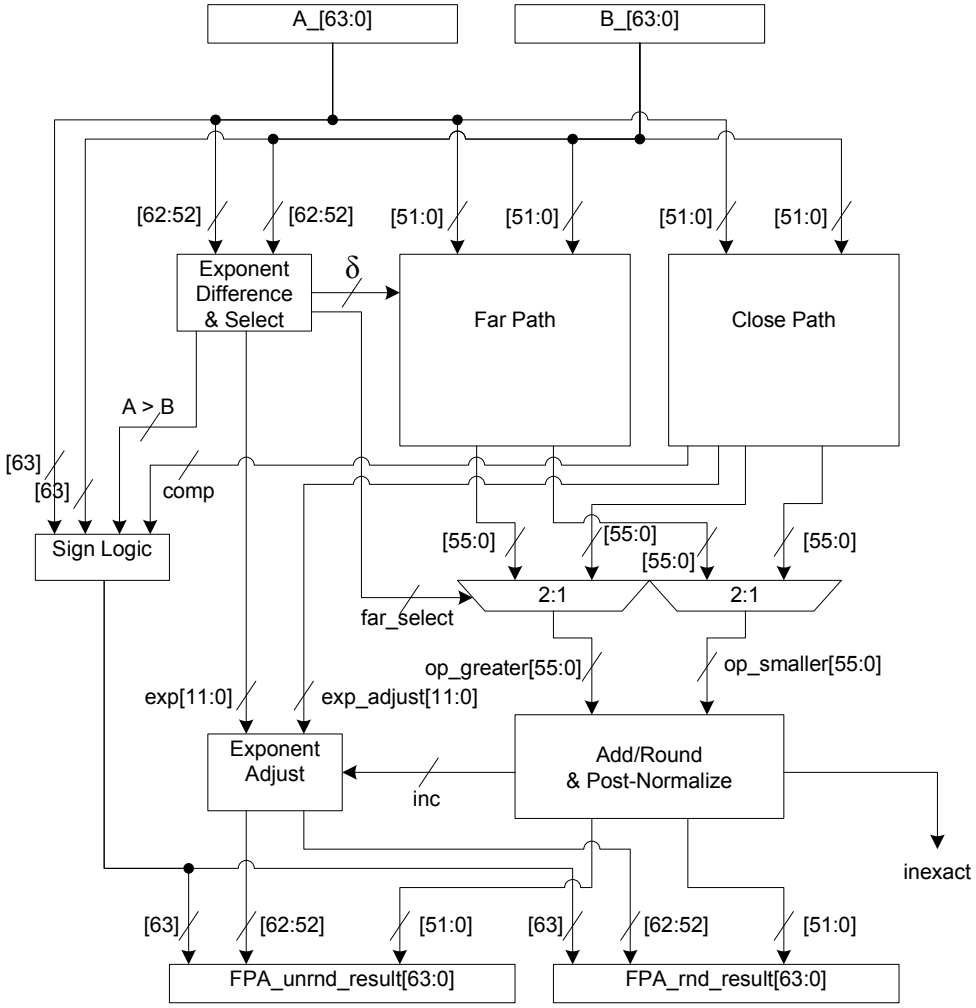

Figure 4.2.1 Double-precision floating-point adder top view

4.2.1 The Far Path

A floating-point adder far path is the significand datapath for all additions and for all subtractions when exponents differ by more than two. This path, shown in Figure 4.2.2, is

set up to determine how far apart the operand exponents are and to align the significands so that correct floating-point addition/subtraction occurs. Additionally, if the smaller operand is out of the range of what these designs refer to as the "anchor," or the larger operand which has a static position, then the smaller operand's data are collected in a sticky bit for rounding.

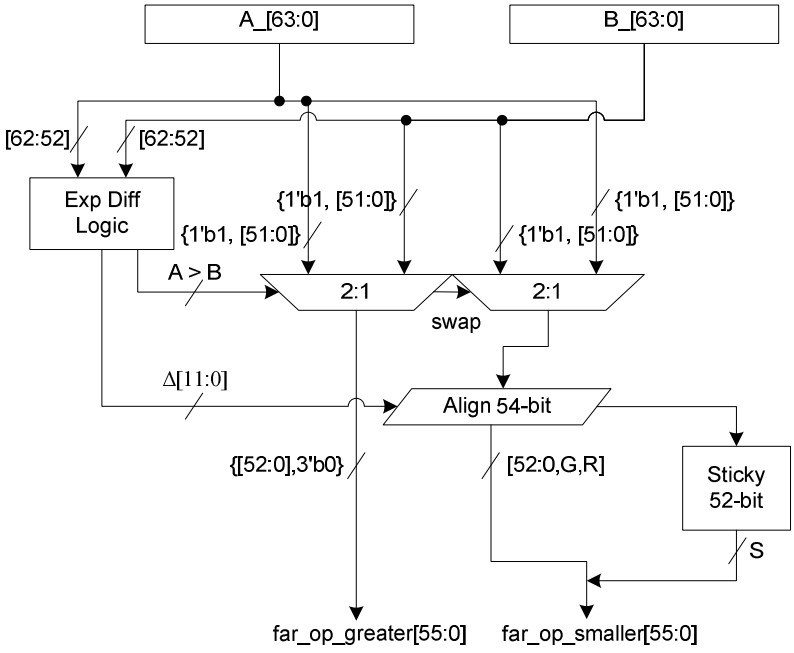

Figure 4.2.2 Floating-point adder far path

The implemented architecture of the far path scheme uses a comparator in the exponent logic to determine which operand is larger. When the large operand is identified, the significands of the inputs pass through the "swap" multiplexer stage, which is the stage that chooses which input is the anchor and which is the one for alignment. In double-precision, the smaller operand enters a 54-bit aligner, and passes any bits that exceed 54-bits to the sticky tree. The stage is complete when the smaller operand is aligned to the anchor and the far path results are passed out of the block.

4.2.2 The Close Path

The floating-point adder close path is the significand data path for all subtractions with operand exponents within the difference range of {-1,0,1}. In this data range, a subtraction may cause massive cancellation, requiring a large normalization before the result may be correctly rounded. Massive cancellation cases, like the floating-point adder design presented here, are commonly handled by leading zero anticipator blocks (LZAs). Chapter 3.3.3 provides a description and example of an LZA block handling massive cancellation.

The close-path architecture is shown in Figure 4.2.3. The input significands are passed to a multitude of blocks, including a swap block, a comparator, and three leading-one's predictors (LOPs). The block begins by determining which exponent, if any, is greater. Once the exponent difference is determined, the operands are swapped, putting the greater exponent in the "greater operand" path. Three LOPs are used: one for A > B, one for A = B, and one for A < B. The exponent control selects the correct LOP at the same time as the shift swap. The operands are sorted and the LOP is sent to a priority encoder.

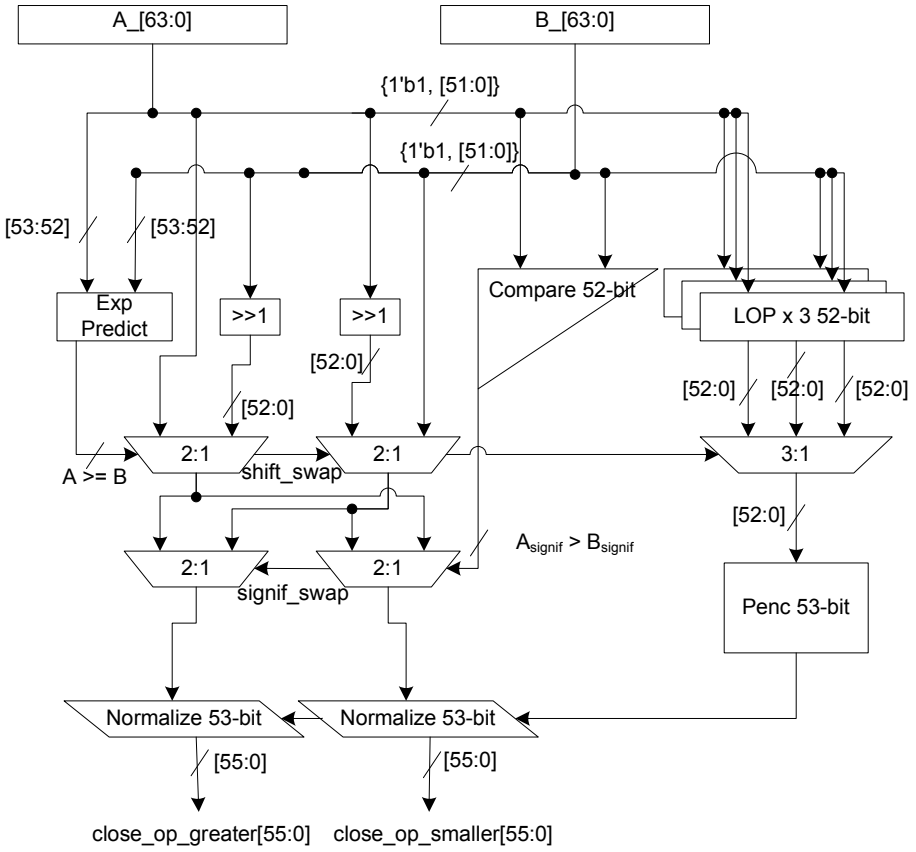

Figure 4.2.3 Floating-point adder close path

However, in the case when the exponents are equal, the greater operand is still unknown. To resolve this, a second swap stage uses a significand compare select to determine the greater operand. When the swap stages are over, the priority encoder pre-normalizes both operands and the results are passed to the round stage. Pre-normalization is valid because cases of massive cancellation will wipe out any leading '1' bits when the numbers are subtracted, so shifting them out early merely saves a stage after the actual subtraction occurs.

4.2.3 The Add/Round Stage

Following the parallel processing of the floating-point adder far and close path, the greater and smaller operands from each merge paths in two parallel multiplexers. Exponent control has by this point determined the correct numerical path, and the operands from the selection are passed to the combined addition and rounding stage.

The add/round stage architectures used in the implemented adder is shown in Figure 4.2.4. The scheme uses two parallel adders, one unbiased and one with a constant, to compute both a rounded and un-rounded case similar to the suggestions by Quach [24], [25], and the implementation of the SPARC64 [21]. The scheme uses the reported concept that a correct IEEE-754 rounded result may be obtained by operating on the LSB in an adder sum or an adder sum + 2.

In this implementation, the two input operands are passed to dual 59-bit adders. One of the adders pre-combines the two operands with a constant: a +2 constant for additions and a +1 constant for subtractions. The MSBs and LSBs of both results are sent to a rounding table, where the correct rounding decision is made based on rounding control. The final rounded significand is selected by the final multiplexer, and both the rounded and un-rounded floating-point addition results are passed out of the block.

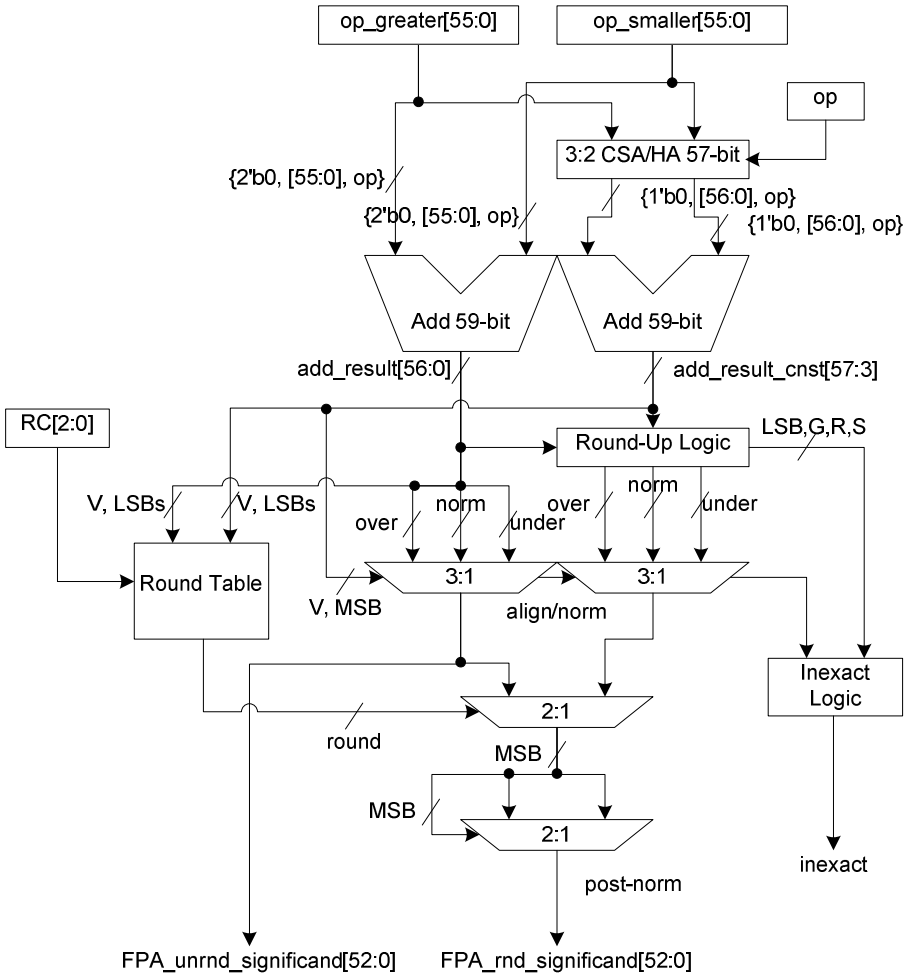

Figure 4.2.4 Floating-point adder add/round stage

4.2.4 Floating-Point Adder Exponent and Sign Logic

The exponent and sign logic in a floating-point adder architecture using a dual-path system is not trivial. The exponent and sign paths must both use parallel prediction paths to calculate their respective results for both far and close path possibilities. Additionally, even after the correct floating-point adder path is known, the exponent logic must continue parallel path processing, as the possibility of an add/round stage overflow or normalization could result in a very late-arriving control signal.

The architecture used in the implemented floating-point adder design is shown in Figure 4.2.5. This block begins by a dual-adder subtraction/comparison of the operand exponents. Two adders, one for $A_{exp} - B_{exp}$ and one for $B_{exp} - A_{exp}$, operate in parallel, with the overflow of $A_{exp} - B_{exp}$ selecting the correct exponent difference. This exponent difference is sent out of the block to the far path swap and alignment stage for immediate use.

When the greater exponent is determined, the operand is selected and passed to another adder to subtract out any incoming close-path normalization adjustment. This difference then splits into four paths which pre-calculate this exponent result by adding {-1, 0, 1, 2}, for all cases of rounding overflow, normalizing, or double overflows. When the rounding stage executes, the correct alignment is known and the exponent result selected.

The sign logic has a simpler path than the exponent operands. Each operand sign bit is passed to parallel logic blocks, each executing under the assumption of either the far path or close path selections. When the path is determined, the correct block is selected and passed to the rounding stage.

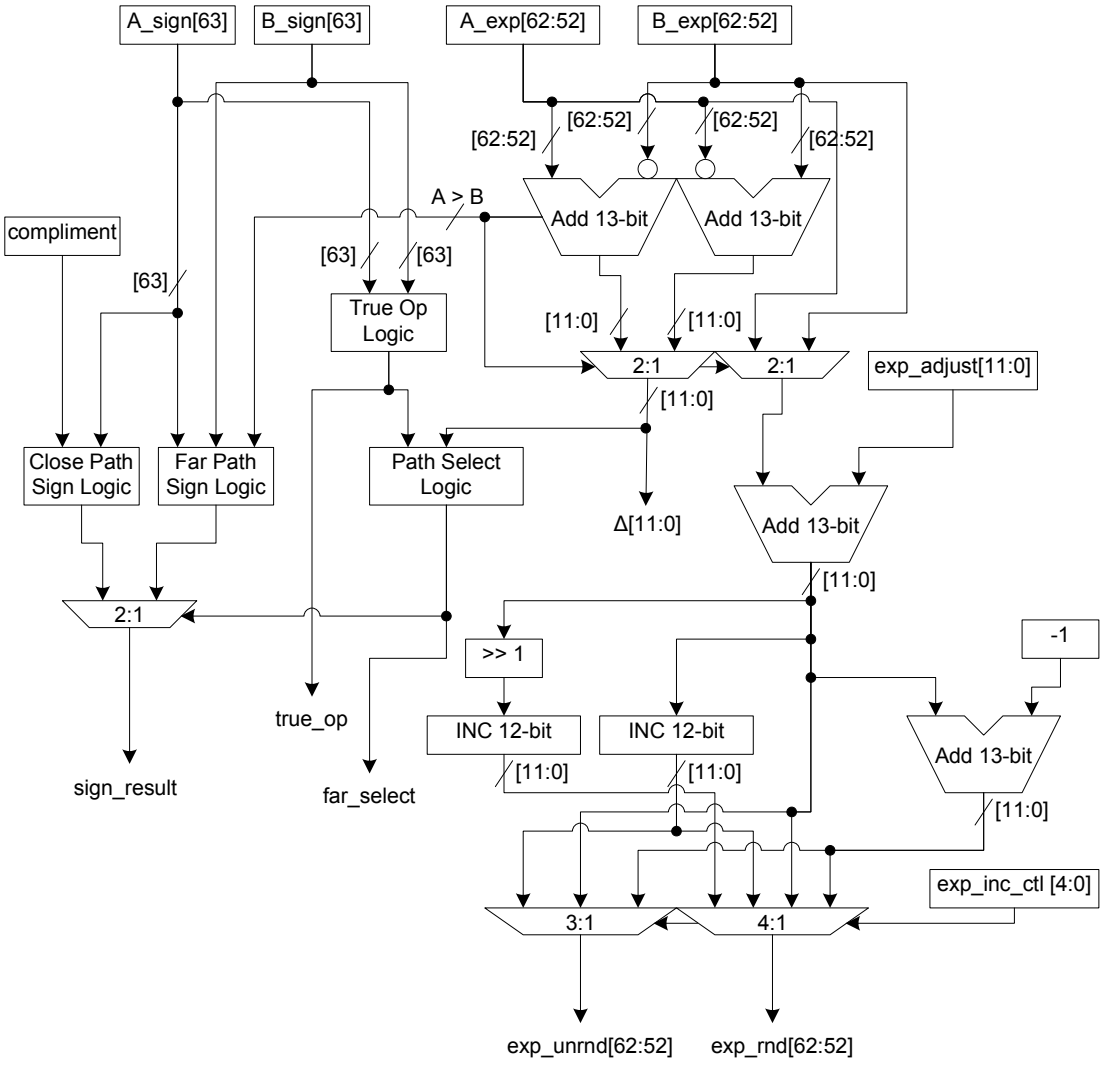

Figure 4.2.5 Floating-point adder exponent and sign logic

4.2.5 Floating-Point Adder Results

The floating-point adder has been designed and implemented with the AMD 65nm silicon on insulator (SOI) technology and design flow. A full floorplan screenshot is shown in Figure 4.2.6 with the floating-point adder in an orientation where data flows from top-to-bottom with bit-positions starting at 63 and going to 0 left-to-right. The data use a pitch of 2-rows / 1-bit to interface with multi RD/WR port register file.

Table 4.2.1 provides the color-key legend for the floorplan in Figure 4.2.6, identifying major components of the floating-point adder.

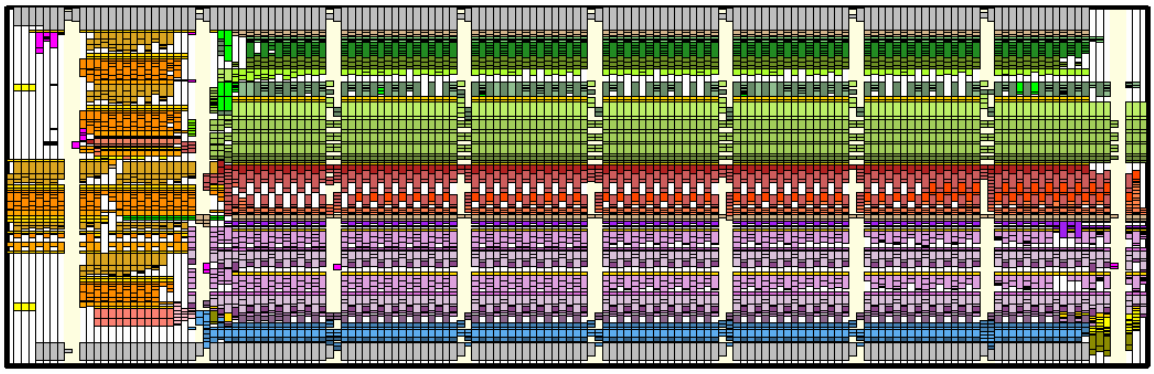

Figure 4.2.6 Floating-point adder floorplan

Table 4.2.1 Floating-point adder color legend

Color	Component
Dark Green	LOP/Penc
Light Green	Dual Normalizers
Red	Aligner
Brown	CSA/HA
Purple	Dual Adders
Blue	Round/Post-Norm
Orange	Exponent

A screenshot of the critical path is shown in Figure 4.2.7 during a circuit simulation at 1.3V 100°C in a typical V_T (TypV_T) process corner.

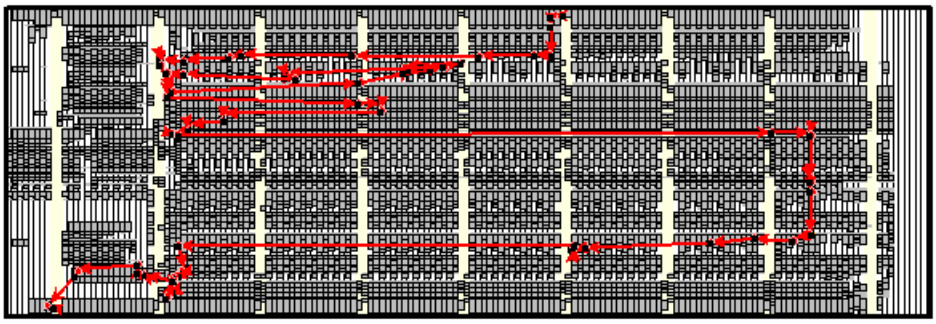

Figure 4.2.7 Floating-point adder critical path

Critical Path: Comp → Penc53 → Normalize53 → Merge → Add59 → Round → ExpInc

Table 4.2.2 shows the results from two timing runs performed at 1.3V 100°C TypV_T and 0.7V 100°C LowV_T respectively. The area calculations come from the actual dimensions of the floorplan, and the power results from HSim power simulations from the floorplan's extracted netlist.

Table 4.2.2 Floating-point adder results

Design	Latency 1.3V 100°C TypV_T	Latency 0.7V 100°C LowV_T	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V TypV_T	Transistors
FPA_DP	946ps	2556ps	155µm x 465µm = 72,075µm^2	118mW	60,526

4.3 Double-Precision Floating-Point Multiplier

The floating-point multiplier is commonly the largest logical block in a floating-point unit, built to take two input operands and provide a multiplied and rounded result. The unit itself, when compared to a floating-point adder, has a simpler overall architecture, but contains very complex components that use large amounts of area and power.

Adding to the floating-point multiplier's size and latency is an array of complex arithmetic functions beyond simple multiplication. A common floating-point unit uses the multiplier to process transcendental, divide, and square root algorithms that use ROM

tables and multiplicative iterations. Without this additional burden, the floating-point multiplier has a very fast performance with low latency. However, with the additional instructions that must be handled by the unit, the latency increases and the unit's cycle count becomes similar to that of a floating-point adder.

The floating-point multiplier described here has been designed without the burden of transcendental, square root, or division algorithms. While these algorithms are necessary in a floating-point unit wishing to comply with the IEEE-754 standard, the design is intended to provide the implementation details of a pure floating-point multiplication instruction without the extra overhead. The proposed fused multiply-add units described in later chapters also do not support these extra functions, keeping the relative calculations consistent in method. These extra functions have been removed in an effort to make the comparison between arithmetic units as direct and straight forward as possible.

The floating-point multiplier implemented architecture is shown in Figure 4.3.1. The unit begins processing data in a 53 x 27-bit radix-4 multiplication tree. The multiplier tree product result passes to a combined add/round stage, where the carry/save product is combined and rounded. The stage outputs both an un-rounded and rounded result, and the floating-point multiplication is complete. Both sign and exponent datapaths run in parallel to the significand processing.

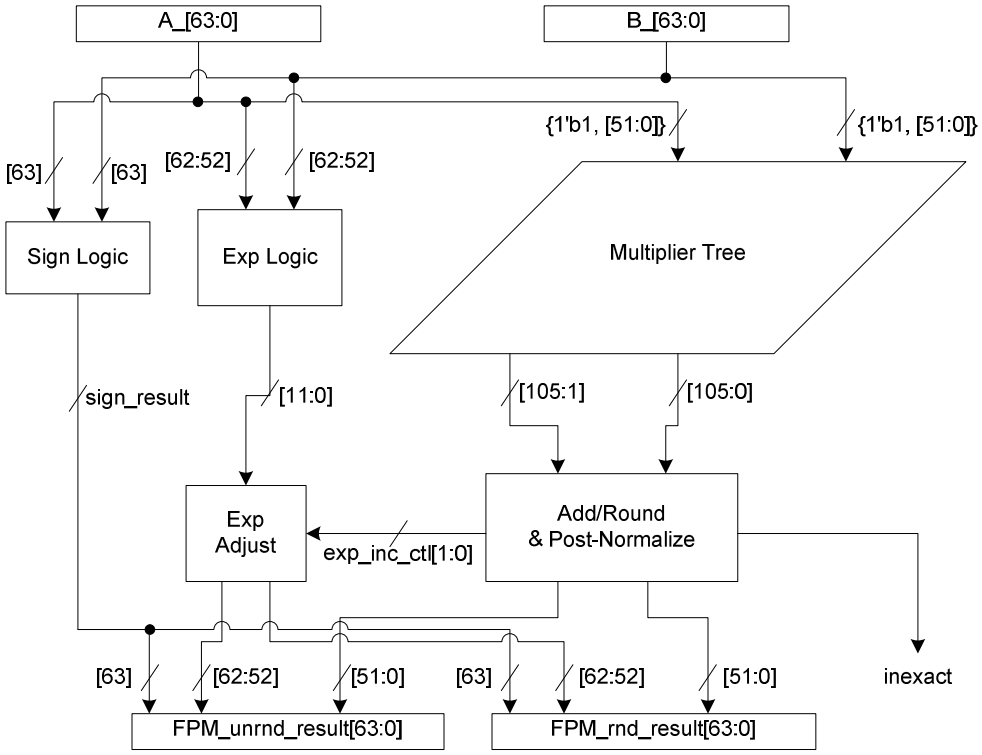

Figure 4.3.1 Floating-point multiplier top view

The following sub-sections describe the details of each major block in the floating-point multiplier from Figure 4.3.1, save the multiplier tree which is already described in Chapter 3.3.1. Following, the results from the implementation floorplanning and simulation are presented.

4.3.1 The Add/Round Stage

The addition and rounding stage in a floating-point multiplier requires a rounding stage more complex than that of a floating-point adder. The multiplier needs unique hardware to produce a rounded result half the size of its input operands, all while correctly propagating data from the parsed lower-half.

The implemented floating-point multiplier stage, shown in Figure 4.3.2, uses an architecture similar to those suggested by [26], [27]. The upper half of the input carry/save product is passed to two half-adder (HA) stages, where the LSB from each

stage is stripped off and sent to a constant 2-bit adder. The remaining upper half enters a compound adder, where the sum and augmented sum (sum + 1) is calculated.

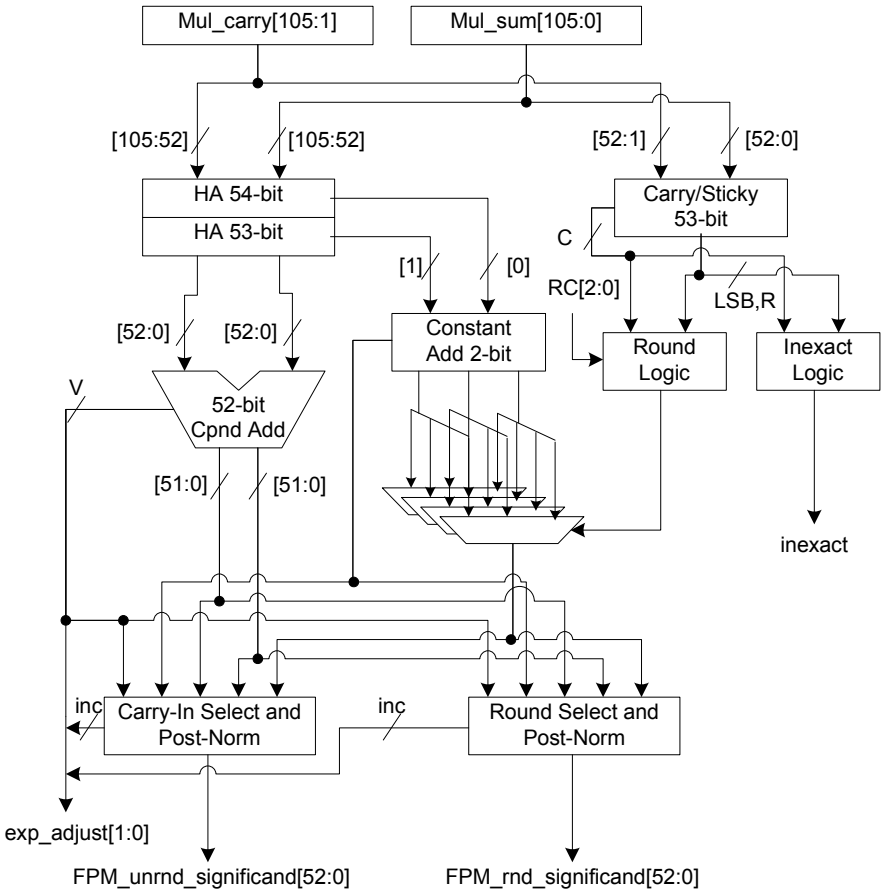

Figure 4.3.2 Floating-point multiplier add/round stage

The lower half of the add/round stage input is sent to a carry and sticky tree, where the LSB, round (R), and sticky (S) bits are produced. These bits combine with rounding control and select the correct increment of the final result's lower 2-bits. Depending on the bit sequence selection of the lower 2-bit constant adder output, the upper half of the result will either be ready for post-normalization or will require the augmented selection. Both the lower 2-bits and the selected compound adder output are post-normalized, and the stage is over.

4.3.2 Exponent and Sign Logic

The exponent and sign logic in a floating-point multiplier is far simpler than that of a floating-point adder. In a floating-point multiplier, the two exponent operands must be added to correctly represent a multiplication. However, since the exponent format specified by the IEEE standard is represented in a form with a *BIAS*, this *BIAS* will become double the value when the two operands are added together (e.g. $[E_A + BIAS] + [E_B + BIAS] = [E_A + E_B] + [2 * BIAS]$). To correct this error, the *BIAS* itself must be subtracted out of the addition.

The implemented floating-point multiplier exponent architecture is shown in Figure 4.3.3. The exponent operands are combined with a negative *BIAS* term, in this case hex C01 for double-precision, in a 3:2 carry-save adder (CSA) followed by a 13-bit add. The result is incremented in anticipation of a rounding increment, and the late arriving increment control signals select the correct exponent un-rounded and rounded result.

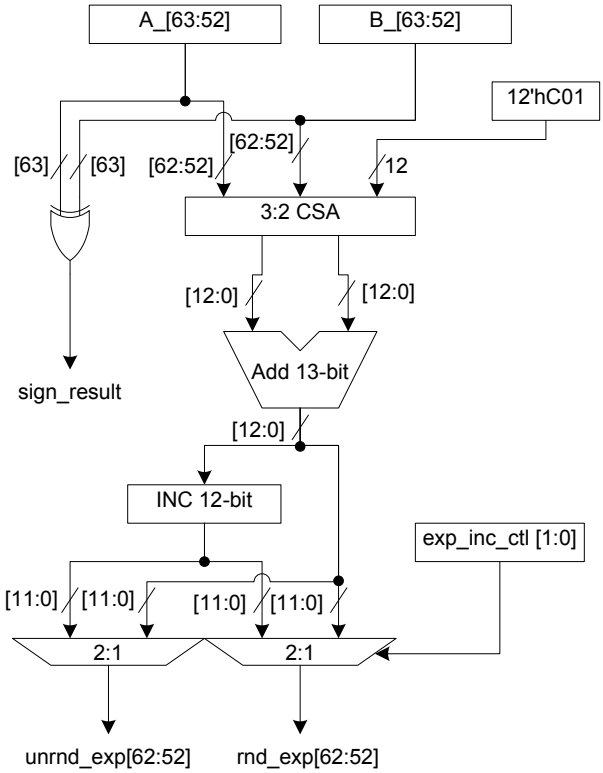

Figure 4.3.3 Floating-point multiplier exponent and sign logic

4.3.3 Floating-Point Multiplier Results

The floating-point multiplier implementation results are presented in the same form as the floating-point adder in Section 4.2.5. The full floorplan of the floating-point multiplier is shown in Figure 4.3.4. Table 4.3.1 provides the color-code legend for the component identification in the floorplan.

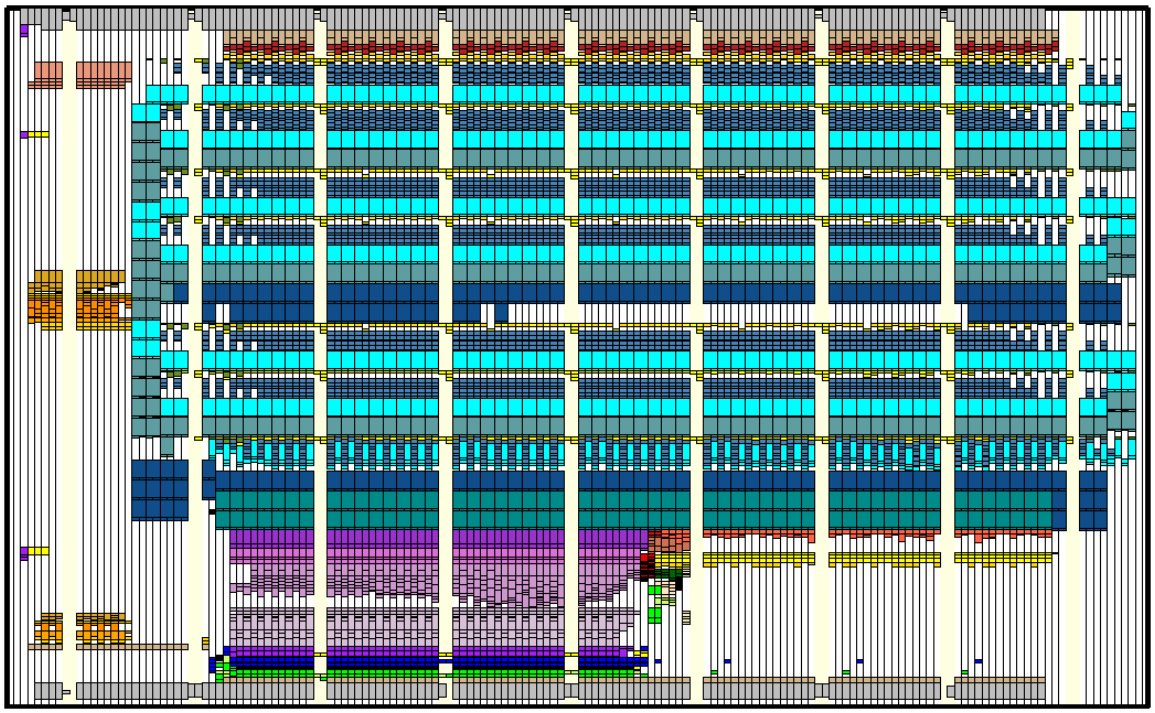

Figure 4.3.4 Floating-point multiplier floorplan

Table 4.3.1 Floating-point multiplier color legend

Color	Component
Brown/Tan	Booth Encoding/Buffering
Blues	Multiplier Array
Purple	Compound Adder
Red	Carry Tree
Yellow	Sticky Tree
Orange	Exponent

The critical path signal from a 1.3V 100°C TypV_T run is shown in Figure 4.3.5.

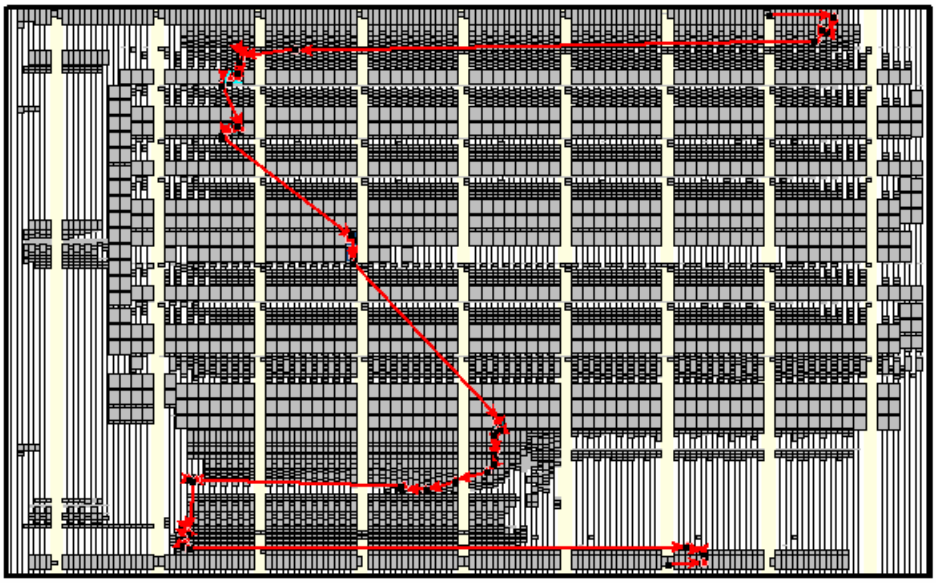

Figure 4.3.5 Floating-point multiplier critical path

Critical Path : BoothEnc → MulTree → 3:2 CSA → Cpnd52 → Post-Norm

Table 4.3.2 provides the timing simulation, area, and power results. The timing results, like the adder, are from a 1.3V 100°C TypV_T corner and a 0.7V 100°C LowV_T corner respectively. The maximum power calculation comes from a HSim floorplan extraction simulation that is held at the same frequency as the simulation from the floating-point adder.

Table 4.3.2 Floating-point multiplier results

Design	Latency 1.3V 100°C TypV_T	Latency 0.7V 100°C LowV_T	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V TypV_T	Transistors
FPM_DP	701ps	1950ps	282µm x 465µm = 131,130µm^2	187mW	125,302

4.4 Double-Precision Classic Fused Multiplier-Adder

The “classic” (IBM RS/6000), serialized floating-point fused multiply-add unit is the centerpiece of this entire design. Since the fused multiplier-adders in this dissertation are new architectures, it is imperative that the comparison base design be both accurate and have the highest performance possible so that the new design data are not skewed with comparative error. The selected architecture for the classic fused multiplier-adder, shown in Figure 4.4.1, is the IBM RS/6000 [1], [2] design with implementation improvements from later builds described in Chapter 2.

The IBM RS/6000 base architecture was selected over newer suggestions found in Chapter 2 due to the practicality and the wide acceptance of the original design. As indicated in the introduction, all major industrial builds of the unit to this day still use the IBM RS/6000 base design, making this architecture the floating-point fused multiply-add standard. Newer suggestions have been too archaic or complex to physically build, so none have been adopted. Since there is no new solution that provides an acceptable deviation from the IBM RS/6000 architecture, it remains the standard for comparison.

The following sections provide a detailed specific look at some of the generalized components shown in Figure 4.4.1. Specifically, the stages from the 161-bit addition component all the way to the add/round stage are generalized in Figure 4.4.1 to simplify understanding, whereas the next few sections show their actual implementation. However, few details are provided on the functionality of the IBM RS/6000 base fused multiply-adder architecture, as the description of a fused multiply-add instruction execution is already provided in Chapter 2.

The section concludes with the implementation results of the classic floating-point fused multiply-adder designed with an IBM RS/6000 base architecture.

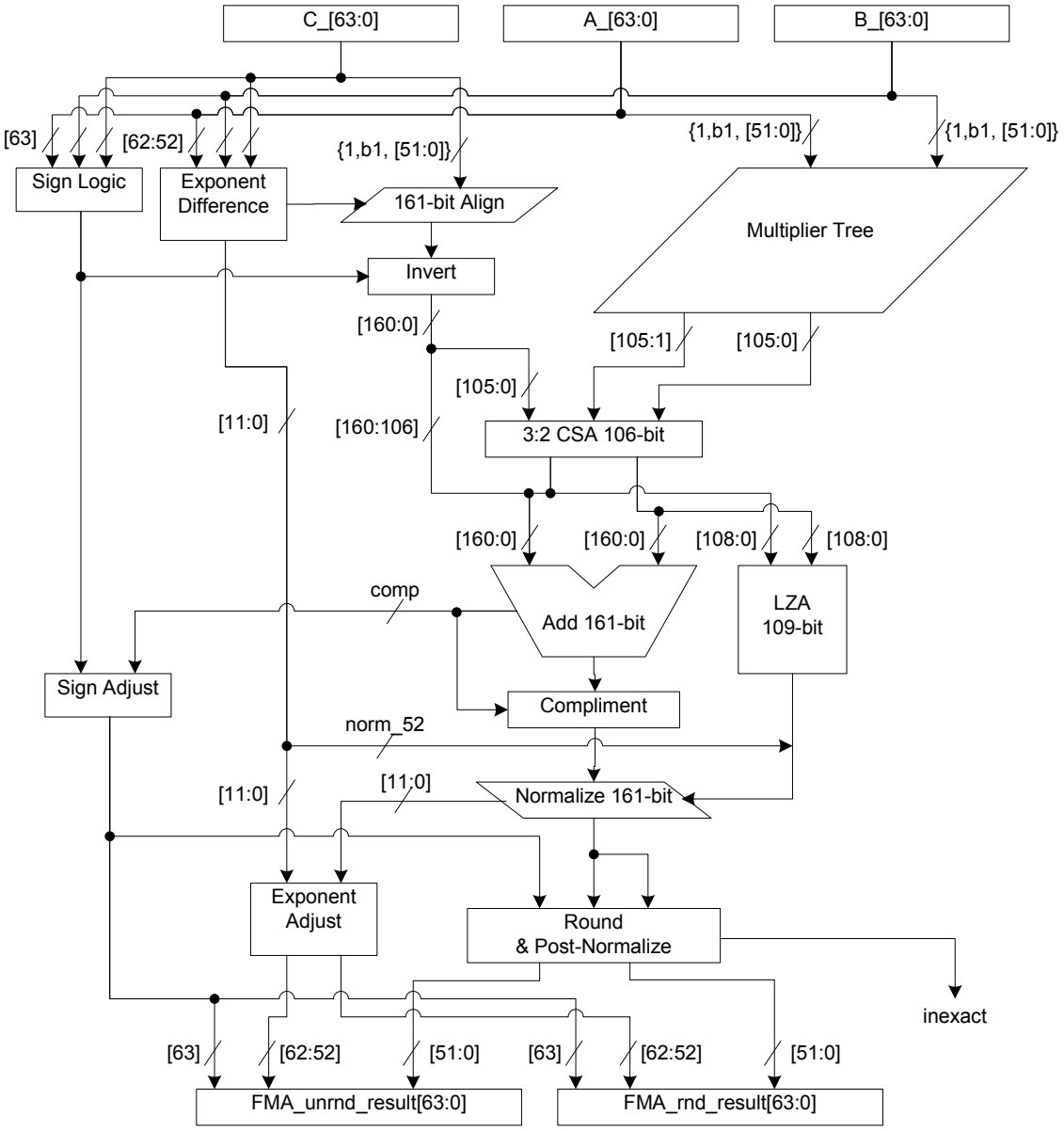

Figure 4.4.1 Floating-point fused multiply-add top view

4.4.1 Addition to Rounding Stage Specifics

The implemented double-precision classic fused multiply-adder uses much of the base IBM RS/6000 architecture at the head of the block. The unit begins with a 53 x 27 radix-4 multiplier in parallel with a 161-bit aligner/inverter used by the third operand. The

lower 106-bits are combined in a 3:2 CSA and passed to a generalized 161-bit adder stage.

The actual implementation of this adder stage splits the datapath into a 109-bit adder and a 52-bit incrementer, as shown in Figure 4.4.2. The carry-out of the 109-bit adder is used to select the correct increment of the upper 52-bits, and the total 161-bit significand enters a 52-bit normalization. This first normalization is a single 2:1 multiplexer selection, as the exponent logic knows by this stage whether the addend or product operand is larger. The correct 109-bit remainder is selected, and the bottom 52-bits fall to sticky.

Following the 52-bit normalizer/selection, the data enter a 109-bit incrementer 2's complement stage. A 2's complement solution has been selected, as it provided less latency in early simulations than a complementation solution that uses an end-around-carry (EAC) 109-bit adder. While this result may be counter-intuitive, as an incrementer requires more stages than a EAC adder, the increased parasitics seen by the scaling of interconnects [3] - [5] in the presence of weak drive strengths from AOI/OAI cells used in prefix adders created a critical path worse than that of the 2's complement method. Therefore, a 109-bit incrementer is used in this scheme.

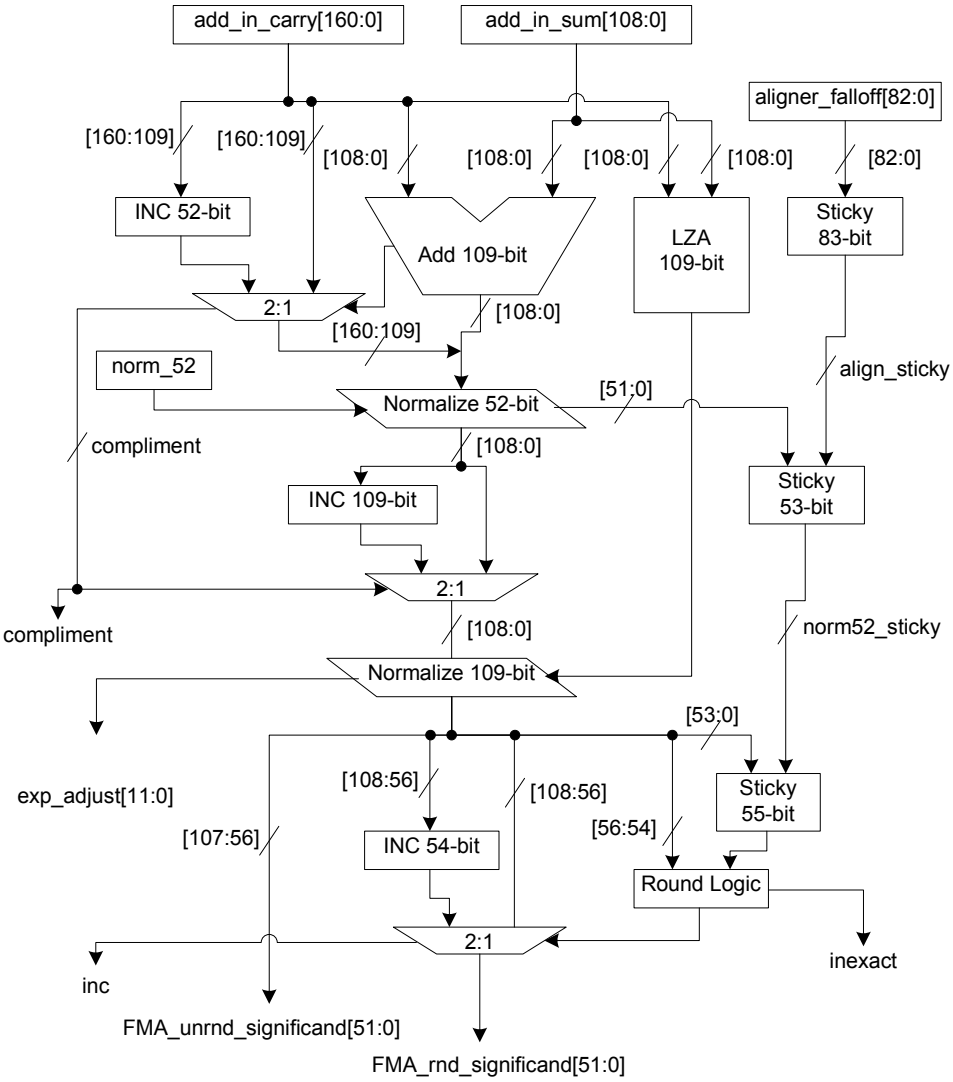

Figure 4.4.2 Floating-point fused multiply-add addition and rounding

The 2's complemented 109-bit vector is passed to a 109-bit normalization stage controlled by the LZA block. The resulting data are split into two paths, with the upper half entering a 54-bit incrementer stage and the lower bits falling to sticky and round logic. A rounding block calculates the round and selects the correctly rounded fused multiply-adder output, completing the instruction.

4.4.2 Exponent and Sign Logic

The exponent logic implemented in the double-precision fused multiply-adder is far more complex than either the stand-alone adder or multiplier schemes. Several paths are required for pre-computation and large normalization values may happen in a variety of cases. However, the sign logic for the stage is rather simple, following that of a floating-point adder.

The fused multiply-add exponent scheme, shown in Figure 4.4.3, requires three separate parallel calculations at the beginning of the block. The first calculation required is the exponent difference between $A * B$ and C to provide an alignment control to the 161-bit align stage. As described in the floating-point multiplier exponent section, the exponent sum for the product $A * B$ must be offset by the *BIAS*, which is done here. Additionally, the C exponent begins un-aligned 55 places above the multiplier product, so this range must be added in for a correct alignment value. Finally, the C value is inverted, and requires a 2's complement—adding 1. Therefore, the final equation of the first pipe is $A_{exp} + B_{exp} - C_{exp} + 55 + 1 - 1023 = A_{exp} + B_{exp} - C_{exp} - 967$.

The second required calculation is the difference of the exponents for the product $A * B$ and C without alignment, so that a comparator flag may be generated for path selection. For this combination, the *BIAS* must still be subtracted off and the 2's complement still added. This calculation equation is $A_{exp} + B_{exp} - C_{exp} + 1 - 1023 = A_{exp} + B_{exp} - C_{exp} - 1022$.

The third calculation is the exponent value of the product $A * B$ itself. When the correct path is found from the comparator exponents, the base exponent value for the final solution will be either that of $A * B$ or C. This correct value, when selected, adds to a normalized alignment value from the 161-bit aligner control and waits for the normalization stage.

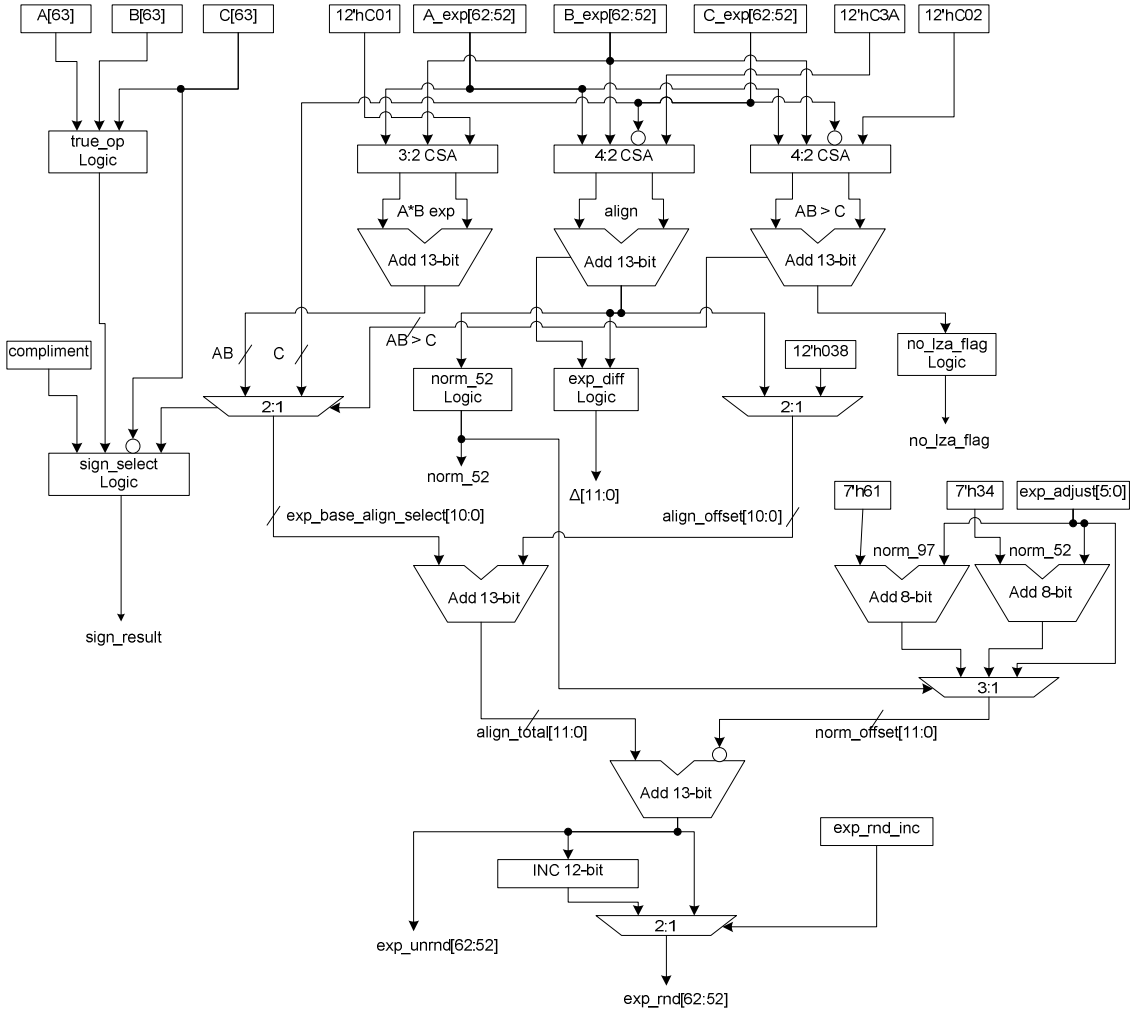

Figure 4.4.3 Floating-point fused-multiply add exponent and sign logic

The two normalization stages and the LZA from the significand datapath send a combined control to adders used for a normalization offset needed in the final exponent calculation. First, a 52-bit normalize may occur, requiring a fixed value to be added to any LZA shifting. Second, the 109-bit LZA is split into two 64-bit halves, with the first half only really consisting of 45 bits. This 45-bit shift is easy to detect, and can also be added to the 52-bit constant for another normalization option. The remaining 6-bits of LZA control are added into the selected constant, and a normalization offset is calculated.

The exponent stage finalizes by subtracting the normalization constant from the aligned exponent value, and an incrementer/multiplexer stage makes an adjustment for rounding overflows.

4.4.3 Floating-Point Classic Fused Multiplier-Adder Results

The double-precision floating-point classic fused multiplier-adder results are presented in the same format as that of the multiplier and adder. The classic fused multiplier-adder floorplan is shown in Figure 4.4.4. Table 4.4.1 provides the floorplan component color legend.

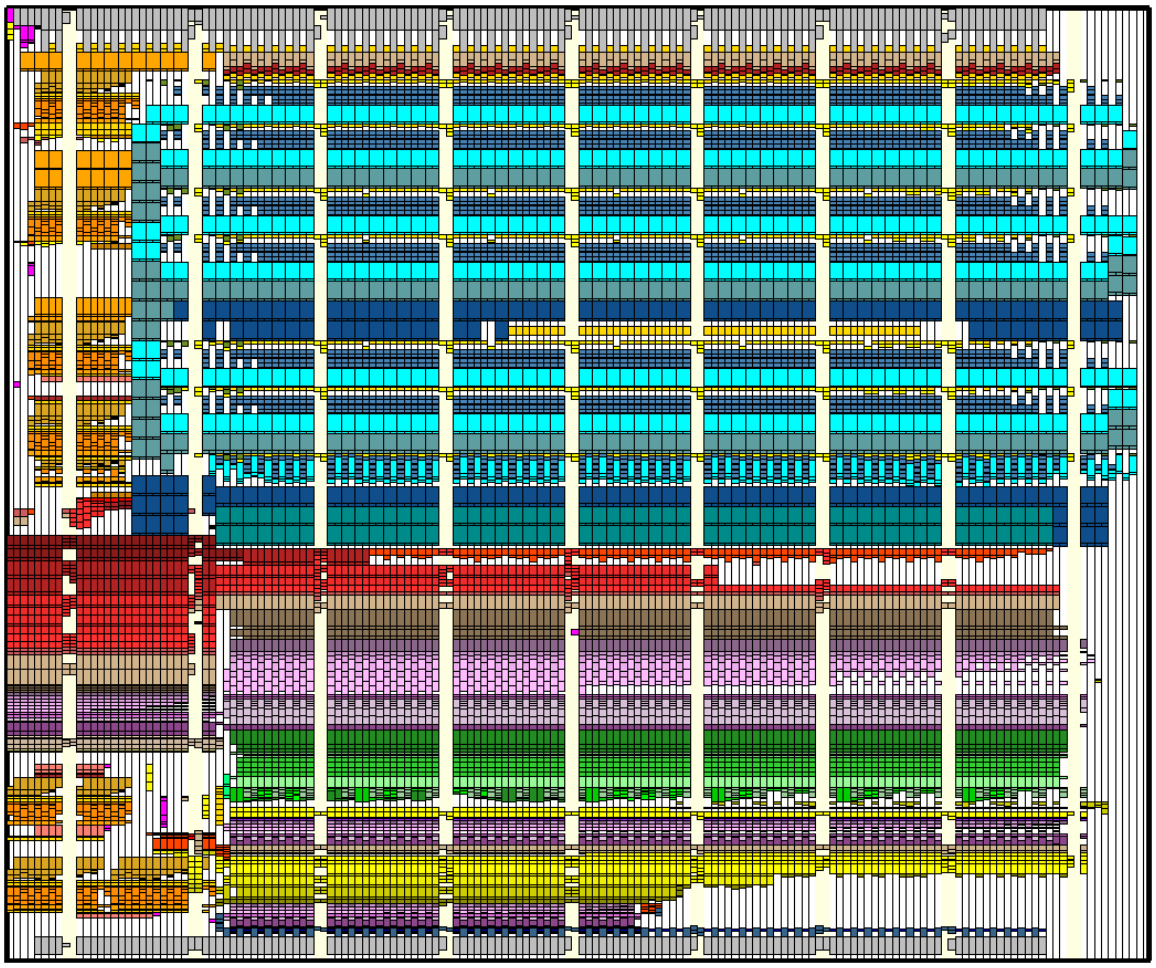

Figure 4.4.4 Floating-point classic fused multiply-add floorplan

Table 4.4.1 Floating-point classic fused multiply-add color legend

Color	Component
Brown/Tan (top)	Booth Encoding/Buffering
Blues	Multiplier Array
Red	Aligner
Brown/Tan (mid)	3:2 CSA
Purples	Adders/Incrementer
Green	LZA
Yellow	Normalizer
Orange	Exponent

The floating-point classic fused multiplier-adder 1.3V 100°C TypV_T critical path is shown in Figure 4.4.5.

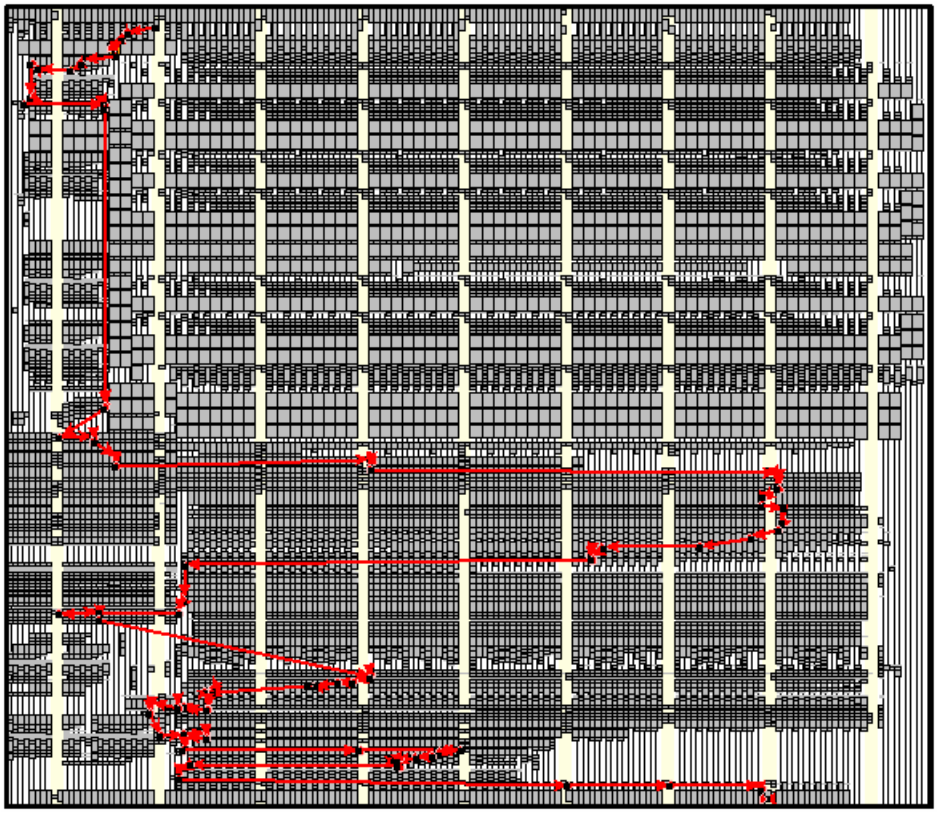

Figure 4.4.5 Floating-point classic fused multiply-add critical path

Crit Path: ExpDiff → Align161 → 3:2 CSA → Add109 → Inc52Sel → Norm52 → Comp → Norm109 → Inc54 → Post-Norm

Table 4.4.2 shows the results of the 1.3V 100°C TypV_T and 0.7V 100°C LowV_T timing runs, area calculation, and maximum power consumption at the normalized frequency. All simulations have been performed with AMD 65nm SOI technology.

Table 4.4.2 Floating-point classic fused multiply-add results

Design	Latency 1.3V 100°C TypV_T	Latency 0.7V 100°C LowV_T	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V TypV_T	Transistors
FMA_Classic	1224ps	3363ps	402µm x 465µm = 186,930µm^2	416mW	177,338

Chapter 5

The Three-Path Fused Multiply-Add Architecture

This chapter provides the design and implementation details of a new floating-point three-path fused multiplier-adder created using the AMD 65nm silicon on insulator circuit design flow. The three-path architecture shows an approximate reduction of 12% in latency as well as a reduction of about 15% in power consumption relative to the classical fused multiply-adder.

5.1 Introduction

Since its public introduction in 1990, industrial implementations of the floating-point fused multiply adder have seen little architectural change from the original IBM RS/6000 [1], [2]. As reported in Chapter 2, several proposals for the improvement of fused multiply-add execution units have been made. However, these new proposals for the reduction of latency or power consumption have either never actually been implemented, or were built at the cost of a loss in arithmetic precision and original functionality.

This chapter presents the design and implementation of a new architecture that reduces both the latency and power consumption of fused multiply-add instructions without any loss in functionality or precision. This new three-path fused multiply-add unit uses parallel hardware paths designed to reduce latency by returning to the floating-point arithmetic fundamentals presented in Farmwald's dual-path floating-point adder [35]. The new architecture's parallelism is not an attempt to force a fused multiply-add into a floating-point adder dual-path system, as suggested by previous works. Instead it uses Farmwald's analysis (not implementation) for different data cases, which logically leads

to a three-path system for a fused multiply-add design. Finally, a three-case hardware system that turns on paths for selected arithmetic data ranges provides a unique opportunity for power savings. In this design, only one of the three paths is ever turned on for any possible instruction.

The following sections include the design philosophies and architectural details of the floating-point three-path fused multiply-adder, including an architecture with an optional floating-point multiplication bypass. After the architectural description, a results section presents the implementation results of the three-path fused multiply-adder that has been built in AMD 65nm silicon on insulator (SOI) technology. The chapter finalizes by comparing the three-path results against the classic fused multiply-adder implementation presented in Chapter 4. When compared, the three-path architecture shows about a 12% reduction in latency and about a 15% reduction in power consumption relative to the classical fused multiply-adder design.

5.2 Three-Path Fused Multiply-Add Architecture

The design concept of the three-path fused multiply-add architecture followed a complete study of the classic fused multiply-add architecture, implementation, and critical paths, as described in Chapter 4, as well as a full review of the academic literature, seen in Chapter 2, with its push for fused multiply-adder parallelism. The conclusion from this extended study on fused multiply-adder units is that a new multiply-add architecture should consider several basic design philosophies for architectural improvement:

1. Parallelize fused multiply-add hardware so the data are not subjected to prediction stages that, for most cases, provide little computational benefit.
2. When designing parallel hardware paths, follow the basic concept of the Farmwald dual-path floating-point adder [35].
3. Fix the fused multiply-add wire dominance problem [3] - [5] by reducing the size of the critical path components or by completely removing them.

4. Use the front-end multiplier CSA delay as an opportunity to pre-compute the necessary parallel fused multiply-add path.
5. Design the parallel hardware paths so that they are logically exclusive, allowing for path pre-computation logic to shut them down in a power-reduction effort if incoming data will not use them.

The three-path fused multiply-adder shown in Figure 5.2.1 has been designed to follow these guidelines. The global design splits the data-path following the CSA multiplier array into three case specific blocks, each designed with different data "anchors." This partitioning of anchor cases removes the need for a massive aligner as well as a complementing stage. Instead, the design partitions alignments and inversions at local levels.

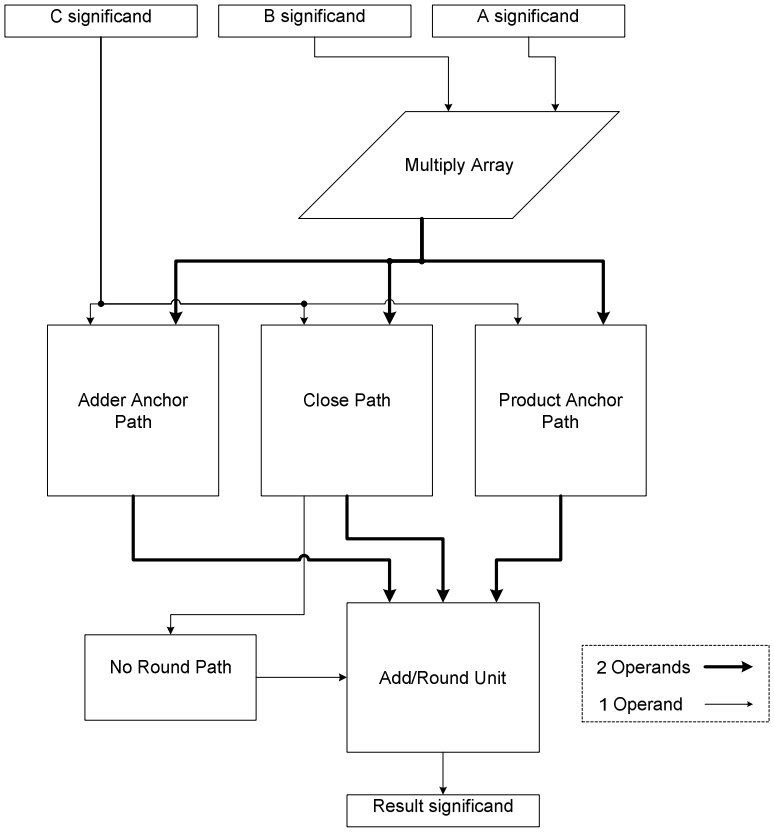

Figure 5.2.1 Three-path fused multiplier-adder architecture

Following the path selection, the appropriate block processes and prepares the numerical data for a combined add/round stage. As in many modern arithmetic unit designs, a combined add/round stage removes the requirement for a massive adder followed by another addition/increment unit for the purpose of IEEE-754 compliant rounding.

The specifics of each path, as well as an explanation of the selected add/round scheme, are described in detail in the following sections.

5.2.1 The Anchor Paths

The three-path fused multiply-adder uses two "anchor" paths for data dependent processing. As shown in Figure 5.2.1, these two blocks are the adder anchor path and product anchor path. The use of the term "anchor" is a reference to the design philosophy found in the Farmwald floating-point adder designs.

As seen in Chapter 4, a dual-path floating-point adder uses a "far" path that always begins by finding the larger number and locking its position, i.e., using it as an "anchor." Once the larger operand is known, the second operand may be aligned and inverted in the case of subtraction without ever needing a corrective complement.

For the case of a fused multiply-add unit, a similar use of a far path is not feasible, as the range of positions in double-precision format spans 161-bits as compared to 52-bits in a floating-point adder. Additionally, if such a system were applied to a fused multiplier-adder, both 161-bit ranges of addition and product operands would need to have the option of inversion and swapping. Applying this system to something as massive as the fused multiply-add data range is not realistic.

A better solution for dealing with the fused multiply-add data range is by splitting the anchor-based algorithm into two cases. To start, a benefit of a fused multiply-adder unit is that the exponent difference is known well ahead of the significand product, so a logical data-range may be selected early in the circuit. In cases of large exponent

differences, either the addend or the product will be larger and always without ambiguity. The three-path fused multiply-add unit design takes this early-known exponent difference and anchors whichever operand is larger, forcing the other operand to invert and align. This anchoring method requires partitioning of the 161-bit data range into two smaller sets.

Figure 5.2.2 shows the adder anchor path in detail. This path is selected when the exponent difference detects that the addend is larger (specifically by >= 1 for additions, and > 2 for subtractions). For this case, the addend is anchored. The later arriving product terms are then aligned over a 57-bit range and inverted for subtracts. Following inversion stages, all three operands are combined in 3:2 carry save adders (CSAs) or half adders (HAs) to produce two 163-bit numbers. The most significant bits of both results are used for corner-case correction, and the lower 55-58 bits are sent to a carry/sticky tree, as the least significant bits will never be selected in the final result.

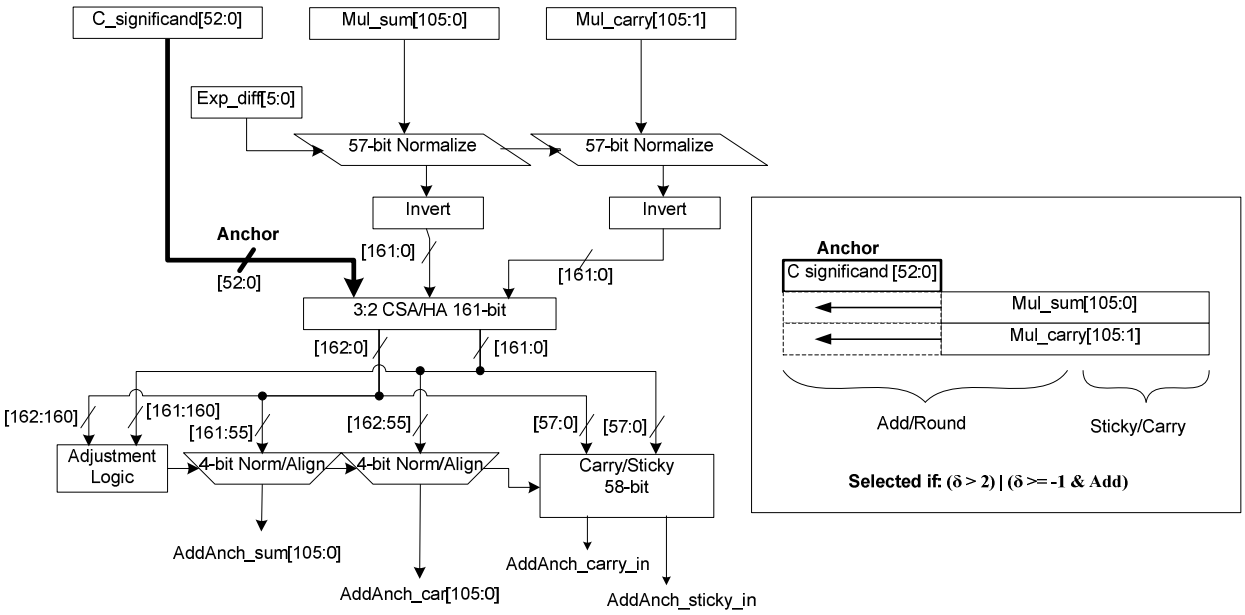

Figure 5.2.2 The Adder Anchor Path

The adder anchor path finalizes with two 106-bit operators ready for addition and rounding as well as an input carry and sticky bit generated by the discarded lower bits. The adder anchor unit is not on the critical path, so there is sufficient time to normalize the product terms over a 57-bit range.

Figure 5.2.3 shows the product anchor path. This path is the complement of the adder anchor path and is enabled when the exponent difference determines that the product is larger than the addend (specifically by >1 for all operations). Much like the classic fused multiply-add, the addend is aligned and inverted against the position of the product. However, in this design the data need only cover a 106-bit range as opposed to the original 161-bit range. When the product terms arrive from the multiplier, all the data are combined in a 3:2 carry save adder (CSA), adjusted and sent in 106-bit sum/carry form to the add/round stage.

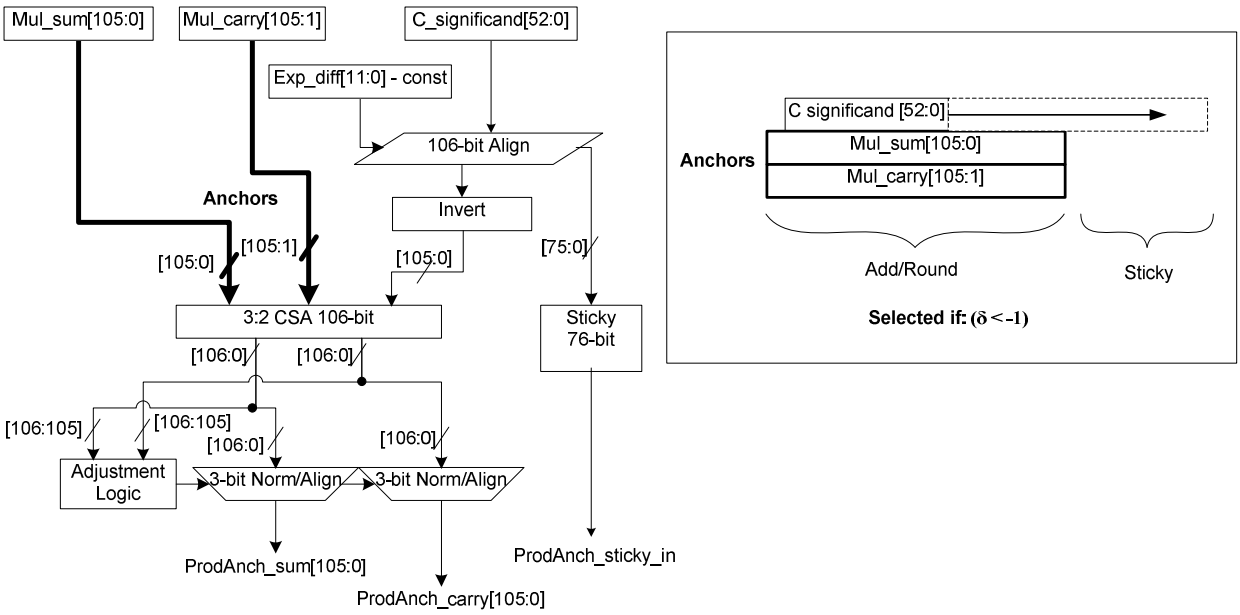

Figure 5.2.3 The Product Anchor Path

5.2.2 The Close Path

For cases when the exponent difference between the addend and the product is too close to easily determine a larger operand, all data are passed to the close path. This path only handles fused multiply-add subtraction operations and is geared specifically for massive cancellation.

To follow suit with the two anchor paths, the close path is designed to remove the requirement of a complementation stage. As shown in Figure 5.2.4, the close path accomplishes this via significand swapping. First, the path uses 3:2 CSAs and HAs to combine an inverted aligned addend with the product. Likewise, the logically opposite term is also created with inverted product operands and an un-complemented adder term.

The first 3:2 combination is passed to a 57-bit comparator (57-bits is selected since all bits after position 57 in the aligned adder term are always ‘0’s) to determine which operands are larger. The comparator signals the swap multiplexers to choose the correct inversion combination and the results are normalized in preparation for addition and rounding. The LZA that controls this normalization is passed only one combination of inversion inputs, as its functionality is not affected by which operand is larger. Depending on the addition/rounding scheme selected, the one-bit LZA correction shift may be handled in the add/round block.

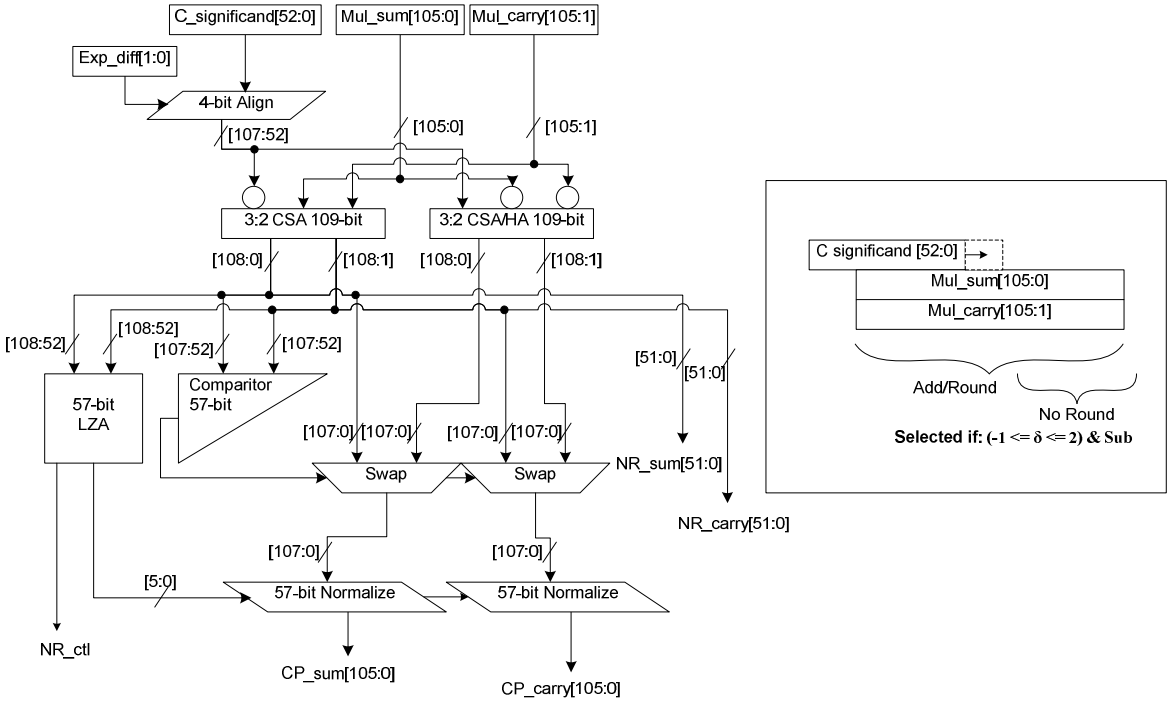

Figure 5.2.4 The Close Path

Timing simulations early in the design of the three-path fused multiply-add unit quickly identified the close path as the critical timing arc. As a result the design was changed to reduce the critical path by shrinking the bit-sizes of the high-latency components. Specifically, the original 109-bit LZA and 109-bit normalizers were reduced to a 57-bit range. Logically, this reduction is a legal move, as cases of massive cancellation exceeding 57-bits in length will produce a result that needs no rounding. This “no round” case is triggered by the 57-bit LZA ‘1’s detection’ term. If selected, data enter the no round path (described in detail in the next section) and performs the remaining addition and normalization in parallel with the add/round stage.

5.2.3 The Add/Round Stage

All three middle stage paths in the three-path fused multiply-add design prepare the data for the 106-bit add/round stage. The combined addition and rounding stage algorithm combines various suggestions for the add/round stages of a floating-point multiplier [26],

[27] with modifications to the control logic, signals, and multiplexer sizes to account for the fused multiply-add functionality. Finally, a “no round” path block has also been added in parallel to the scheme to handle the extra output case from the close path.

The combined fused multiply-add add/round stage is shown in Figure 5.2.5. The stage begins with a control block that selects the correct three path output and directs the data to the add/round scheme. The upper 54-bits of the selected data enter two half adder stages that remove least significant bits for rounding control. The lower 53-bits are passed to a carry and sticky block that produces the round and carry bits for the final round logic. One of the stage input bits is mutual to both.

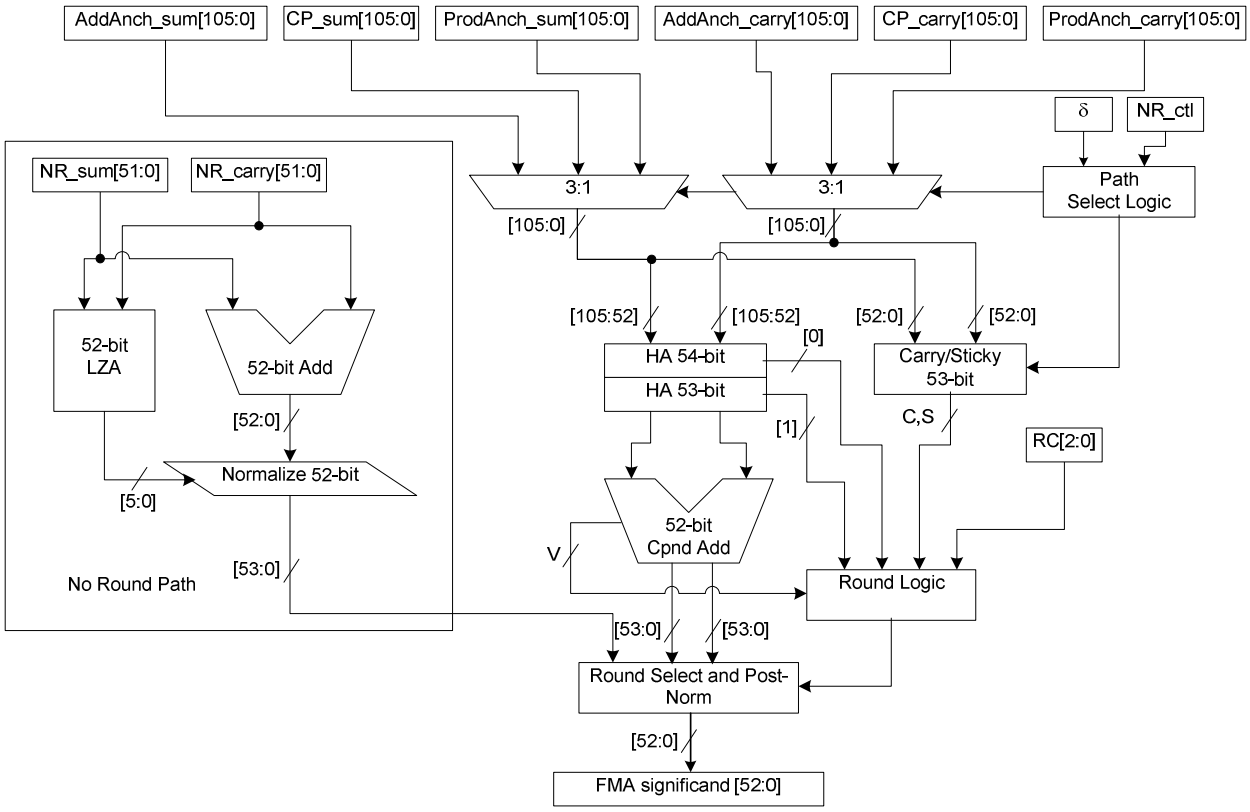

Figure 5.2.5 The No Round Path (left) and Add/Round Stage (right)

The data from the half adder stages enter a 52-bit compound adder, which produces a sum and an augmented sum (i.e., sum+1). Meanwhile, the rounding logic adjusts the

lower 2 bits of the half adders and sends a carry out select signal to choose the correct adder output. The selected result is post-normalized and latched.

In the case of a close path selection with the no round signal assertion, the no round data inputs are added and normalized in a path separate from and parallel to the add/round stage. The result from this “no round” path is forwarded to the add/round result multiplexer, post-normalized, and latched.

When either the no round path result or add/round result is latched, the fused multiply-add instruction is complete and data exit the unit.

5.2.4 Exponent and Sign Logic

The exponent and sign stage architecture in a three-path fused multiply add unit is very similar to that used by a classic fused multiply-add unit, only showing two major differences. The first difference, shown in Figure 5.2.6, is the use of four initial exponent calculations. Three of these four exponent paths are used for alignment, comparison, and multiplier exponent values, much like a classic fused multiply-add unit. The fourth exponent path is needed in the three-path architecture to generate the alignment value specifically for the close path.

The close path alignment calculation is done in the sign/exponent block for timing reasons. Unlike the anchor paths, which have localized exponent adjustments, the close path is in the critical timing arc of the entire fused multiply-add unit. According to simulations, adding local exponent handlers in the close path block increases the unit’s total delay, so the difference calculation has been moved further up the datapath.

The second major difference found in the three-path exponent architecture is the removal of the classic fused multiply-add normalization adjustment adders. In a three-path architecture, the exponent normalization constant is selected at the path merger along with the correct operands. Since each path already takes exponent adjustment into

account, no additional parallel processing is needed in the exponent logic block. Instead, the selected path passes the already calculated normalization vector to the final exponent adder. After this final exponent adder, the path is again similar that used by a classic fused multiply-add unit.

The sign logic used in a three-path architecture is a simple design. Combinational logic takes the true operand control signal, the complement signal generated in the close path, the final path selection signal, and the exponent comparison result signal to determine the final sign bit. The only block dependent on the sign result signal is the three-path rounding logic, but the signal arrives early enough that it does not affect the latency of any critical path before exiting the unit.

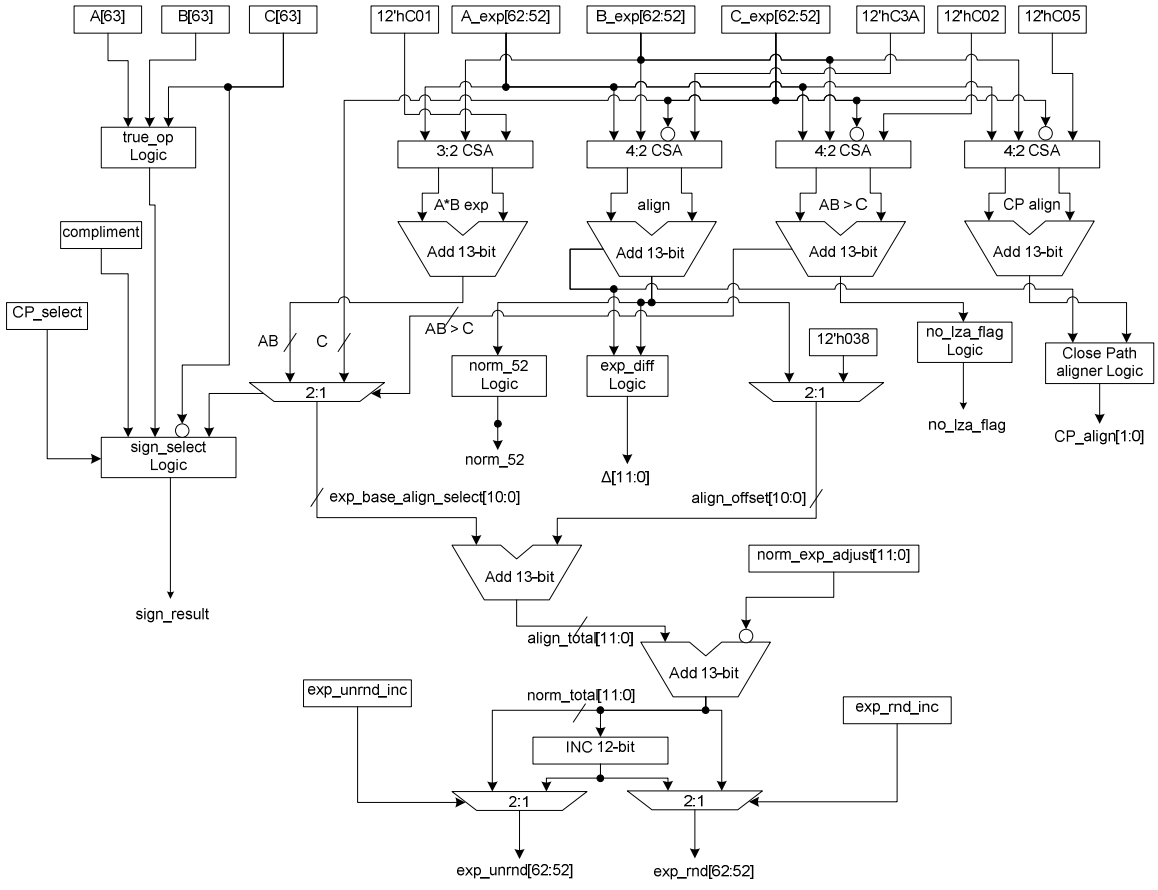

Figure 5.2.6 Exponent and Sign Logic

5.3 Three-Path Fused Multiplier-Adder with Multiplier Bypass

The three-path fused multiply-add unit presents a unique additional option due to the configuration of its components—a floating-point multiplication bypass. Since a fused multiply-add unit always begins with a multiplication array, the hardware capable of handling the first half of a stand-alone floating-point multiply is already in place. Additionally, since the three-path add/round stage is designed from architectures intended for floating-point multipliers, all the necessary hardware for the optional instruction is present. Therefore, to allow for a stand-alone floating-point multiplication in the three-path fused multiply-add unit architecture, only a small update to exponent logic and the introduction of a simple bypass connecting the multiplication array to the add/round stage is needed. This optional configuration is shown in Figure 5.3.1.

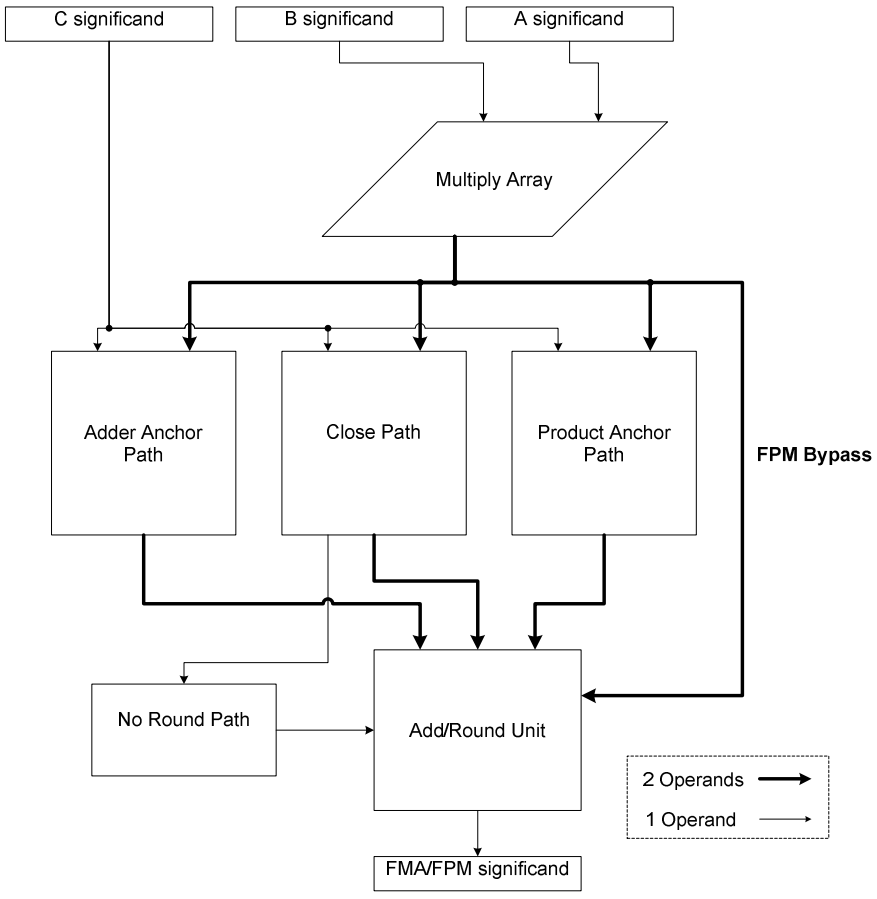

Figure 5.3.1 Three-path fused multiply-add with FPM bypass

5.4 Three-Path Fused Multiplier-Adder Results

The floating-point three-path fused multiply-add unit has been designed and implemented on the AMD 65nm silicon on insulator (SOI) technology and design flow. To provide a comparison of the three-path architecture's capabilities and improvements, a classic floating-point fused multiply-add unit has also been designed and implemented in the same technology as described in Chapter 4.

A floorplan screenshot of the three-path architecture is shown in Figure 5.4.1 in an orientation where data flow from top-to-bottom with bit-positions starting at 63 and going to 0 from left-to-right. The data use a pitch of 2-rows / 1-bit at the input and output to interface with a multiple RD/WR port register file, but compresses to a 1-row / 1-bit pitch internally due to the folding of the multiplier array.

Table 5.4.1 provides the color-key legend for the three-path floorplan from Figure 5.4.1, identifying the major components of the new fused multiply-adder.

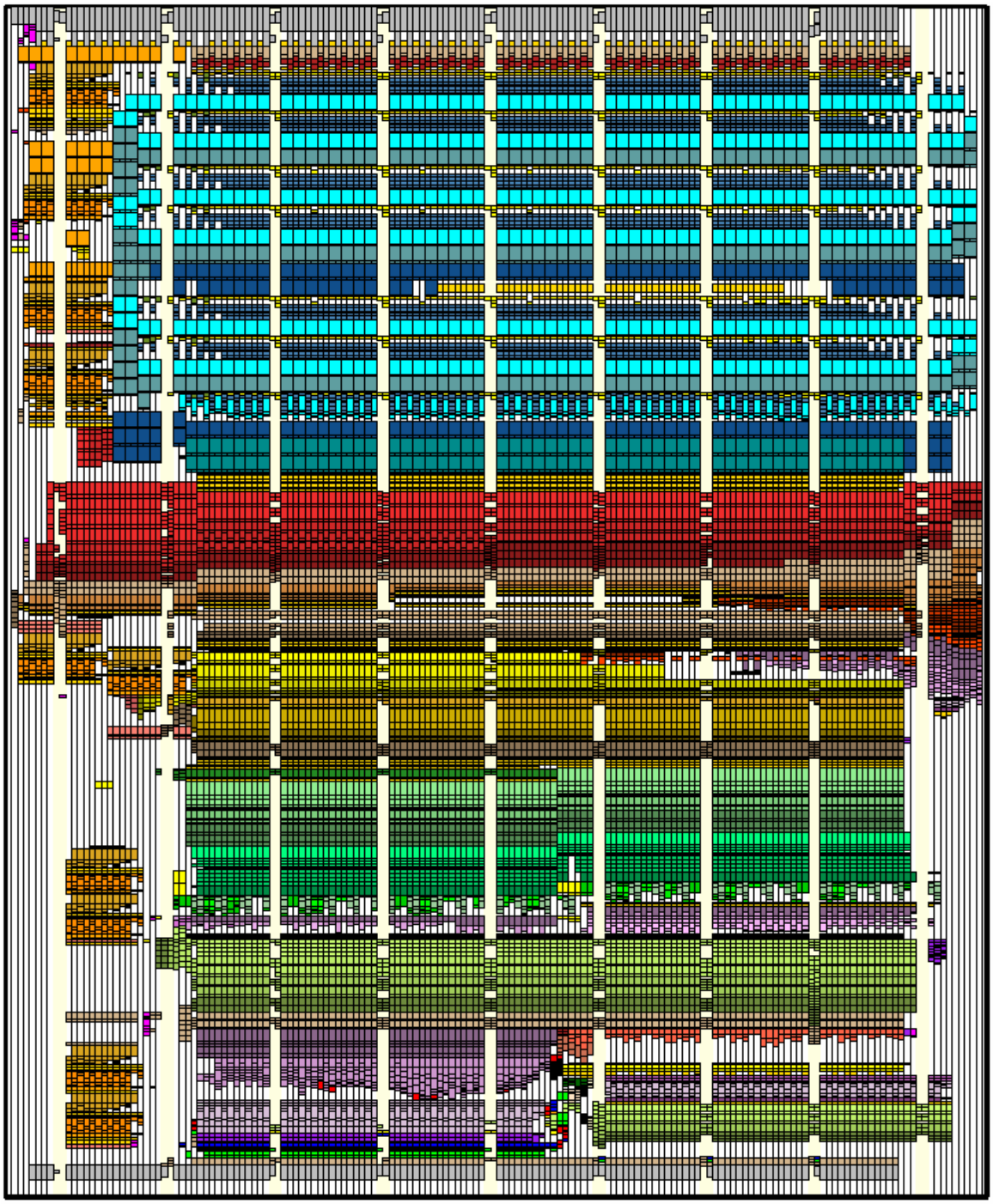

Figure 5.4.1 Three-path fused multiply-add floorplan

Table 5.4.1 Floating-point fused multiply-add color legend

Color	Component
Brown/Tan (top)	Booth Encoding/Buffering
Blues	Multiplier Array
Reds	Adder Anchor Path
Yellow/Gold	Product Anchor Path
Greens	Close Path
Brown/Tan (mid)	Buffering/Muxing
Olive Greens	Close Path/No Round Normalize
Purples	Adder/Comparator/Carry Tree
Orange	Exponent

A screenshot of the critical path is shown in Figure 5.4.2, captured during a circuit simulation at1.3V 100°C in a TypV_T process corner. Below the critical path figure is brief identification of the critical path block sequence.

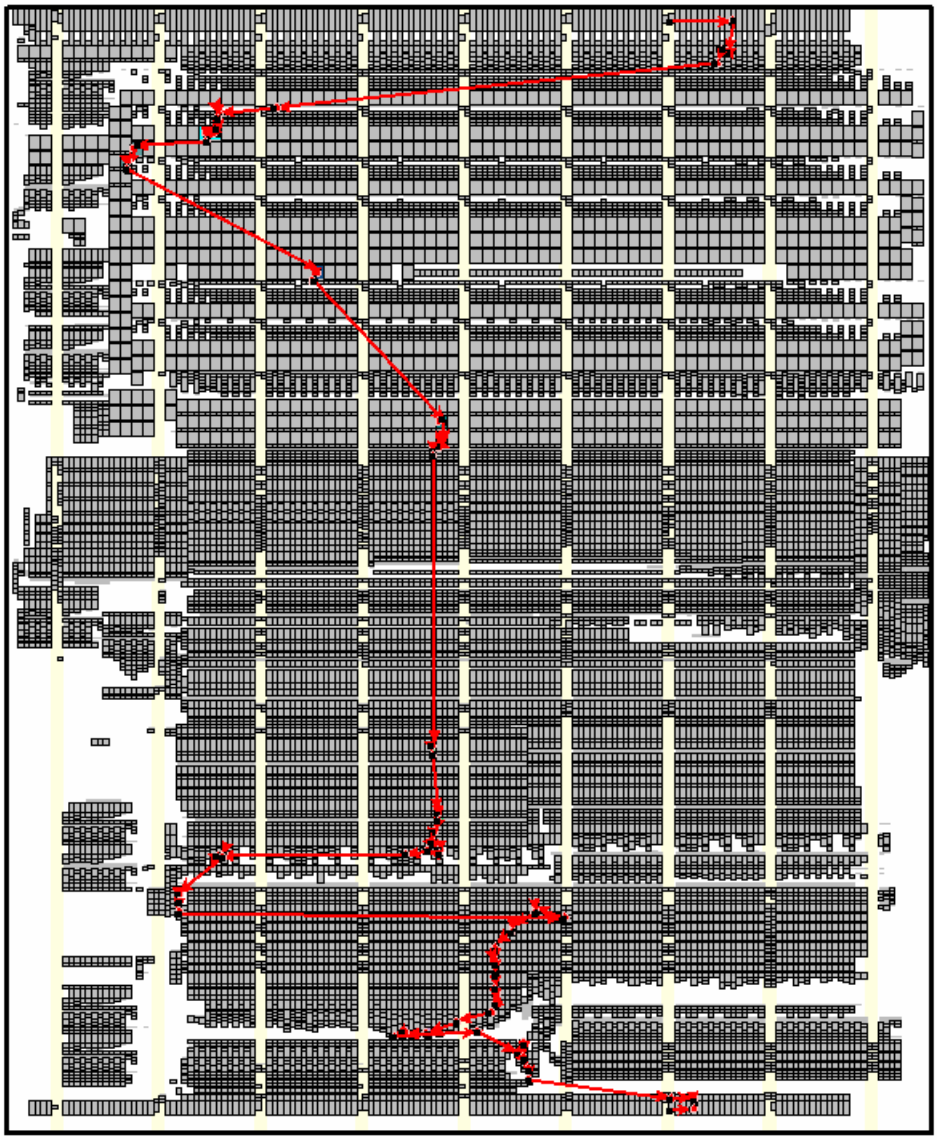

Figure 5.4.2 Three-path fused multiply-add critical path

Critical Path: BoothEnc → Mul Array → Close Path LZA → Normalize57 → 2 x HA → CpndAdd52 → Post-norm

Table 5.4.2 provides the results from two timing runs performed at 1.3V 100°C TypV_T and 0.7V 100°C LowV_T respectively. The area calculations come from the actual

dimensions of the floorplan, and the power results from HSim power simulations from the floorplan's extracted netlist.

Table 5.4.3 compares the classic fused multiply-add and the three-path fused multiply-add designs in the categories of latency, area, and power consumption. The comparison results provide absolute as well as relative results. The difference row provides the increase/decrease of the three-path fused multiply-add relative to the classic fused multiply-add.

As shown in Table 5.4.3, the three-path fused multiply-add design shows about a 12% decrease in latency as compared to a classic fused multiply-add. Additionally, when clocked at the same frequency, the three-path fused multiply-add design provides an about a 15% reduction in the maximum power consumption. Both the power and latency gains of the three-path fused multiply-add architecture come at the price of a nearly 40% increase in area.

Table 5.4.2 Floating-point three-path fused multiplier-adder results

Design	Latency 1.3V 100°C $TypV_T$	Latency 0.7V 100°C $LowV_T$	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V $TypV_T$	Transistors
FMA_3Path	1081ps	2959ps	557µm x 465µm = 259,005μm^2	354mW	246,914

Table 5.4.3 Floating-point fused multiplier-adder comparative results.

Design	Latency 1.3V 100°C $TypV_T$	Latency 0.7V 100°C $LowV_T$	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V $TypV_T$
Classic FMA	1224ps	3363ps	186,930μm^2	416mW
Three-Path FMA	1081ps	2959ps	259,005μm^2	354mW
Difference	–11.7%	–12.0%	38.6%	–14.9%

Chapter 6

The Bridge Fused Multiply-Add Architecture

This chapter provides the design and implementation details of a new floating-point bridge fused multiplier-adder created using the AMD 65nm silicon on insulator circuit design flow. The bridge architecture provides a hardware configuration that may dynamically operate in either a dual-pipeline mode that executes full-performance floating-point addition and floating-point multiplication instructions in parallel or a single-pipeline mode that executes a floating-point fused multiply-add instruction.

6.1 Introduction

A great advantage of a classic floating-point fused multiply-add architecture is its ability to execute all arithmetic instructions in a single unit. Not only does a fused multiplier-adder show increased performance of the instruction (A x B) + C as compared to a floating-point multiplier followed by a floating-point adder, but it may entirely replace them in hardware.

A fused multiply-add unit may emulate a floating-point adder and floating-point multiplier by inserting fixed constants into its data path. A floating-point addition is executed by replacing operand B with 1.0, forming the equation (A x 1.0) + C. Likewise, a floating-point multiplication is executed by replacing operand C with 0.0, forming the equation (A x B) + 0.0. This simple injection of constants allows a floating-point fused multiply-add unit to be built as the stand-alone, all-purpose execution unit inside a floating-point co-processor.

However, the greatest advantage of the modern fused multiply-add is also the greatest argument against its use. Fused multiply-add units make significant gains in multiply-add instruction performance by combining the hardware of a floating-point multiplier and floating-point adder into a tighter datapath, as well as by removing the requirement for an intermediate rounding unit. Due to the faster, yet higher complexity of the fused multiply-add unit, normal addition and multiplication instructions are subject to greater latencies than if they were processed in their original arithmetic unit (i.e., floating-point adder or floating-point multiplier). For developers that do not wish to re-compile their existing code or for algorithms that are nor amenable to implementation with a fused multiply-add instruction, the replacement of a floating-point adder and floating-point multiplier with a fused multiply-add unit may be an unattractive endeavor.

A few possible solutions to this problem have been presented in literature, as seen in Chapter 2. However, though such proposals have been made, they have yet to be implemented. Additionally, no study has yet been presented that identifies the relative costs of creating a floating-point unit capable of performing all three basic floating-point arithmetic instructions in hardware.

This chapter presents a new architecture that builds hardware fused multiply-add unit functionality into the middle of a unit containing a floating-point adder and a floating-point multiplier unit, creating a “bridge” that connects the two. The architecture is designed to re-use as much hardware as possible from both the floating-point multiplier and floating-point adder units to minimize the area and the power consumption. This design is intended to provide an identification of the implementation costs in a floating-point arithmetic unit capable of adds, multiplies, and fused multiply-add operations completely processed by hardware.

The following sections provide the architectural details of the bridge fused multiply-add unit design. After the architectural description, a results section presents the

implementation results of a bridge fused multiply-add unit that has been designed in the AMD 65nm silicon on insulator technology. The chapter finalizes by comparing the bridge results against the implementations of a classic fused multiply-add unit, a floating-point multiplier, and a floating-point adder. The designs and implementations of the units used as the basis for comparison are each presented in Chapter 4.

6.2 The Bridge Fused Multiply-Add Architecture

The bridge fused multiply-add architecture has been created with the intention to find a solution to the performance degradation of single additions and multiplications in current fused multiply-add units. The architecture has also been designed to provide a realistic study of the implementation costs involved when building an arithmetic unit capable of all three fundamental floating-point mathematical instructions.

Figure 6.2.1 shows a high level block diagram of the bridge fused multiply-add architecture. The design begins with common floating-point multiply and floating-point add units capable of independent execution. Several blocks are added between the two arithmetic units, creating a “bridge” capable of carrying data from one unit to the other to perform a fused multiply-add.

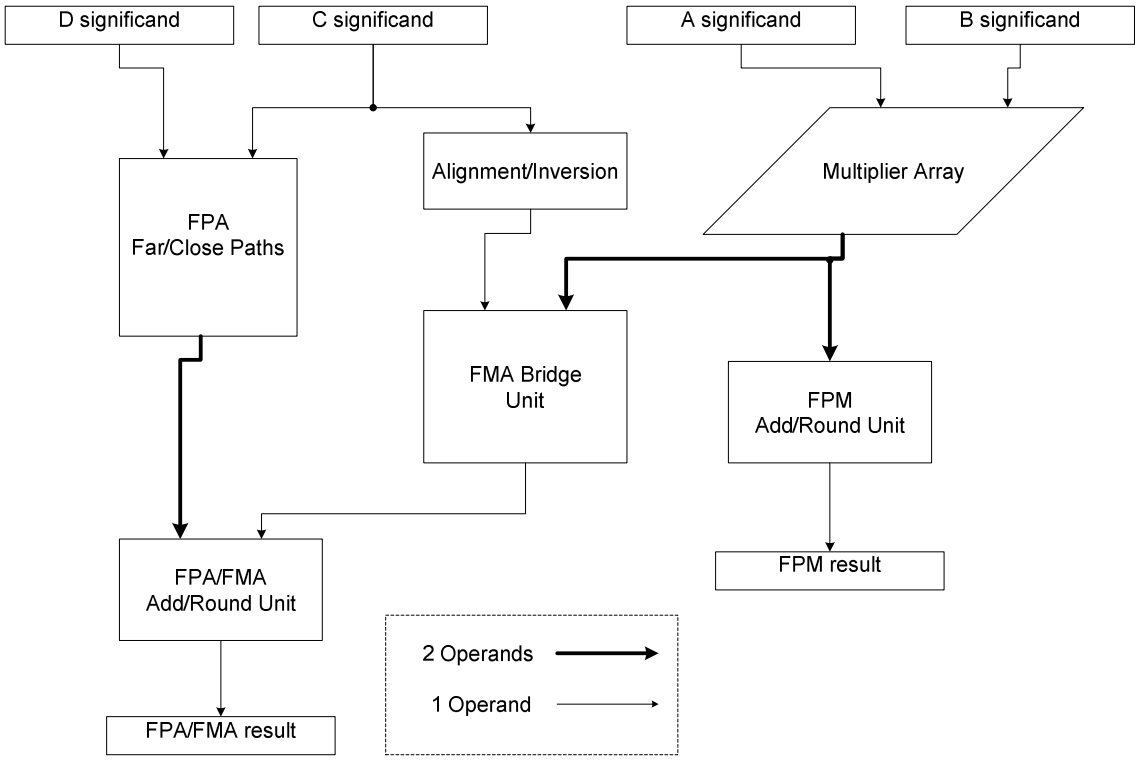

Figure 6.2.1 The bridge fused multiply-add block diagram

The bridge fused multiply-add architecture does not require an entire independent fused multiply-add hardware implementation. Pieces from both the floating-point multiplier and floating-point adder are modified and reused for dual functionality. Specifically, the floating-point adder's add/round unit is used for both single adds and fused multiply-adds, while the multiplier re-uses the largest component block of any arithmetic unit, the multiplier array. The remaining hardware requirements for a complete fused multiply-add instruction are implemented in the bridge unit.

6.2.1 The Multiplier

The bridge fused multiply-add architecture uses a floating-point multiplier that executes stand-alone multiplications as well as the first stage of a fused multiply-add. As shown in Figure 6.2.2, the double-precision multiplier unit takes two 64-bit operands as inputs. The significands are processed in a 53 x 53-bit multiplier, while the exponent and sign bits are processed in parallel. For a multiplication instruction, the multiplier array forwards the

106-bit sum and carry results to a floating-point multiplier rounding unit designed from the suggestions of several multiplication rounding schemes [26], [27].

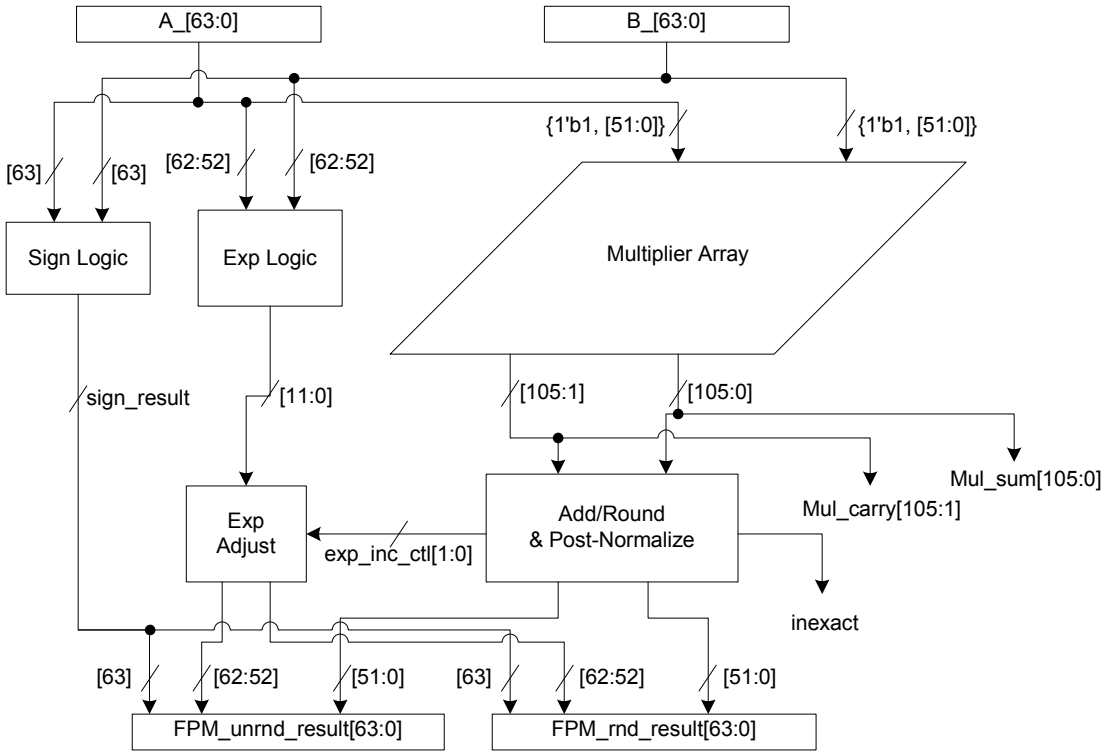

Figure 6.2.2 The Multiplier

When the required operation is a fused multiply-add instruction, the unit begins execution in the same way as a floating-point multiplication. However, when the multiplier array produces a sum/carry result, the data are forwarded outside the unit to the bridge and the floating-point multiplier rounding scheme is shut down.

6.2.2 The Bridge

The bridge unit is shown in Figure 6.2.3. This block is essentially the classic fused multiply-add architecture described in Section 2 without the multiplier array, rounding, or post-normalization units. Instead, the bridge unit begins by accepting the product bits from the floating-point multiplier and combining them with a pre-aligned 161-bit addend.

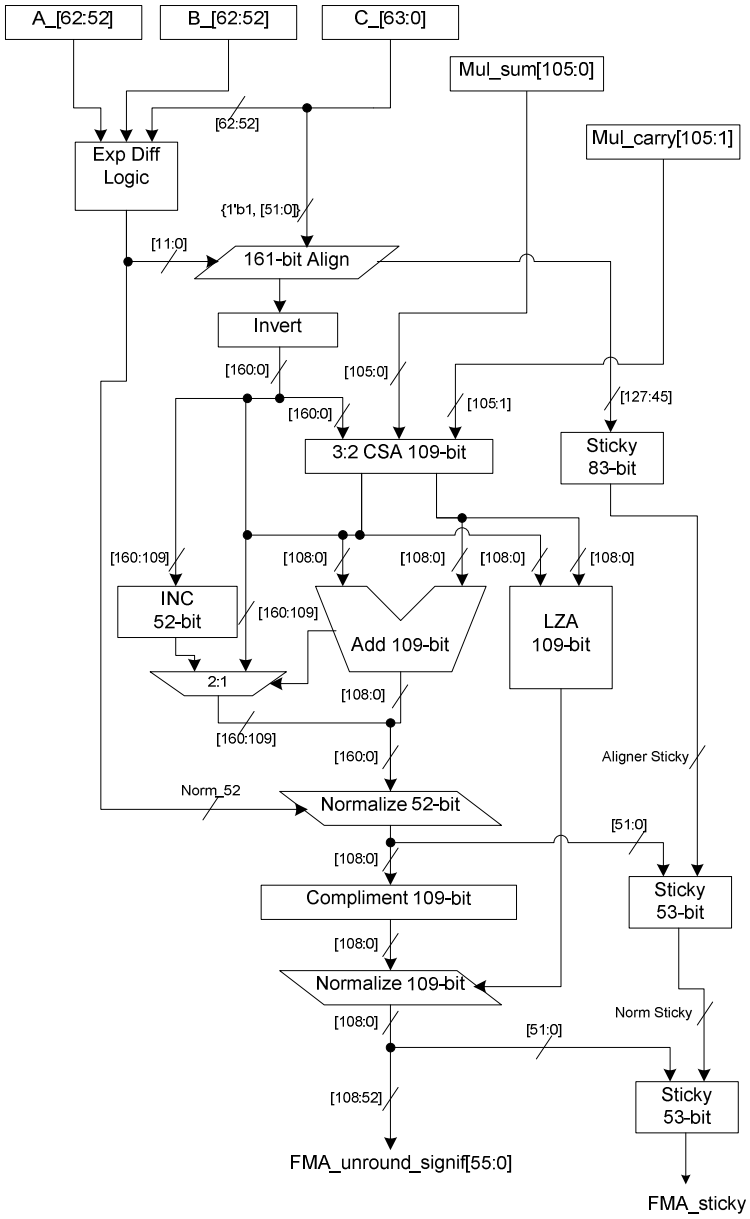

Figure 6.2.3 The Bridge

The combined data enter a 161-bit adder stage (specifically a 109-bit adder with a 52-bit incrementer in the implementation) as well as a 109-bit leading-zero anticipator (LZA) that executes in parallel to the addition. The resulting addition enters a 52-bit normalization stage and is shifted with a single multiplexer based on the range of the final exponent. The remaining 109-bits are complemented if necessary and finally enter a

109-bit normalization stage controlled by the output of the LZA unit. After normalization, the data are ready for rounding, and exits the bridge.

6.2.3 The Adder

The bridge fused multiply-add architecture uses a common Farmwald [35] dual-path floating-point adder design to execute stand-alone addition instructions. As shown in Figure 6.2.4, the addition unit uses a far and close path to handle the two classical floating point addition cases. The far path, shown on the left side of Figure 6.2.4, is used to process input significands for either an addition or a subtraction if their exponents differ by more than 1. For this path, the significands of both inputs are passed to a swap multiplexer that awaits the results of a comparison of the exponents. When the larger significand is detected, it is anchored and the smaller significand is aligned until the exponents match.

For cases of subtraction where the exponents are equal or differ by ±1, the input data are processed in the addition unit close path that is shown on the right side of Figure 6.2.4. The close path pre-shifts both input significands by one and inputs both shifted and non-shifted operands to a swap multiplexer. Meanwhile, a comparator is used to determine the larger significand in the case of no exponent difference, all while three leading one predictors (LOP) operate in parallel on each possible exponent difference case.

The exponent prediction logic and significand comparator drive the select lines on several sets of swap multiplexers. The resulting LOP selection enters a 53-bit priority encoder and is reduced to a 5-bit normalization control. Both the larger and smaller significands in the close path are normalized by up to 54-bits.

Upon each path's completion, the larger and smaller operands from both the far and close path exit the block and are forwarded to the bridge fused multiply-add unit round stage for path merging, rounding, and instruction completion.

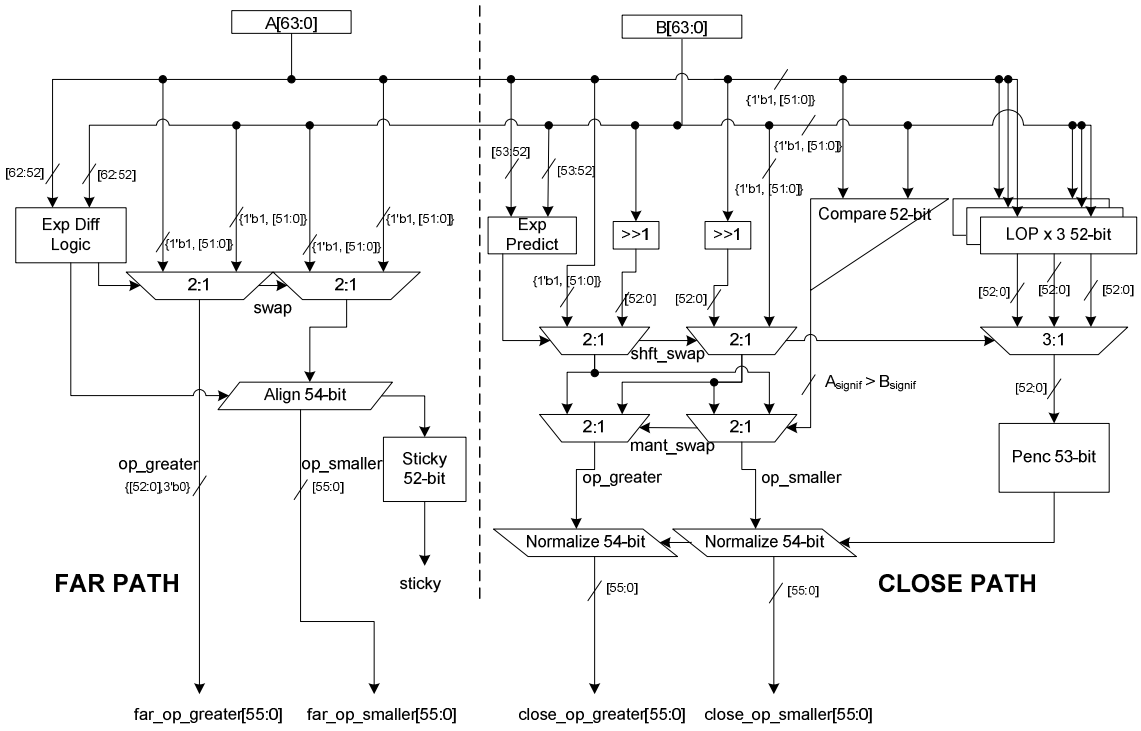

Figure 6.2.4 The Adder

6.2.4 The Add/Round Unit

The bridge fused multiply-add architecture's addition and rounding unit is designed to perform several roles. When a stand-alone addition instruction is required, the add/round unit first acts as a common floating-point adder dual-path merging stage, selecting input operands from either the far path or the close path. For a fused multiply-add instruction, this same multiplexer is expanded to select the fused multiply-add unit's un-rounded result. In this case, the second input operand is passed a null string, as another operator is not needed for the multiply-add rounding completion.

The bridge add/round unit is shown in Figure 6.2.5. The block uses a floating-point addition combined add/round scheme that comes from several suggestions as seen in [24], [25]. The two add/round stage selected double-precision input operands are passed to dual 59-bit adders that produce a result and a result plus 2 (or plus 1 for subtraction).

Providing these arithmetic results, as better explained by the literature, allows for an easy LSB fix-up, shift, post-alignment, and final result selection. The controls for the shifts come from the overflow bits of the adders, and the rounding selections are decided by combinational rounding logic.

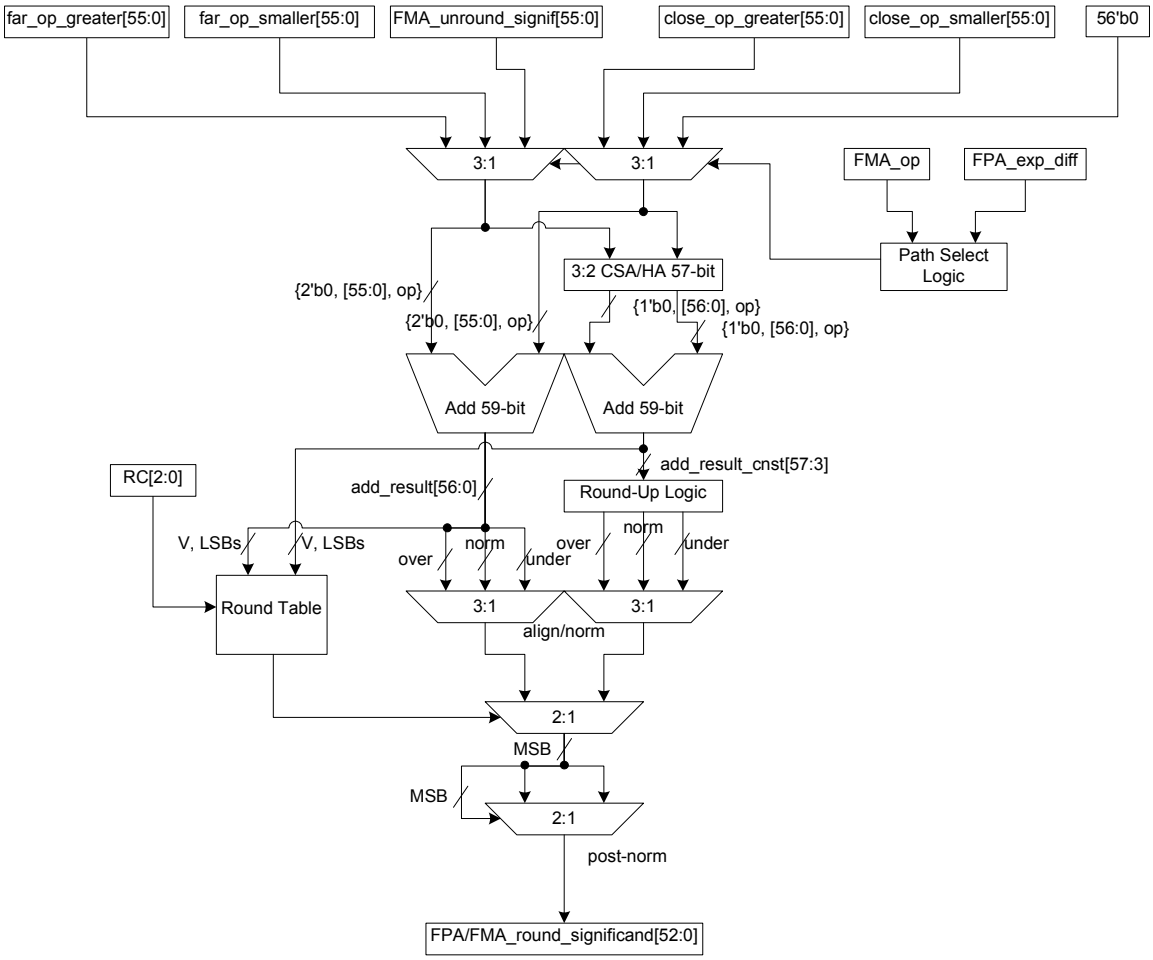

Figure 6.2.5 The Add/Round Unit

6.3 The Bridge Fused Multiplier-Adder Results

The floating-point bridge fused multiplier-adder has been designed and implemented on the AMD 65nm silicon on insulator technology and design flow. To provide a

comparison of the bridge architecture's capabilities and execution options, a classic floating-point fused multiply-adder, a floating-point multiplier, and a floating-point adder have also been designed and implemented in the same environment. Full reports on the design and implementation of these floating-point units are presented in Chapter 4.

A floorplan screenshot of the bridge architecture is shown in Figure 6.3.1 in an orientation where data flow from top-to-bottom with bit-positions starting at 63 and going to 0 left-to-right. The data use a pitch of 2-rows / 1-bit at the input and output to interface with a multiple RD/WR port register file. Flop boundaries (grey cells) may be seen separating the major functional units.

Table 6.3.1 provides the color-key legend for the bridge architecture floorplan from Figure 6.3.1, identifying the major components of the different units.

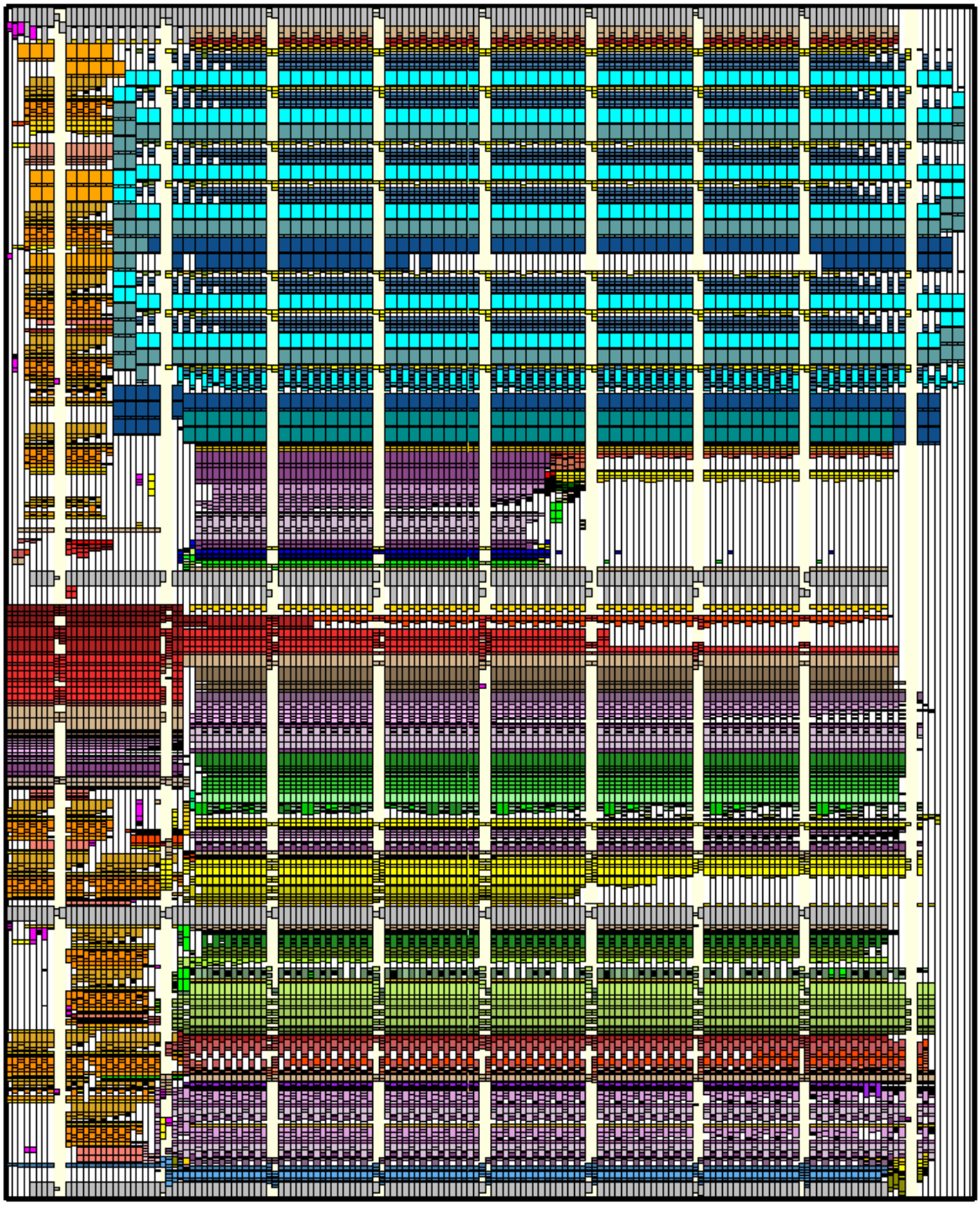

Figure 6.3.1 The bridge fused multiply-add floorplan

Table 6.3.1 Bridge fused multiply-add color legend

Color	Component
Brown/Tan (top)	Booth Encoding/Buffering
Blues	Multiplier Array
Reds	Aligner FMA Far Path FPA
Yellow/Gold	FMA Normalize Sticky/Round FPM/FPA
Greens	Close Paths FMA/FPA
Brown/Tan (mid)	Buffering/CSAs
Olive Greens	Close Path/No Round Normalize
Purples	Adder/Comparator/Carry Tree
Orange	Exponent FMA/FPM/FPA
Grey	Flops/Unit Boundaries

A screenshot of the critical path is shown in Figure 6.3.2, captured during a circuit simulation at1.3V 100°C in a TypV_T process corner. Below the critical path figure is brief identification of the fused multiplier-adder critical path sequence.

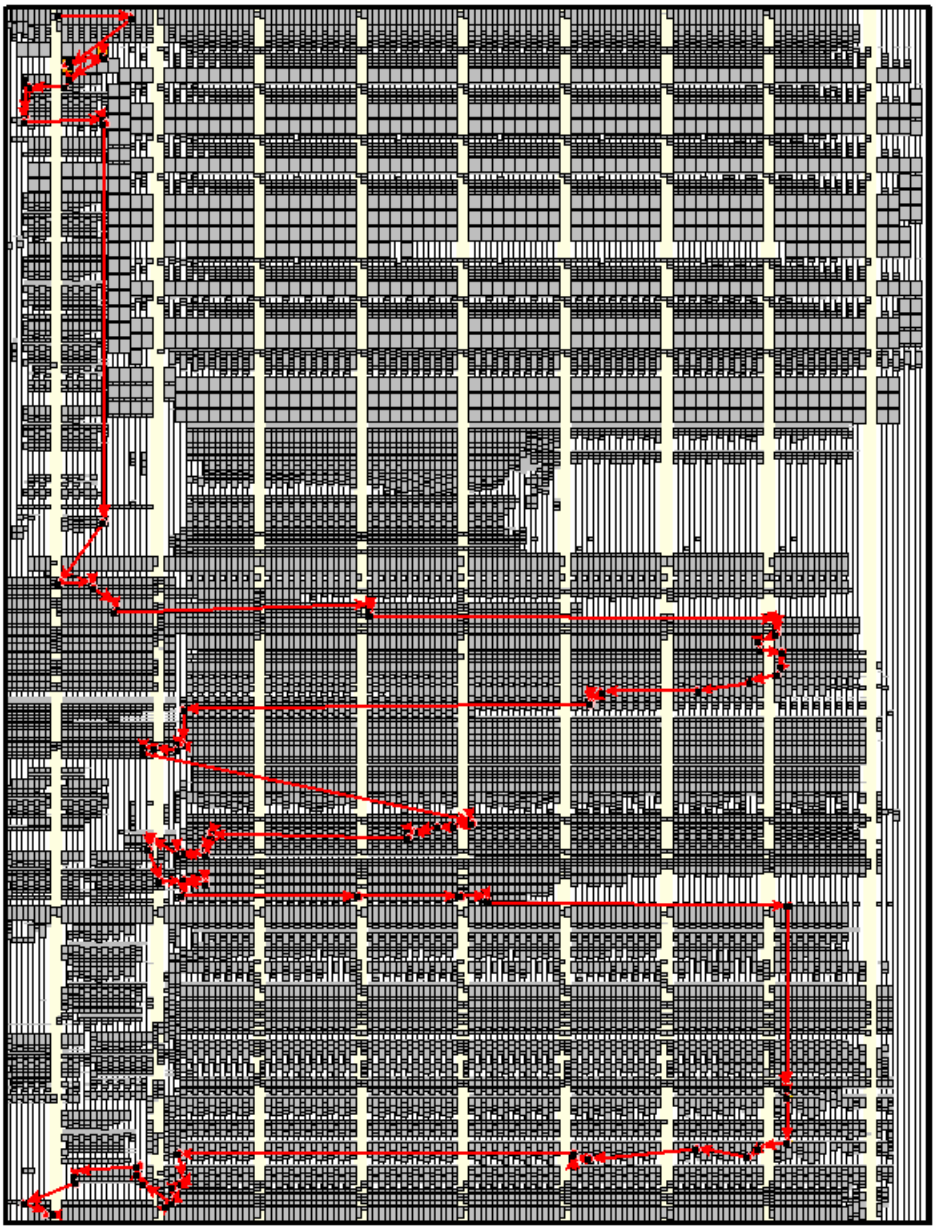

Figure 6.3.2 The bridge fused multiply-add critical path

Critical Path: ExpDiff → Align161 → 3:2 CSA → Add109 → Inc52Sel → Norm52 → Comp → Norm109 → floating-point add/FMA Merge → Add59 → ExpInc

Table 6.3.2 provides the bridge results from two timing runs performed at 1.3V 100°C TypV_T and 0.7V 100°C LowV_T respectively. The area calculations are from the actual dimensions of the floorplan, and the power results are from HSim power simulations using the floorplan's extracted netlist.

The bridge fused multiply-add unit implementation is compared in Tables 6.3.3-6.3.6 to a floating-point adder, floating-point multiplier, and classic fused multiply-add unit implementation over the categories of latency, area, and power consumption. The comparison results provide the absolute simulation calculations as well as the relative performance of all architectures in stand-alone addition, multiplication, and fused multiply-add instructions. The '**Δ**' rows provide the increase/decrease of an implementation's results relative to the first row of the table.

As seen in Tables 6.3.4-6.3.5, the bridge fused multiply-add architecture provides delay and power consumption comparable to stand-alone floating-point adders and floating-point multipliers for individual instructions. The bridge fused multiply-add architecture is about 40% larger than the combination of a stand-alone floating-point adder and a stand-alone floating-point multiplier. The bridge architecture is 30% to 70% faster and 50% to 70% lower in power consumption than a classic fused multiplier-adder when executing single-unit instructions. The bridge fused multiply-add architecture is about 50% larger than a classic fused multiply-add unit.

As shown by Table 6.3.6, when compared to a classic fused multiply-add architecture executing a fused multiply-add instruction, the bridge fused multiply-add unit shows 20% lower speed at 20% higher power consumption. The bridge fused multiply-add unit is about 50% larger than a classic fused multiply-add unit.

Table 6.3.2 Bridge fused multiplier-adder results

Design	Latency 1.3V 100°C TypV_T	Latency 0.7V 100°C LowV_T	Area 65nm AMD SOI	Power (max) 666 MHz 1.3V TypV_T	Transistors
FMA_Bridge	FPA: 978ps FPM: 708ps FMA: 1454ps	FPA: 2625ps FPM: 1979ps FMA: 3973ps	610µm x 465µm = 283,650µm^2	FMA: 501mW	249,694

Table 6.3.3 Raw results from various floating-point units

Design	Latency 1.3V 100°C RV_T	Latency 0.7V 100°C LV_T	Area 65nm AMD SOI	Peak Power 666 MHz 1.3V RV_T
FPA_DP	946ps	2556ps	72,075µm^2	118mW
FPM_DP	701ps	1950ps	131,130µm^2	187mW
Classic FMA	1224ps	3363ps	186,930µm^2	416mW
Bridge FMA	**FPA:** 978ps **FPM:** 708ps **FMA:** 1454ps	**FPA**: 2625ps **FPM**: 1979ps **FMA**: 3973ps	283,650µm^2	**FPA:** 118mW **FPM:** 187mW **FMA:** 501mW

Table 6.3.4 Results Normalized to an FPA Stand-Alone Addition.

Design	Latency Difference 1.3V 100°C RV_T	Latency Difference 0.7V 100°C LV_T	Area Difference 65nm AMD SOI	Power Difference 666 MHz 1.3V RV_T
FPA	0%	0%	0%	0%
Classic FMA Δ	29.4%	31.6%	159.4%	252.5%
Bridge FMA Δ	3.4%	2.7%	293.6%	0%

Table 6.3.5 Results Normalized to an FPM Stand-Alone Multiplication.

Design	Latency Difference 1.3V 100°C RV_T	Latency Difference 0.7V 100°C LV_T	Area Difference 65nm AMD SOI	Power Difference 666 MHz 1.3V RV_T
FPM	0%	0%	0%	0%
Classic FMA Δ	74.6%	72.5%	42.6%	122.5%
Bridge FMA Δ	1.0%	1.5%	116.3%	0%

Table 6.3.6 Results Normalized to a Classic Fused Multiply-Add.

Design	Latency Difference 1.3V 100°C RV_T	Latency Difference 0.7V 100°C LV_T	Area Difference 65nm AMD SOI	Power Difference 666 MHz 1.3V RV_T
Classic FMA	0%	0%	0%	0%
FPA + FPM Δ	34.6%	34.0%	8.7%	-26.7%
Bridge FMA Δ	18.8%	18.1%	51.7%	20.4%

Chapter 7

Conclusions and Future Work

This final chapter presents the concluding remarks on the construction and comparison of the new floating-point fused multiply-add architectures. Following, the chapter ends with a brief summary of the suggested future works that could enhance the design and functionality of the new fused multiply-add architectures.

7.1 Conclusions

This dissertation has presented the results of the research, design, and implementations of several new architectures for floating-point fused multiplier-adders used in the floating-point units of microprocessors. These new architectures have been designed to provide solutions to the implementation problems found in modern-day fused multiply-add units. The new three-path fused multiply-add architecture shows a 12% reduction in latency and a concurrent 15% reduction in power as compared to a classic fused multiply-add unit. The new bridge fused multiply-add architecture presents a design capable of full performance floating-point addition and floating-point multiplication instructions while still providing the functionality and performance gain of a fused multiplier-adder.

The difficult latency, power consumption, and single-instruction performance degradation problems facing a standard floating-point fused multiply-add design are well known problems and have not gone unnoticed by the floating-point design engineering community. As presented in the literature review in Chapter 2, the two new architectures presented in this dissertation are not the first to attempt a resolve of the disadvantages found in a fused multiply-add unit. Both theoretical and implemented solutions have been

presented previous to this work in an attempt to improve the architecture of a fused multiply-add unit.

However, the new designs presented in this dissertation show a clear advantage over all previous fused multiply-add designs. The three-path architecture is the fastest and lowest power numerically correct floating-point fused multiply-add unit implemented to-date. Unlike many of its predecessors, the three-path results are not based on pure theoretical analysis, as the architecture has been designed and implemented in a real industrial strength technology. Additionally, for those previous designs that were synthesized into physical models, the three-path architecture provides its benefits without the introduction of additional mathematical error, massive variable-width multiplier arrays, or archaic specialized components.

The bridge fused multiply-add architecture is the first of its kind to present a complete solution to the problem of single-instruction performance degradation. Previous works have suggested methods of accelerating either a floating-point multiplication or floating-point addition, but never before has an implemented design been capable of executing all three fundamental floating-point operations each in their original form.

To summarize the benefits of the two new architectures presented in this dissertation as compared to previous proposals, Table 2.10.1 from Chapter 2 has been modified in Table 7.1.1 to include the three-path and bridge designs. The table compares each fused multiply-add architecture against the original IBM RS/6000 [1], [2] in the categories of latency, power reduction, implementation, numerical correctness, and whether the design supports max-performance single-instruction execution.

Table 7.1.1 Comparison of proposed fused multiply-add architectures

Design	Latency vs RS/6000	Power vs RS/6000	Implemented or Theoretical	Numerically Correct?	Max-performance FPM?	Max-performance FPA?
IBM RS/6000 [1],[2]	N/A	N/A	Implemented	Yes	No	No
IBM PowerPC 604e [12],[13]	faster SP, slower DP	½ size Mul tree	Implemented	Yes	No	No
HAL SPARC64 (pseudo-FMA) [21]	slower	N/A	Implemented	rounded twice	Yes	Yes
Concordia FMA [22]	-9%	-44%	Implemented	No	Yes	No
Lang/Bruguera [23]	-(15-20%)	N/A	Theoretical	Yes	No	No
Seidel Multi-Path [28]	-30%	N/A	Theoretical	unclear	No	No
Xiao-Lu LZA improvement of Lang/Bruguera [30]	-(15-20%) - (0.17 x LZA)	N/A	LZA Implemented, FMA Theoretical	Yes	No	No
Lang/Bruguera w/ FPA bypass [32]	-10%	N/A	Theoretical	Yes	No	No[§]
Three-Path FMA	-12%	-15%	Implemented	Yes	Yes	No
Bridge FMA	20%	20%	Implemented	Yes	Yes	Yes

Each new architecture presented in this dissertation as well as a collection of modern floating-point arithmetic units used for comparison have been designed and implemented using the Advanced Micro Devices 65 nanometer silicon on insulator transistor technology and circuit design toolset. All designs use the AMD 'Barcelona' native quad-core standard-cell library as an architectural building block to create and contrast the new architectures in a realistic and cutting-edge industrial technology.

[§] 40% faster floating-point add performance as compared to a classic FMA execution of the same

7.2 Future Work

While the three-path and bridge fused multiply-add architectures provide solutions to the major disadvantages of the floating-point fused multiply-add unit, each new design cannot resolve every problem as a single block. The three-path architecture, while lower in power consumption and higher in performance than a classic fused multiply-add unit, still has the disadvantage of single-precision performance degradation. While the optional floating-point multiplier bypass orientation of the three-path design allows for a single-instruction multiplication, the unit continues to add latency to a floating-point addition. Likewise, while the bridge architecture may solve the performance degradation problems of single instructions, the fused multiply-add operation itself is still plagued by high power consumption and difficult timing arcs.

The next logical step for the improvement of these fused multiply-add architectures would be to combine the two solutions into a single unit that reduces latency, lowers power consumption, and allows for maximum performance of single floating-point instructions. While such a combination has not yet been fully executed, the three-path architecture has already been partially designed with this next step in mind.

The three-path add/round stage intentionally uses a rounding scheme from a floating-point multiplier so that a simple floating-point single-instruction multiplication may use the bypass orientation to execute a multiply at no extra latency cost. A future project investigating this architecture could add a maximum performance floating-point addition configuration to this optional configuration. Since the three-path architecture already uses two anchor paths and a close path, a dual-path floating-point adder could be integrated into the design with some creativity and re-use of existing components. In a successfully executed design, this modified three-path architecture could solve each major problem of the floating-point fused multiply-add unit as a stand-alone processor.

Bibliography

[1] R.K. Montoye, E. Hokenek and S.L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal of Research & Development*, Vol. 34, pp. 59-70, 1990.

[2] E. Hokenek, R. Montoye and P.W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE Journal of Solid-State Circuits*, Vol. 25, pp. 1207-1213, 1990.

[3] J. A. Davis, R. Venkatesan, A. Kaloyeros, M. Beylansky, S. J. Souri, K. Banerjee, K. C. Saraswat, A. Rahman, R. Reif and J. D. Meindl, "Interconnect Limits on Gigascale Integration (GSI) in the 21st Century," *Proceedings of the IEEE*, Vol. 89, pp. 305-324, 2001.

[4] S. Natarajan and A. Marshall, "Technological Innovations to Advance Scalability and Interconnects in Bulk and SOI," *Proceedings of the 15th International Conference on VLSI Design,* pp. 297-298, 2002.

[5] R.K. Krishnarnurthy, A. Alvandpour, V. De and S. Borkar, "High-Performance and Low-Power Challenges for Sub-70 nm Microprocessor Circuits," *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference,* pp. 125-128, 2002.

[6] C. Hinds, "An Enhanced Floating Point Coprocessor for Embedded Signal Processing and Graphics Applications," *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers*, pp. 147-151, 1999.

[7] Y. Voronenko and M. Puschel, "Automatic Generation of Implementations for DSP Transforms on Fused Multiply-Add Architectures," *International Conference on Acoustics, Speech and Signal Processing*, pp. V-101-V-104, 2004.

[8] E. N. Linzer, "Implementation of Efficient FFT Algorithms on Fused Multiply-Add Architectures," *IEEE Transactions on Signal Processing*, Vol. 41, pp. 93-107, 1993.

[9] A. D. Robison, "N-Bit Unsigned Division Via N-Bit Multiply-Add," *Proceedings of the 17th IEEE Symposium On Computer Arithmetic*, pp. 131-139, 2005.

[10] R.-C. Li, S. Boldo and M. Daumas, "Theroems on Efficient Argument Reductions," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 129-136, 2003.

[11] F. P. O'Connell and S. W. White, "POWER3: The Next Generation of PowerPC Processors," *IBM Journal of Research and Development*, Vol. 44, pp. 873-884, 2000.

[12] R. Jessani and C. Olson, "The Floating-Point Unit of the PowerPC 603e," *IBM Journal of Research and Development*. Vol. 40, pp. 559-566, 1996.

[13] R.M. Jessani and M. Putrino, "Comparison of Single- and Dual-Pass Multiply-Add Fused Floating-Point Units," *IEEE Transactions on Computers*, Vol. 47, pp. 927-937. 1998.

[14] A. Kumar, "The HP PA-8000 RISC CPU," *IEEE Micro Magazine*, Vol. 17, Issue 2, pp. 27-32, April, 1997.

[15] D. Hunt, "Advanced Performance Features of the 64-bit PA-8000," *Proceedings of Compcon*, pp. 123-128, 1995.

[16] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro Magazine*, Vol. 16, No. 2, pp. 28-40, March, 1996.

[17] B. Greer, J. Harrison, G. Henry, W. Li and P. Tang, "Scientific Computing on the Itanium Processor," *Proceedings of the ACM/IEEE SC2001 Conference*, pp. 1-8, 2001.

[18] H. Sharangpani and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro Magazine*, Vol. 20, No. 5, pp. 24-43, Sept-Oct, 2000.

[19] *DRAFT Standard for Floating-Point Arithmetic P754*, IEEE Standard (proposed), Aug 18, 2006.

[20] *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Standard 754-1985.

[21] A. Naini, A. Dhablania, W. James and D. Das Sarma, "1 GHz HAL Sparc64 Dual Floating Point Unit with RAS Features," *Proceedings of the 15th Symposium on Computer Arithmetic,* pp.173-183.

[22] R.V.K. Pillai, S.Y.A. Shah, A.J. Al-Khalili and D. Al-Khalili, "Low Power Floating Point MAFs – A Comparative Study," *Sixth International Symposium on Signal Processing and its Applications*, Vol. 1, pp. 284-287, August, 2001.

[23] T. Lang and J. D. Bruguera, "Floating-Point Fused Multiply-Add with Reduced Latency," *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 145-150, 2002.

[24] N. Quach, N. Takagi and M. Flynn, *On Fast IEEE Rounding*, Technical Report CSL-TR-91-459, Stanford University, Jan. 1991.

[25] N. Quach and M.J. Flynn, *An Improved Algorithm for High-Speed Floating Point Addition*, Technical Report CSL-TR-90-442, Computer Systems Laboratory, Stanford University, Aug. 1990.

[26] G. Even and P. M. Seidel, "A Comparison of Three Rounding Algorithms for IEEE Floating-Point Multiplication," *IEEE Transactions on Computers,* Vol. 49, pp. 638-650, 2000.

[27] R. K. Yu and G. B. Zyner, "167 MHz Radix-4 Floating-Point Multiplier," *Proceedings of the 12th Symposium on Computer Arithmetic,* pp. 149-154, 1995.

[28] P.-M. Seidel, "Multiple Path IEEE Floating-Point Fused Multiply-Add," *Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems*, pp. 1359- 1362, 2003.

[29] C. Jacobi, K. Weber, V. Paruthi and J. Baumgartner, "Automatic Formal Verification of Fused-Multiply-Add FPUs," *Proceedings of Design, Automation and Test in Europe*, Vol. 2, pp. 1298-1303, March, 2005.

[30] M. Xiao-Lu, "Leading Zero Anticipation for Latency Improvement in Floating-Point Fused Multiply-Add Units," *6th International Conference on ASIC*, pp. 53-56, 2005.

[31] M.S. Schmookler and K.J. Nowka, "Leading Zero Anticipation and Detection – A Comparison of Methods," *Proceedings of the 15th IEEE Symposium on Computer Arithmetic,* pp. 7-12, 2001.

[32] J.D. Bruguera and T. Lang, "Floating-Point Fused Multiply-Add: Reduced Latency for Floating-Point Addition," *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. pp. 42-51, June, 2005.

[33] *IEEE Standard for Verilog Hardware Description Language,* IEEE Standard 1364 -2005.

[34] G. Dimitrakopoulos and D. Nikolos, “High-Speed Parallel-Prefix VLSI Ling Adders,” *IEEE Transactions on Computers*, Vol. 54, pp. 225-231, 2005.

[35] M. P. Farmwald, *On the Design of High Performance Digital Arithmetic Units,* Ph.D. thesis, Stanford University, 1981.

VITA

Eric Charles Quinnell was born in Colorado Springs, Colorado on April 25, 1982, the son of Patricia Erdle MacIver and Charles Wallace Quinnell. After completing his work at Lewis-Palmer High School, Monument, Colorado, in 2000, he entered the University of Texas at Austin. He received the degrees of Bachelor of Science in Electrical Engineering and Master of Science in Electrical Engineering from the University of Texas at Austin in May 2004 and May 2006 respectively. During the following months, while continuing his studies in the Graduate School of the University of Texas, he began employment as an x86 floating-point circuit designer at Advanced Micro Devices in Austin, Texas.

Permanent Address: 11350 Four Points Dr. Apt 438, Austin, Texas 78726

This dissertation was typed by the author.