OPERAND-OPTIMIZED ASYNCHRONOUS FLOATING-POINT ARITHMETIC CIRCUITS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Basit Riaz Sheikh

January 2012

OPERAND-OPTIMIZED ASYNCHRONOUS FLOATING-POINT ARITHMETIC CIRCUITS

Basit Riaz Sheikh, Ph.D.

Cornell University 2012

Fast floating-point computations are critical in a wide range of applications. Today, the performance of these applications is limited by power constraints. The traditional power reduction schemes, which relied primarily on technology and voltage scaling, are not sufficient any more. In this thesis, we propose two novel asynchronous pipeline templates and multiple operand-dependent optimization techniques to significantly reduce the overall power consumption while preserving the average throughput.

Our novel pipeline templates reduce power consumption by minimizing the handshake circuitry and employing single-track handshake protocol. Noise and timing robustness constraints of our pipelined circuits are quantified across all process corners. A completion detection scheme based on wide NOR gates is presented, which results in significant latency and energy savings especially as the number of output tokens increase.

Furthermore, this thesis presents novel operand-dependent optimization techniques to improve the energy efficiency of IEEE-754 compliant floating-point adder and floating-point multiplier designs. Some of these optimizations are highly challenging, if at all possible, in a synchronous design because they increase the worst case critical path but on average have negligible impact on performance. To our knowledge, this is the first detailed design of high-performance asynchronous floating-point adder and floating-point multiplier.

Biographical Sketch

Basit Riaz Sheikh was born in Lahore, Pakistan. He joined the prestigious Aitchison College in Lahore in 1989, where he completed 12 years of his primary, middle, and high school education. He secured first position in his class in the *O-Level* examination of 1999. Due to his exceptional academic and extracurricular record, Basit was appointed the *Head Boy* of the entire student body comprising over two thousand students.

In the fall of 2001, Basit joined the National University of Singapore on *presidential scholarship* to pursue his undergraduate studies in Electrical Engineering. In 2002, he transferred to the University of Texas at Austin where he developed great interest in VLSI chip design. Basit graduated from the University of Texas in 2005 with B.S. in Electrical Engineering with highest honors.

Basit has been enrolled in the M.S. / Ph.D. program in the school of Electrical and Computer Engineering at Cornell University since August 2005. At Cornell, his research has primarily focused on energy-efficient VLSI chip design, arithmetic circuits, and computer architecture. Over the years, Basit gained invaluable industry experience while working at Applied Materials Inc., National Instruments Inc., and Intel Corporation in various positions ranging from software development to hardware design engineer. He is a member of Professor Rajit Manohar's Asynchronous VLSI lab.

For mom and papa

Acknowledgements

First of all, I would like to extend utmost gratitude to my advisor, Professor Rajit Manohar, who has been a great source of inspiration and guidance throughout my graduate studies. He encouraged me to explore unconventional ideas and helped a great deal in refining those into a cohesive research. Whenever I got stuck in my research, he came to my aid and pointed me in the right direction. It has been a great learning experience working under his supervision.

I would like to thank Professor David Albonesi for his guidance in the first few years of my graduate studies at Cornell and for taking keen interest in my progress throughout. I owe my gratitude to Professor Brian Evans and Professor Yale Patt at the University of Texas-Austin for encouraging me to pursue a PhD. They provided the motivation for what has been an intellectually stimulating and enlightening journey.

I want to thank all of my friends for supporting me during the rough and challenging phases of my PhD. Special thanks to Carlos Tadeo and Filipp Akopyan for their technical assistance with tool-flow infrastructure. I want to thank Paula Petrica for her endearing friendship and making six years in Ithaca a truly memorable experience.

My greatest gratitude to my family for their endless love and support. If it wasn't for their unflinching faith in me none of this would have been possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CMOS	Complementary Metal-Oxide Semiconductor
CSA	Carry Save Adder
FPA	Floating-Point Adder
FPM	Floating-Point Multiplier
LOD	Leading One Detection
LOP	Leading One Prediction
NMOS	n-diffusion Metal-Oxide Semiconductor
PCeHB	Pre-charge enable Half Buffer
PMOS	p-diffusion Metal-Oxide Semiconductor
QDI	Quasi-Delay Insensitive
WCHB	Weak Conditioned Half Buffer
VDD	Positive Power Supply Node
VLSI	Very Large Scale Integration

Chapter 1

Introduction

1.1 Motivation

Efficient floating-point computation is important for a wide range of applications in science and engineering. Using computational techniques for conducting both theoretical and experimental research has become ubiquitous, and there is an insatiable demand for higher and higher performing VLSI systems. Despite the remarkable advances in computing in the last few decades, the computing needs of many emerging applications in the fields of molecular biology, quantum chemistry, weather detection patterns, fluid dynamics, speech recognition, and financial services are far from being fully satisfied. Some of these emerging applications are essentially needed to address many critical global challenges. These include climate change, curing life-threatening diseases, discovering sustainable and alternative sources of energy, and predicting natural calamities to name a very few.

To meet the growing demands of some of these critical applications, today's faster super computers have exceeded the 1 *petaFLOP* (quadrillion floating-point operations per second) performance mark. Recently, Japan's K Computer [6] achieved a performance of 8.162 *petaFLOPS* to clinch the title of world's fastest supercomputer. But this performance came at the cost of 9.89 megawatts in power, which may be enough to power a small town. Furthermore, these power hungry systems require elaborate and expensive cooling systems to ensure proper operation. These results mean that the manufacturers of large, fast supercomputers and the VLSI chip designers of the underlying workhorse processors can no longer afford to design for performance only. Top 500 supercomputer ranking takes into account the energy-efficiency of the system as well as its performance.

Traditionally, VLSI designers primarily relied on CMOS technology and voltage scaling to reduce power consumption [11]. However, with the scaling of CMOS technology into ultra-deep sub-micron range, this no longer yields the desired power reduction. With the transistor threshold voltage fixed [28], V_{DD} has been scaling very slowly if at all, which means all performance improvements come at an increased energy consumption. Furthermore, process variations in deep sub-micron range have made devices far less robust, which is increasingly making it difficult for synchronous designers to overcome the problems associated with clock skew rates and clock distribution [18]. The findings of a recent in-depth study, to explore and devise ways to further scale supercomputer *petaFLOP* performance by 1000X, indicate the inadequacy of current design practices and technologies to achieve the desired throughput within a sustainable power budget [1]. This underscores a pressing need for alternate design practices, to reduce energy consumption for floating-point computations

while preserving robust behavior in advanced technology nodes, which is the core motivation behind this thesis.

At the other end of the spectrum, embedded systems that have traditionally been considered low performance are demanding higher and higher throughput for the same power budget to support compute-intensive floating-point applications that improve the user experience. Some of these applications include, but are not limited to, advanced gaming and animation softwares based on complex physics motion equations, voice recognition, facial simulation, graphics rending, advanced image and video processing applications. Since these applications have to be deployed on portable devices with limited battery-life, it is critical that we develop *energy-efficient* floating-point hardware for these embedded systems, not simply high performance floating-point hardware.

1.2 Operand Dependent Floating-Point Unit Design

Synchronous floating-point units, limited by worst-case computation delay, include complex circuitry to attain constant latency and throughput for the best, average, and worst case input patterns alike. Consider an N-bit adder circuit, for example. The delay of this adder depends on how fast carry reaches each bit position. In the worst case, the carry has to be propagated through all bits, which causes synchronous designers to consider complex, power-hungry adder designs to meet their stringent timing requirements. The biggest disadvantage of this design methodology is that it results in the same power consumption for the best case carry propagation of zero bit position, even though the best case could have been done much faster and more energy efficiently using much sim-

pler circuits. A preliminary input profile of a number of floating-point applications from SPEC [5] and PARSEC [9] benchmarks, shown in Figure 1.1, indicates that the worst case of carry propagation happens very rarely, if at all, in a 56-bit adder unit typically found in double-precision floating-point adder design.

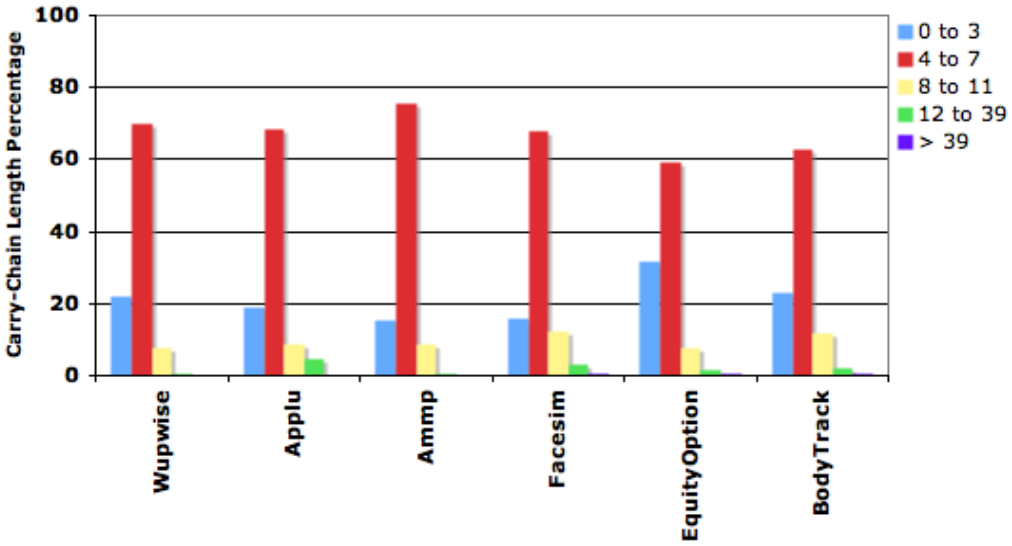

Figure 1.1: Longest carry-chain length in a 56-bit Adder

The observation that there are infrequently occuring cases that make the hardware difficult/slow leads to the natural question: can we design an energy-efficient *asynchronous* floating-point unit? An asynchronous circuit does not use a clock signal, and is not constrained to a global timing constraint. Perhaps we could design an IEEE-compliant floating-point unit that was a bit slower when certain infrequent cases occurred. This could result in a significant energy reduction during normal operation. Self-timing would enable this flexibility at a very fine grain, allowing for operand-dependent performance and energy consumption. In this thesis, we explore the possibilities of data dependent optimizations in floating-point arithmetic circuits.

1.3 Asynchronous Circuit Design

Asynchronous quasi-delay-insensitive (QDI) circuits, with their robustness to process variations, no global clock dependence, and inherent perfect clock gating, represent a highly feasible design alternative for future chip design. QDI circuits are also robust to voltage and delay variations, hence easing some of the design verification efforts. QDI circuits have been used in numerous high-performance, energy-efficient asynchronous designs [63] [23], including a fully-implemented and fabricated asynchronous microprocessor [42]. In this thesis, we harness the operand dependent execution and timing flexibility of asynchronous pipelined circuits to design and implement high-performance, energy efficient, asynchronous floating-point arithmetic circuits with truly data dependent performance and energy footprint.

QDI circuit templates, though robust, lose some of their energy efficiency gains in implementing handshakes between different parallel pipeline processes. To ensure QDI behavior for each handshake, every up and down transition within a pipeline is sensed, which leads to significant handshake circuitry and energy overhead. High throughput QDI pipelines only include a small amount of logic in each stage. The large number of pipeline stages required for high throughput make the handshake overhead a significant proportion of the total power consumption. In this thesis, we try to circumvent the problem of high handshake overhead in commonly used QDI pipelined circuits but without sacrificing robustness.

1.4 Thesis Contribution and Organization

The primary contributions of this thesis are listed as follows:

- Design of two novel pipeline templates, which greatly minimize the handshake circuitry of commonly used QDI templates by taking advantage of some easily satisfiable timing assumptions. Compared to QDI templates, the average throughput and latency are preserved, while the transistor area is greatly minimized. As it is in the case of QDI templates, the correctness of these proposed templates is not a function of input and output arrival times, which makes them very robust. We also present detailed design trade-off analysis of these templates to help future designers make appropriate pipeline selection based on their design constraints.
- Profiling results of various real life floating-point applications from many diverse fields. The bit-level input and computation patterns within several key logic blocks in the floating point unit datapath make a strong case for introducing operand-based optimizations for energy efficiency.
- The design and implementation of a first high-performance, energy-efficient, double-precision, asynchronous floating-point adder (FPA). The FPA is implemented using asynchronous QDI pipelines and is fully IEEE-754 standard compliant. It employs a number of novel operand-dependent optimization techniques to greatly reduce the circuit complexity and power consumption of various key logic blocks within the FPA datapath, while preserving average throughput and latency.
- A full-transistor level implementation of an asynchronous floating-point multiplier (FPM) datapath is presented. To our knowledge, our FPM is

a first high-performance, double-precision, asynchronous unit of its kind. It uses a mix of QDI and the newly proposed pipeline templates to improve energy efficiency of various key logic blocks within the datapath. A higher radix array multiplier design is introduced. The FPM provides full hardware support for difficult to implement special cases in the IEEE-754 standard with minimal complexity.

The rest of the thesis is organized as follows: In Chapter 2, we provide a background on IEEE-754 floating-point standard and several synchronous floating-point unit designs from academia and industry. It provides details of various floating-point applications that were profiled and outlines the key operations within the FPA and FPM datapaths. An introduction to asynchronous QDI pipelines is also presented. Chapter 3 introduces our novel energy-efficient pipeline templates. It provides a detail discussion on various design trade-offs of these templates and evaluates their efficacy using a non-trivial implementation of an 8x8 Booth-encoded array multiplier design. In Chapter 4, we present the design and implementation of our high-throughput, energy-efficient FPA unit. The application profiling results for floating-point addition operations are presented. All operand-dependend optimizations are discussed in detail. We conclude the chapter with an in-depth evaluation of our FPA across various input sets. Chapter 5 presents our FPM datapath design. It introduces a higher radix array multiplier which utilizes input patterns to significantly reduce overall circuit complexity and energy consumption while preserving the average throughput. Hardware implementation of special cases within the IEEE standard is also discussed. In Chapter 6, we summarize our key findings and their usefulness to future research.

Chapter 2

Background

In this chapter, we provide a background on floating-point computations, their various hardware implementations, and a set of diverse floating-point application benchmarks that we used for data dependent optimizations. The chapter also includes a brief introduction to asynchronous quasi-delay-insensitive (QDI) pipelines.

2.1 Floating-Point Computations

Today, most floating-point is IEEE-compliant or has an IEEE-compliant mode. A thorough background knowledge on what the standard entails is very important to understand the various trade-offs involved in floating-point hardware design.

2.1.1 IEEE Floating-Point Standard

The IEEE 754 standard [49] for binary floating-point arithmetic provides a precise specification of floating-point number formats, computation operations, and exceptions and their handling. This specification was determined after much debate, and it took several years before hardware vendors developed IEEE-compliant hardware. Part of the challenge was the belief that: (i) implementing most of the standard was sufficient; (ii) ignoring a few infrequently occuring cases led to more efficient hardware (e.g. [33]). Unfortunately ignoring certain aspects of the standard can lead to unexpected consequences in the context of numerical algorithms. Today, most floating-point hardware is IEEE-compliant or has an IEEE-compliant mode.

The IEEE format specifies two main groups of floating-point format: *single-precision* and *double-precision*. In this thesis, we primarily focus on double-precision format since it is commonly used in most scientific and emerging applications. Figure 2.1 depicts the 64-bit double-precision floating-point number format. It comprises 1-bit of sign, 11-bits of exponent, and 52-bits of mantissa (also known as the significand).

The value of a normalized number, X, being represented is as follows:

$$X = (-1)^S * 1.M * 2^{(E-bias)}$$

where S is the value of the sign bit, M corresponds to the mantissa bits, E corresponds to the exponent bits, and *bias* has a value of 1023 in double-precision floating-point format. The standard also specifies the format of a *denormal* number to represent the result of the computation whose value is between the smallest possible representation of a normalized number and zero. The value of a

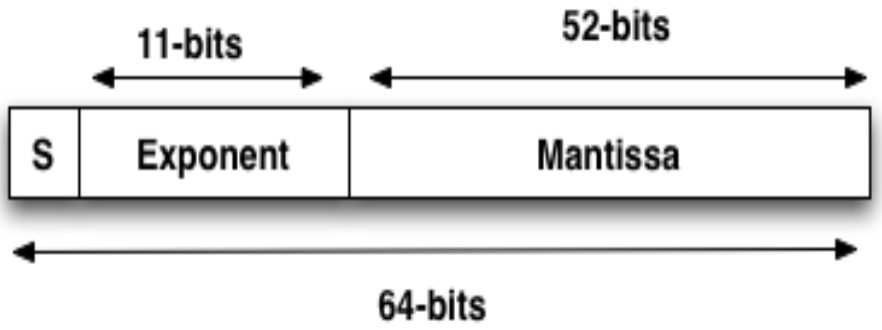

Figure 2.1: Double precision floating-point format

denormal number, X, being represented is as follows:

$$X = (-1)^S * 0.M * 2^{(1-bias)}$$

and differs from a normal number in that there is no implied bit and the exponent with a value of zero is forced up by 1 to *Emin*, which is equal to -1022 in double-precision format. The format specifies other special types such as Not-a-Number (*NaN*), $+\infty$, and $-\infty$. These special cases are detected by checking the exponent and mantissa bits. For *NaN*, all exponent bits are one and the mantissa is non-zero. Similarly, ∞ is indicated by an exponent comprising one in each bit position but with zero mantissa. A denormal number is indicated by a zero exponent and a non-zero mantissa, whereas zero input is detected when all exponent and mantissa bits are zero. The IEEE format also includes four different rounding modes, which specify how to deal with inexact floating-point outputs.

The combination of a vast range of inputs, special cases, and rounding modes makes the hardware implementation of fully IEEE 754 standard compliant floating-point arithmetic a very challenging task.

2.1.2 Floating-Point Adder

A floating-point adder is used for the two most frequent floating-point operations: addition and subtraction. It requires much more circuitry to compute the correctly normalized and rounded sum compared to a simple integer adder. All the additional circuitry makes the FPA a complex, power-consuming structure. Figure 2.2 shows the FPA datapath for two double-precision 64-bit inputs.

The following summarizes the key operations required to implement an IEEE-compliant FPA:

- The first step in the FPA datapath is to unpack the IEEE representation and analyze the sign, exponent, and significands bits of each input to determine if the inputs are standard normalized or are of one of the special types (NaN, Infinity, Denormal).
- Prior to actual addition or subtraction, the absolute difference of the two exponents is used as the shift amount for a variable right shifter which aligns the smaller of the operands.
- In parallel with the right align shifter, the guard, round, and sticky bits are computed to be used for rounding in latter stages of the FPA datapath.
- The next step is the addition or subtraction of two significands based on sign information.
- Most high-performance FPAs use a special-purpose circuit popularly known as a Leading-One-Predictor and Decoder (LOP/LOD) to predict the position of the leading one in parallel with the addition/subtraction step.

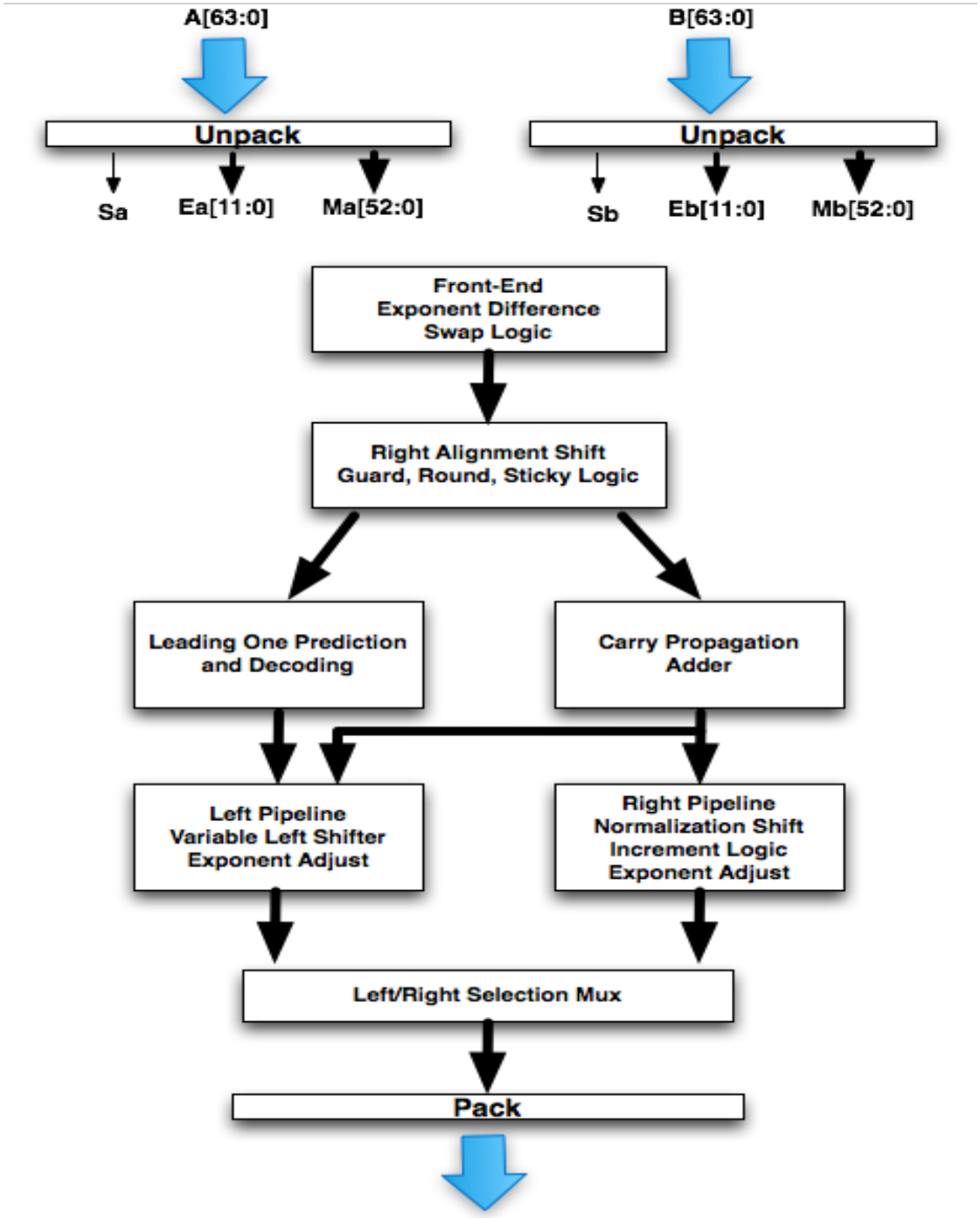

Figure 2.2: Floating-point Adder Datapath

- The post addition steps include normalizing the significands. This may require either a left shift by a variable amount (using the predicted value from LOP), no shift (if the output is already normalized), or a right shift by one bit (in case of carry-out when the addition inputs have the same sign).
- The exponent is adjusted based on the shift amount during normalization. In parallel, the guard, round, and sticky bits are updated and are used, along with the rounding mode, to compute if any rounding is necessary. The sign of the sum is also computed.
- In case of rounding, the exponent and significand bits are updated appropriately.
- The final stage checks for a NaN, Infinity, or a Denormal outcome before producing the correct result.

The complexity of the FPA datapath is not dominated by any single large logic block, but instead it is distributed across multiple logic blocks. This necessitates the need to optimize all blocks to gain significant improvements in energy efficiency.

2.1.3 Floating-Point Multiplier

In terms of micro-architectural complexity, for operations involving normal inputs, the floating-point multiplier (FPM) datapath is relatively simpler than the FPA datapath. It does not require logic blocks such as *Right Alignment Shifter*, *Leading One Detection and Prediction unit*, and *Variable Length Normalization Shifter*, which increase the complexity of FPA datapath. The denormal and

underflow operations in the FPM may require variable length shift for normalization. We discuss these operations and their hardware requirements later in a separate section. The FPM datapath for double precision multiplication operation is shown in Figure 2.3.

The following summarizes the key steps in an FPM datapath:

- The first step in the FPM datapath is to unpack the IEEE representation and analyze the sign, exponent, and mantissa bits of each input to determine if the inputs are standard normalized or are of one of the special types (NaN, infinity, denormal).
- The mantissa bits are extended with the implicit bit. It is set to one for normal inputs and zero for a denormal input.
- The 53-bit long mantissas of both inputs are used to generate partial products corresponding to a 106-bit product. Since high throughput and low latency are of essence in floating-point applications, most FPMs use some form of an array multiplier, such as a booth-encoded multiplier as shown Figure 2.3, to meet the performance demands. Most array multipliers employ an array of carry-save-adders (CSAs) [71] to reduce the large number of partial products to two final full product-length bit streams.
- The most significant 53-bits of the two output bit streams from the CSA array are summed up using a carry propagation adder (CPA) to generate a 53-bit mantissa. The least significant 53-bits are used to generate the carry input to the CPA as well as compute the guard, round, and sticky bits to be used in post normalization rounding.
- In parallel, the exponent logic computes the resulting exponent, which is a sum of the exponent values of both inputs minus the bias. The bias has

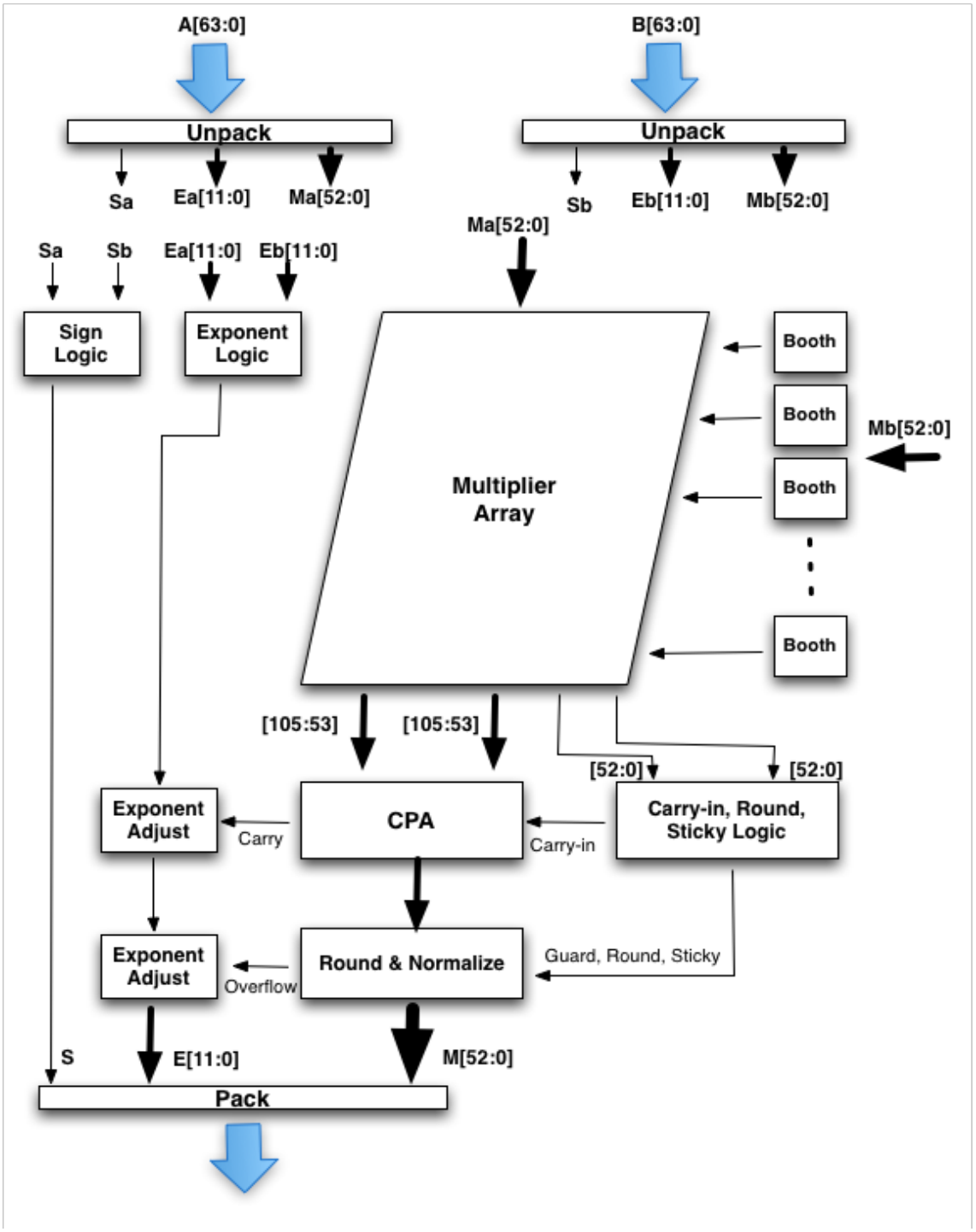

Figure 2.3: Floating-point Multiplier Datapath

a value of *1023* in case of double-precision operations. The sign of final product is also computed.

- The post multiplication step includes normalization of the 53-bit mantissa. For normal inputs and non-underflow cases, either the mantissa is already normalized or it may require a right shift by a single bit position, in which scenario the exponent is adjusted, in parallel, by adding one to it. The guard, round, and sticky bits are updated and are used, along with the round mode, to determine if the product needs to be rounded or not.
- In case of rounding, the mantissa is incremented by one. If rounding yields a carry out, the exponent is adjusted by adding one to it and right shifting the mantissa by one bit position.
- The final stage checks for a NaN, infinity, or a denormal outcome before outputting the correct result in the IEEE format.

With normalization step limited to a simple shift of no more than one-bit position and the exponent logic comprising only 11-bit long arithmetic, the FPM's complexity is largely a function of its *53x53 multiplier*, *sticky bit computation block*, and the final *carry propagation adder*. In this thesis, we present various structural and circuit-level optimization techniques to reduce the complexity and power consumption footprint of the aforesaid logic blocks.

2.2 Floating-Point Application Benchmarks

The high complexity of synchronous floating-point hardware arises out of the need to compute the worst case floating-point operation within a stringent timing margin. The important question to ask is how often the worst-case happens.

If it happens very frequently then it justifies burning extra power with complex circuits to boost overall performance.

To answer this question, we used Intel's PIN [38] toolkit to profile input operands in a few floating-point intensive applications. This profiling analysis gave us a great insight into the average case input properties in various real life applications. We exploit this knowledge to provide a number of operand dependent optimizations, which enable us to use simple asynchronous circuits to meet our performance targets at a much reduced energy consumption.

For the floating-point adder (FPA) design, we profiled the followed application benchmarks from SPEC2006 [5] and PARSEC [9] benchmark suites using reference input sets:

- *447.deal*: This C++ program utilizes a specialized program library targeted at adaptive finite elements and error estimation. It has application in the fields of fluid flow, electro-magnetics, acoustics, and general relativity to name a few.
- *444.namd*: It simulates large bio-molecular systems. Most of the runtime is spent calculating inter-atomic interactions in a small set of functions.
- *416.gamess*: It implements a wide range of quantum chemical computations.
- *450.soplex*: It solves a linear program using the simplex algorithm. Like most other implementations of the simplex algorithm, it employs algorithms for sparse linear algebra.
- *482.sphinx*: This is a widely-known speech recognition system from Carnegie Mellon University.

- *453.povray*: It is an image rendering, ray-tracer program. Intersections of rays with geometry objects are computed by solving complex mathematical equations using numerical methods or directly.
- *437.leslie3d*: It is a computational fluid dynamics solver used to investigate turbulence phenomena such as mixing, combustion, and acoustics.
- *facesim*: It simulates the motions of human face using underlying physics motion equations.
- *bodytrack*: A computer vision application which tracks the human body with multiple cameras. It has applications in video surveillance and character animation fields.
- *swaptions*: A financial application which uses Monte Carlo simulation to price a portfolio of swaptions.

For the floating-point multiplier (FPM) design, we profiled three more applications listed below:

- *FFT*: It measures the floating-point rate of execution of double-precision complex three-dimensional Discrete Fourier Transform [2].
- *LINPACK*: It is a collection of subroutines that analyze and solve linear equations and linear least-squares problems [4]. These subroutines are commonly used to characterize the performance of highest performing supercomputers.
- *SSCA*: A graph theory benchmark representative of computations in informatics and national security [3].

Although, all these applications come from very diverse fields, the underlying computations have their roots in similar mathematical and physics principles. Our goal is to exploit the existing common input patterns in all these benchmark applications to improve energy efficiency and reduce floating-point unit's circuit complexity. The input operands in actual benchmark runs were saved to disk, and then used for statistical analysis. The application profiling statistics in the following chapters were tabulated using ten billion input operands for each application.

2.3 Related Work

2.3.1 Asynchronous Arithmetic

The use of asynchrony to improve the performance of arithmetic circuits has been exploited by a number of different researchers. As early as 1946, von Neumann proposed using an asynchronous integer adder because the average-case delay for a ripple-carry adder is $O(\log N)$ where N is the number of bits in the input assuming that the input bits are independent, identically distributed (i.i.d.) random variables [13]. More recently it was shown that it is possible to design an asynchronous integer adder with an average-case latency of $O(\log \log N)$ for i.i.d. inputs [40] and that the design achieves the optimal asymptotic average-case latency for any input distribution [39]. There have been numerous papers on asynchronous adders with a variety of topologies (e.g. [42, 31, 25, 46]).

In terms of the multiplier design, the delay variability nature of iterative multipliers makes them a popular choice amongst asynchronous design-

ers [19, 30]. An iterative multiplier utilizes a few functional units repeatedly to produce the result. In a simple iterative n by n multiplier implementation, where n is the number of bits, the product is computed after n iterations. Each iteration comprises a minimum n-bit addition and a serial shift by one-bit position. Furber et al [37] proposed a low power integer multiplier which exploits the commonly occurring pattern of low number of significant bits in integer inputs as means to reduce the total number of iterations. These iterative multiplier designs, though highly energy efficient and compact in terms of area, are not feasible to be used in a floating-point multiplier hardware due to their very high latency and low throughput and the fact that unlike the inputs in integer arithmetic, the most significant bits of floating-point mantissa inputs are non zero.

To our knowledge, the work of Joel Noche et al. [45] is the only published work on floating-point unit design using asynchronous circuits. Their work claims a full working single-precision floating-point unit (FPU). However, their FPU is completely non-pipelined, doesn't include any energy optimization techniques, and does not implement rounding logic. Their FPU has many orders of magnitude higher latency compared to all recent floating-point designs from synchronous domain. Their test vector for a floating-point addition operation included one addition of two arbitrary single-precision floating-point inputs for which they claim a completion time (latency) of 79 nanoseconds in a 0.35μm process at 3.3V. For floating-point multiplication, they report a latency of 465 nanoseconds.

2.3.2 Synchronous Floating-Point Adders and Multipliers

There is a large body of work on synchronous FPA and FPM design. Ercegovac and Lang [21] contains an overview of the different techniques used to optimize floating-point addition and multiplication.

Most of the earlier work on the FPA design has focused on improving FPA latency [61, 7, 48, 47]. Oberman [47] proposes the use of two align shifters to improve the latency of their single-precision FPA with only one rounding mode. Seidel and Even [61] propose a two-path FPA design to reduce overall latency. The R-path in their design deals with cases of effective addition (or subtraction with exponent difference greater than 1) and N-path deals with effective subtraction with exponent difference less than or equal to 1. Both paths are in operation at the same time and use their own significand adders.

There is less work on low-power FPAs compared to low-latency FPA design. Pillai et al. [52] propose the partitioning of the floating-point datapath into three distinct, clock-gated datapaths for activity reduction. Only one of the three paths is active during any operational cycle in their FPA. In our proposed transistor-level optimized asynchronous FPA, we also use control-inhibited pipelines but instead of using clock-gating to turn off the pipelines (which may worsen clock skew especially for high performance FPAs in deep submicron technologies) we use local asynchronous conditional split pipelines which have no effect on overall throughput. Also, our design goes beyond pipeline inhibitions as explained in sections 2.2 and 4.3. The FPA design by Quinnell et al. [54] is one of the rare fully-implemented designs (65nm SOI) from academia. Although, they use standard-cell library as opposed to our custom transistor-level construction, their work provides us with a good baseline

to analyze our throughput and power results.

For synchronous FPM designs, the focus of prior work has been the array multiplier block, which is the single largest logic structure within the FPM datapath. Earlier designs have employed various architecture and circuit-level optimizations to reduce array multiplier latency and increase its throughput [73, 57, 48, 50]. However, there is relatively much less work on improving the energy efficiency of multiplier datapath [15], which is one of the primary contributions of this thesis. Traditionally, technology and voltage scaling has been deemed sufficient to provide the necessary reductions in energy consumption every few years. This is no longer the case any more. In this thesis, we propose a number of data dependent optimizations, both at the architectural and circuit-level, to significantly improve the energy efficiency of our FPM datapath.

Recent years have seen a number of contributions in the design of Fused-Multiply-Add (FMA) units [69, 54, 35, 60]. In [35], the authors propose techniques to reduce the latency of a floating-point addition operation in an FMA. In terms of performance and power-efficiency, the P6 Binary Floating-Point Unit [69] represents the state-of-the-art. It supports an extremely aggressive cycle time of 13FO4s. Power saving is done by clock-gating pipeline stages not in use. Power simulations at 1.1V, 4GHz, and 100% utilization in a 65nm SOI process consumed $310mW$.

2.4 Asynchronous Pipelines

High performance asynchronous circuits are composed of many parallel processes. As opposed to synchronous circuits, which use a global clock to synchronize data tokens between different pipeline stages, these asynchronous parallel processes use handshake protocols to communicate with each other. These parallel processes are often referred to as fine-grain pipelined circuits. The fine-grain pipelined circuits use designated channels for communication between processes. A channel comprises a bundle of wires and a communication protocol to transmit data from a sender to a receiver. There are numerous asynchronous fine-grain pipeline implementations [36] [72] [64] [24]. A robust family of these circuit templates is referred to as quasi-delay-insensitive (QDI) circuits.

2.4.1 Quasi-Delay-Insensitive Circuits

QDI circuit templates use 1-of-N encoded channels to communicate between different parallel processes. In an 1-of-N channel, a total of N wires is used to encode data with only one wire asserted at a time. Most high throughput QDI circuits either use 1-of-2 (dual-rail) or 1-of-4 encodings. In an 1-of-4 encoded channel communication as shown in Figure 2.4, *validity* is signified by setting one of the four data rails and *neutrality* is indicated by resetting of all four data rails. In a four phase handshake process, which is commonly used in most high speed QDI circuits, the sender process initiates the communication by sending data over the rails i.e. by asserting one of the data rails. The receiver process detects the presence of data and sends an acknowledge once it no longer needs the data. At this point, the sender process resets all its data rails. The receiver

process detects the neutrality of input tokens. It de-asserts the acknowledge signal once it is ready to receive a new data token. The cycle repeats.

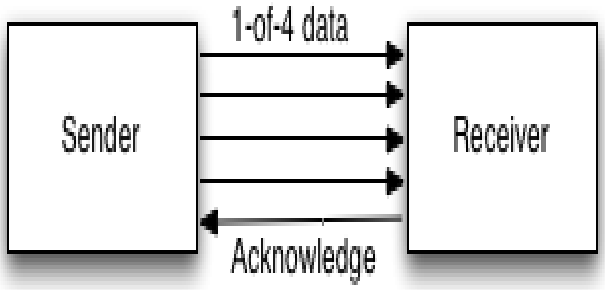

Figure 2.4: Asynchronous pipelines: sender-receiver handshake protocol.

Weak-Condition Half-Buffer

The weak-conditioned half-buffer (WCHB) template is an energy efficient QDI pipeline template. Figure 2.5 shows a dual-rail WCHB pipeline along with a transistor-level depiction of a two input C-element gate which is used in a WCHB template. It is a simple buffer with dual-rail input token *L* and dual-rail output token *R*. The signal *L.e* is the inverted sense of the acknowledge signal seen in Figure 2.4.

A WCHB pipeline satisfies the *weak conditions* [62] i.e. the output being valid implies that the input is valid (checked by the NMOS logic stack of C-element), and the output being neutral implies that the input is neutral (checked by the PMOS logic stack of C-element). For logic computations requiring more than 2 inputs, a WCHB template requires too many stacked PMOS transistors, which makes it slower, more susceptible to noise, and less energy efficient. Therefore,

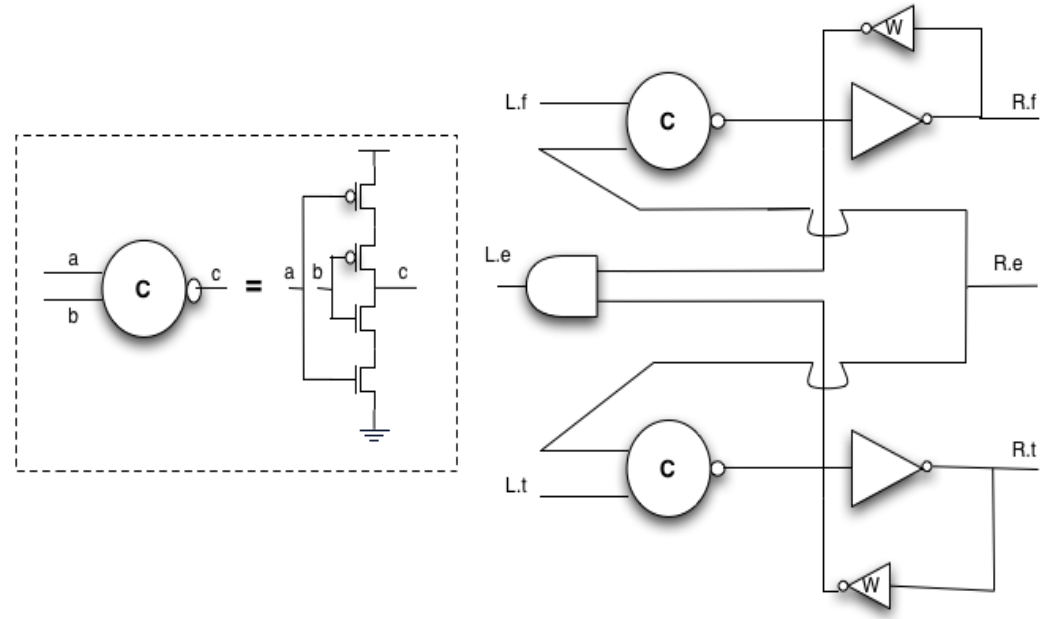

Figure 2.5: A WCHB pipeline stage

use of WCHB templates is usually limited to simple buffers and copy operations.

Pre-Charge enable Half-Buffer

The pre-charge enable half-buffer (PCeHB) [22] template, which is a slightly modified version of pre-charge half-buffer (PCHB) template proposed in [36] [72], is a workhorse for most high throughput QDI circuits. It is both small and fast with a cycle time of 18 transitions. In a PCeHB pipeline, the logic function being computed is implemented by a pull-down NMOS stack. The input and output validity and neutrality are checked using separate logic gates. The actual computation is combined with data latching, which removes the overhead of explicit registers.

A PCeHB template can take multiple inputs and produce multiple outputs. Figure 2.6 shows a simple two input and one output PCeHB template. *L0* and *L1* are dual-rail inputs to the template and *R* is a dual-rail output. A PCeHB template has a forward latency of two transitions. Each pipeline stage computes logic by using a NMOS pull-down stack followed by an inverter to drive the output.

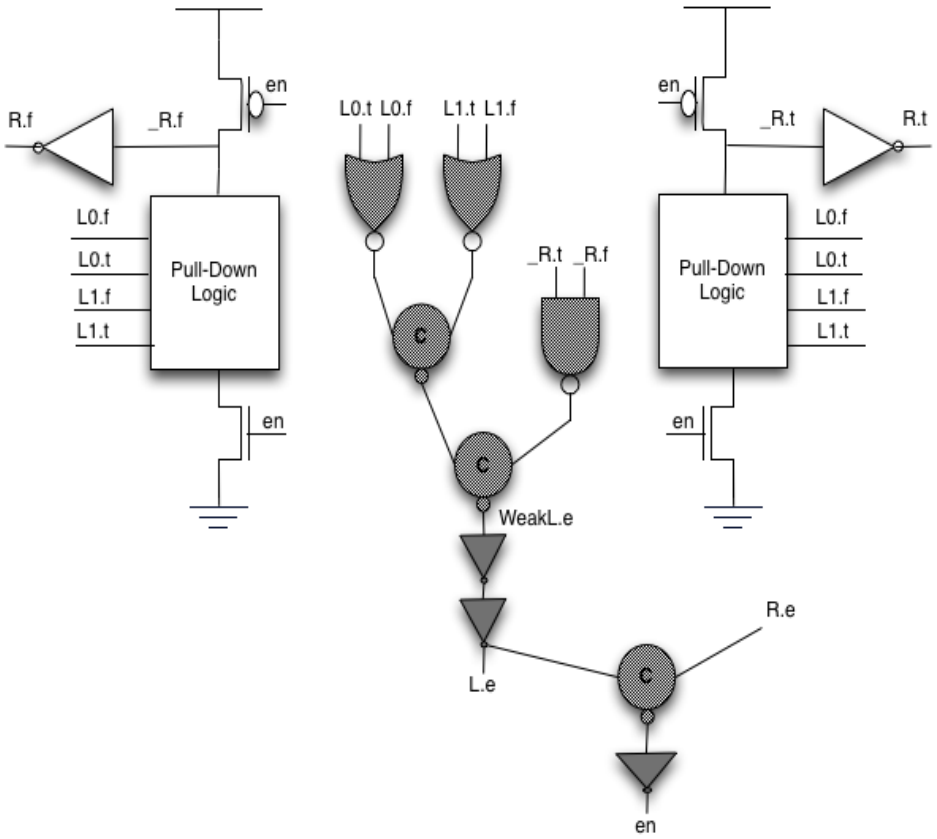

Figure 2.6: A two input and one output PCeHB template.

To understand the cycle time of 18 transitions in a PCeHB template, let us assume two PCeHB pipelines in series with time (t) increments taken in terms of logic transitions.

- At $t = 0$, input tokens arrive at the first PCeHB pipeline block.
- At $t = 2$, first pipeline block produces its output.
- At $t = 4$, second pipeline block produces its output.
- At $t = 5$, *L.e* in the first block goes low.
- At $t = 7$, *L.e* in the following block, which is the *R.e* of the first block, goes low. This indicates that the output from the first pipeline block is no longer needed and can be reset.
- At $t = 9$, *en* signal in the first block is de-asserted.
- At $t = 10$, $\_R$ rails in the first block are pre-charged.
- At $t = 11$, output, *R*, rails of the first block are reset.
- At $t = 12$, $\_R$ rails in the second block are pre-charged.
- At $t = 14$, *L.e* in the first block goes high.
- At $t = 16$, *L.e* in the second pipeline stage goes high. This indicates the neutrality of the inputs in the second pipeline stage.
- At $t = 18$, *en* is set in the first pipeline block, which indicates that the pipeline is ready to accept new input tokens and compute a new output.

The highlighted logic gates in Figure 2.6 are not used for the actual computation but are only required for the handshake protocol. This includes the generation of completion detection signal (*L.e*) as well as the *en* signal that is used to enable computation or latching in the pipeline stage. For high-througput circuits, each PCeHB stage contains only a small amount of logic with only a few inputs and outputs.

As the number of inputs into a PCeHB pipeline stage increases, the input validity tree becomes more complex and may require extra stages to compute, which leads to an increase in the cycle time. The same holds true as the number of outputs increase. Hence, for high-througput circuits each PCeHB stage contains only a small amount of logic with only a few inputs and outputs. This leads to significant handshake overhead, in terms of power consumption and transistor count, as tokens may have to be copied for use in separate processes with each process doing its own validity and neutrality checks.

Figure 2.7 shows the power consumption breakdown of a simple full-adder circuit implemented using a PCeHB template. Only 31% of the total power is consumed in the actual logic, while the rest is spent in implementing the handshake protocol. This is a significant power overhead, which gets worse as the complexity of PCeHB templates increases with more inputs and outputs. The result in Figure 2.7 was one of the main motivating factors that prompted us to consider alternative pipeline solutions with less handshake circuitry. These alternative templates are discussed and analyzed in great detail in the next chapter.

2.4.2 Fine-grain bundled-data pipelines

The fine-grain bundled-data pipelines have an instant area advantage over the QDI pipelines because of their use of single-rail encoded data channels [64]. However, the bundled-data pipelines include far more timing assumptions than QDI circuits which makes them less robust. The bundled-data pipelines contain a separate control circuitry to synchronize data tokens between different

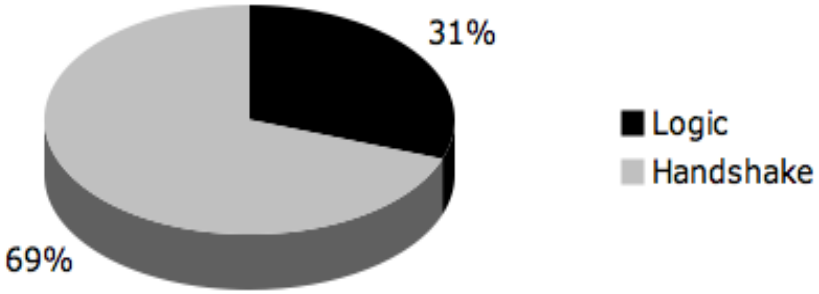

Figure 2.7: Power breakdown of a full-adder circuit in a PCeHB pipeline.

pipeline stages. The control circuitry includes a matched delay line, the delay of which is set to be larger than that of the pipeline's logic delay plus some margin. In [64], for correct operation, the designer has to ensure that the control circuit delay satisfies all set-up and hold time requirements just like in synchronous design. Since our goal was to design pipeline templates with robust timing and with forward latency similar to that of precharged logic, we did not consider any bundled-data pipeline implementations in this thesis.

Chapter 3

Energy-Efficient Pipeline Design

In this chapter, we present two novel energy-efficient pipeline templates for high throughput asynchronous circuits. The proposed templates, called N-P and N-Inverter pipelines, use single-track handshake protocol. There are multiple stages of logic within each pipeline. The proposed techniques limit handshake overheads associated with input tokens and intermediate logic nodes within a pipeline template. Each template can pack significant amount of logic in a single stage, while still maintaining a fast cycle time of only 18 transitions. Noise and timing robustness constraints of our pipelined circuits are quantified across all process corners. A completion detection scheme based on wide NOR gates is presented, which results in significant latency and energy savings especially as the number of outputs increase.

Three separate full transistor-level pipeline implementations of an 8x8-bit booth-encoded array multiplier are presented. Compared to a standard QDI pipeline implementation, the N-Inverter and N-P pipeline implementations re-

duced the energy-delay product by 38.5% and 44% respectively. The overall multiplier latency was reduced by 20.2% and 18.7%, while the total transistor width was reduced by 35.6% and 46% with N-Inverter and N-P pipeline templates respectively.

3.1 Improving Energy-Efficiency of Fine-Grain Pipelines

QDI circuits are robust since each up and down transition within a QDI pipeline template is sensed. But this robustness comes at the cost of significant power consumption in pipeline handshake circuitry as shown in Figure 2.7. The high handshake overhead is one of the serious constraints hampering the wide-range adoption of QDI circuits especially for logic operations with a large number of input and output signals, such as a 32-bit multiplier.

Our goal is to improve the energy efficiency of high performance asynchronous pipelines but without sacrificing robustness. To this end, we kept the following objectives for our resulting pipeline templates:

- Keep the cycle time of each stage within 18 transitions.
- Increase the ratio of logic to handshake. The handshake power overhead must account for less than 50% of total pipeline power.
- No increase in the total transistor count is allowed.
- All timing assumptions are either isochronic fork assumption [41] or have at least the same timing margin as the half-cycle timing assumption [34] according to which the difference in number of transitions between any two delay races must be at least 4.5 transitions.

- Stalls on input and output should not impact correct operation.

We envision these circuits being used for large chunks of local logic (e.g. a multiplier) wrapped with QDI interfaces, rather than globally.

In the past, researchers have tried to increase the logic density of QDI pipelines by adding extra logic stages [8], but this still does not yield the desired reduction in the handshake overhead and leads to an increase in cycle time. To analyze this effect, let us suppose we increase the logic depth of a pipeline by adding extra logic stages. To conform to QDI behavior, the up and down transitions of all newly-created internal signals must be acknowledged. This can be done either by explicitly checking for each transition using completion detection logic as is done in the PCeHB template or using *weak conditions* [62] i.e. the output being valid implies that the input is valid (checked by additional n-fets in the logic stack), and the output being neutral implies that the input is neutral (checked by additional p-fets in the logic stack). The limitations of *weak conditions* for performance are elaborated in [62] [36]. In the case of explicit checking, there is the associated high handshake overhead because of all the extra validity and neutrality detection logic gates. All these extra transitions associated with the newly added logic stages and completion detection logic gates limit energy efficiency gains.

There is clearly a need to look beyond just adding extra logic stages to each pipeline stage. To improve the energy efficiency of high throughput asynchronous pipelines, we look at alternative handshake protocols as well as some timing assumptions in QDI circuits.

3.1.1 Four phase handshake vs. Single-track handshake

In a four phase handshake protocol, the pipeline stage needs to detect the validity and the neutrality of both inputs and outputs. During the second half of the four-phase protocol when the pipeline is waiting for inputs and outputs to be reset, no actual logic is being computed but it still consumes roughly half of the cycle time. Furthermore, the power consumed in detecting the neutrality of inputs and outputs rivals that consumed during their validity detection. Due to these characteristics, the four phase handshake protocol is clearly not an ideal choice for energy efficiency.

Single-track handshake [70] protocol tries to overcome this weakness of four phase protocol by practically eliminating the neutrality phase. Figure 3.1 shows an overview of a single-track handshake protocol. The sender process initiates the communication by sending the data token. The receiver uses the data for computing its logic. Once the data is no longer needed, instead of sending an acknowledge signal back to the sender process, the receiver process resets the input tokens itself by pulling the data wires low through NMOS transistors as illustrated in Figure 3.1. There are as many NMOS discharge transistors as there are data wires, but for simplicity we show only one discharge transistor in Figure 3.1. As the data wires pulled low, the sender detects the token consumption and gets ready to send the next token. Hence, eliminating the transitions associated with second part of the four phase protocol.

There has been very limited work on single-track handshake templates. Most of the prior work has focused on using single-track handshake protocol to reduce the cycle time of asynchronous pipelines to less than 10 transitions and not on how to use these extra transitions to improve logic density and en-

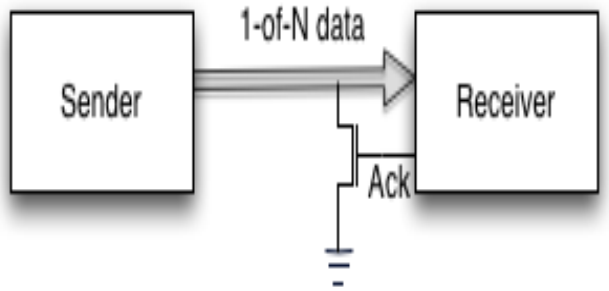

Figure 3.1: Single-track handshake protocol.

ergy efficiency. Ferretti et al [24] provide a family of asynchronous pipeline templates based on single-track handshake protocol. Just like high throughput QDI circuits, each of their pipeline templates contains only a small amount of logic. Furthermore, their 6-transition cycle time pipelines use some very tight timing margins that may require significant post-layout analog verification. Single-track circuits have been used in the control path of GasP [64] bundled-data pipelines. However, the actual data path of the pipeline does not use a single-track handshake protocol.

We employ single-track handshake protocol for our proposed pipeline templates. However, our design effort focuses on increasing the logic density and energy efficiency of each pipeline stage and not on reducing cycle time.

3.1.2 Relative Path Timing Assumption

QDI circuits are highly tolerant of process variations as each transition within a QDI pipeline is sensed. The isochronic fork assumption [41], which states that

the difference in delay between branches of a wire is insignificant compared to the gate delays of the logic reading their values, is the only timing assumption allowed in QDI design. Recently, LaFrieda et al [34] exposed another timing assumption that is quite commonly used in QDI implementations, which they named as the half cycle timing assumption (HCTA). According to HCTA, the difference in number of transitions between any two delay races must be at least 4.5 transitions for PCeHB-style templates. The resulting templates are referred to as Relaxed QDI templates and are shown to be quite robust.

LaFrieda et al [34] exploited HCTA to improve energy efficiency of their four phase handshake protocol pipelines. In this work, we look to improve energy efficiency of single track handshake protocol pipelines by introducing timing assumptions with a margin of at least 5 gate transitions between any two relative delay races.

3.2 High Throughput Energy Efficient Pipeline Templates

3.2.1 N-P and N-Inverter Pipeline Templates

We use single-track handshake protocol for our proposed pipeline templates. Figure 3.2 shows the gate-level depiction of our first proposed template with 5 arbitrary dual-rail outputs indicated by signals *R0* to *R4*. We have named the template N-P pipeline since it computes logic using NMOS pull-down and PMOS pull-up stacks. Each NMOS and PMOS stage can comprise multiple logic stacks. However, for simplicity, we do not show multiple logic stacks and global reset signals.

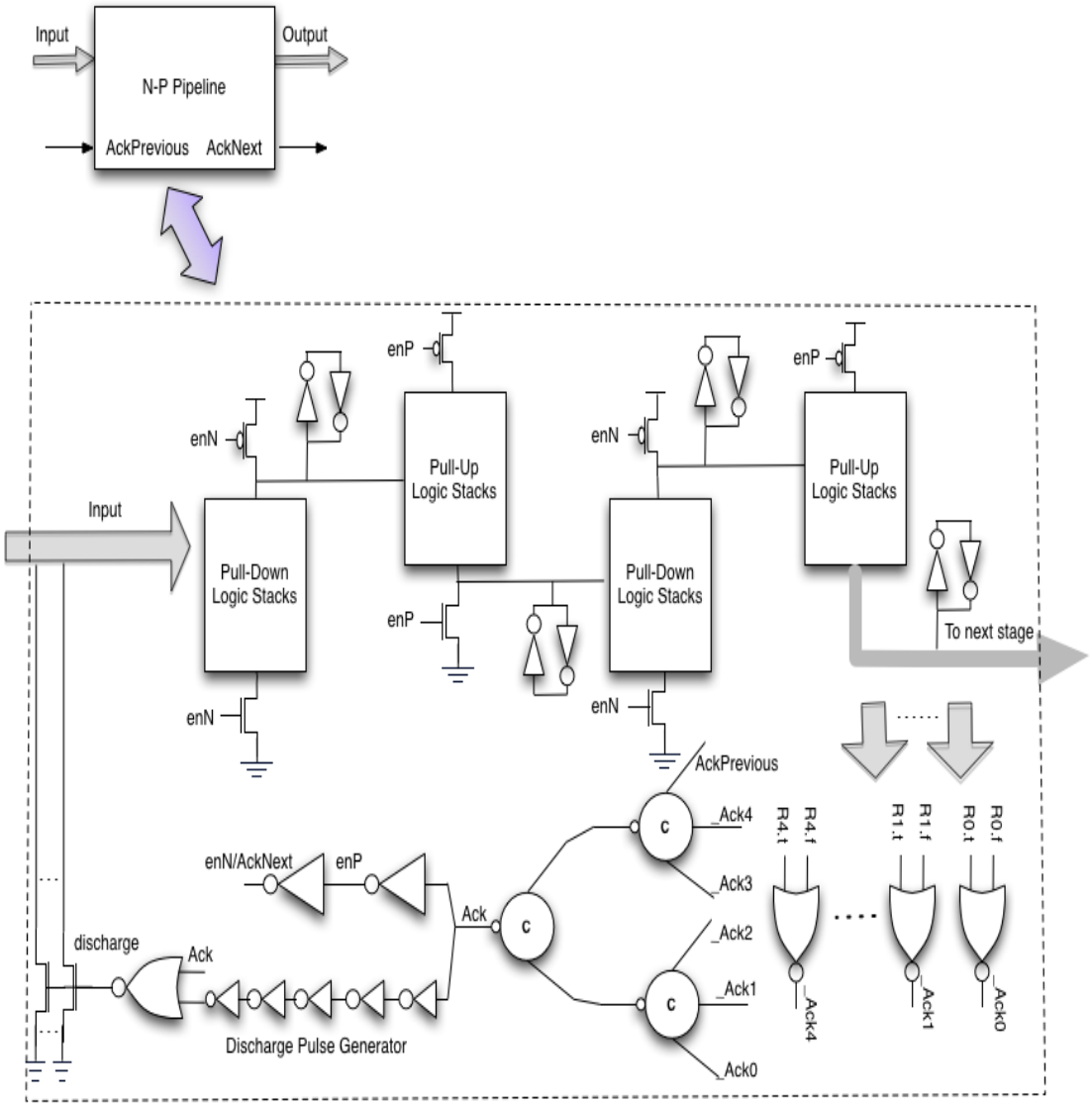

Figure 3.2: N-P pipeline template

A PCeHB template has two logic stages per each pipeline, with the second logic stage comprising an inverter to drive the output rails. Hence, there is only one effective logic computation per pipeline block. In contrast, the N-P template has *N* arbitrary stages of actual logic computations. However, for ease of explanation and to keep cycle time within 18 transitions, we use N-P pipelines

with four stages of logic. In the reset state, the NMOS logic nodes in the pipeline are precharged, whereas the PMOS logic nodes are pre-discharged. Each state-holding gate includes a staticizer, which comprises a keeper and a weak feedback inverter, to ensure that charge would not drift even if the pipeline were stalled in an arbitrary state. The staticizers, drawn as two cross-coupled inverters, for the intermediate as well as the final output nodes are shown in Figure 3.2.

When 1-of-N encoded input tokens arrive, logic is computed in the first stage by pulling down the precharged nodes. This is similar to how logic is computed in QDI templates. We limit the number of series transistors in an NMOS stack to a total of four. The second logic stage uses a stack of PMOS transistors to compute logic by pulling up the pre-discharged nodes. As the PMOS transistors have slower rise times, for throughput purposes we limit the number of series transistors in a PMOS stack to a total of three (including the enable). As the output nodes from the second stage pull up, the pull-down stacks in the third stage get activated and compute logic by pulling down their output nodes. Finally, the fourth stage computes logic by using its pull-up stack of PMOS transistors. The four cascaded stages of logic in our pipeline are similar to cascaded domino logic but without any static inverters in between dynamic logic stages.

There are no explicit validity detection gates for the arriving input tokens nor for any intermediate outputs that are being produced. *AckPrevious* (explained later in this section) signifies the validity of input tokens into the pipeline and alleviates the need to explicitly check for validity. For intermediate outputs produced and consumed within the template, validity must be embedded in a pull-up or pull-down logic stack that uses the intermediate output to compute the

following stage logic output. This could incur additional cost, depending on the function being implemented. However, for a logic stack inherently embedded with input validity, for example a stack that computes the sum of two inputs, there is zero validity detection overhead. The elimination of explicit validity detection gates for input tokens and intermediate output nodes leads to considerable power savings and minimization of handshake overhead.

There is an explicit completion detection logic for all the outputs that eventually leave the pipeline, either at the end of the second stage or the fourth stage. The completion detection of the final outputs automatically signifies the validity of all intermediate outputs as well as that of all the initial input tokens into the N-P pipeline. The completion detection logic comprises a set of NOR gates and a c-element tree as shown in Figure 3.2. Each of the c-element gates includes a staticizer in parallel. These staticizers are not shown for simplicity. The outputs from the NOR gates are combined using a c-element tree which de-asserts the *Ack* signal once all outputs are valid. This leads the discharge signal to go high, which initiates the reset of all input tokens. The discharge signal is only set for a short pulse duration. The de-asserted *Ack* signal also sets the *enP* signal to high which discharges all pull-up nodes in logic stage two. The *enN* signal is set low, which precharges all pull-down nodes in logic stages one and three. Since the neutrality of the internal nodes is not sensed, we introduce a timing assumption on their transition. The discharge of input tokens with a short pulse signal introduces another timing assumption. These two timing assumptions entail the following constraints:

- The pull-down nodes must be fully precharged before *enN* goes high and pull-up nodes must be fully discharged before *enP* transitions low. This

translates into a race condition of 1 pull-up/pull-down transition versus 9 gate transitions, the minimum transition count before both *enN* and *enP* flip when two N-P pipelines are in series.

- All input tokens must be fully discharged within the short pulse discharge period. The pulse has a minimum period of 5 gate transitions. There are as many NMOS discharge transistors as there are input data rails.

The robustness of our pipeline template is not compromised as these timing assumptions satisfy the minimum timing constraint of at least 5 gate transitions between any two relative path delay races.

The discharge of any of the outputs before the validity of all other outputs has been acknowledged can permanently stall the pipeline. To analyze this effect, let us suppose we have three N-P pipelines *A*, *B*, and *C* as shown in Figure 3.3. *A* produces two outputs, one of which goes to *B* and the other one to *C*. *B* uses the output from *A* to compute its output. Since *B* has computed its output, it can now discharge the input it received from *A*. If *A*'s other output, which is headed for *C*, is not yet produced or acknowledged by the completion detection logic of *A*, then *B*'s discharge of its input will make the completion detection logic of *A* unstable. To prevent this, we add the *AckNext* signal to our pipeline template. It is sent to all following pipeline stages that consume the outputs from the current N-P pipeline. This signal is referred to as *AckPrevious* in the destination pipeline as shown in Figure 3.2. It prevents the discharge of the tokens coming from the sender stage before the validity of all outputs in the sender has been acknowledged. As mentioned earlier, *AckPrevious* also signifies the validity of input tokens into the pipeline, hence alleviating the need to check for input token validity in NMOS pull-down stacks. In the case where inputs

come from more than one pipeline block, the *AckPrevious* signals from all corresponding pipeline blocks need to be added to the completion detection logic to ensure against any premature discharge of input data rails.

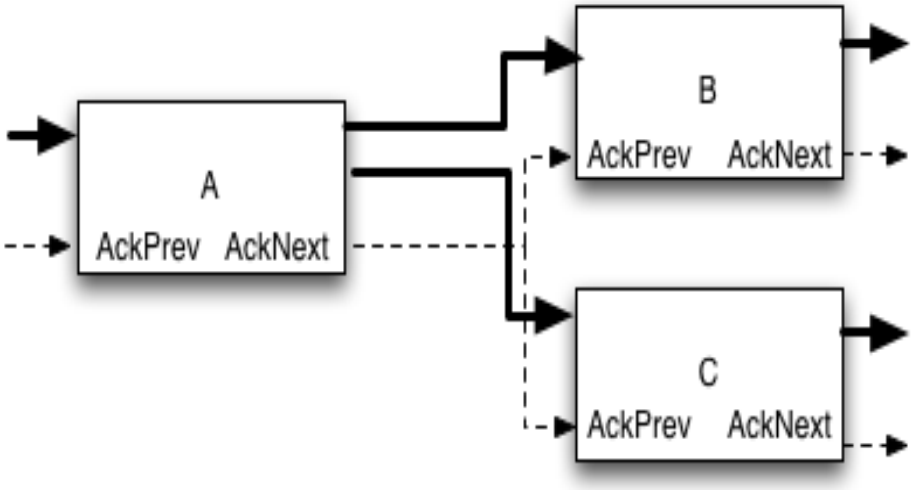

Figure 3.3: Ack signals to ensure correctness

Forking of an output to two successors is also not allowed because then the two successors can reset (discharge) the connection at different times, which could lead to potential conflicts. Hence, we need to create explicit duplicate outputs in the last logic stack for each output that goes to multiple destinations.

To determine the cycle time of the proposed N-P pipeline, let us assume two N-P pipelines in series with time (t) increments taken in terms of logic transitions.

- At $t = 0$, input tokens arrive at the first pipeline block.
- At $t = 4$, first pipeline block produces its output.

- At $t = 7$, *Ack* signal in the first block is de-asserted which signifies the validity of all output signals
- At $t = 8$, second pipeline block produces its output.
- At $t = 9$, input tokens in the first pipeline block are discharged. Internal PMOS logic nodes are discharged.
- At $t = 10$, NMOS logic nodes in the first pipeline are precharged.
- At $t = 13$, output tokens from the first pipeline block are discharged by the second pipeline.
- At $t = 16$, *Ack* signal in the first block is asserted which signifies the reset of all output signals.
- At $t = 18$, *enN* is set and the pipeline is ready to accept new input tokens.

Hence, our proposed N-P pipeline has a cycle time of 18 transitions. Stalls on inputs and outputs do not impact correct operation. The template waits in its present state if inputs arrive at different times. This holds true for outputs being computed at different times as well. The relative path delay assumption has a root, *Ack*, which only changes after all inputs have arrived and all outputs have been computed. As a result, correct operation is not a function of arrival time of signals, which makes the N-P template quite robust.

We could invert the senses of the inputs and outputs by changing the order of the logic stacks within N-P pipeline. With inverted inputs, the first stage comprises PMOS logic stacks and the final logic stage comprises NMOS logic stacks with the outputs produced in inverted sense. This could improve the drive strength of the output signals especially in the case of high fan-out.

Our second proposed pipeline template replaces the PMOS pull-up logic stacks in stage 2 with an inverter, hence the name N-Inverter template, and includes only a single pull-up PMOS transistor in stage 4 as shown in Figure 3.4. As PMOS logic stacks have slower rise times and relatively weak drive strength, the N-P template cycle time may incur a performance hit. The N-Inverter template addresses this by using inverters with faster switching time and strong drive strength . It also results in better noise margins as discussed in detail in Section 3.3. However, these improvements come at the cost of reduced logic density as stage 2 and 4 no longer perform any effective logic computation. Despite these alterations, the N-Inverter and N-P templates use exactly the same timing assumptions. The completion detection and handshake circuitry is also identical.

3.2.2 Completion detection logic for large number of outputs

Since N-P and N-Inverter pipeline templates can pack significant logic in a single pipeline block, there may be cases where a pipeline block has quite a large number of outputs. To detect the validity of these large number of outputs, we may have to expand the c-element validity tree by a couple of extra stages as shown in Figure 3.5.

As a result of these two extra stages in the completion detection validity tree, the cycle time of N-P and N-Inverter templates is no longer 18 transitions. There are four extra transitions, two each for the validity and neutrality detection of the output signals, which increases the cycle time to 22 transitions. Since our goal was to keep the cycle time within 18 transitions, we explored a number of

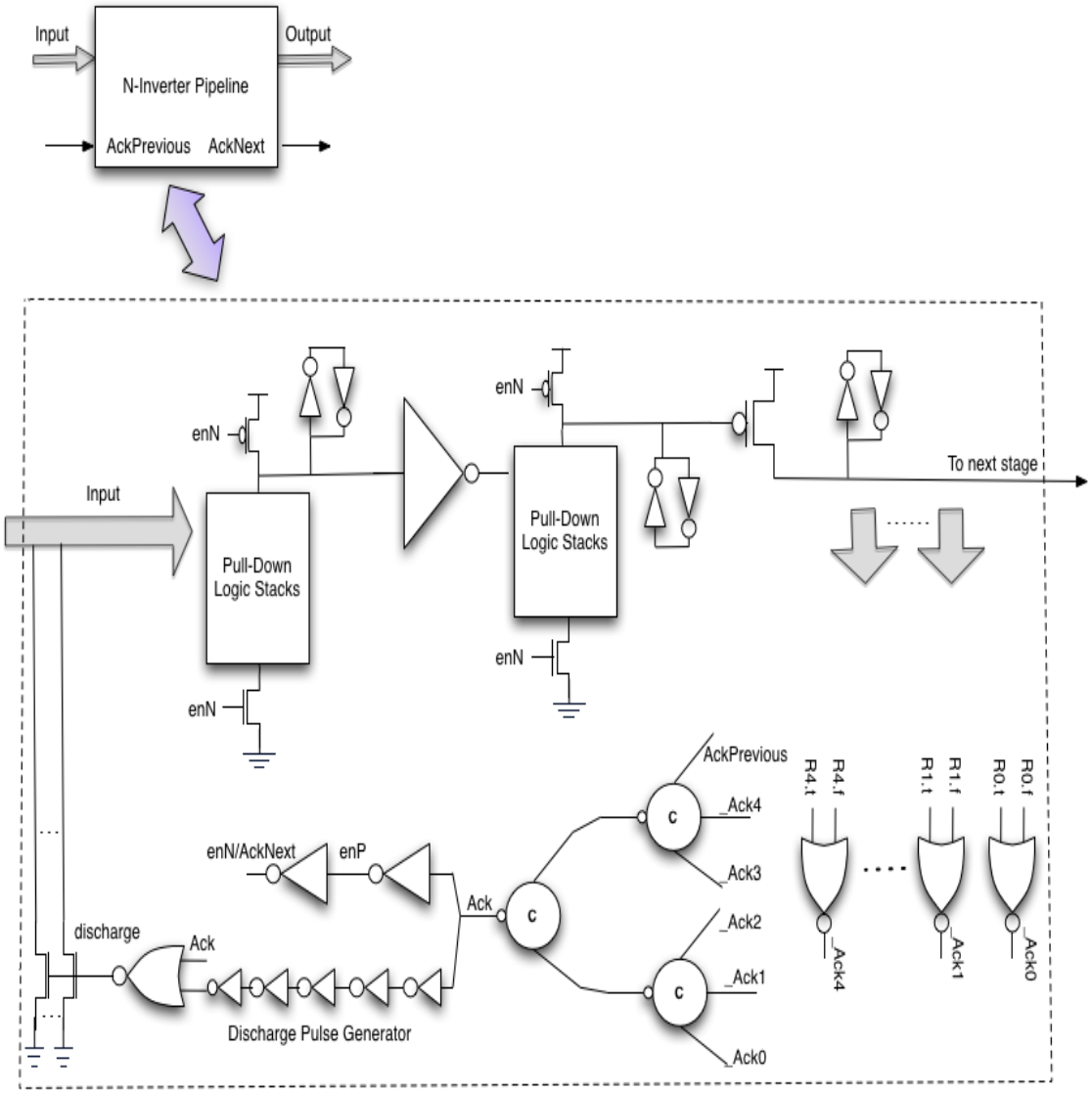

Figure 3.4: N-Inverter pipeline template

other completion detection circuits [56] [14]. To reduce the cycle time back to 18 transitions, we use wide NOR gates based completion detection circuitry as proposed in [14], but with a couple of optimizations to make the circuitry feasible for our proposed pipeline templates. These optimizations include the use of only one output from the set of outputs destined for the same next pipeline

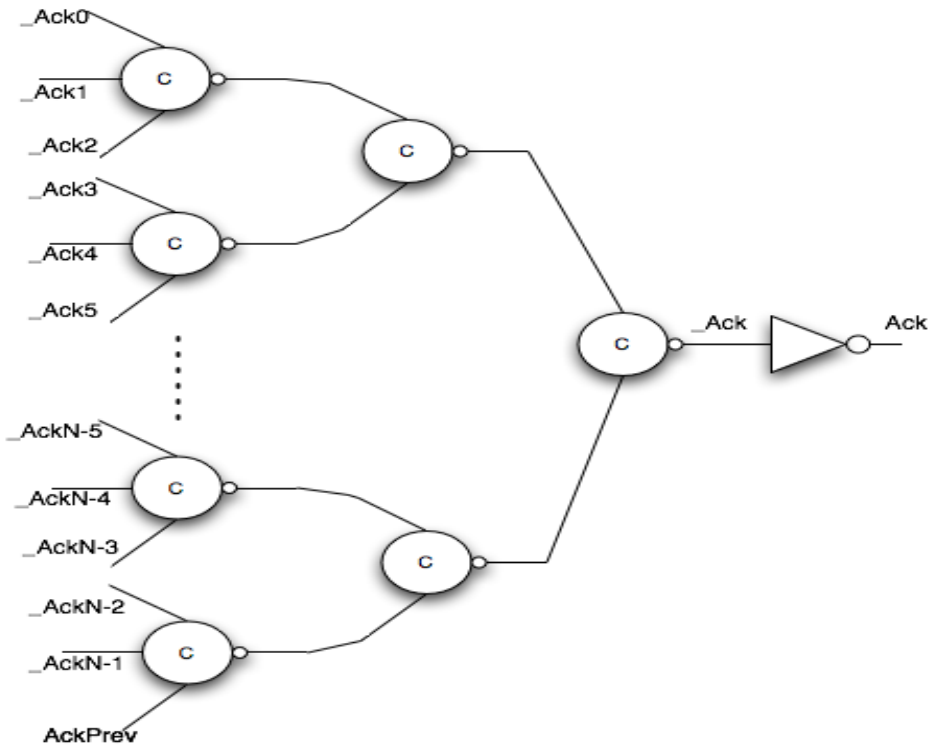

Figure 3.5: Multi-stage c-element tree completion detection logic for large number of outputs

block for neutrality detection and the addition of *enP* and *enN* transistors in the pull-up stacks of *DONE* and *_RST* circuits as seen in Figure 3.6. These optimizations and their benefits are outlined in detail towards the end of this section.

The *_Ack* signals are generated using static NOR gates as previously. The validity of the outputs is signaled by the setting of *Done*. To ensure that the *Done* signal is only set once all *_Acks* have gone low, the pull-up path resistance of the *Done* circuit is set to be at least 4 times as big the pull-down path resistance when only one pull-down transistor is conducting. To prevent a direct path between V_{DD} and *GND*, the *_Ack* from one of the latest (slowest) outputs is used in the pull-up stack.

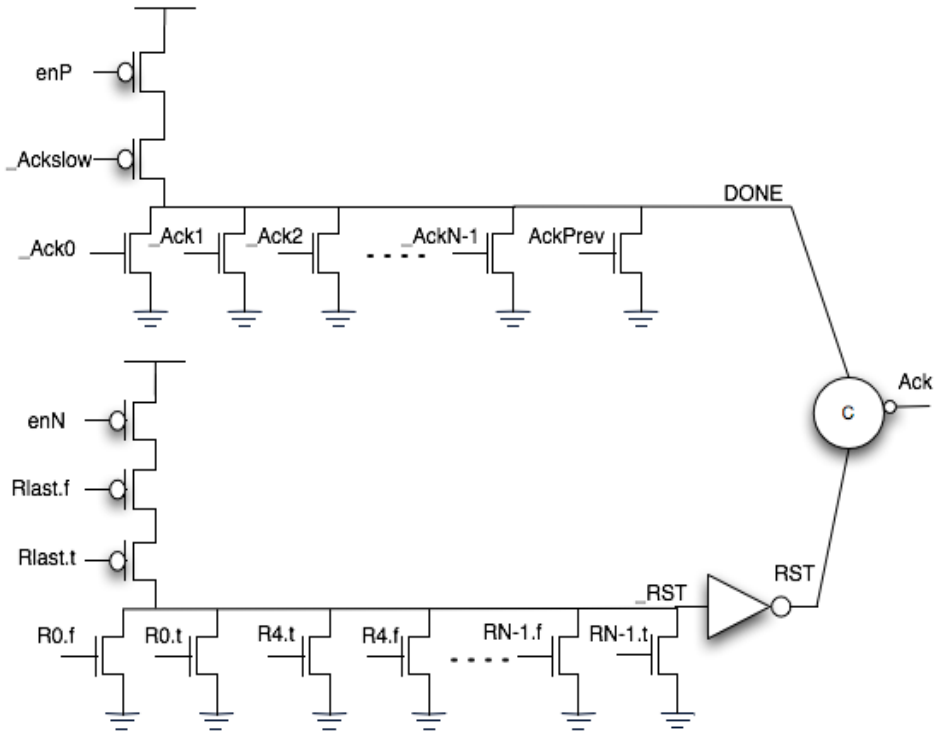

Figure 3.6: Completion detection logic for large number of outputs

The *_RST* signal is used to sense the reset of all outputs. The various *R.t* and *R.f* signals correspond to the actual dual-rail outputs being produced. The latest (slowest) signal to reset is put in the pull-up stack. The pull-up path resistance of the *_RST* circuit is set to ensure that it only goes high once all pull-down transistors in the *_RST* circuit have turned off i.e. all output signals have reset. The *_RST* circuit has two pull-down transistors for each dual-rail output and four pull-down transistors for each 1-of-4 output. As the number of outputs increase, the *_RST* rise time suffers significantly. A close inspection of our proposed pipeline templates made us realize that for outputs destined for the same pipeline block, we only need to check for the reset of one of the outputs and not all because they use the same discharge pulse. Let us assume the dual-rail outputs *R0* to *R3* are all headed for the same pipeline block. We minimize the *_RST*

circuit by only using pull-down transistors corresponding to *R0* output. The transistors corresponding to *R1*, *R2*, and *R3* dual-rail outputs are eliminated as shown in Figure 3.6.

The addition of *enP* and *enN* transistors in the pull-up stacks of *DONE* and *_RST* circuits was another optimization we introduced. The *enP* signal cuts off the pull-up path in the *DONE* circuit while the pipeline is waiting for the outputs to be reset. This prevents the occurrence of a direct path between V_{DD} and *GND* if any of the *_Acks* other than *_Ackslow* goes high first. Similarly, the introduction of *enN* in the pull-up stack of *_RST* cuts off the direct path between V_{DD} and *GND* during the evaluation phase.

3.2.3 Conversion Templates

We envision our proposed N-P and N-Inverter templates to be used for large chunks of local logic wrapped with QDI interfaces. This entails conversion of input tokens from four-phase handshake protocol to single-track protocol and that of output tokens back to four-phase protocol . Figure 3.7 shows an example of a pipeline template used in this protocol conversion. When an input token arrives, it pulls down the pre-charged node, which in turn drives the output. The input and output validity causes *Ack* to go low. This puts the pipeline in pre-charge phase. The inverted *Ack* causes the sender process to reset the input data rails. When the destination pipeline discharges the output rails, the templates sets *Ack* to high, which signals that the template is ready to receive the next token. This template includes no timing assumption.

To convert output tokens from single-track protocol back to four-phase pro-

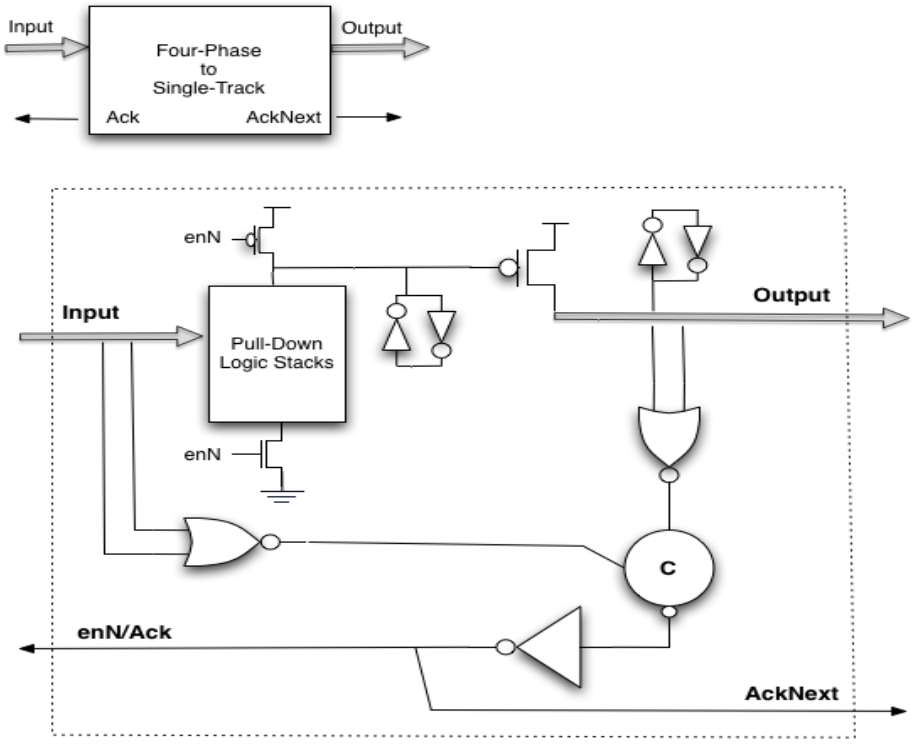

Figure 3.7: Four-phase protocol to single-track protocol conversion template

tocol, we use the template in Figure 3.8. The arriving input token pulls down the pre-charged node, which in turn drives the output. The *AckPrevious* signal is included to prevent premature discharge of input. It goes to low when it is safe to discharge. The receiver process de-asserts the *AckIn* signal once it no longer needs the data. At this point, the discharge pulse goes high. It has a minimum period of 5 gate transitions and requires that input token be fully discharged within this period. As in the case of our proposed pipeline templates, this timing assumption meets our minimum delay race timing constraint.

Although, these templates do not add significant hardware complexity, they nevertheless incur some area and energy overhead. In a design, with large

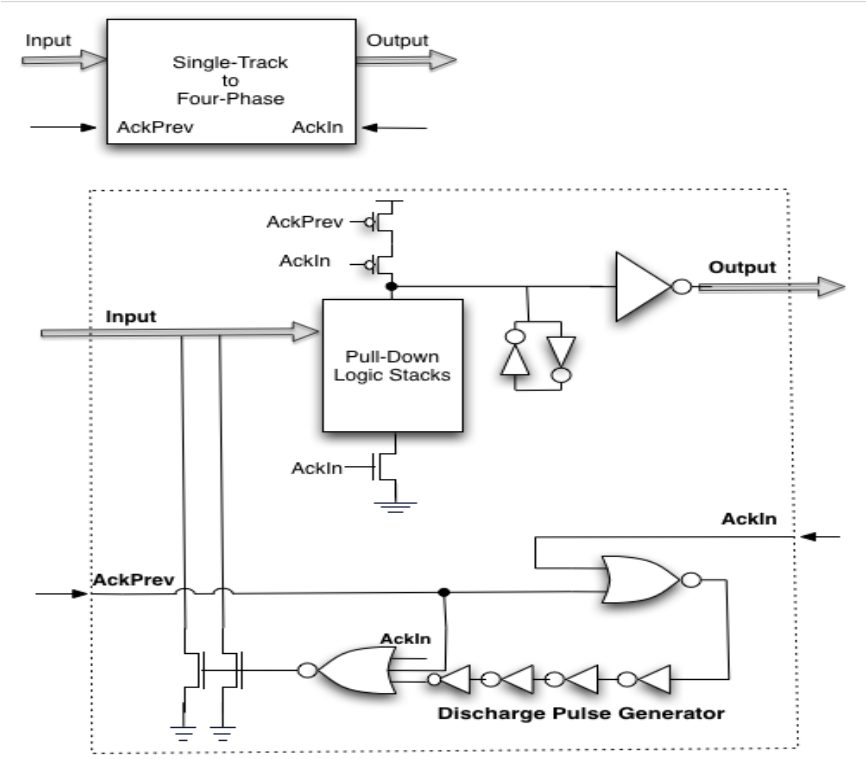

Figure 3.8: Single-track protocol to four-phase protocol conversion template

chunks of local computation, comprising multiple N-P or N-Inverter pipeline stages, the conversion templates are used only once at each end of the design boundary. This helps amortize their cost over the entire datapath. The overhead could be further mitigated by merging useful computation in the logic stacks.

3.3 Design Considerations and Trade-offs

3.3.1 Completion Detection Circuits

To quantify the trade-offs between the two completion detection schemes, we carried out detailed SPICE level simulations with estimated wire loads for each node. It is assumed that each output goes to a separate pipeline block, hence the discharge of each signal is checked. The wide NOR completion detection circuitry results in lower latency relative to multi-stage c-element tree detection completion scheme across a wide range of outputs as shown in Figure 3.9. The latency difference increases as the number of outputs increases since c-element completion may require multiple extra stages. For 15 output signals, the wide NOR completion results in 30% less latency.

In terms of energy consumption, the choice of a completion detection scheme depends not only on the number of outputs but also on the arrival order and the delay of the chosen *latest* signal as shown in Figure 3.10. The x-axis corresponds to the arrival order of the chosen *latest* signal. For example, the data point corresponding to the arrival order of 9 means that our chosen *latest* output was the ninth output to be set or reset. All of the remaining signals arrived after an arbitrary 2-FO4 delay. This corresponds to a period of direct path between V_{DD} and *GND* for wide NOR based completion detection scheme. The c-element based completion scheme consumes the same energy irrespective of the arrival order. It is also more energy-efficient when the number of outputs is 9 or less. However, with a greater number of outputs as may be required for some N-P and N-Inverter pipeline templates, the wide NOR based completion detection scheme consumes significantly less energy. Another noteworthy observation

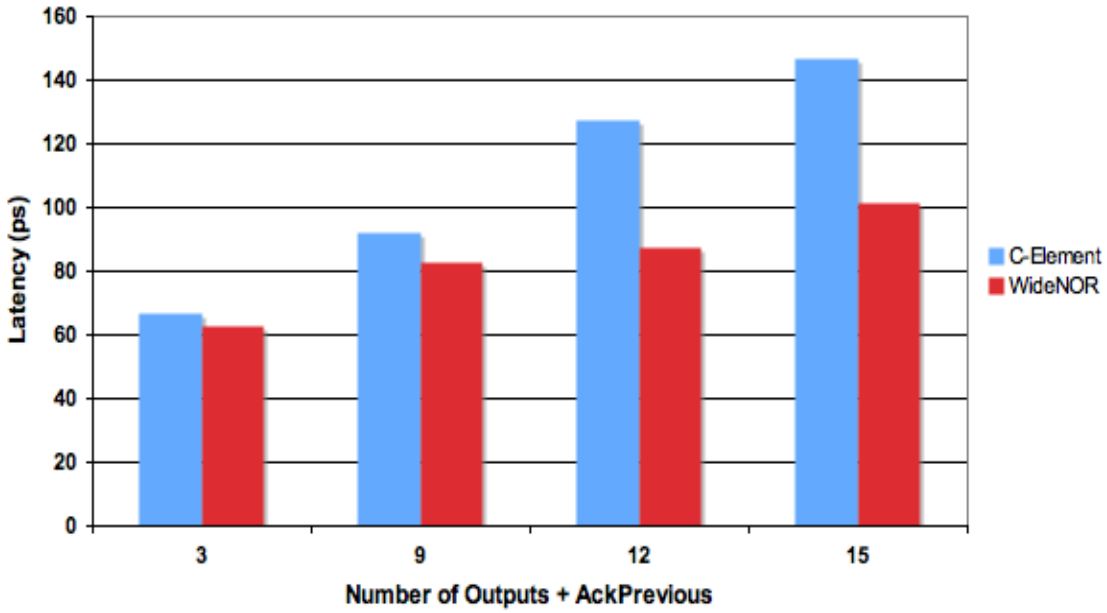

Figure 3.9: Latency comparison of completion detection schemes

from Figure 3.10 is that the effect of arrival order on energy consumption is only profound when the *latest* signal is one of the last few signals.

The longevity of the period of direct path between V_{DD} and GND, when the chosen *latest* signal is the not the last one, may lead to significant energy consumption for wide NOR based completion detection scheme. To explore this effect, we simulated wide NOR completion circuits for 12 and 15 outputs by varying the delay of late arriving signals as seen in Figure 3.11 and Figure 3.12. For 12 outputs, unless any output arrives 3 or more FO4 delays after the chosen *latest* signal, the wide NOR completion consumes less energy compared to the c-element based completion scheme, irrespective of the arrival order of the *latest* signal. For 15 outputs, the margin increases to 5 or more FO4 gate delays for the wide NOR completion to consume more energy than the corresponding

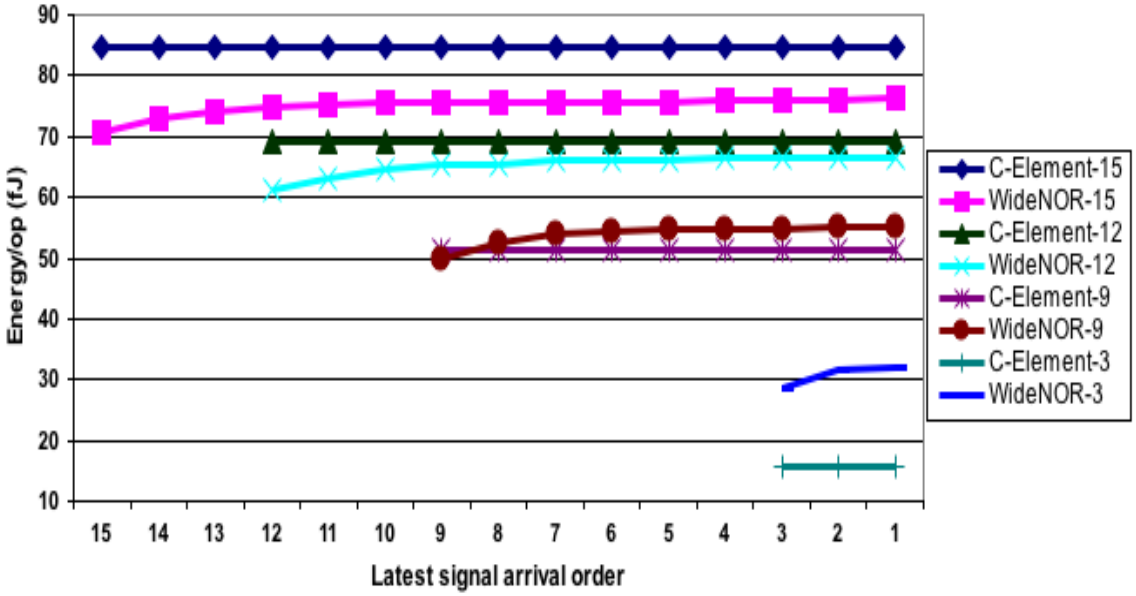

Figure 3.10: Completion detection energy consumption for different arrival order of chosen signal

c-element based completion detection scheme. These results indicate that the energy consumption of the wide NOR completion scheme is largely a function of delay between the chosen *latest* signal and the actual last signal. In the case of small delay variability between outputs produced within the same pipeline block, for example most arithmetic operations, the wide NOR scheme consumes less energy per operation than its c-element tree counterpart. In the unusual scenario of large delay difference between outputs within the same pipeline, the wide NOR scheme still functions correctly, albeit at higher energy consumption.

In terms of transistor area, the wide NOR completion detection circuit becomes more efficient as the number of outputs increase as seen in Figure 3.13. The choice of a particular completion detection circuit is therefore a design

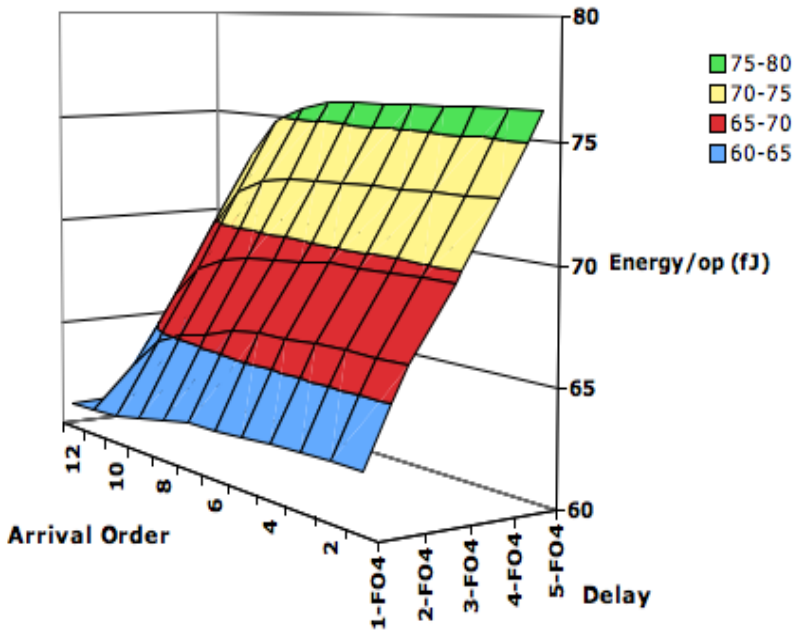

Figure 3.11: WideNOR for 12-outputs with varying delay of latest signal

choice, which may depend on a number of factors: the number of outputs for a pipeline stage, latency and throughput targets, power budget, area constraints, and the delay variability of chosen *latest* output.

3.3.2 Throughput, Energy, and Area Trade-offs

Throughput, energy, and area are critical design considerations for a circuit designer. We choose an 8-to-1 multiplexor design, which produces multiple copies of the output as shown in Figure 3.14, to highlight some of these trade-offs in our proposed templates. PCeHB, N-P, and N-Inverter pipelined versions of the chosen circuit were implemented. Highest precision SPICE simulations were conducted in 65nm bulk CMOS process with estimated wire loads for each node.

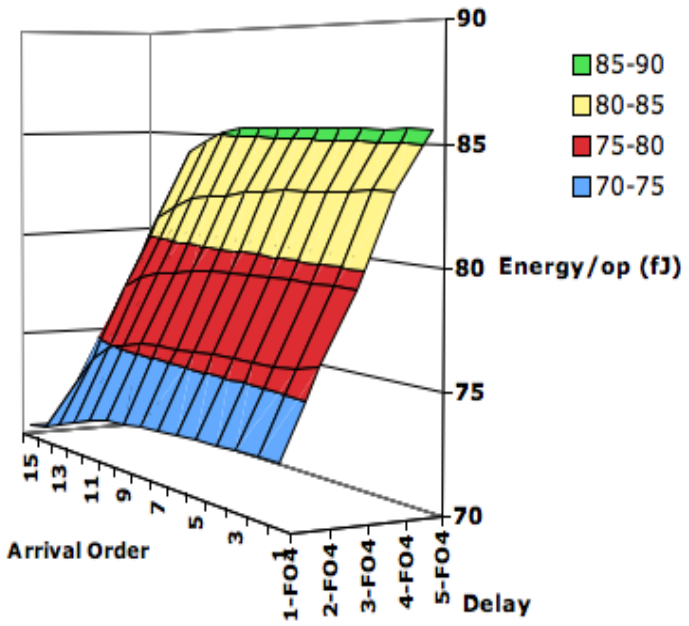

Figure 3.12: WideNOR for 15-outputs with varying delay of latest signal

Although, all three templates have a cycle time of 18 transitions, the N-P implementation results in an 8.5% lower throughput. The N-P implementation is slower because it employs some logic computations in PMOS stacks, which have slower slew rates and weaker drive strength than NMOS stacks. In a PCeHB implementation, each 2-to-1 multiplexor represents a separate pipeline stage with each stage incurring a significant handshake overhead as seen earlier in Figure 2.7. There is a separate pipeline block for copy logic as well. Whereas, in N-P and N-Inverter implementations, the full 8-to-1 multiplexor circuit including copy logic can be packed completely in one and two pipeline blocks respectively. The effect of this logic compaction on energy efficiency and total transistor width is quite profound. Our N-Inverter implementation, operating at the same throughput as a PCeHB design, consumed 52.6% less energy per

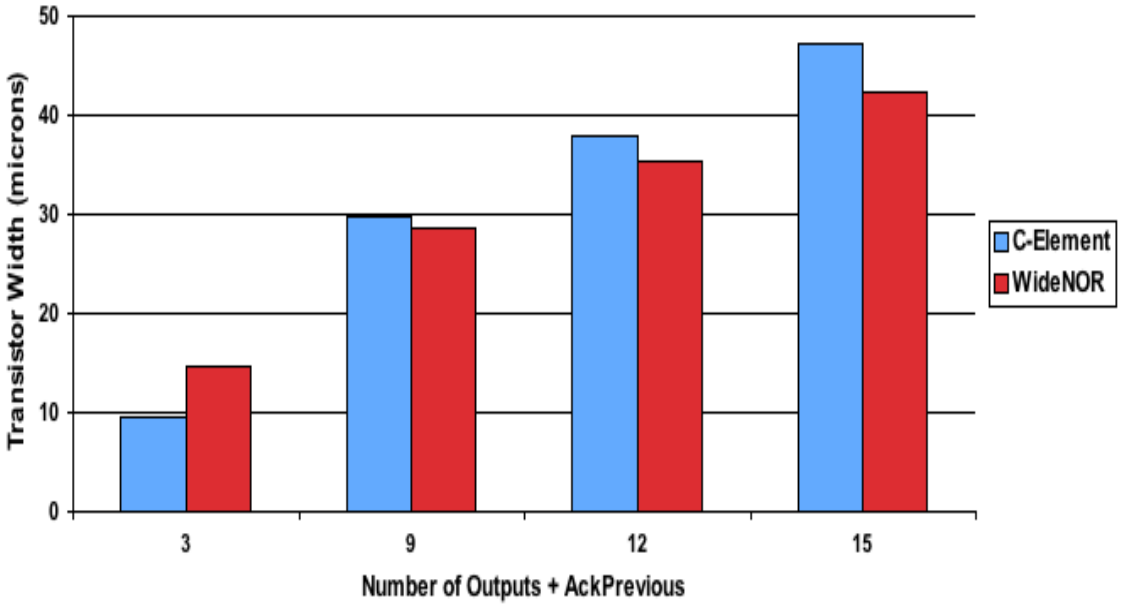

Figure 3.13: C-Element vs WideNOR: total transistor width comparison

operation while using 48% less transistor width. With N-P pipeline, the energy and transistor width savings shoot up to 71.2% and 65% respectively, albeit at an 8.5% throughput penalty.

The proposed N-P and N-Inverter templates enable us to pack more logic computations within a single pipeline stage while maintaining a very high throughput. This flexibility is not available in standard PCeHB designs, which are composed of pipeline stages with only one effective logic computation in a single stage. More logic per a single stage in our proposed templates creates a likelihood of a large number of outputs per pipeline, which may adversely affect overall throughput as shown in Figure 3.16. The dependency of absolute throughput on the number of outputs highlights an important design trade-off. With more outputs, although the number of transitions remain the same with

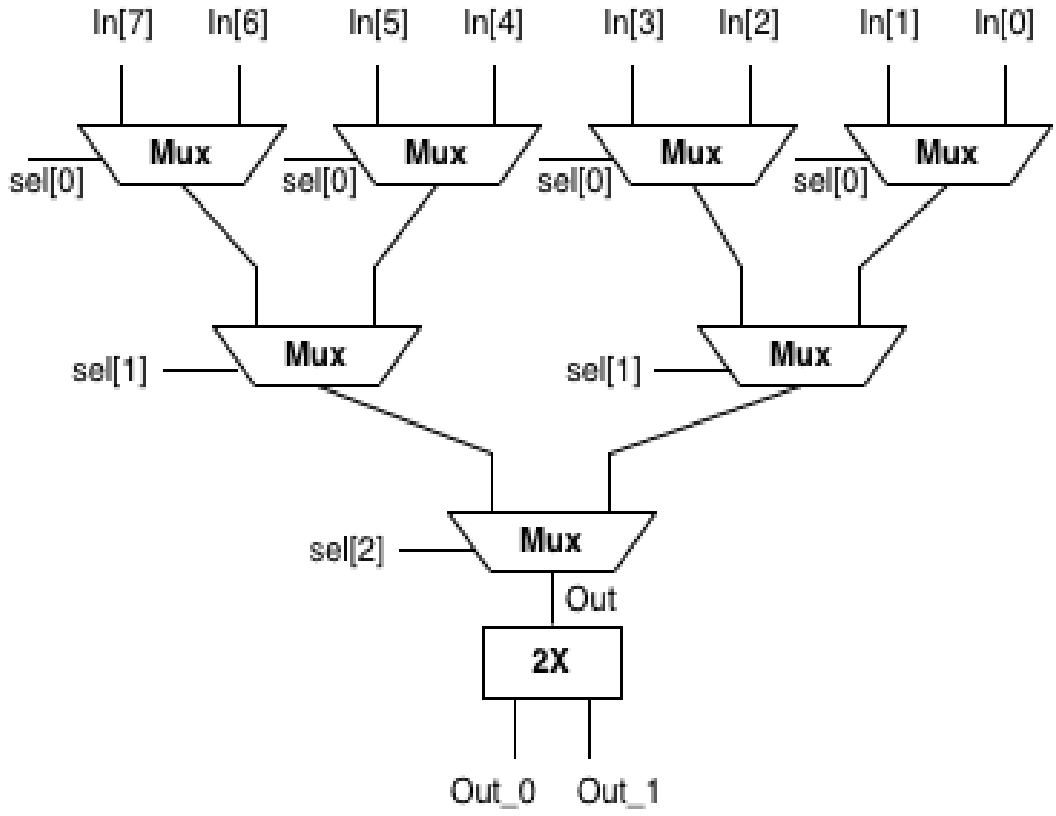

Figure 3.14: 8-to-1 multiplexor with 2 copies of Output

the use of a wide NOR completion detection logic, each of these transitions incur a higher latency as shown earlier in Figure 3.9. The results would be even worse if a c-element based completion logic was used as it would incur 4 extra transitions per each cycle.

3.3.3 Noise Analysis

Noise feedthrough is one of the major concerns when it comes to the use of dynamic gates. Since our proposed pipeline templates use cascaded dynamic gates

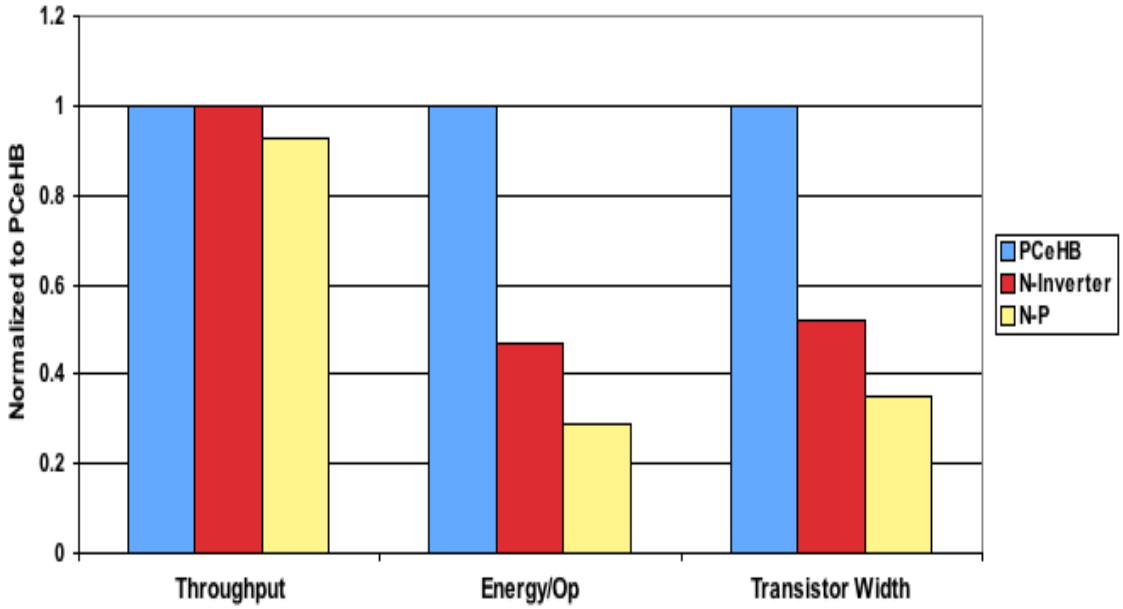

Figure 3.15: 8-to-1 multiplexor design trade-offs for different pipeline styles

for logic computations, we carried out comprehensive noise margin analysis of our circuits. Dynamic gates from each pipeline template were simulated across all process corners, typical-typical (TT), slow-fast (SF), fast-slow (FS), slow-slow (SS), and fast-fast (FF), in a 65nm bulk CMOS technology with highest-precision SPICE configuration at 1V nominal V_{DD} and 85°C operating temperature. Since SPICE simulations do not account for wire capacitances, we included additional wire load in the SPICE file for every gate in the circuit. For each pipeline template, the lowest value of noise margin amongst all process corners was chosen.

For noise feedthrough analysis of N-P template, we analyzed a full-adder NMOS logic stack followed by a two-input AND gate in a PMOS pull-up stack. The noise margin, as defined in [71], of this cascaded N-P configuration is the

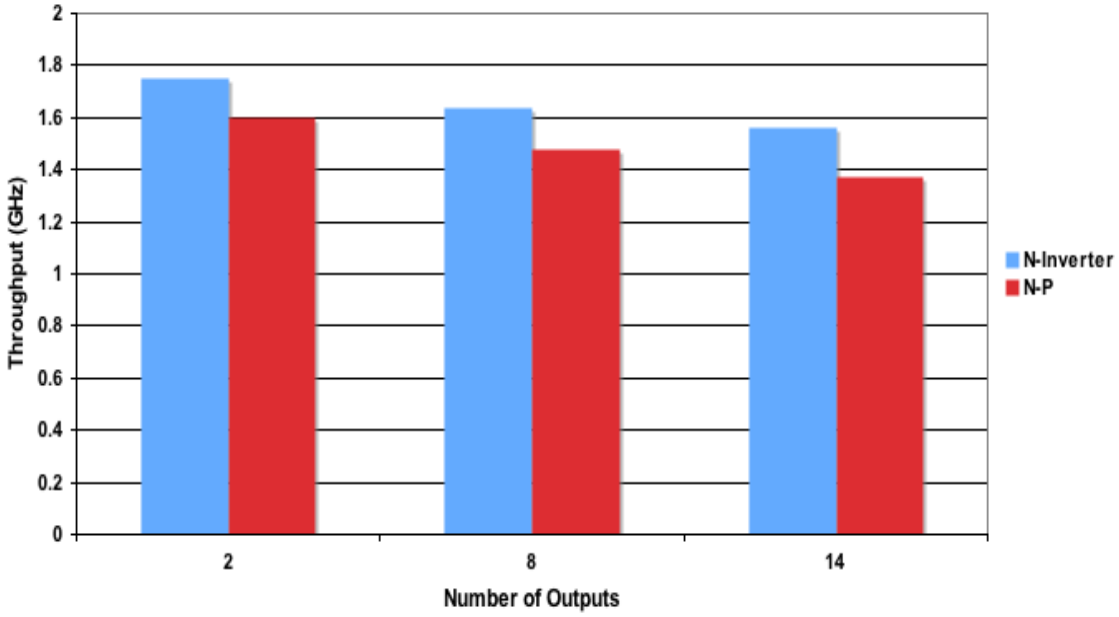

Figure 3.16: Throughput dependency on the number of outputs

difference in magnitude between the minimum low input voltage recognized by NMOS logic stack on one of the inputs at unity gain point and maximum low output voltage of the driving PMOS pull-up stack. For N-Inverter and PCeHB templates, we analyzed a full-adder NMOS logic stack followed by a static CMOS inverter, with noise margin defined as the difference in magnitude between the minimum low input voltage recognized by NMOS logic stack on one of the inputs and maximum low output voltage of the driving output inverter. The results are shown in Figure 3.17 which also shows the noise margin of a two-input static CMOS NOR gate for comparison.

The N-P template has the lowest noise immunity. However, the noise margin can be significantly improved by increasing the relative drive strength of the staticizers to dynamic logic stacks. But this improvement comes at the cost of

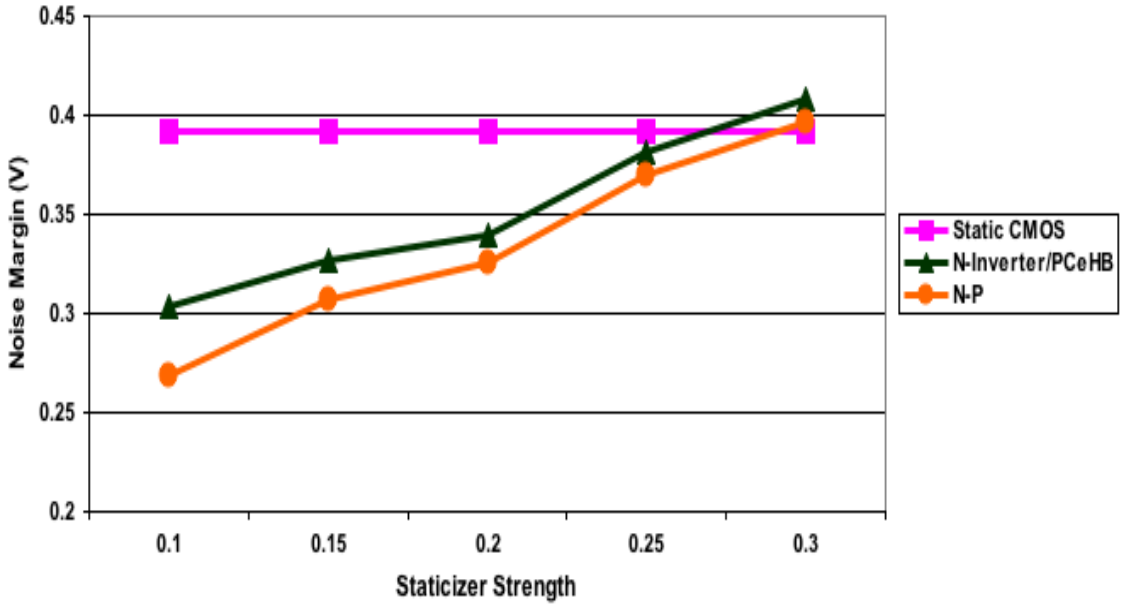

Figure 3.17: Noise margin analysis

throughput degradation and a slight increase in energy per operation as shown in Figure 3.18 and Figure 3.19 respectively. The energy per operation results are normalized to a PCeHB implementation energy per operation at a staticizer strength of 0.1. As seen from these results, the choice of an exact strength value for a staticizer represents a design trade-off, which should be made on the basis of final throughput target, desired robustness, as well as circuit application.

3.3.4 Timing Margin

The N-P and N-Inverter pipeline templates include multiple timing assumptions, the breach of which could impact correct operation or stall the pipeline. In Section 3.2, we discussed the timing margins necessary to ensure correct-

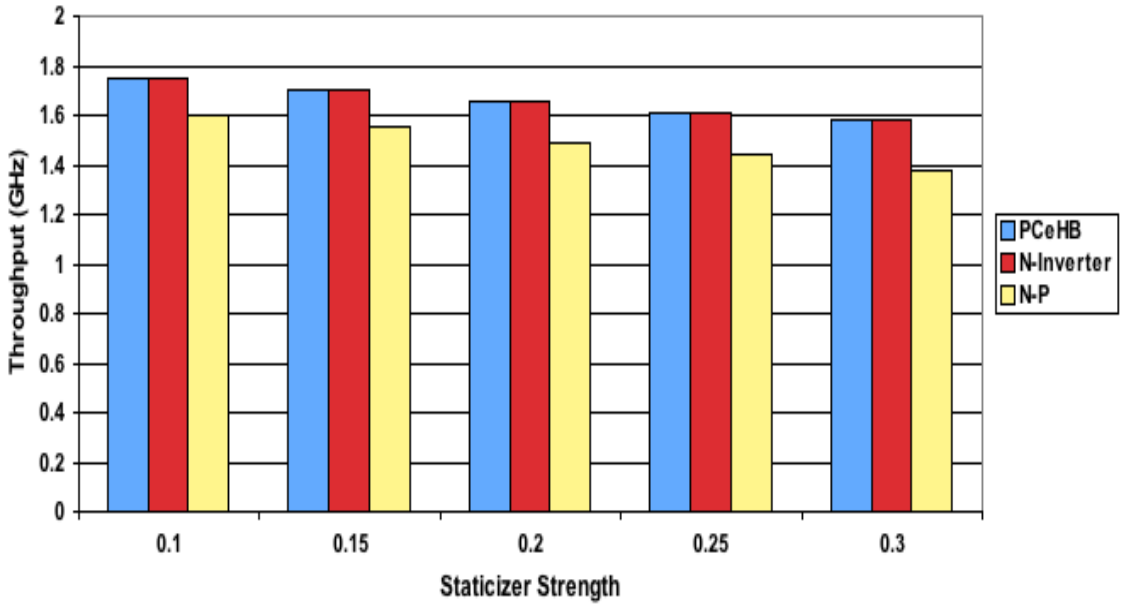

Figure 3.18: Effect of staticizer strength on pipeline throughput

ness, but these timings margins were given in terms of gate transitions. To ensure sufficient robustness of our templates, we analyzed the exact timing constraints of full transistor-level implementations of our proposed pipelines in a 65nm bulk CMOS technology with highest-precision SPICE configuration at 1V nominal V_{DD}, 85°C operating temperature, and estimated wire loads for each gate. The timing constraint of 9 gate transitions for precharge and discharge of internal nodes translated into 14.8 FO4 and 12.2 FO4 delays for N-P and N-Inverter pipelines respectively, whereas the worst case transition corresponding to precharge or discharge of an internal node took no longer than 2.67 FO4 delays. This yields a very safe timing margin of over 12 FO4 and 9.5 FO4 delays for N-P and N-Inverter pipelines respectively.

The second timing assumption in the N-P and N-Inverter pipelines pertains

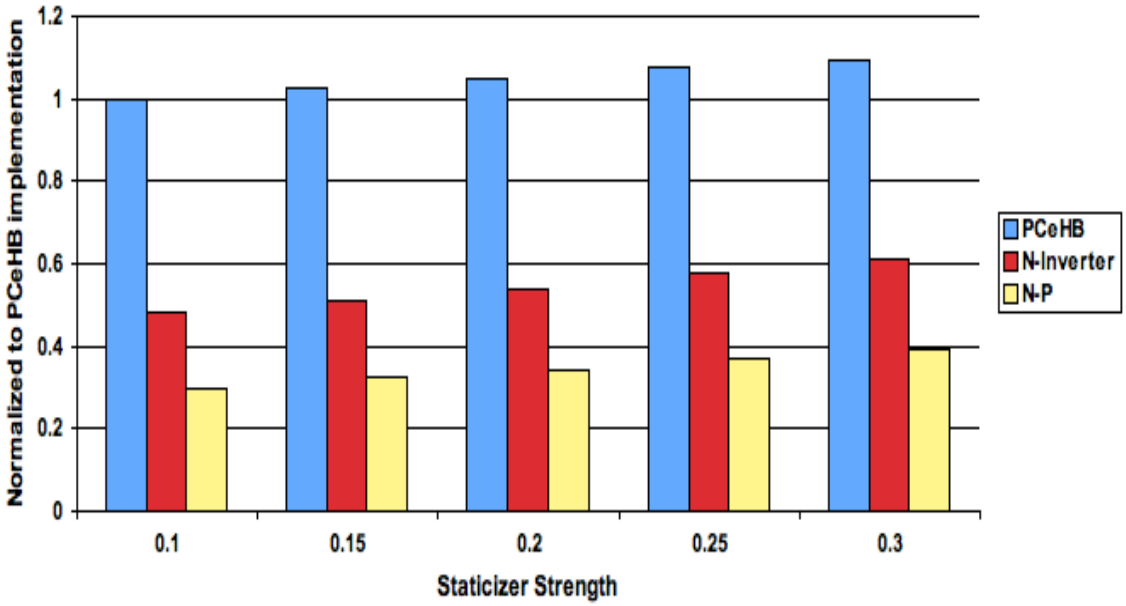

Figure 3.19: Effect of staticizer strength on energy/op

to the full discharge of all input tokens within the short pulse discharge period. The 5 transition discharge pulse translates into 5 FO4 delays for both N-P and N-Inverter templates. The discharge pulse timing margin is a function of input load, which in turn is a function of input gate and wire capacitances. Since we envision our proposed templates to be used for large chunks of local computation and not for global communication, we found the short pulse period sufficient for full input token discharge including the added wire capacitance for each node, which corresponds to 12.5 μm wire length. In the worst case, an input token took no longer than 2.5 FO4 delays to fully discharge, which yields a timing margin of 2.5 FO4s. Since the discharge pulse period is not on pipeline critical path for both forward latency and throughput, the timing margin could be improved by adding two extra inverters to the pulse generator inverter chain

without affecting performance. With these two extra inverters, the timing safety cushion increases from 2.5 FO4 to 4.5 FO4 delay, which makes the templates significantly more robust.

3.4 8x8 Booth-Encoded Array Multiplier

This section highlights the effectiveness of our proposed templates in an array multiplier design. High performance multiplier circuits are an essential part of modern microprocessors [55] [69] and digital signal processors [68]. The FPM datapath's complexity is a function of the array multiplier it uses. To achieve high throughput and low latency, most high performance chips use some form of booth encoded array multiplication hardware [10]. The array multiplier architecture requires a large number of tokens to be in flight at the same time. Each multiplication operation produces a number of partial products which are then added together to produce the final product. We implemented an 8x8-bit radix-4 booth-encoded array multiplier (at the transistor level) using PCeHB pipelines to act as our baseline. To quantify the energy efficiency and other characteristics of our proposed low-handshake pipeline templates, we implemented two full transistor level 8x8-bit radix-4 booth-encoded array multipliers using N-P and N-Inverter pipeline templates respectively. Since improving energy-efficiency was the primary goal, we pack considerable logic within each pipeline stage, even at the cost of incurring throughput degradation of up to 25% compared to PCeHB style pipelines.

Figure 3.20 shows the top-level specification of our 8x8-bit multiplier. The top part of Figure 3.20 shows the partial product generation for the array mul-

tiplier. Each of the Y inputs is in a radix-4 format. The multiplicand bits are used to generate the booth control signals for each partial product row. Since a PCeHB pipeline can only compute a small amount of logic, each of the rectangular boxes labeled *PP* represents a separate pipeline stage. The booth control signals and multiplier input bits are sent from one pipeline stage to another, while each pipeline stage produces a two bit partial product.

The second half of Figure 3.20 shows the order in which the partial products are produced and summed up. The horizontal dotted lines separate different time periods. Each of the dotted polygons represent a separate PCeHB pipeline stage. The entries inside each polygon represent the inputs which are added together to produce the sum and carry outputs for the next pipeline stage. *PP* stands for two-bit partial product entry, C' corresponds to sign bit for each partial product row, *SS* stands for two-bit sum output from a previous stage, and C stands for a single-bit carry output from a previous stage sum computation. The final product bits are generated in a bit-skewed fashion, indicated by the symbol *RR*. Hence, we need to add slack-matching buffers on the outputs as well as some of the inputs to optimize the multiplier throughput [16]. For simplicity, we do not show these slack-matching buffers in Figure 3.20.

The multiplier is highly pipelined but contains very little logic in each pipeline stage. Section 2.4 includes an example of a PCeHB template similar to the ones used in the 8x8-bit multiplier design. The highlighted logic gates in Figure 2.6 are not used for the actual computation but are only required for the handshake protocol. Since each PCeHB stage contains only a small amount of logic, it leads to significant handshake overhead, in terms of power consumption and transistor count, as tokens may have to be copied for use in separate

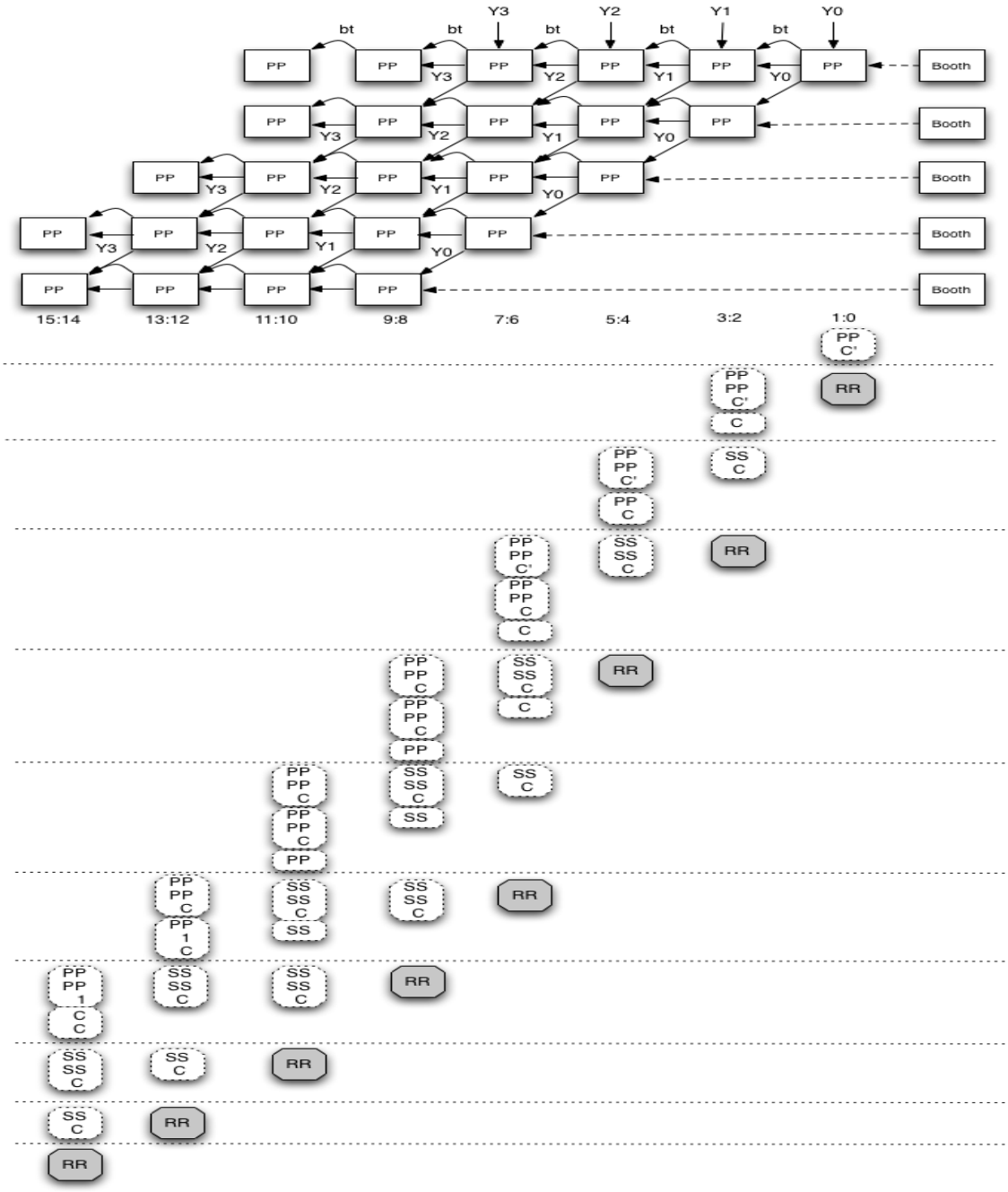

Figure 3.20: 8x8-bit multiplier architecture using PCeHB pipelines

processes with each process doing its own validity and neutrality checks.

Figure 3.21 shows an overview of N-P pipelines and their logic stacks for the 8x8-bit array multiplier. Both N-P pipelines have four stages of logic. The first stage of the first pipeline generates all partial product entries. This is clearly a big power saving, as booth control signals and multiplier inputs need to be generated only once and not for each separate pipeline block as in the PCeHB implementation. Each dotted polygon represents a logic stack and not a separate pipeline stage, which leads to very high logic density in each pipeline block. Each *RR*, *SS*, and *C* signal represents a single output channel, which translates into 14 outputs for the first N-P pipeline block and 4 outputs for the second N-P block. The N-Inverter pipeline implementation requires twice as many pipeline stages as N-P implementation since no effective logic computation is performed in its PMOS pull-up stacks. However, the rest of the design is similar to N-P pipeline implementation with considerable logic within each pipeline stage.

In contrast to the large number of fine-grain pipeline blocks in the PCeHB implementation shown in Figure 3.20, we only need two N-P and four N-Inverter pipeline stages to implement the bulk of 8x8-bit multiplication logic. The inputs to the first pipeline for both N-P and N-Inverter implementations are four radix-4 multiplier bit entries and booth control signals for all rows, which are generated separately using PCeHB style pipelines. We use conversion templates, shown in Section 3.2.3, to convert these tokens from four-phase protocol to single-track protocol. For pipeline blocks with more than nine outputs, we use wide NOR completion detection scheme. For outputs destined for the same pipeline block, we only track the neutrality of one of the outputs going to the second pipeline. This optimization greatly reduces the complexity of *_RST* circuitry, reduces power consumption, and increases the throughput by up to 6.3% for our proposed pipeline templates. To highlight the seamless integration of N-

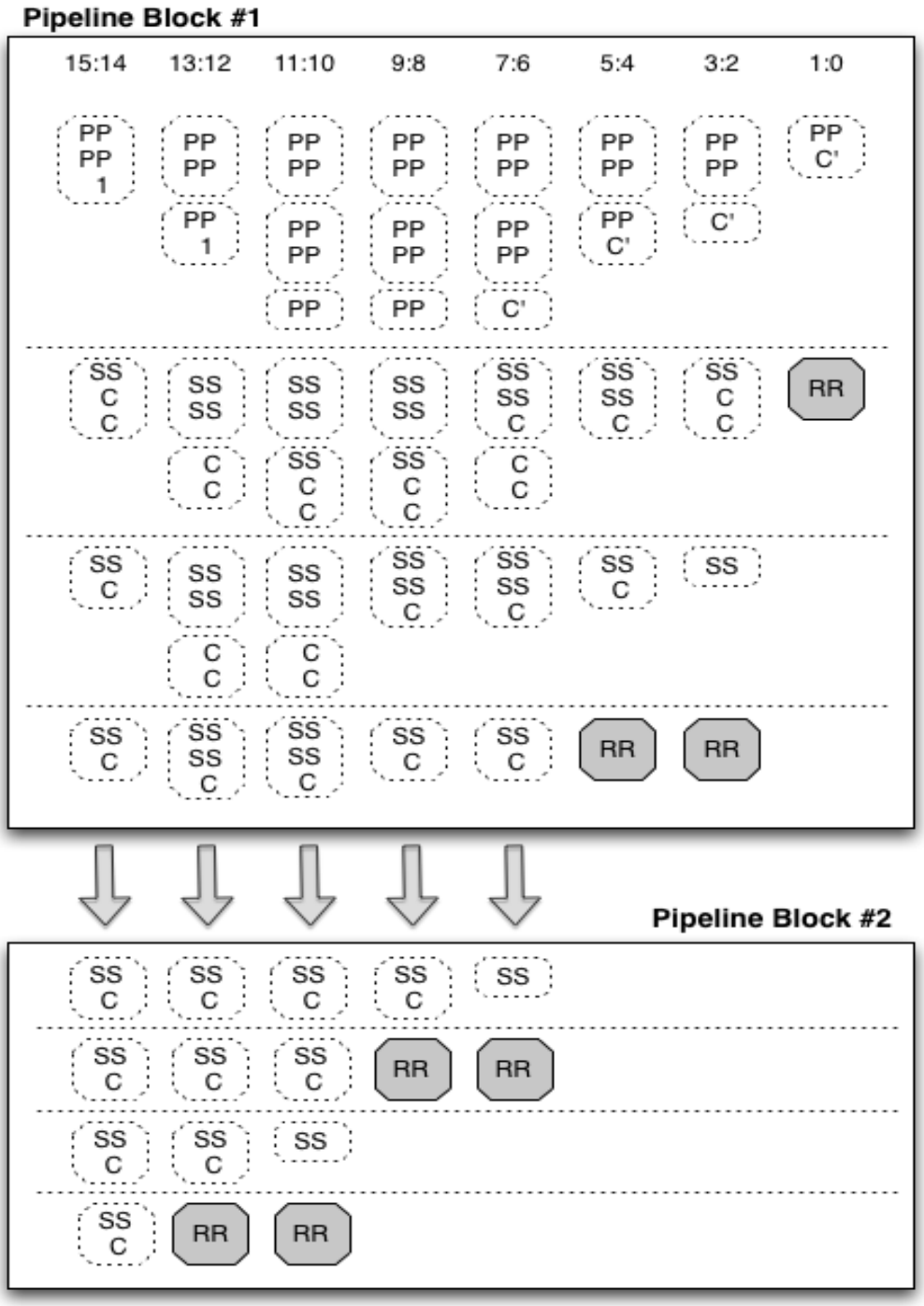

Figure 3.21: 8x8-bit multiplier using N-P pipelines

P and N-Inverter pipelines within any four phase handshake environment, we convert the resultant product outputs into four phase *1-of-4* encoding.

3.4.1 Evaluation of 8x8-bit Multiplier Designs

The transistors in our baseline PCeHB multiplier implementation and our proposed N-P and N-Inverter pipeline implementations were sized using standard transistor sizing techniques [71]. The slow and power-consuming state-holding completion-elements were restricted to a maximum of three inputs at a time. Keepers and weak feedback inverters were added for each state-holding gate to ensure that charge would not drift even if the pipeline were stalled in an arbitrary state.

Since HSIM/HSPICE simulations do not account for wire capacitances, we included additional wire load in the SPICE file for every gate in the circuit. Based on prior experience with fabricated chips and post-layout simulation, we have found that our wire load estimates are conservative, and predicted energy and delay numbers are typically 10% higher than those from post-layout simulations. Our simulations use a 65nm bulk CMOS process at the typical-typical (TT) corner. Test vectors are injected into the SPICE simulation using a combined VCS/HSIM simulation, with Verilog models that implement the asynchronous handshake in the test environment. All simulations were carried out at the highest-precision setting.

Figure 3.22 shows the power-consumption breakdown of our proposed pipeline templates. In contrast to the PCeHB pipelines, which consume over 69% power in handshake overheads, the handshake and completion detection

logic accounts for only 26% of the total power in our proposed pipelines. The elimination of validity and neutrality detection logic for a large number of intermediate nodes in each pipeline is the main reason for the reduction of handshake related overheads.

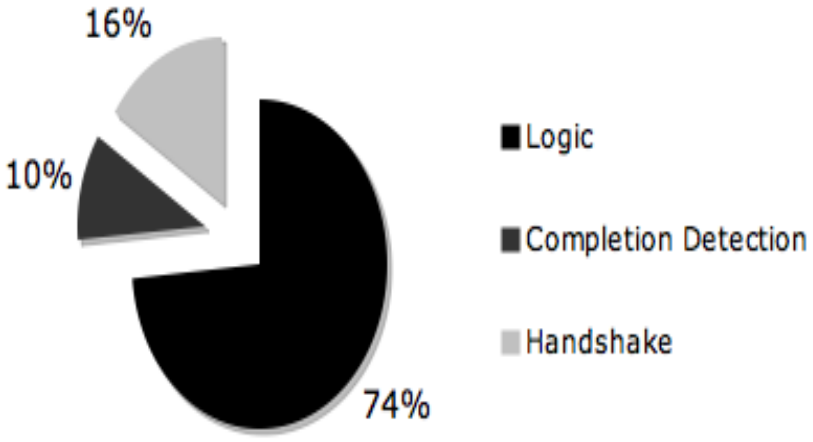

Figure 3.22: Power consumption breakdown of N-P and N-Inverter pipelines

To fully quantify and evaluate our proposed pipeline templates, we simulated all three 8x8-bit array multiplier implementations across a wide range of voltages. All experimental results presented in this section include the explicit overhead of conversion templates. These templates convert input tokens from four phase protocol to single-track protocol and the outputs from single-track protocol back to four-phase protocol.

The throughput and energy consumption results for all three pipeline implementations with data points corresponding to 0.6V to 1.1V at 0.1V intervals

plotted from left to right in Figure 3.23. As stated earlier, the N-Inverter and N-P implementations were designed from energy efficiency perspective while allowing throughput degradation of up to 25% compared to PCeHB design. To ensure fair comparison, the N-P implementation used a higher staticizer strength to yield similar noise margin as PCeHB and N-Inverter implementations. To minimize handshake circuitry, each N-Inverter and N-P pipeline block was packed with considerable logic computations and produced a large number of outputs, which reduced overall throughput. Hence, in terms of throughput, the PCeHB pipeline implementation yields the best results across all voltages. But this performance improvement comes at the cost of 45.4% and 59.5% higher energy per operation compared to the N-Inverter and N-P pipeline implementations respectively. Another key observation from Figure 3.23 is that for any single throughput target, be it in low throughput range such as 400-500 MHz or in high throughput range such as 1.3-1.5 GHz, our proposed templates consume far less energy per operation than the PCeHB implementation.

The fact that our proposed pipelines worked across a vast voltage range without requiring any transistor re-sizing highlights the robustness of our proposed templates. The experimental results include the power consumed in templates that are required to convert the inputs from four phase protocol to single-track protocol and the outputs from single-track protocol to four phase protocol.

The energy savings are largely due to:

- The massive reduction in the handshake circuitry because of the elimination of validity and neutrality detection gates for all internal nodes.
- The sharing of inputs and intermediate outputs within a same pipeline block. In a PCeHB implementation, the inputs and outputs are copied

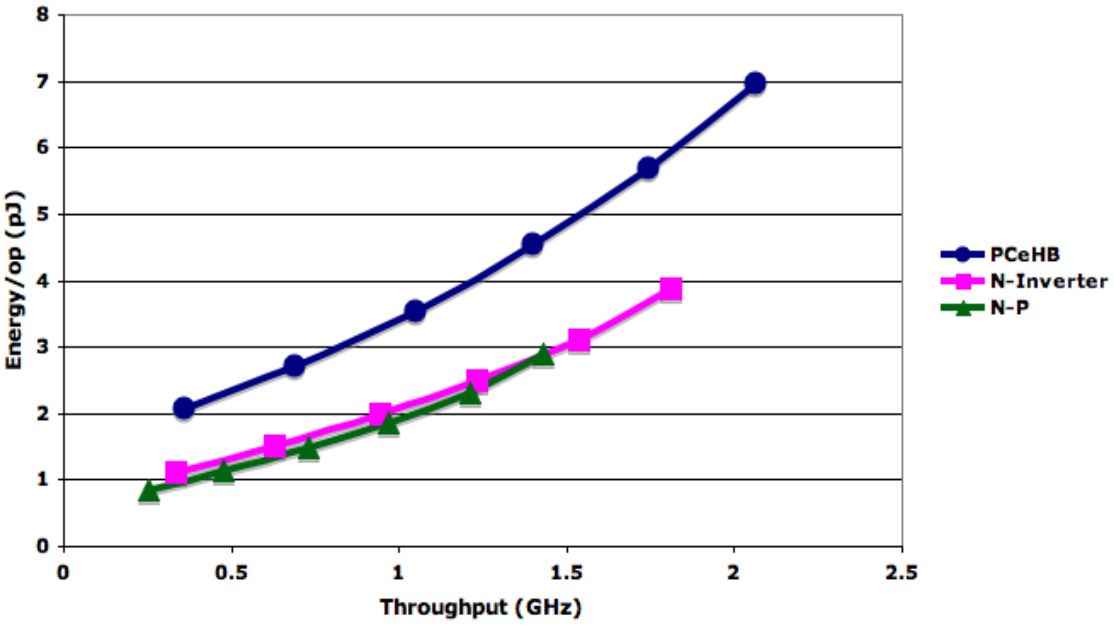

Figure 3.23: 8x8-bit multiplier throughput vs energy for three different pipeline styles

from one stage to another and are subjected to separate validity and neutrality detection checks within each pipeline block.

- The use of a more energy-efficient completion detection scheme.

To consider performance and energy together, we use two metrics: energy-delay product and energy-delay^2 product as shown in Figure 3.24. The results are normalized to the PCeHB implementation. The N-Inverter and N-P pipelines reduce the energy-delay product by 38.5% and 44% respectively. For energy-delay^2 product, N-Inverter implementation yields a 30.3% reduction and N-P pipelines result in 22.2% reduction when compared to the PCeHB implementation.

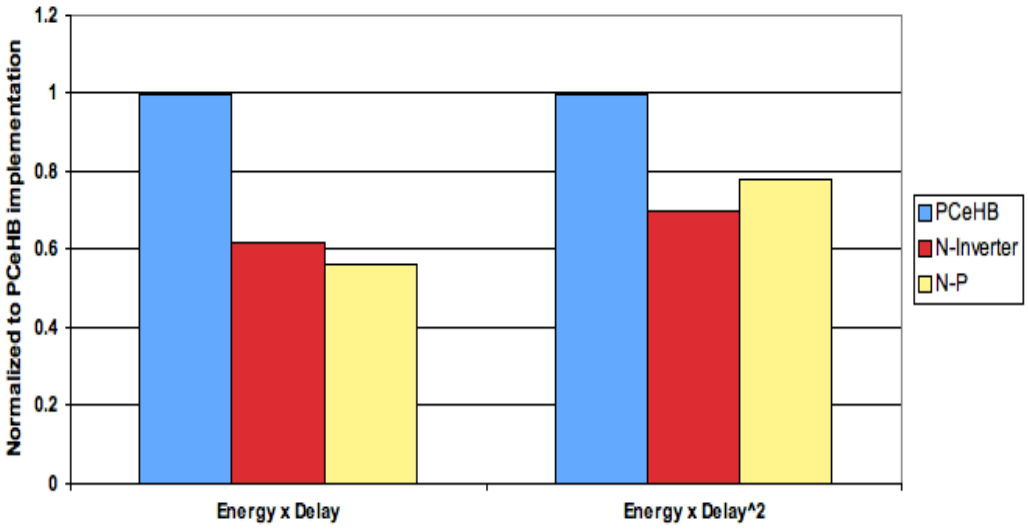

Figure 3.24: 8x8-bit Multiplier energy-delay analysis for three different pipeline styles

The N-Inverter and N-P implementations reduce the overall multiplier latency by 20.2% and 18.7% respecitvely as shown in Table 4.4. These two pipeline templates can pack significant amount of logic within a single pipeline block, which reduces the total number of pipeline stages required and hence results in latency reduction. Although, N-Inverter implementation requires twice as many pipeline stages as N-P implementation, it results in a 1.85% lower overall latency. This could be attributed to the use slower pull-up logic stacks in N-P templates.

In terms of the total transistor count, the N-Inverter and N-P implementations use 42.2% and 54.2% less transistors respectively than the PCeHB implementation as shown in Table 3.2. The total transistor width in N-Inverter and N-P designs is 35.6% and 46% less respectively than that in the PCeHB implementation. This huge saving in the transistor count and width can be directly

Table 3.1: 8x8-bit Array Multiplier Latency

Pipeline Style	**Latency**
PCeHB	663 ps
N-Inverter	529 ps
N-P	539 ps

attributed to the packing of more logic stacks within a single pipeline block and the elimination of handshake logic for all intermediate nodes.

Table 3.2: 8x8-bit Array Multiplier Transistor Count

Pipeline Style	**No. of Transistors**	**Width (**μm)
PCeHB	17083	5290
N-Inverter	9864	3402
N-P	7819	2853

The choice of a particular pipeline implementation represents a design trade-off. Critical factors such as target throughput, logic complexity, power budget, latency range, total transistor count, noise margins, and timing robustness will have to be taken into account simultaneously before choosing a particular pipeline implementation. The N-P and N-Inverter templates represent a good energy efficient alternative to QDI templates, especially for logic computations which require a large number of inputs or outputs or those with multiple intermediate logic stages. We envision these circuits being used for large chunks of local logic (e.g. an array multiplier in a floating point unit) wrapped with QDI interfaces, rather than globally.

3.5 Summary

We propose two energy-efficient pipeline templates for high throughput asynchronous circuits. These two templates, named N-P and N-Inverter pipelines, use single-track handshake protocol. Each pipeline contains multiple stages of logic. The handshake overhead is greatly minimized by eliminating validity and neutrality detection logic gates for all input tokens as well as for all intermediate logic nodes. Both of these templates can pack significant amount of logic within each pipeline block, while still maintaining a fast cycle time of only 18 transitions. Stalls on inputs and outputs do not impact correct operation. A comprehensive noise analysis of dynamic gates within our proposed templates shows sufficient noise margins across all process corners. Since our templates introduce multiple timing assumptions, we also analyzed the timing robustness of our pipelines. A completion detection scheme based on wide NOR gates is presented, which results in significant latency and energy savings especially as the number of outputs increase.

Three separate full transistor-level pipeline implementations of an 8x8-bit booth-encoded array multiplier are presented. Compared to the PCeHB implementation, the N-Inverter and N-P pipeline implementations reduced the energy-delay product by 38.5% and 44% respectively. The overall multiplier latency was reduced by 20.2% and 18.7%, while the total transistor width was reduced by 35.6% and 46% with N-Inverter and N-P pipeline templates respectively.

Chapter 4

An Operand-Optimized Floating-Point Adder

In this chapter, we present the design and implementation of an asynchronous high-performance IEEE 754 compliant double-precision floating-point adder (FPA). We begin with a baseline asynchronous FPA that corresponds to a state-of-the-art high performance synchronous FPA design. We provide energy-consumption breakdown of a high-performance asynchronous FPA datapath, and use this to guide our optimizations for energy-efficiency. We present our operand-dependent optimization techniques to reduce the energy per operation of asynchronous floating-point addition, including some that result in poor throughput in pathological cases. It is these optimizations that are challenging in the synchronous context, because they increase the worst-case critical path making the common case slower even though on average they have negligible impact on throughput. To our knowledge, this is the first detailed design of a

high-performance asynchronous double-precision floating-point adder.

4.1 A Baseline Asynchronous FPA

Our baseline unit is the first fully-implemented (at the transistor-level) asynchronous double-precision floating-point adder of its kind. It supports all four rounding modes and is fully IEEE-754 compliant. Fig. 4.1 shows the block diagram of our FPA datapath, which is loosely based on recent high-performance FPA/FMAs. It uses standard state-of-the-art techniques such as leading one prediction and decoding, use of parallel prefix tree adder, and fast logarithmic shifters to keep an overall low latency and high throughput. To reduce latency and overall complexity, the post-addition normalization datapath is separated in two paths. The *Left* path contains a variable left-shifter, whereas the *Right* path includes a single-position right or left shifter along with all rounding and increment logic. We equally weighed performance and power trade-offs in the choice of our circuits for various functional blocks of the FPA. The following subsections explain our choice of asynchronous pipelines, 56-bit significand adder and LOP/LOD functional block.

4.1.1 Fine-grain Asynchronous Pipelining

We use quasi-delay-insensitive (QDI) asynchronous circuits in our FPA design. Each QDI pipeline contains only a small amount of logic (e.g. a two-bit full-adder). This fine-grain control over each logic computation enables us to introduce very low level operand dependent optimizations.

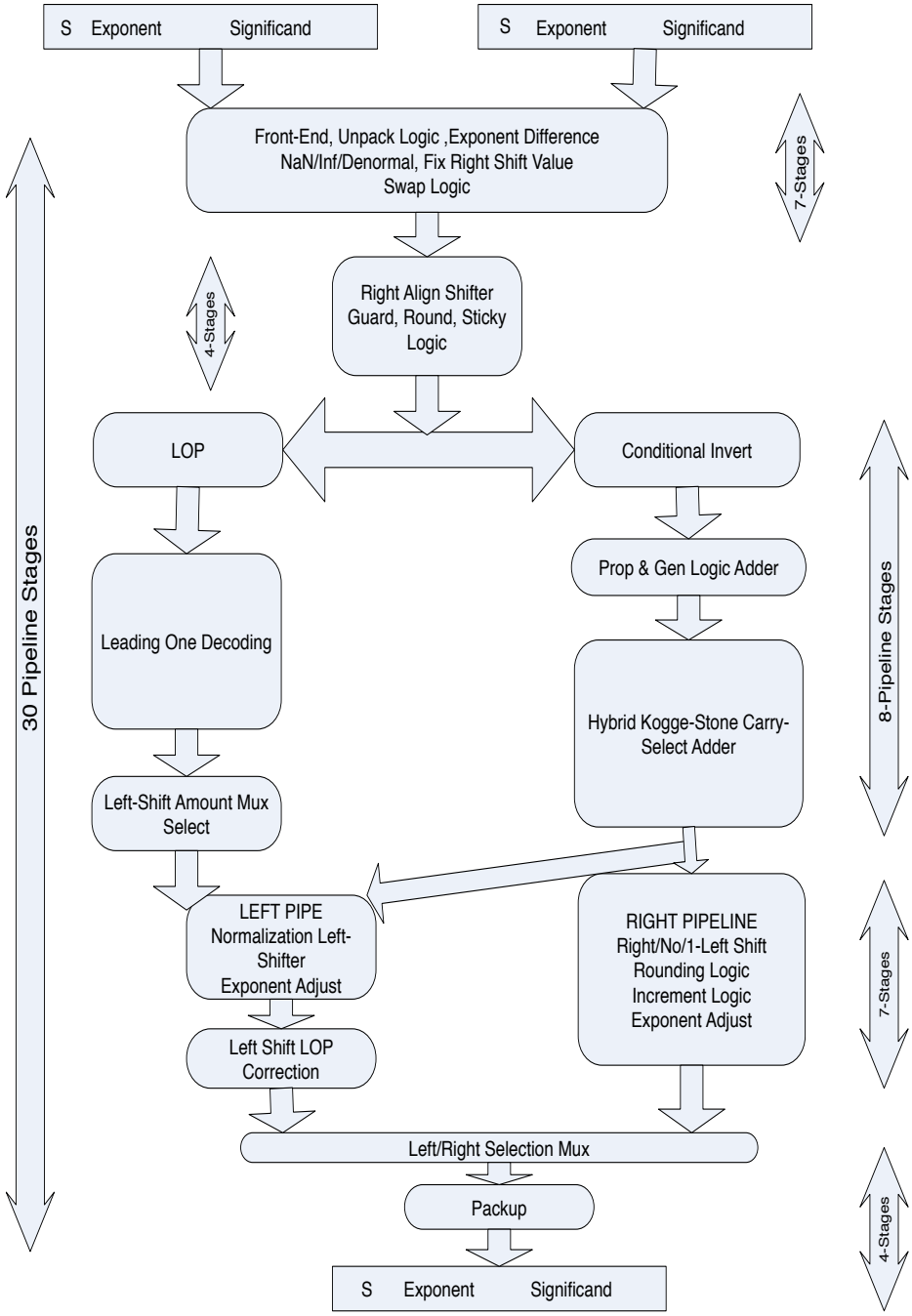

Figure 4.1: Asynchronous Baseline FPA Architecture

Our baseline asynchronous FPA's datapath is highly pipelined (thirty pipeline stages) to maximize throughput. Unlike the standard synchronous pipelines, the forward latency of each asynchronous pipeline is only two logic transitions (the pull-down stack followed by the inverter), hence the thirty stage asynchronous pipeline depth results in acceptable FPA latency. The fine-grain asynchronous pipelines in our design contain only a small amount of logic (e.g. a two-bit full-adder).

We use pre-charge enable half-buffer (PCeHB) pipeline, explained in Section 2.4, for all data computation [22]. For simple buffers and copy tokens, we use a weak-conditioned half-buffer (WCHB) [36] pipeline stage, which is much smaller circuit than a PCeHB and hence is more energy-efficient for simple data buffering and copy operations.

4.1.2 Hybrid Kogge-Stone Carry-Select Adder

The 56-bit significand adder is on the critical path of the FPA and is the single largest functional block in the FPA datapath. Improvements in the adder design usually have the largest overall impact on the FPA, hence designers spend considerable time in optimizing their adder circuits for performance and power. Parallel prefix logic networks that use tree structures to compute the carry are usually preferred for any adder with a large number of input bits. Tree adders like Kogge-Stone [32], Brent-Kung [71], and Sklansky [71] can compute any N-bit sum with a worst-case latency of $O(\log N)$ stages. Many commercial chips use some form of these tree adders in their FPA implementations.

Our baseline asynchronous FPA uses a hybrid *Kogge-Stone carry-select* adder

topology. A Kogge-Stone adder is a parallel prefix form carry look-ahead adder [32, 71]. It generates carry outputs in logarithmic time. We use eight-bit Kogge-Stone blocks that compute two speculative sum outputs (assuming the carry-in is either zero or one). There are a total of seven such blocks to support the full 56-bit addition. In parallel, we use a parallel prefix carry computation logic to compute the actual carry inputs for each of the seven Kogge-Stone adder blocks. The final stage selects the correct eight-bit sum output from each block using actual carry values, hence the name *carry-select*.

The choice of eight-bit Kogge-Stone sub-blocks was made for energy-efficiency as blocks with more bits would have resulted in higher energy due to long wiring tracks that have to run across the total width of the block [51]. Most blocks in the adder use radix-4 arithmetic and 1-of-4 codes (like the adder in [42]) to minimize energy and latency. The rationale behind the choice of a hybrid *Kogge-Stone carry-select* adder is that most high-performance floating-point adders use a similar topology.

Subtraction is done in the usual way by inverting the inputs and using a carry-in of one for the entire adder. The choice of significand to invert is important from the energy perspective. Since IEEE floating-point uses a sign-magnitude representation, a final negative result requires a second two's complement step. To avoid this, our asynchronous FPA always chooses to invert the smaller of the two significands.

4.1.3 Leading One Prediction and Decoding

Most modern FPA implementations use LOP/LOD logic to determine the shift amount for normalization in parallel with the significand adder. This reduces the latency of the FPA, because the shift amount is ready when the adder outputs are available.

Our LOP logic is inspired from the LOP scheme proposed by Bruguera et al. [12]. It subtracts the two significands using a signed digit representation producing either a 0, 1, or -1 for each bit location. There is no carry propagation in signed digit subtraction, which alleviates the need to use an expensive adder topology. The output for each bit position is encoded using an 1-of-3 encoding, which sets a separate data rail for each case. The bit string of 0s, 1s, and -1s can be used to find the location of the leading one [12], except that it could be off by one in some cases. Instead of using a correction scheme that operates in parallel with the LOP hardware (requiring significant more energy), we use the speculative shift amount and then optionally shift the final outcome by one in case there was an error in the estimated shift amount. This also requires an adjustment to the exponent. To make this adjustment efficient, both values of the exponent are computed concurrently by using a dual-carry chain topology for the exponent adder.

4.1.4 Evaluation of Baseline Asynchronous FPA

We use a 65nm bulk CMOS process at the typical-typical (TT) corner. The steady state throughput and energy per operation results for our baseline asynchronous FPA with highest-precision HSIM/HSPICE simulation configuration

are shown in Fig. 4.2. The different data points correspond to different supply voltages (0.6V and 1.1V). We added additional wire load in the SPICE file for every gate in the circuit.

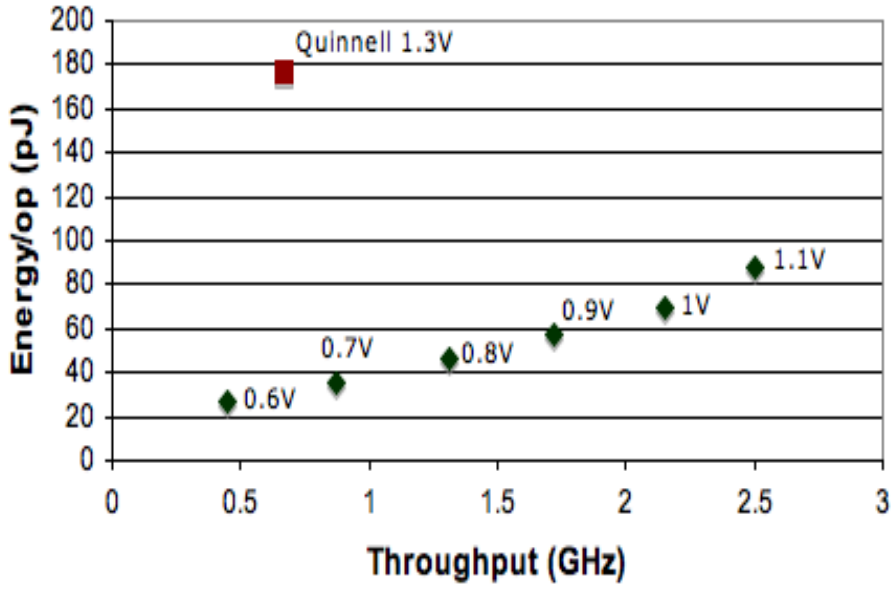

Figure 4.2: Baseline FPA Energy vs Throughput

At a V_{DD} of 1V, the FPA operates at a throughput of 2.15 GHz with an average power dissipation of $149mW$, an energy/operation of 69.3 pJ/op. The power values include the gate and sub-threshold leakage power. Compared to the standard-cell library FPA in a 65nm SOI process by Quinnell et al. [54] operating at a throughput of 666 MHz with an average power-consumption of $118mW$, our baseline FPA design operating at 3.2 times higher throughput consumes 2.6 times less energy per operation even though we are using a bulk process.

4.1.5 Power Breakdown and Analysis

The last decade witnessed a significant change in the focus of arithmetic circuit designers from purely performance oriented high-speed circuits to energy-efficient circuit implementations. To improve the efficiency of any VLSI system, it is critical to first understand where energy and power are dissipated. We have not found a detailed energy/power breakdown of a state-of-the-art FPA datapath in the open literature.

Fig. 4.3 shows a detailed energy/power breakdown of our FPA datapath. Starting with 11% of Front-End and proceeding in the clock-wise direction, the energy/power contributions are in the same order as listed in the legend in the figure. Since in asynchronous PCeHB and WCHB pipelines the actual computation is folded and coupled into the pipelines, the percentage power usage of any particular functional block includes all pipeline overhead i.e. input validity, output validity and handshake acknowledge computation. Although, the Hybrid Kogge-Stone carry-select Adder is the largest power-consuming functional block in the pipeline, it is interesting to note that there is no single dominant high-power component in the FPA datapath. Hence, any effective power-saving optimizations would require us to tackle more than one function block.

The *Right-Align Shift* block which comes second in terms of power-consumption includes logic to compute the guard, round, and sticky bits to be used in the rounding mode. In the worst case, the sticky bit logic has to look at all 53 shifted out bits. To do this fast and in parallel with the right-align shifter, considerable extra circuitry is needed which consumes more power. The post addition *Right Pipeline* block is the third most power-consuming component of the FPA datapath. It includes the single position left or right shifter as well as

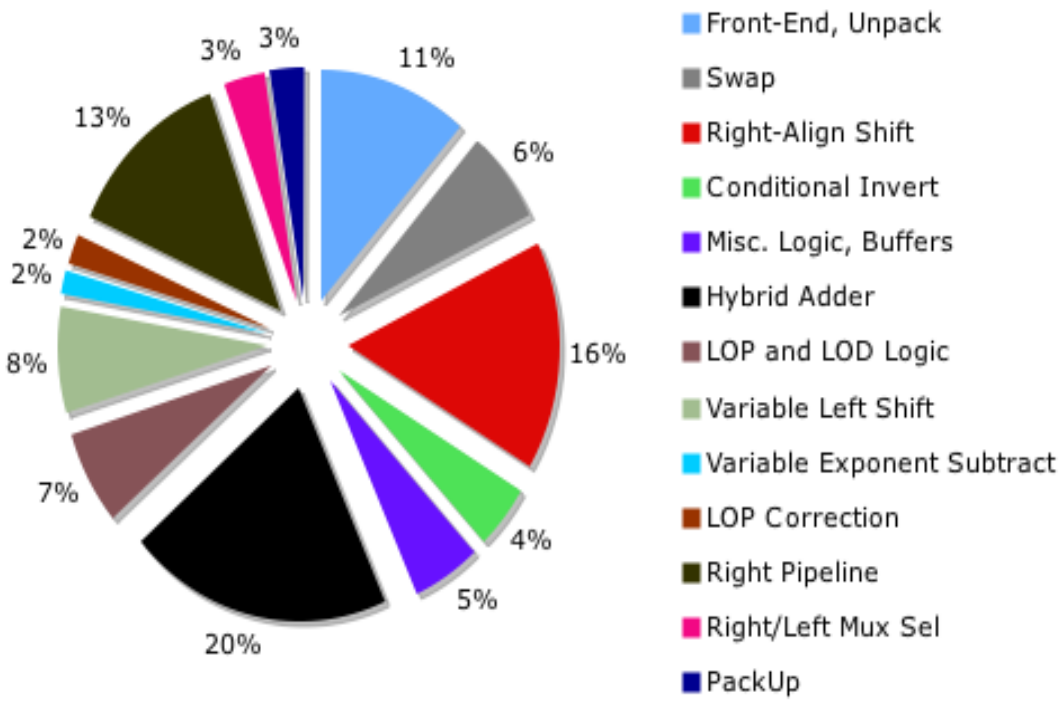

Figure 4.3: FPA Pipeline Power Breakdown

complete rounding logic which includes significand increment logic and exponent increment/decrement logic blocks.

4.2 Coarse-Grain Power Reduction

The delay of an N-bit adder primarily depends on how fast the carry reaches each bit position. In the worst-case, the carry may need to be propagated through all bits, hence synchronous implementations resort to tree adder topologies. However, as shown in Fig. 4.4, for most application benchmarks,

almost 90% of the time the maximum carry-chain length is limited to 7 radix-4 positions.

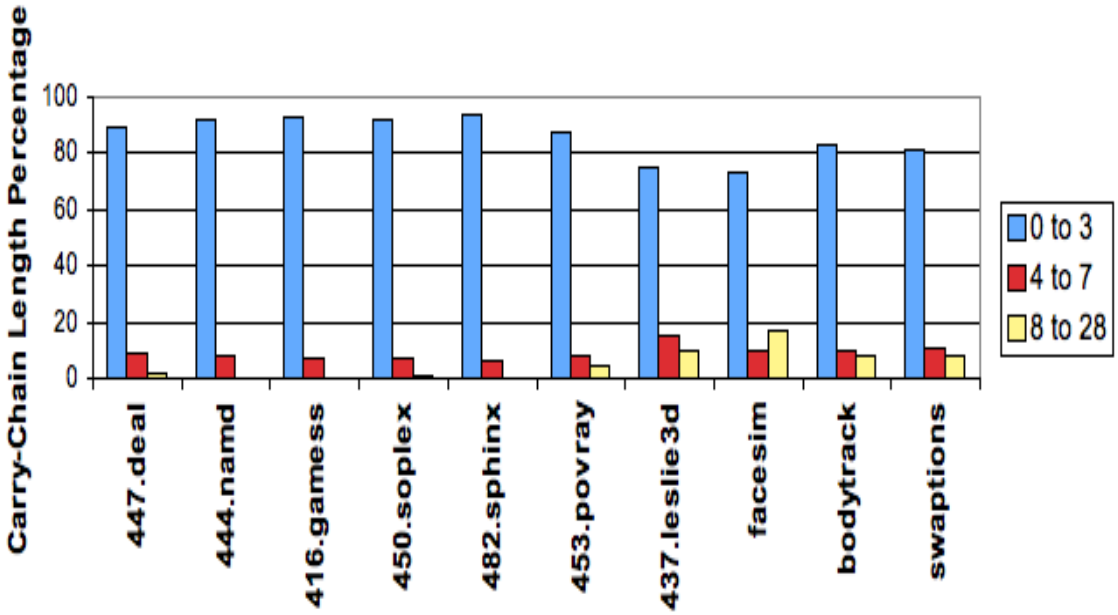

Figure 4.4: Radix-4 Ripple-Adder Carry-Length

An N-bit ripple carry asynchronous adder has an average case delay of $O(\log N)$, the same order as a more complex synchronous parallel-prefix tree adder such as Kogge-Stone. However, the use of ripple-carry asynchronous adders is not feasible for high-performance FPA circuits because the pipeline stage waiting for the carry input stalls the previous pipeline stage until it computes the sum and the carry-out. Even a delay of one carry-propagation (which is two gate delays) stalls the preceding pipeline by a significant amount.

4.2.1 Interleaved Asynchronous Adder

To circumvent the average throughput problem, we use an *interleaved* asynchronous adder as shown in Fig. 4.5. It uses two radix-4 ripple-carry adders: the *left* and *right* adders. Odd operand pairs are summed by the *right* adder, and even operand pairs are summed by the *left* adder. The notion of interleaving blocks has been used for a number of different structures in the past, including FIFOs [16] and high-speed communication circuits [67].

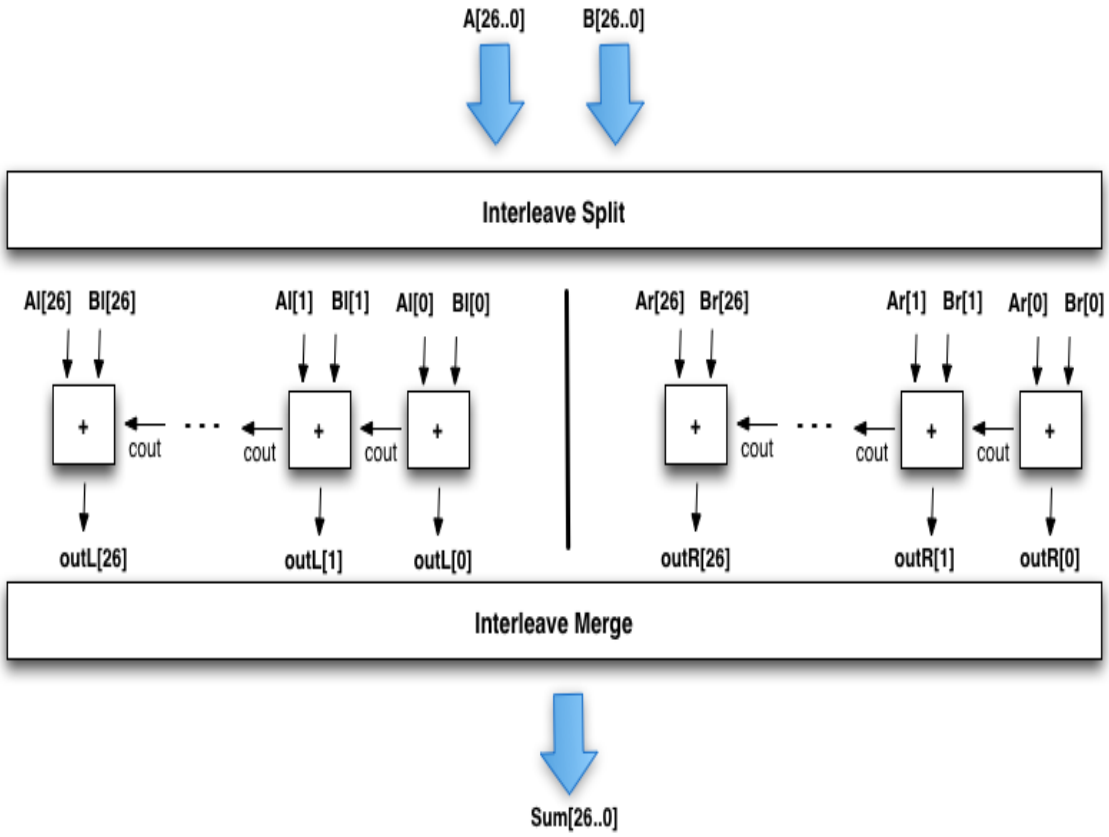

Figure 4.5: Interleaved Asynchronous Adder

In a standard PCeHB reshuffling, the interleave stage has to wait for the acknowledge signal from ripple-stage before it can enter neutral stage and accept

new tokens. However, this would cause the pipeline to stall in case of a longer carry chain. Hence, we do not use PCeHB reshuffling in our adder topology. Instead of waiting for the output acknowledge signals from the *right* ripple-carry adder, the interleave stage checks to see if the *left* ripple-carry adder is available. If it is, the interleave stage asks for new tokens from the previous pipeline stage and forwards the arriving tokens to the *left* adder. The two ripple-carry adders could be in operation at the same time on different input operands. Since our pipeline cycle time is approximately 18 logic transitions (gate delays), the next data tokens for the *right* adder are scheduled to arrive after 36 transitions of the first one. This gives ample time for even very long carry-chains to ripple through without causing any throughput stalls.

Table 4.1 shows the throughput results of our *interleaved* asynchronous adder using SPICE simulations with different input sets. Compared to the 56-bit Hybrid Kogge-Stone Carry-Select Adder which gave a throughput of 2.17 GHz and energy/operation of $13.6pJ$ when simulated by itself, the *interleaved* adder operates at an average throughput of 2.2 GHz for input cases with carry-length of fourteen or less while consuming only $2.9pJ$ per operation. Not only it reduces the energy/operation by more than 4X, it also reduces the number of transistors in the 56-bit adder by 35%.

Deal corresponds to operand data from *447.deal* SPECFP 2006 application benchmark. Other applications from the SPECFP suite had similar statistics, so we simply picked one representative benchmark for comparison. The synthetic input sets (I to IV) are designed to have specific carry chain lengths, as can be seen from the statistics in Table 4.1. The synthetic input sets III and IV generate input operands for the adder that yield fixed maximum carry-chain lengths of

Table 4.1: Throughput across different carry lengths

Input	0-3	4-7	8-14	15-20	27	Frequency
Deal	88%	9%	2.7%	0.3%	0%	2.2 GHz
I	0%	100%	0%	0%	0%	2.2 GHz
II	0%	0%	100%	0%	0%	2.2 GHz
III	0%	0%	0%	100%	0%	1.38 GHz
IV	0%	0%	0%	0%	100%	0.78 GHz

20 and 27 (maximum for radix-4 56-bit addition) respectively. We did observe a dip in throughput for these two input sets, but since our statistical analysis reported earlier in the section show the probability of such high carry-chain lengths to be quite rare, it is feasible to take a throughput penalty for such rare occurrences (0.5% or less) in order to save more than four times the energy per operation for the 99.5% of input patterns with maximum carry-chain length of 14 or less.

4.2.2 Left or Right Pipeline

In our baseline asynchronous FPA, the post-addition datapath is divided into two separate pipelines: *Right* pipeline and *Left* normalize pipeline as shown in Fig. 4.1. The two pipelines handle disjoint cases that could occur during floating-point addition. The *Left* normalize pipeline handles cases when destructive cancellation can occur during floating-point addition, requiring a large left shift for normalization. The destructive cancellation scenario happens only when the exponent difference is less than two, and the FPA is subtracting the

two operands. The *Right* pipeline handles all other cases.

Instead of activating both pipelines and selecting the result, we compute the selection condition early (prior to activating the LOP/LOD stage) and then only conditionally activate the appropriate path through the floating-point adder. The LOP/LOD function blocks determine the shift value for the left normalization shifter. The shift amount determined by LOP/LOD is only needed in cases which could potentially result in destructive cancellation. Hence, in the case of *Right* pipeline utilization, we also save energy associated with the LOP/LOD stage, because the results of the LOP/LOD are only used by the *Left* normalize pipeline. Compared to the baseline FPA, we get power savings of 13% for operands using the *Left* pipeline and power savings of up to 18% (11% *Left* pipe & 7% LOP/LOD) for operands using the *Right* pipeline which is the more frequent case as shown in Fig. 4.6.

4.3 Operand-Based Optimizations

This section further improves the energy-efficiency of the FPA by examining other properties of the input operand distribution. We optimize four additional aspects of the FPA pipeline: (i) initial right align shifter; (ii) leading one prediction; (iii) post-addition increment; (iv) zero input operands.

4.3.1 Two-Way Right-Align Shift

The *Right-Align Shift* block is the second-most power consuming structure in the baseline FPA. It includes the right shifter logic as well as the logic to compute

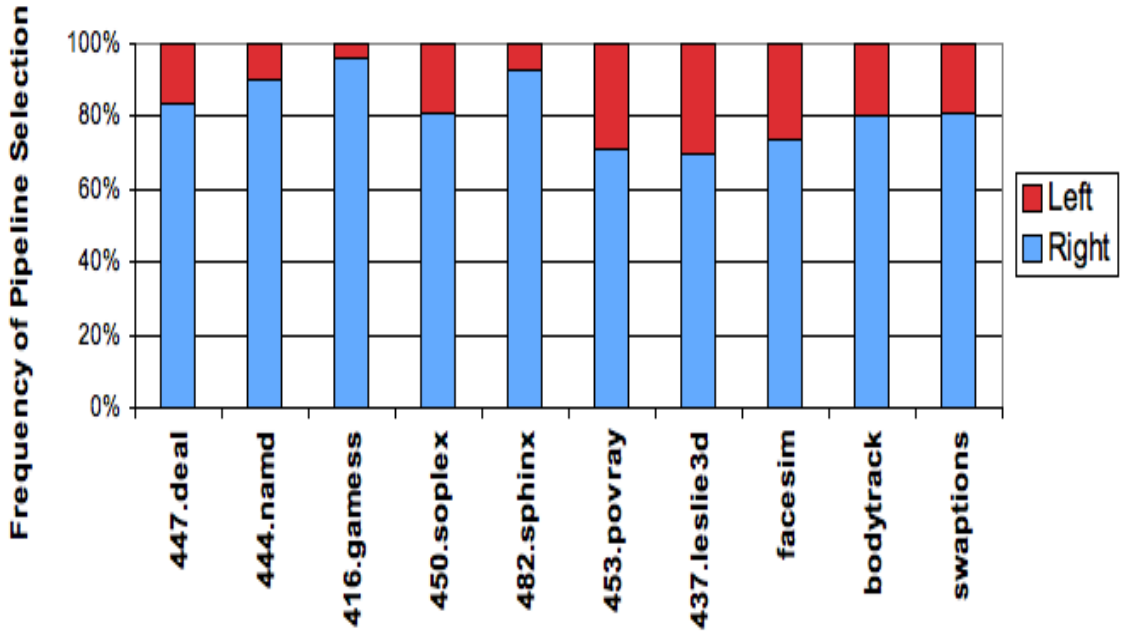

Figure 4.6: Left/Right Pipeline Frequency

the guard, round, and sticky bits used for rounding. The sticky bit is set to one if any of the shifted out bits from the alignment shift stage is one; otherwise it is set to zero. In the worst case, the sticky bit logic has to examine all 53 shifted bits. To do this fast and in parallel with the right-align shifter, considerable extra circuitry is needed which consumes more power. For high throughput, the other (non-shifted) significand is slack-matched to the right-align shift logic using a number of WCHB pipeline stages. The *Right-Align Shift* block also compares the two significands to determine which of the two significands should be inverted in case of subtraction. The exponent difference and sign bit is used to generate enable control for the LOP. Each control bit is shared for two (one for each operand) radix-4 significand entries. Overall, this comparison of significands and generation of large number of control bits is not cheap in terms of

power consumption.

The shifter comprises of three pipeline stages. The first stage shifts the significand between 0 to 3 bit positions based on the shift-control input. The second pipeline shifts by 0, 4, 8, or 12 bit positions and the third stage shifts by 0, 16, 32, or 48 bit positions using the shift-control input signals for the respective stages. Each radix-4 significand entry shift pipeline resembles a PCEHB template with a 4-to-1 multiplexor as the pull-down logic. Each stage produces multiple output copies to feed into 4 different PCEHB multiplexor blocks of the following pipeline stage. All this circuitry makes the shifter a costly structure in our FPA datapath.

The key advantage of the shifter topology is its fixed latency for any shift value ranging between 0 and 55 (the maximum align shift in a double-precion addition/subtraction). This advantage is also one of its drawbacks as it consumes the same power to do a shift by zero and a shift by a large value. Fig. 4.7 shows the right align shift patterns across 10 different benchmarks using operands gathered through PIN application profiling. Although, these benchmark applications are from totally unrelated disciplines, they exhibit a common property: a significant proportion of right align shift values range between 0 to 3 inclusive. For one benchmark, the proportion of right align shifts of 0 to 3 is almost 81%.

In our baseline right-align shift topology, shifts by 0 to 3 are done in the first pipeline stage. However, in spite of that the significand still needlessly goes through the other two shift stages and in doing so wastes considerable power. It would have been an acceptable trade-off if most operations required align shifts by a large value, but the shift patterns shown in Fig. 4.7 make it evident

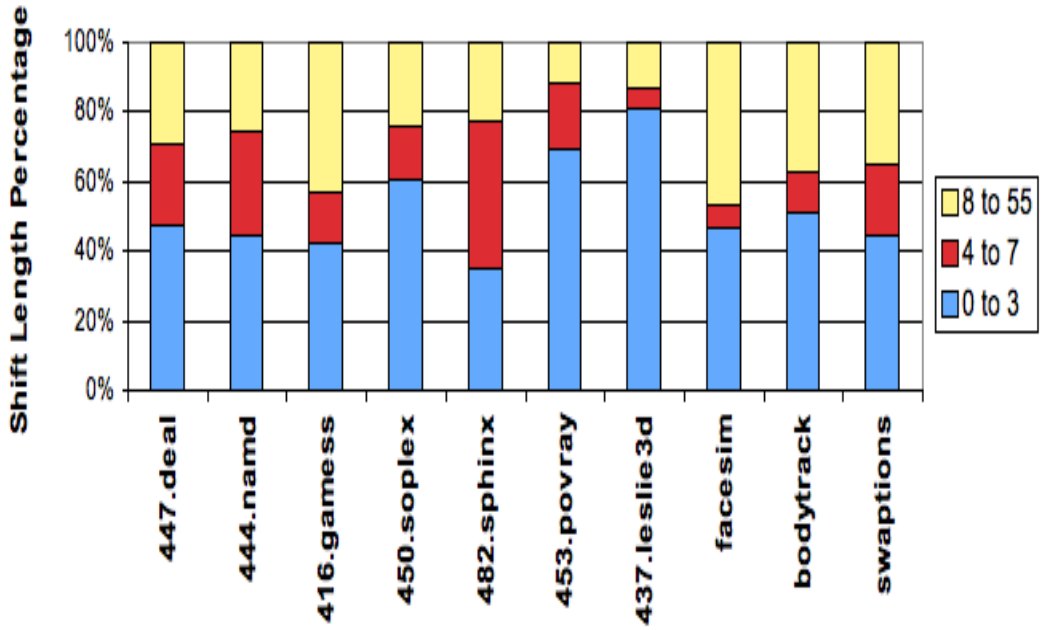

Figure 4.7: Right Align Shifter Statistics

that our baseline align shifter topology is highly non-optimum from an energy perspective.

To improve the energy-efficiency of the align shifter, we split it into two paths. The first stage dealing with a right shift of 0 to 3 is shared between two paths. In case of a shift greater than 3 bit positions, the significand is forwarded to the second shift pipeline stage as in the original topology. However, for shifts of 0 to 3 bit positions, the significand output is bypassed to the post align-shifter pipeline stage as shown in Fig. 4.8. The post align-shift stage consists of a merge pipeline which receives inputs from both the regular shift path and the short bypass shift path. It selects the correct input using the buffered control signal which was earlier used to direct the significand to one of the two paths.The short shift path has multiple features which lead to significant power savings:

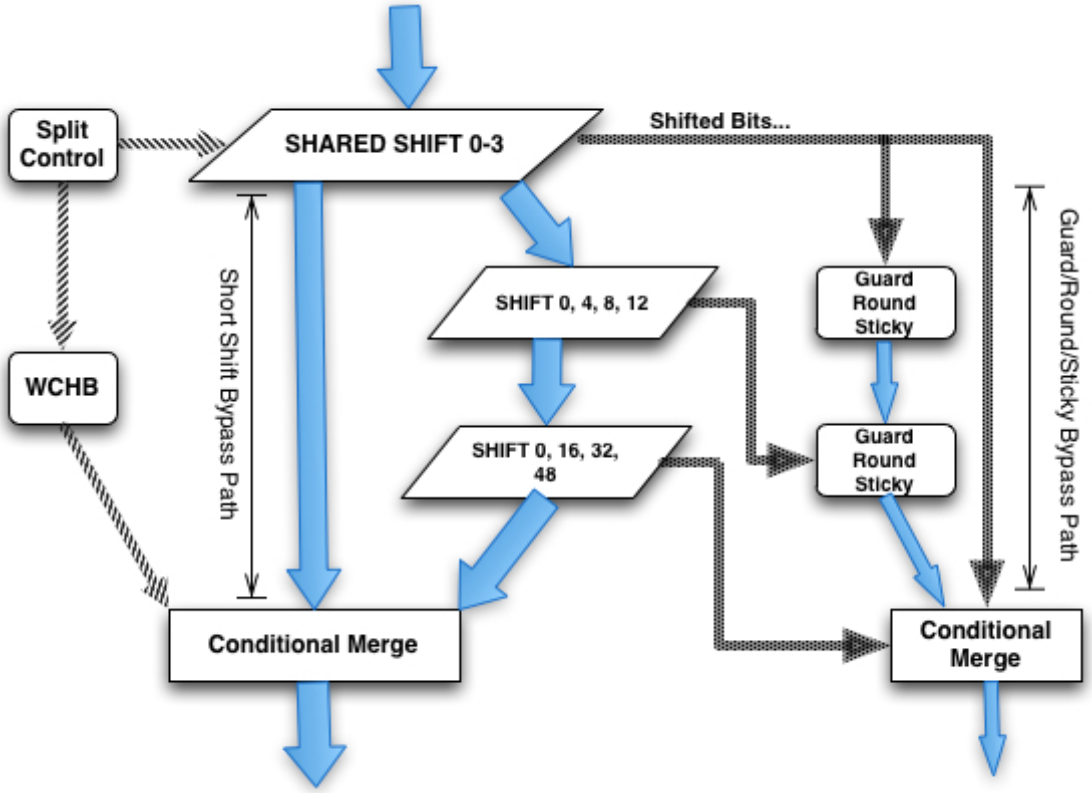

Figure 4.8: Two-Path Right-Align Shift

- The shifted significand skips the remaining two shift pipelines.
- In contrast to the baseline topology which produces multiple significand outputs to be consumed in the following shift stages, the bypass shift path needs only one output for each significand.
- The guard, round, and sticky computation becomes quite simple and requires minimal energy as only a maximum of 3 bits are shifted out.
- The other (non-shifted) significand also bypasses the WCHB slack-matching buffers.
- No shift select signals need to be generated and copied for the second and

third shift pipeline stages.

The new shifter topology poses a design choice of slack-matching the control to either the long-shift path with two pipeline stages or the short-bypass path with no pipeline buffering at all. If control is slack-matched to the short path, the shifts requiring long path may suffer from stalls and degrade the FPA throughput. Slack-matching the control to the long path increases the short path latency. The worst-case scenario is when the pipeline alternates between the two paths. However, our application profiling analysis in Fig. 4.9 reveal that across all application benchmarks, the proportion of times a short path shift follows another shift along the same path is considerably high. We saw similar results for the long path shifts. A detailed throughput and latency analysis, based on the profiled shift patterns, favored a control path which is slack-matched to none of the two shift paths. In our implementation, the merge control input has only one WCHB pipeline and has a throughput within 1.3% of the baseline FPA in the worst-case scenario.

4.3.2 Minimizing LOP Logic

For subtraction, the bits of the shifted significand are inverted except when the exponent difference is zero which then requires input from the significand comparison block to determine which one of the two significands is smaller. Since the case of exponent difference of zero corresponds to the bypass shift path, the significand comparison logic requiring multiple pipeline stages cannot be done in parallel with the bypass path without incurring a throughput penalty. Hence, the significand comparison is moved to earlier pipeline stages in the optimized

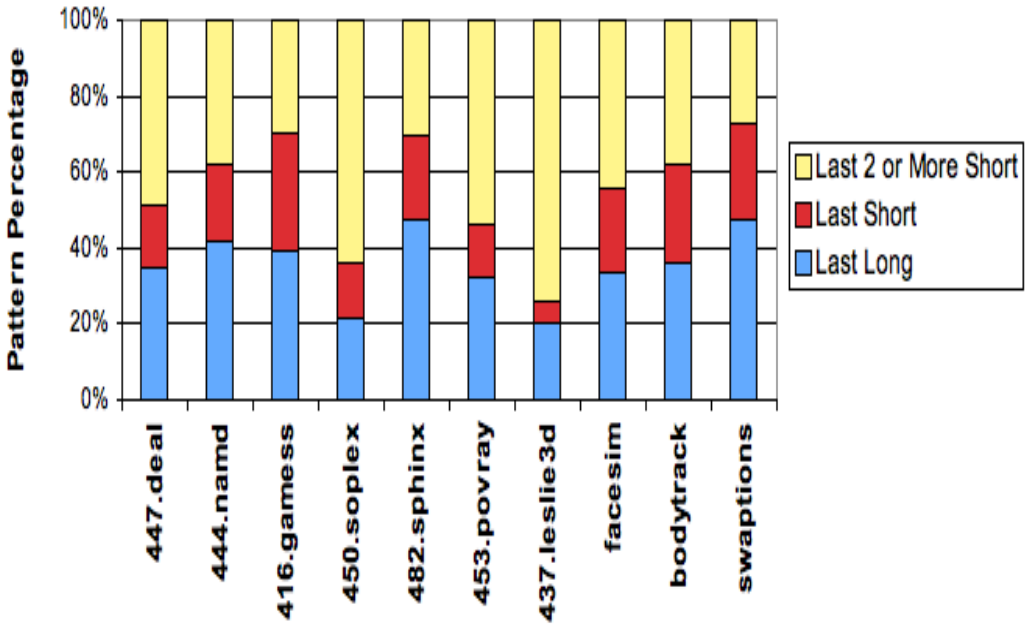

Figure 4.9: Right Align Shift Short Path Pattern

FPA datapath.

With the result of significand comparison available early, the LOP logic stack can be simplified. As Bruguera et al. point out in [12], the logic to predict leading one when the sign-digit difference of two operands is positive is different from the case when the sign-digit difference of two operands is negative. In our optimized FPA, using the significand comparison result early in the FPA enables the LOP computation to assume that its first operand always corresponds to the larger significand. This information enables us to significantly reduce the circuitry required for LOP computation.

In the baseline FPA, there is a separate pipeline stage to conditionally invert bits in case of subtraction. The baseline FPA generates control signals for each radix-4 position specifying which of the two significands if any need to be

inverted. Since the LOP control bits in our optimized FPA already contain information about the larger significand, we merged the conditional invert stage with pre-LOP selection pipeline which determines the larger of the two significands as LOP's first operand. This eliminates the need of separate control signals for inverting bits and including savings from cutting a full pipeline stage leads to energy reduction of over 3%.

4.3.3 Post-Add Right Pipeline

The *Right Pipeline* block is the third most power-consuming structure in the baseline FPA. It includes a single-position right or left shifter, a 53-bit significand incrementer, rounding logic, and final exponent computation block for operands utilizing the *Right Pipeline*. As shown earlier in Fig. 4.6, on average more than 80% of the FPA operations use this block. Hence, power-optimization techniques for the circuits in this block have a notable impact on average FPA power savings.

The baseline carry-select incrementer comprises of four-bit blocks with each computing the output for the carry input of one into that block. In parallel, there is a fast carry-logic which computes the correct carry-input for each four-bit block. Lastly, there is a mux pipeline stage which selects either the incremented output or the buffered non-incremented bits for each four-bit block using the carry select input. In case of a carry-out of one, the significand is right shifted by one bit position.

The key advantage of our baseline incrementer topology is its fixed latency for the best (no carry propagation) and worst-case (carry propagates through

all the bits) alike. However, as seen in Fig. 4.10, for over 90% of the operations using the increment logic, the carry propagation length is less than four radix-4 bit positions. Also, the case of a final carry-out occurs no more than 0.5% of the time.

The carry-select incrementer targeted for worst-case scenarios is a non-optimum choice for the average-case incrementer carry-length patterns. To improve energy-efficiency, we instead use an interleaved incrementer similar to earlier described interleaved adder. Instead of using two ripple-carry adders, it uses much simpler two radix-4 ripple-carry incrementers. The odd data token is forwarded to the *right* incrementer. For the next arriving data token, the interleave stage checks to see if the *left* incrementer is available. If it is, the interleave stage forwards the arriving tokens to it. The interleave merge stage receives the inputs from both incrementers and forwards those to the next pipeline stage in the same interleaved order in which they were scheduled. This allows the two incrementers to be in operation at the same time on different input operands.

The incrementer is used to adjust the result due to rounding. Our new incrementer topology computes either the correct incremented or non-incremented output (not both) using the round-up bit as the carry-in, hence alleviating the need to have a separate mux stage to choose between two possible outputs. Our simulation results for the new topology show no throughput penalty for average-case inputs. Also, there is no need for a separate post-increment right shift pipeline stage. The case where the final result must be right shifted by one only occurs when all significand bits are one, and the result must be rounded up. In that scenario, the incrementer output is all zero and hence both shifted and unshifted versions of the incrementer result are identical. Hence, for correct

output, only the most significant bit needs to be set to one.

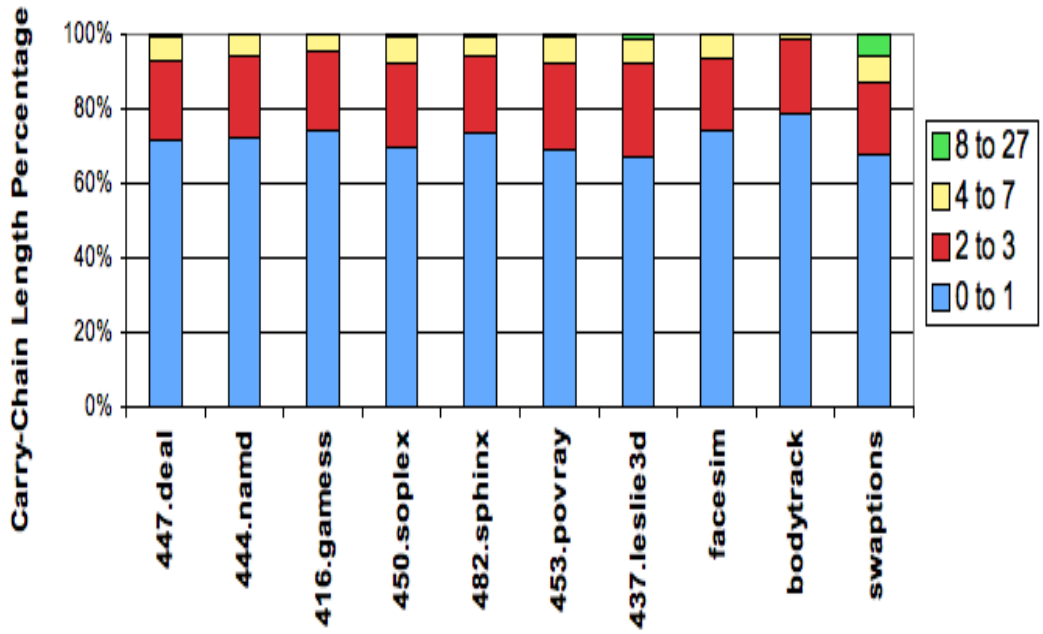

Figure 4.10: Radix-4 Incrementer Carry Length

In the baseline FPA, until the incrementer *carry-out* is computed the correct exponent value cannot be computed. Since the carry-out is not available until the fourth pipeline stage in the *Right Pipeline* block, to prevent latency penalty the exponent values of *exponent*+C are always computed for $C = 0, \pm 1, +2$, with a mux stage choosing the correct output. To circumvent the problem of latency penalty, we replace the exponent computation block with an interleaved incrementer/decrementer which mitigates any latency degradation with its average-case behavior. It uses a two bit carry-in (first bit is set to 1 for increment, second bit is 1 for decrement, and both bits are 0 for a simple pass through) to compute *exponent*. Using dual-carry chain, *exponent + 1* is also computed simultaneously to be selected in case of a carry out. Overall, this computation of two exponent values is far more energy-efficient than the baseline.

4.3.4 Zero-input Operands

Fig. 4.11 shows that a few application benchmarks have a significant proportion of zero input operands. For the applications involving sparse-matrix manipulations such as *Deal* and *Soplex*, in spite of the use of specialized sparse-matrix computation libraries, the percentage of zero inputs can be as high as 36%. For other benchmarks, the zero-input percentage varies widely. In our baseline FPA and almost all synchronous FPA designs, operations involving zero-input operands use the full FPA datapath. Although, if one or both of the FPA operands are zero, the final FPA output could be computed without needing power-consuming computational blocks such as right-align shifter, significand adder, LOP/LOD, post-add normalization, and rounding.

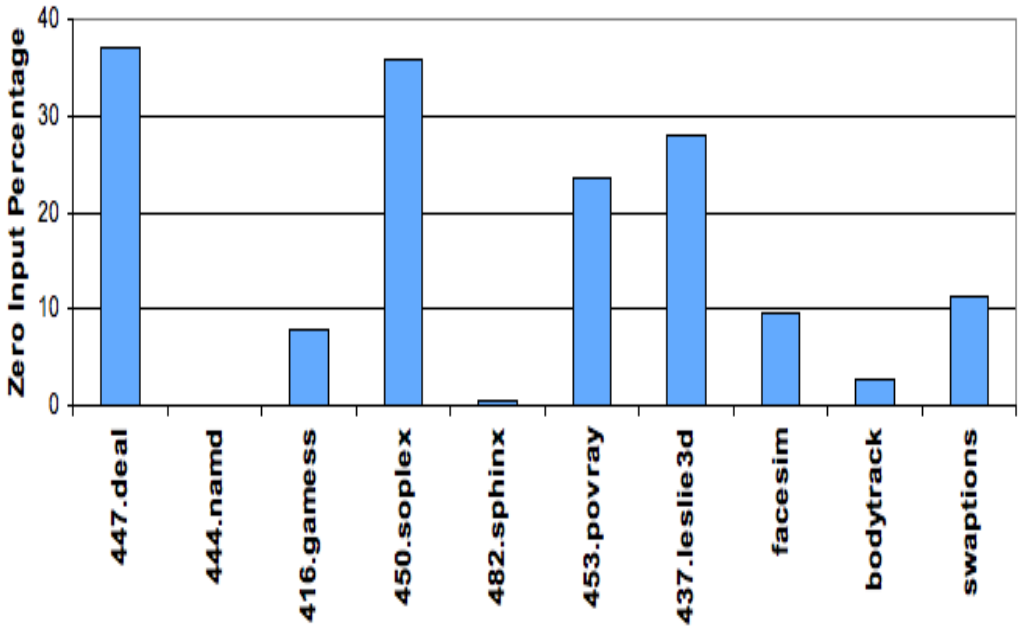

Figure 4.11: Zero-input Operands

Since the *Unpack* pipeline stage already checks to see if any operand is zero,

our optimized FPA utilizes the zero flag to inhibit the flow of tokens into the regular datapath. The zero flag is used as a control in the conditional split pipeline just prior to *Swap* stage to bypass the final sign, exponent, and significand bits to the last pipeline stage in case of a zero input. The last stage is replaced with a conditional merge pipeline which uses the buffered control signal to choose the input from either the zero bypass path or the regular FPA datapath. The huge slack disparity between two split pipelines makes the choice of control slack a critical one.

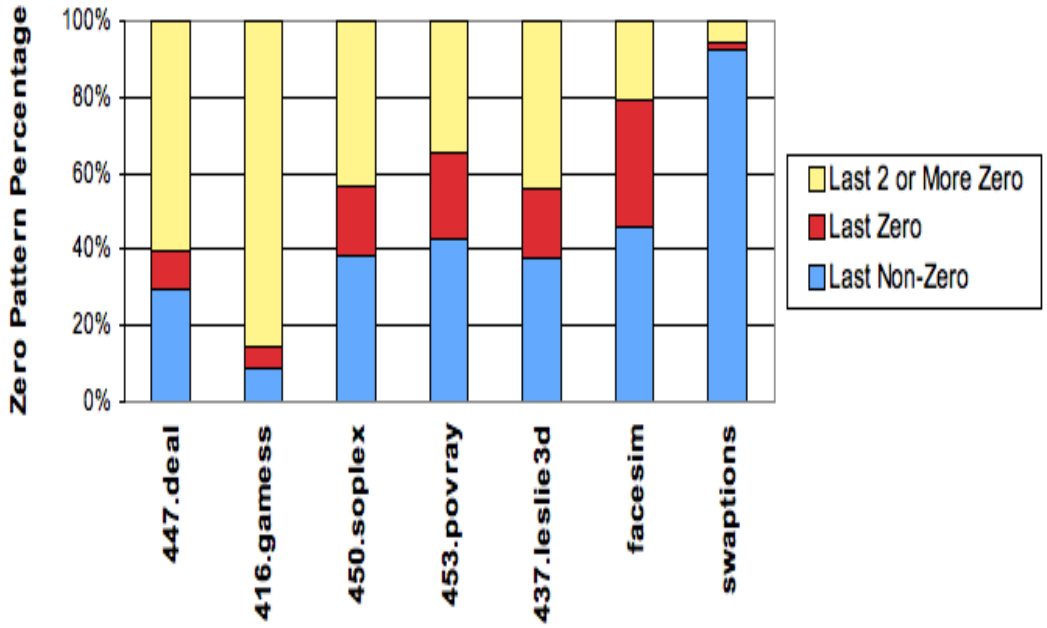

Figure 4.12: Zero-input Pattern

Fig. 4.12 shows that for benchmark applications with significant proportion of zero inputs, the percentage of a zero-input followed by another zero-input operation is quite high except for the *Swaptions* benchmark. To choose the optimum level of control buffering, we simulated the optimized FPA with a number of synthetic input-patterns over a wide-range of control slack possibilities

as seen in Fig. 4.13. *Mix-flip* refers to input sequence with alternating zero-input and nonzero-input operands. *Mix-pattern* sequence closely resembles the zero-input pattern seen in most benchmark applications. Based on these results, we chose to buffer the control with eight WCHB pipeline stages.

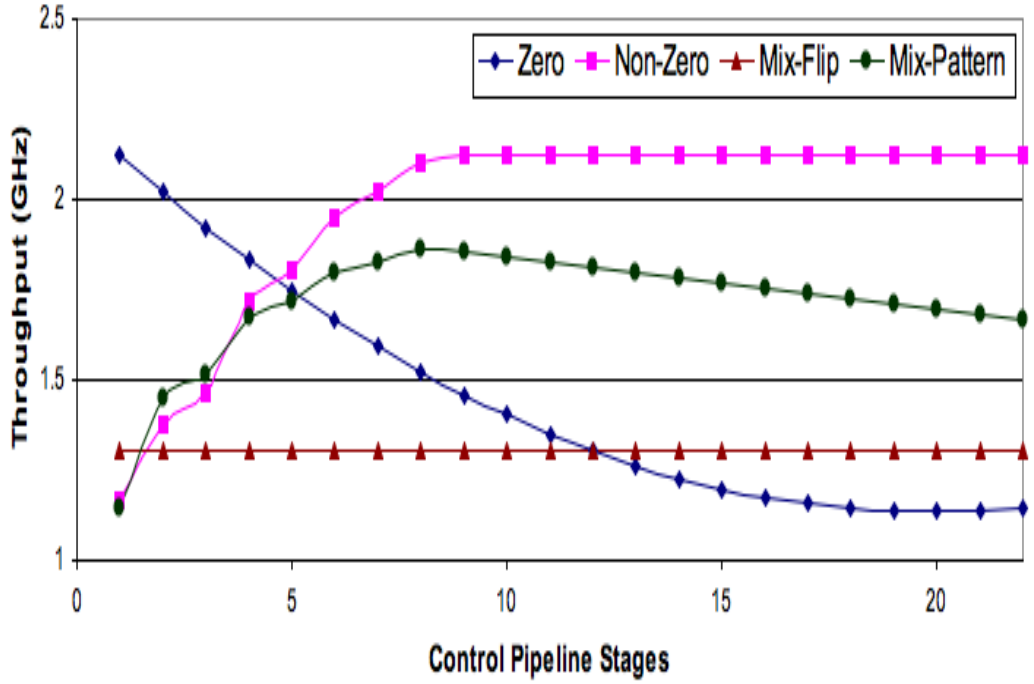

Figure 4.13: Zero-Path Control Slack Analysis

Some zero-input patterns take a significant throughput hit even with eight WCHB pipeline stages for the control. To circumvent this problem, we explored the effect of adding some slack on the bypass path. Fig. 4.14 shows that the addition of two WCHB stages on the bypass path for sign, exponent, and significand bits greatly alleviates the throughput penalty albeit at a small cost in energy. Overall, the best throughput results are again attained with a slack of eight WCHB stages on the control. For *Mix-pattern* sequence, the throughput increases by 7.5% to 2 GHz. For the worst-case input set, *Mix-flip*, throughput

increases by 49.8% to 1.95 GHz. The improvement in throughput comes at a cost of extra WCHB logic and hence more power. Our simulations using only one WCHB stage didn't show such profound throughput improvement and for cases beyond two WCHB pipeline stages, the small increases in throughput are overshadowed by power consumed in additional buffer stages.

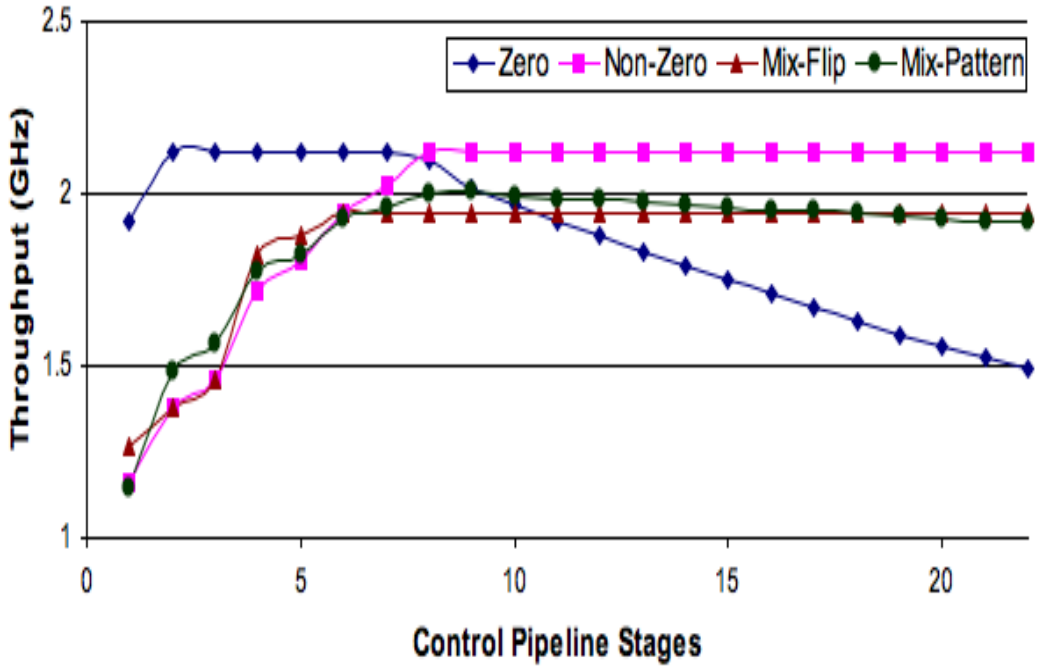

Figure 4.14: 2-WCHB Zero-Path Control Slack

4.4 Evaluation of Operand-Optimized FPA

The functional correctness of our asynchronous operand-optimized FPA was verified using `prsim`, our in-house asynchronous gate-level simulation tool. Ten billion randomly generated inputs were sourced into the FPA and the out-

puts were verified against the expected values from a standard processor. The random input set included verification tests for all four IEEE rounding modes as well as denormal, NaN, and infinity data inputs. The FPA was further tested with one billion stored inputs from actual application benchmarks. In the past, many of the designs have opted to handle denormal numbers using software traps [55] which can lead to long execution times [59]. Our design includes hardware support for denormal numbers.

Our improved asynchronous FPA combines all optimization techniques discussed in sections 2.2 and 4.3. On top of the energy savings associated with the these techniques, we were able to compact more logic together and in doing so eliminated a full pipeline stage. The transistors in our baseline FPA were sized using standard transistor sizing techniques [71]. To meet high performance targets, the pull-down stack was restricted to a maximum of six transistors in series (including the enable). The slow and power-consuming state-holding completion-elements were restricted to a maximum of three inputs at a time. Keepers and weak feedback inverters were added for each state-holding gate to ensure that charge would not drift even if the pipeline were stalled in an arbitrary state.

Since HSIM/HSPICE simulations do not account for wire capacitances, we included additional wire load in the SPICE file for every gate in the circuit. Based on prior experience with fabricated chips and post-layout simulation, we have found that our wire load estimates are conservative, and predicted energy and delay numbers are typically 10% higher than those from post-layout simulations. Our simulations use a 65nm bulk CMOS process at the typical-typical (TT) corner. Test vectors are injected into the SPICE simulation using

a combined VCS/HSIM simulation, with Verilog models that implement the asynchronous handshake in the test environment. All simulations were carried out at the highest-precision setting.

As seen in Fig. 4.15, the energy per operation of the optimized FPA is approximately 2.3X (56.7%) less than that of baseline FPA across a wide range of throughput values for the same non-zero operands. In terms of overall throughput, our optimized FPA is within ±1.5% of the baseline FPA across a range of voltages (0.6V to 1.1V). As noted earlier, it is possible to create pathological input operands that could degrade the throughput, for example long carry-chain lengths in the interleaved adder/incrementer or the case of alternating zero and non-zero operands; however, in practice, such inputs are rare. Even if they do occur, our FPA still operates correctly and produces IEEE-compliant output albeit at lower throughput.

The baseline FPA gives an energy-per-operation of $69.3pJ$ at an average throughput of 2.15 GHz for all input operands alike. The optimized FPA's energy-per-operation and throughput vary considerably based on the input operands as seen in Table 4.2. These results, for SPICE simulations at a V_{DD} of 1V with no slack on the zero operand bypass path, show our improved FPA design to be far superior in energy-efficiency than our baseline FPA.

The energy-efficiency and throughput results for the FPA implementation with two WCHB pipeline stages on the zero bypass path are shown in Table 4.3. The results for non-zero operands remain the same as before and hence are not repeated. The improvement in throughput for all zero-input patterns comes with additional power consumption. This offers a design choice to be made based on throughput and energy targets.

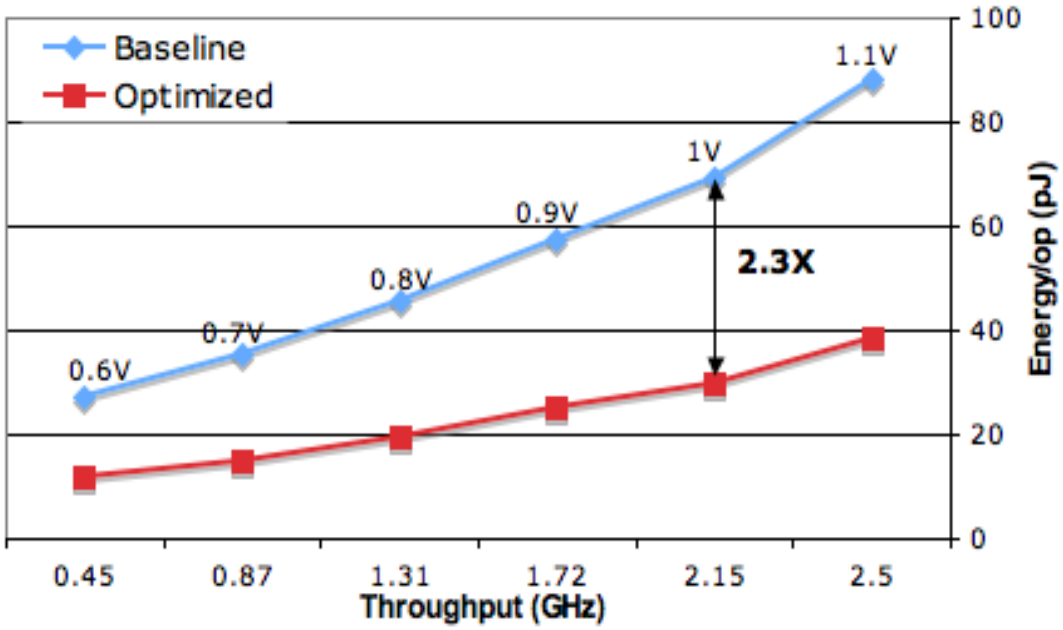

Figure 4.15: Optimized vs. Baseline

Table 4.2: Optimized FPA Energy & Throughput

Input Set	Energy/FLOP	Throughput
Nonzero (Align Shift 0-3)	30.2 pJ	2.15 GHz
Nonzero (Align Shift 4-55)	35.1 pJ	2.10 GHz
Nonzero (Align Shift Mix)	32.4 pJ	2.12 GHz
Zero Only	13.1 pJ	1.51 GHz
Zero-Nonzero Alternate	25.1 pJ	1.31 GHz
Zero 30%	27.4 pJ	1.85 GHz
Zero 8%	31.0 pJ	1.96 GHz

Table 4.3: Optimized FPA 2-WCHB Zero Bypass

Input Set	**Energy/FLOP**	**Throughput**
Zero Only	14.2 pJ	2.1 GHz
Zero-Nonzero Alternate	26.1 pJ	1.95 GHz
Zero 30%	28.4 pJ	2.0 GHz
Zero 8%	32.1 pJ	2.1 GHz

In terms of actual application benchmarks, *Zero 8%* input mix corresponds to *416.gamess*, whereas *Zero 30%* corresponds to an average mix of operands from three applications: *447.deal*, *450.soplex*, and *437.leslie3d*.

The latency of our optimized FPA is also highly operand dependent. Table 4.4 shows that compared to the baseline FPA's average latency of approximately 1098ps, the optimized FPA has an average latency of 737ps for zero operand cases (same for both two WCHB slack matching and no slack matching zero bypass implementations) and 1060ps for nonzero operands with align shifts of 0 to 3; a latency reduction of 32.8% and 3.5% respectively. The increase in latency, seen for rare some cases, could be attributed to the use of a variable-latency interleaved adder instead of fixed latency parallel-prefix tree adder.

Table 4.4: Optimized FPA Latency

Input Set	**Latency**
Nonzero (Align Shift 0-3)	1050-1070 ps
Nonzero (Align Shift 4-55)	1080-1120 ps
Zero	737 ps

Since leakage power has become an important design constraint, our simulations model sub-threshold and gate leakage effects in detail. Table 4.5 compares the total leakage power of our baseline and optimized FPAs at a V_{DD} of 1V. Although, our optimized FPA includes extra control circuitry for multiple split-merge pipelines, there is a 19% reduction in leakage power.

Table 4.5: Leakage Power

	Leakage Power
Baseline FPA	0.72 mW
Optimized FPA	0.58 mW

The decrease in leakage power could be attributed to the use of the interleaved adder and incrementer which use far fewer transistors compared to the Hybrid Kogge-Stone Carry-Select Adder and Carry-Select Incrementer. Also, compacting of logic stages eliminated a full pipeline stage and helped to reduce the total leakage power further. In terms of the total number of transistors, our optimized FPA uses 12% less transistors than the baseline.

Table 4.6 compares the performance, power, and energy of our optimized FPA against both our own baseline and some of the latest FPAs and FMAs from industry and academia. The computer arithmetic literature has a large body of work on FPA and FMA designs, but few contain a detailed implementation that provides a reasonable point of comparison in a modern process. This guided our choice of other FPA/FMAs in Table 4.6. Our baseline and optimized FPA results are for simulations with an input-set comprising non-zero operands with right align shifts of 0 to 3.

We caution that the FMA numbers are not meant to be a direct comparison

with our proposed FPA since an FMA contains additional circuitry. The FMA performance and power numbers were only included to show what is the best out there in industry and academia and that in spite of using a bulk process, our proposed FPAs are competitive both in terms of performance and energy-efficiency. Quinnell [54] has a lower overall latency for nonzero operands than our optimized FPA as well as our baseline FPA. However, this lower latency does not take into the latching overhead, which could be significant in high performance designs, and comes at the cost of 3.2X lower throughput and 5.9X higher energy per operation, as well as a higher V_{DD}.

Table 4.6: Comparison to other FPAs and FMAs

Name	**Type**	**Process**	**VDD**	**Frequency**	**Latency**	**Energy/Op**
Async Optimized	FPA	65nm	1	2.15 GHz	57.2FO4s 1060ps	30.2pJ
Async Baseline	FPA	65nm	1	2.15 GHz	59.3FO4s $\approx$ 1098ps	69.3 pJ
IBM Power6 [69]	FMA	65nm SOI	1.1	4 GHz	78FO4s	77.5 pJ
Merrimac [17]	FMA	90nm	1	1 GHz	NA	110 pJ
Quinnell [54]	FPA	65nm SOI	1.3	666 MHz	946ps	177.17 pJ

All of our transistor-level simulation results quoted so far were for HSIM/HSPICE simulations done at a default temperature of 25°C. A set of simulations at 85°C showed a similar trend between the baseline and optimized FPAs but with an expected small performance degradation (10%) at higher temperature.

The high GFLOPS/Watt ratio of our optimized asynchronous FPA (26 GFLOPS/Watt at 2.5 GHz 1.1V) make a case for adopting asynchronous cir-

cuit solutions, similar to ours, in future high performance computing systems. Since asynchronous chips have been shown to work at fairly low voltages and are quite robust [20, 43, 23], getting 85.4 GFLOPS/Watt at a decent throughput of 450 MHz (at 0.6V) also shows the potential of our solution for embedded systems that require floating-point computation.

4.5 Summary

We presented the detailed design of an asynchronous high-performance energy-efficient IEEE 754 compliant double-precision floating-point adder. Using QDI asynchronous pipelines, we created a high-performance design based on state-of-the-art FPA architectures. We analyzed the power consumption of the FPA datapath, identifying opportunities for energy reduction. By using asynchronous techniques that exploit average-case behavior, we reduced the energy of the FPA operation with nonzero operands by 56.7% compared to our baseline implementation while preserving the average throughput.

Chapter 5

Floating-Point Multiplication

In this chapter, we present the details of our energy-efficient asynchronous floating-point multiplier design. We discuss design trade-offs of various multiplier implementations. A higher radix array multiplier design with operand-dependent carry-propagation adder is presented which yields significant energy savings while preserving the average throughput. We provide a number of operand-dependent optimizations across the entire FPM datapath to reduce energy consumption. Our FPM implementation also includes a hardware implementation of special cases in the IEEE-754 standard such as denormal and underflow cases.

5.1 Introduction

Traditionally, most floating-point units have been designed from the perspective of scientific applications. In these traditional benchmark applications, ad-

dition and subtraction operations account for most of the floating-point operations [53, 65]. As a result, optimizing the floating-point adder datapath has remained the prime focus of arithmetic circuit designers. However, a dynamic instruction profile, as seen in Figure 5.1, of a number of emerging commercial, consumer, engineering, and advanced scientific applications shows floating-point multiplication operations to be as frequent as floating-point addition/subtraction operations.

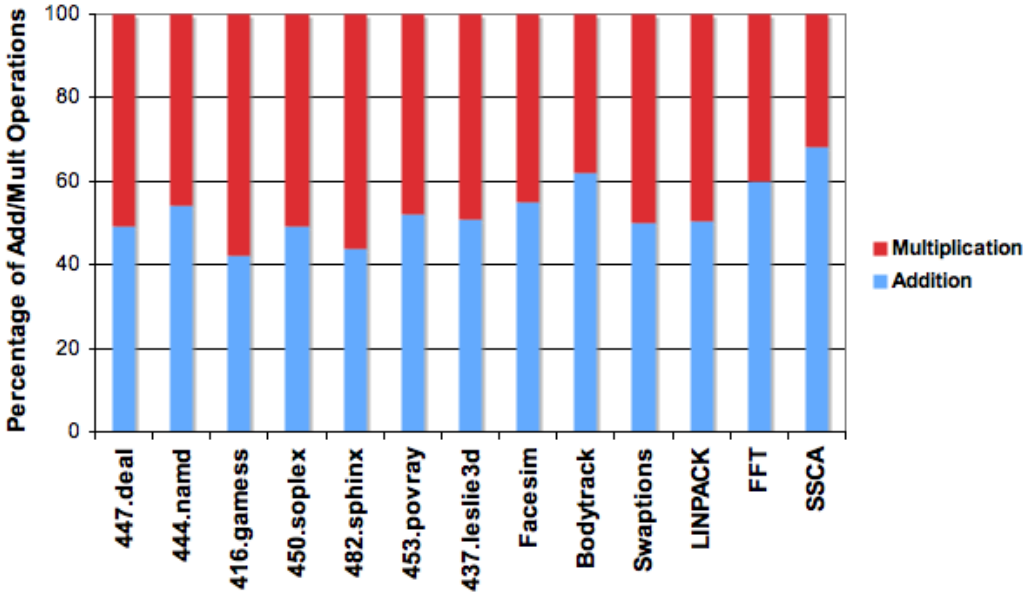

Figure 5.1: Frequency of floating-point instructions

A floating-point multiplier (FPM) consumes significantly more energy compared to a floating-point adder (FPA) [54]. This combined with the knowledge that the frequency of floating-point multiplication operations in emerging applications is similar to that of floating-point addition computations makes energy and power optimizations in the FPM datapath highly essential for an efficient full floating-point unit (FPU) design. In this thesis, we introduce a number of

micro-architectural and circuit level optimizations to reduce power consumption in the FPM datapath.

5.2 Power Breakdown and Analysis

Unlike in the FPA datapath where total power is distributed roughly evenly amongst a number of different logic blocks, the FPM's complexity is largely a function of its *53x53 multiplier*. This is highlighted in Figure 5.2 which shows the power breakdown estimates of a fully QDI FPM datapath. QDI booth-encoded array multiplier accounts for roughly 76% of the total power consumption. Hence, in this thesis, we primarily focus on reducing energy/power of the array multiplier block.

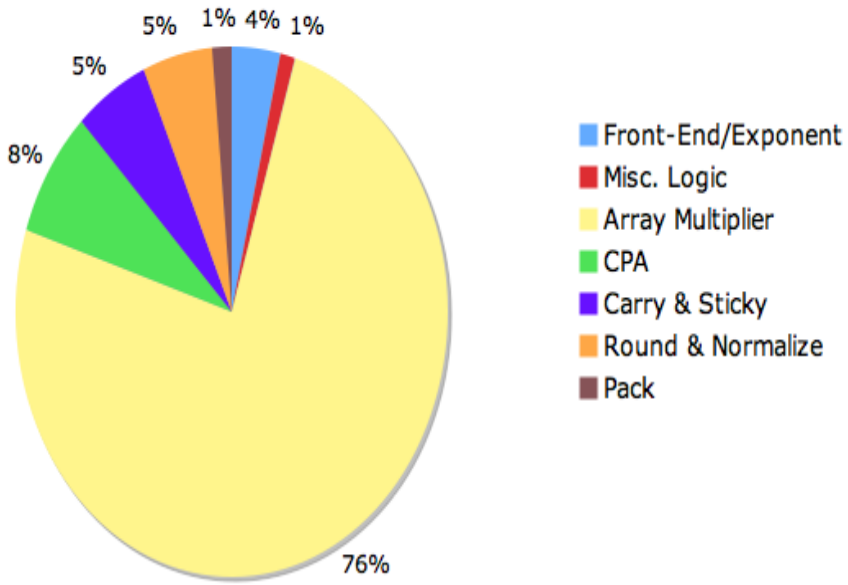

Figure 5.2: FPM Pipeline Power Breakdown

The *Front-End/Exponent* block corresponds to the logic that unpacks IEEE format inputs and analyzes the sign, exponent, and mantissa bits of each input to determine if the inputs are standard normalized or are of one of the special types (NaN, infinity, denormal). It also includes the logic to compute the resultant exponent of the FPM product, which is a sum of the exponent values of both inputs minus the bias. The bias has a value of *1023* in case of double-precision operations. The array multiplier outputs two 106-bit streams. The most significant 53-bits of the two output bit streams from the array multiplier are summed up using a carry propagation adder (CPA) to generate a 53-bit mantissa. The least significant 53-bits are used to generate the carry input to the CPA as well as compute the guard, round, and sticky bits to be used in post normalization rounding. The *sticky bit computation block* and the final *carry propagation adder* are the other power consuming structures within the FPM datapath which show opportunities for operand-dependent optimizations. The post multiplication step includes normalization of the 53-bit mantissa. For normal inputs and non-underflow cases, either the mantissa is already normalized or it may require a right shift by a single bit position, in which scenario the exponent is adjusted, in parallel, by adding one to it. The *pack* block checks for NaN, infinity, or denormal outcome before outputting the correct result in the IEEE format.

In this thesis, we present various structural and circuit-level optimization techniques to reduce the complexity and power consumption footprint of the aforesaid logic blocks.

5.3 Multiplier Design Trade-offs

The choice of a particular multiplier design depends on a number of factors. These include: desired throughput, overall latency, circuit complexity, and the allowed power budget. There is a large body of work on multiplier designs [71, 21]. Traditionally, high performance has been the key driving factor in multiplier design. However, as power consumption has become a major design constraint lately, a number of low-power multiplier designs have been proposed both in synchronous [29, 15] and asynchronous domains [37, 30, 27].

5.3.1 Iterative Multipliers

Iterative multipliers represent a low complexity design choice. An iterative multiplier utilizes a few functional units repeatedly to produce the result. In a simple iterative n by n multiplier implementation, where n is the number of bits, the product is computed after n iterations. Each iteration comprises a minimum n-bit addition and a serial shift by one-bit position.

Iterative multipliers can be used to reduce energy consumption by exploiting input data patterns; stages which add zero to the partial product could be detected in advance and skipped, hence reducing delay and energy consumption. The delay variability nature of iterative multipliers has made them popular amongst asynchronous designers [19, 30]. Furber et al [37] proposed a low power integer multiplier which exploits the commonly occurring pattern of low number of significant bits in integer inputs as means to reduce the total number of iterations. These iterative multiplier designs, though highly energy efficient

and compact in terms of area, are not feasible to be used in a floating-point multiplier hardware due to their very high latency and low throughput and the fact that unlike the inputs in integer arithmetic, the most significant bits of floating-point mantissa inputs are non zero.

Reduction in the total number of partial products is the key goal of all multiplier optimization techniques, as it helps to reduce both latency as well as energy consumption. Along these lines, Efthymious et al [19] proposed an asynchronous multiplier implementation based on the *original* Booth algorithm [10]. Their design scans the multiplier operand and skips chains of consecutive ones or zeros. This can greatly reduce the number of partial product additions required to produce the product. The downside is that it requires a variable length shifter to correctly align multiplicands for generating each partial product row. The effectiveness of this algorithm for high performance FPM hardware is dependent on the number of variable length shifts, which in turn depends on the number of partial product rows that are to be generated. Our application profiling results in Figure 5.3 indicate that although the *original* Booth algorithm is able to reduce the number of partial products from the maximum of 27, a sufficiently large number of partial products rows, more than 18 on average, still need to be generated, each of which requires the use of variable shifter. The latency overhead of such a large number of variable shift operations is too costly for any high performance FPM design. Hence, we did not pursue this algorithm any further.

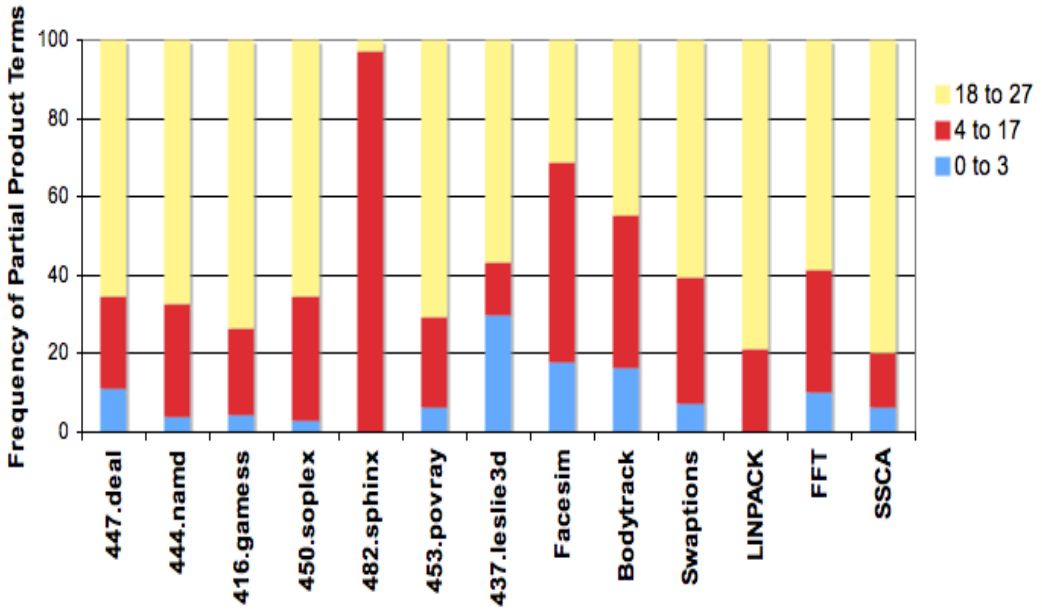

Figure 5.3: Number of partial products terms with original Booth algorithm .

5.3.2 Array Multipliers

Array multipliers are the common choice for high throughput and low latency multiplication operations in most commercial FPM designs [69, 50]. They produce a pre-determined fixed number of partial products, which greatly minimizes if not fully eliminates the opportunities for exploiting data dependent optimizations. For example, introducing logic to bypass a zero partial product instance may add the same amount of delay as summing the extra term in a carry save adder (CSA) used to reduce the partial product terms. As array multipliers present very limited opportunities for data dependent optimizations, there has not been much work on asynchronous array multiplier solutions.

The simplest implementation of an n by n array multiplier produces n partial

products in parallel, which are then summed up using CSAs. The large number of partial products makes this simple design unfeasible for both latency and power consumption perspective. As a result, many advanced multiplier implementations from academia [54] and industry [69, 50, 73] use some form of radix-4 modified booth algorithm, which cuts the number of partial products to *n/2*. The reduction in the number of partial products yields significant savings in energy consumption, latency, as well as the total transistor count.

For a 53x53-bit multiplier in an FPM datapath, a radix-4 booth-encoded algorithm produces 27 partial products as shown in Figure 5.4. Each of the Y and X inputs is in a radix-4 format. The multiplier bits, X, are used to generate booth control signals for each partial product row. One of the big advantages of radix-4 booth multiplication is the relative simplicity of the logic which generates partial product rows. The only multiples of the multiplicand that are needed are: 0, $\pm Y$, and $\pm 2Y$. Partial product term Y is generated by simply assigning it the multiplicand. The $2Y$ multiple can be generated with relative ease by assigning it one bit right shifted value of the multiplicand. Bitwise inversion is used to generate complemented multiples. To reduce these 27 partial product rows to two partial product rows, a reduction tree comprising 7 stages of 3:2 counters/carry-save-adders (CSAs), is usually employed [71].

The energy consumption of the multiplier array is directly correlated to the number of partial product terms. With more partial product terms, more logic is needed first to produce those terms and then to sum and reduce those terms using a reduction tree. To further improve energy efficiency, one of the alternatives is to use a radix-8 Booth-encoded multiplier which reduces the number of partial product rows from 27 down to 18 as shown in Figure 5.5. The biggest

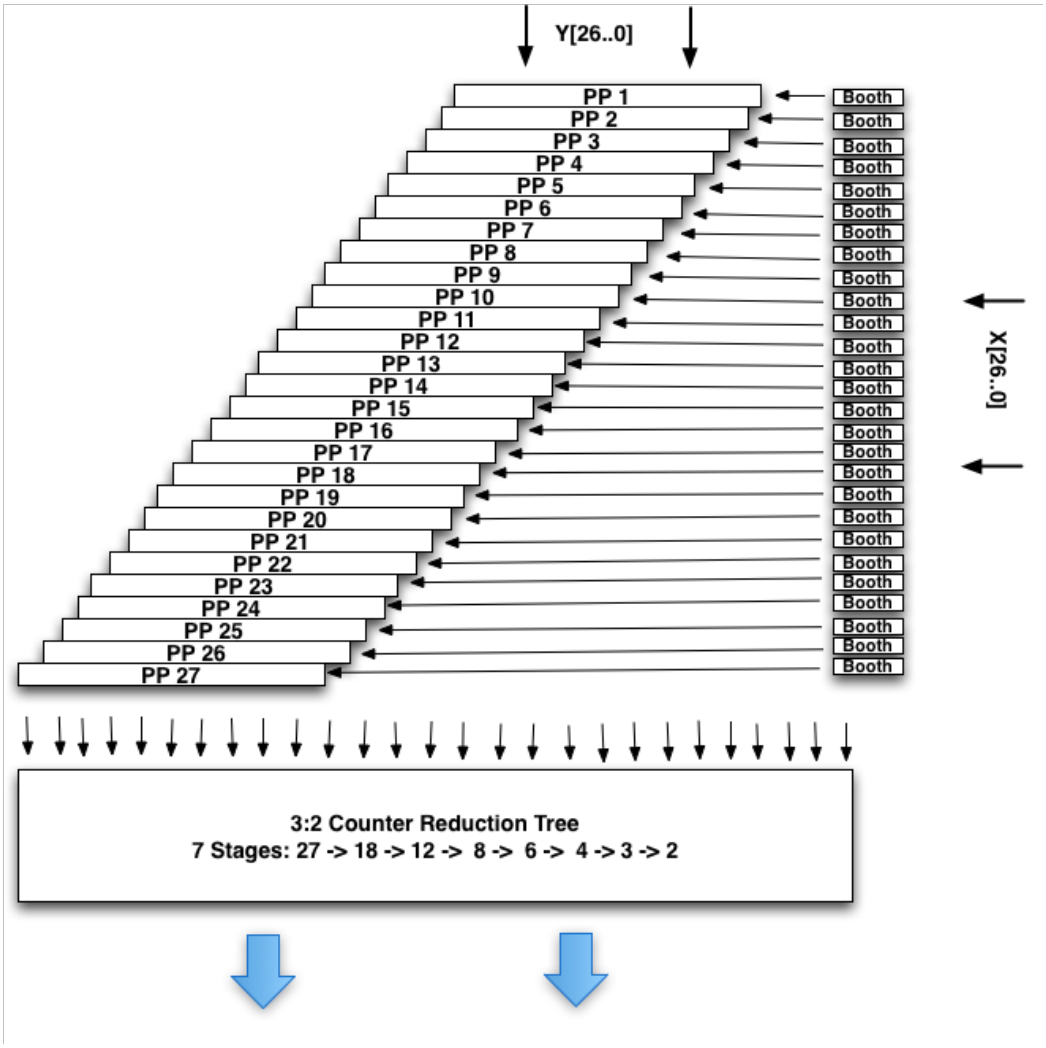

Figure 5.4: Radix-4 modified Booth multiplier.

disadvantage of a radix-8 multiplier is that it requires a $3Y$ multiple which needs a full length carry propagation adder to compute. Since the $3Y$ multiple must be available before any partial product term is computed, a tree adder topology such as a hybrid Kogge-Stone carry-select adder, described in Section 4.1, must be used to minimize any latency degradation in a synchronous design.

Table 5.1 compares three different radix length implementations of a 53x53-bit multiplication unit in terms of the total partial products bits and the number

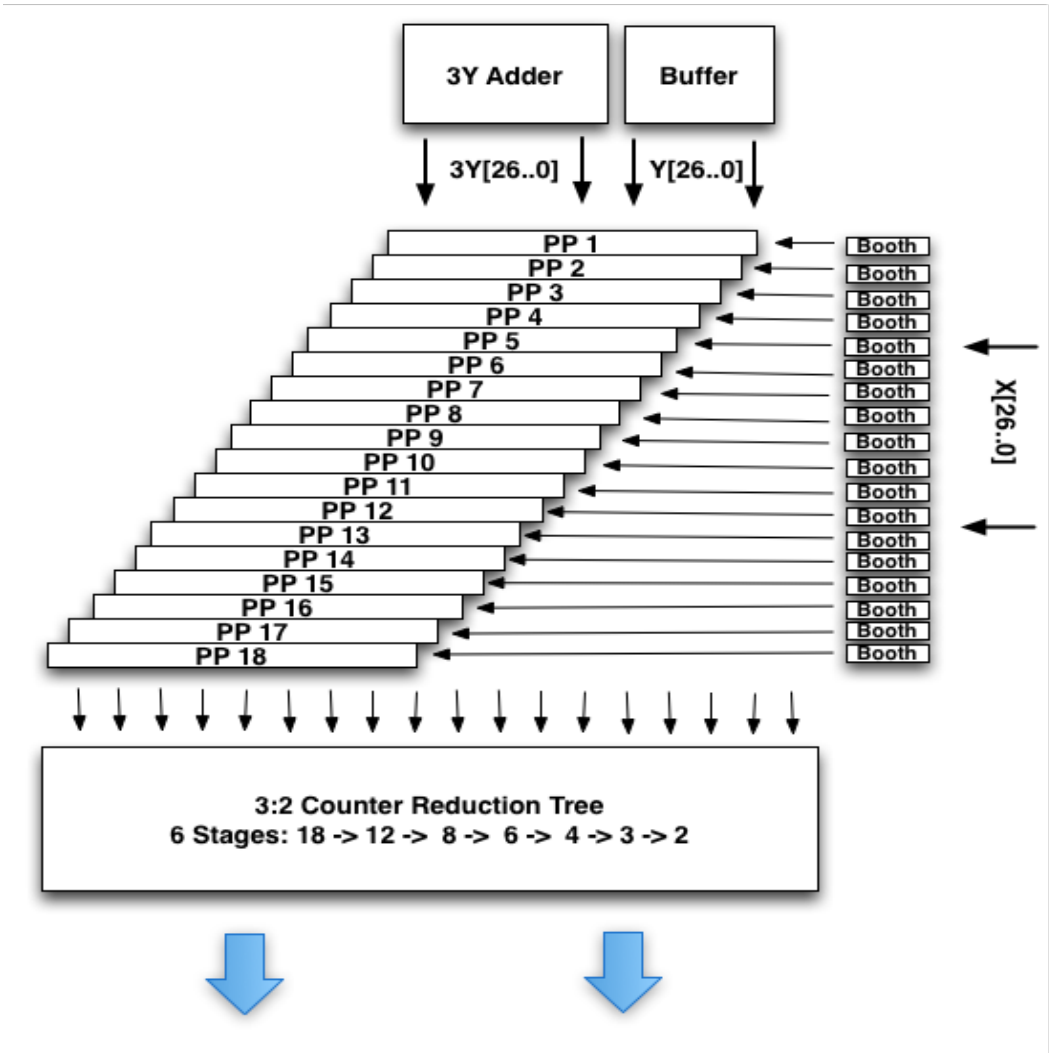

Figure 5.5: Radix-8 modified Booth multiplier.

of logic stages required to reduce the total number of partial product rows to two rows. A radix-8 Booth-encoded implementation produces 62.4% and 31.3% less partial products bits compared to bitwise radix-2 and Booth-encoded radix-4 multipliers respectively. But in terms of latency, when compared to a radix-8 version, a radix-4 implementation needs only one extra logic stage because partial product terms are summed and reduced using CSAs in a tree structure, which has logarithmic logic depth. This gives a radix-8 multiplier a single logic

stage cushion to compute the tough $3Y$ multiple. Hence, for any radix-8 Booth multiplier to be considered a viable alternative, it must provide a very low latency $3Y$ computation unit with energy consumption significantly lower than the savings attained with the use of 31.3% less partial product bits. The use of power intensive tree adders, discussed in detail in Section 4.2.1, greatly diminish the savings that result from the reduction in the number of partial product terms. As a result, radix-8 multipliers are not commonly used in synchronous FPM implementations

Table 5.1: Array Multiplier

53x53-bit Multiplier Type	**Partial Product Bits**	**Reduction Tree Logic Stages**
Radix-2 Bitwise	2809	9
Radix-4 Booth	1539	7
Radix-8 Booth	1056	6

5.4 53x53-Bit Radix-8 Array Multiplier

5.4.1 3Y Adder

The highly operand dependent nature of the $3Y$ multiple computation makes it a strong potential target for asynchronous circuit optimizations similar to those used earlier in our FPA design. The application profiling results in Figure 5.6 show that the longest carry chain in a radix-4 $3Y$ ripple-carry addition is limited to 3 ripple positions for over 90% of the operations across most floating-point application benchmarks. The delay of an adder depends on how fast the carry

reaches each bit position. For input patterns that yield such small carry chain lengths on average, we need not resort to an expensive tree adder topology designed for the worst-case input pattern of carry propagating through all bits.

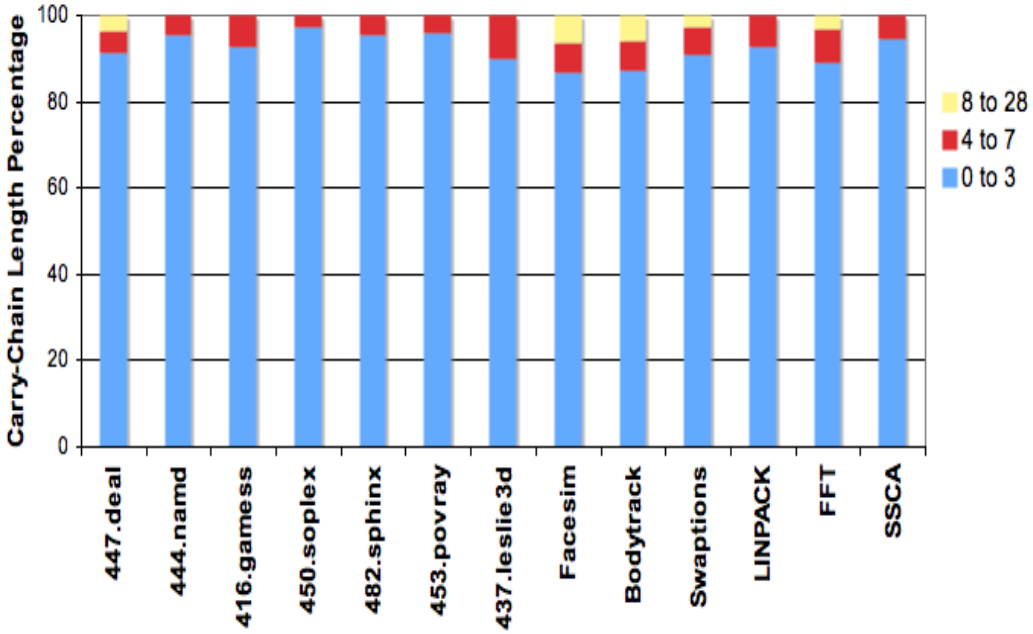

Figure 5.6: Radix-4 3Y Adder Longest Carry Length

The *interleaved* adder topology, first explained in Section 4.2.1, provides an energy efficient solution for computing the bottleneck *3Y* multiple term required in radix-8 Booth multiplication. It comprises two 53-bit radix-4 ripple-carry adders, where each *3Y* block shown in Figure 5.7 computes the *3Y* multiple for the corresponding *Y* input. The first arriving data tokens *YRs* are forwarded to the *right 3Y* adder. In a standard PCeHB reshuffling, the *interleave split* stage has to wait for the acknowledge signal from ripple-carry adder before it can enter neutral stage and accept new tokens. However, this would cause the pipeline to stall in case of a long carry chain. The *interleaved* adder topology circumvents this problem by instead issuing the next arriving data tokens to the *left 3Y* adder.

Hence, the two ripple-carry adders could be in operation at the same time on different input operands. The *interleave merge* stage receives outputs from both *right* and *left* adders and forwards them to the next stage in the same interleaved order. With our pipeline cycle time of approximately 18 logic transitions (gate delays), the next data tokens for the *right* adder are scheduled to arrive after 36 transitions of the first one. This gives ample time to quite rare inputs with very long carry-chains to ripple through as well without causing any throughput stalls.

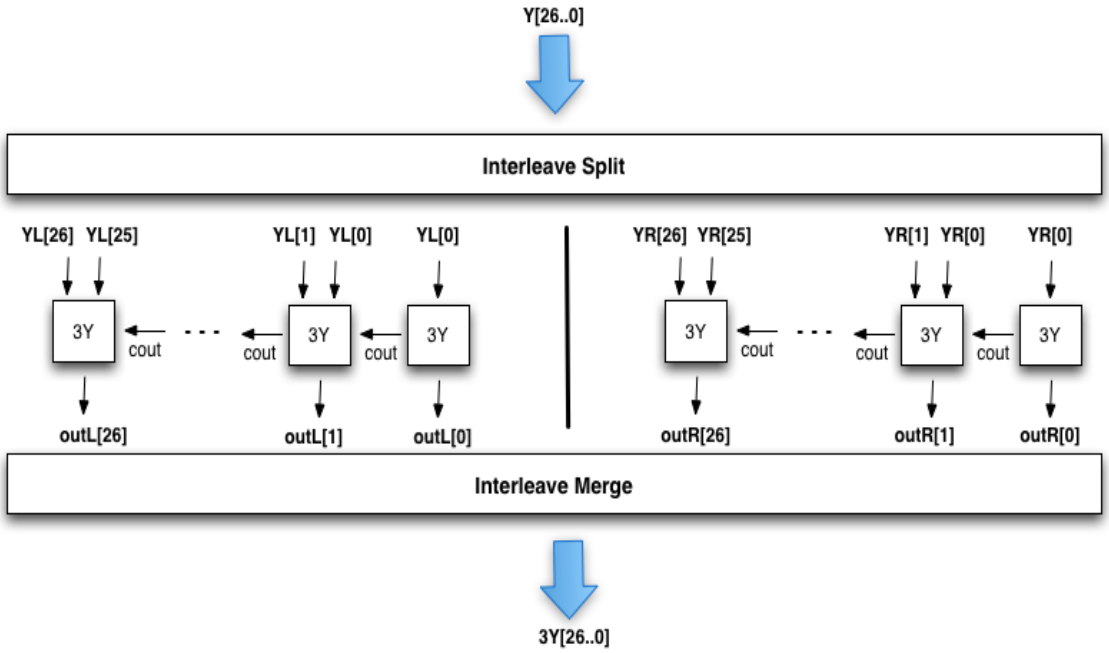

Figure 5.7: Interleaved 3Y Adder

For inputs patterns observed in our various floating-point application benchmarks, the forward latency of computing the *3Y* term using the *interleaved* adder is less than that attained with power-intensive tree adders, which are frequently used in synchronous designs to guarantee low latency compu-

tation. Compared to a 53-bit hybrid Kogge-Stone carry-select tree adder implementation, the *interleaved* adder consumes approximately 68.1% less energy at 8.3% lower latency for the average case input patterns shown in Figure 5.6. We exploit this data dependent adder design topology, not possible within a synchronous domain, to design an energy-efficient radix-8 Booth-encoded multiplier for our asynchronous FPM datapath.

5.4.2 Pipeline Design

Although, the radix-8 multiplier reduces the number of partial products bits by 31.3% compared to a radix-4 implementation, it still needs to produce and sum over 1050 partial product bits. As discussed earlier in Section 2.4, the standard PCeHB pipelines, though very robust, consume considerable power in handshake circuitry, which gets worse as the complexity of PCeHB templates increases with more input and output bits. We showed earlier that the handshake overhead, in a two-bit full adder PCeHB pipeline implementation, is as high as 69% of the total power consumption. Therefore, for circuits with large number of inputs, intermediate and final outputs, such a multiplier array, the PCeHB pipelines represent a non-optimum choice from energy efficiency perspective.

We use *N-Inverter* pipeline templates, first proposed in Section 3.2, to implement the multiplier array. An N-Inverter pipeline reduces the total handshake overhead by packing multiple stages of logic computation within a single pipeline block, in contrast to PCeHB template which contains only one effective logic computation per pipeline. The handshake complexity is amortized over a large number of computation stacks within the pipeline stage. In Section 3.3.2,

we showed that compared to a PCeHB pipelined implementation the N-Inverter pipelines can reduce the overall energy consumption by 52.6% while maintaining the same throughput. These improvements come at the cost of some timing assumptions and require the use of single-track handshake protocol. The design trade-offs associated with N-Inverter templates are discussed extensively in Section 3.3.

The block-level pipeline breakdown of our radix-8 multiplier array is depicted in Figure 5.8 . The granularity at which the array is split is critical from both performance and energy efficiency perspective. The N-Inverter templates allow us to pack considerable logic within each stage, which helps to reduce the handshake associated power consumption significantly. However, as the number of logic computations within a pipeline block increase, so do the number of outputs. With more outputs, although the number of transitions per pipeline cycle remain the same with the use of wide NOR completion detection logic, each of these transitions incur a higher latency as shown earlier in Section 3.3.2. The choice of *8x4* pipeline blocks, with 15 outputs per each stage, was made to provide a good balance of low power and high throughput. The pipeline block labeled *8x4 Sign* is identical to an *8x4* block except that it includes a sign bit for each partial product row. The sign bit acts as an input of one in the least significant position for any of the cases involving a complemented partial product multiple of *-Y*, *-2Y*, *-3Y*, or *-4Y*. The pipeline blocks labeled *10x4 Sign Ext* are similar in design to the frequent *8x4* block, except that it provides support for sign extension bits required for supporting complemented multiples. The *8x2* block is a reduced version of an *8x4* block with only two booth rows. The similarity between these different pipeline blocks and the frequent use of the *8x4* pipeline block provides us with great design modularity, which helped to

reduce the overall design effort required to optimize the multiplier array for throughput and energy efficiency.

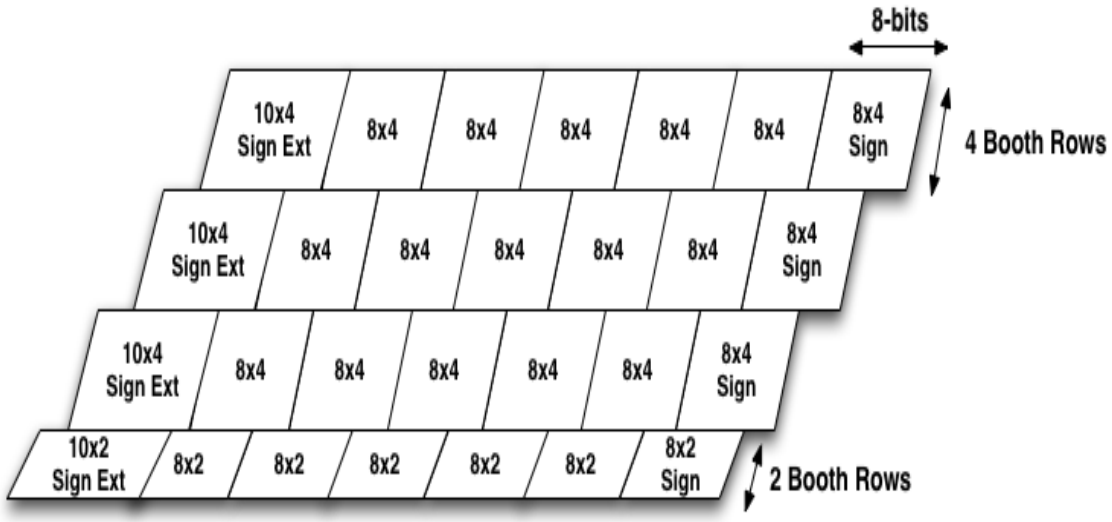

Figure 5.8: Radix-8 Multiplier Array

Due to the similarity between different pipeline blocks, we only present the details of the *8x4* block. Each *8x4* pipeline block receives Booth-control, Y and $3Y$ input tokens. The eight bits of Y and $3Y$ inputs are encoded as four 1-of-4 tokens each. Figure 5.9 shows the intermediate and final logic outputs within an *8x4* pipeline. It also shows the corresponding mapping of these outputs to a simplified circuit level depiction of an N-Inverter pipeline template. The NMOS stacks in the first stage compute four rows of eight bit partial product terms in inverted sense. These inverted outputs drive the inverters in the second stage of the pipeline block to produce corresponding partial product, *PP*, outputs. The next stage of NMOS stacks implements carry-save addition logic [71] to sum and reduce these four rows of partial products to two rows of inverted sum and carry outputs. These inverted outputs drive the PMOS transistors in the last

stage to produce sum and carry outputs, *SS* and *CC*, in correct sense for the following pipeline blocks.

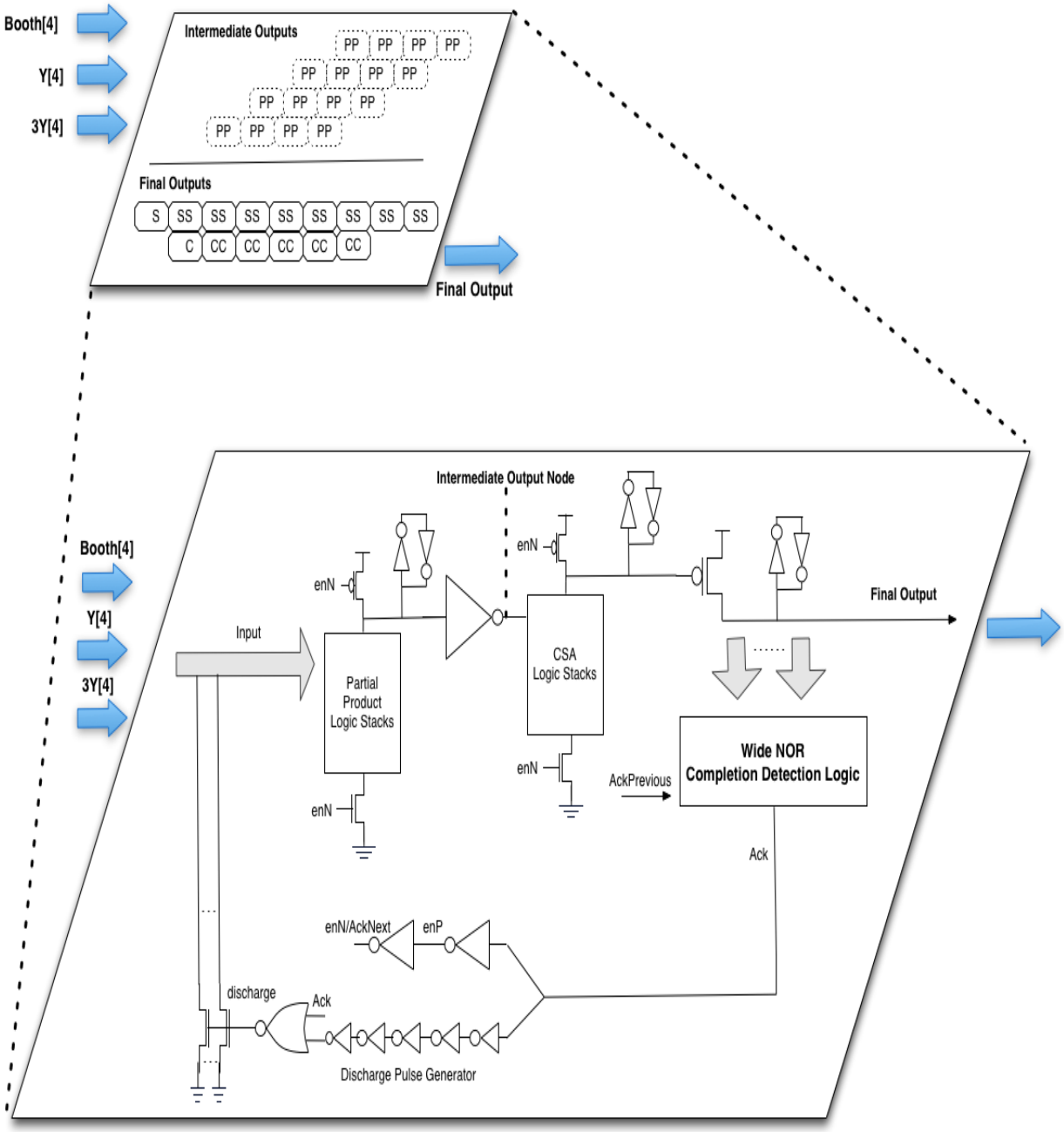

Figure 5.9: 8x4 Multiply Logic Block

For array multiplication, all pipeline blocks have to be in operation in parallel. The parallel operation requires multiples copies of input tokens to be consumed simultaneously by multiple pipeline blocks. For example, each booth control token is required in seven different pipeline blocks. To facilitate this, we include multiple *copy* stages prior to initiating the array computation. These copy blocks generate the desired number of copies for each input token. These tokens are then forwarded to the pipeline blocks which consume them to produce sum and carry outputs.

The next computation step is the summation of the large number of *SS* and *CC* outputs that are produced in parallel. This summation step is commonly referred to as *reduction tree* in arithmetic literature. A *reduction tree* basically employs 3:2 counters, often referred to as carry-save-adders (CSAs), to sum and reduce three inputs to two outputs at each stage of the tree. Within a few stages, the large number of tokens spanning over many partial product rows are reduced to mere two 106-bit long rows, which are finally summed using a carry-propagation adder. We implemented a full 3:2 counter reduction tree [71] using multiple N-Inverter pipeline blocks. The NMOS stacks within each block implement carry-save addition logic. In terms of logic density, each pipeline block was restricted to produce no more than 15 outputs to maintain cycle time similar to *8x4* pipeline blocks.

The N-Inverter templates use single-track handshake protocol. As a result, the input tokens are first converted from four-phase handshake protocol into single-track protocol using conversion templates. This adds an additional logic stage to the FPM datapath latency. Since the final carry-propagation adder uses four-phase handshake protocol, the output tokens from the *reduction tree* are

converted back to four-phase protocol. We hide the latency of this conversion stage by implementing the final stage of the *reduction tree* within these conversion templates.

The energy, latency, and throughput estimates of FPM implementations with radix-4 and radix-8 array multipliers are presented in Figure 5.10. The results are normalized to FPM datapath with a radix-4 multiplier. The 31.3% reduction in the number of partial product bits translates into 19.8% reduction in energy per operation. But this improvement in energy efficiency comes at a cost of 5.9% increase in the FPM latency because of the *3Y* partial product computation that needs to determined prior to initiating the multiplier array logic. A part of the *3Y* computation latency is masked within booth control token-generation and copy pipelines. Since the radix-4 multiplier requires one extra computation stage in the *reduction tree* compared to a radix-8 multiplier implementation, the latency overhead of the *3Y* computation can be further hidden. The 5.9% latency increase is attributed to the *3Y* multiple computation part which is not masked. Despite the increase in latency, the throughput for both implementations remains the same due to sufficient slack availability within the interleaved *3Y* computation block. The choice of a particular multiplier implementation represents a design trade-off. Since our goal was to optimize for energy consumption and throughput, we chose the radix-8 multiplier implementation in our final FPM design.

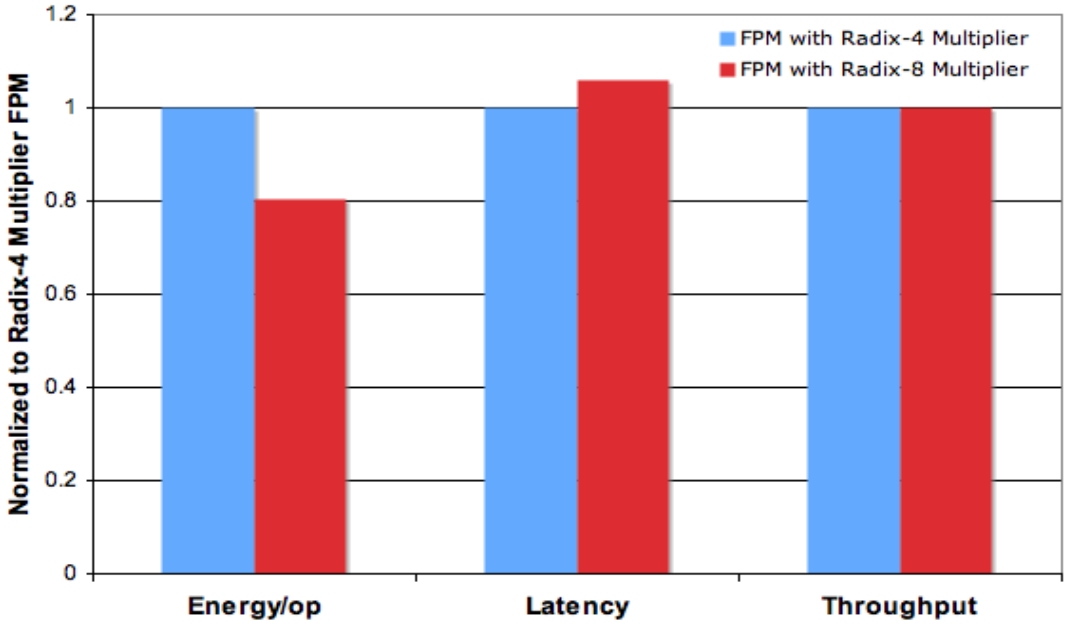

Figure 5.10: Radix-4 Multiplier vs. Radix-8 Multiplier

5.5 Sticky-bit Logic and Carry-Propagation Adder

The multiplier array outputs two rows of 106-bit long partial sum and carry terms. The next step is to compute the 53-bit mantissa of the FPM output. This requires the summation of the most significant 53-bits of the two incoming partial sum and carry terms using a carry-propagation adder (CPA). The least significant 53-bits of the partial sum and carry terms are needed to compute the *carry* input into the CPA as well as the *guard*, *round*, and *sticky* [49] terms required during the rounding step.

In advanced high performance synchronous FPM designs [73, 50], the multiplier array and the CPA blocks occupy back to back pipeline stages. Each of these units are designed, implemented, and optimized individually, in most

cases by separate group of engineers, with a design goal to meet the cycle time. The strict separation of these computation blocks in distinct pipeline stages means that all 106-bit sum and carry terms from the multiplier array become available together. This forces the high performance synchronous FPMs to use some kind of speculative tree adder [71] topology in their CPA design as the carry input is yet to be computed. The speculative computation makes the CPA design expensive in terms of energy and area as we showed earlier in Section 4.2.1. Towards the end of the pipeline stage, the actual carry input, computed using the least significant 53-bit partial sum and carry terms, becomes available and is used to select one of the two speculatively computed 53-bit mantissa outputs.

The strict separation of computation blocks in distinct pipeline stages in synchronous designs is done to minimize latching overhead. For example, splitting the multiple array at a finer granularity would result in many more latching gates due to the large number of intermediate partial product terms. Another reason for such coarser pipeline split is that in arithmetic datapaths, most computations usually operate on a full-width scale.

In contrast, asynchronous circuits provide much fine-grain pipelining, which can be exploited to greatly reduce the complexity of the carry computation and CPA logic in the FPM design.

5.5.1 Carry Computation and Sticky-bit Logic

The multiplier array requires relatively less number of summation steps to produce its least significant output bits. This is because there are less partial prod-

uct terms to be summed since each successive partial product row is skewed by three bit positions from the previous one in radix-8 multiplication, as seen in Figure 5.5 as well. As a result, the least significant bits are available relatively earlier than rest of the multiplier array outputs. We take advantage of our fine-grain pipelining by initiating the carry computation as soon as the least significant bits arrive. Furthermore, the application profiling results in Figure 5.11 show that for over 90% operations across all applications the longest ripple-carry length to compute the carry input term is less than four radix-4 bit positions. These average-case patterns indicate that the carry term could be computed well in time for the CPA operation, hence alleviating the need of any speculative CPA implementations.

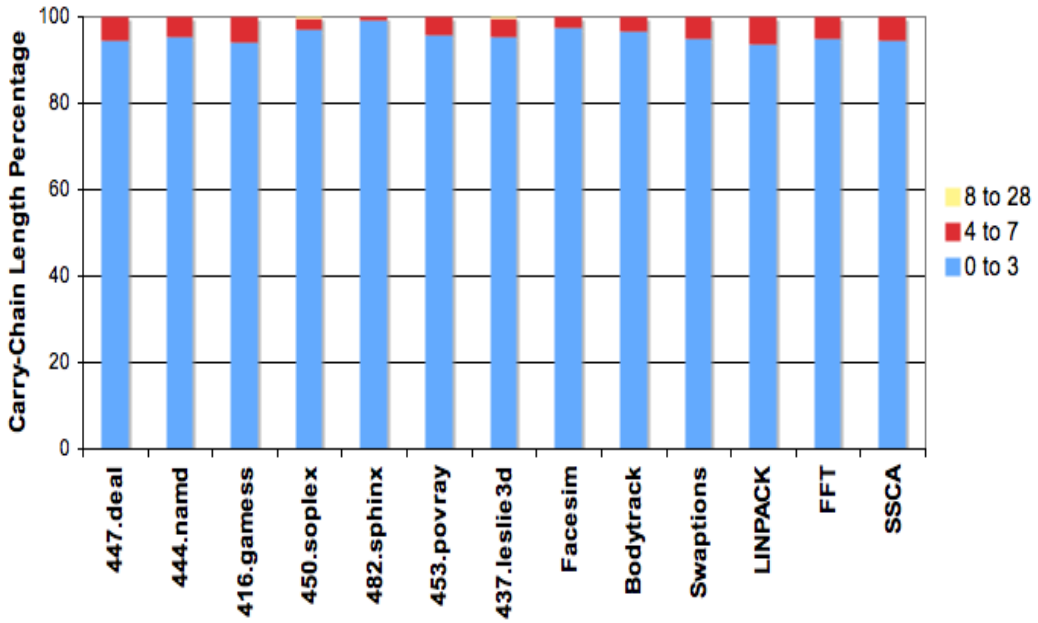

Figure 5.11: Longest ripple-carry length for computing CPA carry input

The micro-architecture of carry and sticky-bit computation is depicted in Figure 5.12. It uses *interleaved* split and merge pipelines, first introduced with the

design of *interleaved* adder in Section 4.2.1. The inputs *A* and *B* in Figure 5.12 are in one-of-four encoded format and correspond to 52 least significant bits of partial sum and carry output terms from the multiplier array. The odd data tokens are sent on the output channels labeled with *R* prefix, while the next arriving even data tokens are sent on channels with *L* prefix. Each *Carry Sticky* block computes the carry and sticky bit terms at that bit position. With carry chain lengths of less than four, as seen in Figure 5.11, the final carry term is computed within four logic levels on average. This represents logarithmic average latency. A synchronous design, constrained by worst-case carry propagation pattern, would need an expensive tree adder topology to attain similar latency. The *interleaved* topology prevents pipeline stalls in case of long carry chains by dispatching the next arriving inputs to the alternate computational unit. The odd tokens are used to compute the carry term *cinR* used as carry input in the odd ripple-carry adder of our *interleaved* CPA, whereas the next arriving even data tokens compute the carry term *cinL* used as carry input in the even ripple-carry adder of our *interleaved* CPA topology.

For sticky-bit computation, we use parallel tree topology which combines bitwise sticky-bit values to compute the final sticky-bit. A ripple flow architecture similar to the one used to compute carry input term was deemed not feasible as it yielded consistently long ripple chains as shown in Figure 5.13, which caused throughput degradation. Our *interleaved* topology prevents throughput degradation up to ripple lengths of *14* bit positions only. The sticky-bit ripple-flow implementation yields ripple lengths of *15* or more quite frequently. The sticky-bit is set to one if any of the bits is one, but for it to be set to zero it has to ensure that all prior bits in the sequence are zero. This is what causes the long ripple chains and renders ripple-flow design infeasible.

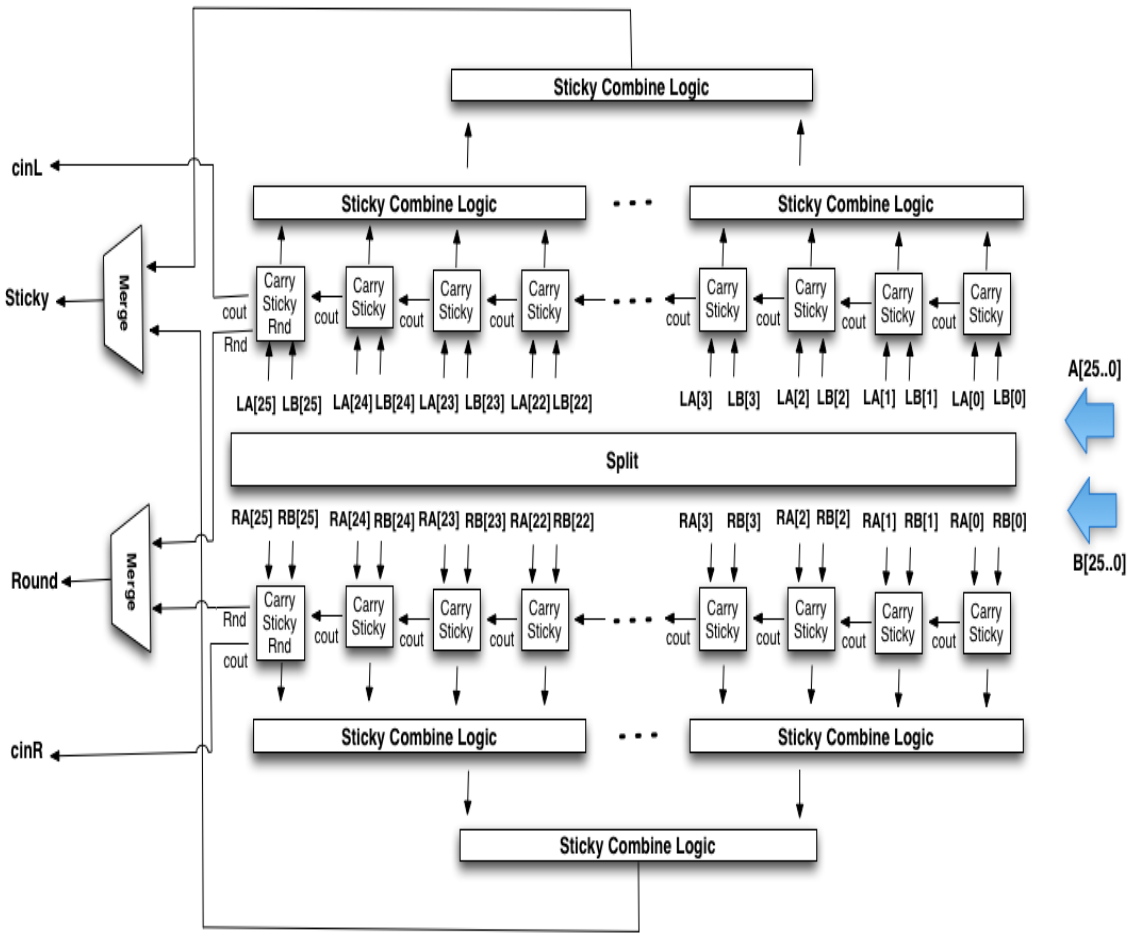

Figure 5.12: Interleaved topology to compute sticky-bit and carry input

5.5.2 53-bit Carry-Propagation Adder

The 53-bit carry-propagation adder is on the critical path of the FPM datapath. The delay of an N-bit adder primarily depends on how fast the carry reaches each bit position. The high throughput synchronous FPM designs, constrained by worst case computational latency, employ expensive tree adder designs to guarantee low fixed latency for all carry chain lengths. In contrast, asynchronous designs have no timing constraints. We harness this timing flexi-

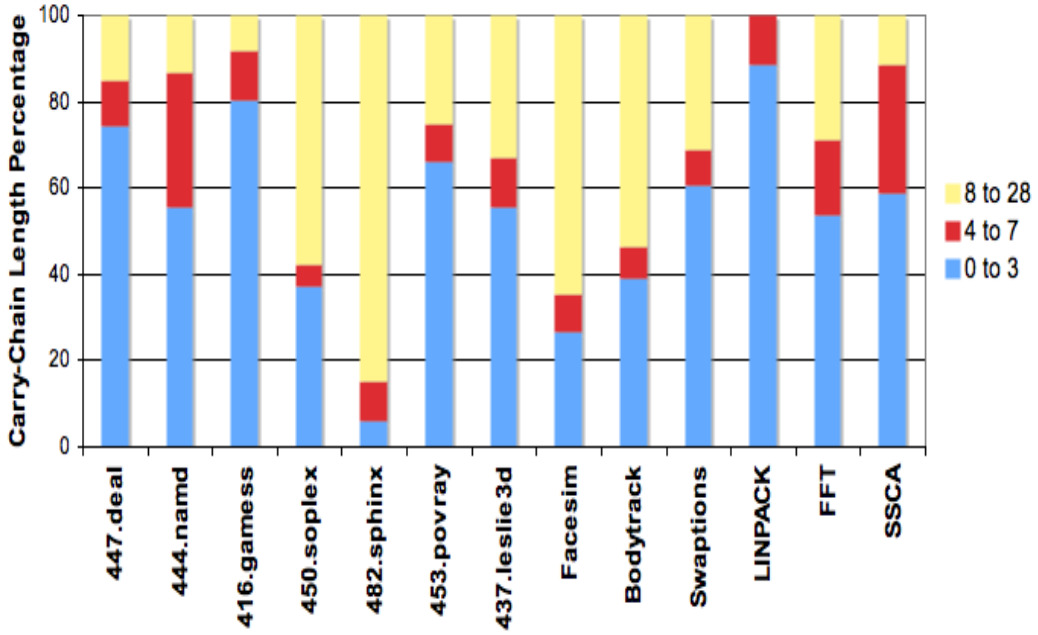

Figure 5.13: Sticky-bit ripple-carry chain length

bility of our underlying asynchronous circuits by using *interleaved* adder topology in our 53-bit carry-propagation addition. The *interleaved* adder comprises two ripple-carry adders, one each for odd and even data tokens respectively. The adder topology is identical to the one used earlier for significand addition in our FPA design. Our choice of the *interleaved* adder was made on the basis of application profiling results in Figure 5.14, which indicate very small carry chain lengths on average across all application benchmarks. For such carry chain lengths, the *interleaved* adder represents the most energy-efficient choice. It yields average throughput similar to that attained with expensive tree adder designs while consuming up to 4X less energy per operation. Section 4.2.1 discusses and quantifies the design trade-offs of *interleaved* adder topology in great depth.

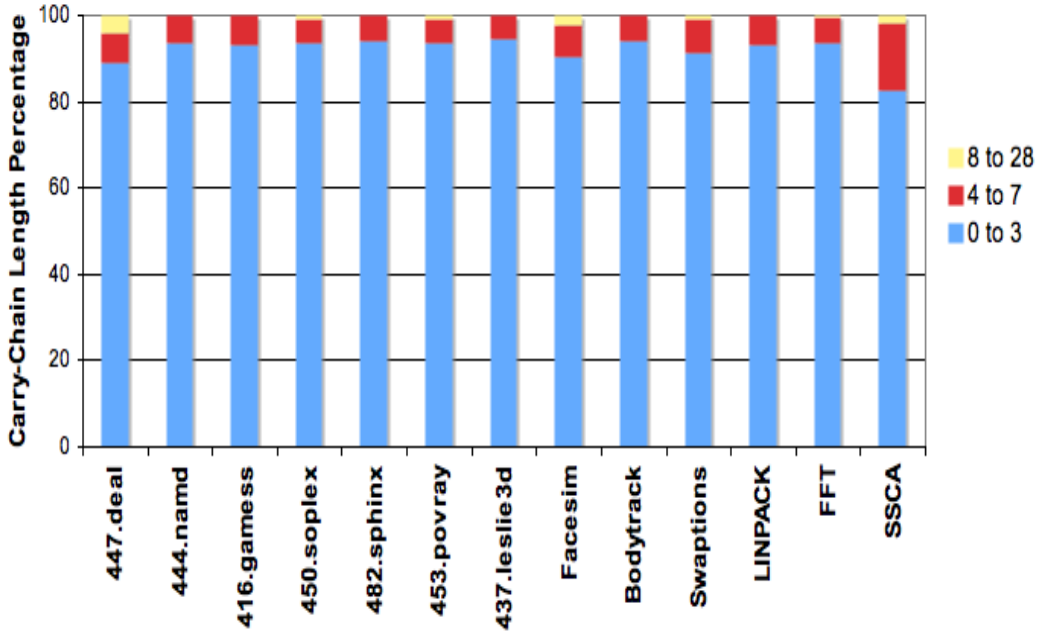

Figure 5.14: CPA ripple-carry chain length

5.6 Denormal, Underflow, and Zero-input Case

While discussing the various trade-offs involved in the FPM datapath design, we have so far ignored certain special cases specified in the IEEE format [49]. Two of these special cases: the denormal numbers and underflow case represent the most difficult operations to implement in an FPM datapath. The scenarios under which these two special cases arise and the tasks that need to be performed are summarized as follows:

- One of the FPM inputs is a denormal number, which yields a mantissa with zeroes in its most significant bit positions. If the non-bias exponent for the product is greater than *Emin* value of one, the product needs to be left shifted while decrementing the exponent until it is normalized or the

exponent reaches the *Emin* value of one. We refer to this scenario as the *Denormal* case.

- One of the FPM inputs is a denormal number or both FPM inputs are very small numbers and the resulting exponent is less than the *Emin* value of one. In this case, the mantissa needs to be right shifted. The value of right shift is equal to the difference between *Emin* and resulting exponent or an amount which zeroes out the mantissa, whichever of the two is smaller. We refer to this scenario as the *Underflow* case.

The need of variable left shift and right shift logic blocks makes the hardware support for denormal and underflow cases expensive. In the denormal case, the hardware also needs to analyze the full mantissa to figure out the exact left shift amount, which further adds to the FPM latency. In contrast, for all other regular inputs the FPM datapath requires no variable shift blocks and only needs to check the most significant mantissa bit for normalization.

The synchronous FPM implementations have a global clock constraint. The use of logarithmic variable shifters is one design option to support these special cases while staying within the clock bounds. The hardware complexity of logarithmic shifters makes this an infeasible option especially since these special cases do not happen frequently. Another solution that has been employed in commercial FPM design includes the utilization of the existing logarithmic shifters within the floating-point unit datapath [66, 58]. It requires a control mechanism to first stall the pipeline and squash all earlier instructions. This then enables the current special case instruction to use the variable logarithmic shifters in other parts of the floating point datapath. The flushing of earlier instructions and subsequent feedback of the current instruction results in a con-

siderable throughput penalty. It is also wasteful in terms of energy consumption since many already computed tasks of multiple earlier instructions in the pipeline are discarded and need to be computed all over again. The big advantage, however, is the hardware savings resulting from no additional logarithmic shifters.

One technique to prevent pipeline stall and throughput penalty is to provide a separate dedicated unit to handle these special cases. In this technique, employed in a few commercial chips [26], the intermediate results are forwarded to this special unit which includes a large expensive shifter unit. The solution, however, requires support for out-of-order execution, such as a reorder buffer, since many subsequent instructions in the FPM datapath may complete before the special case instructions.

The infrequent occurrence of these special case inputs and the extensive hardware complexity required to support these operations has meant that many FPM designs [73, 44] do not fully support these operations in hardware. Instead, these operations are implemented in software via traps. This yields very long execution time, which renders denormal numbers useless to programmers [59]. It also means that the FPM hardware is no longer fully IEEE compliant.

Asynchronous circuits are not constrained by any global clock. This timing flexibility allows us to use some very simple circuits, such as serial shifters, to provide full hardware support for these special case inputs. Figure 5.15 shows the FPM datapath with hardware support for special case instructions. Using conditional split pipelines, the output bits from the CPA are directed to either *Normal* or *Denormal/Underflow* logic path. The control to divert the output bits is computed using input type, exponent value, and the most significant mantissa

bits.

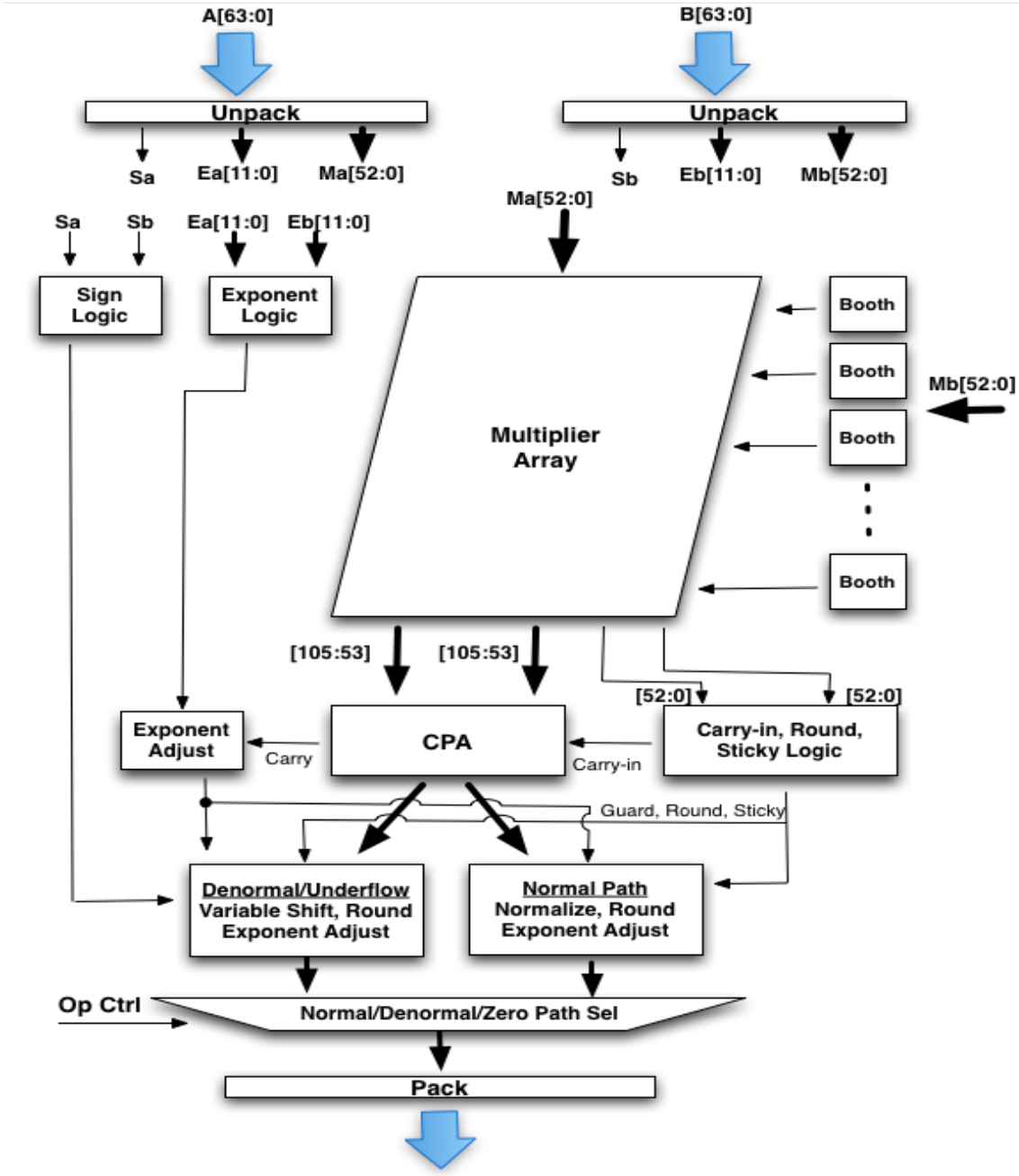

Figure 5.15: Floating-point multiplier with support for special case inputs

The *Normal* datapath includes a single-bit normalization shift block and rounding logic. The *Denormal/Underflow* unit comprises variable left and right shift blocks and a combined rounding block. As instructions do not have to be fed back to earlier pipeline stages to utilize variable shift blocks in other parts of the FPU, there is no need to flush any earlier instructions. The slack depth of the FPM datapath also minimizes potential pipeline stalls.

For input tokens diverted to the *Normal* datapath, no dynamic power is consumed within the *Denormal/Underflow* block and likewise for input tokens headed for *Denormal/Underflow* block, there is no dynamic power consumption in the *Normal* datapath. In contrast, synchronous design requires significant control overhead to attain fine-grain clock gating.

Prior to the final *Pack* pipeline, there is a merge pipeline stage, which selects the output from either the *Normal* or the *Denormal/Underflow* datapath using buffered control token, *Op Ctrl*, which is generated much earlier in the datapath.

5.6.1 Denormal Numbers

We provide hardware support for this special case by implementing a full mantissa length 1-of-4 encoded bitwise serial left shifter and an 1-of-4 encoded bitwise exponent decrement logic as shown in Figure 5.16. Each box labeled *Left Shift* and *Dec* represents a separate pipeline stage. The functionality of each pipeline stage is dictated by the centralized control unit via its output tokens, which travel serially through all pipeline stages. Each *control* token is encoded using a 1-of-3 data rail with each rail indicating one of the following three tasks:

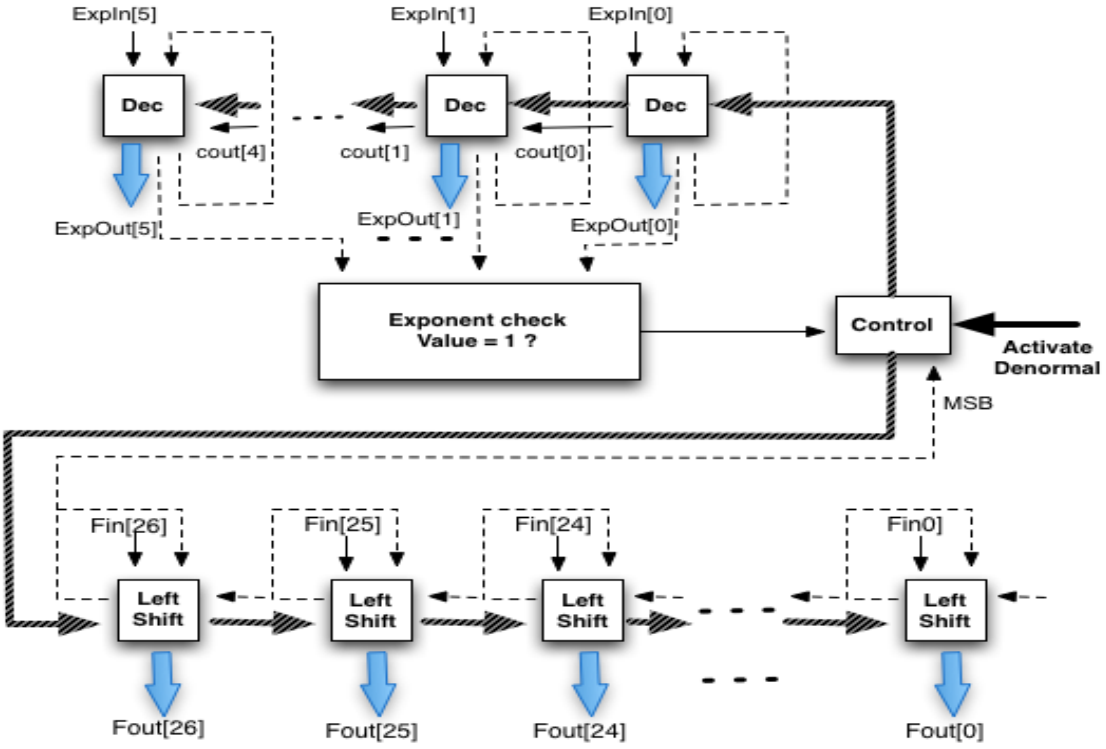

Figure 5.16: Denormal operation hardware

- *Init*: This is the first *control* token received by each block and is received only once per each operation. It directs the pipeline block to read the incoming token, *ExpIn* or *Fin*, and forwards its value on to the *feedback* channel to be fed back into the pipeline block in the next cycle. *ExpIn* input corresponds to exponent and *Fin* input corresponds to mantissa bits. The feedback loop for each pipeline stage is shown as a dotted line in Figure 5.16. The feedback path contains a full buffer stage for slack, which is not shown for simplicity.
- *Op*: This token is received as many times as the number of one-bit left shifts that are required. On its receipt, each *Left Shift* block uses its *feedback* channel inputs to produce a left shifted copy for the feedback loop. Sim-

ilarly, the *Dec* block reads its feedback channel inputs, decrements it, and forwards it to its feedback loop. It also sends a copy to the *Exponent Check* logic, which checks to see if the decremented exponent is equal to one.

- *End*: This is the last *control* token received by each block. It indicates the ends of the computation. The *feedback* channel input is forward to the final output channels, *expOut* and *Fout*.

On receiving the *Activate Denormal* token, the centralized control block activates the rest of the hardware by issuing an *Init* token for both left shift and decrement logic blocks. It then waits for the output of *Exponent Check* logic block and the most significant left shift block. If the most significant bit is not one and the exponent is greater than one, then it issues a new *Op* token. This step is repeated until the number is normalized or the exponent reaches the minimum value, at which point the control unit issues an *End* token. Although the left shift and decrement blocks are functionally serial, their bitwise pipelined implementation allows overlapped execution of multiple iterations. For example, the most significant bit left shift blocks may start producing the final outputs while the least significant blocks are still operating on the previous left shift iteration. As a result, the overall execution time per each denormal case is greatly reduced.

5.6.2 Underflow Output

Our FPM datapath handles the case of underflow outputs with bitwise serial right shift and decrement logic blocks as shown in Figure 5.17. Both of these units are encoded using 1-of-4 data rails. This reduces the number of serial right

shift pipeline blocks by half, which in turn reduces the latency of underflow operations. The input *count* into the decrement logic specifies the number of right shifts that are required to align the the mantissa such that the exponent is equal to *Emin*. In the case where the exponent value is off from *Emin* by more than the number of bits in the mantissa, the value of *count* is capped by a maximum shift amount that causes the mantissa to zero out.

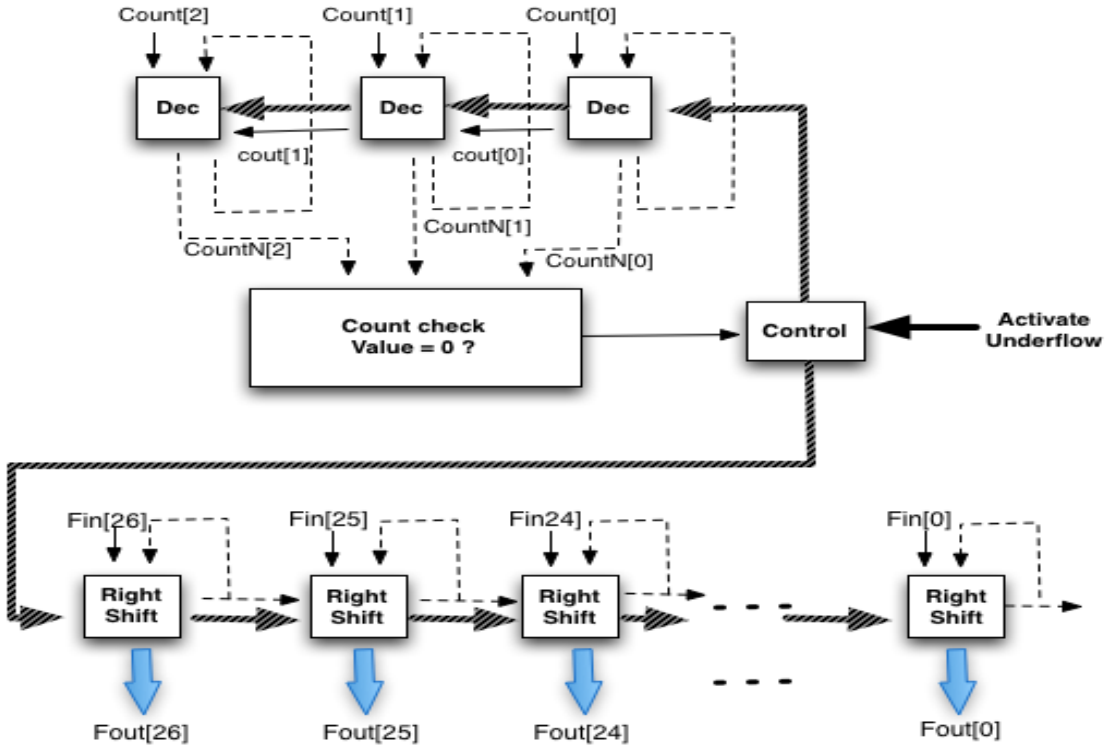

Figure 5.17: Supporting underflow case in hardware

In terms of the overall architecture, the underflow datapath is quite similar to the earlier explained denormal datapath. The central control unit activates the datapath by issuing an *Init* token. It then waits for the output of *Count Check* logic block before issuing the next *control* token, which is an *Op* token unless *count* has been decremented to zero, in which case an *End* token is issued.

5.6.3 Denormal/Underflow: Unified Rounding

Once the mantissa has been correctly aligned using variable left or right shift block, a subsequent rounding operation may be required to increment the 53-bit mantissa by one. We utilize ripple-carry 1-of-4 encoded increment logic to implement rounding. An expensive increment logic topology would have been futile since the output from variable shift blocks arrives in bitwise fashion. The rounding logic is shared between the *Denormal* and *Underflow* datapaths as shown in Figure 5.18 to further minimize the area overhead of supporting these special case operations. The *Rnd* block receives incoming *guard*, *round*, *sticky*, and *rounding mode* bits from both special case datapaths. It selects the correct set of inputs to determine whether to increment the mantissa or not. Similarly, the entire increment logic block is implemented using conditional merge pipeline blocks [36]. As in the case of *Rnd* block, a copy of the *activate* token used to initiate *Denormal* or *Underflow* datapath is used as control input in each 1-of-4 encoded increment pipeline block to select the correct input token from either of the two datapaths. The exponent selection logic is implemented in similar manner using conditional merge pipelines to enable the selection of input bits from the correct datapath.

5.6.4 Zero-input Operands

Operand profile of floating-point multiplication instructions reveals that a few application benchmarks have a significant proportion of zero input operands. These primarily include applications with sparse matrix manipulations, such as *447.deal* and *437.leslie3d* [5], despite their use of specialized sparse matrix

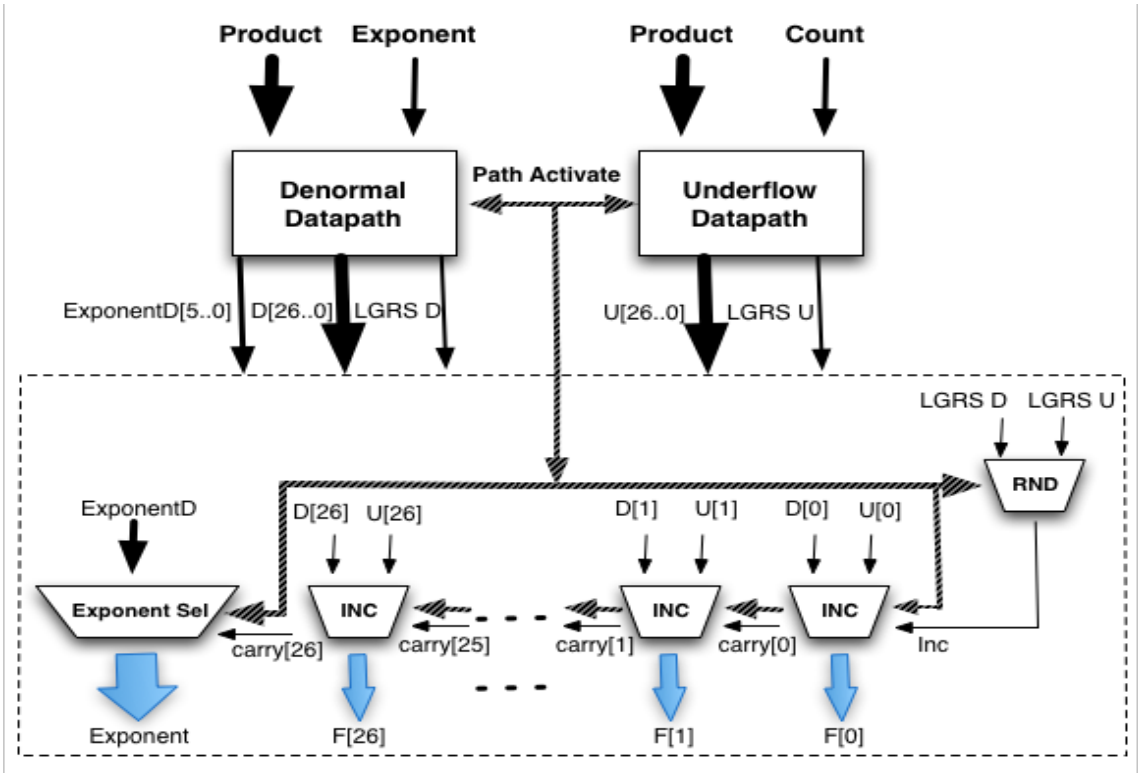

Figure 5.18: Unified rounding hardware for denormal/underflow cases

libraries. For other benchmarks, the zero-input percentage varies widely as shown in Figure 5.19. In most state-of-the-art synchronous FPM designs that we came across [54, 73, 69], the zero-input operands flow through the full FPM datapath. They yield similar latency and consume same power as any other non-zero operand computation. This is highly non optimum since if one or both of the FPM operands are zero, the final zero output could be produced much earlier and at much reduced energy consumption by skipping most of the compute intensive power consuming logic blocks such as the multiplier array, carry propagation adder, normalization, and rounding unit.

We provide a zero bypass path in the FPM datapath to optimize its latency

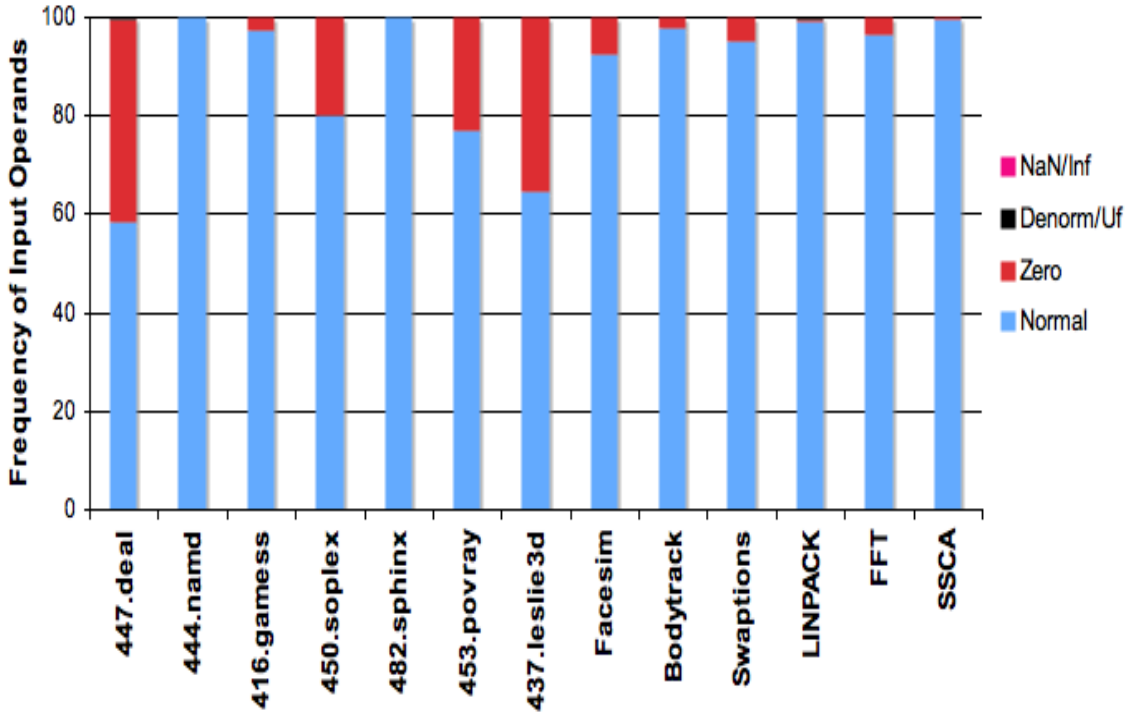

Figure 5.19: Operand profile of floating-point multiplication instructions

and energy consumption in the case of zero operands. To activate the bypass path, the FPM utilizes the zero flag control output from *Unpack* stage, which checks if any of the input operands is zero. But this information is not available in time before the start of pipeline stages pertaining to Booth control and *3Y* multiple generation. One possible solution was to delay these pipeline stages until the zero flag is computed and then use it to divert the tokens to either the regular or the bypass path. Since this solution incurs a latency hit for non-zero operands, it was discarded. In our design, instead of delaying the multiplier array, we inhibit the flow of tokens much deeper in the datapath. As a result, in our design the energy footprint of zero operand computations includes the overhead of computing Booth control token as well as some parts of the *3Y* multiple

computation. But this still yields roughly 82% reduction in energy consumption for each zero operand computation, while preserving same latency and throughput for non-zero operand operations.

The *Op Ctrl* signal used to distinguish between *Normal* and *Denormal/Underflow* block outputs is augmented to indicate zero input operations as well. Although, a zero input operation skips all logic intensive stages, the *Op Ctrl* token guarantees in-order operation as outputs are chosen in the same order their corresponding instructions were issued to the FPM. A big difference between the zero bypass paths of FPA and FPM is that unlike in the case of FPA zero bypass path, the bypass path in the FPM contains no data tokens as the output of the computation is zero irrespective of the value of the other operand (be it zero or non-zero). As a result, no buffer stages are needed on the bypass path despite the presence of huge slack disparity between regular and zero bypass paths.

5.7 Floating-Point Multiplier: Experimental Results

This section presents the SPICE simulation results of our proposed FPM datapath. Our design provides hardware support for denormal and underflow operations. Operations involving zero-input operands skip the multiplier array, carry-propagation adder, normalization, and rounding logic blocks. The transistors in the FPM were sized using standard transistor sizing techniques [71]. To meet high performance targets and to minimize charge sharing problems, each NMOS stack was restricted to a maximum of four transistors in series (including the enable). The slow and power-consuming state-holding completion-

elements were restricted to a maximum of three inputs at a time. Keepers and weak feedback inverters were added for each state-holding gate to ensure that charge would not drift even if the pipeline were stalled in an arbitrary state.

Since HSIM/HSPICE simulations do not account for wire capacitances, we included an additional wire load equivalent to a wire length of 8.75 μm in the SPICE file for every gate in the circuit. Our simulations use 65nm bulk CMOS process at 1V nominal V_{DD} and typical-typical (TT) process corner. Test vectors are injected into SPICE simulation using a combined VCS/HSIM simulation, with Verilog models that implement the asynchronous handshake in the test environment. All simulations were carried out at the highest-precision SPICE setting.

The denormal and underflow datapaths in our FPM design depict a truly operand dependent behavior. Figure 5.20 shows the FPM throughput at various different proportions of denormal or underflow cases. Another factor that greatly affects the throughput is the number of bit shifts required in each case. Each of the four lines on the graph corresponds to a different constant shift amount and yield different throughput results. The throughput degrades considerably at higher percentage of these special case operations, however, as seen earlier in Figure 5.19, these special cases happen very rarely, if at all. Even if they do occur frequently, our FPM still operates correctly and produces IEEE-compliant output albeit at lower throughput. At less than 0.1% frequency of these special case operations, our FPM throughput for all shift amounts is within 1% of the maximum FPM throughput attained with non-zero input operands. The synchronous FPMs that need to stall and flush the pipeline first to support these operations suffer a much greater throughput loss. When com-

pared to an operating system trap implementation, our throughput results are at least three orders of magnitude faster.

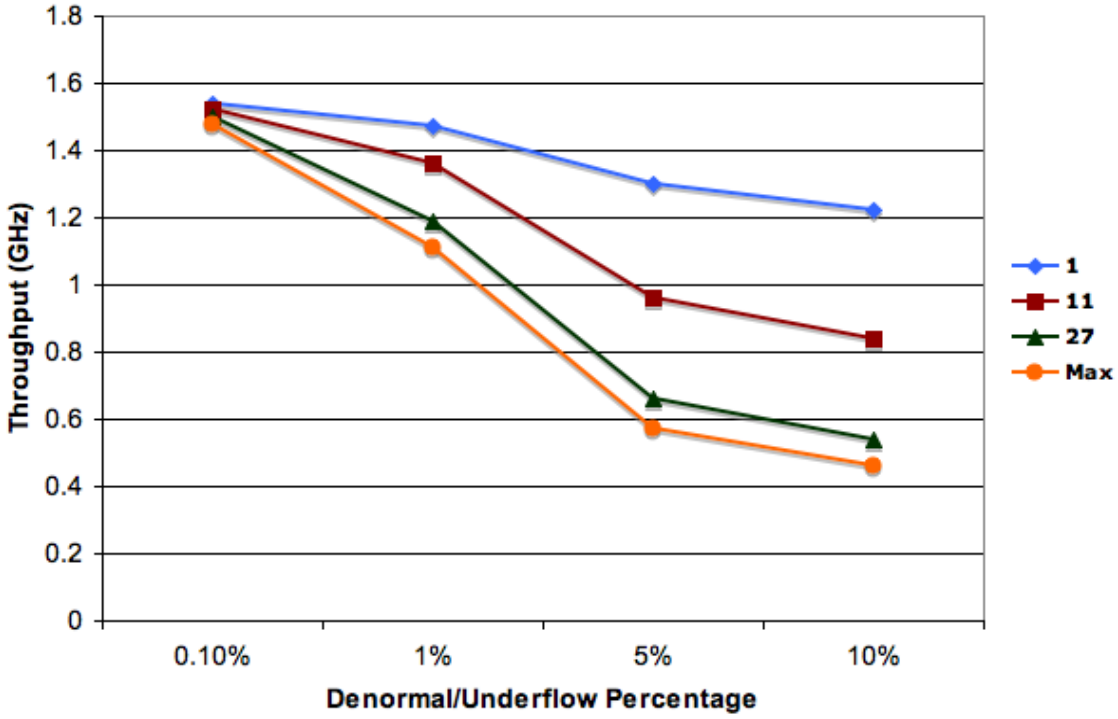

Figure 5.20: FPM throughput with varying proportion of denormal/underflow cases

In terms of total transistor area, our proposed denormal and underflow datapaths have a modest cost of only 8.5% of the total FPM transistor area. In contrast, a hardware solution comprising full mantissa length logarithmic right and left shift blocks and separate increment and rounding logic blocks would have cost approximately 21.2% of the total FPM transistor area.

The FPM throughput and energy per operation results across all application benchmarks are shown in Figure 5.21 and Figure 5.22 respectively. For non-zero operands, the FPM registers a highest throughput of 1.53 GHz. In applications

with a considerable percentage of zero operands, the average FPM throughput rises to as high as 1.78 GHz, since zero input operations skip throughput constraining N-Inverter templates in the multiplier array. This average case throughput property highlights another advantageous aspect of our asynchronous design which is not possible in a synchronous design, where the average throughput is limited by worst-case throughput.

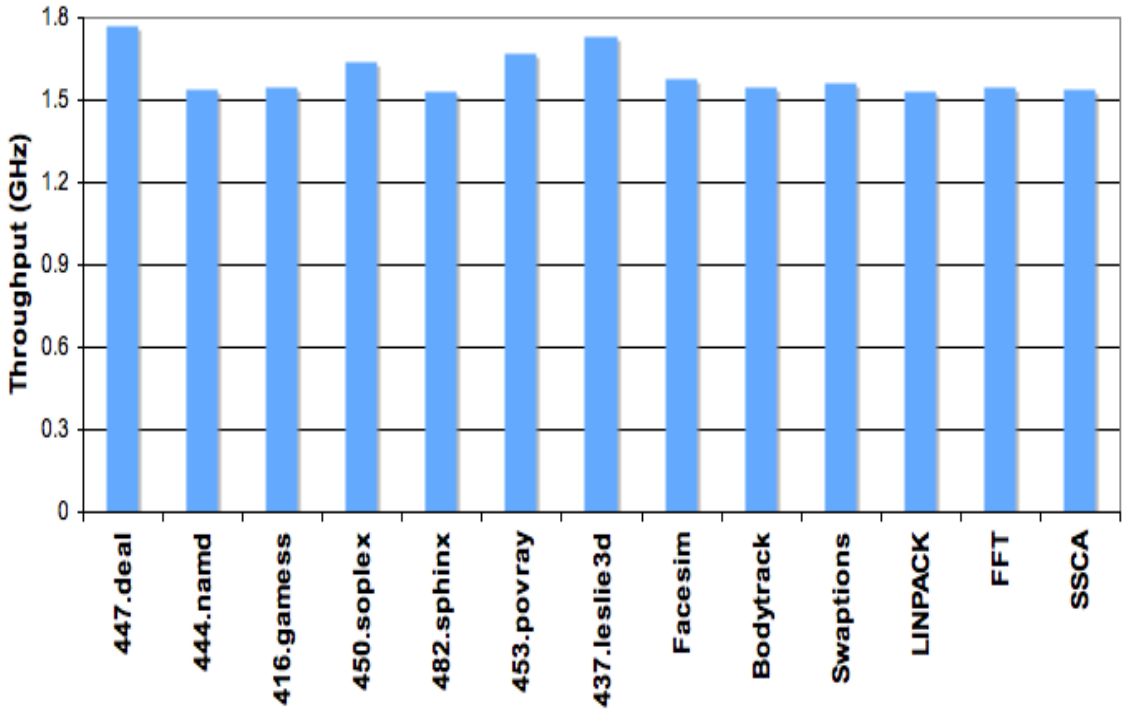

Figure 5.21: FPM throughput across various floating-point applications

In Table 5.2, we compare our proposed asynchronous FPM design against a custom FPM design by Quinnell et al [54] in 65nm SOI process at 1.3V nominal V_{DD}. The energy, throughput, and latency results include only non-zero operand operations in order to provide the worst-case comparison. Despite using 65 nm bulk process, our FPM design consumes 3X less energy per operation

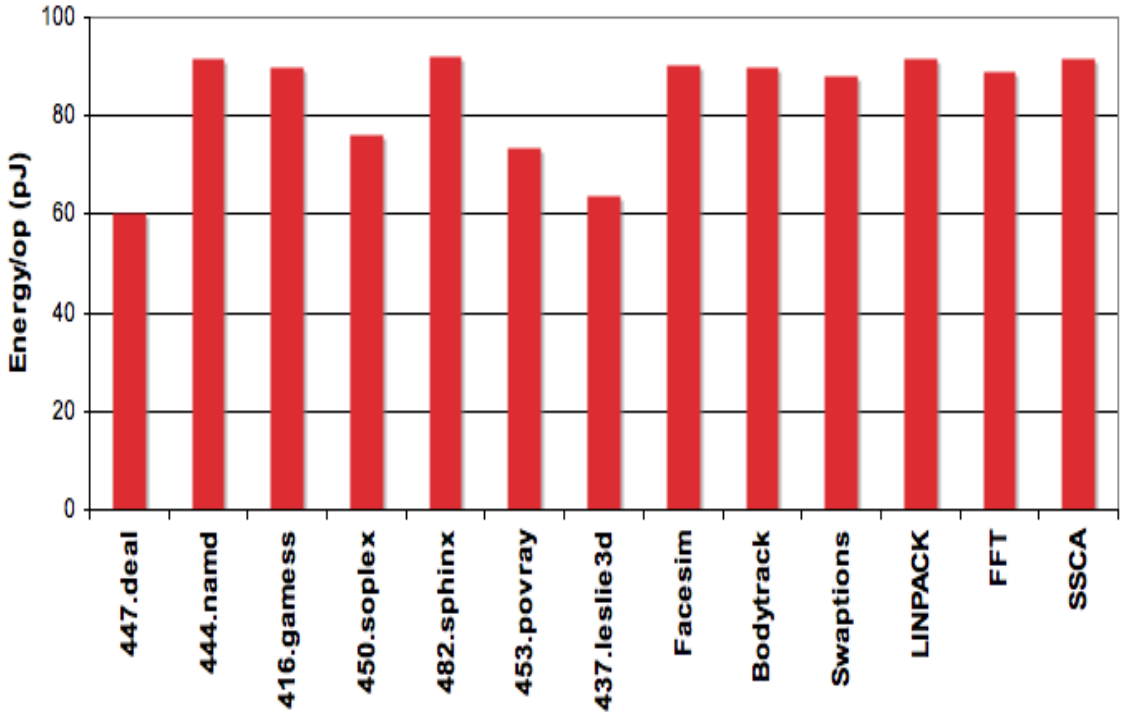

Figure 5.22: FPM energy per operation across various floating-point applications

while operating at 2.3X higher throughput. Both designs have similar latency at 1.3V. However, the custom FPM latency results do not include any internal pipeline latches, which account for a significant proportion of overall latency especially in high throughput designs. Our asynchronous FPM design compares quite favorably against the custom synchronous FPM implementation despite employing radix-8 Booth-encoded multiplier, which has an average 5.9% higher latency than a radix-4 Booth-encoded multiplier design.

For frequently occurring zero input operations in sparse matrix applications, our proposed FPM yields an even lower latency and energy per operation. The results for zero input operands are shown in Table 5.3, which highlights the

Table 5.2: Asynchronous FPM vs Synchronous FPM

Design	**Energy/op**	**Throughput**	**Latency @1V**	**Latency @1.3V**
Proposed FPM	92.1 pJ	1.53 GHz	1070 ps	705 ps
Quinnell FPM	280.8 pJ	666 MHz	NA	701 ps

efficacy of zero bypass path.

Table 5.3: Zero Operand Features

Design	**Energy/op**	**Latency**
Proposed FPM	15.8 pJ	464 ps @ 1V
Quinnel FPM	280.8 pJ	701 ps @ 1.3V

Since leakage power has become an important design constraint, our simulations model sub-threshold and gate leakage effects in detail. The total leakage power of our FPM in idle mode was estimated at 1.62 mW using typical-typical process corner at 90°C and a V_{DD} of 1V.

Chapter 6

Conclusion

Fast floating-point hardware is critical in a wide range of applications. Emerging applications in various disciplines of science, engineering, and finance are further pushing the demand for faster and faster performing floating-point hardware. Today, this performance is limited by power constraints. The traditional power reduction schemes, which relied primarily on technology and voltage scaling, are not sufficient any more. In this thesis, we circumvent the problem of power and energy inefficiency by taking a two-pronged approach. This includes an improved pipeline design and multiple operand-dependent optimization techniques.

Firstly, we discovered the inherent inefficiency of asynchronous QDI pipelines for operations involving a large number of tokens in flight, such as an array multiplier. QDI circuits, though very robust, incur a significant energy overhead in orchestrating handshake protocol between different parallel pipeline processes. To circumvent this problem, this thesis proposes two novel

energy-efficient templates for high throughput asynchronous pipelines. The proposed templates, called N-P and N-Inverter pipelines, use single-track handshake protocol. Noise and timing robustness constraints of our pipelined circuits are quantified across all process corners. A completion detection scheme based on wide NOR gates is presented, which results in significant latency and energy savings especially as the number of outputs increase. Compared to a standard QDI pipeline implementation of an 8x8-bit Booth-encoded array multiplier, the N-Inverter and N-P pipeline implementations of 8x8-bit Booth-encoded array multiplier reduced the energy-delay product by 38.5% and 44% respectively. The overall multiplier latency was reduced by 20.2% and 18.7%, while the total transistor width was reduced by 35.6% and 46% with N-Inverter and N-P pipeline templates respectively.

Furthermore, this thesis presents novel operand-dependent optimization techniques to improve the energy efficiency of IEEE-754 compliant floating-point adder (FPA) and floating-point multiplier designs (FPM). We begin with a baseline FPA that corresponds to a state-of-the-art high performance synchronous FPA design. We provide a detailed breakdown of the power consumption of the FPA datapath implemented using standard asynchronous QDI pipelines, and use it to motivate a number of different data-dependent optimizations for energy-efficiency. Some of these optimizations are highly challenging, if at all possible, in a synchronous design because they increase the worst case critical path but on average have negligible impact on performance.

Our baseline asynchronous FPA has a throughput of 2.15 GHz while consuming 69.3 pJ per operation in a 65nm bulk process. For the same set of nonzero operands, our optimizations improve the FPA's energy-efficiency to

30.2 pJ per operation while preserving average throughput, a 56.7% reduction in energy relative to the baseline design. To our knowledge, this is the first detailed design of a high-performance asynchronous double-precision floating-point adder.

This thesis provides a thorough analysis of the trade-offs involved in using radix-4 and radix-8 array multiplier designs. Both designs were implemented using the newly proposed energy-efficient N-Inverter templates. The radix-8 design was preferred since it further reduced the total FPM energy consumption by 19.8% while preserving the average throughput. These significant energy savings were made possible by the highly energy-efficient *interleaved* adder topology, which exploits the average-case short carry propagation chains to greatly minimize the adder hardware complexity. This optimization is not possible within synchronous design because of the worst-case cycle time constraint.

This thesis also presents a modest hardware implementation for denormal and underflow operations. With average-case input patterns, the proposed design's performance matches that of a much complex hardware design which uses 2.5X more transistors. Compared to many synchronous designs, which do not fully support these operations in hardware [73, 44], the proposed design has many orders of magnitude higher performance.

The full FPM datapath with numerous operand-dependent and pipeline optimizations is fully quantified using 65nm bulk process. When compared against a custom synchronous FPM design [54] in 65nm SOI process, it consumes 3X less energy per operation while operating at 2.3X higher throughput.

6.1 Future Work

In future, we plan to extend this work to develop asynchronous fused multiply-add architectures guided by similar principles to those outlined in this thesis. We intend to explore operand-dependent opportunities in floating-point division, square-root, and other functional units to design a complete energy-efficient floating-point unit. In this thesis, we restricted our novel low-handshake pipeline templates to the array multiplier design only. However, in future we intend to utilize these energy-efficient templates across the entire FPU datapath to further reduce the overall energy consumption.

An asynchronous FPU's application is not restricted to an asynchronous processor only. Since a floating-point unit is a self-contained system and has been employed as a co-processor in many commercial microprocessors, we envision asynchronous FPUs with operand-dependent optimizations similar to those introduced in this thesis to be used within synchronous microprocessors. As part of our future work, we plan to explore all design trade-offs involved in a hybrid synchronous and asynchronous microprocessor design with an operand-dependent FPU.

Bibliography

[1] Exascale computing study: Technology challenges.
www.science.energy.gov/ascr/Research/CS/DARPAexascale-hardware(2008).pdf.

[2] FFT: A fast fourier transform package.
www.ffte.jp.

[3] HPCS scalable synthetic compact applications.
www.highproductivity.org/SSCABmks.htm.

[4] LINPACK.
www.netlib.org/linpack.

[5] SPEC benchmark suite.
www.spec.org.

[6] World's fastest supercomputer.
www.fujitsu.com/global/about/tech/k.

[7] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. Lim. Reduced latency IEEE floating-point standard adder architectures. In *Proceedings of the International Symposium on Computer Arithmetic*, 1999.

[8] P. Beerel, A. Lines, and M. Davies. Logic synthesis of multi-level domino asynchronous pipelines, 2009. Fulcrum Microsystems, U.S. patent 7,584,449 B2.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th*

International Conference on Parallel Architectures and Compilation Techniques, 2008.

[10] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.

[11] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4), July-August 1999.

[12] J. D. Bruguera and T. Lang. Leading-one prediction with concurrent position correction. *IEEE Transactions on Computers*, 48(10), October 1999.

[13] A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. institute for advanced study, June 1946.

[14] F. C. Cheng. Practical design and performance evaluation of completion detection circuits. In *Proceedings of the International Conference on Computer Design*, 1998.

[15] B. S. Cherkauer and E. G. Friedman. A hybrid radix-4/radix-8 low power signed multiplier architecture. *IEEE Transactions on Circuits and Systems*, 44(8), August 1997.

[16] U. V. Cummings, A. M. Lines, and A. J. Martin. An asynchronous pipeline lattice-structure filter. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994.

[17] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labont, J. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proceedings of IEEE Conference on Supercomputing*, 2003.

[18] W. J. Dally and J. Poulton. *Digital Systems Engineering*. Cambridge University Press, Cambridge, UK, 1998.

[19] A. Efthymious, W. Suntiamorntut, J. Garside, and L. E. M. Brackenbury. An asynchronous, iterative implementation of the original booth multiplication algorithm. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2004.

[20] V. Ekanayake, C. Kelly IV, and R. Manohar. An ultra-low-power processor for sensor networks. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[21] M. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan-Kaufmann, 2004.

[22] D. Fang and R. Manohar. Non-uniform access asynchronous register files. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2004.

[23] D. Fang, J. Teifel, and R. Manohar. A high-performance asynchronous FPGA: Test results. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[24] M. Ferretti and P. Beerel. Single-track asynchronous pipeline templates using 1-of-n encoding. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2002.

[25] J. Garside. A CMOS VLSI implementation of an asynchronous ALU. In *Proceedings of the IFIP Workshop on Asynchronous Design Methodologies*, 1993.

[26] S. Gupta, R. Periman, T. Lynch, and B. McMinn. Normalizing pipelined floating point processing unit. u.s. patent no. 5,267,186, 1993.

[27] J. Hensley, A. Lastra, and M. Singh. A scalable counterflow-pipelined asynchronous radix-4 booth multiplier. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2005.

[28] M. Horowitz. Scaling, power and the future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design*, 2007.

[29] Z. Huang and M. D. Ercegovac. High-performance low-power left-to-right array multiplier design. *IEEE Transactions on Computers*, 54(3), March 2005.

[30] D. Kearny and N. W. Bergmann. Bundled data asynchronous multipliers with data dependent computation times. In *Proceedings of the Advanced Research in Asynchronous Circuits and Systems*, 1997.

[31] D. Kinniment. An evaluation of asynchronous addition. *IEEE Transactions on Very Large Scale Integrated Systems*, 4(1):137–140, March 1996.

[32] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22, August 1973.

[33] K. Krewell. Cell moves into the limelight. microprocessor report, February 2005.

[34] C. LaFrieda and R. Manohar. Reducing power consumption with relaxed quasi delay-insensitive circuits. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2009.

[35] T. Lang and J. D. Bruguera. Floating-point fused multiply-add: Reduced latency for floating-point additions. In *Proceedings of the International Symposium on Computer Arithmetic*, 2005.

[36] A. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[37] Y. Liu and S. Furber. The design of a low power asynchronous multiplier. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2004.

[38] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[39] R. Manohar. *The Impact of Asynchrony on Computer Architecture*. PhD thesis, Department of Computer Science, California Institute of Technology, June 1998.

[40] R. Manohar and J. A. Tierno. Asynchronous parallel prefix computation. *IEEE Transactions on Computers*, 47(11):1244–1252, November 1998.

[41] A. J. Martin. *Programming in VLSI: from communicating processes to delay insensitive circuits*. Addison-Wesley, 1990.

[42] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proceedings of Conference on Advanced Research in VLSI*, 1997.

[43] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G.

Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J. T. Tong, and A. Tura. The lutonium: A sub-nanojoule aynchronous 8051 microcontroller. In *Proceedings of the 9th IEEE International Symposium on Asynchronous Circuits and Systems*, May 2003.

[44] A. Naini, A. Dhablania, W. James, and D. D. Sarma. 1-ghz hal sparc65 dual floating point unit with RAS features. In *Proceedings of the International Symposium on Computer Arithmetic*, 2001.

[45] J. R. Noche and J. C. Araneta. An asynchronous IEEE floating-point arithmetic unit. *Proceedings of Science Diliman*, 19(2), 2007.

[46] S. Nowick, K. Y. Yun, P. A. Beerel, and A.E. Dooply. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems*, 1997.

[47] S. F. Oberman. Floating-point arithmetic unit including an efficient close data path. AMD, U.S. patent 6094668, 2000.

[48] S. F. Oberman, H. Al-Twaijry, and M. Flynn. The SNAP project: Design of floating point arithmetic units. In *Proceedings of the International Symposium on Computer Arithmetic*, 1997.

[49] The Institute of Electrical and Inc. Electronic Engineers. IEEE standard for binary floating-point arithmetic. ansi/ieee std 754, 1985.

[50] N. Ohkubo, M. Suzuki, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, and Y. Nakagome. A 4.4 ns CMOS 54 x 54-b multiplier using pass-transistor multiplexor. *IEEE Journal of Solid-State Circuits*, 30(3), March 1995.

[51] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman. Robust energy-efficient adder topologies. In *Proceedings of the International Symposium on Computer Arithmetic*, 2007.

[52] R. V. K. Pillai, D. Al-Khalili, and A. J. Al-Khalili. A low power approach to floating point adder design. In *Proceedings of the International Conference on Computer Design*, 1997.

[53] N. T. Quach and M. J. Flynn. An improved algorithm for high-speed floating-point addition. Technical Report CSL-TR-90-442, Computer Systems Lab., Stanford University, 1990.

[54] E. Quinnell, Jr E. E. Swartzlander, and C. Lemonds. Floating-point fused multiply-add architectures. In *Proceedings of the Fortieth Asilomar Conference on Signals, Systems, and Computers*, 2007.

[55] M. Schmookler, M. Putrino, A. Mather, J. Tyler, H. Nguyen, C. Roth, M. Pham, J. Lent, and M. Sharma. A low-power, high-speed implementation of a PowerPC microprocessor vector extension. In *Proceedings of the International Symposium on Computer Arithmetic*, 1999.

[56] S.W. Schuster and P.W. Cook. Low-power synchronous-to-asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. *IEEE Journal of Solid-State Circuits*, 38(4):622–630, April 2003.

[57] E. M. Schwarz, R. M. Averill, and L. J. Sigal. A radix-8 cmos s/390 multiplier. In *Proceedings of the International Symposium on Computer Arithmetic*, 1997.

[58] E. M. Schwarz, B. Giamei, C. Krygowski, M. Check, and J. Liptay. Method and system for executing denormalized numbers. u.s. patent no. 5,903,479, 1999.

[59] E. M. Schwarz, M. Schmookler, and S. D. Trong. FPU implementations with denormalized numbers. *IEEE Transactions on Computers*, 54(7), July 2005.

[60] P. Seidel. Multiple path IEEE floating-point fused multiply-add. In *Proceedings of the International Midwest Symposium on Circuits and Systems*, 2003.

[61] P. Seidel and Guy-Even. On the design of fast ieee floating-point adders. In *Proceedings of the International Symposium on Computer Arithmetic*, 2001.

[62] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[63] B. R. Sheikh and R. Manohar. An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2010.

[64] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, 2001.

[65] D. W. Sweeney. Analysis of floating-point addition. *IBM Systems Journal*, 4:31–42, 1965.

[66] M. P. Taborn, S. M. Burchfiel, and D. T. Matheny. Denormalization system and method of operation. u.s. patent no. 5,646,874, 1997.

[67] J. Teifel and R. Manohar. A high speed clockless serial link tranceiver. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 2003.

[68] Z. Tian, D. Yu, and Y. Qiu. A high effective algorithm of 32-bit multiply and MAC instructions' VLSI implementation with 32x8 multiplier-accumulator in DSP applications. In *Proceedings of the International Conference on Signal Processing*, 2002.

[69] S. D. Trong, M. Schmookler, E. M. Schwarz, and M. Kroener. P6 binary floating-point unit. In *Proceedings of the International Symposium on Computer Arithmetic*, 2007.

[70] K. van Berkel and A. Bink. Single-track handshake signalling with application to micropipelines and handshake circuits. In *Proceedings of the International Symposium on Asynchronous Circuits and Systems*, 1996.

[71] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2004.

[72] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Computer Systems Lab, Stanford University, 1991.

[73] R. K. Yu and G. B. Zyner. 167 mhz radix-4 floating point multiplier. In *Proceedings of the International Symposium on Computer Arithmetic*, 1995.