12-2015

ARITHMETIC LOGIC UNIT ARCHITECTURES WITH DYNAMICALLY DEFINED PRECISION

Getao Liang
University of Tennessee - Knoxville, gliang@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Getao Liang entitled "ARITHMETIC LOGIC UNIT ARCHITECTURES WITH DYNAMICALLY DEFINED PRECISION." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Jeremy Holleman, Qing Cao, Joshua Fu

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

ARITHMETIC LOGIC UNIT ARCHITECTURES WITH DYNAMICALLY DEFINED PRECISION

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Getao Liang

December 2015

DEDICATION

This dissertation is dedicated to my wife, Jiemei, and my family.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to everyone who offered support and encouragement during this long and meaningful journey. Without their generosity and patience, none of this would have been possible.

I would like to specially thank my advisor Dr. Gregory D. Peterson for his continuous support and insightful guidance toward completing my Ph.D degree. His valuable advice helped me overcome numerous obstacles over the years, and his inspiration and philosophy will always play an important part in my future life. I would also like to thank my committee member Dr. Jeremy Holleman for his encouragement and help with standard cell libraries and CAD tools. I also want to thank Dr. Qing Cao and Dr. Joshua S. Fu for serving on my doctoral committee and offering constructive advices that greatly improved my dissertation. In addition, I would like to express again my great appreciation to Dr. Donald W. Bouldin for his mentorship and support.

Special thanks to my friends and colleagues at the Tennessee Advanced Computing Laboratory (TACL) for their help, encouragement and support. Thank you very much, JunKyu Lee, Depeng Yang, Yeting Feng, Yu Du, Shuang Gao, Rui Ma, Gwanghee Son, David Jenkins and Rick Weber. I would also like to thank Song Yuan, Kevin Tham, Liang Zuo, Kai Zhu, Kun Zheng, and my long-time roommate Shiliang Deng for being supportive and buddies.

Heartfelt thanks to my beloved family for always being there to support me.

ABSTRACT

Modern central processing units (CPUs) employ arithmetic logic units (ALUs) that support statically defined precisions, often adhering to industry standards. Although CPU manufacturers highly optimize their ALUs, industry standard precisions embody accuracy and performance compromises for general purpose deployment. Hence, optimizing ALU precision holds great potential for improving speed and energy efficiency. Previous research on multiple precision ALUs focused on predefined, static precisions. Little previous work addressed ALU architectures with customized, dynamically defined precision. This dissertation presents approaches for developing dynamic precision ALU architectures for both fixed-point and floating-point to enable better performance, energy efficiency, and numeric accuracy. These new architectures enable dynamically defined precision, including support for vectorization. The new architectures also prevent performance and energy loss due to applying unnecessarily high precision on computations, which often happens with statically defined standard precisions. The new ALU architectures support different precisions through the use of configurable sub-blocks, with this dissertation including demonstration implementations for floating point adder, multiply, and fused multiply-add (FMA) circuits with 4-bit sub-blocks. For these circuits, the dynamic precision ALU speed is nearly the same as traditional ALU approaches, although the dynamic precision ALU is nearly twice as large.

TABLE OF CONTENTS

CHAPTER I INTRODUCTION 1

CHAPTER II LITERATURE REVIEW 8

A. Software Solutions for Arbitrary Precision 9

B. Statically Defined Multiple Precision Solutions 10

C. Multiple Precision Solutions for Fixed-point ALUs 11

D. Multiple Precision Solutions for Floating-point ALUs 13

E. Literature Overview 16

CHAPTER III FIXED-POINT ALU ARCHITECTURES 18

A. Dynamically Defined Precision 18

B. Dynamically Defined Precision Adder Architectures 20

1) Carry Extraction and Insertion 21

2) Carry Manipulation 22

C. Dynamically Defined Precision Multiplier Architectures 23

1) Dynamic Precision Partial Product Generator 24

2) Reduction Tree and Final Faster Adder 25

D. Dynamically Defined Precision Multiply-Accumulator Architectures 26

CHAPTER IV FLOATING-POINT ALU ARCHITECTURES 29

A. Data Organization 31

B. Floating-point Multiplier Architectures 34

1) Vectorized Unpacking and Packing 36

2) Vectorized Exception Handling 37

3) Vectorized Exponent Processing 38

4) Vectorized Significand Multiplication 40

5) Vectorized Significand Normalization and Rounding 49

C. Floating-point Adder Architectures 52

1) Overview of Dynamic Precision Adders 55

2) Vectorized Sign Processing 60

3) Vectorized Exponent Processing 62

4) Vectorized Significand Comparator and LZD 65

5) Vectorized Barrel Shifter 69

6) Vectorized Significand Adder 73
D. Floating-point Fused-Multiply-Add Architectures 75
1) Vectorized Radix Alignment for FMA Units 79
2) Vectorized Significand Addition for FMA Units 83
3) Vectorized LZD for FMA Units 85
CHAPTER V IMPLEMENTATION AND ANALYSIS 87
A. Fixed-point Adders 88
B. Fixed-point Multipliers 92
C. Eight-mode Floating-point ALUs 96
1) 128-bit Eight-mode Floating-point Multipliers 96
2) 128-bit Eight-mode Floating-point Adder 101
3) 128-bit Eight-mode Floating-point FMA Unit 105
CHAPTER VI CONCLUSIONS AND FUTURE WORK 110
REFERENCES 116
VITA 120

LIST OF TABLES

Table 4.1. Configurations of the eight-mode scheme……..30
Table 5.1. Latency for fixed-point adder implementations (FPGA)……89
Table 5.2. Area for fixed-point adder implementations (FPGA)…….89
Table 5.3. Normalized latency for fixed-point adder implementations (FPGA)…….89
Table 5.4. Normalized area for fixed-point adder implementations (FPGA)……..90
Table 5.5. Synthesis results for traditional and dynamic precision multipliers……...94
Table 5.6. Synthesis results for key modules of a bit-wise multiplier…….97
Table 5.7. Synthesis results for specific modules of a modified Booth-4 multiplier…...98
Table 5.8. Synthesis results for pipeline stages of both multipliers……...100
Table 5.9. Place-and-Route results for pipeline stages of both multipliers……...101
Table 5.10. Synthesis results for key modules of a dynamic precision adder……102
Table 5.11. Synthesis results for key modules of a traditional adder……..103
Table 5.12. Synthesis results for pipeline stages of adders……..104
Table 5.13. Place-and-Route results for pipeline stages of adders……..105
Table 5.14. Synthesis results for key modules of FMA units……..106
Table 5.15. Synthesis results for pipeline stages of FMA units……...108
Table 5.16. Place-and-Route results for pipeline stages of FMA units……...109

LIST OF FIGURES

Figure 2.1. Shared resource tree multiplication between 8-bit and 4-bit operators…….12
Figure 2.2. Simplified block diagram for dual-mode quadruple multiplier in [35]…….15
Figure 2.3. Previous work addressing precision for ALUs……..17
Figure 3.1. Adder units. (a) Traditional unit. (b) Multiple precision unit……...….19
Figure 3.2. Dynamic precision defining mechanism……...…20
Figure 3.3. Carry extraction and insertion circuit……22
Figure 3.4. Dynamic precision multiplier architecture……24
Figure 3.5. Dynamic precision bit-wise partial product generation…….25
Figure 3.6. Dynamic precision multiply-accumulator architecture…….27
Figure 4.1. Examples of external data arrangement……32
Figure 4.2. Internal data arrangement for the eight-mode scheme……..33
Figure 4.3. Block diagram for a dynamic precision floating-point multiplier…….35
Figure 4.4. Vectorized unpacker……..36
Figure 4.5. Examples of 5-level OR tree sharing for a 128-bit significand vector……..39
Figure 4.6. Dual mode 16-bit adder with a fixed carry-in of ‘1’…….40
Figure 4.7. Bit-wise significand multipliers. (a) Generic; (b) Vectorized……...42
Figure 4.8. Bit-wise vectorized partial product generator……...42
Figure 4.9. (a) 8-to-2 reduction tree; (b) Regular 4-to-2 implementation……...44
Figure 4.10. (a) Radix-4 multiplication; (b) Modified Booth-4 multiplication…….45
Figure 4.11. Significand multiplier with modified Booth-4 recoding……...46
Figure 4.12. Modified Booth-4 vectorized partial product generator……47
Figure 4.13. Special 4-to-2e block with carry elimination……49
Figure 4.14. Shift between sub-blocks for the first vectorized normalizer…..……...50
Figure 4.15. Vectorized normalizer for multipliers……...50
Figure 4.16. Block diagram for a vectorized rounding module…….51
Figure 4.17. Simplified block diagram for a floating-point adder…….53
Figure 4.18. Example of mapping between vector formats……...56
Figure 4.19. Block diagram for a dynamic precision floating-point adder……58
Figure 4.20. Vectorized effective operation detector…….61

Figure 4.21. Vectorized exponent difference module……63
Figure 4.22. Dual mode exponent increment module……64
Figure 4.23. Vectorized significand comparator……65
Figure 4.24. Block comparator module……66
Figure 4.25. A complete vectorized LZD module for adders……67
Figure 4.26. Sub-block LZD modules. (a) 4-bit LZD; (b) 3-bit LZD……68
Figure 4.27. Vectorized barrel shifter……70
Figure 4.28. Vectorized row shift. (a) Right shift; (b) Left shift……...71
Figure 4.29. Vectorized row shifter for significand sub-blocks……72
Figure 4.30. General implementation of a vectorized row shift for sticky bits……73
Figure 4.31. Dynamic precision adder for significand and grs vectors……74
Figure 4.32. Block diagram for a dynamic precision FMA unit……78
Figure 4.33. Radix point alignment process for FMA operations……79
Figure 4.34. Vectorized exponent difference module for FMA units……81
Figure 4.35. Examples of vectorized right-shift for FMA operations……...82
Figure 4.36. Vectorized row right-shifter for FMA operations……83
Figure 4.37. Vectorized significand adder for FMA units……84
Figure 4.38. Vectorized LZD for FMA units……85
Figure 5.1. Latency for ASIC implementation of fixed-point adders……..91
Figure 5.2. Area for ASIC implementation of fixed-point adders……...91
Figure 5.3. Latency and area for bit-wise PPGen implementations……93
Figure 5.4. Latency and area for generic reduction tree implementations……...93
Figure 5.5. Latency and area for modified Booth-4 PPGen implementations……95
Figure 5.6. Latency and area for dynamic precision reduction tree……95
Figure 5.7. Design layouts for both multipliers (Stage 1)……..101
Figure 5.8. Design layouts for both stages of the eight-mode adder……106
Figure 5.9. Design layouts for the three stages of an eight-mode FMA unit……108

CHAPTER I

INTRODUCTION

For the past decades, scientific computation has been a popular research tool among scientists and engineers from numerous areas, including climate modeling, computational chemistry, and bioinformatics [1 - 3]. With the maturing of application algorithms, developing high performance computing platforms that satisfy the increasing computational demands, search spaces, and data volume have become a huge challenge. As predicted by Moore's law [4], faster and faster computers are built to provide more processing power every year. Apparently, speeding-up the computation serially alone is no longer enough. In 1967 Amdahl presented a model to estimate the theoretical maximum improvement when multiple processing units were available [5]. Inspired by it and Gustafson's law [6], engineers started to explore thread parallelism by porting applications to multiple homogeneous nodes or multiple cores with the help of parallel programing interfaces such as MPI and OpenMP [42, 43]. High performance hardware accelerators were also introduced, combining with traditional processors, to construct heterogeneous computing platforms, such as the SRC-6 [7] and the Cray XK7 [8]. The reconfigurable accelerator of field programmable gate arrays (FPGAs) has "the potential to exploit coarse-grained functional parallelism as well as fine-grained instruction-level parallelism through direct hardware execution" [10, 46, 56 - 58]. Benefiting from the tightly-packed streaming multiprocessors (SMs) and improved programmability, graphics

processing units (GPUs) as accelerators offer impressive data-level parallelism and computational horsepower [11].

Increasing the processing speed and concurrency is not the only approach that is exploited to improve the performance of large scale scientific computations. Operational accuracy, or more precisely the hardware computational precision an arithmetic logic unit (ALU) can support, has also been continually improving with the development of computer architectures. Arithmetic logic units are the fundamental components within processors for performing operations. Ever since the binary system dominated the digital computer design, the operand bit-width of an ALU block was commonly used to represent its operational precision, for example 8-bit for the popular commercial Intel 8008 in the 1970s [12] and 256-bit for Intel's latest CPU Haswell microarchitecture [13]. The data format used to represent the numerical value of a particular bit stream has also been evolving. Various representation systems were introduced to represent signed integer numbers, including signed magnitude, biased, and complement systems [14]. Both the signed and unsigned integer representations can be extended to encode fractional data by employing an implicit or explicit radix point that denotes the scaling factor. These integer-based real number representations are generally called fixed-point systems, where the term "fixed-point" implies that the scaling factor is constant during the computation without externally reconfigured. As a result, the precision (least representable value of the unit in the last place, *ulp*) for a fixed-point number system cannot be improved without sacrificing its representable value range. Floating-point representation on the other hand provides not only the ability for wide dynamic range

support, but also high precision for small values. A typical floating-point number is represented in four components, and its value can be calculated with the equation $x = \pm s \times b^e$, where *s* is the significand and *e* is the biased exponent (with implicit *bias*). The standard precision of floating-point arithmetic units is also improving. In 2008, IEEE released a revision [15] for its popular IEEE 754-1985 standard for binary floating-point arithmetic, where the new quadruple precision format (binary128) was introduced to accommodate increasing precision requirements for high performance scientific computing. Recently, a new number format called *unum*, short for universal number, was introduced by Dr. Gustafson [48]. unums are described as a superset of all standard integer and floating-point formats, and it is aiming to provide solutions for some of the problems in current floating-point systems. By definition, "a unum is a bit string of variable length that has six sub-fields: Sign bit, exponent, fraction, uncertainty bit, exponent size, and fraction size [48]." Compared to standard floating-point formats, the variable size in a unum offers ability to change its representative range and precision, and the uncertainty bit (the ubit) indicates the exactness of the value represented. Thus, unums use fewer bits, obey algebraic laws, and do not require rounding, overflow, and underflow for proper operations.

Despite the trend of faster processing speed and higher operating precision for the past decades, computer processors to continue face barriers due to physical constraints. Theoretically, the processor dynamic power consumption increases linearly with increasing clock frequency. However, the total power consumption, including dynamic and static power, becomes dramatically larger when the silicon density and thermal

effects are taken into account [16]. Most advanced processes enable the tighter packing of more smaller devices on the same die, which makes it extremely difficult to dissipate the enormous heat generated from the leakage current and the fast switching activities, and the heat in return will increase the current required for the operation, resulting in escalating power consumption. Also, one cannot simply implement more transistors into a single chip by shrinking the size of a transistor to the physical limits of atomic structure [16]. The trend of building higher precision ALUs demands more transistors, implying a more power consumption and a worse critical delay. For example, doubling the bit-width of a fixed-point tree multiplier would worsen the critical delay by at least a factor of 2, and more importantly quadruple the number of transistors, which results in a huge power increase. As power becomes a serious constraint, especially for supercomputers, increasing the clock rate of CPUs or the ALU precision becomes expensive and no longer practically feasible for both the silicon industry and computing communities.

Therefore, exploiting parallelism and precision optimization becomes an alternative approach, which is independent of process technology, to improve the performance for computing architectures. Previous research results [17 - 20] have shown promising performance improvement by taking advantage of mixed precisions for iterative computing. Most current computing platforms employ general purpose processors that are powered by standard precision ALU hardware (e.g., IEEE single and double precision floating-point). While the standardization enables a more predictable numeric behavior and portability, it offers scientists with limited options when performing deep precision optimization for mixed precision algorithms. There are many software solutions that

provide arbitrary precision arithmetic support for different programming languages, such as GMP, ARPREC, MPFR, and MPACK [23 - 26]. The obvious limitation with software solutions is that their performance relies on the hardware ALUs to which their functions are mapped. Higher performance is achievable with non-standard custom precision ALUs, given that the prescribed solution accuracy is satisfied by manipulating the algorithm with least-sufficient-precision ALUs. Generally, a lower precision ALU requires fewer transistors to build, so multiple copies of smaller ALUs can be implemented in the same area as one higher precision ALU. With this scheme, not only the operational delay for each individual ALU is shortened, but also the number of operations per cycle is increased. Since power management and scheduling optimization can be applied to ALUs individually, energy efficiency can be also improved greatly.

Reconfigurable computing platforms provide the flexibility in constructing statically defined arbitrary precision ALUs on FPGAs, whose performance impact has been shown in some applications [20 - 21]. The impact of utilizing arbitrary precision (limited by the hardware resources) with FPGAs can be further enhanced by allowing the configuration of ALUs with the desired precision dynamically during the computation. Changing the ALU precision usually requires the download of a new design file that reflects the new precision onto FPGAs, resulting in highly degraded performance due to configuration delays. Although the technique of partial reconfiguration (PR) can reduce the re-programming time with smaller partial bit-streams [22], the performance improvement is still limited for designs with relatively small and frequently switching PR modules (ALU modules). The small PR modules make the cost of the PR control interface and

embedded/external processor even more expensive, frequent reconfiguration makes it harder to guarantee the solution accuracy, and the device-specific PR makes it difficult for porting the current design to other vendors, and impossible to other technologies.

It is desired to eliminate the need for reconfiguration, but at the same time offer the capability to perform arbitrary precision arithmetic operations according to the user's preference. Here, I propose to develop a dynamic precision ALU architecture to enable better computational performance, energy efficiency, and fault tolerance. The proposed new architectures do not only support dynamically defined precision with both fixed-point and floating-point system to allow precision-optimized computations, but also support vectorized operations in a SIMD fashion to maximize the shared hardware utilization. This new ALU paradigm prevents unnecessary loss of performance and energy due to applying unnecessarily high precision on computations (e.g., double precision when only single is needed). As the number of SIMD data paths can be dynamically adjusted based on the precision requirements (e.g., more concurrent vector operations for lower precision operands), high performance for both computation and energy efficiency can be effectively achieved by taking advantage of shared hardware utilization, precision optimization, and parallelism. With the flexibility of precision defining and hardware sharing, a better fault tolerance scheme can be implemented.

Chapter II discusses previous related work that addresses precision for ALUs. Both the software and hardware solutions are covered. The hardware designs are categorized as statically defined precision solutions, multiple precision fixed-point solutions, and multiple precision floating-point solutions. Chapter III provides a detail discussion on the

dynamic precision defining and the dynamic precision architecture for fixed-point ALUs, including different adders, multipliers and multiply-accumulators (MACs). Chapter IV is focused on the discussion of the dynamic precision floating-point ALU architectures, as well as the eight-mode scheme that is designed to better demonstrate how the architectures are implemented for floating-point adder, multiplier and fused-multiply-add (FMA) units. Chapter V describes the detail implementations of the proposed ALU architectures on FPGA and ASIC technologies. Performance analysis in terms of hardware and latency requirements is provided in this chapter with comparisons between the implementations. And finally, Chapter VI gives a conclusion on this dissertation and discusses potential future work.

CHAPTER II

LITERATURE REVIEW

In general, many efforts have been made to optimize arithmetic units for better performance from different perspectives. Being one of the most important aspects in computer arithmetic, several conventional and unconventional number representation systems were implemented serving different purposes [14]. For example, the decimal system is understandable for humans, while the binary system and its family (radix-2^n systems) are favorable for their simple implementation in digital computers; complement representations simplify the arithmetic for signed numbers; a redundant system is valuable as an intermediate format to eliminate carry propagation [47]. In this research, we will be focusing on approaches that employ the commonly used 2's complement system.

When an implementation platform is considered, solutions can be categorized as software and hardware solutions. For both platforms, ALUs can be divided into fixed-point and floating-point units based on the data format supported. From the functional point of view, there are adders/subtractors, multipliers, dividers, and others. From the perspective of precision support, implementations can be seen as statically defined precision, multi-mode precision, and arbitrary precision. Some designs vectorize a high precision processing core to provide simultaneous operations on lower precision sub-words, while others use multiple instances of lower precision units, or reuse the same unit

through a recursive iteration algorithm. In the rest of this section, related work will be reviewed and analyzed using these categories.

A. Software Solutions for Arbitrary Precision

Many software libraries were developed to facilitate arbitrary precision arithmetic for scientific computing. With the flexibility of software, it is possible for the libraries to implement a rich set of arithmetic logic functions using the provided hardware. The GNU Multiple Precision Arithmetic Library (GMP), a popular library for signed integers, rational numbers, and floating-point numbers, supports precision that is only limited by the host's available memory. GMP provides both C interface and C++ class based interface to all functions. Its main target is to speedup applications that require higher precision than basic C types by using sophisticated algorithms and optimized assembly code [23]. The MPFR library, as a supplement to the GMP, offers an efficient and well-defined semantics for floating-point computations with a rounding and exception scheme similar to IEEE 754 [24]. The MPACK library is specialized in providing high accuracy for arbitrary precision linear algebra with the help of multiple libraries, such as BLAS, LAPACK, and GMP [25]. The ARPREC package is another C++/Fortran-90 library that supports flexible computation of arbitrary precision [26]. Given that a higher precision operation is performed by these libraries with multiple routines running on the fixed hardware ALUs, extra instructions are required for coordination and control. On the other hand, since lower precision operations have to be mapped to a higher precision hardware unit, their performance is bounded (e.g., half being mapped to single). All the limitations suggest that a better hardware solution is the foundation for a better software solution.

B. Statically Defined Multiple Precision Solutions

With the flexibility of reconfigurable computing platforms, performance improvement can be achieved directly by implementing ALUs with the exact required precision. Wang and Leeser spent years to develop and refine a static variable-precision synthesizable library (VFloat), written in parameterized VHDL, which supports floating-point arithmetic units with standard or non-standard user-defined precision [27]. Besides most common arithmetic operators, such as adder, multiplier, and divider, this library also includes the format conversion between fixed-point and floating-point numbers, allowing a broader design space for precision optimization. As with all the other statically defined ALUs, once the units are implemented, any change of the precision requires a full time-consuming programming process.

In [28], authors introduced architectures for adders and multipliers which take advantage of the dynamic dual fixed-point format (DDFX) and runtime partial reconfiguration. The DDFX number system is just a simplified floating-point with a 1-bit exponent, where the binary point position (*p0* and *p1*) is determined by its MSB. The dual binary point configurations of a DDFX ALU can be changed with a smaller partial reconfiguration bit-stream, thus changing the dynamic range and precision but not the total bit-width. In spite of the reduced reconfiguration time, the partial reconfiguration cost is still high and impractical for rapidly changing precision requirements, along with other problems from partial reconfiguration.

Statically defined solutions allow arbitrary configuration (limited by hardware resources) of ALU precision before the hardware realization, but stay unchangeable afterwards until another time consuming full/partial reconfiguration.

C. Multiple Precision Solutions for Fixed-point ALUs

In order to eliminate the requirement of reconfiguration, it is much better to implement multiple precision ALUs with internal precision control mechanisms, which can adapt the precision dynamically. In 2004, Perri et al. proposed a FPGA-based variable precision fixed-point multiplier that performs one high precision multiplication (32x32), one mixed-precision multiplication (32x16), and two SIMD-like parallel multiplications on lower precision numbers (two 16x16, or four 8x8) [29]. This architecture employs the divide-and-conquer technique described in Eq. (2.1), where a high precision operation is done by organizing and processing the results from multiple copies of a lower precision unit. An optimized 32x9 multiplier is implemented for this purpose. Along with a control unit, adders, and shifters, this multiplier allows operations on signed, unsigned, and signed-unsigned operands. One disadvantage of using the $n \times m$ multiplier, instead of an $n \times n$ one, is the underutilization of HW resources for certain precision modes.

$$a_{[31:0]} \times x_{[31:0]} = 2^{24}[a_{[31:0]} \times x_{[31:24]}] + 2^{16}[a_{[31:0]} \times x_{[23:16]}] + 2^{8}[a_{[31:0]} \times x_{[15:8]}] + [a_{[31:0]} \times x_{[7:0]}] \qquad (2.1)$$

A twin precision technique for integer tree multiplication is presented in [30], where this technique allows the operations of one $n \times n$ or two parallel $\frac{n}{2} \times \frac{n}{2}$ to share the same HW resources. A simplified shared tree multiplier is illustrated in Fig. 2.1. In order to

support two modes of operations, the partial product arrays need to be manipulated according to the operational mode. When $\frac{n}{2} \times \frac{n}{2}$ multiplications are required, part of the partial product array has to be discarded to generate the correct products.

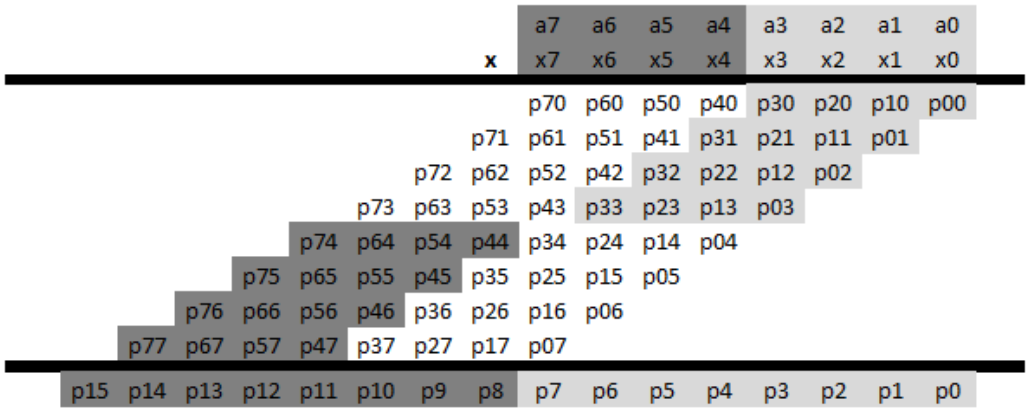

Figure 2.1. Shared resource tree multiplication between 8-bit and 4-bit operands.

In 2010, Kuang proposed a power-efficient 16x16 multiple precision multiplier [31]. Using the slightly different divide-and-conquer technique similar to [29], four small 8x8 modified Booth multipliers are employed for the generation of partial products for the higher and lower portions of the computation. The resulting partial products are re-organized and fed to a large reduction tree followed by a fast adder for the generation of the final results. The basic idea is illustrated in Eq. (2.2), where CV is the correction vector required for Booth multiplication. They also investigated the potential of saving power when single lower precision operation or operations with truncation were required. A dynamic range detector with supplement shutdown circuit was presented to handle such power-saving scenarios.

$$
\begin{aligned}
a \times x &= (2^{n/2} a_H + a_L) \times (2^{n/2} x_H + x_L) \\
&= 2^n a_H x_H + 2^{n/2} [a_H x_L + a_L x_H] + a_L x_L + CV
\end{aligned}
\tag{2.2}
$$

Danysh and Tan presented a 64-bit multiply-accumulator (MAC) architecture in [32], which performs vectorized operations on one 64-bit, two 32-bit, four 16-bit, or eight 8-bit unsigned/signed operands using shared segmentation. The underlying technique is similar to that in [30], but with more precision modes support and with the advantage of the modified booth multiplication to reduce total partial products generated, thus reducing the size of the reduction tree approximately in half. However, this comes with the cost of multiplexing in the partial product generator and carry handling in the following reduction tree and final adder.

The fixed-point multiple precision designs discussed above employ either divide-and-conquer or resource sharing techniques to support limited and uniform pre-defined options for fixed-point operations.

D. Multiple Precision Solutions for Floating-point ALUs

Floating-point representation provides better dynamic range support, thus is more useful for scientific computations. In 2008, Akkas presented a technique capable of modifying an IEEE adder architecture to a new dual-mode one that allows one operation of native precision or two parallel additions on half of the native precision (e.g., one double or two single) [33]. The author showed the detail designs for a 5-stage pipelined dual-mode double precision adder with improved single-path algorithm, and a 3-stage dual-mode quadruple precision adder with the two-path algorithm. Both designs support

only normalized numbers. The implementation (0.11 um CMOS) area and latency overhead of the dual-mode double precision adder is around 26% and 10%, while those for the dual-mode quadruple adder is 13% and 18%.

Even et al. proposed a dual precision IEEE floating-point multiplier that can compute one single-precision result in 2 clock cycles or one double-precision product in 3 cycles, supporting all IEEE-compliant rounding modes [34]. The half-size multiplication array (e.g., 27×53) is used in the first clock cycle for single precision, or the first two cycles for double precision, with the following cycle allocated for the final addition and rounding/normalization. Therefore, there will be one stall cycle after a double precision operation.

In [35], the authors presented two architectures, a dual-mode quadruple and a dual-mode double multiplier, for enabling dual-mode floating-point multiplications with the divide-and-conquer technique commonly used in many recursive architectures, as in [29], [31], and [34]. As illustrated in Fig. 2.2, by reusing the two low precision multiplier cores (57×57 for quadruple and 27×27 for double) in two consequent clock cycles, a high precision operation can be performed using only roughly half size of a fully parallel high precision multiplier. One of the cores has two more rows in its reduction tree for accommodating the partial products in carry-save format from the first cycle if high precision operation is required. One extra cycle is required to perform the final addition and rounding process, resulting in a two-cycle latency for low precision operations, or a three-cycle latency for high precision ones. This multi-cycle design helps saving expensive hardware resources and power consumption.

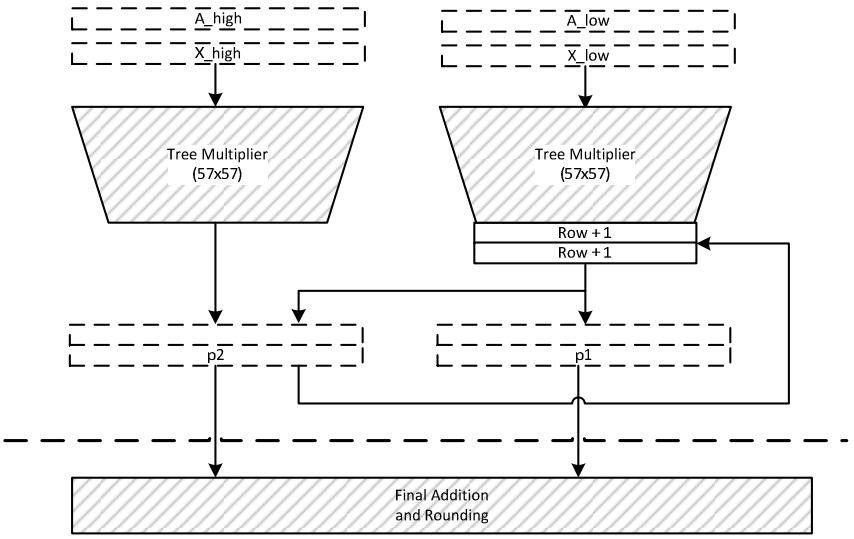

Figure 2.2. Simplified block diagram for dual-mode quadruple multiplier in [35].

Tan et al. also proposed their iterative floating-point multiplier that allows operation on three different precisions with a different number of stall cycles and latency [36]. There is a 2-cycle latency and 0 stall cycle before subsequent operations for one/two (packed) parallel single precision operations, a 4-cycle latency and 1 stall cycle for double, or a 5-cycle latency and 2 stall cycles. The architecture is similar to that in [35], but with a rectangular multiplier ($m \times n$) as the processing core instead of two half ones ($\frac{n}{2} \times \frac{n}{2}$). Single precision multiplications can be compute directly, while the other two require iterations of recursive multiplications.

Huang et al. presented a 128-bit floating-point fused multiply-add (FMA) unit that supports one 128-bit (quadruple) operation, or SIMD operation on two 64-bit (double) operands or four 32-bit (single) ones [37]. The core processing units is two special designed 57×57 multipliers, which can be used for 64-bit operations directly. On the

other hand, in this design, an iterative recursive algorithm is employed for performing 128-bit operations similar to that in [35], while vectorization based on shared hardware is used for 32-bit operations as in [30].

Isseven and Akkas presented a design for a dual-mode precision floating-point divider that based on radix-4 SRT algorithm, supporting one quadruple or two parallel double precision operations [38].

E. Literature Overview

For a clear overview and better understanding, Fig. 2.3 shows all the literature reviewed in this chapter as a summary, organized by their corresponding solution platforms, functionality, targeted technology, and other properties.

It is obvious that the fixed-point multiple precision solutions suffer from the same limitation: they only support a small set of pre-defined modes, all of whose operand's bit-width are of power of two (e.g., 8-, 16-, or 32-bit), not to mention supporting variant precisions in one single mode to fully utilize hardware resources. On the other hand, the reviewed floating-point architectures were only focused on the IEEE standard precisions, such as single and double precision. Although these architectures with regular precision configurations make the transition from traditional ALUs easier for users, they miss the opportunity to exploit the potential performance improvement with deeper precision optimizations: fully custom precisions.

The potential improvement with highly custom precision is one of the reasons that motivate our research on ALU architectures that provide the possibility for dynamic

arbitrary precision high performance computing. Since true arbitrariness is not currently practical, especially for floating-point ALUs, we are pushing our way towards semi-arbitrary precision, where both performance and practical implementation constraints can be satisfied.

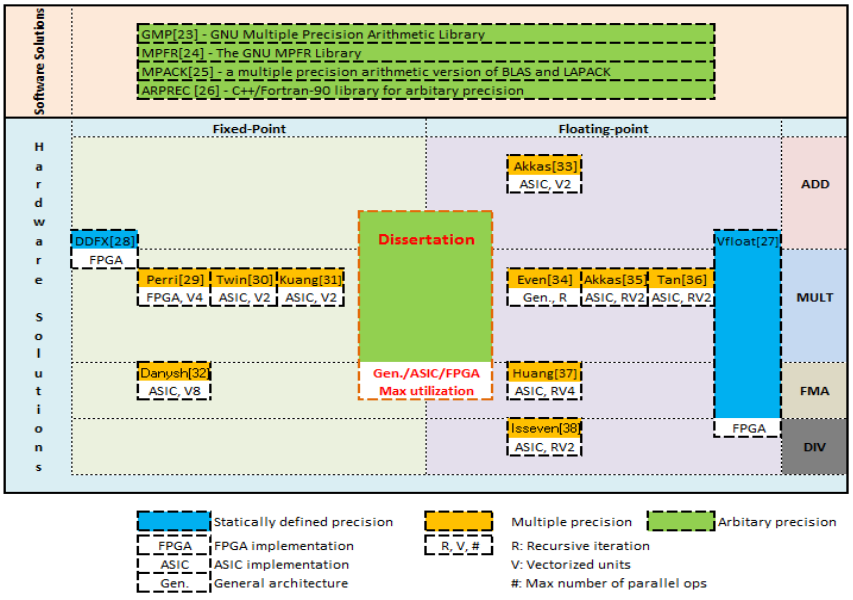

Figure 2.3. Previous work addressing precision for ALUs.

In the following two chapters, the detail on the dynamic precision defining, vectorized operation, and the general dynamic precision architectures is discussed for both fixed-point and floating-point systems, as well as the their corresponding hardware implementations.

CHAPTER III

FIXED-POINT ALU ARCHITECTURES

Arithmetic logic units are the fundamental components within a processor to perform arithmetic and logical operations on operands. Aside from the computation, ALUs must also be able to detect exceptions and generate status flags correspondingly. To simplify the design process, we will be choosing the most commonly used binary system as the basic number system for our architectures, and using two's complement representation whenever signed values must be represented.

A. Dynamically Defined Precision

In contrast to traditional statically defined precision, where once the ALU is implemented the operational precision is fixed, ALUs with dynamically defined precision allow changing the operational precision according to runtime accuracy requirements. Fig. 3.1 illustrates the block diagrams of a traditional two-input adder and an adder architecture that supports more than one precision. The difference between the two can be observed: the bit-width for the control (*ctrl*) and carry-out (*carry*) signals has increased from 1-bit to multiple bits to support multiple precision. Multiple precision designs usually support only certain pre-defined precision modes. For example, the design described in [32] can only operate on four different precision modes controlled by a 4-bit mode signal, namely 64-, 32-, 16-, and 8-bit modes.

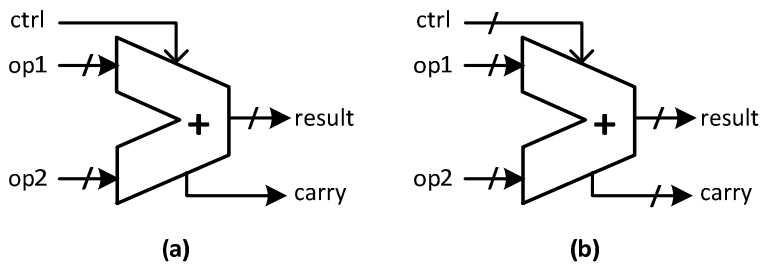

Figure 3.1. Adder units. (a) Traditional unit. (b) Multiple precision unit.

An ALU with true arbitrary precision support allows operations with any combination of different precisions simultaneously, as long as the total bit-width for the operands is not exceeded. To achieve such flexibility, an *n*-bit operand is partitioned into *k* sub-blocks with block size of *n/k*-bits. Depending on the design requirement on precision granularity, the number of blocks, *k*, can be any value between 1 and *n*. This bit size arrangement can be more aggressive, allowing an optimized non-uniform, asymmetric distribution among different blocks. Adjacent sub-blocks can be combined together dynamically to form a super-block, allowing independent operations to be performed on each super-block pair (one for each operand) without affecting others. This dynamic grouping process is controlled by a (*k-1*)-bit control signal (*ctrl*). The concept of operand segmentation and dynamically defined mechanism is illustrated in Fig. 3.2, where an *n*-bit operand is divided into two (*n/4*)-bit numbers and one (*n*/2)-bit number. More precision modes can be obtained by changing the control signal. It is obvious that the proposed mechanism guarantees that the support for arbitrary precision-combinations is only limited by the sub-block segmentation.

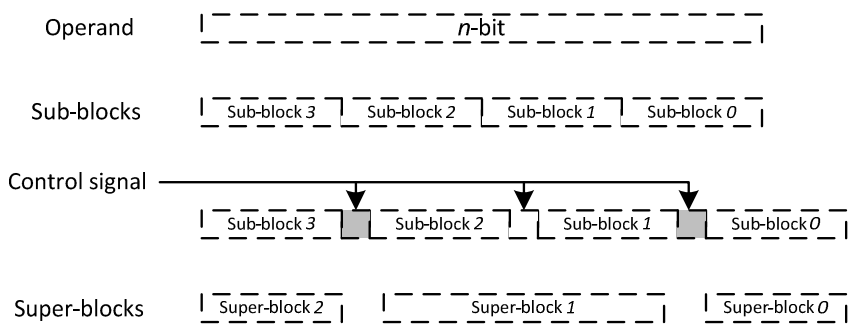

Figure 3.2. Dynamic precision defining mechanism.

Any ALU architecture that is capable of performing operations on these dynamically re-grouped super-blocks independently becomes a dynamic precision ALU. In the rest of this chapter, we will first discuss the detailed architecture for fixed-point adder and multiplier datapaths, as fixed-point arithmetic is the fundamental for fixed-point ALUs, as well as the core datapath for handling the actual computation on the significand and exponent of a floating-point operation. Then, we will expend the discussion to supplement sub-modules for floating-point arithmetic units.

B. Dynamically Defined Precision Adder Architectures

The critical path within an adder design is the carry chain that connects separated full-adders and produces the carries from the lowest position to the output, which makes it the key factor for modifying a traditional adder to support dynamically defined operands. Though implementation varies, the generation process of carry signals can be classified into two categories: (1) carry propagated from previous bits and (2) carry generated using operands [14].

1) Carry Extraction and Insertion

Pure carry propagation is mainly used for slow adder realizations that require less area. In such adders, the carry-out for the current bit position cannot be evaluated until the carry from the previous position is available. Thus, in order to perform independent calculations on the re-grouped super-blocks, the carry propagation between two super-block neighbors needs to be specially handled.

First, it is necessary to terminate the carry propagation, so that the computation of one super-block will not be affected by the result from the previous one. Then, carry extraction and insertion, as described in Eq. (2.3), can be performed to compute the carry-out (or overflow) and carry-in values for the *i*-th and *(i-1)*-th sub-blocks. Given the nature of SIMD operations, an identical carry-in, denoted as c_0, is shared by all the super-blocks as the initial carry-in to identify ADD or SUB operation. But different carry-ins for sub-blocks can be supported for specific applications.

$$\begin{aligned} carry\_out_{i-1} &= ctrl_{i-1} \cdot c_{i-1} \\ carry\_in_i &= ctrl_{i-1} \cdot c_0 + \overline{ctrl_{i-1}} \cdot c_{i-1} \end{aligned} \tag{2.3}$$

The hardware implementation is shown in Fig. 3.3. One can see that from both Eq. (2.3) and Fig. 3.3 the corresponding *ctrl* signal bit is used to generate the correct carry-in and carry-out signal for that respective sub-block. In fact, this circuit also suggests the dynamic construction of super-blocks implicitly, as described in the precious section. Thus no extra hardware resources have to be allocated for a dedicated segmentation circuit. Note also that for every sub-block, there are 4 extra gates, resulting in two extra

gate-levels delay. The latency from every extra block accumulates due to the fact that these circuits are cascaded on the critical path. Therefore, the total area and delay overhead introduced to a traditional ripple-carry adder can be calculated with Eq. (2.4).

$$\begin{aligned} area_{overhead} &= 4k \\ delay_{overhead} &= 2k \end{aligned} \tag{2.4}$$

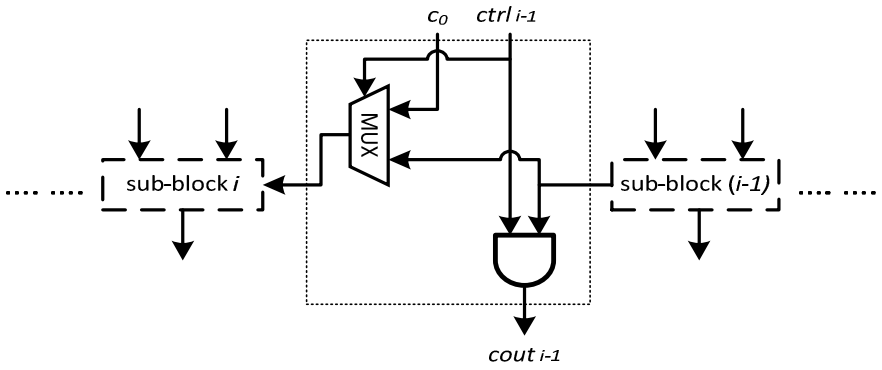

Figure 3.3. Carry extraction and insertion circuit.

2) Carry Manipulation

In contrast to the ripple adders whose worst-case delays increase linearly with the bit-width, most modern processors employ carry-lookahead adders, where the carries are generated from the operands directly or indirectly [14]. This scheme can achieve logarithmic time delay at the cost of extra hardware area. Instead of working on the carry signals directly as in ripple-carry adders, carry manipulation implements the carry processing at the edge between two continuous super-blocks by engineering the MSB's

generate (*gi*) and propagate (*pi*) signals from every sub-block. The logic of this process is shown in Eq. (2.5).

$$\begin{aligned} p_i^{'} &= \overline{ctrl_i} \cdot p_i \\ g_i^{'} &= ctrl_i \cdot c_0 + \overline{ctrl_i} \cdot g_i \end{aligned} \tag{2.5}$$

By comparing Eq. (2.5) with Eq. (2.3), one can see that they are arithmetically similar to each other. Thus, the hardware logic for carry manipulation can also be implemented with one two-input MUX and one AND gate. However, with this method, the calculation of all generate and propagate signals are performed in parallel without having to wait for the completion of carry propagation from previous positions. Since the carry generation process is modified, the calculation of the final carry-out (overflow) for each super-block requires one extra gate-level delay. Therefore, the total latency overhead for adding dynamic precision support is only *(2+1)* gate-levels. For better comparison with the ripple-carry adder, the absolute total area and delay overhead introduced is listed in Eq. (2.6).

$$\begin{aligned} area_{overhead} &= 4k \\ delay_{overhead} &= 3 \end{aligned} \tag{2.6}$$

C. Dynamically Defined Precision Multiplier Architectures

A generic tree multiplier architecture that supports dynamically defined operations is illustrated in Fig. 3.4. It consists of three major modules: a partial product generator, a partial product reduction tree, and a final fast carry propagate adder. Based on the

generation of the partial products, there are two types of multiplier. One is bit-wise multipliers and the other is high radix multipliers.

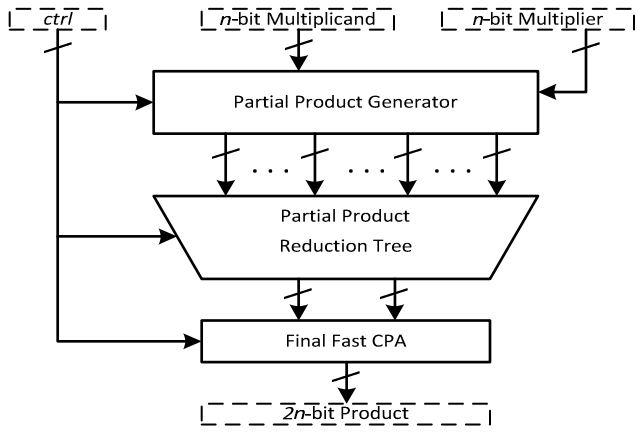

Figure 3.4. Dynamic precision multiplier architecture.

1) Dynamic Precision Partial Product Generator

For a dynamic precision bit-wise multiplier, instead of copying the whole multiplicand as the partial product for every '1' in a multiplier bit during the generation of partial products, the generator selectively masks the corresponding multiplicand super-block bits according to the position of multiplier bit, and discards all the others. Fig. 3.5 shows the hardware for the masking process of super-blocks and the generation of an un-shifted partial product. The additional area and delay introduced is listed in Eq. (2.7).

$$\begin{aligned} area_{overhead} &= k(k-1) + \sum_{i=2}^{k-1}\left(\sum_{j=1}^{i-1}(i-j) + \sum_{j=i+1}^{k}(j-i)\right) = \frac{1}{3}k^3 - \frac{1}{3}k \\ delay_{overhead} &= \log_2(k) \end{aligned} \tag{2.7}$$

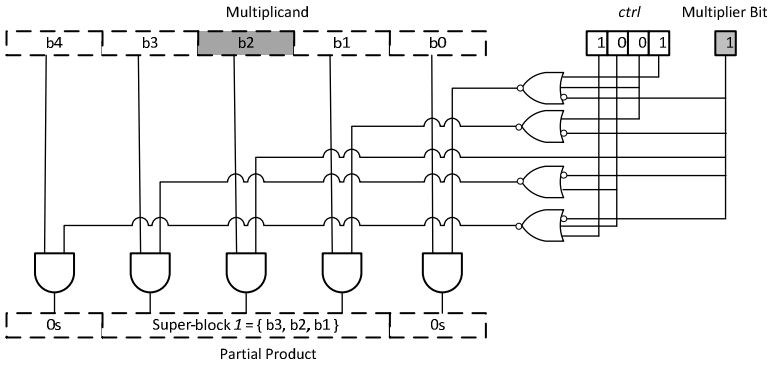

Figure 3.5. Dynamic precision bit-wise partial product generation.

In a high-radix multiplier, one partial product is generated for every two or more multiplier bits, depending on the actual radix. Booth recoding is one of the most popular high-radix partial product generator. When Booth recoding is used, additional modules are required to generate the proper partial products, responsible for the generation of sign extension and hot blocks, as well as their respective placement within the partial product. The detailed discussion on a modified Booth-4 partial product generator is covered in Chapter IV when presenting the dynamic precision floating-point multiplier architecture.

2) Reduction Tree and Final Faster Adder

If a bit-wise partial product generator is employed, the carry will never be generated or propagated outside of the super-block (2x the size of the operands) due to the nature of unsigned multiplication. Therefore, no additional circuit is required in the reduction tree. However, when Booth recoding is used, some of the partial products might become negative numbers represented in two's complement. Summation of two's complement numbers can cause overflow, resulting in unwanted carries propagating through super-

block boundaries. The *ctrl* signal is used to filter any unwanted carries at the sub-block boundaries. At the sub-block boundaries, one extra AND gate is added to the critical path for every level of the reduction tree. In fact, this can be further optimized by reducing the number of level that might cause unwanted carries, or integrating the termination circuit with the reduction elements. Accordingly, depending on the actual implementation on the reduction tree, the final carry propagate adder might be required to support dynamically defined super-blocks. A detailed explanation on the dynamic precision reduction tree is presented in Chapter IV when discussing the significand multiplier for a floating-point multiplier.

D. Dynamically Defined Precision Multiply-Accumulator Architectures

It is quite common for a multiplication of two numbers to be immediately followed by an addition with another number or an accumulation for some applications, such as polynomial evaluation and vector dot product. This essential operation is often referred to as a multiply-accumulate operation, and the arithmetic logic unit that performs this operation is called a multiply-accumulator or a MAC. A MAC computes $p = a \cdot x + b$, where *a* and *x* are n-bit operands and *b* is a *2n*-bit number. The additive operand *b* can be from either the product of another multiplier or the output from the MAC itself.

A MAC can improve the total operation latency and reduce the hardware requirement when compared to the two individual ALUs (a multiplier and an adder) required for the same operation. In a multiplier implementation, a carry propagate adder is employed to generate a final product by adding the reduced partial product in carry-safe format. When

combining the two operations, this final adder can be replaced by one extra row in the reduction tree, resulting in a smaller and faster MAC. Fig. 3.6 depicts the implementation of a MAC that supports dynamically defined precision, which is basically a dynamic precision multiplier with one extra 3-to-2 row added to the reduction tree to accommodate the additive operand. Thus, the overhead analysis for the dynamic precision support discussed for the multiplier and adder also applies to the MAC. Unlike the multiplier, the extra row of 3-to-2s must be able to terminate any unwarranted carries that are propagated through super-block boundaries with the help of the *ctrl* signal, no matter which of the two discussed partial product generators is employed. This is due to the fact that the addition of the product and the additive operand no longer guarantees generated carries being valid. In fact, this extra row of reduction tree can be further integrated with the original tree rather than an extra physical 3-to-2s row for a faster implementation.

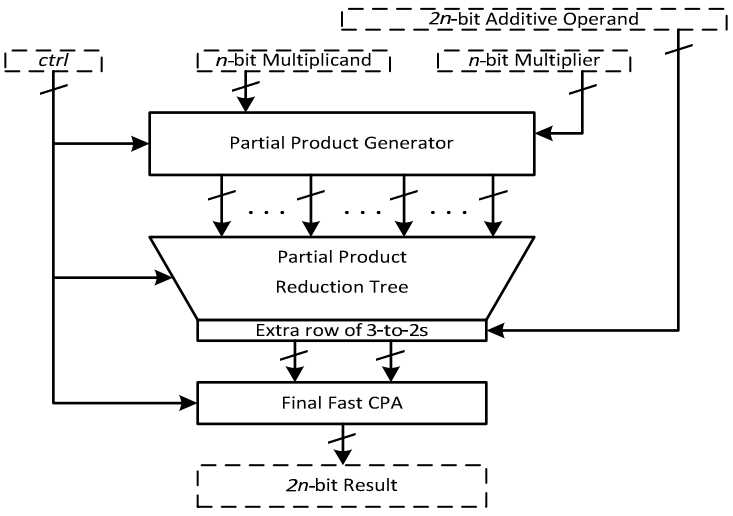

Figure 3.6. Dynamic precision multiply-accumulator architecture.

In this chapter, two different schemes for handling carry chain for adders of different architectures are presented in detail with their theoretical overheads in area and latency. Each of the schemes has its unique characteristics and is effective for certain type of adder architectures. The approaches for designing key components for both bit-wise and modified Booth-4 multipliers with dynamic precision support are also introduced in general. And a brief discussion is given on how a dynamic precision multiply-accumulator can be constructed using the techniques presented for the adder and multiplier architectures. All the fundamental designs can be extended and incorporated into the designs of floating-point ALUs, which is discussed in detail in the next chapter.

CHAPTER IV

FLOATING-POINT ALU ARCHITECTURES

A typical floating-point representation usually consists of four parts (the sign, the exponent, the implicit exponent base, and the significand), making the dynamic precision defining process more complicated than that of a fixed-point system due to the increasing design space. Changing the accuracy in the floating-point system can be achieved by changing the size of the significand (i.e., mantissa), the size of the exponent, the exponent base, or any combination of the above. It is true that we have more potential flexibility with floating-point systems in terms of re-defining precision. However, the more flexibility we exploit, the more complicated the architecture is, implying a larger and slower implementation. Besides, we also need to address programmability concerns for the end users. Thus, we make a few assumptions when designing floating-point architectures. First, uniform sizes for both exponent and significand will be used, meaning that all re-defined sub-operands (similar to super-blocks in the fixed-point system described in Chapter III) are of the same size. The floating-point ALU can be treated as a vectorized processing unit performing operations in a SIMD fashion. Second, the exponent bias should always follow Eq. (4.1).

$$bias = 2^{e-1} - 1 \quad \text{where } e \text{ is the exponent bit-with} \tag{4.1}$$

To better demonstrate the proposed arithmetic logic unit architectures supporting dynamically defined multiple precision operations, an eight-mode scheme is employed

throughout the dissertation to help visualize the general multiple precision architectures with a more specific and practical scenario. In this scheme, an ALU allows operations in eight different preset precision modes. According to a 3-bit select signal, *mode*, the ALU design can perform one or multiple parallel operation(s) on operands with a selectable precision configuration. Table 4.1 shows the vector and precision configurations for each mode, in which k is the number of concurrent operations supported, e is the size of the exponent for each operand, and s is the size of the significand for each operand. These configurations for the specific eight mode scheme are decided after a considerable tradeoff between maximizing utilizing/sharing of fixed hardware resources, simplifying architectures, and providing approximate support for IEEE standard precisions. For instance, an ALU operating in mode '000' can perform 5 concurrent corresponding computations on IEEE single-precision numbers.

Table 4.1. Configurations of the eight-mode scheme.

Mode	**Number of Element**	**Precision Configurations**	
	k	***e***	***s***
000	10	8	11
001	8	8	15
010	6	8	19
011	5	8	23
100	4	11	31
101	3	11	39
110	2	15	63
111	1	15	127

A. Data Organization

An arithmetic logic unit is required to exchange data with the external registers/bus before and after computations, so the external interface vector format for data exchange is crucial in terms of programmability and efficiency for multiple precision ALUs. For example, the performance of the computation will definitely suffer from resource overhead when the return data from an ALU requests complicated post-processing by either hardware or software before use by another part of the computation.

First of all, let's look at the bit-width boundary for an operand vector in a multiple precision ALU. Suppose an arithmetic logic unit supports *m* different floating-point precision modes, and the bit arrangement for the vector element for mode *i* is 1 bit for the sign, e_i bits for the exponent, and s_i bits for the significand. Then the total bit-width *n* for the input port for one operand vector is constrained by Eq. (4.2), where k_i represents the number of vector elements (i.e., sub-words).

$$n \geq \max\left(k_i \cdot (1 + e_i + s_i)\right) \text{ where } i \in m \tag{4.2}$$

Unlike fixed-point numbers, floating-point numbers consists of three explicit distinct parts (except the implicit exponent base), providing more options when defining the exchange format. However, in order to minimize the complexity for applications to adopt our ALU architecture, the external interfacing data format is simply designed as concatenating multiple sub-operands into one long vector and padding with zeros if the total bit-width is less than *n*, as shown in Fig. 4.1. Doing this helps to minimize the extra

burden on the external hardware or software responsible for the construction of operand vectors, but relies on the ALU itself to interpret the information during data exchange.

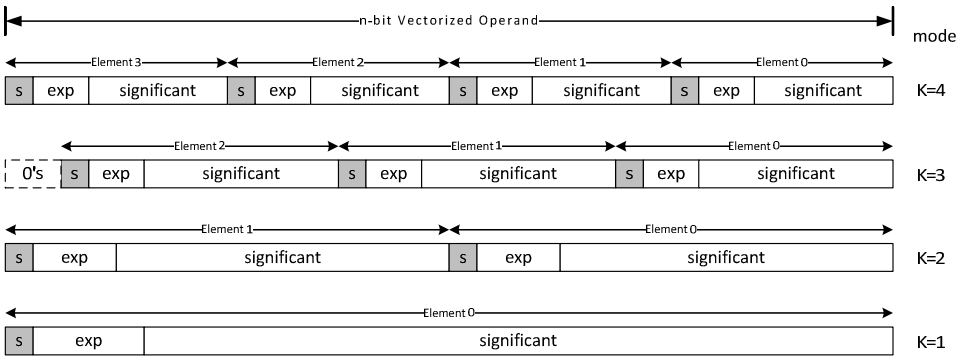

Figure 4.1. Examples of external data arrangement.

Before being further process by the ALU, the data in external format must be converted to internal format. This conversion involves extracting the three floating-point components for each vector element from the input, and reorganizing them into three smaller sub-vectors accordingly, namely a sign vector, an exponent vector, and a significand vector. The detailed element arrangements of the three sub-vectors for the eight modes are illustrated in Fig. 4.2. It is straightforward for the 10-bit sign vector, where all the signs are concatenated together. However, instead of simply combining all exponent elements together as with the sign vector, the 80-bit exponent vector is organized in a way that all the elements are aligned to 8-bit sub-blocks by padding with zeros as necessary. In this way, we can take advantage of the dynamically defining process described above when processing exponent addition and increment. For example,

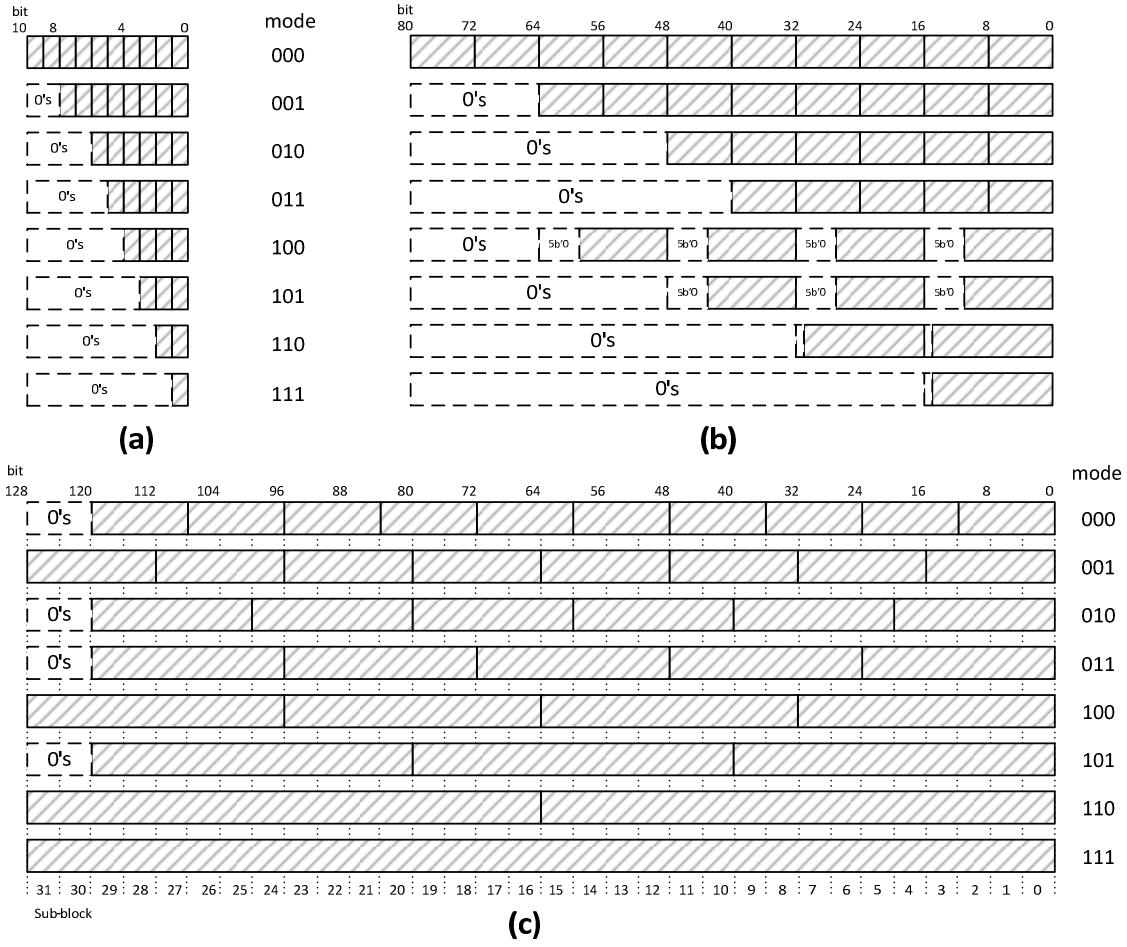

Figure 4.2. Internal data arrangement for the eight-mode scheme.

(a) Sign vector; (b) Exponent vector; (c) Significand vector.

for mode ‘101’, 5 zeros are inserted before the 11-bit exponent making it so each element occupies two 8-bit sub-blocks. The same strategy is also applied to the 128-bit significand vector. The precision configuration for significand elements is designed such that, after adding the hidden *‘1’* for normalized numbers, those elements are already aligned to 4-bit sub-blocks, thus simplifying the exception detector and the significand datapath design, and at the same time maximize hardware utilization to certain degree.

B. Floating-point Multiplier Architectures

The block diagram of the proposed multiple precision multiplier is illustrated in Fig. 4.3. It does not look too much different from that of a traditional floating-point multiplier other than the highlighted *mode* signal, where the operating precision requirement is encoded as described in Table 4.1. This *mode* signal is the key for enabling multiple precision support. The diagram also shows three datapaths responsible for the vectorized processing on the three internal operand sub-vector pairs, as well as a dedicated exception detecting and handling module.

Note that the sign logic module represents the only exception that the *mode* signal is not required for its operations. Given the fact that this module is only one level of relative small XOR gates in order to generate the final signs, and the fact that it is not on the critical path, it would be wise to fully implement the logic on the sign vector rather than designing a complicated mode-dependent mechanism. This means the XOR operation is performed on all the vector elements regardless of current operational mode, as described in Eq. (4.3).

$$\{signP_{k-1},...,signP_1,signP_0\} = \{signA_{k-1} \otimes signX_{k-1},...,signA_1 \otimes signX_1, signA_0 \otimes signX_0\} \quad (4.3)$$

In fact, the correctness of the sign generation is guaranteed by zero-padding the sign vector during the unpacking process, which will be discussed later. To simplify the design and avoid unnecessary delay overheads, this strategy is applied to logic throughout the whole design, whenever the resulting extra hardware cost is considered negligible when compared to the gain in speed.

The rest of the discussion in this section will be focused on the indispensable data format conversions, exception handling, and how the vectorized processing is realized for the two major operation datapaths, especially the exponent and significand, making it possible for multiple precision support.

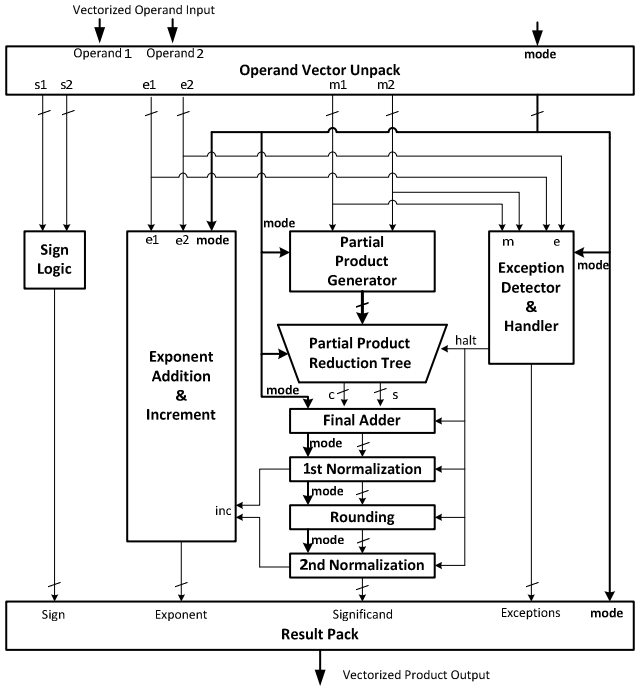

Figure 4.3. Block diagram for a dynamic precision floating-point multiplier.

1) Vectorized Unpacking and Packing

The unpacking and packing modules are the interfaces between the multiplier and the outside circuit responsible for the internal/external data format conversions. Once the mode and precision configurations are determined, the vectorized operand unpacker is just a set of hardwired multiple-input multiplexers (one for each bit in the resulting vector), the size of whose inputs is decided by the number of modes supported. The unpack module for our eight-mode multiplier is shown in Fig. 4.4, where only the connections for the exponent vector are listed. Vectorized significand unpacking shares the same idea. The hidden '1' for the significand segment is inserted accordingly during the unpacking process to generate the internal significand vector. On the other hand, the functionality for the final product packer is just a reverse of the unpacking process with exception-controlled zeros/ones injections to handle the IEEE standard defined special cases (e.g., ±0 and INF) [39]. Therefore, the structure is similar to that of the unpacker with one extra gate delay for the injections.

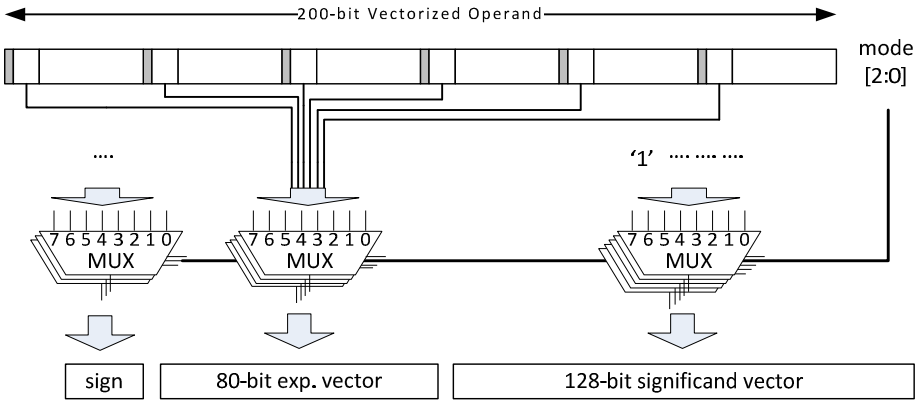

Figure 4.4. Vectorized unpacker.

2) Vectorized Exception Handling

In order to support exceptions and special values defined by the IEEE 754 standard, the vectorized exception detector must be able to distinguish the all-zeros or all-ones scenarios in all exponent and significand elements. For example, if both the exponent and significand of one of the operands are all zeros, then the numerical value of this operand is 0, suggesting no further operation is necessary since the final product is 0. A halt signal is generated accordingly to stop the significand and exponent processing circuit responsible for that particular sub-word(s) from unnecessary switching, resulting in dynamic power reduction. The final product for that multiplication should be set to 0. However, if the other operand sub-word is determined to be INF (all-ones for the exponent and all-zeros for the significand), the result should be set as NaN, and the INVALID register should be flagged [39].

From the data arrangement for the exponent vector shown in Fig. 4.2(b), it is obvious that each element consists of one or two sub-blocks, and their structure is relatively regular. The detection of all-zeros/ones for the exponent vector is straightforward. A three-level OR/AND binary tree is designed for the all-zeros/ones detection for each sub-block, and one extra OR/AND gate is used to combine the tree outputs from two adjacent sub-blocks if required. Note that the padded zeros for the exponent element should be handled (e.g., inverted) when detecting all-ones exponents.

However, due to the relatively irregular arrangement of the significand vector, its detector circuit is slightly more complicated than that of the exponent vector. Fig. 4.5 describes a five-level OR tree that is used to generate the intermediate or final

block_allzeros signals for any sub-word structure. Those numbered blocks represent the components (4-input OR gates) for generating allzeros signals for the basic 4-bit sub-blocks. Two examples are highlighted to explain in detail how the sub-word detectors are constructed. The significand's allzeros detection can be directly extracted from the OR tree, when the number of sub-blocks used to construct a significand is a power of two, and their locations are aligned to those used to generate one of the OR tree nodes. Example 1 highlighted in black shows the output of the allzeros detector for significand element 6 in mode '001' is the output of the 7th node of the 2nd level OR tree. On the other hand, when the sub-blocks are not aligned, the detection can be realized by combining outputs of multiple OR tree nodes from the same or different levels. As illustrated in example 2 highlighted in grey, the outputs of two 2nd level nodes and one 1st level node are combined with two extra OR gates to generate the final allzeros signal for element 1 in mode '101'. It can also be seen from Fig. 4.5 that the extra gates only increase the area but not the total latency (2+2 levels). In fact, for our eight-mode multiplier, the latency for generating allzeros for all significand elements is equal to or less than that for a 128-bit significand (5 levels). This mechanism is also applied to the vectorized sticky logic in the rounding module.

3) Vectorized Exponent Processing

The exponent of a floating-point number is represented in a biased format, as in $\exp = exponent + bias$. Therefore the biased exponent output must be calculated by adding two exponents and then subtracting the *bias* as described in Eq. (4.4).

$$\exp_{out} = \exp 1 + \exp 2 - bias \tag{4.4}$$

However, subtracting *bias*, represented in (*e-1*)-bit ones, requires the borrow to be propagated through the whole carry chain, resulting in a longer calculation time. By re-arranging Eq. (4.4) into Eq. (4.5), the slow subtraction can be transformed into a simple act of flipping one single bit [14].

$$\exp_{out} = (\exp 1 + \exp 2 + 1) - (bias + 1) \tag{4.5}$$

To compensate for the extra *'1'* required for the flipping, an adder with a fixed *'1'* as carry-in is employed to implement the first part of Eq. (4.5).

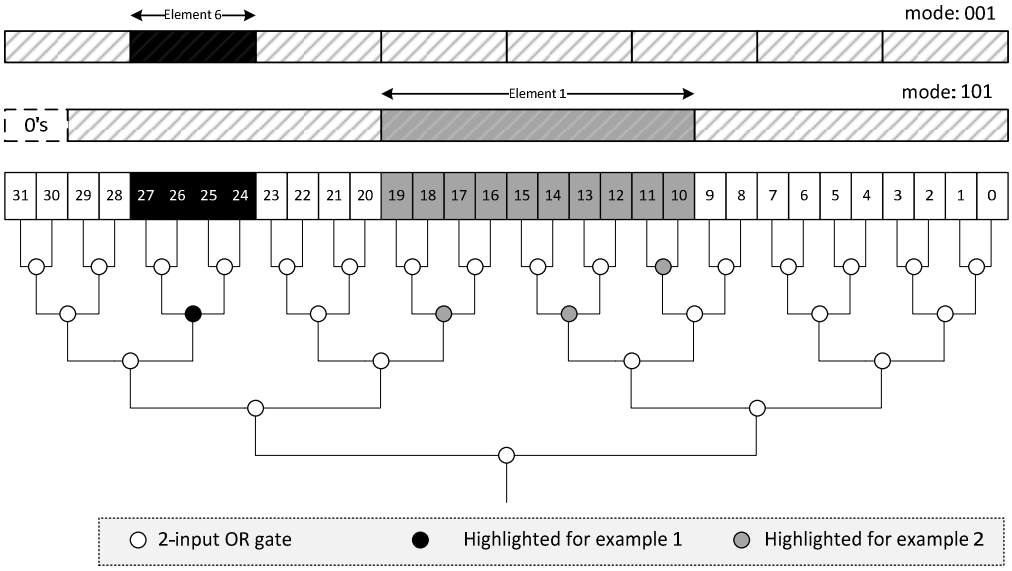

Figure 4.5. Two examples of 5-level OR tree sharing for a 128-bit significand vector.

A special dual mode 16-bit exponent adder, illustrated in Fig. 4.6, is designed to perform addition on two adjacent 8-bit sub-blocks in the exponent vector. A total of five

copies of this special adder are required to cover the 80-bit exponent vector for the eight-mode multiplier. According to the description in Fig. 4.2(b), an adder should perform two individual 8-bit additions in mode ‘000’, ‘001’, or ‘010’, or one 16-bit addition in all the other modes. The same dual mode structure is used for the exponent increment used for vectorized normalizations. In order not to aggravate latency for the critical path, the vectorized exponent increments are performed in advance, and then final exponent outputs are selected by the *inc* signals from the normalizers. Combined with the flipped bit and increment requirement from the significand normalizers, the carry-out bits from the special adders are used to determine exceptions of overflow and underflow.

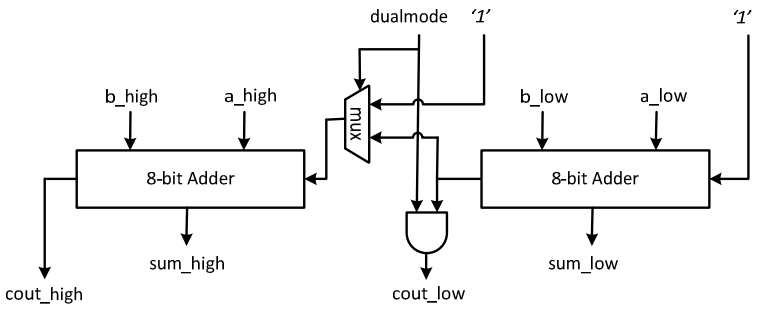

Figure 4.6. Dual mode 16-bit adder with a fixed carry-in of ‘1’.

4) Vectorized Significand Multiplication

As shown in Fig. 4.3, significand processing circuitry, being the most important part within the critical path, majorly determines the latency and size of the design. The significand processing modules include a full significand multiplier and post-processing for the generated intermediate product vector, such as normalization and rounding.

Fig. 4.7(a) depicts the operation of a generic bit-wise significand multiplier, where at first a partial product is generated for each of the multiplier bits. These partial products are then shifted to the left respectively based on the position of the multiplier bit, and finally are added together to form the final product. However, for vectorized multiplication, operations on one of the vector elements must be separable from the others. The separation can be handled by performing special operand eliminations during the addition stage, or it can be achieved earlier during the generation of partial products, where each generated partial product contains only its effective multiplicand element. Obviously, the latter approach is a better choice in terms of circuit complexity and dynamic power consumption, by preventing irrelevant partial product bits from being generated and traveling through the following addition stage. A vectorized significand multiplier using this approach is illustrated in Fig. 4.7(b), and the mechanism of multiplicand selection is explained in Eq. (4.6), where *sel* equals *'1'* when the multiplicand part and the multiplier bit are from the same vector element, otherwise *sel* equals *'0'*.

$$\begin{aligned} y &= (2^h a_H + a_L) \times (2^h x_H + x_L) \\ &= (2^h a_H \cdot sel_{HH}) \times 2^h x_H + (2^h a_H \cdot sel_{HL}) \times x_L + (a_L \cdot sel_{LH}) \times 2^h x_H + (a_L \cdot sel_{LH}) \times x_L \\ &= 2^{2h} a_H \times x_H + a_L \times x_L \end{aligned} \tag{4.6}$$

The block diagram of a vectorized partial product generator responsible for the selection of the corresponding element is presented in Fig. 4.8. Each of the sub-blocks in a partial product is determined by the multiplier bit and the corresponding bit of a block select signal, called *block_sel*, which rules whether the current sub-block index belongs to the same vector element as the multiplier bit or not. The *block_sel* signal is generated

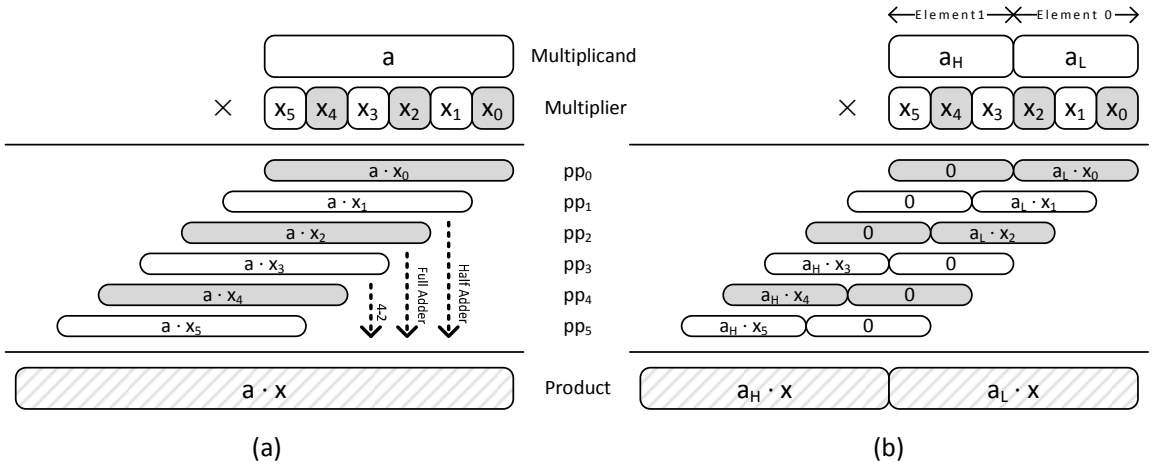

Figure 4.7. Bit-wise significand multipliers. (a) Generic; (b) Vectorized.

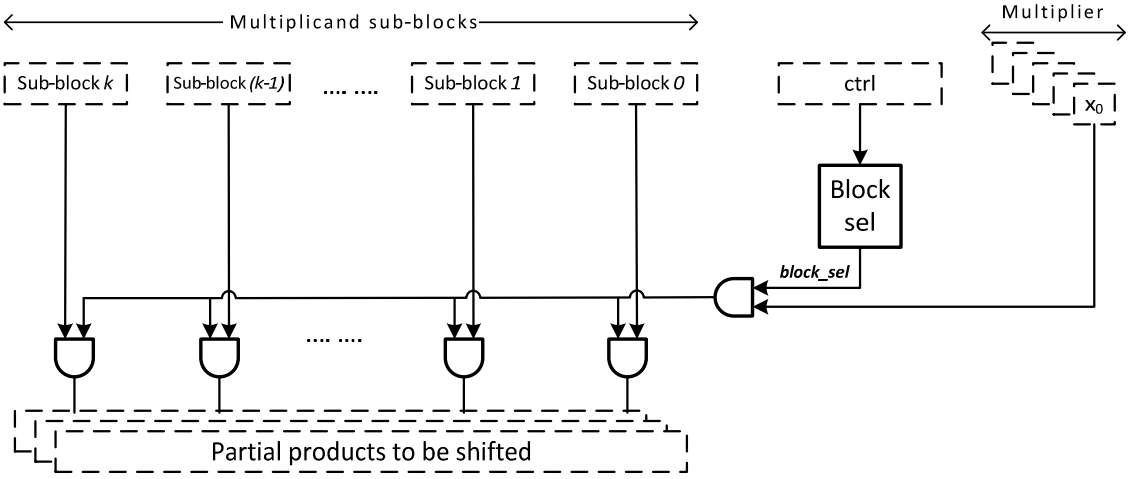

Figure 4.8. Bit-wise vectorized partial product generator.

by a series of OR trees from another set of control signals, *ctrl*, which is used to identify the boundaries between vector elements.

Specific to the eight-mode multiplier design, the deterministic nature of the mode configurations allows the generation of the block select signal for each modes to be directly hardwired, resulting in a much faster and simplified *block_sel* generating block consisting of only multiple parallel copies of 8-to-1 multiplexors selected by the *mode* signal.

The fact that the significand elements are represented in binary format and are always non-negative guarantees the multiplication between two *n*-bit operands will never generate or propagate a carry outside of the *2n*-bit product element boundary as shown in Eq. (4.7).

$$product_{\max} = a_{\max} \times x_{\max} = (2^n - 1) \times (2^n - 1) = 2^{2n} - 2^{n+1} + 1 \tag{4.7}$$

Thus, a traditional reduction tree is sufficient to generate the vectorized intermediate product in carry-save format without worrying about carries between element boundaries. Each slice of the reduction tree, which is either a half adder, a full adder, or a 4-to-2 tree [40] depending on the slice position, is responsible for adding all the partial product bits within the same column. An example of an 8-bit reducer slice constructed with 4-to-2 blocks is illustrated in Fig. 4.9(a), with the implementation detail of a regular 4-to-2 block shown in Fig. 4.9(b). Similarly, it is not necessary to specially design the final adder for vectorized significand multiplication, avoiding extra latency and area costs over a traditional single-precision multiplier. Hence, a total of 256 slices of a 4-to-2 tree of up

to $log_2(128)$ levels is required in order to reduce the 128 partial products generated by the eight-mode multiplier into 256-bit vectorized *s* and *c* components to be added by a 256-bit fast adder to form the final product.

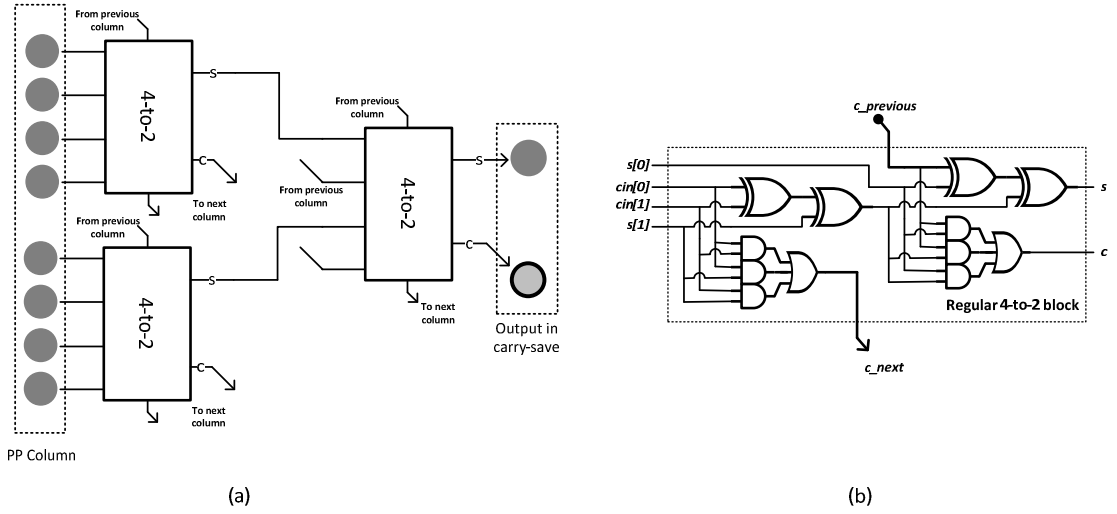

Figure 4.9. (a) 8-to-2 reduction tree; (b) Regular 4-to-2 implementation.

Compared to the bit-wise partial product generator, a high-radix generator, as shown in Fig, 4.10(a) could reduce the number of partial products by half by generating one partial product for every two multiplier bits, resulting in the un-shifted partial product being 0X, 1X, 2X, or 3X of the multiplicand. With modified Booth-4 recoding [44], the generator avoids the time-consuming 3X multiplication with the cost of possible negative partial products represented as 2's complement number. Fig. 4.10(b) illustrated a modified Booth-4 multiplier with simplified sign extension scheme. Note that an extra

partial product is required due to the fact that the MSB of a normalized significand in a floating-point number is always one (the hidden one), making the number of partial product being *n/2+1*.

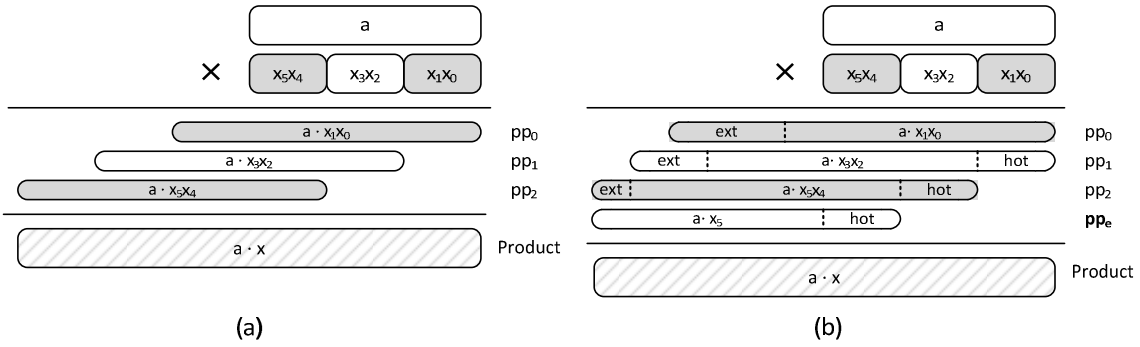

Figure 4.10. (a) Radix-4 multiplication; (b) Modified Booth-4 multiplication.

An example of a vectorized 16-bit significand multiplier operating on multiple precision mode, depicted in Fig. 4.11, shows the details on how this recoding and sign extension scheme is adapted in the proposed architecture. First of all, for this multiplier the possible selections for a partial product sub-block increase from two to seven. A bit-wise generated partial product sub-block could be either zeros or a copy of its corresponding multiplicand sub-block. When the modified Booth-4 recording is applied, the content of a partial product sub-block is determined by Eq. (4.8), where x_{Booth4} represents the effective multiplier for current generation. Note that two new selections, EXT and HOT, are introduced to handle the negative partial product generated from a negative effective multiplier.

$$PP_{sub\_block} = \begin{cases} \text{all zeros} & \text{when } x_{Booth4} = 0 \text{ or } block\_sel = 0 \\ a_{sub\_block} & \text{when } x_{Booth4} = 1 \text{ and } block\_sel = 0 \\ a_{sub\_block} >> 1 & \text{when } x_{Booth4} = 2 \text{ and } block\_sel = 0 \\ \text{INV}(a_{sub\_block}) & \text{when } x_{Booth4} = -1 \text{ and } block\_sel = 0 \\ \text{INV}(a_{sub\_block} >> 1) & \text{when } x_{Booth4} = -2 \text{ and } block\_sel = 0 \\ \text{EXT} & \text{for sign extension block} \\ \text{HOT} & \text{for hot ones block} \end{cases} \tag{4.8}$$

Secondly, the fact that none of the partial products for any significand element ever reaches beyond the product element boundary, making it possible to use a similar reduction structure as the one for the bit-wise multiplier described above. And lastly, the extra partial product for each element can be packed into one single row for all the possible precision combinations, without affecting the computation. This leads to a constant one extra row in the reduction module regardless of the number of elements, which offers a huge flexibility during implementation.

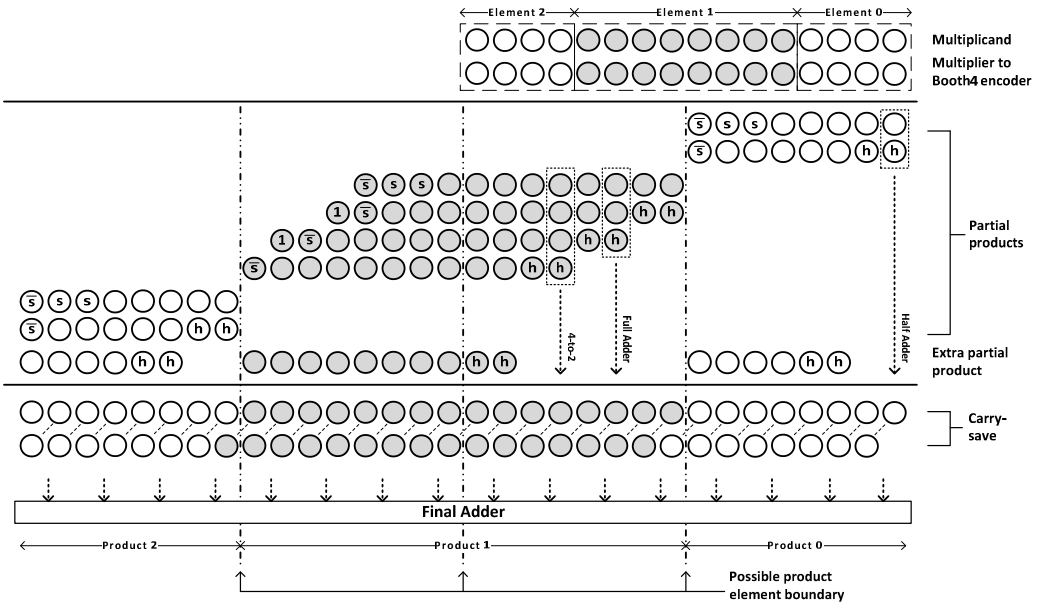

Figure 4.11. Significand multiplier with modified Booth-4 recoding.

The architecture of a vectorized partial product generator empowered with modified Booth-4 recoding and simplified sign extension is illustrated in Fig. 4.12. The *ctrl* signal is used to generate the *block_sel* signal, and two edge flags to indicate the most and least significant sub-blocks for an element, named *block_left*, and *block_right* respectively. The *block_sel* signal not only serves as the selection signal for the final multiplexor to generate the partial product block as in its bit-wise counterpart, but also helps to identify the locations of special extra blocks for the 4-bit EXT and 2-bit HOT. Based on the 3-bit multiplier input, a specially designed Booth-4 encoder produces three indicators, namely *x2*, *non0* and *neg*, to facilitate the generation of intermediate sub-block contents. The *block_right* flag is used by the encoder to prevent the MSB from previous one being grouped and processed for the current element. Depending on the relative sub-block location to the product element, the two edge flags and three Booth-4 indicators are then used to generate either the EXT or HOT block to be connected to the final multiplexor as one of selections.

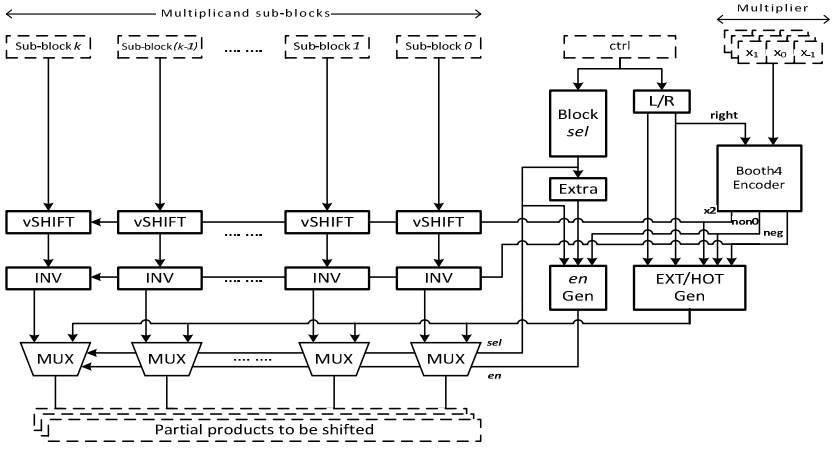

Figure 4.12. Modified Booth-4 vectorized partial product generator.

On the other hand, each multiplicand sub-block goes through the vectorized shifter and inverter controlled by the *x2* and *neg* indicator respectively, and is then connected to the final multiplexor as the other selection (number blocks as opposed to extra blocks like EXT or HOT). These multiplexors are selected by the *block_sel* signal, and only enabled when their corresponding *block_sel*, *block_ext*, and *non0* signals are all high. Otherwise, the outputs are all zeros. With the similar techniques, one extra partial product is generated for every operand pair, but consists only the number block and HOT. Then the extra partial products from all the elements are grouped as one single partial product row to be reduced by the reduction tree.

Unlike those created by the bit-wise generator, the partial products generated with Booth-4 recoding can be either positive or negative numbers. Therefore, there will be invalid carries generated or propagated beyond the product element boundaries when performing reduction with 4-to-2 trees and the final addition. These unwarranted carries must be eliminated in order to ensure the correctness of the vectorized significand multiplication. Fig. 4.13 depicts a specially designed 4-to-2e reducer that performs controllable carry elimination. The output of the two carries generated within the 4-to-2e block is controlled by the *ctrl* signal using two extra AND gates. In a reduction tree designed for Booth-4 multiplier, the slice of 4-to-2 tree is replaced with 4-to-2e tree at all possible element boundaries. Other than this, there is no difference between the two reduction trees. A multiple precision adder discussed in the previous section is required to calculate the final significand product.

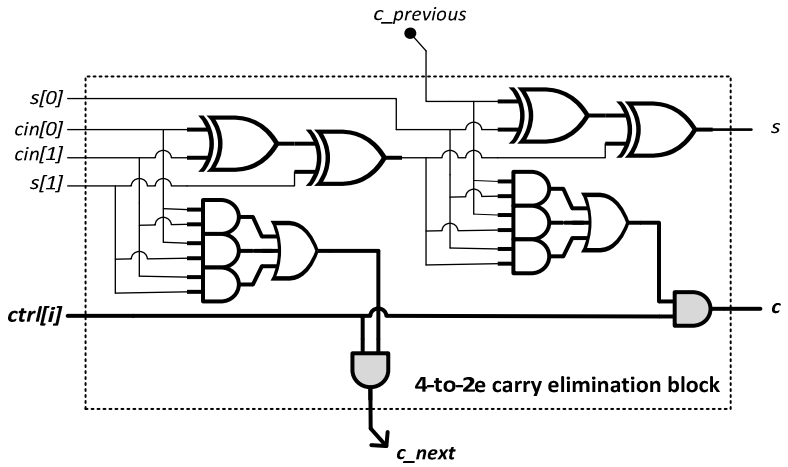

Figure 4.13. Special 4-to-2e block with carry elimination.

5) Vectorized Significand Normalization and Rounding

Post-processing of normalization and rounding are required to generate the final significand product that complies with the floating-point representation standard and operation rules.

For multiplication of normalized significands, each of the product elements needs to be shifted one bit to the right at most. The hardware for both the first and second normalizer is similar, except for the size definition of sub-block. In order to support vectorized normalization on the intermediate product vector, two challenges must be addressed. One is to ensure the shift does not happen between different sub-words, and this is achieved with the help of the *ctrl* signal generated from the *mode* signal to identify the sub-word boundaries. As illustrated in Fig. 4.14, bits of the *ctrl* signal are connected to the *sel* ports of a series of multiplexors at the edge of each sub-block to prevent the shifts between sub-words. The product sub-block size is increased to 8-bits for the first

normalizer to preserve the relative position among sub-blocks with the intension of sharing the same *ctrl* signal and applying a unified sub-block management.

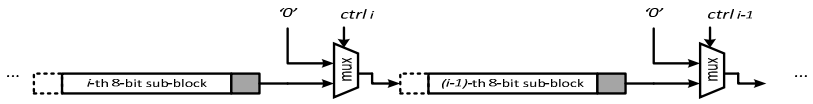

Figure 4.14. Shift between sub-blocks for the first vectorized normalizer.

Another challenge for the vectorized normalizer is to shift only the sub-words that are required. Due to the relative irregularity of the significand vector arrangement, it is difficult to perform such operations directly on the product vector. We use a *shift-first-select-later* strategy to perform shifting on all the sub-words, and then select the corresponding output (shifted/unshifted) for each sub-word according to the *shift_sel* signal generated from the *mode* signal and the MSB of the corresponding sub-word. An example for our eight-mode multiplier is shown in Fig. 4.15 to illustrate the idea of a vectorized normalizer.

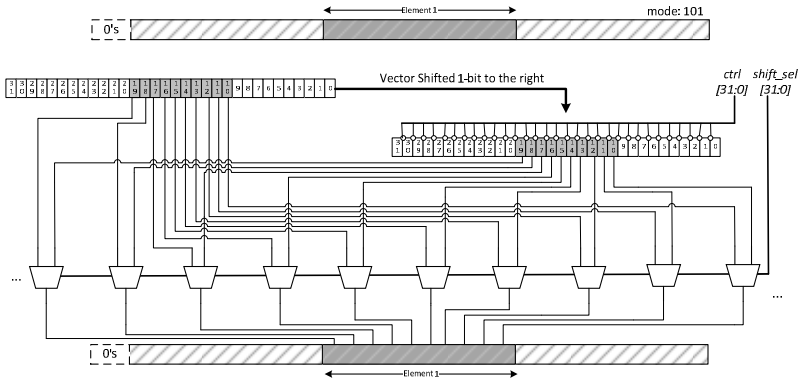

Figure 4.15. Vectorized normalizer for multipliers.

A block diagram of the vectorized rounding module is illustrated in Fig. 4.16. The shifted intermediate vector is partitioned into two vectors, *high_vector* and *low_vector*. During the partitioning, *block_ulp* and *block_r*, which represent the unit in the last place (*ulp)* and the rounding bit for sub-blocks respectively, are also extracted with the help of the *ctrl* signals. The *low_vector* is used to generate the sub-block sticky bits *block_s* using a vectorized sticky logic, which applies the same shared OR tree method as discussed before. Based on the *block_ulp, block_r*, and *block_s* signals, IEEE rounding modes (e.g., round to nearest even) can be performed in the rounding decision module controlled by the *round_mode* requirement. A dynamic precision INC/ADD, which is similar to the dual mode adder in the exponent processing datapath, but with a better flexibility, is used to compute the rounded product vector.

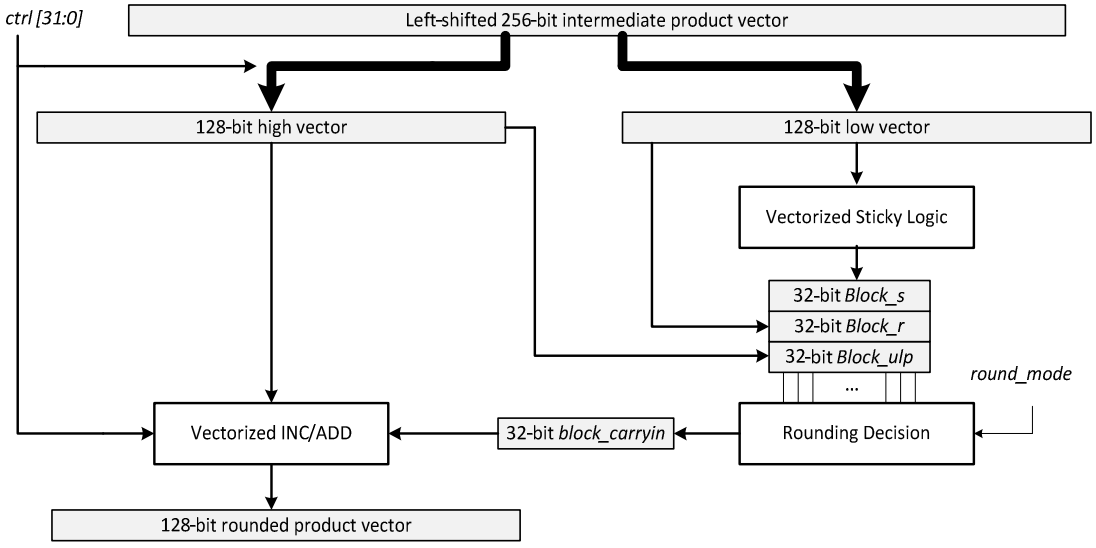

Figure 4.16. Block diagram for a vectorized rounding module.

C. Floating-point Adder Architectures

Contrary to the daily life experience, the operation of adding two numbers together is more complicated than the multiplication when it comes to floating-point system. Recall from the floating-point multiplier structure shown in Fig. 4.3, there are three distinctly separated datapaths for the processing of the signs, the exponents and the significands. It allows a much simplified implementation, especially when tackling the critical path, which being solely the significand path. For example, for a traditional multiplication, the significand of the final product can be calculated with only the information of the significands of the input operands themselves, regardless of the signs and exponents. Yet this no longer applies to a floating-point addition. The generation of both the final sign and significand relies on values of all three components from both operands, meaning that in a design there should be either duplicated processing units for each datapath, or shared intermediate data between the three datapaths. Obviously, the latter is preferred.

A simplified block diagram for a traditional floating-point adder using shared resources is depicted in Fig. 4.17. By comparing this with the diagram for a floating-point multiplier shown in Fig. 4.3, there is clearly more information exchanged among the three main datapaths in order to generate the final output. The basic procedure of performing an addition between two floating-point numbers involves serval steps that will be discussed using the three datapaths.

First of all, effective operation for current inputs must be determined based on the *op* input and the signs. This effective operation decides whether an addition or a subtraction

should be actually performed on the significands. Yet this alone is not enough to determine the final sign, because the output of the significand adder could be either positive or negative depending on which operands input has the larger absolute value. Therefore, to generate the final sign, the sign logic block requires the comparison results from both the exponent and significand datapaths, which combined represent the comparison of operand magnitudes.

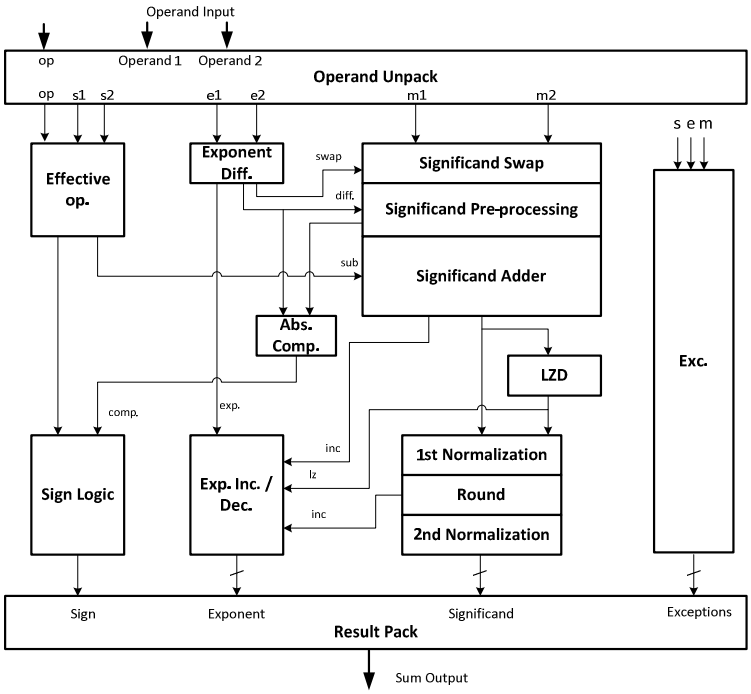

Figure 4.17. Simplified block diagram for a floating-point adder.

On the other hand, the difference between the two exponents should be calculated. The exponent in a floating-point number represents the actual position of the radix point in the number's significand, so the difference determines the direction and the number of

positions a radix point should be shifted before proper computation can be performed. The larger exponent becomes the shared exponent for the aligned significands, assuming only right shift is implemented for the smaller significand. This shared exponent is then updated, either increased or decreased during the two normalization stages in the significand datapath, depending on the sum value.

Before a fixed-point adder can be employed to perform the significand addition, it is necessary to pre-process the significand inputs. One of the pre-processing involves shifting the significand from the number with a smaller absolute value to the right such that radix point of the shifted significand is aligned with that of the other, then a proper fixed-point operation can be performed. The implementation of an arbitrary position shifter for this purpose is relatively huge in size. Thus, instead of one arbitrary shifter for each significand, it is preferred to only implement one instance of a single-directional shifter in a fix input position (e.g., left). To facilitate this area saving scheme, a swapper must be employed to make sure the significand to be shifted is always placed in that exact position. For example, only the significand with the smaller exponent is to be shifted to the right to aligned with the one with a larger exponent, as described in Eq. (4.9).

$$\begin{aligned} |a|+|b| &= \exp_a \cdot \mathrm{sig}_a + \exp_b \cdot \mathrm{sig}_b \\ &= \exp_a \cdot (\mathrm{sig}_a + \mathrm{sig}_b \cdot 2^{\exp_a - \exp_b}) \end{aligned} \quad \text{where} \quad \exp_a > \exp_b \tag{4.9}$$

Also, a selectable inversion for each significand is a part of the pre-processing to prepare the significands for the following addition or subtraction that is depending on the effective operation. Employing two copies of selectable invertor instead of one makes it possible to generate positive significand sum for all possible scenarios, avoiding the slow

plus one operation when converting a negative significand to a positive one. The post-processing after the fixed-point adder is quite similar to that for a multiplier, except that the significand shift during the first normalization could be towards either direction. The number of bits that is to be shifted is no longer limited to 0 or 1, but any number from 0 to (n+2).

1) Overview of Dynamic Precision Adders

When looking at the block diagram, one of the most significant differences between the floating-point multiplier and adder architectures is the elaborated interconnections among the three main datapaths. This not only makes the critical path of the adder architecture more complicated, but also posts a great challenge for the design of a dynamic precision adder. The interconnection aspect is briefly discussed in the previous section, yet the design complexity caused by this issue does not seem to pose a big problem for the traditional floating-point adder. It is because that there is only one sign, one exponent, and one significand to be processed with each datapath, so it is deterministic when the information is transferred between datapaths.

However, this no longer holds when vectorization is introduced to support dynamic precision. Given the fact that there is more than one element in an operand vector for a dynamic precision adder, it is important that the information exchange between two datapaths is within the range of an element pair. Unless the element arrangements for all three vectors are identical, which is rarely practical in scientific computation, a mapping mechanism must be implemented in between the two interrelated vector formats. Take significand pre-shift under mode “100” of the eight-mode scheme as an example, whose

mapping details between vector formats is illustrated in Fig. 4.18. For the shift of significand element 0, the sub-block 0 and sub-block 1 of the difference vector, which is derived from the exponent vector and thus saved in the same format, should be mapped to the sub-block 0 to sub-block 7 of the significand vector as the exponent difference input. Although they are going to put negative impacts on the design in terms of delay and area, these additional mapping blocks are unavoidable as the cost of supporting extra precision modes. Once the precision configurations are determined, these mapping blocks can be implemented as series of hardwired interconnections followed by a multiplexer, just like the previous discussed packer/unpacker, thus limiting the impacts on the critical path of a design. Several versions of mapping blocks are required for the proposed dynamic precision floating-point adder under the eight-mode vector arrangement scheme, whose block diagram is shown in Fig. 4.19. These blocks include those mappings from each of the three vector formats to the other two formats, except the ones from sign to exponent format.

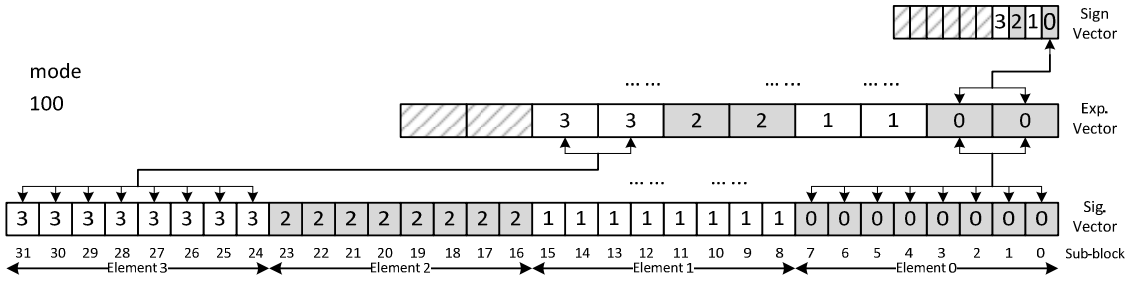

Figure 4.18. Example of mapping between vector formats.

If the mapping blocks were to be removed from the block diagram for a dynamic precision adder, the remaining structure is quite similar to that of a traditional floating-point adder, except that all the processing blocks are vectorized and provide parallel operations on multiple elements. A few intermediate control signals derived from the signal *mode* are generated during the unpacking stage to facilitate the vectorized operations. These control signals are generated, either with hardwired multiplexers or with combinational logics, every time the operation mode is changed, and are shared by many of the vectorized processing blocks throughout the operation. One of the control signals is the signal *ctrl*, which indicates the valid element boundaries among all the vector sub-block. For example, this signal helps to terminate the carry chain between two significand sub-words during the fixed-point addition. Another important control signal is the block selection signal (i.e., *bsel* or *block_sel*), which identifies the relationship of a sub-block to a specified vector elements.

Besides producing the final sign vector for the vectorized floating-point addition, the sign datapath is also responsible for determining the effective operation vector that is required by the vectorized significand addition. Similarly, in addition to calculate the final exponent vector, the exponent datapath also provides the vectorized significand barrel shifter with the number of bits that are required to be shifted in order for the radix point for both significand vectors aligned, along with a few signals that are critical to other modules. Keep in mind that the larger exponent is used for the following updating for the generation of the final result.

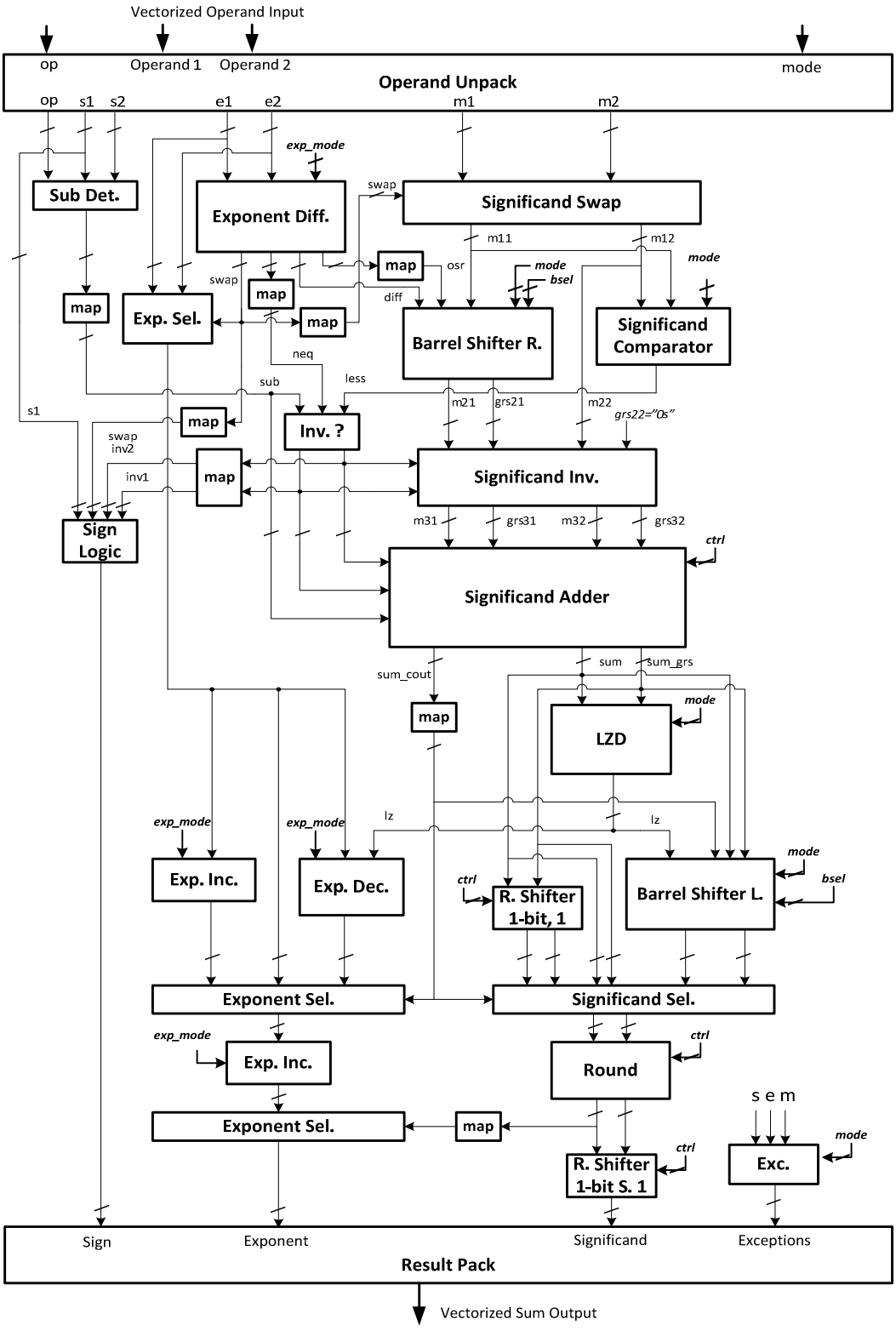

Figure 4.19. Block diagram for a dynamic precision floating-point adder.

The significand datapath, on the other hand, is a bit more complicated, compared to that for a dynamic precision floating-point multiplier. First of all, whenever necessary, the significand element pair must be swapped such that the significand sub-word with a smaller exponent is shifted a certain bits to the right by the vectorized barrel shifter. None of these special mode control signals is needed, because the swap signal after being mapped from exponent vector format has already contained requisite information. A vectorized significand comparator is implemented for testing which one in the significand sub-word pair is larger. Combined with the effective operation *sub* form the sign datapath and the non-equal vector from the exponent datapath, it is then able to obtain the enable vectors for the selectable inversion module. The generation of the enable vectors can be explained in Eq. (4.10).

$$
\begin{aligned}
inv1_n &= \begin{cases} sub_n & \text{when } neq_n = \text{'0'} \text{ and } less_n = \text{'1'} \\ \text{'0'} & \text{When others} \end{cases} \\
inv2_n &= \begin{cases} \text{'0'} & \text{when } neq_n = \text{'0'} \text{ and } less_n = \text{'1'} \\ sub_n & \text{When others} \end{cases}
\end{aligned}
\tag{4.10}
$$

Note that after the barrel shifter, a new GRS vector is introduced to accommodate the guard bit, round bit and sticky bit for each possible significand sub-words. This new vector shares the same structure as the significand one, but with a sub-block size of 3-bit. Next the significand and GRS vectors from the same operand as a whole are treated as one of the inputs for the specially designed dynamic precision fixed-point adder. Once the sum and sum_grs vectors are obtained, a vectorized leading zero detector (LZD) is employed to prepare the required number of left shift during the first normalization phase. This left shift only happens to sub-words whose carry-out bit is ‘0’, otherwise, a 1-bit

right shift is necessary. Note that, the GRS vector is required up to the rounding phase, thus can be removed thereafter.

Similar to a multiplier, the exceptions and special values defined in IEEE standards are handled by the exception block, which will not be discussed again in detail.

2) Vectorized Sign Processing

The sign vector is constructed in a fairly straightforward manner, where one bit is reserved for the sign from each operand element. Due to the simple vector format, it does not require extra control signals to manage the vectorized processing on the sign datapath, not even the *mode* signal. All the bits in a vector are processed no matter what mode the adder is operated under. The correctness of the logic is guaranteed by the zero-padding and filtering during the unpacking and packing stage respectively. In fact, doing unnecessary work is more desirable than introducing mode control mechanism to a relatively effortless logic, which costs more than it can gains. Since all the latency introduced by the sign datapath can be perfectly hided from the critical path, keeping the circuit as small as possible is the goal when designing this path.

First step of the sign datapath is to determine the effective operations for each sub-word. The effective operation for an adder that supports both addition and subtraction operations with signed operands can be obtained by feeding both signs and the *op* for an operator pair to a three-input XOR gate, as in $sub_n = op_n \otimes sign1_n \otimes sign2_n$. Fig. 4.20 shows how this is performed in parallel for multiple sub-words (bits in this case). Totally *n* copies of XORs are needed for a *n*-bit sign vector.

The next step is to find the final sign for the sum of each sub-word pair. As discussed in the previous section, the generation of the final sign requires information from the original signs as well as the magnitude comparison between two operands. Since the latter is also used to determine which one of the significands has to be inverted before performing the fixed-point addition, two derived signals, inv1 and inv2 that control the selectable invertor for significands, are used for generating the final sign. The generation process is described as in Eq. (4.11), where $swap_n$ is a signal that indicates whether a significand pair is swapped in position or not.

$$sign_n = \begin{cases} sign1_n \cdot inv2_n & \text{when } swap_n = '1' \\ sign1_n \cdot inv1_n & \text{when } swap_n = '0' \end{cases} \tag{4.11}$$

The implementation for the final sign generation is similar to that for the effective operation, where the XOR gates are replaced with the logic described in Eq. (4.11).

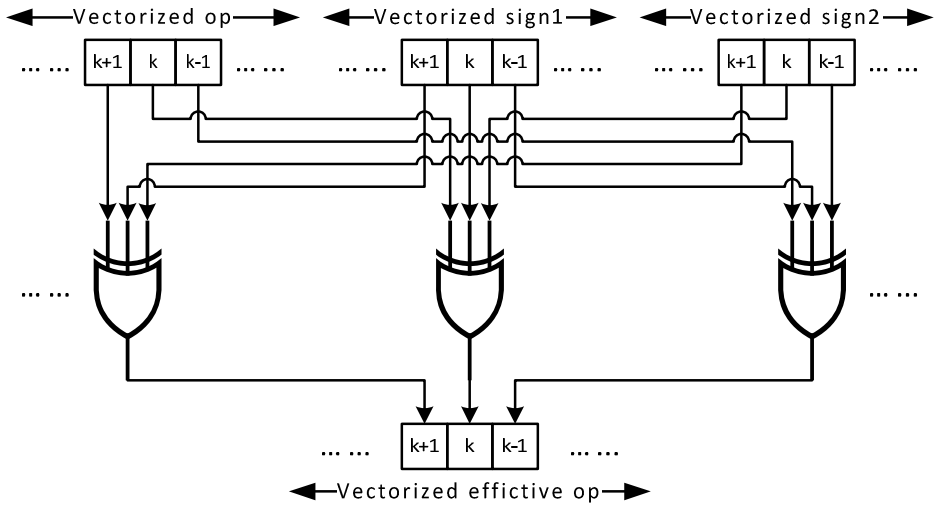

Figure 4.20. Vectorized effective operation detector.

3) Vectorized Exponent Processing

Before the significand could be shifted and aligned for further processing, the difference between the exponents of the two floating-point numbers must be provided. Hence the first task for the exponent datapath is to obtain the exponent difference. Unlike the exponent addition for a multiplier, the subtraction for an adder requires no special handling to cope with the biased format.

Since the exponent sub-word can occupy one or two exponent sub-blocks depending on the operation mode, a dual mode adder is designed to perform the subtraction. This dual mode adder is similar to the exponent dual mode adder used in the dynamic precision floating-point multiplier shown in Fig. 4.6, except that for this application one of the exponents is inverted. As there is no certainty of which of the exponent holds the larger value, it is possible that the output is negative that requires conversion before used by the shifter. This conversion is basically a slow and expensive process of adding one to the bit-wise inverted input. Instead of doing this, two individual dual mode adders are implemented, providing subtraction for *exp1-exp2* and *exp2-exp1* respectively. Then the positive output is selected as the difference between the exponents, and at the same time signal *swap* required for the significand swapper can be also obtained. A slice of the vectorized exponent difference module is shown in Fig. 4.21. One slice of this implementation is required for every two exponent sub-blocks. Thus the exponent mode signal (*exp_mode*) used for the dual mode adder is also used to construct the final outputs of several signals, such as the *swap* and *diff*.

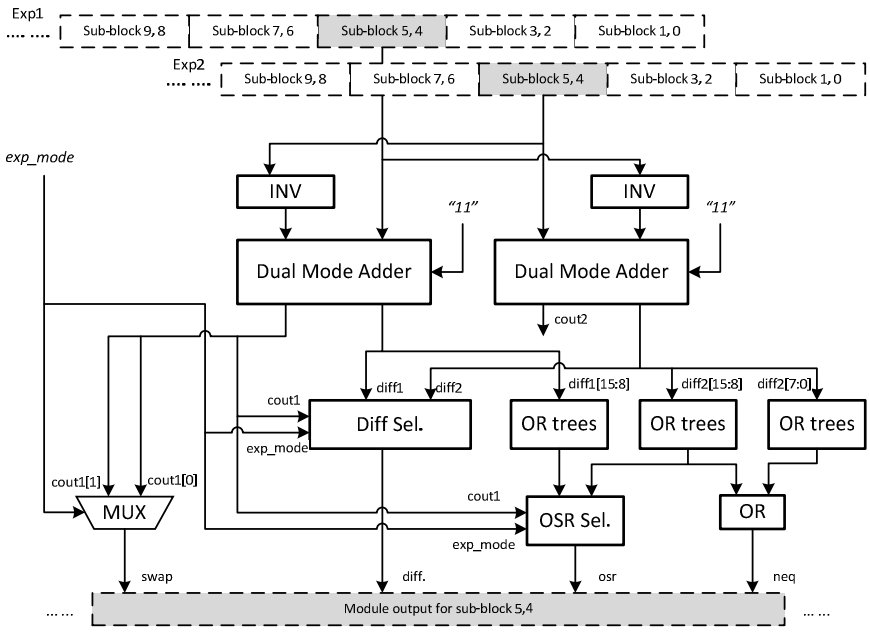

Figure 4.21. Vectorized exponent difference module.

Moreover, in order to facilitate the magnitude comparison for significands, a non-equal signal is also generated, by examining all the *diff* bits with OR trees. Depending on how the bit-width for significand is setup, a signal that indicates the exponent difference is out of the shifter range might become necessary. For example, the maximum number of bits the shifter supports is 202 for mode "111" under the eight-mode scheme. Only the lower 8-bit of the exponent difference is necessary for the shifter to operate properly. Any difference represented in more than 8-bit will result in incorrect shift. Therefore, a signal named *osr* (out-of-shifter-range) is used to monitor if any of the higher bits is non-zero, and is then use by the shifter to produce correct result.

The shared exponent vector needs to be update twice at most during the two normalization stages. For the purpose of not putting more negative impact on the critical

path, the updating processes are always performed in advance, and then the required one is selected once the decision is made. By doing this, it is possible to hide the extract latency injected into the critical path as much as possible. For example during the first normalization, both the exponent increment and decrement modules start calculating as long as their depending inputs are ready. These operations are in parallel with those shifters but faster, avoiding worsening the critical path. A specially designed dual mode increment module utilizing the carry-select technique, shown in Fig. 4.22, is implemented to improve the speed for the final vectorized exponent increment during the second normalization. Combined with the mapping block, this module is expected to introduce latency to the critical path. Hence, an improved version of the vectorized increment module becomes significantly desired.

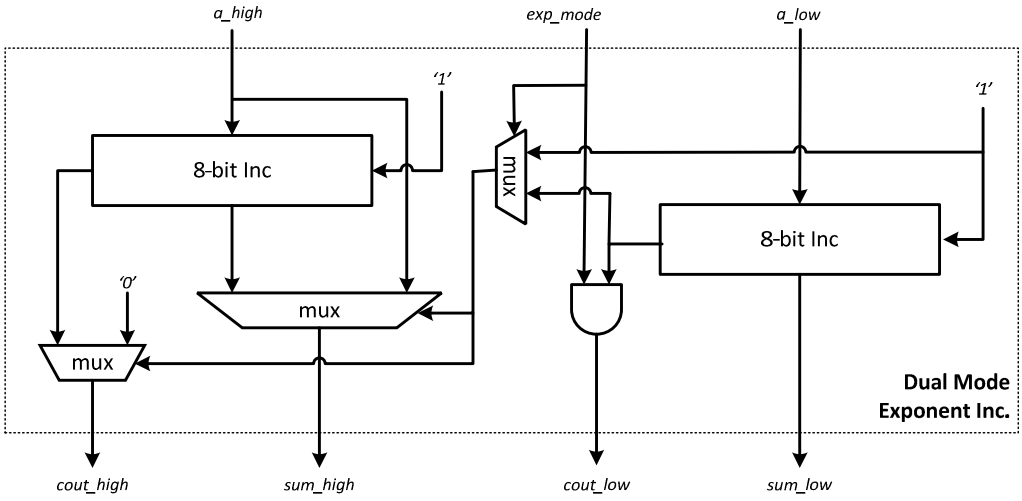

Figure 4.22. Dual mode exponent increment module.

4) Vectorized Significand Comparator and LZD

A vectorized significand comparator is developed to compare the magnitude of sub-words from two significand vectors. A detail diagram is depicted in Fig. 4.23, showing how a multi-level comparator tree is constructed to enable the possibility of finding the corresponding comparison result for any possible element within a vector format. This structure shares the same basic idea of the generation of allzeros signal for the exception processing that is illustrated in Fig. 4.5, replacing the OR gates with two different modules accordingly.

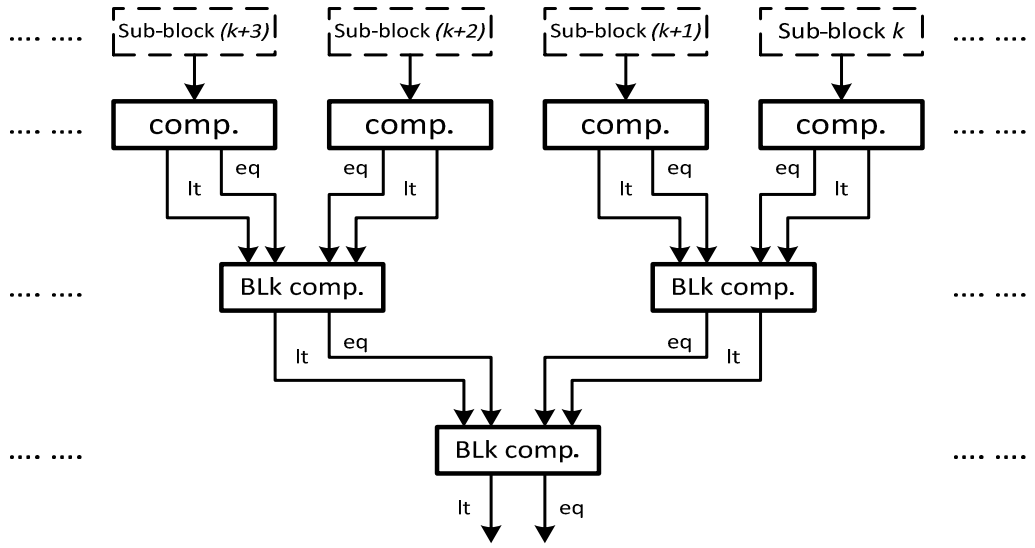

Figure 4.23. Vectorized significand comparator.

At first, a sub-block comparator is employed for each of the significand sub-block pairs responsible for generating the *eq* and *lt* signals. The value of the *eq* signal indicates whether the two inputs are identical or not, while the value of *lt* signal being '1' suggests

that the first input is less than the second one. Once the sub-block comparisons are established, a multi-level block-wise comparison tree is built on top of that. The key processing unit for this tree is a block comparator that takes in the comparison results from two consecutive blocks from the previous level and generates the result for this new combined block. The structure for this block comparator is illustrated in Fig. 4.24. Similar to that for the allzeros tree, exact results for some regularly constructed sub-words can be obtained directly, while intermediate comparisons used to produce the final comparison for those irregularly constructed significand sub-words can be extracted from nodes on the tree and combined. Thus a complete vectorized significand comparison can be performed.

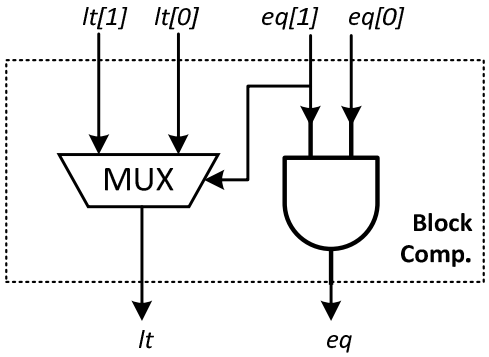

Figure 4.24. Block comparator module.

The vectorized leading zeros detector required for the first normalization stage is quite similar to the comparator in terms of the tree structure. It has a series of basic LZD blocks on the first level to determine the number of leading zeros in a sub-block, then a level of block-wise LZD modules are employed to generate the leading zeros information

for the corresponding consecutive block pairs from the previous level. This tree continues until it reached the final level, where the number of leading zeros of the whole vector is found. A complete diagram of the proposed vectorized LZD is illustrated in Fig. 4.25.

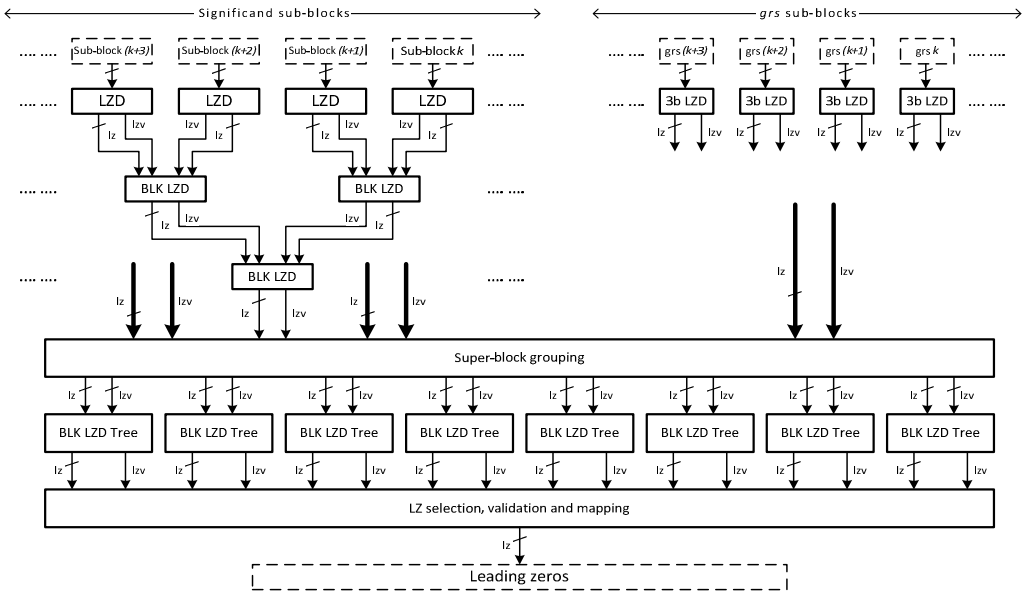

Figure 4.25. A complete vectorized LZD module for adders.

A LZD tree similar to the comparator tree is employed to process the significand vector, but this time none of the nodes could produce direct results due to the introduce of a separated GRS vector. A complete sub-word to be processed by the LZD consists of sub-blocks from the significand vector plus one from the GRS vector. Therefore, one level of basic 3-bit LZDs is enough for the GRS vector. The basic concept of leading zero detection is discussed in detail by the author in [45]. An implementation of the basic LZD for 4-bit sub-block size under the eight-mode scheme is depicted in Fig 4.26(a), and a 3-bit LZD specially designed for the GRS sub-blocks is illustrated in Fig 4.26(b). A LZD

counts the leading zeros in an input bit stream, and then outputs the counter using an *lz* signal. At the same time a 1-bit validating signal called *lzv* is generated as long as the input is not all zeros. With these two signals from the previous level, a block-wise LZD module can produce the *lz* and *lzv* for current level. The underling logic for the block-wise LZD can be explained in Eq. 4.12.

$$
\begin{aligned}
lzv_{k+1,n} &= lzv_{k,2n+1} \text{ OR } lzv_{k,2n} \\
lz_{k+1} &= \begin{cases} \{\ '0'\ ,\ lz_{k,2n+1}\ \} & \text{when } lzv_{k,2n+1} = '1' \\ \{\ '1'\ ,\ lz_{k,2n}\ \} & \text{when } lzv_{k,2n+1} = '0' \end{cases}
\end{aligned} \tag{4.12}
$$

Note that the bit-width for the *lz* signal will grow by one bit for each level, which is different than all the other trees discussed.

Once the intermediate data are obtained, a super-block grouping process is then performed to collect the corresponding segments for a complete sub-word from the significand LZD tree and the GRS LZD blocks. And then multiple copies of a smaller block-level LZD tree (up to 3-level for eight-mode scheme) are employed to generate the output candidates for different operational modes, the correct one of which is then selected, validated and mapped to the exponent vector format for the updating of exponent vector.

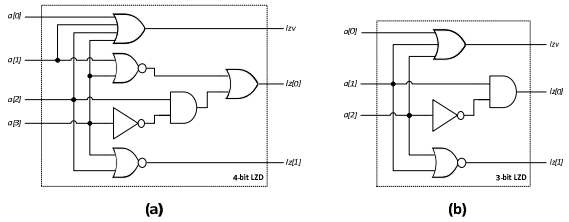

Figure 4.26. Sub-block LZD modules. (a) 4-bit LZD module; (b) 3-bit LZD module.

5) Vectorized Barrel Shifter

A vectorized barrel shifter is able to shift all the sub-words within a significand vector according to their individually specified shift requirement under different operational modes, such that the radix points for the sub-word pair are aligned for further processing. A typical barrel shifter is constructed with multi-level cascaded row shifters, each of which is responsible for shifting 2^k bits, where the k is usually the index of the level. For example, the first level row shifter shifts the input by one bit, and the second level shifter shifts the output of the first level by two bits, assuming both shifter are enabled. The number of row shifter levels is determined by $\lceil log_2(n) \rceil$, where n is the width of the input.

The block diagram for a vectorized barrel shifter is shown in Fig. 4.27. First of all, the exponent difference vector is mapped to significand format in order to be used by the row shifters. Then, the difference bits are regrouped into several smaller vectors, each of which is distributed to its corresponding vectorized row shifter as the level select vector. Furthermore, the out of shifter range vector (*osr*) also generated by the exponent difference module is used to prevent incorrectly shifted data being produced. For the purpose of preserving precision during the addition, a GRS vector is first introduced in the significand datapath. Each of the sub-word has a 3-bit GRS extension block to hold the information of the out-shifted bits. The first two bits serve as a regular 2-bit extension to the significand, while the last bit named sticky is used to indicate the existence of any shifted-out information. And the vectorized processing of the row shifter is controlled by the block select signal.

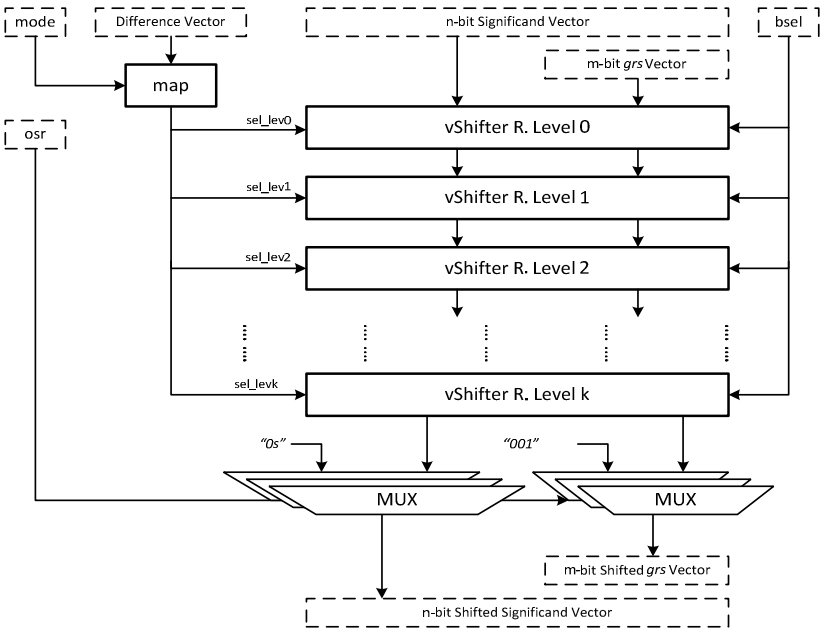

Figure 4.27. Vectorized barrel shifter.

Two variants of the vectorized barrel shifter are implemented for different purposes. A right barrel shifter is responsible for the pre-shift that aligns the radix point for the addition, and a left barrel shifter is implemented during the first normalization to remove all the leading zeros from all the sum sub-words. The only difference between the two variants is the row shifters that in charge of the actual shifting. These row shifters must be able to provide complicated vectorized shift that involves two interrelated vectors. Fig. 4.28(a) and (b) depict the shift traffic traveling between the two vectors, for right shift and left shift respectively. Take the left shift as an example. The source for a significand sub-block is either its previous sub-block when they are from the same sub-word, or zeros if the supposed shifted bits are from another sub-word or out of the vector range. And there are also two possible sources for the GRS sub-block, specially a previous

significand sub-block and zeros. The same idea applied to the left shift, except that it is reversed.

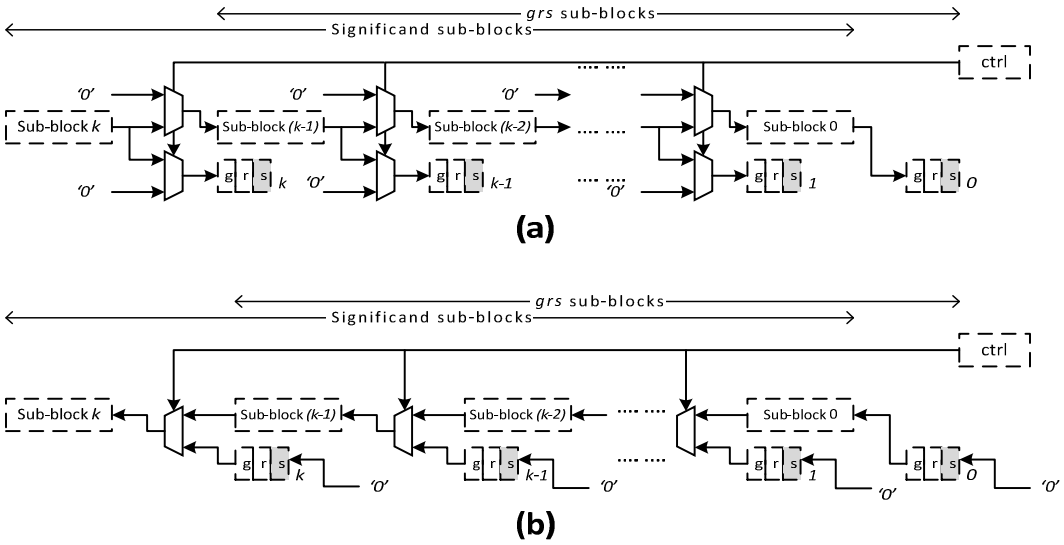

Figure 4.28. Vectorized row shift. (a) Right shift; (b) Left shift.

Now, it becomes clear that this complicated shift could be treated as two independent generating processes, one for the significand sub-blocks and the other for the GRS sub-blocks. Fig. 4.29 depicts the implementation detail for the significand vector. First, the significand vector is logically shifted by a specified position. Zeros are inserted to the most significant bits. Then the *bsel* signal is employed to validate the shift-in bits through a multiplexer for each sub-block. Only the shift-in bits from the same significand element will be selected. And finally, another series of multiplexers are employed to select the valid shifted significand or the origin, controlled by the select signal derived from the exponent difference vector.

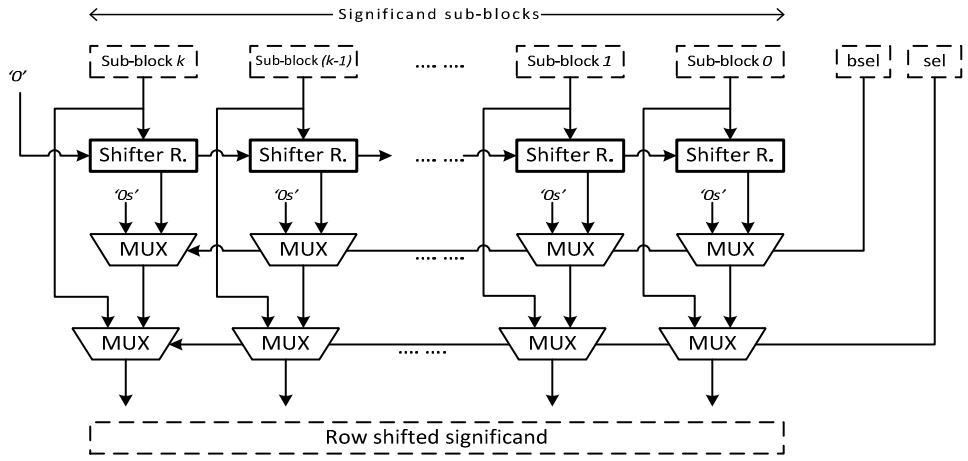

Figure 4.29. Vectorized row shifter for significand sub-blocks.

Within a GRS sub-block, the generation of the new guard and round bits are quite similar to that for a significand sub-block, except that the original GRS sub-block must be taken into account when performing the first unconditional shift. However, the implementation for the generation of the sticky bits does not follow the same steps due to the nonlinear nature of sticky bits. A sticky bit will be set as long as there is any valid data shifted out the extended significand range (significand plus GRS). A general implementation of the row shifter for sticky bits is illustrated in Fig. 4.30. First of all, OR trees are employed to test all the significand sub-blocks for allzeros, followed by the validation via multiplexers controlled by the block select signal just like the cases discussed above. Then, the allzeros signal for all the significand bits that are supposed to be shifted out of the extended range is produced through another OR tree. By combining it with the allzeros from the GRS sub-block, the new sticky bit is obtained. When the required number of shift is less than 4, the left portion of the diagram can be omitted, resulting in a smaller design.

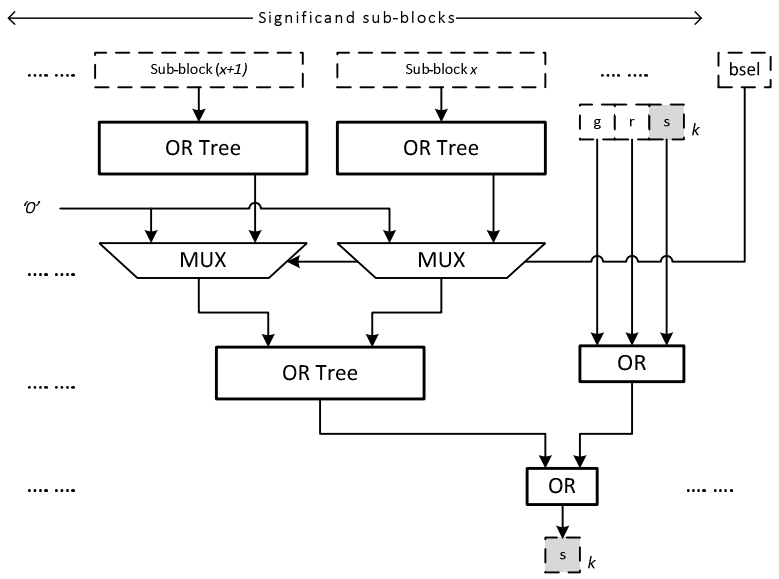

Figure 4.30. General implementation of a vectorized row shift for sticky bits.

6) Vectorized Significand Adder

A specially designed dynamic precision adder is the core component that actually performs the addition on the pre-processed significand vectors and the newly introduced GRS vectors. In fact, there are several advantages when separating the GRS blocks from the regular significand blocks. By doing this, not only the original structure of the significand vector is preserved, but also the newly established GRS vector is able to be constructed in an identical manner with a different sub-block size. Any complicated operation involving the two vectors can be handled relatively easy by using divide and conquer technique. Also, it allows the same strategies for tackling dynamic precision from the previous designs to be applied to the two vectors individually. And finally, the *ctrl* signal that indicates the sub-word foundries can be used to link the separated processes to generate the final results.

The basic idea of a dynamic precision adder is illustrated in Fig. 4.31. It is worth to point out that this basic idea can be applied using any of the dynamic precision adder architectures discussed in the previous chapter for a faster implementation. The serially connected multiplexers shown in the figure are provided only to simplify the general demonstration.

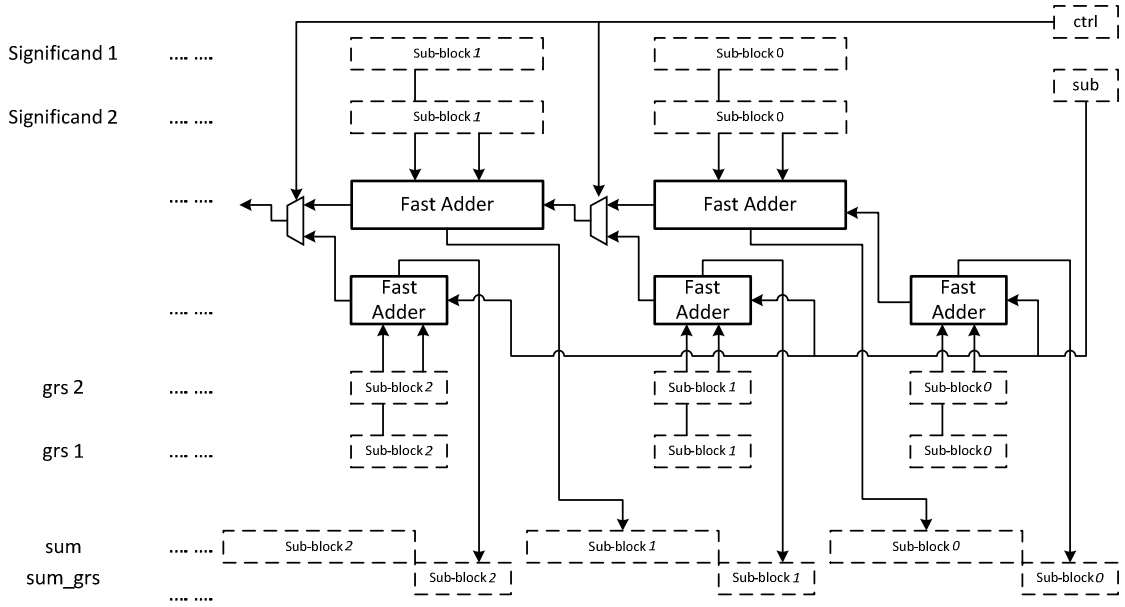

Figure 4.31. Dynamic precision adder for significand and grs vectors.

On one hand, the sub-blocks from the significand vectors are added using larger fast adder. On the other hand, a series of fast adders are employed to calculate the sub-block sum for the GRS vectors. No carry between these sub-blocks are required, as there is only one GRS sub-block for each significand sub-word. The effective operation signal is used to provide the carry-in for the adders if necessary. Then the carry-outs from the GRS

adders are injected into the significand carry chain through multiplexers controlled by the *ctrl* signal. This guarantees no carries are carried across sub-word boundaries, and the correct carries from the corresponding GRS sub-blocks are transferred to the right significand segments. The outputs are saved in the same two-vector format, thus allowing further processing follows the same strategies. The final sub-word carry-outs, which are used for the normalization, are determined by the mapped effective operation and the enable vectors for the significand inversions. Note that this process is not shown in the diagram.

D. Floating-point Fused-Multiply-Add Architectures

A fused-multiply-add is the hardware implementation that performs the computation of $p = a \cdot x + b$ on floating-point numbers, which is essential to several computation applications, including vector dot product and polynomial evaluation [14]. Although this operation can be carried out by performing floating-point multiplication followed by an addition via two independent ALUs, a fused implementation of the two serial units required only one operation is desirable in many ways. First of all, just like its fixed-point counterpart (MAC), a fused-multiply-add unit can significantly reduce the timing and area requirements for the FMA operation by simplifying, merging, and parallelizing parts of the independent multiplication and addition processes. Another advantage of a fused-multiply-add is that it preserves the precision lost by combining the two rounding processes into one. Besides, a fused-multiply-add can also be employed to perform standardized floating-point multiplication or addition if necessary, providing more flexibility for processor designers when specifying the microarchitecture. That means a

FMA unit can be treated as a duplicated multiplier or adder for many purposes like fault tolerance or data parallelism. It can also be used to replace the multiplier and adder completely for some specific fields. Finally, a fused unit generally helps to reduce the program size due to the fact that only one instruction is required for a series of consecutive operations. Especially for applications that involve repeated operations of the fused operation, the memory access for instruction fetches can also be significantly reduced, thus leaving doors opened for many performance improving opportunities.

The block diagram of a dynamic precision fused-multiply-add unit is illustrated in Fig. 4.32. The architecture of the dynamic precision FMA is generally similar to that of an adder, especially for the sign and exponent datapaths, but with substantial modifications underneath modules within the significand datapath. The fundamental difference between a FMA and an adder is how the significands are handled before the actual addition, which in fact is the root cause responsible for most of the block modifications. This time only the significand form the additive operand is to be shifted for the radix point alignment, and/or to be inverted for the effective subtracting operation. Therefore, significand swapping and comparing modules are no longer necessary, simplifying the sign logic and the significand pre-processing steps. By designating the pre-processing only to the additive significand and decoupling it from the multiplication, it is possible for both paths being implemented in a parallel manner, thus minimizing the impact on the critical path when fusing the two operators. Note the partial product generator and reduction tree from the dynamic precision multiplier can be reused without any changes.

However, nothing comes without a cost. First of all, designating the pre-processing to a specific significand complicates the shifting process. For an adder, the difference used for the shifter is always positive, meaning that the shifter only needs to shift the significand with smaller exponent to the right. In contrast, a FMA must support shifting the radix point to both directions for the alignment due to the fact that the difference between the additive significand and the product significands can be positive or negative. A problem of the dual-directional shifting scheme is that it makes the LZD and normalization way more complicated. A common practice is to initially pre-shift the additive significand a certain bits to the left before proceeding the radix alignment. By doing this, most of the rest processing remains the same but with a wider bit-width. In order to preserve the precision before the final rounding, a *(3n+2)*-bit shifter (instead of *(n+3)*-bit one are now required for an *n*-bit additive significand. Adapting the same strategy as in the adder, a completed shifted significand is separated in to three sections, each of which is allocated in its designated vector in the original significand format, namely, an *n*-bit high vector, a $(2 \cdot \#sub\_block)$-bit save vector, and a *2n*-bit low vector. By doing this, the addition with the product can be directly applied for the low vector and the *s* and *c* product vectors. Moreover, a complement operation is necessary after the significand addition, because it is possible for the sum to be a negative number, which must be converted before the post-processing. The LZD module must also be modified to handle the 3-segment scheme, while the post-processing modules after the first normalization remain the same.

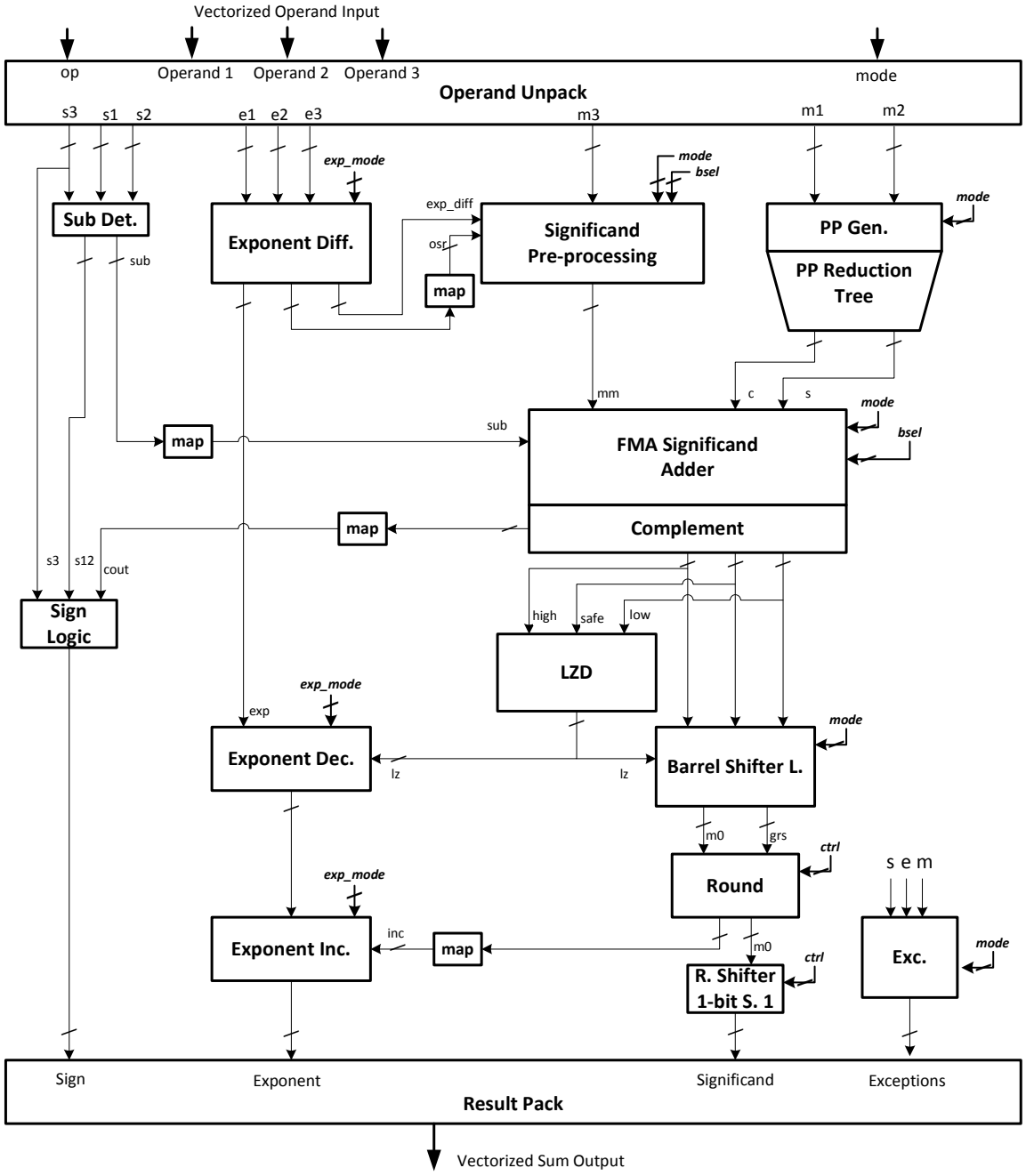

Figure 4.32. Block diagram for a dynamic precision FMA unit.

1) Vectorized Radix Alignment for FMA Units

The pre-shift process is illustrated in Fig. 4.33 to better understand how the radix alignment is realized for a FMA via a *(3n+2)*-bit shifter, where the sticky bit is not included. Suppose that the exponents for the multiplication operands are exp_1 and exp_1 respectively, and the exponent for the additive operand is exp_3. All exponents are in biased format. Then the exponent of the multiplication product is $exp1 + exp2 - eBias$, where $eBias$ is the bias of the exponent format. On the other hand, the additive significand is initially pre-shifted *iBias*-bit to the left, where $iBias = n + 3$.

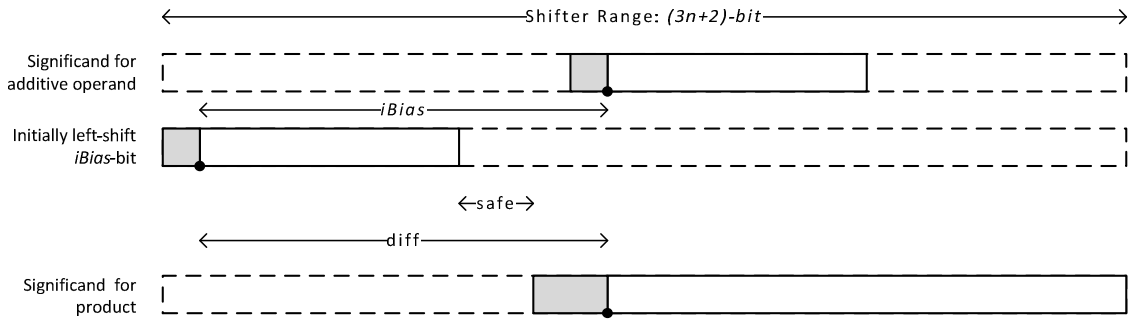

Figure 4.33. Radix point alignment process for FMA operations.

In that case, the number of positions that the additive significand is required to shift to the right to align its radix point with the product is determined by Eq. (4.13).

$$\begin{aligned} diff &= (\exp_1 + \exp_2 - eBias) - (\exp_3 - iBias) \\ &= (\exp_1 + \exp_2) - [\exp_3 + (eBias - iBias)] \\ &= (\exp_1 + \exp_2) - (\exp_3 + preBias) \end{aligned} \tag{4.13}$$

Since both the *eBias* and *iBias* are known and fixed, they can be combined into one constant, *preBias*. By re-arranging the computation of difference as shown in Eq. (4.13), the whole calculation can be performed through two levels of additions instead three, thus avoiding worsening latency of the critical path. The right-shift only happens when the difference are positive value. Otherwise, there is no necessary for any shifting for the additive significand, and this is guaranteed by the *2*-bit safe block, placed in between the un-shifted significands as a buffer zone. And the exponent for the aligned radix point is determined by Eq. (4.14).

$$\exp = \exp_1 + \exp_2 - preBias \tag{4.14}$$

Therefore totally four adders are required for the exponent difference module.

A slice of the vectorized exponent difference module is depicted in Fig. 4.34, showing how four dual mode adders are employed for the calculation of the difference and shared exponent vector. The *preBias* values are determined once the precision configurations are defined, thus they are implemented as constants. The first two dual mode adders are responsible for the computation of $exp_1 + exp_2$ and $exp_3 + preBias$ respectively. And two dual mode adders in the second level are employed to calculate the temporary difference and shared exponent vector. Whenever a sub-word difference is positive, the temporary *diff* and *exp* element are valid. Otherwise, this *diff* is set to zeros meaning no shift is necessary, and exp_3 becomes the shared exponent for the corresponding sub-word. Therefore, a final selection and construction of these two vectors is required for the outputs. Similar to the dynamic precision adder, the *osr* signal

that indicates the required shift being out of shifter's range is generated using OR trees. Because the shifter supports up to 386 bits instead of 202 for mode "111" under the eight-mode scheme, only the lower 9 bits of the exponent difference are considered valid for a FMA. Any number expressed with more than 9 bits will be treated as out of range.

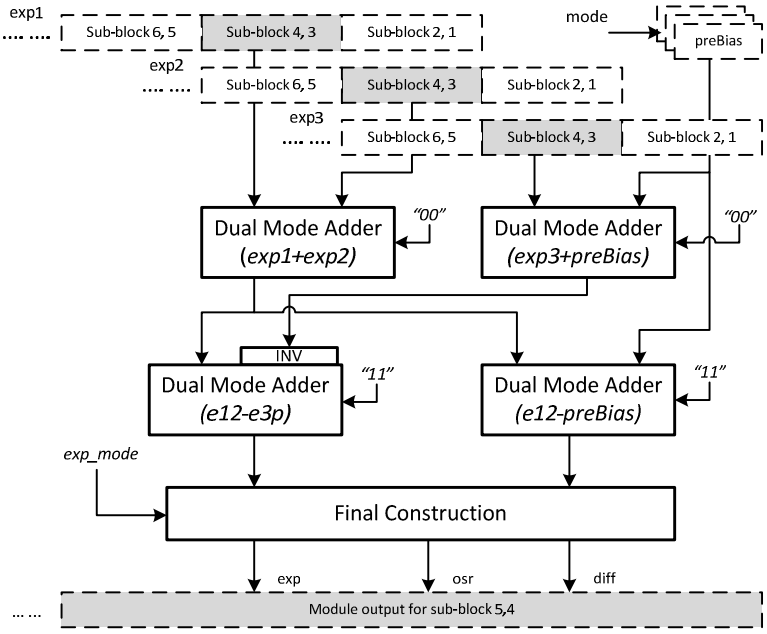

Figure 4.34. Vectorized exponent difference module for FMA units.

Once the exponent difference is determined and mapped to the significand vector format, the right shift can be performed with a vectorized barrel right shifter. The challenge posted by the fact that the input is represented in three vectors, is the uncertainty of the source for a particular low sub-block. Four examples of the right-shift for a simplified dynamic precision FMA is shown in Fig. 4.35. It is observable that the source of the shift-in data for the most significant "low" sub-block changes according to

the operation mode. In example (a), the most significant “high” sub-block is shifted to the most significant “low” sub-block via the most significant “safe” sub-block. But it is the least significant “high” sub-block shifted via the least significant “safe” sub-block to the same location in example (d).

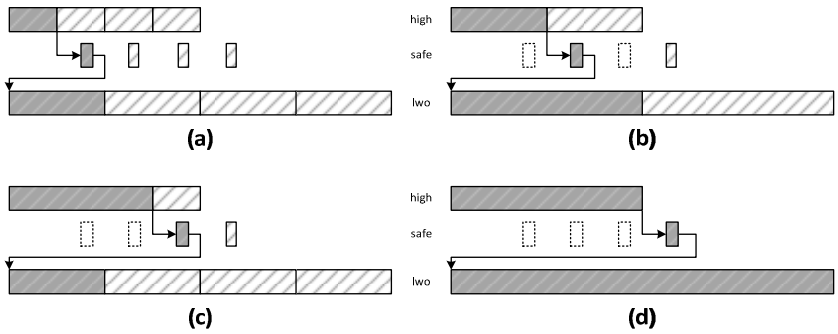

Figure 4.35. Examples of vectorized right-shift for FMA operations.

The mode dependent behavior of the shifter complicates the design of the vectorized row shifter, although the structure of the barrel shifter remains unchanged but with 9-level of row shifters. The basic idea of a vectorized row right shifter is presented in Fig. 4.36. The major difference between this version and that for the adder is the mapping process between the two versions of “safe” sub-blocks. By doing this, the source of a certain “safe A” sub-block are always determined as the “high” sub-block with the same index. And the same idea applies to the “low” vector. The mapping module is responsible for connecting a “safe B” sub-block with its corresponding “safe A” sub-block according to the operation mode, just like the mapping blocks of the dynamic precision adder discussed above. The left-shifter used for the first normalization shares the exact idea.

Nonetheless, for a FMA design with smaller number of precision modes, the row shifters can be implemented with pre-determined hardwired multiplexers to simplify the shifter's structure.

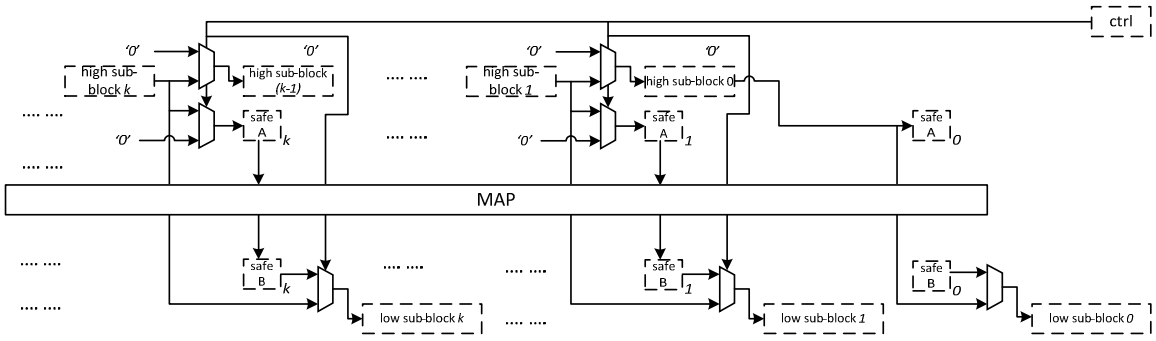

Figure 4.36. Vectorized row right-shifter for FMA operations.

2) Vectorized Significand Addition for FMA Units

The vectorized significand addition includes several steps. First of all, the three vectors representing the shifted significand are inverted if the effective operation is determined to be subtraction. In fact, the way the shifted additive significand separated into three vectors in the same format allows a fairly simple and fast implementation for following addition operations. The lower portion of the sum can be obtained by adding the "low" vector to the two product vectors (in carry-save form) with a row of 3-to-2 blocks followed by a *2n*-bit dynamic precision adder. Due to the possibility of carries generated or propagated between sub-word boundaries, the row of 3-to-2 blocks must be able to terminate the invalid carries controlled by the *ctrl* signal. At the same time, an

unconditional increment operation is performed on both the “high” and “safe” vectors. Note that only the valid “safe” sub-blocks are processed within the module to prevent unnecessary circuit switching. Then, with the help of the *ctrl* signal, the carries generated from the “low” and “safe” operations are used to select the correct outputs for the construction of the sum’s “safe” and “high” vectors respectively. The mechanism of transferring carries between the three vectors is the same as discussed for the dynamic precision adder. Given the fact that the sum can be negative and the significand output must be positive, it must go through the conditional complement block before it can be outputted. The implementation of the vectorized complement module is similar to the top half shown in Fig. 4.37, except that the dynamic precision adder for the “low” vector is replaced with a dynamic precision increment module.

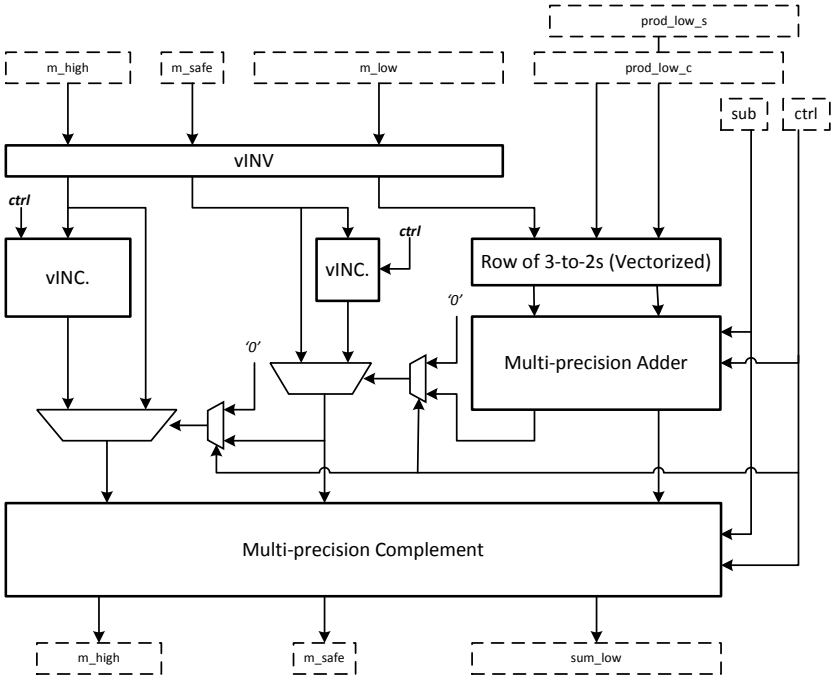

Figure 4.37. Vectorized significand adder for FMA units.

3) Vectorized LZD for FMA Units

The block diagram of a vectorized LZD for FMA units is illustrated in Fig. 4.38. The process of finding leading zeros for the combined "high" and "safe" vectors, shown in the upper-left of Fig. 4.38, is almost the same as that for the dynamic precision adder, except that the "safe" vector consists of 2-bit sub-blocks. On the other hand, a large vectorized LZD tree is employed to calculate the leading zeros for the "low" vector. As the sticky is not necessary for the *lz_low* generation, the structure is slightly simpler.

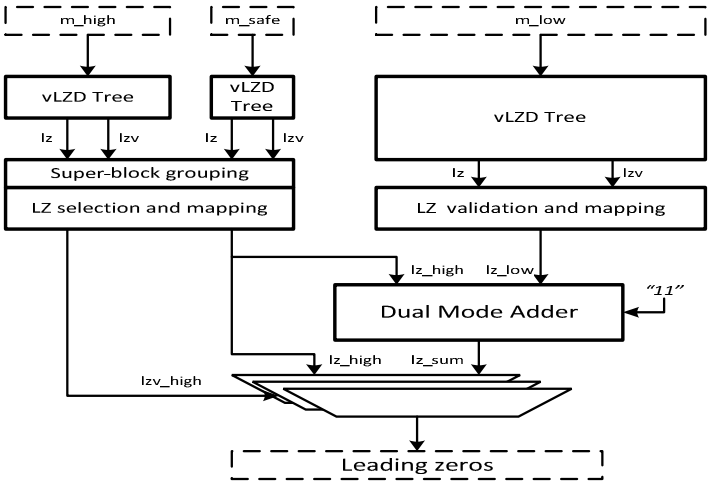

Figure 4.38. Vectorized LZD for FMA units.

By the time both LZD threes are finished, the leading zeros vectors are grouped, selected, and then mapped to the exponent vector format for the final addition. Because the bit-with of the *lz_high* is smaller than that of the *lz_low* , the final lead zero can only be obtained with a dual mode adder instead of the block-wise LZD module. The process of the second step for generating the final outputs is explained in Eq. (4.15),

$$lzv_i = lzv_{i,high} \text{ OR } lzv_{i,low}$$
$$lz_i = \begin{cases} lz_{i,high} & \text{when } lzv_{i,high} = '1' \\ lz_{i,high} + lz_{i,low} + 1 & \text{when } lzv_{i,high} = '0' \end{cases} \quad (4.15)$$

where *i* is the sub-word index. The fixed carry-in of "1" for the dual mode addition is to accommodate the extra one zero that is not representable in the *lz_high* when the two parts are supposed to be combined.

In Chapter III and Chapter IV, we present several proposed design techniques and architectures for adder, multiplier, and fused-multiply-add (MAC and FMA) units supporting dynamically defined precision for both fixe-point and floating-point operations. These approaches are examined and evaluated through actual hardware implementation, and the analysis result will be presented in the next chapter.

CHAPTER V

IMPLEMENTATION AND ANALYSIS

The proposed mechanism for supporting dynamically defined precision demands additional hardware resources and introduces extra latency in the critical path of ALU architectures. In order to investigate the area and latency impact imposed by our dynamic precision architecture on the hardware implementations of variant architectures, two implementations were developed for the popular fixed-point and floating-point ALUs, the results from both versions were later compared and analyzed. A golden version created with the traditional architecture that only supports operations with one precision was used as the baseline design, then another version based on the same algorithm was developed using our proposed architecture to support dynamically defined precision. For fair comparison, both versions were implemented with the same level of effort and optimization.

All the implementations were created using VHDL with parameterizable bit-width for operands and sub-blocks. All the designs were extensively simulated with uniformly distributed random number operands, and the accuracy was verified by comparing the results with those generated by the MATLAB fixed-point toolbox and Perl programs. The same set of stimuli was also used for the verification of the post-synthesis and post-layout netlists.

A. Fixed-point Adders

We have implemented two versions of the following fixed-point adders: (1) ripple-carry adder (*RCA*) with small footprint and linear delay; (2) single-level carry-lookahead adder (*CLA*); (3) multiple-level carry-lookahead adder (*CLA_net*); (4) Brent-Kung prefix adder; (5) Kogge-Stone prefix adder; and (6) Hybrid prefix adder with a moderate latency-area tradeoff. All the adder designs were synthesized with Xilinx ISE 13.4 for the Xilinx Virtex-5 XC5VLX110T FPGAs.

Table 5.1 and Table 5.2 show the area and latency information for the traditional version of all the adder implementations. The results for highly optimized Xilinx proprietary adder macros are also listed for reference. It is obvious that our implementations cannot outperform the highly optimized Xilinx IP cores. As the results suggested, the Kogge-Stone adder has the lowest delay, while the ripple-carry adder uses the least FPGA LUTs, which is expected given that it has the simplest structure.

The latency and area results, normalized to that of the traditional version, for certain adder implementations are listed in Table 5.3 and Table 5.4 to illustrate the impact imposed by the dynamic precision supporting circuits. The maximum latency and area overheads range from 10% to 24%, depending on the actual adder architecture. It can be observed that the prefix adders suffer less in terms of both latency and area in general.

Please note that our implementations were designed with gate-level optimizations. Therefore, our design could not take fully advantage of the Xilinx's highly optimized primitive of fast carry-lookahead logic, and more importantly the enhanced DSP48E hard

Table 5.1. Latency for fixed-point adder implementations (FPGA).

	Data Width					**Delay: ns**
Adder	4	8	16	32	64	128
Xilinx_IP	1.92	2.03	2.23	2.65	3.48	5.15
RCA	3.02	5.33	9.97	19.23	37.76	74.81
CLA	3.19	3.79	4.41	6.45	11.70	17.27
CLA_net	3.22	5.04	5.63	7.52	8.20	10.10
B-K Prefix	3.69	4.96	6.24	7.49	8.77	10.05
K-S Prefix	3.69	4.47	5.15	5.83	6.51	7.20
Hybrid	3.69	4.94	5.72	6.40	7.08	7.76

Table 5.2. Area for fixed-point adder implementations (FPGA).

	Data Width					**Area: LUTs**
Adder	4	8	16	32	64	128
Xilinx_IP	4	8	16	32	64	128
RCA	8	16	32	64	128	256
CLA	16	39	88	154	300	707
CLA_net	20	48	107	227	470	958
B-K Prefix	24	54	116	242	496	1006
K-S Prefix	26	66	162	386	898	2050
Hybrid	24	56	128	288	640	1408

Table 5.3. Normalized latency for fixed-point adder implementations (FPGA).

Adder	**Block Size**	Data Width				**Normalized Latency %**
		8	**16**	**32**	**64**	**128**
RCA	**4**	1.11	1.17	1.21	1.23	1.24
	8		1.06	1.09	1.11	1.12
	16			1.03	1.05	1.05
	32				1.02	1.02
	64					1.01
CLA_NET	**4**	1.08	1.17	1.14	1.13	1.10
	8		1.07	1.11	1.10	1.08
	16			1.00	1.06	1.06
	32				1.00	1.06
	64					1.00
Hybrid	**4**	1.07	1.15	1.15	1.14	1.13
	8		1.03	1.12	1.12	1.11
	16			1.03	1.10	1.10
	32				1.01	1.08
	64					1.00

Table 5.4. Normalized area for fixed-point adder implementations (FPGA).

Adder	Block Size	Data Width				Normalized Area %
		8	16	32	64	128
RCA	4	1.13	1.19	1.22	1.23	1.24
	8		1.06	1.09	1.11	1.12
	16			1.03	1.05	1.05
	32				1.02	1.02
	64					1.01
CLA_NET	4	1.08	1.11	1.12	1.13	1.13
	8		1.04	1.05	1.06	1.06
	16			1.02	1.03	1.03
	32				1.01	1.01
	64					1.00
Hybrid	4	1.07	1.09	1.10	1.09	1.09
	8		1.03	1.04	1.04	1.04
	16			1.01	1.02	1.02
	32				1.01	1.01
	64					1.00

macros that enable even higher performance for additions and multiplications [41]. So, to better investigate the actual impacts when gate-level implementations are applied (e.g., ASIC design), we also synthesize our designs with Synopsys Design Compiler for the FreePDK standard cell library of 45 nm CMOS technology. Fig. 5.1 and Fig. 5.2 give an overview on the latency and area information from the ASIC implementations of all the fixed-point adders we designed. The lower numbers in the X-axis are the bit-widths for the operands, and the upper numbers are the size of the sub-blocks that enables the dynamic precision operations as described in the previous chapter. For better comparison, the results for traditional designs are listed as designs using the operand bit-width as the sub-block bit-width. The results shown are used for showing the relative performance between different architectures and different precision configurations, thus none of the synthesis optimizations were activated.

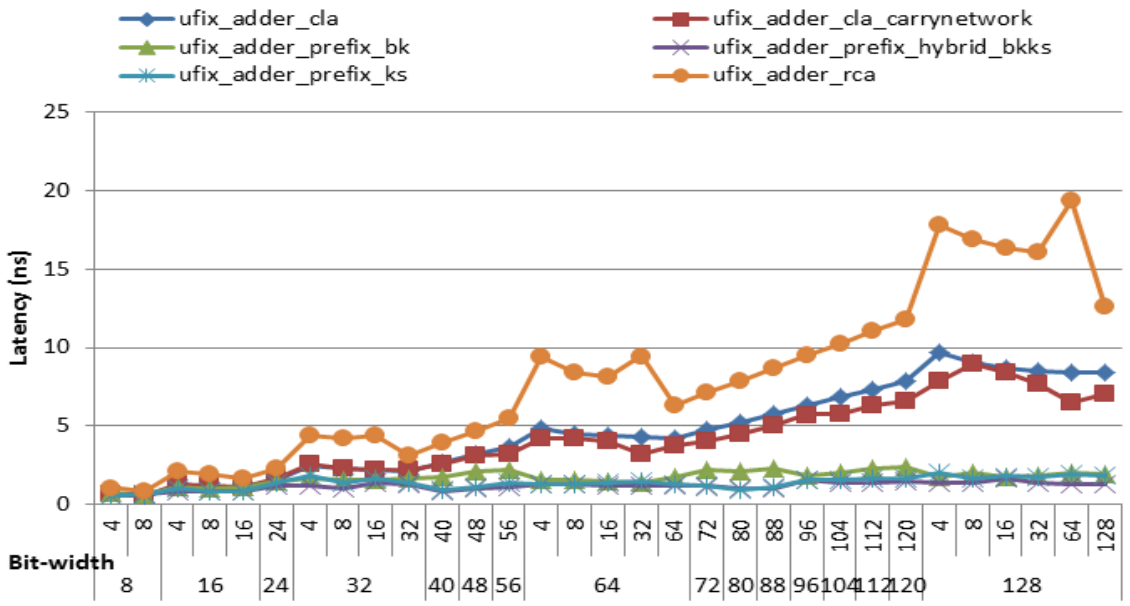

Figure 5.1. Latency for ASIC implementation of fixed-point adders.

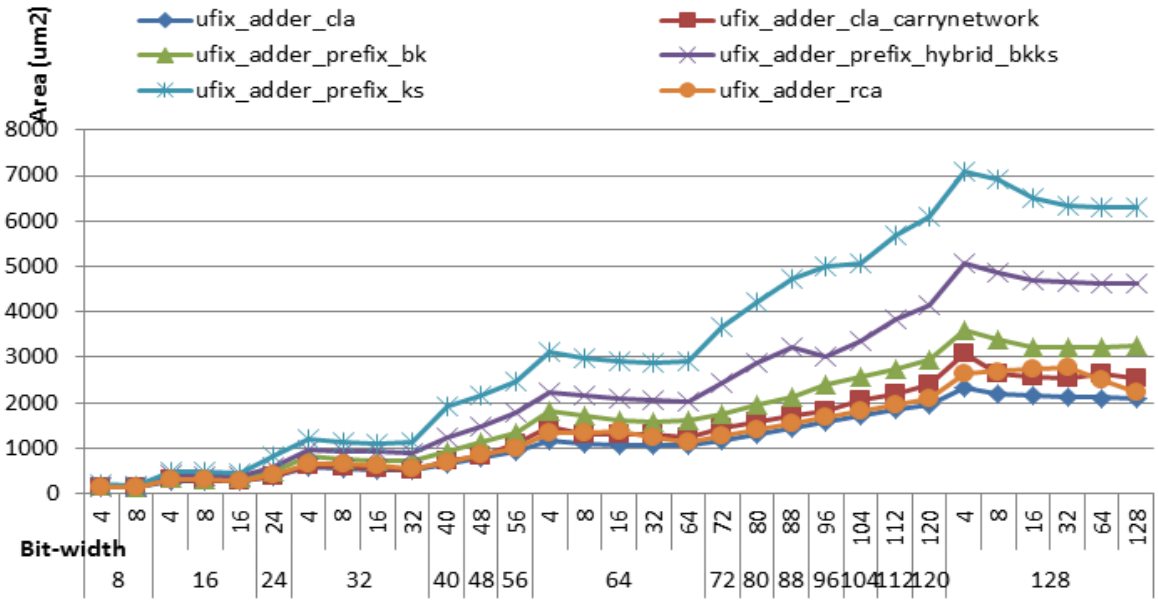

Figure 5.2. Area for ASIC implementation of fixed-point adders.

From both figures, it appears that Brent-Kung prefix adders have the best results in terms of latency, hardware requirement, and adapting the mechanism for dynamic precision support for ASIC implementation. On the other hand, a hybrid prefix adder should be used for FPGA designs when dynamic precision operation is required according to the analysis above.

B. Fixed-point Multipliers

This section shows the implementation results for two of the key components that are used in fixed-point datapath for both the bit-wise and modified Booth-4 multipliers, namely the partial product generator (PPGen) and the reduction tree. Just as in the previous section, two versions of each module are developed and synthesized with Synopsys Design Compiler for the FreePDK standard cell library of 45 nm CMOS technology.

The latency and area information for bit-wise PPGen modules with different configurations are shown in Fig. 5.3. The latency for a 128-bit dynamic precision design with 4-bit sub-blocks is around 10 times of that for a traditional 128-bit design. Yet, the PPGen delay has little effect on the overall performance compared to the large and slow reduction tree. As discussed in Chapter III, a generic reduction tree constructed with regular 4-to-2 blocks is sufficient for a bit-wise dynamic precision PPGen, thus no extra latency will be introduced in the reduction stage. Fig. 5.4 illustrates the synthesis results for generic reduction trees with different product bit-widths. As a result, there is no requirement for any specially designed final adder for a bit-wise multiplier.

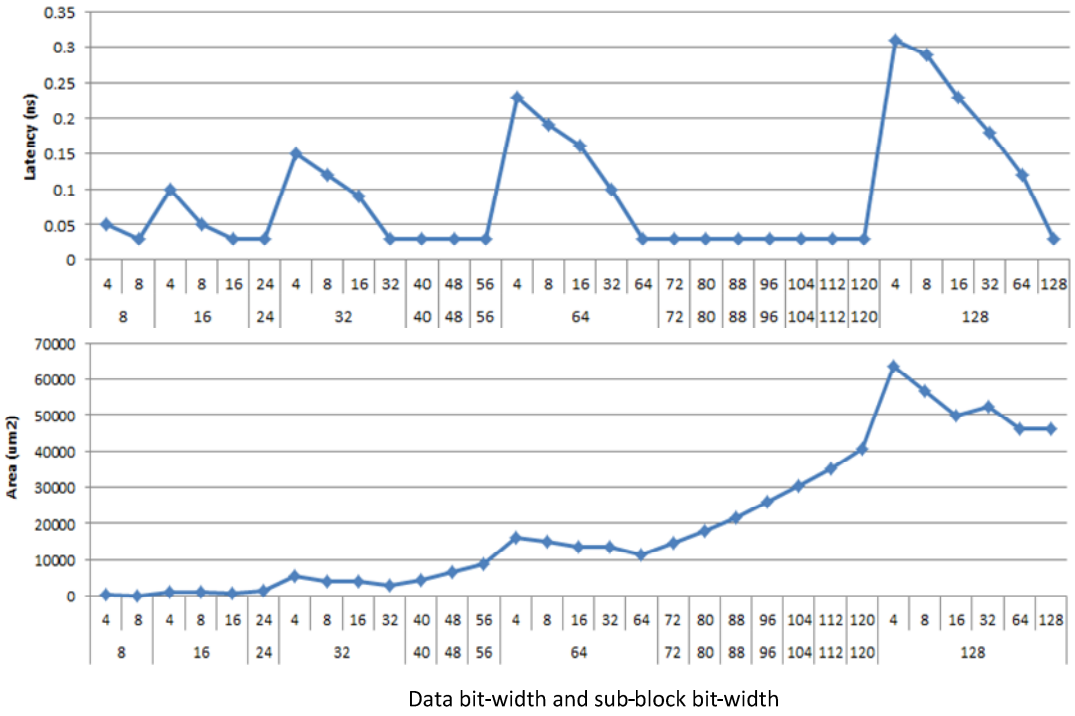

Figure 5.3. Latency and area for bit-wise PPGen implementations.

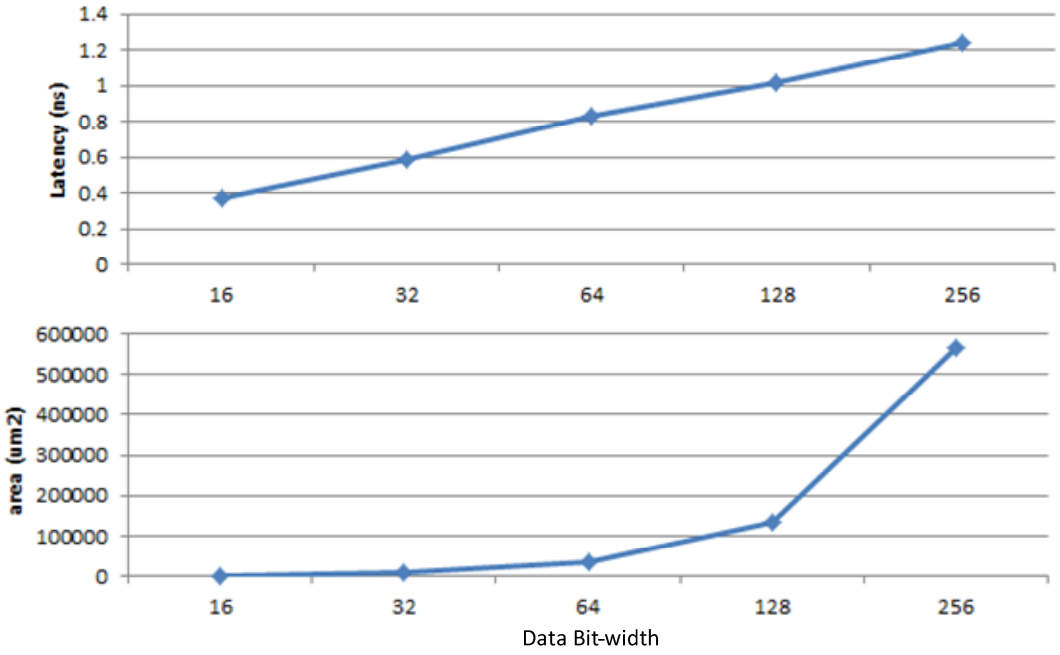

Figure 5.4. Latency and area for generic reduction tree implementations.

On the other hand, a modified Booth-4 PPGen can reduce the number of partial products almost in haft with the price of a more complicated generating process. Fig. 5.5 illustrates the detailed synthesis results for Booth-4 PPGens with different configurations. For a 128-bit PPGen with 4-bit sub-blocks, the latency is about 31% more than that for a traditional Booth-4 PPGen, and the size is 36% larger. It is 10% slower when compared to its bit-wise counterpart. As seen from the results shown in Fig. 5.6, the latency impact from changing the sub-block size is not significant, suggesting that the overhead introduced by the extra carry manipulating circuit is negligible compared to the total latency. A dynamic precision adder previously discussed is required for this type of multipliers.

To examine the performance of a complete dynamic precision multiplier, the synthesis results for both versions of a 128-bit traditional multiplier and a 128-bit dynamic precision multiplier with 4-bit sub-blocks are listed in Table 5.5. The bit-wise dynamic precision multiplier is only 3% slower and 9% larger than its traditional counterpart, while the modified Booth-4 multiplier is 19% slower and 33% larger.

Table 5.5. Synthesis results for traditional and dynamic precision multipliers.

Result	**128-bit Traditional**		**128-bit with 4-bit sub-blocks**	
	bit-wise	**Booth-4**	**Bit-wise**	**Booth-4**
Latency (ps)	2,050	1,870	2,120	2,220
Area (μm^2)	327,573	259,284	355,644	345,437
Ratio (Dynamic precision over Traditional)			103.41%	118.72%
			108.57%	133.23%

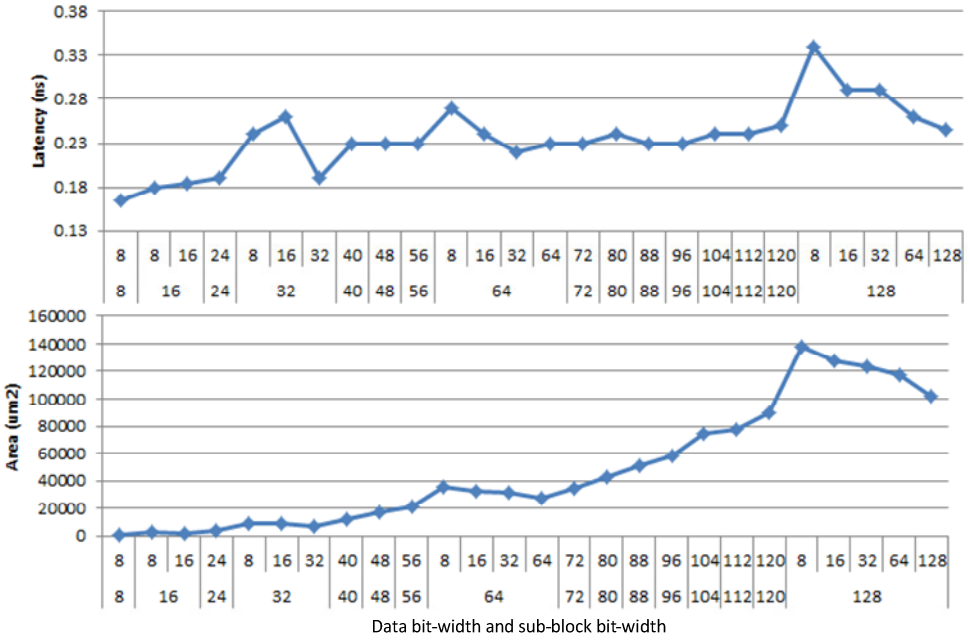

Figure 5.5. Latency and area for modified Booth-4 PPGen implementations.

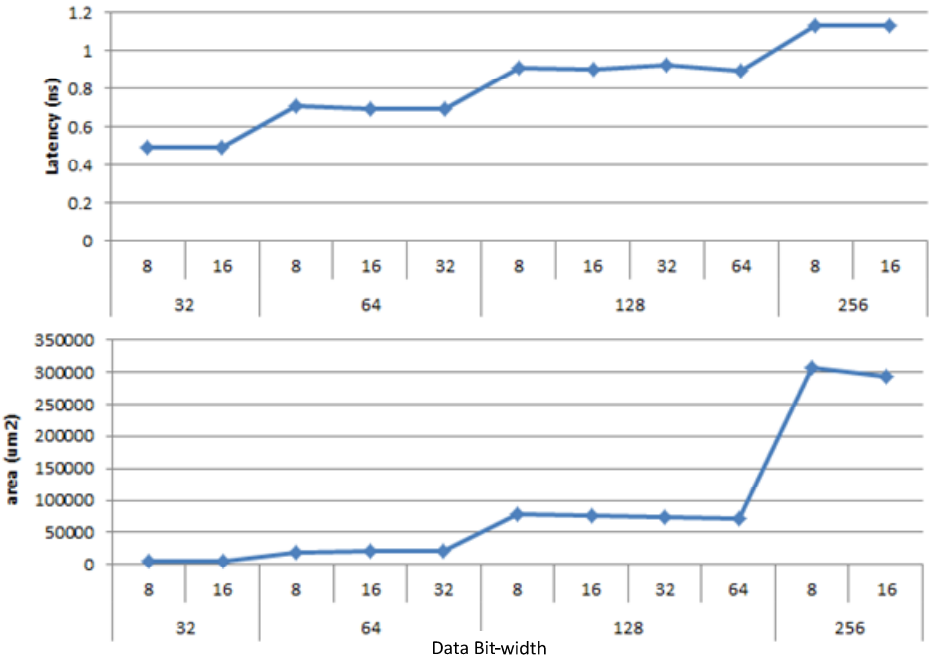

Figure 5.6. Latency and area for dynamic precision reduction tree implementations.

C. Eight-mode Floating-point ALUs

Three common floating-point ALUs are implemented with the proposed dynamic precision architectures discussed in Chapter IV, including the multiplier, adder and fused multiply-add. Although the complete ALUs are developed using the more practical 128-bit eight-mode scheme, most of the modules supporting dynamic precision are implemented with parameterized VHDL hardware description language, allowing the key parameters such as sub-block size and vector width to be statically re-defined when targeting a different mode scheme. Few modules require sophisticated hardwiring specifically for the mode scheme, such as the packing/unpacking modules and those variant mapping modules. Once the designs are verified against golden results through simulations, Synopsys Design Compiler is then used to synthesize the design with a standard cell library for IBM 8RF 0.13 μm CMOS technology. And finally, Cadence Encounter Digital Implementation System is employed to perform placement and routing to generate the final layout. Verification is performed throughout the different stages of the design flow.

1) 128-bit Eight-mode Floating-point Multipliers

Two eight-mode customized multiple precision floating-point multipliers with slight variations are implemented. One is built with a dynamic precision bit-wise significand multiplier and the other is powered by a dynamic precision multiplier with modified Booth-4 encoding technique.

For the bit-wise multiplier, the eight key modules form Fig. 4.3, can be divided into three zones based on their dependency on the multiplier datapaths, and each of these modules is initially synthesized with medium optimization effort only for latency or area respectively. The synthesis results for the eight key modules of a bit-wise multiplier are listed in detail in Table 5.6.

Table 5.6. Synthesis results for key modules of a bit-wise multiplier.

Module		Target for Latency		Target for Area	
		Latency (ps)	Area (μm^2)	Latency (ps)	Area (μm^2)
Zone 1	Exc. Detector	418	14,536	1,623	8,279
Zone 2	Exp. Operation	799	41,223	4,242	11,971
Zone 3	PP Generator	437	541,613	4,061	464,598
	Reduction Tree*	2,286	4,098,596	5,941	1,754,319
	Final Adder*	848	73,065	50,294	13,390
	1st Normalizer	502	18,323	3,325	12,787
	Rounding	941	70,478	5,832	29,279
	2nd Normalizer	477	12,726	2,557	7,674

As discussed in the previous section, a modified Booth-4 multiplier shares the same design for the five floating-point support modules with a bit-wise multiplier, but with a different significand partial product generator, reduction tree, and final adder. Table 5.7 shows the synthesis results for these three modules specific to a modified Booth-4 multiplier. The partial product generator in a Booth-4 multiplier is around 45% slower and 56% larger than that in a bit-wise multiplier, which is expected due to the great complexity of generating partial products with modified Booth-4 recoding. Obviously, the dynamic precision final adder required for a Booth-4 multiplier is a bit slower than the traditional final adder used for a bit-wise multiplier. However, these negative impacts

on the Booth-4 multiplier are compensated by the gains from a smaller and faster reduction tree. As a reward with the high-radix recoding, the number of partial products required for reduction is reduced from *n* to *n/2+1* for the modified Booth-4 multiplier, resulting in a reduction tree that is 46% smaller and 8% faster. The reason that the speed is not increased dramatically is because of the carry elimination circuits added to the critical path in the reduction tree. Note that the reduction tree and the final adder, which represent the majority of a bit-wise multiple-precision multiplier, are exactly the same as those in a 128-bit traditional multiplier. Therefore the latency and area overheads introduced to the relatively smaller modules by the proposed bit-wise architecture can be regarded as not significant when the whole design is considered.

Table 5.7. Synthesis results for specific modules of a modified Booth-4 multiplier.

Module		Target for Latency		Target for Area	
		Latency (ps)	**Area (μm^2)**	**Latency (ps)**	**Area (μm^2)**
Zone 3	PP Generator	632	842,793	10,320	805,918
	Reduction Tree	2,115	2,213,254	8,894	822,729
	Final Adder	987	66,019	51,864	14,301

From the data shown in Table 5.6 and Table 5.7, not only are we able to characterize the timing and resource requirements for each individual module, but also gain an understanding on how the total latency is distributed among the modules on the critical path (Zone 3). Characterizing the requirements allows us to tradeoff optimization goals between latency and area for non-critical modules in the following phases of synthesis and layout. For example, according to Table 5.6, the latency for the exception detector is

relatively small compared to that for the significand datapath, so it would be wise to set a higher priority to area optimization for that particular module.

By carefully examining the timing data shown in the tables, it becomes obvious that the best option for pipelining the critical path is a two-stage design with the partial product generator and the reduction tree being in the first stage and the rest of the modules in Zone 3 in the second stage. After multiple synthesis runs with different optimization settings on the two-stage pipeline design for both multipliers, we are able to achieve a clock rate of 2.79 ns for the bit-wise multiplier, and 2.68 ns for the Booth-4 multiplier. The detailed latency and area information for both multiplier implementations are listed in Table 5.8. For the first stage, the Booth-4 multiplier is about 4% faster and 12% smaller than the bit-wise one. For comparison, the synthesis results for 128-bit traditional multipliers with two different generators using the same pipeline strategy are also listed in Table 5.8. The only differences between the two traditional multipliers are the partial product generator and the reduction tree, which happen to be in the first stage. Therefore, their second stage is exactly the same. The achieved clock rate of our bit-wise multiplier is 15% slower than its traditional counterpart. But the total area requirement is slightly smaller, which we believe is caused by the synthesizer optimization algorithm. Thus, the achievable clock rate could have been faster if the synthesizer treats both multipliers with the exact strategy. For the Booth-4 multiplier, our dynamic precision implementation is 8% slower and 3% larger (total size) compared to a traditional multiplier.

Table 5.8. Synthesis results for pipeline stages of both multipliers.

Pipeline			Latency (ps)	Area (µm2)		
				Logic	Interconnect	Total
Traditional		First Stage (Bit-wise)	2,431	1,583,540	944,329	2,527,868
		First Stage (Booth-4)	2,476	1,376,215	736,973	2,113,188
		Second Stage	1,669	65,782	41,521	107,303
Eight-mode	**Bit-wise**	First Stage	2,787	1,568,842	851,914	2,420,756
		Second Stage	2,144	91,433	57,542	148,975
	Booth-4	First Stage	2,681	1,409,376	717,208	2,126,583
		Second Stage	2,182	102,128	58,397	160,525
Ratio (Eight-mode over Traditional)			114.67%	99.07%	90.21%	95.76%
Bit-wise			128.47%	138.99%	138.59%	138.84%
Ratio (Eight-mode over Traditional)			108.26%	102.41%	97.32%	100.63%
Booth-4			130.74%	155.25%	140.64%	149.60%

After verifying the correctness of the synthesized netlist, design placement and routing are performed for both multipliers with Cadence EDI. Table 5.9 lists the detailed layout reports on latency, chip area, and design density. Both multipliers become slower as expected due to the massive design and the resulting complicated interconnection after layout and routing, and their respective layouts (stage 1) are presented in Fig. 5.7. An eight-mode multiplier implementation using the proposed bit-wise multiple-precision architecture can achieve a clock rate of 3.56 ns, while the one with proposed modified Booth-4 architecture is 11% faster and 9% smaller, resulting in a 3.18 ns clock period. Further latency optimization can be performed by reduce the pre-set density during placement with the cost of increasing area. Yet, a faster ALU clock rate is not always the only goal for most modern processor designs, given that more cores and memory are packed into a single area-limited chip. For the high-precision multiplier designs presented in this paper, design tradeoffs between speed and silicon area must be taken into serious consideration.

Table 5.9. Place-and-Route results for pipeline stages of both multipliers.

Pipeline		Latency (ps)	Density	Pre-route Area (µm2)	
				StdCell	Total
Bit-wise	First Stage	3,557	72.12%	1,532,022	2,166,519
	Second Stage	2,319	68.07%	87,686	130,581
Booth-4	First Stage	3,180	74.41%	1,376,304	1,942,332
	Second Stage	2,411	68.50%	97,993	145,850

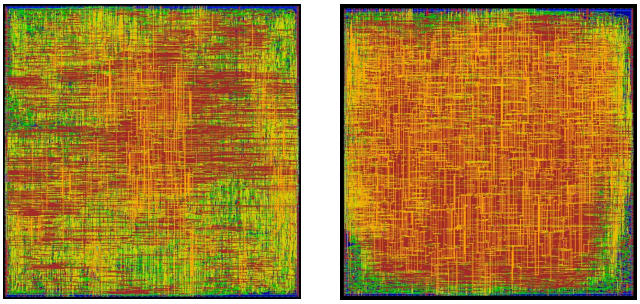

Figure 5.7. Design layouts for both multipliers (Stage 1).

2) 128-bit Eight-mode Floating-point Adder

A 128-bit dynamic precision floating-point adder with the eight-mode scheme is implemented. The synthesis results of the key modules from this adder are listed in Table 5.10. Both the latency and area information from two synthesis targeting different objectives are listed. The numbers from synthesis targeting latency are more important as an adder is expected to run as fast as possible for most applications. By revisiting the block diagram for the adder illustrated in Fig 4.19, we can briefly identify the critical path for the design, which goes from the exponent difference module through the whole significand datapath back to the exponent increment for the second normalization. The modules that contribute to the total delay are highlighted in Table 5.10.

Table 5.10. Synthesis results for key modules of a dynamic precision adder.

Module		Target for Latency		Target for Area	
		Latency (ps)	Area (μm^2)	Latency (ps)	Area (μm^2)
Stage 1	Exp. Difference	**668**	51,774	4,556	16,005
	Sig. Comp.	**706**	24,214	2,637	10,739
	Barrel Shift R.	**1,082**	177,223	4,817	103,563
	Sig. Adder	**909**	52,762	30,557	15,364
Stage 2	LZD	**574**	32,853	3,134	16,468
	Barrel Shift L.	**1,083**	144,980	5,029	64,362
	Shift R. 1b1	**97**	16,043	267	9,090
	Round Adder	**715**	23,844	25,041	8,131
	Shift R. 1b1 Simple	**97**	9,334	203	5,336
	Exp. Dec	**473**	26,591	3,246	5,688
	Exp. Inc	**280**	12,214	2,665	3,588
	Exp. Inc Cin	**308**	11,675	2,727	3,885

It is noticeable that the largest and slowest modules are the two barrel shifter variants. The latencies for both modules are almost identical, but the left barrel shifter is roughly 20% smaller than the right shifter. This size difference is contributed by the sticky logic that is required for the right barrel shifter but not the left shifter. Yet the latency impacts introduced by the extra sticky logic for each row shifter are completely removed by simply reversing the order of the row shifters, allowing the slower sticky logic having more time to settle. A design optimization has also been made to replace the unconditional exponent increment (plus the following multiplexers) with a controllable one during the final exponent updating processing. This reduces the resource requirements without increasing the clock period for a pipelined implementation.

For comparison purpose, a 128-bit traditional floating-point adder is also implemented using the same level of optimization effort. The timing and area requirements for the key modules that define the critical path are listed in Table 5.11. Our

vectorized version of the exponent processing modules, such as the exponent increment and the difference module, are about 6 times the size of their scalar counterparts, which is expected because of the exponent vector format and supporting circuits. It can also be observed that the vectorized barrel shifters are 50% slower compared to the ones for a traditional 128-bit adder. This comes from the extra gates that are responsible for the block selections to ensure proper vectorized shifts for both the significand and GRS vectors. Approximately 20% of delay overhead is applied to the significand adder and the rounding adder after adding dynamic precision support, and our vectorized LZD is only 7% slower.

Table 5.11. Synthesis results for key modules of a traditional adder.

Module		Target for Latency		Target for Area	
		Latency (ps)	Area (μm^2)	Latency (ps)	Area (μm^2)
Stage 1	Exp. Difference	**643**	7,590	3,323	3,041
	Barrel Shift R.	**720**	60,532	3,673	3,673
	Sig. Adder	**770**	41,721	26,128	6,934
Stage 2	LZD	**536**	9,828	3,494	4,567
	Barrel Shift L.	**692**	60,565	3,226	30,588
	Round Adder	**587**	24,713	10,489	7,418
	Exp. Inc Cin	**296**	2,002	2,620	697

Characterizing the timing and resource requirements for the key components helps to establish different optimization strategies for critical and non-critical modules for the following phases of synthesis. It can also provide insightful guidance for determining the pipeline stages. After carefully reviewing the synthesis results, we came to the conclusion that a 2-stage pipeline design is best for the eight-mode adder. The first stage includes

components along the critical path up to the significand adder, while the rest belongs to the second stage. The details of the stage partitioning are also presented in Table 5.10 and Table 5.11. Both a 128-bit traditional adder and an eight-mode design are synthesized using the same two-stage pipelining strategy, and the detailed synthesis results for both designs are shown in Table 5.12. The two-stage eight-mode adder is able to achieve a clock rate of 2.72 ns, while the traditional adder is rated at 2.32 ns. The first stage of the dynamic precision adder is about 17% slower, and the second stage is around 20% slower. However, the silicon area for the dynamic precision adder is increased more than 100% for both stages.

Table 5.12. Synthesis results for pipeline stages of adders.

Pipeline		Latency (ps)	Area (μm2)		
			Logic	Interconnect	Total
Traditional	First Stage	2,322	97,234	55,552	152,786
	Second Stage	1,849	63,056	43,550	106,606
Eight-mode	First Stage	2,718	191,964	131,859	323,823
	Second Stage	2,216	128,171	88,211	216,381
Ratio (Eight-mode over Traditional)		117.05%	197.42%	237.36%	211.94%
		119.83%	203.27%	202.55%	202.97%

After necessary verification of the synthesized netlist, design placement and routing are performed. The detail layout reports on latency, chip area, and design density for both adders are listed in Table 5.13. An eight-mode adder implementation can achieve a clock rate of 3.01 ns, while a traditional adder requires a clock rate of 2.98 ns. After design place-and-route, the clock rate difference between the two designs is significantly reduced from 17% to 1%, but the area ratio between the two remains in the same 200%

range. The actual design layouts for both stages from an eight-mode adder are depicted in Fig. 5.8.

3) 128-bit Eight-mode Floating-point FMA Unit

Two 128-bit dynamic precision floating-point FMA units with the eight-mode scheme are implemented. One of the FMA units has a bit-wise partial product generator and traditional reduction tree, and the other is powered with a modified Booth-4 partial product generator and a vectorized reduction tree. The synthesis results of the key modules for the FMA are listed in Table 5.14. Although both the latency and area information from synthesis targeting two different objectives are listed, the numbers from latency-optimized synthesis are treated with more attention. Since many of the modules are identical to those in a dynamic precision adder or multiplier, they are reused for the FMA designs. Design reusability is one of the merits to evaluate how well a sub-module is designed. The modules listed with a * in the module name are the ones that are reused.

Table 5.13. Place-and-Route results for pipeline stages of adders.

Pipeline		Latency (ps)	Density	Pre-route Area (μm2)	
				StdCell	Total
Traditional	First Stage	2,978	64.81%	90,634	135,717
	Second Stage	1,966	67.12%	58,160	87,606
Eight-mode	First Stage	3,011	69.11%	182,656	265,664
	Second Stage	2,262	70.78%	122,359	175,773
Ratio (Eight-mode over Traditional)		101.11%	106.63%	201.53%	195.75%
		115.06%	105.45%	210.38%	200.64%

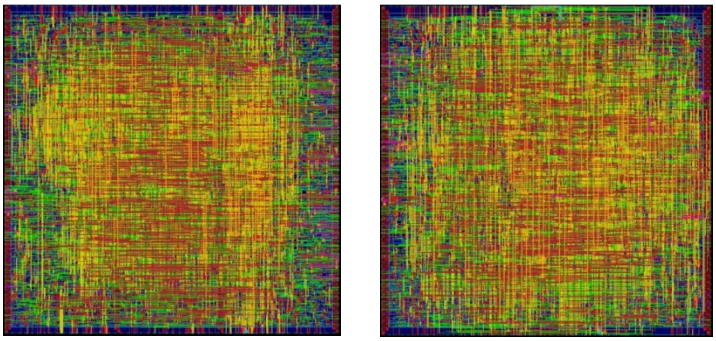

Figure 5.8. Design layouts for both stages of the eight-mode adder.

Table 5.14. Synthesis results for key modules of FMA units.

Module			Target for Latency		Target for Area	
			Latency (ps)	Area (μm^2)	Latency (ps)	Area (μm^2)
Stage 1		Exp. Difference	**1,486**	52,442	5,685	17,672
		Barrel Shift R.	**1,435**	295,605	6,695	217,303
	op.1	PP Gen *	**437**	541,613	4,061	464,598
		Tree *	**2,286**	4,098,596	5,941	1,754,319
	op.2	PP Gen (Booth4) *	**632**	842,793	10,320	805,918
		Tree (Booth4) *	**2,115**	2,213,254	8,894	822,729
Stage 2		Sig. Adder	**1,559**	158,107	56,123	59,003
		Sig. Complement	**1,139**	80,106	30,123	36,936
		LZD stage 1	**686**	64,257	3,589	36,815
Stage 3		LZD stage 2	**574**	22,501	3,761	8,670
		Barrel Shift L.	**1,496**	302,539	8,091	212,628
		Exp. Dec *	**473**	26,591	3,246	5,688
		Round Adder *	**715**	23,844	25,041	8,131
		Shift R. 1b1 Simple *	**97**	9,334	203	5,336
		Exp. Inc Cin *	**308**	11,675	2,727	3,885

In Table 5.14, the latencies for the modules on the critical path are highlighted for a better overview on how the total latency is distributed among the critical path. Except the reused modules, the dominating modules in terms of latency are the significand adder, barrel shifters, and the exponent difference module. It is understandable that the significand add is slow considering its bit-width is 256. The exponent difference module, where there are two levels of dual mode adders inside, is about 122% slower when compared to that for an adder, where there is only one level of the same adders. After evaluating the netlist, the significantly increased delay for the barrel shifter comes from the 8-to-1 multiplexers. After analyzing the data and performing multiple trials, we decided to implement the FMAs as 3-stage pipeline designs with a target for the clock rate for the FMAs close to that for the dynamic precision multipliers and adder. The detailed pipeline strategy is also presented in Table 5.14. The first stage consists of a reused significand multipliers and the pre-processing module for the additive operand. To be able to have a better latency distribution between the second and third stage, the LZD module is divided into two portions, with the two LZD trees in the second stage and the final adder in the last stage. Both FMAs are implemented with the same pipeline strategy, and the detail synthesis reports are listed in Table 5.15. The only difference between the two FMA implementations is the significand multiplier in the first stage, and the rest are all identical. After synthesis, it is observed that the pre-processing operations require less time (~2,353 ps) than any of the significand multipliers. Thus, the latency listed for the first stage for both implementations are directly copied from Table 5.8, and the area numbers are the sum of the two sub-paths. A FMA with Booth-4 multiplier is around 4% faster and 11% smaller than a FMA design with a bit-wise multiplier. When compared to

a 128-bit traditional FMA unit, the bit-wise implementation is 15% slower and 12% larger, while the modified Booth-4 design is 8% slower and 18% larger.

Table 5.15. Synthesis results for pipeline stages of FMA units.

Pipeline		Latency (ps)	Area (μm^2)		
			Logic	Interconnect	total
with bit-wise MULT	First Stage	2,787	1,776,110	1,025,662	2,801,772
	Second Stage	2,864	208,535	120,764	329,299
	Third Stage	2,723	189,546	171,185	360,731
with Booth4 MULT	First Stage	2,681	1,616,643	890,956	2,507,599
	Second Stage	2,864	208,535	120,764	329,299
	Third Stage	2,723	189,546	171,185	360,731
Ratio (Stage1: Booth4 over Bit-wise)		96.18%	91.02%	86.87%	89.50%

The detailed layout reports for latency, density, and chip area for both FMAs are listed in Table 5.16. Again, since the time required by the pre-processing (~2,560 ps) is smaller than that of the multiplication, the first stage latency for both implementations is copied. After place-and-layout, the pipeline bit-wise FMA can achieve a clock rate of 3.56 ns, while the Booth-4 one has a clock rate of 3.18 ns (10% faster and 9% smaller). The actual layouts for the tree stages from the eight-mode FMA with modified Booth-4 multiplier are depicted in Fig. 5.9.

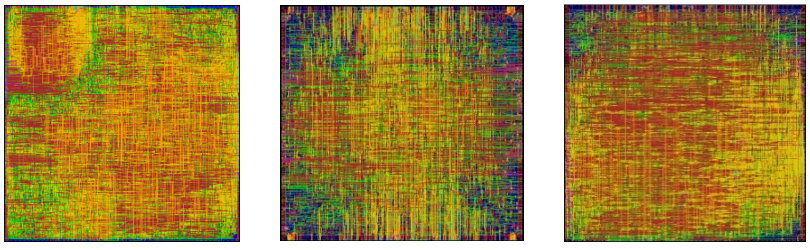

Figure 5.9. Design layouts for the three stages of an eight-mode FMA unit.

Table 5.16. Place-and-Route results for pipeline stages of FMA units.

	Pipeline	Latency (ps)	Density	Pre-route Area (μm²)	
				StdCell	Total
with bit-wise MULT	First Stage	3,557	72.14%	1,730,848	2,456,169
	Second Stage	3,145	71.09%	202,608	288,841
	Third Stage	3,108	72.67%	182,211	262,177
with Booth4 MULT	First Stage	3,180	74.13%	1,575,130	2,231,982
	Second Stage	3,145	71.09%	202,608	288,841
	Third Stage	3,108	72.67%	182,211	262,177
Ratio (Stage1: Booth4 over Bit-wise)		89.40%	102.77%	91.00%	90.87%

In this chapter, we present the implementation results on FPGA or ASIC for many ALU architectures that support dynamically defined precision. These ALUs include fixed-point adders and multipliers, as well as floating-point adders, multipliers, and fused-multiply-add units. For ASIC implementations, synthesis results for each key component are presented and analyzed for the potential pipeline strategy. And the results from both the synthesized netlist and layout are also presented and analyzed. By comparing the results from dynamic precision implementation to those from the traditional implementations, we are able to obtain an understanding on the performance impacts when dynamic precision and vectorized operations are supported.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Scientific computing has been a popular and effective research tool for scientists and engineers from different research areas for many years. In order to face the challenge of increasing computational demands, efforts have been spent on computing platforms with faster processing units, better parallelism, and hardware accelerators. On the other hand, the computational precision of arithmetic units in physical processors is also improving over the years. Yet, it is impossible to keep the trend of increasing processor frequency or operational precision without finally hitting the barriers imposed by physical constraints. Besides, the energy consumption has become a serious problem for computer centers powered by variant high performance computing platforms. Therefore, exploiting precision and parallelism optimization that is independent of process technology becomes an alternative approach to improve the overall performance. Custom precision (standard and non-standard) computing enables better fine-grained mixed-precision operations for iterative algorithms, taking advantage of least-sufficient-precision arithmetic units [49 - 55]. ALU-level parallelism allows better fault tolerance and performance per operation by dynamically sharing hardware resources.

We reviewed several software libraries that support arbitrary precision arithmetic through sophisticated software algorithms. Yet, all of them are heavily relied on the provided hardware ALUs, which inspires us with the idea that a better hardware ALU implementation is the foundation for a potential better software solution. We also

reviewed many hardware ALU solutions for both fixed-point and floating-point systems targeting FPGA and ASIC technologies. However, all the discussed related work addressing multiple precision hardware architectures mainly focuses on standard precision formats, neglecting the potential benefit from fully customized precision, which is another reason that motivates our research on ALU architectures supporting dynamically defined precision computing.

In this dissertation, we presented the details on developing dynamic precision ALU architectures to enable better computational performance, energy efficiency, and fault tolerance. First, the mechanism of dynamically defined precision that is the foundation of the proposed ALU architectures is presented. Next, the detailed discussions are provided on the vectorized dynamic precision architectures for fixed-point adders, multipliers, and multiply-accumulators, which also serve as the core datapath for their respective floating-point processing counterparts. Furthermore, we also spent a chapter focusing on the proposed dynamic precision floating-point ALU architectures, including adder, multiplier, and fused-multiply-add units. An eight-mode dynamic precision scheme is applied during the discussion to simplify the demonstration of the general vectorized architectures with a more specific configuration.

All the presented architectures are implemented for Xilinx FPGAs or ASIC standard cell libraries (45 nm FreePDK and 0.13 μm IBM8RF) in order to evaluate the performance impacts imposed by the vectorized dynamic precision processing mechanism. All the designs are developed with parameterized VHDL hardware description language, and verified using extensive simulations with randomly generated

operands. The correctness is demonstrated by comparing results with the outputs from trusted software, such as MATLAB and Perl programs. Synthesis results of latency and area are provided for fixed-point ALUs, while a more detailed report, including both synthesis and layout results for pipelined designs, is presented and analyzed for each of the floating-point architectures. When evaluating the presented ALUs, we used the results from their respective traditional design with similar implementation efforts as the baseline. In general, a 128-bit dynamic precision fixed-point adder (except RCA adder) with 4-bit sub-blocks is 10%~15% slower and 10%~20% larger, depending on the actual adder structure. Similarly, a 128-bit bit-wise multiplier is 3% slower and 9% larger, and a 128-bit modified Booth-4 multiplier is 19% slower and 33% larger than a traditional multiplier. From the implementation results of our floating-point multipliers, the eight-mode bit-wise multiplier is 15% slower but with a similar hardware requirement, and the Booth-4 multiplier is 8% slower and 3% larger. After place and route, the latency performance of the eight-mode adder is on par with a traditional adder, but with a doubled size. Compared to a 128-bit traditional FMA unit, the presented bit-wise implementation is 15% slower and 12% larger, while the modified Booth-4 design is 8% slower and 18% larger. From the implementations and analysis, it can be concluded that it is promising to enable dynamically defined precision support and vectorized operations for common ALUs with reasonable extra costs of processing speed and silicon area, even for a high precision 128-bit ALU with relatively small sub-block size.

We performed a broad design space exploration before developing the ALU architectures with dynamically defined precision, and considered many options that are

not presented in this dissertation. Take the most fundamental number representation system for an example. We also considered several conventional and unconventional representations, such as decimal, biased, redundant, and residue number systems [14]. Decimal representation, although the best option for human understanding, is not suitable for digital system where data are encoded with binary bits, resulting in a redundant representation and complicated arithmetic. A redundant system enables speedup operations by removing possible carry propagation, but faces problems due to the requirement of conversions between the internal and external format when interfaced with outside systems. Although a residue system might speedup certain operations with true parallelism, it is determined to be not suitable for general applications due to the difficulty of some basic arithmetic operations [14]. In terms of actual operator architecture, we also investigate many other options. Some of these options can be considered equivalent in some degree to the implementations presented, while some are only suitable for specific applications. For example, a carry-select adder requires extra hardware for each option offered, thus the hardware requirement increases with the increasing selection options (e.g., very high precision adders). Nevertheless, we were able to incorporate the underlying idea into our floating-point ALU implementation. For instance, the addition of the three significand vectors in a FMA unit takes advantage of the carry-select method to reduce latency and hardware requirements. Although a floating-point adder with dual-path algorithm is not presented, it can be easily implemented using the modules from our single-path adder with slight modifications. Combining the presented dynamic precision multiplier and the divide-and-conquer technique, higher precision hybrid iterative multipliers can be implemented. When

designing the eight-mode scheme to demonstrate the general dynamic precision floating-point architectures, we decided to employ uniform exponent and significand size for all elements. This scheme can be expanded to handle more complicated configurations, such as higher total bit-width, variable sub-block sizes, more operational modes (standard and non-standard), and non-uniform sizes for exponents and significands.

Arithmetic logic units with dynamically defined precision have great potential for improving scientific computation, by allowing lower-level precision optimization for applications. Applications that currently seek dynamic precision support from software libraries can be ported to run directly on the hardware ALUs. The performance can be improved greatly thanks to direct hardware support and the removal of library overhead. More importantly, the dynamic precision ALU implementations can become the foundation for even more sophisticated software solutions that is temporary impractical in the hardware domain, resulting in broader powerful options for scientific researchers and engineers. Vectorized processing is an indispensable capability for efficient dynamic precision architectures, because of the desire for better performance by maximizing hardware utilization and parallelism in hardware units throughout the whole computation, regardless of the operating precision. The realization of this dynamic precision approach for scientific computing involves significant changes in many aspects of the computing infrastructure, including but not limited to the arithmetic units (the ultimate goal of this research), the processor microarchitectures, the memory, the compilers, the applications, and more importantly the users. There is a dilemma for the dynamic precision approach: people do not develop dynamic precision algorithms because there is no general-purpose

hardware to support them, and people do not implement general-purpose dynamic precision ALUs because there are no applications to run on them. The changes are difficult, but sometimes we must make a first step and exploit other promising options, especially when some of the current directions are not sustainable. We made our first small step, and we believe it will be a solid stepping-stone to a better scientific computing era.

This dissertation laid out a framework for ALU architectures with dynamically defined precision, on which many architectural and implemental improvements can be built in the future. For example, more effort should be spent on the optimization of the presented dynamic precision architectures and sub-modules for better performances, as well as on the optimization tuning during the synthesis and automatic place-and-rout (APR) stages for better implementations. We can also further expand our exploration of potential architectures for the implementation of dynamic precision operators. It will be interesting to see how our presented architectures can be adapted for the hardware implementation of the new *unum* system [48]. Because the *unum* system uses variable exponent and significand sizes, it is a perfect candidate for adapting our presented dynamic precision architectures. Last but not least, we can try to implement the presented architectures with fully custom design flows, which is the industry standard for commercial processors, for a more realistic performance comparison to arithmetic units in modern CPUs. Custom design can generate the most optimized implementation in terms of latency, area, and power requirement, and therefore reveal the true potential of the presented dynamic precision architectures for future scientific computation.

REFERENCES

[1] P. Lynch, “The origins of computer weather prediction and climate modeling,” *Journal of Computational Physics*, vol. 227, no. 7, pp. 3431–3444, Mar. 2008.
[2] R. a. Kendall, E. Aprà, D. E. Bernholdt, E. J. Bylaska, M. Dupuis, G. I. Fann, R. J. Harrison, J. Ju, J. a. Nichols, J. Nieplocha, T. P. Straatsma, T. L. Windus, and A. T. Wong, “High performance computational chemistry: An overview of NWChem a distributed parallel application,” *Computer Physics Communications*, vol. 128, no. 1–2, pp. 260–283, Jun. 2000.
[3] A. F. W. Coulson, J. F. Collins, and a. Lyall, “Protein and Nucleic Acid Sequence Database Searching: A Suitable Case for Parallel Processing,” *The Computer Journal*, vol. 30, no. 5, pp. 420–424, Oct. 1987.
[4] Gordon E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, pp. 114–117, April 19, 1965.
[5] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in Proceedings of the April 18-20, 1967, *spring joint computer conference on - AFIPS ’67 (Spring)*, 1967, p. 483.
[6] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988.
[7] T. El-Ghazawi, E. El-Araby, and M. Huang, “The promise of high-performance reconfigurable computing,” *Computer*, no. February, pp. 69–76, 2008.
[8] M. Feldman, "GPUs Will Morph ORNL's Jaguar Into 20-Petaflop Titan," *HPC Wire*, 2011
[9] K. Compton and S. Hauck, “Reconfigurable computing: a survey of systems and software,” *ACM Computing Surveys*, vol. 34, no. 2, pp. 171-210, Jun. 2002.
[10] D. Buell and T. El-Ghazawi, “High-performance reconfigurable computing,” *COMPUTER-IEEE COMPUTER SOCIETY*, no. March, pp. 23–27, 2007.
[11] J. Owens and D. Luebke, “A Survey of General-Purpose Computation on Graphics Hardware,” *Computer graphics forum*, vol. 26, no. 1, pp. 80–113, 2007.
[12] “MCS-8 Micro Computer Set Users Manual.” Intel Corporation, 1972.
[13] David Kanter, "Intel’s Haswell CPU Microarchitecture," http://www.realworldtech.com/haswell-cpu, 2012.
[14] B. Parhami, *Computer arithmetic: algorithms and hardware designs*., 2nd edition, Oxford University Press, 2010.
[15] “IEEE Standard for Floating-Point Arithmetic,” IEEE Std 754-2008, vol. 1–58, 2008.
[16] R. Hiremane, “From Moore’s Law to Intel Innovation - Prediction to Reality Technology,” *Technology @ Intel Magazine*, no. April, pp. 1–9, 2005.
[17] D. A. Kramer and I. D. Scherson, "Dynamic Precision Iterative Algorithms," in The IEEE 1992 Symposium on the Frontiers of Massively Parallel Computation, 1992.
[18] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," presented at the Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florida, 2006.
[19] J. Kurzak and J. Dongarra, "Implementation of mixed precision in solving systems of linear equations on the Cell processor: Research Articles," *Concurr. Comput.: Pract. Exper.*, vol. 19, pp. 1371-1385, 2007.
[20] J. Lee and G. D. Peterson, "Iterative Refinement on FPGAs," in *Application Accelerators in High-Performance Computing (SAAHPC)*, 2011 Symposium on, 2011, pp. 8-13.
[21] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-Performance Mixed-Precision Linear Solver for FPGAs," *IEEE Trans. Comput.*, vol. 57, pp. 1614-1623, 2008.
[22] C. Kao, “Benefits of partial reconfiguration,” *Xcell journal*, pp. 65-67, 2005.
[23] T. Granlund and the GMP development team, “GNU MP Manual,” 2013.
[24] G. Hanrot, V. Lefevre, P. Pélissier, P. Théveny, and P. Zimmermann, “The GNU MPFR library,” 2005. [Online]. Available: http://www.mpfr.org/.
[25] M. Nakata, “The MPACK (MBLAS/MLAPACK); a multiple precision arithmetic version of BLAS and LAPACK,” 2010. [Online]. Available: http://mplapack.sourceforge.net/.
[26] D. H. Bailey, Y. Hida, X. S. Li, B. Thompson, K. Jeyabalan, and A. Kaiser, “High-Precision Software Directory,” 2012. [Online]. Available: http://crd-legacy.lbl.gov/~dhbailey/mpdist/.

[27] X. Wang, "VFloat: A Variable Precision Fixed-and Floating-Point Library for Reconfigurable Hardware," *ACM Transactions on Reconfigurable Technology*, vol. 3, no. 3, pp. 1-34, 2010.
[28] G. A. Vera, M. Pattichis, and J. Lyke, "A Dynamic Dual Fixed-Point Arithmetic Architecture for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2011, pp. 1–19, 2011.
[29] S. Perri, P. Corsonello, M. A. Iachino, M. Lanuzza, and G. Cocorullo, "Variable precision arithmetic circuits for FPGA-based multimedia processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 9, pp. 995–999, Sep. 2004.
[30] M. Sjalander and P. Larsson-Edefors, "Multiplication Acceleration Through Twin Precision," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 9, pp. 1233–1246, Sep. 2009.
[31] S. R. Kuang and J. P. Wang, "Design of Power-Efficient Configurable Booth Multiplier," *IEEE Transactions on Circuits and Systems* I: Regular Papers, vol. 57, no. 3, pp. 568–580, Mar. 2010.
[32] A. Danysh and D. Tan, "Architecture and implementation of a vector/SIMD multiply-accumulate unit," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 284–293, Mar. 2005.
[33] A. Akkas, "Dual-mode floating-point adder architectures," *Journal of Systems Architecture*, vol. 54, no. 12, pp. 1129-1142, Dec. 2008.
[34] G. Even, S. M. Mueller, and P. M. Seidel, "A dual precision IEEE floating-point multiplier," Integration*, the VLSI journal*, vol. 29, no. 2, pp. 167–180, 2000.
[35] A. Akkas and M. Schulte, "Dual-mode floating-point multiplier architectures with parallel operations," *Journal of Systems Architecture*, vol. 52, no. 10, pp. 549-562, Oct. 2006.
[36] D. Tan, C. E. Lemonds, and M. J. Schulte, "Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 175–187, Feb. 2009.
[37] L. Huang, S. Ma, L. Shen, Z. Wang, and N. Xiao, "Low Cost Binary128 Floating-Point FMA Unit Design with SIMD Support," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1–8, 2011.
[38] A. Isseven and A. Akkas, "A Dual-Mode Quadruple Precision Floating-Point Divider," in Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on, 2006, pp. 1697–1701.
[39] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
[40] P. Kornerup, "Reviewing 4-to-2 adders for multi-operand addition," *The Journal of VLSI Signal Processing*, no. 21, pp. 143–152, 2005.
[41] T. Ahmed, P. D. Kundarewich, J. H. Anderson, B. L. Taylor, and R. Aggarwal, "Architecture-specific packing for virtex-5 FPGAs," in Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays - FPGA '08, 2008, p. 5.
[42] D. Culler, J. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.
[43] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, 2009, no. c, pp. 427–436.
[44] A. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied*, vol. 4, no. 2, pp. 236–240, 1951.
[45] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 124–128, Mar. 1994.
[46] M. C. Smith and G. D. Peterson, "Optimization of Shared High-Performance Reconfigurable Computing Resources," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 2, pp. 1–22, Jul. 2012.
[47] B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 89–98, 1990.
[48] J. Gustafson, *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.
[49] Junqing Sun, Gregory D. Peterson, and Olaf Storaasli "High Performance Mixed-Precision Linear Solver for FPGAs." *IEEE Transactions on Computers*, 57(12):1614-1623, December 2008.
[50] JunKyu Lee and Gregory D. Peterson, "The Role of Precision for Iterative Refinement." *Symposium on Application Accelerators for High Performance Computing*. Chicago, July 2012.

[51] Getao Liang, JunKyu Lee, and Gregory D. Peterson, "ALU Architecture with Dynamic Precision Support." *Symposium on Application Accelerators for High Performance Computing*. Chicago, July 2012.
[52] JunKyu Lee and Gregory D. Peterson, "Iterative Refinement on FPGAs", *Symposium on Application Accelerators for High Performance Computing*. July 2011.
[53] JunKyu Lee, Junqing Sun, Gregory D. Peterson, Robert Harrison and Robert Hinde, "Power-aware Performance of Mixed Precision Linear Solvers for FPGAs and GPGPUs." *Symposium on Application Accelerators for High Performance Computing*. July 2010.
[54] JunKyu Lee, Junqing Sun, Gregory D. Peterson, Robert J. Harrison, and Robert J. Hinde, "Accelerator performance comparison for mixed precision linear solvers", *IEEE Conference on Field Programmable Custom Computing Machines*, April 2010.
[55] JunKyu Lee, Gregory D. Peterson, R.J. Hinde, and Robert J. Harrison, "Mixed Precision Dense Linear System Solvers for High Performance Reconfigurable Computing," *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, UIUC, July, 2009.
[56] Melissa C. Smith and Gregory D. Peterson, "Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources." *Performance Evaluation*. 60(1-4):107-125, 2005.
[57] Melissa C. Smith and Gregory D. Peterson, "Optimization of Shared High-Performance Reconfigurable Computing Resources," *ACM Transactions on Embedded Computing Systems*, 11(2): Article 36, July 2012.
[58] Gregory D. Peterson and Roger D. Chamberlain, "Parallel Application Performance in a Shared Resource Environment." *IEEE Distributed Systems Engineering Journal*, 3:9-19, August 1996.

VITA

Getao Liang was born in Deqing, Guangdong, China in 1979. He received his B.S. degree in Electrical Engineering from South China University of Technology in 2003. He joined the Department of Electrical and Computer Engineering at the University of Tennessee, Knoxville in Fall 2004 and received his M.S. degree in Computer Engineering in 2007.