University of California
Los Angeles

Variable Precision Arithmetic with Lookup Table Based Field Programmable Gate Arrays

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Marianne Elise Louie

1994

The dissertation of Marianne Elise Louie is approved.

Charles Taylor

David Rennels

Algirdas Avižienis

Miloš Ercegovac, Committee Chair

University of California, Los Angeles
1994

In loving memory of my father, William Louie,
who emphasized the importance of education and
who dreamed of seeing his daughter earn a PhD.

To my family
who have lovingly supported my years of study.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to my advisor, Professor Miloš Ercegovac, for his guidance, and for obtaining fellowships and a research assistantship for me. Professor Ercegovac introduced me to the subject of field programmable gate arrays, and both he and Professor Tomás Lang introduced me to the area of computer arithmetic. I have enjoyed exploring these combined subjects and hope to continue research in these areas for many years.

I also thank Professor Algirdas Avižienis, Professor David Rennels and Professor Charles Taylor for being on my dissertation committee and for their questions and comments.

I appreciate the many others who have been helpful at UCLA. I thank Dr. John Fernando for teaching me about the graduate study process. Verra Morgan and Shelly Backlar have also been very helpful with graduate program affairs. Bobby Weng programmed the LSA conversion algorithm in the C language.

To E. Jean Dryer and Dr. Charles Weir, I am grateful for their advice and fatherly guidance while I was both working and attending school.

I am especially indebted to my family for their support. Terrance Domae, my husband, has been invaluable for his constructive criticisms, discussions, and patience during my research. My father, William Louie, inspired me to pursue a PhD and financially supported much of my education. My mother, Faye Louie, provided encouragement and continuously helped with many tasks so that I would have more time for studies. My aunt, Alice Louie, offered many prayers.

To my Father in Heaven, I give thanks for providing the strength and insights needed throughout my education.

This research was sponsored in part by a State of California MICRO Grant and by Xilinx, Inc.

"This is what the Lord says ...
'Call to me and I will answer you and
tell you great and unsearchable things
you do not know.' "

Jeremiah 33:2-3

In this dissertation, portions of Chapter 2 and Chapter 6 are excerpts, with permission of Kluwer Academic Publishers, from:
M.E. Louie, M.D. Ercegovac, "A Variable Precision Square Root Implementation for Field Programmable Gate Arrays," to appear in the Journal of Supercomputing, 1994.

Chapter 6 is also based on work presented in:
M.E. Louie, M.D. Ercegovac, "A Digit-Recurrence Square Root Implementation for Field Programmable Gate Arrays," Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, April 5-7, 1993, Napa Valley, CA, pp. 178-183.
Permission to use excerpts from this publication has been granted by IEEE.

Portions of Chapter 4 and Chapter 5 are excerpts, with permission of Kluwer Academic Publishers, from:
M.E. Louie, M.D. Ercegovac, "Implementing Division with Field Programmable Gate Arrays," to appear in the Journal of VLSI Signal Processing (special issue on computer arithmetic), vol. 7, no. 3, 1994, pp. 271-285.

Chapter 5 is also based on work presented in both:
M.E. Louie, M.D. Ercegovac, "On Digit-Recurrence Division Implementations for Field Programmable Gate Arrays," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 202-209.
and
M.E. Louie, M.D. Ercegovac, "Mapping Division Algorithms to Field Programmable Gate Arrays," Proc. 26th Asilomar Conference on Signals, Systems, and Computers, Oct. 26-28, 1992, Pacific Grove, CA, pp. 371-375.
Permission to use excerpts from these publications has been granted by IEEE.

VITA

1983	B.S. (Summa Cum Laude), Engineering, University of California, Los Angeles
1986	M.S., Computer Science, University of California, Los Angeles

PUBLICATIONS AND PRESENTATIONS

M.E. Louie, M.D. Ercegovac, "Mapping Division Algorithms to Field Programmable Gate Arrays," Proc. 26th Asilomar Conference on Signals, Systems, and Computers, Oct. 26-28, 1992, Pacific Grove, CA, pp. 371-375.

M.E. Louie, M.D. Ercegovac, "A Digit-Recurrence Square Root Implementation for Field Programmable Gate Arrays," Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, April 5-7, 1993, Napa Valley, CA, pp. 178-183.

M.E. Louie, M.D. Ercegovac, "On Digit-Recurrence Division Implementations for Field Programmable Gate Arrays," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 202-209.

M.E. Louie, M.D. Ercegovac, "A Variable Precision Multiplier for Field Programmable Gate Arrays," 2nd Int'l ACM/SigDA Workshop on Field Programmable Gate Arrays, Feb. 13-15, 1994, Berkeley, CA.

M.E. Louie, M.D. Ercegovac, "Linear Sequential Arrays: Pipelining Arithmetic Data Paths," Technical Report CSD-940010, Computer Science Dept., University of California, Los Angeles, March 1994.

M.E. Louie, M.D. Ercegovac, "Implementing Division with Field Programmable Gate Arrays," to appear in the Journal of VLSI Signal Processing (special issue on computer arithmetic), vol. 7, no. 3, 1994, pp. 271-285.

M.E. Louie, M.D. Ercegovac, "A Variable Precision Square Root Implementation for Field Programmable Gate Arrays," to appear in the Journal of Supercomputing, 1994.

Abstract of the Dissertation

Variable Precision Arithmetic with Lookup Table Based Field Programmable Gate Arrays

by

Marianne Elise Louie

Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1994
Professor Miloš Ercegovac, Chair

The computation time of many arithmetic intensive applications would be improved with an application-specific integrated circuit (ASIC), but due to high development costs and the long time to fabrication ASICs are often not created. In these cases, the simple hardware reconfigurability of lookup table based Field Programmable Gate Arrays (FPGAs) can make custom hardware practical. FPGAs feature rapid configuration; FPGAs also provide good hardware speed that is faster than a software implementation but slower than an ASIC due to internal hardware for programmability.

This work examines digit-recurrence arithmetic of any precision with lookup table based FPGAs. Despite the growth of broadcasting delays as a function of precision, digit-recurrence arithmetic algorithms typically broadcast the control signals to all digit slices in the data path. This study develops a generalized pipelined technique to avoid the common problem of broadcasting delays in the critical path. With a pipelined approach, cycle time becomes independent of precision. An algorithm is developed to convert any broadcasting digit-recurrence arithmetic data path to this pipelined approach. The technique is also extended for multiple chip designs where it can avoid performance degradation due to interchip communication.

For the logic design, three mapping strategies create efficient variations of the original recurrence algorithm. A mapping process combines these strategies to iteratively reduce the logic delay of the critical path. Using the XC4010, variable precision designs are created for SRT division, radix-2 square root, and radix-16 multiplication. Extrapolated performance results demonstrate that a custom arithmetic design with FPGAs can have comparable speeds to a general purpose microprocessor and that high precision designs with FPGAs can provide a significant speedup over software even when vectorized on a Cray.

CHAPTER 1

Introduction

For many arithmetic intensive applications, an application-specific integrated circuit (ASIC) would provide significant computation speed improvements. However, the time and cost of fabricating a custom chip often prevent its development. Such applications have typically been relegated to slower performance software implementations on general purpose computers. Since the introduction of Field Programmable Gate Arrays (FPGAs), a hardware implementation alternative has become available that features good hardware speeds combined with the design flexibility of software.

FPGAs consist of an array of uniform logic resources that are custom interconnected via programming, much like an EEPROM or EPROM. Some FPGAs rely on simple device programmers while others are reprogrammable directly on the circuit board. As a result, FPGAs feature a faster development time than the creation of ASICs but a slower development time than software due to the need for detailed bit-level logic specification. The programmable hardware provides faster performance than software, but also adds delays that are not found in ASICs.

Arithmetic operations that can experience improved processing speed with application specific hardware, but for which ASICs are not always available, include:

1. functions not provided by a given general purpose CPU, e.g., square root,
2. operations with non-standard input/output requirements, e.g., very high precision, and
3. specific operation sequences comprising more complex functions, e.g., inner product.

For the first and second cases, application-specific hardware improves performance by avoiding the need for software implementations. For the third case, custom hardware exploits parallelism and optimizes performance by combining several arithmetic operations into a composite hardware algorithm; examples are found in on-line arithmetic [Erce77, BrEW89, Tu90, Fern93], CORDIC arithmetic algorithms [Walt71, Desp74, CaLu88, Hu92, DuMu93], and algorithms in cryptography

[Mont85, ShVu93, Korn93]. Performance in all cases depends on the creation of efficient arithmetic algorithms that can also be efficiently implemented. If a portion of an algorithm cannot be implemented efficiently, the actual critical path may differ from the theoretical critical path, and the gains achieved by algorithm optimizations are lost. For example, most digit-recurrence arithmetic algorithms require broadcasting of the result digit (control signal) to the data path and assume that broadcasting is only a small portion of the critical path delay. In actuality, these broadcasting delays can be significant and can degrade overall performance. For very high precision designs, the resulting large broadcasting delays can dwarf speedup gains from other algorithm optimizations. This research explores the development of efficient application-specific arithmetic designs for lookup table based FPGAs. Specifically, this study examines digit-recurrence arithmetic algorithms of variable precision. The creation of algorithm variations for efficient implementation, including a method to avoid the broadcasting delay problem, is addressed.

1.1 FPGA Overview

A lookup table based FPGA is characterized as a two-dimensional array of uniform logic cells interconnected by programmable routing lines. Within each cell is a set of logic resources including (1) either a single lookup table or a small group of tables that may be connected in a hierarchy [Xil92, Xil91], and (2) flip-flops for sequential designs. By providing an n-input lookup table, each cell offers easily customized logic where any function of n bits can be implemented. By offering hierarchies of lookup tables, larger functions can be implemented with minimal interconnection delay. Programmable interconnections within a logic cell enable the use of resources only as needed for each custom design. Configuring an FPGA design thus consists of programming the lookup tables, the resource interconnections within a logic cell, and the interconnections between logic cells.

1.1.1 The Xilinx XC4010

The examples in this study use the Xilinx XC4010 lookup table based FPGA. The XC4010 consists of an array of 20x20 configurable logic cells known as CLBs (Configurable Logic Blocks) that are interconnected horizontally and vertically by programmable routing lines (Fig. 1.1). Several long lines provide a reduced delay of large fanouts for each row and column of interconnects. The minimum interconnect delay, approximately 1 ns. [XACT], occurs between adjacent CLBs, and the delays grow with increasing fanout and routing distance. The delay for a large fanout on a long line is at least that of a single CLB's lookup tables.

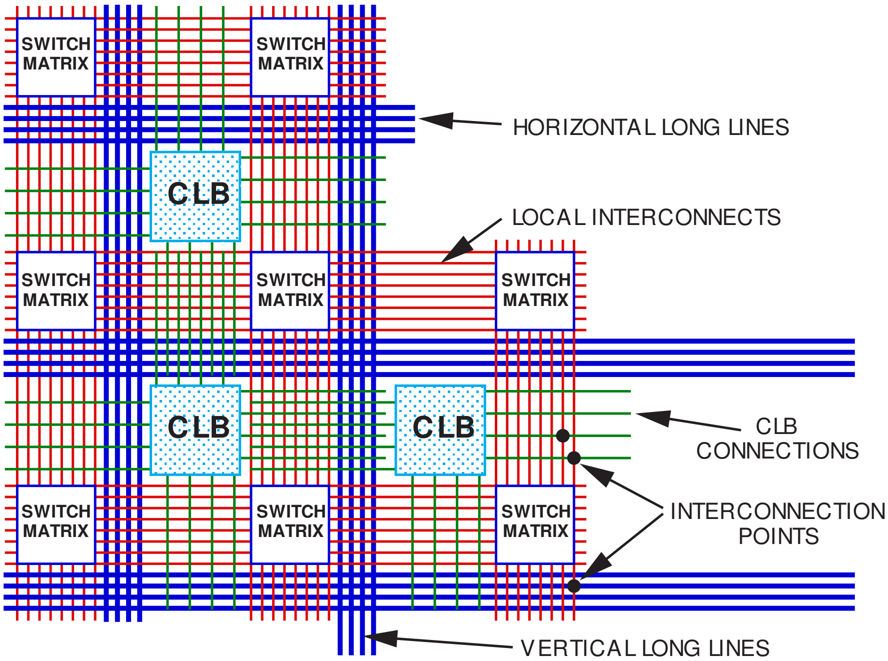


Figure 1.1: XC4010 CLB Interconnections (adapted from [XACT])

The precise delay of individual interconnects and logic is dependent on fabrication and varies among chips. Each manufactured batch yields a small volume of very high performance chips where all interconnects and logic delays satisfy a strict maximum delay. As the requirements for acceptable delay decreases, more chips in the batch satisfy the requirements. The Xilinx chips are thus sorted according to performance, known as *speed grades*. Two speed grades (-6 and -5) are currently available for the Xilinx XC4010. The fastest is speed grade -5, which is used in this study. For speed grade -5, the minimum local interconnect delay is 1.3 ns [XACT].

In a simplified form, Figure 1.2 shows the configuration of each XC4010 logic block, and Table 1.1 presents the logic delay. Each XC4010 CLB is configurable to a variety of functions including all functions of 5 or fewer input bits and some functions of up to 9 inputs. In Figure 1.2, let the inputs to lookup table F be labeled F_1, F_2, F_3, and F_4. Similarly label the inputs to table G as G_1, G_2, G_3, and G_4, and the third input to table H as H_1. The functions that may be computed

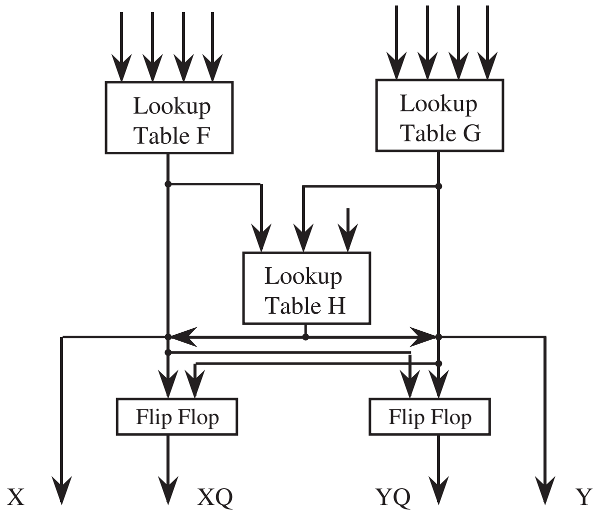


Figure 1.2: Simplified Form of an XC4010 CLB [LoEr92] ©1992 IEEE

Configuration	Speed Grade -6 Delay (ns)	Speed Grade -5 Delay (ns)
Table F or G to Output X or Y	6.0	4.5
Tables F and/or G through H to Output X or Y	8.0	7.0
Flip Flop Input to Output (Clock-to-Q)	5.0	3.0

Table 1.1: Logic Delays of an XC4010 CLB [XACT]

for output X are thus:

$$x = \begin{cases} f(F_1, F_2, F_3, F_4) \\ h(H_1, f(F_1, F_2, F_3, F_4)) \\ h(H_1, f(F_1, F_2, F_3, F_4), g(G_1, G_2, G_3, G_4)) \end{cases}$$

where the lookup table functions are defined by $f()$, $g()$, and $h()$, and any of the inputs to the lookup tables may be a *don't-care* value. The functions for Y and the flip flop inputs driving XQ and YQ are similarly defined where these functions share the inputs H_1, F_i and G_i, $1 \leq i \leq 4$, and the outputs from $f()$, $g()$ and $h()$. From the above definitions and Table 1.1, a CLB executes functions of more than 4 inputs with the same delay.

The XC4010 also has a fast carry propagate adder (FCPA) feature which provides dedicated circuitry for the computation and interconnection of carries. The delay of an n-bit FCPA along a single column of CLBs on the XC4010 speed grade -5 is:

$$T_{\text{delay}} = 11.5 + 1.5(\lfloor (n-3)/2 \rfloor) \text{ ns., } n \geq 3$$

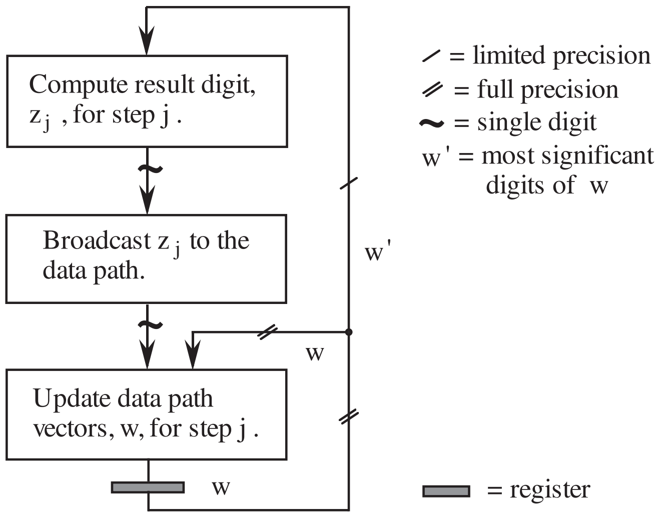


Figure 1.3: Conventional Digit-Recurrence Arithmetic Model

1.2 Arithmetic Algorithm Structure: Overview

This work studies digit-recurrence arithmetic algorithms and their mapping to FPGAs. Digit-recurrence arithmetic [ErLa94] is characterized by a residual vector, w, that is updated with each iteration, j. Almost all digit-recurrence algorithms involve a feedback process where z_j (the jth digit of the result which is also computed at the jth step) determines the manner of updating $w[j]$ (w at step j). Multiplication is the exception to the feedback model in which the value of z_j is not a result digit but rather a digit of the multiplier. The digit z_j can be positive, negative, or zero (signed-digit) [Aviz61, Aviz64]. Updating the residual also requires another full precision operand (the divisor in the division operation, the multiplicand in a multiplication operation, and the j-digit result in square root). For simplicity, this operand is not explicitly mentioned in the general discussion, but is assumed to be part of the updating process. Figure 1.3 shows a model of conventional digit-recurrence arithmetic. When z_j is a function of w, the arithmetic algorithm is designed to limit the selection of z_j to a function of only a few most significant bits, w', of w. Since z_j is a function of fewer input bits, the selection logic generally has reduced logic depth and area. The bits in the data path, w'', consist of the portion of $w[j]$ that is not involved in the selection function. The boolean expressions for the data path usually consist of fewer terms than the expressions for the selection function and hence have lower combinational logic delay. Reducing the delay of the selection function is thus the typical approach toward reducing the delay of the critical path.

Computer arithmetic algorithms exhibit a regular structure in both the selection function and in the data path. For the data path, each residual bit is a function of the result digit and a set of neighboring residual bits where all bits are arranged according to weight. Generally, the function is identical for all bits, w''_i, in the data path and is of the form $w''_i = f(z_j, w''_{i+k}, w''_{i+l} \ldots)$, k, l, integers. This uniformity enables modular designs and bounding of the data path delay in variable precision implementations.

The selection function also features some uniformity. A selection function usually begins with the assimilation of the most significant bits of the residual vectors (commonly the sum and carry vectors) into a single vector representation. Assimilation is most typically achieved by a serial scheme in which each assimilated output is a function of the same weighted inputs and carry data from previous iterations in the scheme. The result digit is then selected based on comparing the assimilated value with predefined constants. By finding parallelism in bit level evaluations, efficient selection function designs can be developed.

1.3 Problem Statement and Approach

While FPGAs offer the design flexibility of software, FPGAs also feature design obstacles such as large routing delays and limited logic and interconnection resources. Large routing delays can cause serious cycle time delays for arithmetic since typical algorithms use large fanout broadcasting for passing result digits to the data path. In all technologies, routing delays grow with increases in fanout and routing distance. However, with FPGAs, reprogrammability magnifies routing delay problems due to support of many programmable interconnection points. One of the problems addressed in this research is how a digit-recurrence arithmetic algorithm can avoid the use of large fanout broadcasting. The presented solution, the Linear Sequential Array (LSA), is a generalization of the pipelining concept of Figure 1.4 [Erce84] and is a form of linear systolic array [KuLe79]. The data path in an LSA is developed as stages of uniform modules where each module extends the precision of the design. Local interconnections between modules maintain small routing delays so that cycle time is independent of precision. Chapter 2 of this study performs a theoretical derivation of the LSA. It also presents a general method to convert any data path in a digit-recurrence scheme from a broadcasting approach to an LSA design [LoEr94b]. Since high precision designs can span multiple FPGAs, a method is presented to interface LSA modules across chips without increasing the delay.

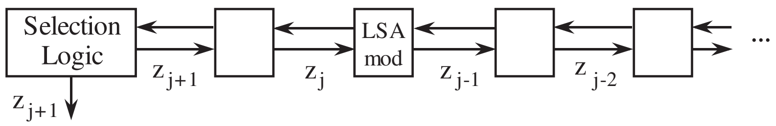


Figure 1.4: The Linear Sequential Array Organization

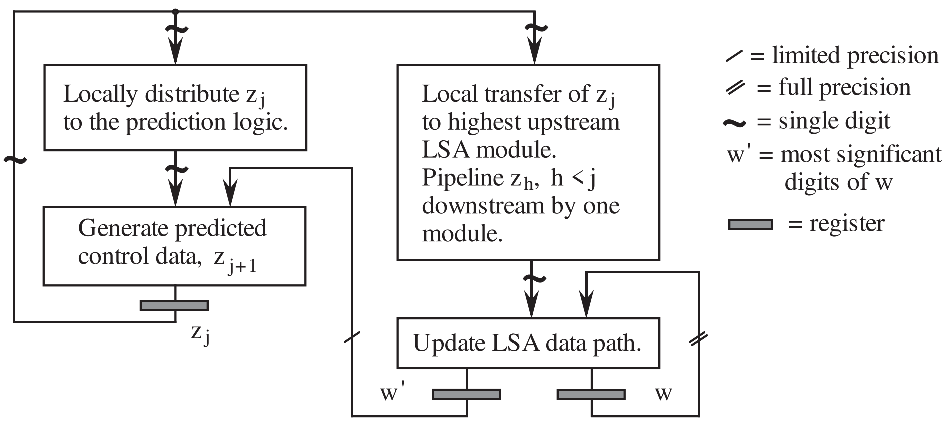


Figure 1.5: Arithmetic Model with an LSA and Prediction

Regarding the selection function, a prediction scheme [ErLa85] enables the next iteration's selection to proceed in parallel with the transfer of result digits and evaluation of the data path logic. Since many arithmetic data paths consist primarily of a carry-save adder, the LSA delay is small. As a result, the combination of the LSA and prediction can eliminate the data path from the critical path. Past prediction schemes have involved prescaling the operands and developing new selection functions [ErLa85, ErLa89]. This work describes a simple approach to prediction for radix-2 schemes through implementation variations of the selection function. The method is based on repositioning the registers (Chapter 4) of the digit-recurrence arithmetic model of Figure 1.3 to that of Figure 1.5. Register repositioning is similar to retiming [LeRS83]. In this concept, the registers are moved within an algorithm to allow the precomputation of variables.

For digit-recurrence designs in which the data path is removed from the critical path, the cycle time depends on the efficient implementation of the selection function. Delay reduction for the selection function is achieved through design variations that balance the delay of the logic paths. Along with register repositioning, two additional strategies (Chapter 4), merging and efficient encodings, attempt to lower the delay of the critical path [LoEr94c]. Since an n-input lookup table implements any function of n input bits, merging (similar to bin packing [FrRV91]) consolidates successive algorithm steps into a common logic block. Efficient encodings aid in merging by reducing the number of inputs to an algorithm function. In this work, the above three strategies are combined to create efficient implementation variations of the original recurrence algorithm. The general mapping approach is that development of new functions of fewer input bits generally causes a decrease in logic delay. Based on this approach, a mapping process to iteratively reduce the logic delay of the critical path is presented.

1.4 Related Work

Several coprocessor boards and computers employ reprogrammable FPGAs as customizable processing elements where the FPGAs are configured for each application (PAM [Ber89, BeRV93], Splash [ArBD92], AnyBoard [Van92], etc.). Efficient arithmetic implementations for reprogrammable FPGAs thus serve as building block templates for custom processors in arithmetic intensive designs. Each template is easily tailored to applications with variable precision requirements. Very high precision digit-recurrence algorithms are easily accommodated by replicating the logic blocks of the lower precision bits. Alternatively with standard precision off-the-shelf arithmetic hardware, very high precision problems could not be satisfied without a significant amount of software intervention.

Some existing work in the area of arithmetic processors and FPGAs includes the Flexible Processor Cell [WoSh88] and Programmable Active Memories (PAM) [BeRV93]. Performance estimates for some mathematical applications are given in [BeRV93], an arithmetic processor is proposed in [WoSh88], and an FPGA design for a digital filter is presented in [ChME93], but none of these describes the process of mapping specific algorithms.

Regarding variable precision arithmetic, two sources of potentially large delay exist — (1) broadcasting the result digit to the data path, and (2) propagating carries when updating the residual. To limit carry propagation, redundant number systems [Aviz62] allow each digit to incorporate a carry value. Commonly used redundant formats include signed-digit [Aviz61, Aviz64] and carry-save [CS252a, Hwan79]. To avoid the broadcasting delay problem, a high precision modular multiplication implementation on the PAM [ShBV91] compensates for broadcasting delays with an extended prediction scheme for the result digits [ShVu93]. With prediction [ErLa85, LoEr92, LoEr93a, MoCi94], broadcasting of the current iteration's result digit begins simultaneously with computation of the next iteration's result digit. In the PAM extended prediction approach, the output of the current iteration is dependent upon result digits computed several iterations earlier; thus broadcasting can be started earlier. However, since the necessary amount of precomputing depends upon the precision of the design, the selection logic area also grows as a function of precision.

Systolic arrays [KuLe79] offer another approach to avoid broadcasting delays. In systolic arrays, processing elements are modular cells that are arranged in a regular structure with local pipelined interconnections. These modular features aid in the development of area efficient and high performance designs. In a small class of systolic designs, *bit-level systolic arrays*, the processing elements perform either bit-slice [MCMW82, MCMW87] or digit-slice [IrOw87, IrOw89] operations. A majority of bit-level systolic designs are two-dimensional structures (e.g., multibit convolver, correlator, rank-order filter [MCMW87], divider [MQMC92, DaFe92] and square rooter [MQMC92]) while a few are one-dimensional (linear) arrays (e.g., radix-2 and radix-4 digit-slice multipliers [Lyon76, MCMW86, IrOw89], adder [IrOw87], and FIR filter [IrOw89]). Linear configurations offer a reduced amount of hardware and simpler construction over two-dimensional designs [Foun88]. However, some arithmetic operations such as division and square root have been difficult to implement in a bit-level linear array [IrOw87]. In multiply-add type operations, the direction and ordering of the pipelined data (most-significant digit first or least-significant digit first) can be specified so that each processing element's data dependencies are satisfied at the beginning of each cycle. Unfortunately in division and square root, the direction and ordering of the pipelined data is fixed by the algorithm, and thus some bit slices in a pipelined scheme may be dependent on other bit slices that have not yet been computed. This dissertation solves this data dependency problem by redefining the size of the processing element and allowing an additional bounded delay for the computation of the necessary data dependencies. Hence, processing elements generally consist of more than one bit slice and use small fanout broadcasting within each processor. This small fanout broadcasting usually involves only a few bit slices (four bit slices for the presented designs of Chapters 3, 5 and 6) and features a delay comparable to local interconnections.

In the past, the design of systolic arrays has been an *ad hoc* process [Megs92]. Many systematic design methodologies for systolic arrays [Megs92, MCMK90, LiWa85] have been proposed. Progress has been made toward automatic synthesis, but most synthesis algorithms do not provide satisfactory results [Megs92]. This dissertation thus presents an algorithm that maps any digit-recurrence arithmetic data path from a broadcasting approach to a linear array scheme.

1.5 Research Contributions

The contributions of this research include:

1. generalization of the LSA technique to avoid large fanout broadcasting in digit-recurrence arithmetic data paths,
2. development of an algorithm to convert broadcasting schemes in digit-recurrence arithmetic to an LSA approach,
3. extension of the LSA to avoid interchip communication delays in the cycle time of multiple chip designs,
4. development of strategies to map digit-recurrence arithmetic selection functions to lookup table based FPGAs,
5. presentation of a simple approach to result digit prediction for radix-2 schemes, and
6. design of variable precision templates with the Xilinx XC4010 for radix-16 multiplication, SRT division, and radix-2 square root.

1.6 Outline

This dissertation is organized into two parts: (1) the derivation and examples of the Linear Sequential Array and (2) a mapping method for digit-recurrence selection functions to lookup table based FPGAs. For the LSA, Chapter 2 presents a theoretical derivation including an extension for multiple FPGA chips. Corresponding proofs for the LSA are also given, and an algorithm is presented to convert digit-recurrence broadcasting schemes to LSA designs. A variation of the LSA for radix-16 multiplication is presented in Chapter 3. Performance comparisons are made with a 32-bit integer multiplier on a microprocessor and a higher precision software implementation on the Cray. For the selection function, Chapter 4 discusses three mapping strategies and an iterative method to reduce the critical path. The mapping method is then applied to both SRT division (Chapter 5) and radix-2 square root (Chapter 6).

CHAPTER 2

The Linear Sequential Array

Although routing delays grow with fanout and routing distance, large fanout broadcasting has typically been used by arithmetic algorithms for passing result digits to the data path. For example, conventional digit-recurrence dividers [Hwan79, ErLa94], add-and-shift iterative multipliers [Hwan79], and various on-line units [Tu90, Fern93] currently employ broadcasting. Unfortunately, broadcasting delays in the critical path degrade performance and cause arithmetic implementations of differing precisions to experience different cycle times. For high precision designs, broadcasting delays can dominate the critical path.

Broadcasting can create serious cycle time delays when designing with Field Programmable Gate Array (FPGA) technology. An FPGA's reprogrammability magnifies routing delay problems due to support of the many programmable interconnection points. A few special long interconnection lines are usually provided within an FPGA for global broadcasting (e.g., for clocks and reset signals), but due to their very limited interconnection to resources, the long lines are usually insufficient for arithmetic algorithms.

One approach to alleviating the performance degradation due to broadcasting has been prediction of the result digit [ErLa85, LoEr92, LoEr93b, LoEr94c, ShVu93, MoCi94]. With prediction, broadcasting of the current result digit proceeds in parallel with computation of the next iteration's result digit. For very small precisions, the combined delay of broadcasting and the simple data path logic is smaller than the delay of the selection logic. For these small precisions, broadcasting does not appear in the critical path. However, on the Xilinx XC4010 broadcasting delays can redefine the critical path for precisions greater than 25 bits [LoEr93b].

This chapter generalizes the Linear Sequential Array (LSA) approach [Erce84, LoEr93b, LoEr94d] which replaces broadcasting in digit-recurrence arithmetic with a pipelined scheme (Fig. 2.1). The data path is developed as stages of uniform modules where each module extends the precision of the design. Modules are simply appended as needed to create a data path of the desired precision. For FPGAs the modular approach enables rapid construction of custom precision designs. Interconnections between modules are local with small fixed fanout to maintain low routing delays. Although the LSA modules are interconnected as a linear systolic array, the modules internally use small fanout broadcasting. Due to the small routing distance, the broadcasting delays within a module are comparable to local interconnection delays. Therefore, cycle time is independent of precision. Combining the LSA with result digit prediction can often prevent the data path from appearing in the critical path [LoEr93b, LoEr94d]. The LSA also features comparable combinational logic requirements as a conventional broadcasting scheme, but the LSA uses more latches for passing data through the pipeline.

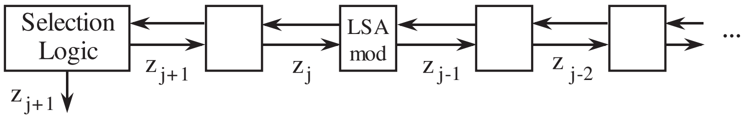


Figure 2.1: The Linear Sequential Array Organization

LSA pipelining is made possible by the uniform and localized data dependencies that characterize arithmetic data paths. Converting a data path from broadcasting to an LSA design requires examining the bit level dependencies as a function of time. Section 2.2 presents observations and proofs describing the data path dependency types, and discusses the mapping of these dependencies to the LSA. Section 2.3 presents the general conversion algorithm and examples. A discussion of LSA implementation characteristics as compared to a conventional broadcasting technique is given in section 2.4. Section 2.5 then extends the LSA for multiple chip designs.

2.1 Notation and Definitions

The following definitions simplify the position/weight descriptions of modules and bits relative to another module or bit.

upstream module A module that precedes a reference module in the pipeline (Fig. 2.2). An upstream module thus operates on step $j+i$, $i \geq 1$ when the reference module operates on step j.

upstream bit A bit that precedes a reference bit in vector position relative to the direction of the data path pipeline (Fig. 2.3). For example, given the vector x, if the result digits are pipelined from higher weighted bits to lower weighted bits, the bits x_i, $i < 5$ are upstream from the bit x_5. Alternatively, if the result digits are pipelined from lower weighted bits to higher weighted bits, the bits x_i, $i > 5$ are upstream from the bit x_5.

downstream module A module succeeding a reference module in the pipeline (Fig. 2.2). A downstream module thus operates on step $j - i$, $i \geq 1$ when the reference module operates on step j.

downstream bit A bit that succeeds a reference bit in vector position relative to the direction of the data path pipeline (Fig. 2.3). For example, given the vector x, if the result digits are pipelined from higher weighted bits to lower weighted bits, the bits x_i, $i > 5$ are downstream from the bit x_5. Alternatively, if the result digits are pipelined from lower weighted bits to higher weighted bits, the bits x_i, $i < 5$ are downstream from the bit x_5.

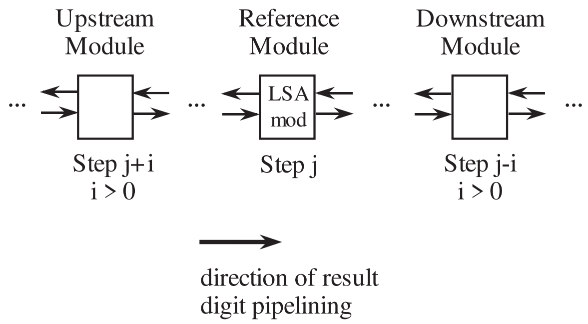


Figure 2.2: Upstream and Downstream Modules

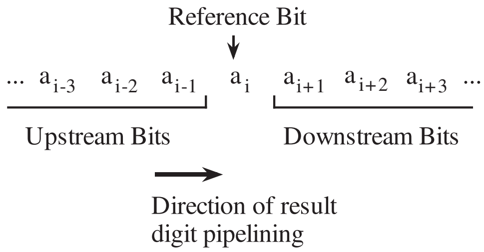


Figure 2.3: Upstream and Downstream Bits

Arrays are used in the definition of the conversion algorithm. For the vector A, define A_{ij} as the cell in row i and column j.

The following notations are used throughout this chapter. Additional notations are defined as needed. Iterative arithmetic algorithms can be summarized as having the format $u[j+1] = f(u[j], z_{j+1})$. The following notation describes this format:

$u[j]$ The set of iteration variables with values at step j. This set includes the residual vector w, and any other iterated variables within the arithmetic algorithm (e.g., variables for on-the-fly conversion [ErLa87])

z_{j+1} result digit for step $j+1$

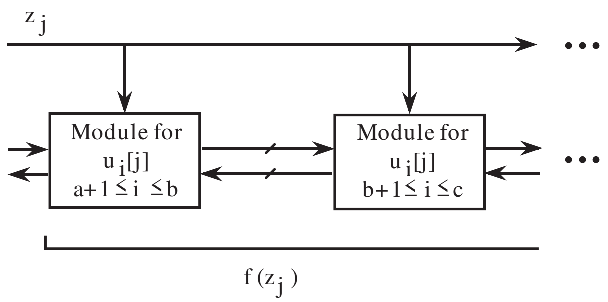
In general, the digit labels in arithmetic algorithms and arithmetic designs have the format d_i where i indicates that the digit is of weight 2^{-i}. Hence, the subscripts for fractions are positive values. For the LSA conversion algorithm, however, the subscripts for data path equation digits and matrix elements are referenced with respect to upstream and downstream position. When the result digit flows from higher weighted bits to lower weighted bits, a reference bit will have subscript i while its downstream bits (of lower precision) have subscript $i-p$, $p > 0$. Analogously, when the result digit flows from lower to higher weighted bits, a reference bit will have subscript i whiles its downstream bits (of higher precision) have subscript $i+p$, $p > 0$.

2.2 The Data Path

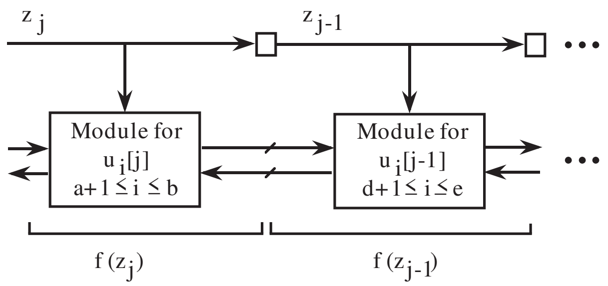
Throughout this section, the LSA modules are designed to directly satisfy the data path equation of $u[j+1] = f(u[j], z_{j+1})$ where z_{j+1} is the value of the result digit for step $j+1$.

2.2.1 Conventional versus LSA Modules: Overview

In both conventional broadcasting and the LSA approach, the data path for iterative arithmetic implementations can be described as a modular design. Figure 2.4 shows a comparison of conventional and LSA modules. Like conventional modules, each LSA module computes a unique set of weighted bits in the vectors of $u[j+1]$. For any given algorithm step, adjacent modules form adjacent bits according to the equation $u[j+1] = f(u[j], z_{j+1})$. Unlike conventional modules, however, LSA modules operate on different algorithm steps relative to other LSA modules where adjacent modules operate on adjacent steps. Hence in Figure 2.4, the values of c and e (similarly for b and d) will not refer to the same bit weight due to their reference to different algorithm steps.



(a) Conventional modules



(b) LSA modules

Figure 2.4: Conventional vs. LSA Modules

2.2.2 Data Dependencies Between LSA Modules

As in conventional modules, an LSA module requires data from other modules. Due to the step time differences between LSA modules, data can be required from modules operating at either a later time step $(j+T)$ or an earlier time step than the requesting module. This subsection shows the effects of both dependency types on the LSA design. Also presented are observations and proofs that derive the theoretical basis for the LSA conversion algorithm. The derivation is an application of the systolic array retiming theories presented in [LeSa83, KuLa84].

The first dependency type is illustrated in Figure 2.5. In this dependency it is given that module A operating on step $j+1$ requires data of step j from any upstream module B. Module B, meanwhile, is currently operating on step $j+i$, $i>1$ and would have computed its step j output i cycles earlier. This step time difference is resolved by implementing a pipeline of i latches. The desired data is thus stored until needed.

The second dependency type is shown in Figure 2.6 where module A must wait

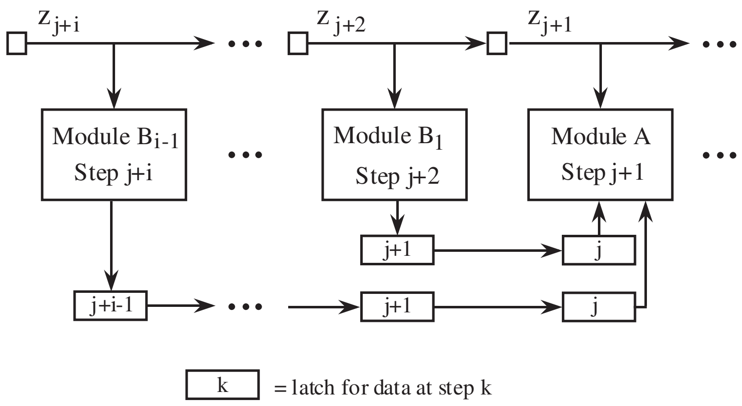


Figure 2.5: Transfer of Data to a Downstream Module

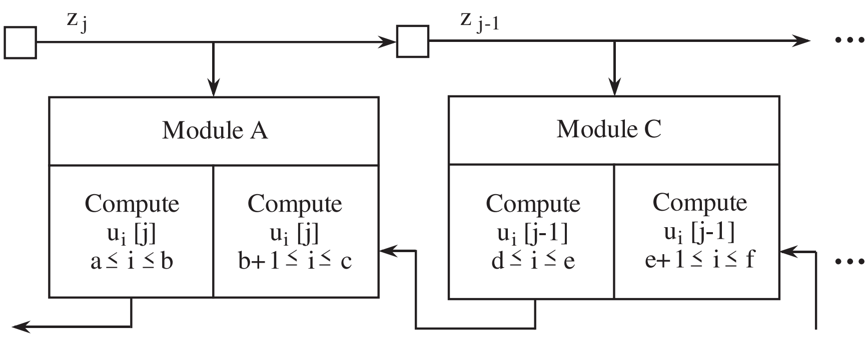


Figure 2.6: Transfer of Data to an Upstream Module

for data that a downstream module C has not yet computed. In Figure 2.6, $u_i[j]$, $b+1 \leq i \leq c$, of module A is dependent on $u_p[j-1]$, $d \leq p \leq e$, of its neighboring module C.

Theorem 1 shows that module A operating at step j may depend on downstream data from only its neighboring module, C, operating on step $j-1$. For this theorem and others presented in this subsection, it is given that:

1. At each cycle, an LSA module receives a result digit for only one algorithm step and generates its output for that step within the same cycle.
2. Adjacent LSA modules operate on adjacent algorithm steps.
3. Data computed by a module at step j for transmission to a module at step $j+1$ must be computable from data that is already latched and readily available.

4. The LSA modules implement the data path equation $w[j+1] = f(w[j], z_{j+1})$.

Theorem 1 *An LSA module operating on step j may receive data only from modules operating on step $j+i$ where $i \geq -1$.*

Proof: Define module A to be operating on step j while module B operates on step $j+i$. Let module A require data from module B where, from the fourth given, this data is of step $j-1$. Relative to generation of step $j-1$ data, three cases exist for module B's computational step:

case (1) $i \geq 0$: From the first given, if module B is currently operating on step $j+i$, module B must have already computed its output for step $j+i-1$. Therefore for $(i \geq 0)$, module B has completed step $j-1$ and the data is readily available through latching.

case (2) $i = -1$: Since module B has not yet completed step $j-1$, module A must wait for this data. As specified by the first given, module B, which is currently operating on step $j-1$, completes the step within the cycle. Since module A has not yet completed its algorithm step, module A must then receive this data and complete its operation on step j within the current cycle (according to the first given).

case (3) $i < -1$: For $i < -1$, module B is operating at most on step $j-2$. Since only one algorithm step can be computed per cycle, module B cannot compute the output of step $j-1$ in the given cycle. Since module A computes $w[j] = f(w[j-1], z_j)$ module A therefore cannot receive input from module B when $i < -1$.

Therefore, from cases (1), (2), and (3) the statement of Theorem 1 is shown to be true.
QED

To prevent the dependency type of Figure 2.6 from causing iterative dependencies that ripple through all downstream modules, data from step j that is calculated and then transmitted to an upstream module must be computable from currently latched values of step $j-1$. Theorem 2 demonstrates that this requirement limits the combinational logic delay of an LSA to twice that of a conventional module. Figure 2.7 shows the resulting two substages.

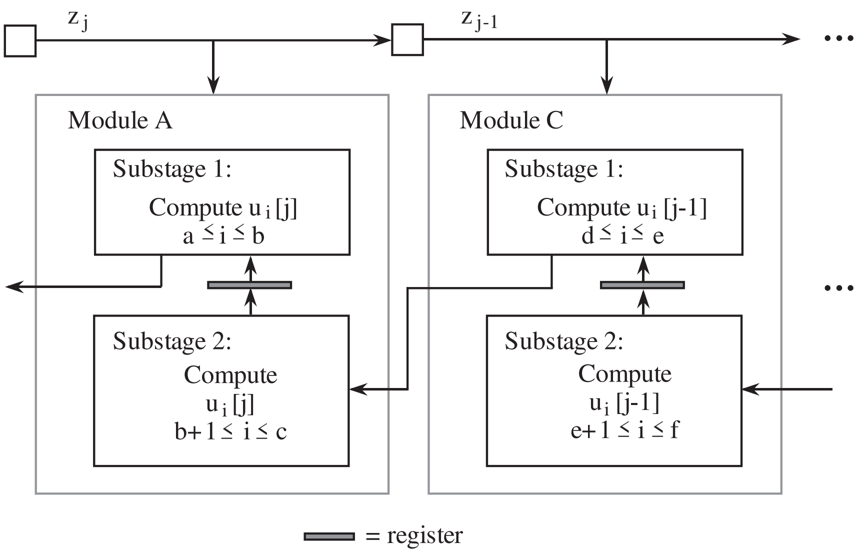


Figure 2.7: The Two LSA Module Substages

Theorem 2 *The logic delay of an LSA module is at most twice that of computing* $w[j] = f(w[j-1], z_j)$.

Proof: From Theorem 1, LSA module A operating on step j may receive data from modules operating on (1) step $j+i$, $i \geq 0$ or (2) step $j-1$. If module A requires data only from modules satisfying case 1, the data is readily available from latches; for this case, module A has only the delay of computing $w[j] = f(w[j-1], z_j)$. If, however, module A also requires data from module B operating on step $j-1$, module A must wait for the computation of the data it needs from $w[j-1] = f(w[j-2], z_{j-1})$. From the third given, the inputs to compute this data are readily available. Thus, the logic expression for the output of module A is summarized as $w[j] = f((w[j-1], f(w[j-2], z_{j-1})), z_j)$ which is twice the combinational logic delay of computing the general expression $w[t] = f(w[t-1], z_t)$ where $w[t-1]$ and z_t are readily available.

QED

Theorem 3 then shows that when downstream dependencies are satisfied, upstream dependencies can also be satisfied. Finally, Theorem 4 demonstrates that all LSA modules for a given set of data path equations are identical to enable simple modular designs.

Theorem 3 *When the input data to compute $v_i^k[j]$ from step $j-1$ are available, the inputs to compute $v_h^k[j]$ in the same LSA module are also available where $v_h^k[j]$ are upstream bits from $v_i^k[j]$.*

Proof: Let $v_h^k[j]$ and $v_i^k[j]$ be generated by module m. By definition, all bits of the data path vector $v^k[j]$ are computed with the same boolean expression but with skewed inputs; thus the computation of each bit of $v^k[j]$ can be represented as $v_{i+s}^k[j] = f(v_{i_1+s}^q[j-1], v_{i_2+s}^{q_1}[j-1], \ldots)$ where s represents the number of bits of skew from bit i. Since $v_h^k[j] = v_{i+s}^k[j]$ is upstream from the bit $v_i^k[j]$, all inputs for computing $v_h^k[j]$ either overlap or are upstream from the inputs for computing $v_i^k[j]$. Overlaps can occur when $v_i^k[j]$ is a function of contiguous bits in the same vector. Since inputs are available for $v_i^k[j]$, overlapping inputs are also available for $v_h^k[j]$. Regarding the non-overlapping inputs for $v_h^k[j]$, if these inputs are computed by module m, they were computed during the previous cycle. Similarly, if the inputs are generated by a module upstream from m, they were also derived in earlier cycles. In both cases the inputs are available through latches (delays). For the case that the inputs are from a downstream module, as specified by the third given, the data needed to compute the inputs transferred to module m must be currently available in latches. Since (1) the downstream inputs for $v_h^k[j]$ are computed from the same boolean expressions as the downstream inputs of $v_i^k[j]$, (2) the data to compute the downstream inputs transferred to module m are available, and (3) all inputs to compute $v_i^k[j]$ are available, it must therefore be true that the downstream data for $v_i^k[j]$ can also be generated and is available.
QED

Corollary 1 *If an LSA module design is based on satisfying the input requirements of the most downstream bit slice in the module, the requirements of upstream bits within the same module are also satisfied.*

Proof: For any given vector at step j, the corollary assumes that the inputs are available to compute its most downstream bit in a particular LSA module. Within the same module, all other bits of that vector for step j are upstream to the given bit. Thus, from Theorem 3 the inputs to compute the other bits of the vector for the LSA module are also available. *QED*

Theorem 4 *The modules in an LSA design are identical when all elements in a data vector are defined by the same boolean expression.*

Proof: From Corollary 1, an LSA module can be constructed by satisfying the input requirements for one bit slice along all vectors, where the bits in this slice are defined as the most downstream bits in the LSA module. Let this reference bit slice, s, belong to module A (Fig. 2.8). Define module B as the neighboring downstream module to module A. From Theorem 2, after two substages of computation, the input requirements of bit slice s are satisfied. The input requirements of bits upstream from s are also satisfied according to Theorem 3. Since s is the most downstream bit slice of module A, the two downstream substages that compute s belong to module B where those two substages contain the only input dependencies involved in computing data for upstream transfer to module A (from Theorem 3). From Theorem 2, let downstream dependencies of substage 1 be satisfied in substage 2. Let bit slice t represent the most downstream slice currently defined in the two substages and define t as the most downstream slice in module B (Fig. 2.8). Four cases exist regarding the computation of each bit, b, in slice t:

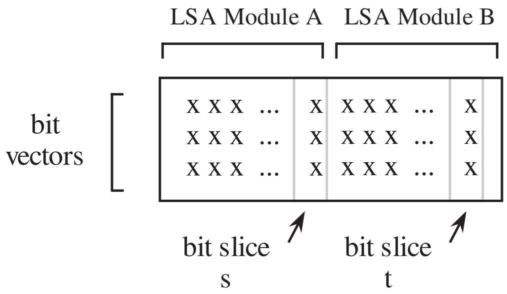


Figure 2.8: Downstream Bit Slices in Adjacent LSA Modules

case (1) The bit b must be computed in substage 2 as a result of satisfying the downstream dependencies of substage 1 and bit slice s.

case (2) The bit b is computed in substage 1 but has no downstream dependencies.

case (3) Bit b of slice t is not a downstream dependency of bit slice s, and thus is not specified by Theorem 2 for computation in either substage 1 or substage 2. In this situation, let the computation of b occur in substage 2 so that the downstream dependencies of bit b will be satisfied according to Theorems 1 and 2. Bit b's upstream dependencies will be satisfied via latched data.

case (4) There are no downstream dependencies in the data path equations. Only one computational substage (substage 2 which includes the latches) is required. In this case, latches will satisfy the upstream dependencies of any grouping of contiguous bits defining the LSA module.

In all of the above cases except for case 2, bit slice t is computed and latched entirely in substage 2. Those bits of slice t that are computed in substage 1 (case 2) have no downstream dependencies. Thus by Theorems 1 and 2, the downstream dependencies of bit slice t can be satisfied by another LSA module using two substages of computation. The LSA module that satisfies the dependencies of t will be identical to Module B since both were generated from one bit slice and use the same boolean expressions. Continuing this process, all successive modules using the same boolean expressions will be identical. Therefore, the creation of one LSA module suffices as a template for the other LSA modules in the same design.
QED

2.3 Conversion to LSA Modules

This section presents a general method to convert conventional modules to LSA modules. The algorithm satisfies all assumptions for the theorems presented in Section 2.2. To verify the conversion algorithm, various test data path equations have been converted to an LSA configuration where the results have been simulated and compared with the original broadcasting scheme. In this section, a simplified overview of the method's approach is first presented, followed by the algorithm's details and examples.

By definition, an LSA module computes a unique set of weighted bits for the step j. LSA construction begins with the minimum definition that one bit of the same weight from each vector in $u[j]$ is computed and latched within the same module. Satisfying the downstream dependency constraints (and Theorem 3) expands the number of bits computed by an LSA module and defines the computations in the first LSA substage. The second substage is then created by satisfying the downstream dependencies of data in the first substage. Finally, latches are added to satisfy upstream dependencies. The algorithm for conversion is given next.

2.3.1 Conversion Algorithm

Let $u = \{v^1, v^2, \ldots, v^M\}$ = the set of all bit vectors in the arithmetic algorithm's data path where M is the total number of vectors in the data path. Each bit in a given vector v^k is based on the same boolean expression. Let $v_b^k[j]$ represent a bit of weight 2^b in v^k at step j. Define three matrices, $S1$, $S2$, and $S3$, each having M rows. $S1$ and $S2$ represent the first and second substages, respectively, in an LSA module. $S3$ maintains a record of the number of latches (delays) required for passing data between different cycles. Each matrix row represents one vector, v^k. The matrix columns represent bit slices in the vectors. To represent the relative bit weights, the matrix columns are numbered increasingly from right to left. Bits in the matrices are denoted as $Sx_{k,c}$ where (1) x is either 1, 2, or 3 corresponding to matrix $S1$, $S2$, or $S3$, (2) k refers to the vector v^k, and (3) c denotes the bit of v^k of weight 2^c.

In the conversion algorithm, cells in matrices $S1$ and $S2$ are gradually marked to show the stage of maximum delay in which a particular bit must be available. Only downstream bits relative to the initial bit slice v_b^k are marked due to the constraints of receiving data from a downstream module. Once these constraints are satisfied, the remaining bits of the module are guaranteed by Theorem 3 of Appendix A to have available input data. The resulting marked bit slices in matrices $S1$, $S2$, and $S3$ define a single LSA module. If the computation of a bit slice does not have dependencies on downstream bits, the conversion algorithm will not mark any bits in $S1$ and $S2$. In this case, the minimum width of an LSA module is one bit and only one substage is needed. Latches, though, continue to be required for the passing of data to modules operating on delayed algorithm steps. The steps of the conversion algorithm are given below:

Initialization. Unmark all matrix cells (cell value = 0):

> For all k, c
>
> $S1_{k,c} = S2_{k,c} = S3_{k,c} = 0$

Step 1. Satisfy the downstream input dependencies of the initial latched bits:

> For all k, k' in $\{1, \ldots, M\}$
>
> if $v_b^{k'}[j] = f(v_{b+i}^k[j-1])$ then $S1_{k,b+i} = 1$
>
> where $i \leq -1$ if pipelining progresses from higher to lower weighted bits and $i \geq 1$ otherwise

Step 2. Mark dependencies on stage 1 by bits upstream from the initial latched bits.

Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
$a = 1$ otherwise
For each k in $\{1, \ldots, M\}$
{ Let L = index of farthest marked column in $S1$ from b for row k
For $c = b + a$ to L
$S1_{k,c} = 1$
}

Step 3. Satisfy the downstream input dependencies of marked bits in substage 1.

For all k, k' in $\{1, \ldots, M\}$ and $S1_{k',b+i} \neq 0$
if $v^{k'}_{b+i}[j-1] = f(v^{k}_{b+i+h}[j-2])$ then $S2_{k,b+i+h} = 1$
where $h \leq 0$, $i \leq -1$ if pipelining proceeds from higher to lower weighted bits and $h \geq 0$, $i \geq 1$ otherwise

Step 4. Since each bit is computed only once per cycle, unmark cells in $S2$ that are marked in both $S1$ and $S2$; also mark the cell's latch counter in $S3$.

For all k, c
if $S2_{k,c} = S1_{k,c} = 1$ then $S2_{k,c} = 0$ and $S3_{k,c} = 1$

Step 5. Define the remaining bits to be computed by the LSA module. Specify a latch if the remaining bits are dependent on downstream bits in stage 1.

Let L = index of farthest marked column from b of combined $S1$,$S2$
Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
$a = 1$ otherwise
For all k, k' in $\{1, \ldots, M\}$ and all i
For $c = b + a$ to L
if $S1_{k',c} = S2_{k',c} = 0$
then { $S2_{k',c} = 1$
if $((v^{k'}_{c}[j] = f(v^{k}_{c+a|i|}[j-1]))$ and $(S1_{k,c+a|i|} = 1))$
then $S3_{k,c+a|i|} = 1$ }

Step 6. Mark latches for all bits in the second step.

> For all k, c
>
> if $S2_{k,c} = 1$ then $S3_{k,c} = 1$

Step 7. Satisfy the upstream input dependencies and record the maximum number of required latches.

> Let L = index of farthest marked column from b of combined $S1$,$S2$
> Let $a = -1$ if pipelining proceeds from higher to lower weighted bits;
> $a = 1$ otherwise
> For all k, k' in $\{1, \ldots, M\}$ and all i
> For $c = 0$ to $(b - L + a)$
> { if $v^{k'}_{L+c}[j] = f(v^k_{L+c+i}[j-1])$ where
> $i \geq 0$ if pipelining from higher to lower weights,
> and $i \leq 0$ otherwise
> Let $e = \text{mod}\ (|c+i|, |L-b|)$
> Let $d = \lfloor |c+i|/|L-b| \rfloor + 1$
> $S3_{k,L-ae} = \max\ (d, S3_{k,L-ae})$
> }

The marked cells provide a template for each LSA module where the number of columns spanned by nonzero cells in $S1$ and $S2$ defines the module's incremental precision. Marked (nonzero) cells in matrices $S1$ and $S2$ identify in which stage, respectively, that each bit must be computed. The first substage generates data that is always transferred to an upstream module but can also be stored for downstream modules or the module that generated it. Likewise, the second substage generates data that is stored for downstream modules or the module that generated it. The combinational logic of the combined substages is identical to that used by a conventional module. Since no additional combinational logic is implemented in an LSA design, the differences between LSA hardware and conventional modules are the latching schemes and interconnections.

2.3.2 Examples

As an example, an LSA design is created for SRT division [Rob58]. The SRT data path consists of three vectors, v^1 (sum), v^2 (carry), and v^3 (multiple of the divisor) where v_3 is a function of both the result digit and divisor (d). The result, z, is pipelined from higher to lower weighted bits. Only downstream and result

$$v_i^1[j+1] = f_1(v_{i-1}^1[j], v_{i-1}^2[j], v_{i-1}^3[j])$$
$$v_i^2[j+1] = f_2(v_{i-2}^1[j], v_{i-2}^2[j], v_{i-2}^3[j])$$
$$v_i^3[j+1] = f_3(z_{j+1}, d_i)$$

(a) Data path equations

$$\begin{array}{ccc}
S3\begin{bmatrix} 0\,0\,0\,0 \\ 0\,0\,0\,0 \\ 0\,0\,0\,0 \end{bmatrix} & S3\begin{bmatrix} 0\,1\,0\,0 \\ 0\,1\,0\,0 \\ 0\,1\,0\,0 \end{bmatrix} & S3\begin{bmatrix} 0\,1\,1\,1 \\ 0\,1\,1\,1 \\ 0\,1\,1\,1 \end{bmatrix}\begin{array}{l} \leftarrow v^1 \\ \leftarrow v^2 \\ \leftarrow v^3 \end{array} \\
S2\begin{bmatrix} 0\,1\,1\,1 \\ 0\,1\,1\,1 \\ 0\,1\,1\,1 \end{bmatrix} & S2\begin{bmatrix} 0\,0\,1\,1 \\ 0\,0\,1\,1 \\ 0\,0\,1\,1 \end{bmatrix} & S2\begin{bmatrix} 0\,0\,1\,1 \\ 0\,0\,1\,1 \\ 0\,0\,1\,1 \end{bmatrix} \\
S1\begin{bmatrix} 1\,1\,0\,0 \\ 1\,1\,0\,0 \\ 1\,1\,0\,0 \end{bmatrix} & S1\begin{bmatrix} 1\,1\,0\,0 \\ 1\,1\,0\,0 \\ 1\,1\,0\,0 \end{bmatrix} & S1\begin{bmatrix} 1\,1\,0\,0 \\ 1\,1\,0\,0 \\ 1\,1\,0\,0 \end{bmatrix}
\end{array}$$

matrix column b-1   matrix column b-4

(b) After steps 1 - 3   (c) After step 4 (same after step 5)   (d) After step 6 (same after step 7)

Figure 2.9: Development of an LSA Module for SRT Division

digit dependencies are present in this example. Figures 2.9 and 2.10 show the output matrices and LSA design after applying the conversion algorithm. In the LSA design of Figure 2.10, the vectors are labeled s (sum vector, v^1), c (carry vector, v^2), and D (multiple of the divisor, v^3) where the superscripts denote the algorithm step from which each bit is derived.

A second example is shown in Figure 2.11. While this example does not represent any specific arithmetic data path, it does demonstrates all steps in the conversion. Again, the result digits are passed from higher to lower weighted bits.

2.3.3 Multi-output Operators

The conversion algorithm creates basic LSA modules for any given set of arithmetic data path vectors. To maintain simplicity, the algorithm does not identify multi-output operators (multiple functions using the same group of inputs, e.g., full adder). Thus, the individual functions of the multi-output operation may be placed on different modules where downstream computations are delayed. Latches must then store their common inputs for several cycles. Let H represent the

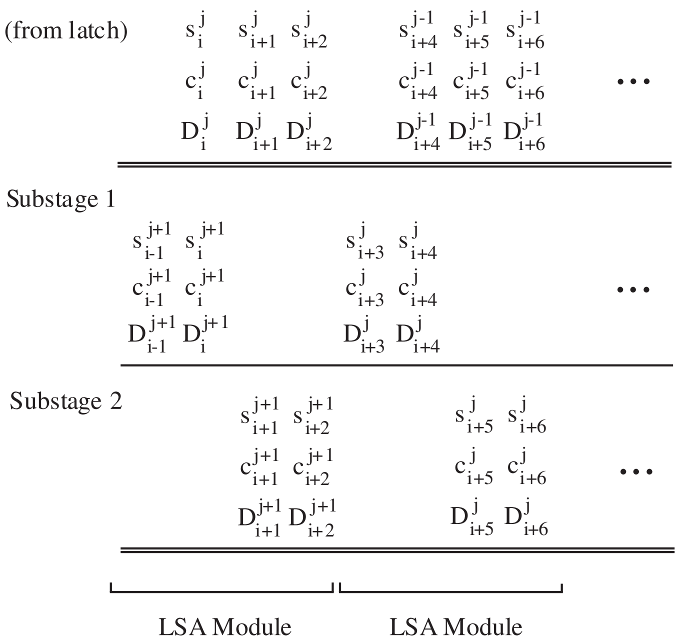


Figure 2.10: LSA Design for the SRT Division Example

most upstream LSA module associated with a given multi-output operator. If the number of shared inputs is greater than or equal to the number of delayed outputs generated downstream from H, rejoining the outputs within the same LSA module can reduce the number of latches and conserve area. To satisfy all data dependencies, rejoining of multiple outputs occurs at module H, the most upstream LSA module among the group. With rejoining, latches for delaying the common input data can be removed. Additional latches, however, are needed to delay the rejoined outputs to the time specified by the LSA conversion algorithm.

Examples of separated multi-output bits in Figure 2.10 are the pairs of sum and carry bits $(s^{j}_{i+3}, c^{j+1}_{i+2})$ and $(s^{j-1}_{i+7}, c^{j}_{i+6})$. The SRT LSA design after rejoining these bits is shown in Figure 2.12 where the superscripts denote the algorithm step of each bit. For the SRT example, rejoining the separated bits saves two latches per module.

$$v_i^1[j+1] = f_4(v_{i-1}^2[j], v_{i-2}^2[j], v_{i-3}^2[j], v_{i+3}^3[j])$$
$$v_i^2[j+1] = f_5(v_{i-1}^1[j], v_{i-2}^1[j], v_{i+8}^3[j])$$
$$v_i^3[j+1] = f_6(v_i^2[j])$$

(a) Data path equations

$$S3\begin{bmatrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix}\quad S2\begin{bmatrix}0&1&1&1&1\\0&1&1&1&1\\0&0&0&0&0\end{bmatrix}\quad S1\begin{bmatrix}1&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}$$

matrix column b-1   matrix column b-5

(b)
After steps 1 - 3

$$S3\begin{bmatrix}0&1&0&0&0\\0&1&1&0&0\\0&0&0&0&0\end{bmatrix}\quad S2\begin{bmatrix}0&0&1&1&1\\0&0&0&1&1\\0&0&0&0&0\end{bmatrix}\quad S1\begin{bmatrix}1&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}$$

(c)
After step 4

$$S3\begin{bmatrix}0&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}\quad S2\begin{bmatrix}0&0&1&1&1\\0&0&0&1&1\\1&1&1&1&1\end{bmatrix}\quad S1\begin{bmatrix}1&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}$$

(d)
After step 5

$$S3\begin{bmatrix}0&1&1&1&1\\1&1&1&1&1\\1&1&1&1&1\end{bmatrix}\quad S2\begin{bmatrix}0&0&1&1&1\\0&0&0&1&1\\1&1&1&1&1\end{bmatrix}\quad S1\begin{bmatrix}1&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}$$

(e)
After step 6

$$\begin{array}{cc} S3\begin{bmatrix}0&1&1&1&1\\1&1&1&1&1\\2&2&3&3&3\end{bmatrix} & \begin{array}{l}\leftarrow v^1\\ \leftarrow v^2\\ \leftarrow v^3\end{array}\end{array}\quad S2\begin{bmatrix}0&0&1&1&1\\0&0&0&1&1\\1&1&1&1&1\end{bmatrix}\quad S1\begin{bmatrix}1&1&0&0&0\\1&1&1&0&0\\0&0&0&0&0\end{bmatrix}$$

(f)
After step 7

Figure 2.11: Conversion Example

(from latch)

$$\begin{array}{ccccccccc} s_{i+1}^{j} & s_{i+2}^{j} & s_{i+3}^{j} & & s_{i+5}^{j-1} & s_{i+6}^{j-1} & s_{i+7}^{j-1} & & \\ c_{i+1}^{j} & c_{i+2}^{j} & & & c_{i+5}^{j-1} & c_{i+6}^{j-1} & & & \cdots \\ D_{i+1}^{j} & D_{i+2}^{j} & & & D_{i+5}^{j-1} & D_{i+6}^{j-1} & & & \end{array}$$

Substage 1

$$\begin{array}{cccccc} & s_{i}^{j+1} & & & s_{i+4}^{j} & \\ c_{i-1}^{j+1} & c_{i}^{j+1} & & c_{i+3}^{j} & c_{i+4}^{j} & \cdots \\ D_{i-1}^{j+1} & D_{i}^{j+1} & & D_{i+3}^{j} & D_{i+4}^{j} & \end{array}$$

Substage 2
(to latch)

$$\begin{array}{ccccccccc} s_{i+1}^{j+1} & s_{i+2}^{j+1} & s_{i+3}^{j+1} & & s_{i+5}^{j} & s_{i+6}^{j} & s_{i+7}^{j} & & \\ c_{i+1}^{j+1} & c_{i+2}^{j+1} & & & c_{i+5}^{j} & c_{i+6}^{j} & & & \cdots \\ D_{i+1}^{j+1} & D_{i+2}^{j+1} & & & D_{i+5}^{j} & D_{i+6}^{j} & & & \end{array}$$

Figure 2.12: The LSA for SRT Division after Rejoining Full Adder Outputs

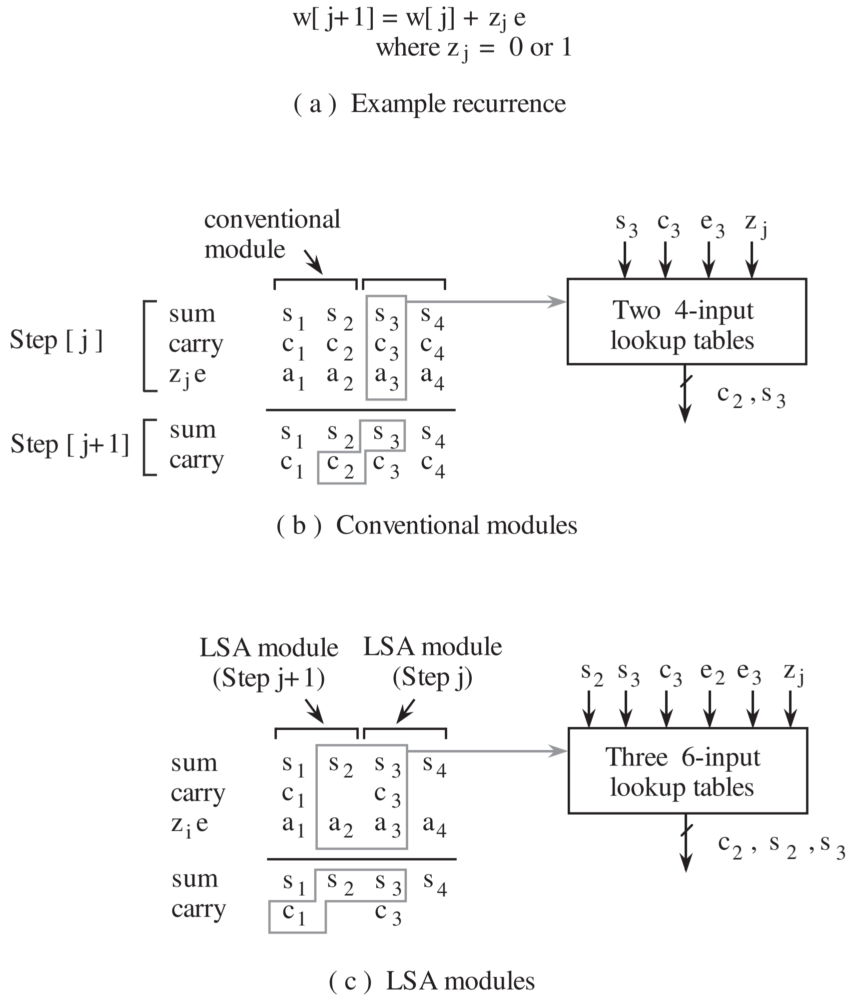


Figure 2.13: Conventional and LSA Modules for an Iterative Addition Example

2.4 A Comparison of LSA and Conventional Modules

Regarding hardware requirements, the combinational logic of both the LSA and a conventional design are identical because both use the same boolean expressions. For the LSA, however, some additional opportunities for combinational logic minimization may occur at the interface of neighboring modules. A logic minimized LSA module may therefore require less combinational logic than its corresponding conventional module. An example is given in Figure 2.13 in which a simple iterative addition is presented. For this example, only 6-input lookup tables are available in each logic cell. Implementing two modules with the conventional broadcasting approach (Fig. 2.13b) requires eight 6-input logic blocks, one for each s_i or c_i; the functionality of 4-input lookup tables is minimally required, but due to the constraints specified by the example, 6-input tables must be used. Implementing two LSA modules for the same recurrence (Fig. 2.13c) requires only six 6-input logic blocks, a 25% savings in area. In support of sequential logic, however, the LSA generally requires more latches than a conventional module for both pipelining the result digits and for saving data to be transferred downstream.

With respect to data path delay, the two LSA substages at most doubles the combinational logic delay over that of conventional modules. In trade, the LSA does not experience the additional broadcasting delays of conventional designs. The LSA can therefore improve performance whenever broadcasting exceeds the delay of the data path's combinational logic.

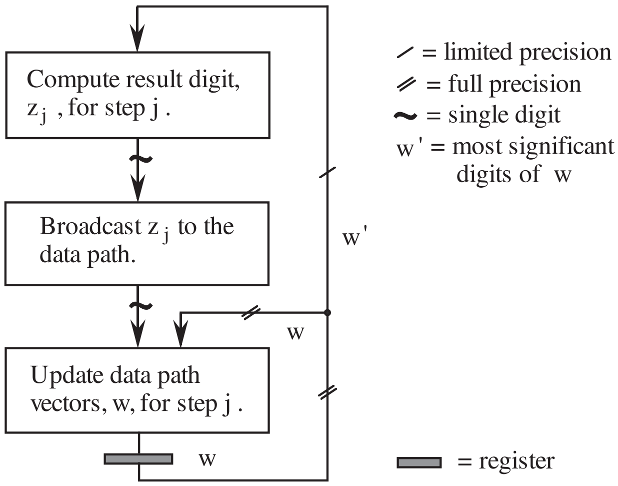


Figure 2.14: Arithmetic Model

The use of result digit prediction schemes [ErLa85, LoEr92, LoEr93b, LoEr94c, MoCi94] helps to remove the LSA delay from the critical path. In most digit-recurrence arithmetic algorithms that do not feature prediction, each cycle can be modeled (Fig. 2.14) as three successive steps: (1) computation of the result digit, (2) distribution of the result digit to the data path, followed by (3) updating of the residual. Result digit prediction, alternatively, enables the distribution of the result digit and updating of the data path to occur simultaneously with computation of the next iteration's result digit. By combining result digit prediction with the bounded LSA delay, the data path can often be completely removed from the critical path. The iterative arithmetic model using both the LSA and prediction appears in Figure 2.15. A simple method for creating a prediction scheme is discussed in Chapter 4.

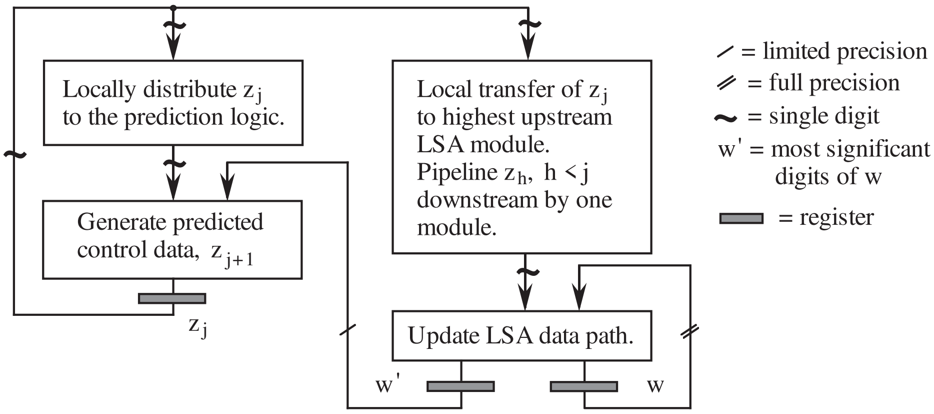


Figure 2.15: Arithmetic Model with an LSA and Prediction

The LSA technique is especially useful with Field Programmable Gate Array (FPGA) technology where broadcasting delays can equal a logic stage delay at precisions as low as 25 bits [LoEr93b]. Due to the many programmable interconnection points, interconnection delays within an FPGA grow rapidly as a function of fanout and routing distance. By using the LSA, routing delays along the data path remain small and localized. As demonstrated with an SRT divider [LoEr93b] (Chapter 5) and a radix-2 square rooter [LoEr94d] (Chapter 6), combining the LSA with result digit prediction can remove the data path from the critical path for any precision.

2.5 Multiple Chip LSA Design

As precision increases, the LSA design must eventually span multiple chips. The additional routing delays caused by interchip communication can be accommodated by precomputing the transferred data. This section develops the LSA chip interface modules [LoEr94d] that can enable LSA performance as if all modules are within a single chip. Thus, with the interface module, operation latency can be independent of interchip communication delay while cycle time remains independent of precision. The interface is placed only on the appended FPGA chip. LSA modules are attached to the interface module to continue expanding the precision of the overall design (Fig. 2.16).

The LSA interface is developed for two interconnect cases. The first and most

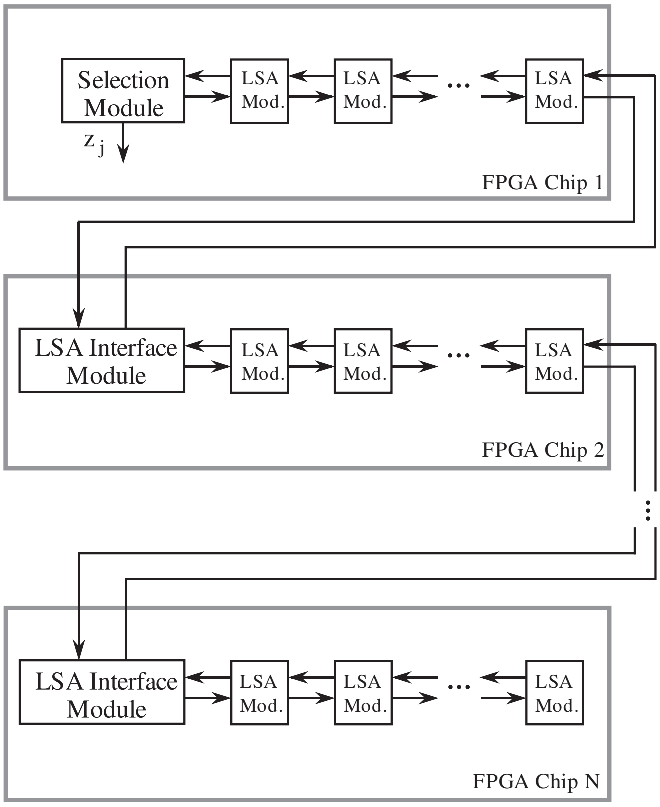


Figure 2.16: Multiple Chip LSA Design

basic case assumes nearest neighbor connections where either (1) the transfer time between chips (I/O buffer to I/O buffer) is at most equal to the cycle time of the LSA or (2) the interconnection structure between chips is pipelined every clock cycle. In a second case, it is assumed that pipelining is unavailable and that data can be passed only every d clocks. The second case is then easily extended to a bus structure in which an application-specific upper bound on transfer delay is known but the actual transfer time varies below the maximum bound.

2.5.1 Case 1: Local Interconnections with Pipelining

The following steps occur within a single cycle when LSA modules k and $k+1$ are located on the same chip: (1) module $k+1$ computes and transfers step j data to module k, and (2) module k uses this data within the same cycle to generate data for step $j+1$. When modules k and $k+1$ are separated across different chips, module k cannot complete the computation of step $j+1$ data until it receives step j data from module k. To allow module k to proceed without performance degradation, module $k+1$ must derive and transfer its data early. Module $k+1$ is similarly dependent on data passed from module k, but the pipelining startup delay prevents module k from compensating for interchip communication delays. The chip interface module must therefore obtain data from a more upstream module than k and then more rapidly compute the output of module k. Figure 2.17 illustrates the interconnection model.

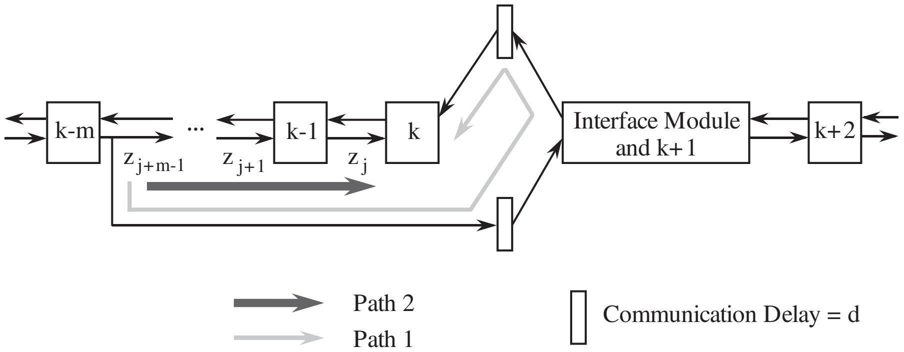


Figure 2.17: The LSA Design with Chip Interface

The chip interface module handles all LSA data transferred between chips where it compensates for the interconnection time and computes the output of LSA module $k+1$. The fixed communication delay, d, represents the transfer time (in cycles) between chips and includes delays for transferring data from the LSA buffers to the chip I/O ports as well as for interchip communication.

The LSA chip interface module uses a conventional broadcasting scheme to avoid the pipelining startup delay and to rapidly compute the output of module k. The logic within the chip interface mimics the computation from LSA modules $k-m+1$ through k. By using a conventional broadcasting scheme within the chip interface, each of the mimicked modules and module $k+1$ operate on the same algorithm step simultaneously. If m is sufficiently large that broadcasting increases the critical path delay of the implementation, the chip interface module can be broken into two or more broadcasting submodules of smaller size. The adjoining submodules would then be interconnected via pipelining as in the LSA modules.

In the first case, data enters the interconnection every clock cycle, and pipelining between chips occurs when $d > 1$. To prevent degradation to the latency of the operation, the LSA chip interface must simultaneously satisfy both the upstream and downstream data dependencies at module k (similarly $k+1$). If d clock cycles are needed to transmit data between LSA modules on different chips, the LSA chip interface module (Fig. 2.17) must generate module $k+1$'s output d cycles in advance. Likewise, the inputs for the chip interface module must be sent an additional d cycles in advance. Two dependency paths arise (Fig. 2.17) since data sent to the interface module on the adjoining chip continues through the pipeline on the originating chip. Both paths must reach module k at the same time for the same step j. Path 2 experiences a delay of $m-1$ cycles. For path 1, let c represent the pipelining time (in cycles) of the interface module where $c>1$ if a combination of conventional broadcasting and LSA pipelining is used, and $c=1$ if only broadcasting is used. In a pure broadcasting scheme, all modules operate on the same algorithm step, and the registers contain data for the same step. With a pure broadcasting approach in the chip interface module, the I/O buffers to and from the chip interface contain data for the same algorithm step, and path 1 has a resulting delay of $2d-1$. Generalizing with respect to c, the I/O buffers to and from the chip interface have a relative delay of $c-1$ algorithm steps. Therefore, path 1 of Figure 2.17 has a pipelining delay of $2d+c-2$ cycles. The interface must then satisfy the equation:

$$m = 2d + c - 1 \tag{2.1}$$

Figure 2.18 presents a generalized timing diagram tracing the position of data for step j through LSA module latches and an LSA chip interface.

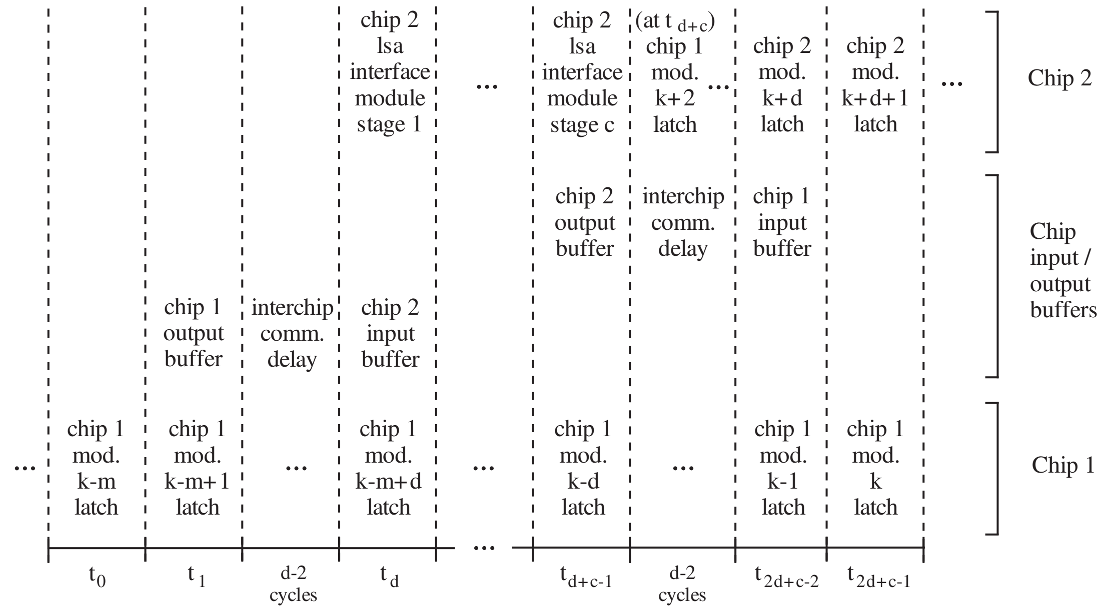


Figure 2.18: Timing Diagram of Latched Data for Step j

Figure 2.19 provides an example of the LSA modules and interface when $d = 2$ and $c = 1$. A snapshot of the chip interface's operation during a given cycle is shown. In Figure 2.19, each LSA module is labeled with its current computational step, and each buffer indicates the algorithm step for which its data is intended.

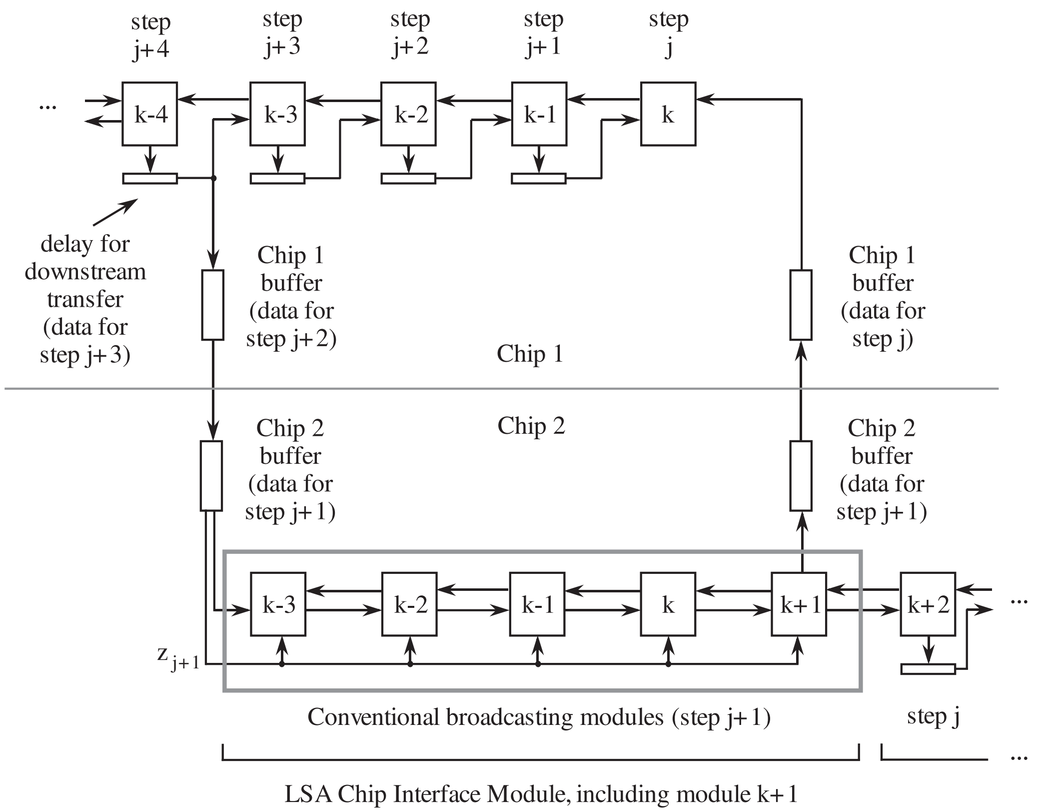


Figure 2.19: LSA and Chip Interface Example ($d = 2$, $c = 1$)

2.5.2 Case 2: Interconnections without Pipelining

When pipelining is unavailable in the chip interconnection scheme, data can be transferred only every d cycles. To support continuous processing in the adjoining chip, blocks of data for d cycles of operation should be simultaneously transferred. Figure 2.20 shows the buffering before transfer to the adjoining chip. The buffer saves all data transferred downstream to module $k - M + 1$. In Figure 2.20, the interface module receives from the buffer: (1) d cycles of latched output from module $k - M$ and (2) delayed data that passed through module $k - M$ during the d-cycle accumulation period. Similarly d cycles of output data from module $k + 1$ are accumulated within the interface and passed upstream to module k.

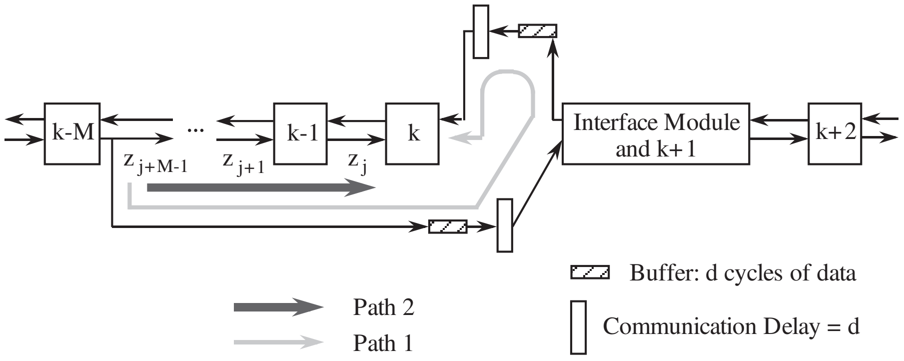


Figure 2.20: LSA with Chip Interface and Buffering

For this interconnection scheme, the LSA chip interface module must compensate for both the communication delay, d, and the d cycles to accumulate the buffered data. Hence, the following equation must be satisfied:

$$M = 4d + c - 1 \tag{2.2}$$

Case 2 forms the basis for a bus scheme in which the application-specific upper bound on communication delay is known in advance. Since a bus is shared by several or all of the chips, interconnection time is variable but not longer than the upper bound of d clock cycles. The buffering approach ensures that input data is always available at the receiving chip, but the variable delay of a bus may allow a set of data to arrive early. A modulo-d counter at the receiving chip is then needed to provide a variable delay on the arriving inputs.

As in Case 1, the LSA chip interface module may use either broadcasting or a combination of broadcasting and pipelining. For both cases, broadcasting within the interface allows the LSA to proceed across multiple chips without increasing the cycle time.

CHAPTER 3

An LSA for a Radix-16 Multiplier

For many digit-recurrence algorithms, the complexity of the selection function causes a delay greater than that of an LSA. For digit-recurrence multiplication, however, the digit selection consists of either the direct transfer or a simple recoding of the multiplier bits[Hwan79], and thus the selection function experiences a lesser delay than the two steps of the LSA. This chapter presents an LSA variation for multiplication in which the combinational logic delay remains identical to that of a conventional broadcasting scheme. In the presented variation, the LSA features of small fanout and localized routing are maintained. A design for a variable precision radix-16 multiplier with Booth recoding is developed, and performance comparisons with a software implementation on the Cray 2 and a microprocessor implementation are given [LoEr94a]. Application of this design to inner product computations is also proposed.

3.1 Notation

The following notation is used throughout this chapter. Additional notation is defined as needed.

x	two's complement multiplicand
y	two's complement multiplier
$w[j]$	accumulated product at step j in carry save form
w_j	bit of $w[j]$ of weight 2^j
s_i	sum bit of $w[j]$ in carry save form of weight 2^i
c_i	carry bit of $w[j]$ in carry save form of weight 2^i
c'_i	intermediate carry bit of $w[j]$ from 4-2 add of weight 2^i
c''_i	carry bit of $w[j]$ from 4-2 add of weight 2^i

3.2 Multiplier Overview

This chapter explores an add-and-shift multiplier that is similar to the conventional pencil-and-paper technique. Figure 3.1 provides a general description of a radix-16 add-and-shift multiplier that uses Booth recoding [Booth51] of the multiplier operand.

Recurrence:

$$w[j+1] = \frac{1}{16}(w[j] + x16^e y_j) \text{ for } j = 0, 1, \ldots, e-1$$

where

$w[j]$=accumulated product at step j (generally in carry save form),

$w[0] = 0$

P = product = carry assimilated form of $w[e]$,

y_j = radix-16 recoded digit of weight 16^j of y,

$\frac{1}{16}$ = a right shift by 4 bits.

Figure 3.1. Recurrence for Iterative Add-and-Shift Multiplication

The uniform structure of an iterative add-and-shift approach allows simple modular designs. Unlike non-iterative approaches such as trees [Hwan79, CS252a], precision on an LSA add-and-shift design can be increased by simply appending modules. Iterative approaches also require less area than non-iterative approaches, a feature that is important to the limited logic resources within an FPGA. For this study, the standard add-and-shift technique is chosen over other iterative multiplication methods because it requires simple selection logic and hence a smaller cycle time. On a radix-2 add-and-shift multiplier, for example, only one multiplier bit is needed to determine the multiple to be accumulated.

To save area, Booth recoding of the multiplier [Booth51] is employed. Booth recoding slightly increases the complexity of the selection logic, but it provides (1) area savings due to simplified radix-4 multiple generation [CS252a] and (2) uniform handling of positive and negative operands [Hwan79]. Figure 3.2 describes the Booth recoding algorithm. The presented design uses two radix-4 digits per step thus effectively implementing a radix-16 multiplier.

Recoding Equation:

$$y_{-1} = 0$$

$$z_j = y_{2j} + y_{2j-1} - 2y_{2j+1} \text{ for } j = 0, 1, \ldots, m-1$$

where

$Y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ = 2's complement multiplicand, $\quad y_i \in \{0, 1\}$

$Z = (z_{m-1}, z_{m-2}, \ldots, z_0)$ = recoded radix-4 multiplier,

$m = \lceil \frac{n}{2} \rceil \quad z_i \in \{-2, -1, 0, 1, 2\}$

Figure 3.2. Booth Recoding Algorithm for Radix-4

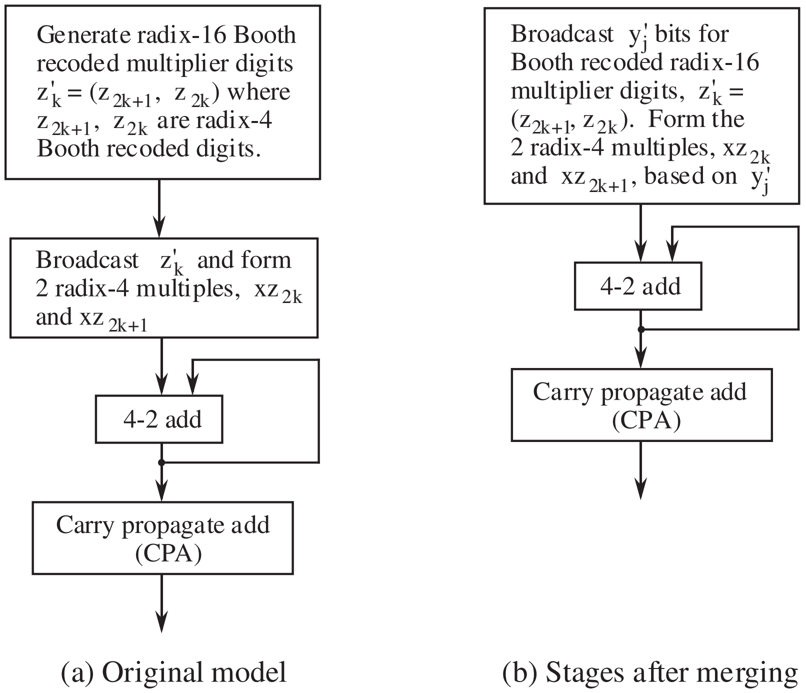
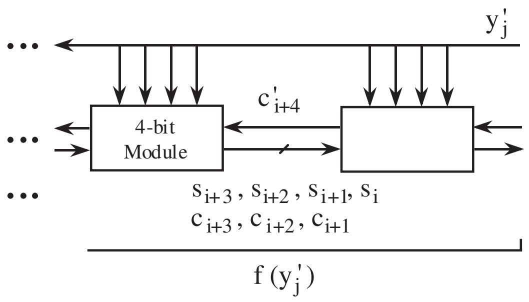


Figure 3.3: Multiplier Model Before Developing the LSA

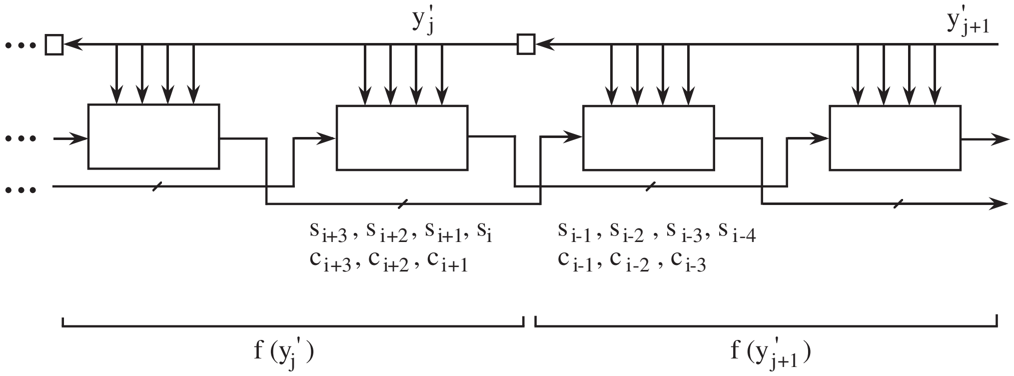
3.2.1 LSA Modules for the Multiplier

This study creates radix-16 LSA modules for a recoded add-and-shift multiplier. Figure 3.3a shows the multiplier model before implementing the linear sequential array [CS252a, Hwan79]. At each step, j, the generation of $w[j]$, z'_{j+1} (a radix-16 recoded digit consisting of two radix-4 Booth recoded digits), and xz'_j (the multiple) are performed simultaneously and pipelined. With the Xilinx XC4010 logic blocks (CLBs), the delay of each of the model's pipelined stages can be limited to one CLB logic level plus interconnection and latching time. To reduce the pipeline's start-up delay, the computation of the recoded multiplier digit and radix-4 digit of xy_j are combined into a single stage (Fig. 3.3b). In the approach of Fig. 3.3b only five bits of y_j (designated as y'_j) are broadcast for two digits of Booth recoding whereas the approach of Fig. 3.3a requires broadcasting of six bits to represent the two radix-4 digits of z'_{j+1}.

Figure 3.4 compares conventional modules with LSA modules for a radix-16 add-and-shift multiplier. Like conventional modules, each LSA module computes a unique set of weighted bits of the vector $w[j+1]$. An LSA approach, however, reduces the distribution delay of the multiplier bits to reduce the cycle time needed by the data path. Figure 3.5 shows the result after conversion to an LSA scheme.



(a) Conventional modules



(b) LSA modules

Figure 3.4: Radix-16 Conventional vs. LSA Multiplier Modules

The superscripts in Figure 3.5 denote the algorithm step of the corresponding bit. A_i^j and B_i^j (Fig. 3.5) denote bits of the accumulated multiples xz_{2j} and xz_{2j+1} where z_{2j} and z_{2j+1} represent radix-4 recoded digits at step j.

Minimizing the interconnections between LSA modules simplifies the transfer of data across multiple chips. For example, when the transfer of data to a downstream module is limited to the pipelined control digit, the chip interface module is not required to mimic any of the upstream LSA modules. A variation of the two-stage LSA multiplier module (Fig. 3.6) thus shifts the carry bits c_i' and c_i'' and utilizes them within the same module at the next cycle. With this variation only the control digit is passed to a downstream module.

Each cycle of Figure 3.6 requires two 4-2 addition substages (4 input vectors summed to 2 output vectors) where a substage consists of a single XC4010 logic level. Observe in Figure 3.6 that the two substages are closely coupled where output data from each substage is right-shifted by 4 bits and input to its adjacent

(From latch)		$A^{j-2}_{i+11}\ A^{j-2}_{i+10}\ A^{j-2}_{i+9}\ A^{j-2}_{i+8}\ A^{j-2}_{i+7}\ A^{j-2}_{i+6}$		$A^{j-1}_{i+3}\ A^{j-1}_{i+2}\ A^{j-1}_{i+1}\ A^{j-1}_{i}\ A^{j-1}_{i-1}\ A^{j-1}_{i-2}$	
	…	$B^{j-2}_{i+11}\ B^{j-2}_{i+10}\ B^{j-2}_{i+9}\ B^{j-2}_{i+8}\ B^{j-2}_{i+7}\ B^{j-2}_{i+6}$		$B^{j-1}_{i+3}\ B^{j-1}_{i+2}\ B^{j-1}_{i+1}\ B^{j-1}_{i}\ B^{j-1}_{i-1}\ B^{j-1}_{i-2}$	…
		$s^{j-2}_{i+11}\ s^{j-2}_{i+10}\ s^{j-2}_{i+9}\ s^{j-2}_{i+8}\ s^{j-2}_{i+7}\ s^{j-2}_{i+6}$		$s^{j-1}_{i+3}\ s^{j-1}_{i+2}\ s^{j-1}_{i+1}\ s^{j-1}_{i}\ s^{j-1}_{i-1}\ s^{j-1}_{i-2}$	
		$c^{j-2}_{i+11}\ c^{j-2}_{i+10}\ c^{j-2}_{i+9}\ c^{j-2}_{i+8}\ c^{j-2}_{i+7}$		$c^{j-1}_{i+3}\ c^{j-1}_{i+2}\ c^{j-1}_{i+1}\ c^{j-1}_{i}\ c^{j-1}_{i-1}$	
Substage 1			$A^{j-1}_{i+7}\ A^{j-1}_{i+6}\ A^{j-1}_{i+5}\ A^{j-1}_{i+4}$		
			$B^{j-1}_{i+7}\ B^{j-1}_{i+6}\ B^{j-1}_{i+5}\ B^{j-1}_{i+4}$		
	…		$s^{j-1}_{i+7}\ s^{j-1}_{i+6}\ s^{j-1}_{i+5}\ s^{j-1}_{i+4}$		…
			$c^{j-1}_{i+7}\ c^{j-1}_{i+6}\ c^{j-1}_{i+5}\ c^{j-1}_{i+4}$		
Substage 2 (latched)		$A^{j-1}_{i+11}\ A^{j-1}_{i+10}\ A^{j-1}_{i+9}\ A^{j-1}_{i+8}$		$A^{j}_{i+3}\ A^{j}_{i+2}\ A^{j}_{i+1}\ A^{j}_{i}$	
	…	$B^{j-1}_{i+11}\ B^{j-1}_{i+10}\ B^{j-1}_{i+9}\ B^{j-1}_{i+8}$		$B^{j}_{i+3}\ B^{j}_{i+2}\ B^{j}_{i+1}\ B^{j}_{i}$	…
		$s^{j-1}_{i+11}\ s^{j-1}_{i+10}\ s^{j-1}_{i+9}\ s^{j-1}_{i+8}$		$s^{j}_{i+3}\ s^{j}_{i+2}\ s^{j}_{i+1}\ s^{j}_{i}$	
		$c^{j-1}_{i+11}\ c^{j-1}_{i+10}\ c^{j-1}_{i+9}\ c^{j-1}_{i+8}$		$c^{j}_{i+3}\ c^{j}_{i+2}\ c^{j}_{i+1}\ c^{j}_{i}$	

Figure 3.5: Accumulator After LSA Conversion

(From latch)		$A^{j-2}_{i+11}\ A^{j-2}_{i+10}\ A^{j-2}_{i+9}\ A^{j-2}_{i+8}$		$A^{j-1}_{i+3}\ A^{j-1}_{i+2}\ A^{j-1}_{i+1}\ A^{j-1}_{i}$	
	…	$B^{j-2}_{i+11}\ B^{j-2}_{i+10}\ B^{j-2}_{i+9}\ B^{j-2}_{i+8}$		$B^{j-1}_{i+3}\ B^{j-1}_{i+2}\ B^{j-1}_{i+1}\ B^{j-1}_{i}$	…
		$(c')^{j-2}_{i+12}\ s^{j-2}_{i+11}\ s^{j-2}_{i+10}\ s^{j-2}_{i+9}\ s^{j-2}_{i+8}$		$(c')^{j-1}_{i+4}\ s^{j-1}_{i+3}\ s^{j-1}_{i+2}\ s^{j-1}_{i+1}\ s^{j-1}_{i}$	
		$(c'')^{j-2}_{i+12}\ c^{j-2}_{i+11}\ c^{j-2}_{i+10}\ c^{j-2}_{i+9}\ c^{j-2}_{i+8}$		$(c'')^{j-1}_{i+4}\ c^{j-1}_{i+3}\ c^{j-1}_{i+2}\ c^{j-1}_{i+1}\ c^{j-1}_{i}$	
Substage 1			$A^{j-1}_{i+7}\ A^{j-1}_{i+6}\ A^{j-1}_{i+5}\ A^{j-1}_{i+4}$		
			$B^{j-1}_{i+7}\ B^{j-1}_{i+6}\ B^{j-1}_{i+5}\ B^{j-1}_{i+4}$		
	…		$(c')^{j-1}_{i+8}\ s^{j-1}_{i+7}\ s^{j-1}_{i+6}\ s^{j-1}_{i+5}\ s^{j-1}_{i+4}$		…
			$(c'')^{j-1}_{i+8}\ c^{j-1}_{i+7}\ c^{j-1}_{i+6}\ c^{j-1}_{i+5}\ c^{j-1}_{i+4}$		
Substage 2 (latch)		$A^{j-1}_{i+11}\ A^{j-1}_{i+10}\ A^{j-1}_{i+9}\ A^{j-1}_{i+8}$		$A^{j}_{i+3}\ A^{j}_{i+2}\ A^{j}_{i+1}\ A^{j}_{i}$	
	…	$B^{j-1}_{i+11}\ B^{j-1}_{i+10}\ B^{j-1}_{i+9}\ B^{j-1}_{i+8}$		$B^{j}_{i+3}\ B^{j}_{i+2}\ B^{j}_{i+1}\ B^{j}_{i}$	…
		$(c')^{j-2}_{i+12}\ s^{j-1}_{i+11}\ s^{j-1}_{i+10}\ s^{j-1}_{i+9}\ s^{j-1}_{i+8}$		$(c')^{j}_{i+4}\ s^{j}_{i+3}\ s^{j}_{i+2}\ s^{j}_{i+1}\ s^{j}_{i}$	
		$(c'')^{j-2}_{i+12}\ c^{j-1}_{i+11}\ c^{j-1}_{i+10}\ c^{j-1}_{i+9}\ c^{j-1}_{i+8}$		$(c'')^{j}_{i+4}\ c^{j}_{i+3}\ c^{j}_{i+2}\ c^{j}_{i+1}\ c^{j}_{i}$	

Figure 3.6: Accumulator with LSA Variation

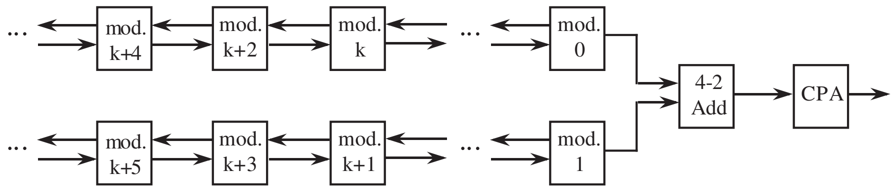


Figure 3.7: The Two Multiplier Subpipelines

substage. By repositioning the registers within the multiplication algorithm, these substages can be decoupled to cut the cycle time in half. Data that is shifted by 4 bits is instead shifted by 8 bits (transferred to step $j+2$) and input to a module on the same substage. Two separate LSA subpipelines result (Fig. 3.7) where one handles the $w[j]$ digits of weight 16^i, i even, and the other handles the digits where i is odd. The two subpipelines are merged with a 4-2 addition on the least significant output of each subpipe.

Mathematically, register repositioning has transformed the recurrence of

$$w[j+1] = \frac{1}{16}(w[j] + 16^e x z_j) \tag{3.1}$$

to two independent equations such that:

$$\begin{aligned}
w[j+1] &= w'[j+1] + \frac{1}{16}p'[j] && (3.2)\\
&\text{where} \\
w'[k] &= \frac{1}{16}\left(\frac{1}{16}w'[k-2] + 16^e x z_{k-1}\right),\ k \text{ odd, and} && (3.3)\\
w'[i] &= \frac{1}{16}\left(\frac{1}{16}w'[i-2] + 16^e x z_{i-1}\right),\ i \text{ even} && (3.4)\\
& \quad \text{where } w'[h] = 0, h \leq 0
\end{aligned}$$

Showing the equivalence of equations 3.1 and 3.2 is straightforward.

The presented multiplier design (Fig. 3.7) satisfies equation 3.2 at the merging of the subpipelines. The individual subpipelines do not distinguish between odd and even steps; instead, they differentiate between odd and even LSA modules (representing odd and even digits of the accumulated multiples). Equations 3.3 and 3.4 show that this segregation is allowable. Placing module $i-2$ adjacent to module i results in a smaller interconnection delay.

3.2.2 Multiple Chip Implementations

For high precision, the modular LSA multiplier easily spans multiple chips without logic block overhead or additional delays. Observe in Figure 3.4 that only the recoded multiplier bits are pipelined from an LSA module to its higher weighted neighbor. Since all modules eventually receive all multiplier bits, any module on a given chip can pipeline the multiplier data to another chip. No mimicking of LSA modules from the upstream chip is needed, and hence the LSA chip interface module consists of solely LSA module k (Figure 3.8). As discussed in Chapter 2, a pipelined input buffer and variable delay mechanism my be required depending on the chip interconnection scheme.

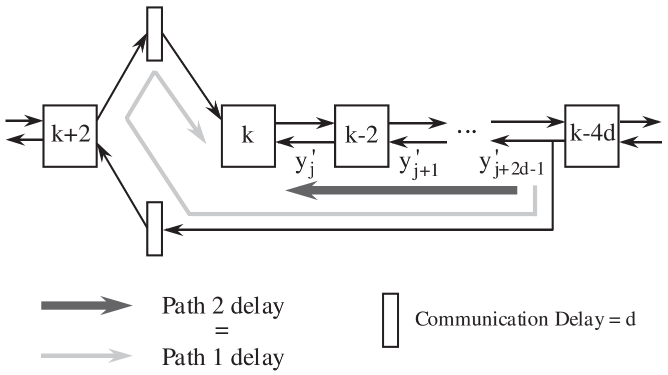


Figure 3.8: Multiple Chip Interfacing

3.2.3 The Carry Propagate Addition

In the algorithm of Figure 3.3, a final carry propagate addition (CPA) transforms the partial sum and carry vectors to its 2's complement result. A pipelined CPA operates in parallel with the accumulator to generate the least significant half of the product at the rate of one radix-16 digit per cycle. In a conventional implementation, a second n-bit CPA assimilates the most significant half of the product after the iterative cycles are complete. To save area and provide performance that is independent of interchip communication delays, the proposed multiplier does not implement the second CPA. Instead, all bits of the result are pipelined through the LSA and output from a single 4-bit CPA. With this approach, the most significant half of the product is generated at the rate of 4 bits per cycle (12.7 ns.). If the second CPA were implemented with the fast CPA hardware on the XC4010, the result would be produced at the rate of 2 bits every 1.5 ns. The fast CPA hardware, however, experiences performance degradation due to interchip communication. Area with the fast CPA also increases by 8 CLBs for every 4 bits of operand precision to assimilate c' and c'' into the sum and carry vectors and to implement the fast CPA.

The pipelined output can be useful for further pipelines, such as an inner

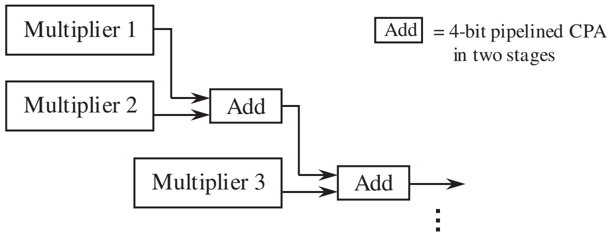


Figure 3.9: Pipelined Computation of an Inner Product

product computation of the form $AB + CD + \ldots$ (A, B, C, D, n-bit 2's complement integers) where the addition is pipelined with the multipliers (Fig. 3.9). Figure 3.9 linearly arranges the multipliers to minimize the routing distance between pipelined stages. Although a tree structure reduces the number of adders in the critical path from $a-1$ to $\lceil log_2 a \rceil$ where a is the number of products to be summed, the maximum routing distance between pipelined adders grows as a function of a. A linear structure thus has the advantage of uniform routing time. The computation time of the linear arrangement is only $2(a-1) + \left\lceil \frac{\log_2 a}{4} \right\rceil$ total cycles over the computation time of one n-bit multiplication. The extra $\left\lceil \frac{\log_2 a}{4} \right\rceil$ cycles generate the most significant bits of the sum which is larger than the $2n$-bit multiplier output by $\lceil \log_2 a \rceil$ bits.

3.3 Implementation Results

With the LSA multiplier modules, multiplying two n-bit 2's complement operands has a computation time of $\left(4 + \left\lceil \frac{n}{2} \right\rceil\right) t_{cycle}$. Four cycles are needed to start the pipeline; this includes cycles for multiple generation in the least significant LSA module, a 4-2 addition by the LSA module, a 4-2 addition to merge the subpipelines, and an intermediate stage addition in the pipelined CPA. On the fifth cycle, the least significant four bits of the 2's complement result appear from the second and final stage of the CPA pipeline. Four more bits of 2's complement result appear at each cycle thereafter until all $2n$ bits are derived.

With regard to area, a theoretical maximum of 22 LSA modules (88 bits of precision) fit on a single XC4010. Each LSA module requires 18 CLBs and extends the precision of the multiplier by 4 bits. The logic to merge the subpipelines consists of 8 CLBs (a 4-bit 4-2 adder). The two 4-bit pipelined stages for the CPA utilize 6 CLBs. Area can be decreased by reducing the radix, but this will lower the performance. For example, a radix-4 design requires only half the number of

Multiplier Design	Operand Precision (number of bits)	Computation Delay
LSA Multiplier Modules	2000	13 μs
DEC PAM P_1 [BeRV93]	2000	30 μs
LSA Multiplier Modules	334	2 μs
DEC PAM P_1 [BeRV93]	334	5 μs
Cray software (vectorized) [BuWa89]	334	55 μs
Cray software (no vectorization) [BuWa89]	334	232 μs
LSA Multiplier Modules	8	102 ns
LSA Multiplier Modules	16	152 ns
LSA Multiplier Modules	24	203 ns
LSA Multiplier Modules	32	254 ns
TMS390S10 microSPARC [TI92]	32	340 ns

Table 3.1: Performance Comparisons of Various Multipliers

recoded multiples and employs a 3-2 carry save adder which uses half the area of a 4-2 adder.

To verify routability, three neighboring LSA multiplier modules have been placed and routed on the Xilinx XC4010 speed grade -5. The implementation was created with the Xilinx XACT [XACT] design tool and determined to have a cycle time of 12.7 ns. (79 MHz). With the locality of interconnections between LSA modules and its ability to span multiple chips without introducing additional delays, the performance for very high precisions can be extrapolated. Table 3.1 compares the extrapolated LSA performance with other designs. In Table 3.1, the computation time for the PAM P_1 [BeRV93] (Programmable Active Memories part P_1) is extrapolated from their reported time of 66 Mbits/s. The Texas Instruments microSPARC [TI92] timing describes their 32-bit integer multiply unit executing at 50 MHz and does not include the time for register loads and stores. For the Cray 2 with software [BuWa89], Table 3.1 presents the average time of a single multiplication when 10^6 multiplications of two 100-decimal digit operands are performed. To create a fair comparison, normalization time is excluded. Cray performance both with and without vectorization are given.

As shown in Table 3.1, the LSA can also be used for small precisions. Despite the register delay for sequential operation, an 8-bit radix-16 LSA multiplier has a comparable delay to a radix-2 combinational logic array multiplier [ChME93] which is also implemented with the XC4000-series FPGAs. The combinational logic multiplier uses broadcasting and a carry save adder approach to provide a delay approaching 100 ns. [ChME93].

A linear sequential array provides a simple means for constructing multiplier hardware of any desired precision. While it cannot match the speed of custom ASICs, it can provide good performance application-specific hardware where ASICs are too costly or time consuming to be developed. As shown in Table 3.1, high precision integer multiplication can be computed with the LSA multiplier modules at more than 25 times the speed of a multiple precision software package executing on a Cray supercomputer.

For n-bit operands, an LSA multiplier is created by appending $\left\lceil \frac{\lceil (n/4) \rceil + 1}{2} \right\rceil$ modules to the even radix-16 digit subpipeline and $\left\lceil \frac{\lceil (n/4) \rceil}{2} \right\rceil$ modules to the odd digit subpipeline (Fig. 3.7) where the least significant radix-16 multiplicand digit is even and of weight 16^0. The most significant weighted LSA module is responsible for sign extension for the partial sum and carry. The two resulting subpipelines are then merged with a 4-2 adder and pipelined with a 4-bit CPA. For high precision designs spanning multiple chips, a module is selected to pipeline the multiplier data to the next chip; this module is chosen based on the interchip communication delay. The resulting implementation features a multiplication time of $\left(4 + \left\lceil \frac{n}{2} \right\rceil\right)$ 12.7 ns. on the XC4010 speed grade -5 and utilize $30 + 18\left(\left\lceil \frac{n}{4} \right\rceil\right)$ CLBs.

CHAPTER 4

Mapping the Selection Function

With elimination of the broadcasting delay problem, further performance optimizations are dependent on combinational logic reductions in the critical path. This chapter presents a method for mapping digit-recurrence arithmetic algorithms to lookup table based FPGAs. The primary goal is the optimization of performance, but area is reduced whenever it avoids sacrificing delay on the critical path. The presented mapping philosophy is based on *input reduction* where the number of binary inputs to combinational logic is decreased to permit designs of smaller depth [LoEr94c].

A simple approach to result digit prediction for radix-2 digit-recurrence algorithms is also proposed. With result digit prediction, an algorithm's data path may be removed from the critical path. Previous prediction schemes [ErLa85, ErLa89, MoCi94] have required the customization of the selection function specifically for prediction. With the presented approach, prediction is developed during the mapping process as a variation of an existing selection function.

4.1 Input Reduction

Digit-recurrence selections are an iterative process in which a *residual* variable, w, [ErLa94] is updated at each cycle. The recurrence is usually of the form:

$$w[j+1] = f(w[j], z_{j+1}, P^i) \tag{4.1}$$

where $w[j]$ denotes the residual at algorithm step j, z_{j+1} denotes a value that controls the updating of $w[j]$, and P^i denote input constants. The recurrence typically consists of a series of scalings (arithmetic shifts or generation of multiples) and additions/subtractions that maintain a bounded residual value.

For almost all digit-recurrence algorithms, the computation of z_{j+1} is a feedback process in which z_{j+1} is a result digit of the arithmetic algorithm. The feedback value of z_{j+1} is generally computed with the function:

$$z_{j+1} = g(\hat{w}[j], K^i) \tag{4.2}$$

in which a limited precision estimate of the residual, $\hat{w}[j]$, is compared with a series of selection constants, K^i. The multiplication operation is the exception to this feedback method, in which z_{j+1} is a multiplier digit rather than a result digit.

Specification of the addition/subtractions of Equation 4.1 involves a redundant addition, such as signed-digit [Aviz61] or carry-save, to avoid the propagation of carries. In this study, only carry-save addition is considered. Multiples to be added/subtracted are typically implemented as sums of arithmetically-shifted values (scaling by powers of 2). For the selection of z_{j+1}, carry assimilation on $\hat{w}[j]$ is first performed to simplify the subsequent comparison with selection constants. In summary, digit-recurrence selection functions can be characterized by the following sequence of steps: (1) carry-save addition (CSA), (2) short carry-propagate addition (CPA), and (3) comparison. Each step reduces the given input data until the proper selection value becomes known — the CSA reduces three bit vectors to two; the CPA reduces these two vectors to one; and the comparison narrows the selection choices after examining each bit of the resulting vector beginning with the most significant bit. Since these steps are a series of data reductions, a scheme that performs preliminary reduction of data in parallel with the critical path will generally improve performance.

Input reduction attempts to decrease the logic delay of digit-recurrence selections through preliminary reductions of the data at the algorithm level. Already within the algorithm development process of any selection function is the goal of finding low precision selection constants combined with low precision residual estimates (truncated vectors that are summed in the CSA and CPA for the selection) [CS252a, ErLa94]. A lower precision residual estimate reduces the number of input bits to the carry-propagate adder thus decreasing its delay. Lower precision selection constants similarly reduce the number of input bits to the selection logic and also decrease the delay.

Reducing the number of input bits within the implementation can further decrease delay over that provided by the limited precision selection algorithm. This reduction is based on precomputation and substitution of functions to propagate dependencies across bit positions. For example, Figure 4.1a shows an addition with a carry-in when implemented with only 4-input lookup tables. In Figure 4.1a:

$$\begin{aligned} (i) \qquad & 2e_3 + e_4 = a_4 + b_4 + c_4 \\ (ii) \qquad & 4d_1 + 2d_2 + d_3 = 2a_2 + a_3 + 2b_2 + b_3 \\ (iii) \qquad & 4f_1 + 2f_2 + f_3 = 2d_2 + d_3 + e_3 \end{aligned}$$

operands			.	a_1	a_2	a_3	a_4
			.	b_1	b_2	b_3	b_4
			.				c_4
intermediate step 1			.	a_1	d_2	d_3	e_4
			.	b_1		e_3	
			.	d_1			
intermediate step 2			.	a_1	f_2	f_3	g_4
			.	b_1			
			.	d_1			
			.	f_1			
sum	g_{-1}	g_0	.	g_1	g_2	g_3	g_4

(a) Without reduction in number of input bits

operands			.	a_1	d_2	d_3	a_4
			.	b_1			b_4
			.	d_1			c_4
intermediate step 1		k_0	.	k_1	d_2	h_3	h_4
			.		h_2		
sum	g_{-1}	g_0	.	g_1	g_2	g_3	g_4

(b) With reduction in number of input bits

Figure 4.1: Addition Example

$$(iv) \qquad 4g_{-1} + 2g_0 + g_1 = a_1 + b_1 + d_1 + f_1$$

$$(v) \qquad g_4 = e_4$$

$$(vi) \qquad g_i = f_i; \quad i = 2, 3$$

A depth of three logic levels are needed for the design. Figure 4.1b then shows the same addition when the middle four bits (a_2, a_3, b_2, b_3) have been replaced with their 3-bit carry-propagate added equivalent. The reduced input situation of Figure 4.1b may occur, for example, if computing $\sum_{i=2}^{3}(a_i + b_i)2^i$ was possible during a previous step. In Figure 4.1b:

$$\begin{array}{rl} (i) & 4h_2 + 2h_3 + h_4 = a_4 + b_4 + c_4 \\ (ii) & 2k_0 + k_1 = a_1 + b_1 + d_1 \\ (iii) & 8g_{-1} + 4g_0 + 2g_1 + g_2 = 4k_0 + 2k_1 + d_2 + h_2 \\ (iv) & g_i = h_i; \quad i = 3, 4 \end{array}$$

The result is a two logic level delay, one level fewer than in Figure 4.1a. By performing precomputations in parallel with other steps during the selection process, implementations of reduced delay can be achieved. Mapping strategies to promote input reductions are presented in the next section.

4.2 The Mapping Strategies

The goal of the proposed mapping method is the reduction of delay, but area reductions often occur when creating new functions for improved performance. Additional area optimizations are performed only if they do not increase the delay of the critical path.

Each mapping strategy develops an efficient variation of the logic in the critical path. One strategy merges algorithm steps into a common CLB to create a new consolidated step. The remaining strategies are based on *input reduction* which reduces the number of combinational logic input bits on the critical path. Specifically, input reduction (i) decreases the number of subfunctions involved in computing each register bit and (ii) reduces the number of input bits to each subfunction. Input reduction thus creates new simplified functions that can be implemented with a lower logic depth and often fewer logic blocks.

4.2.1 Merging

Given an algorithm composed of serial steps, a straightforward approach toward minimizing delay is the merging of successive steps into a new consolidated function. Figure 4.2 provides a simple example where two steps, A and B, are merged as one consolidated step, AB. This example is similar to bin packing [FrRV91] where the resulting merged implementation has both a reduced delay and area.

Figure 4.3 shows another merging example which is similar to that used in the divider designs. In this case, three operands, a, b, and c, (Fig. 4.3) are added to form the result, d. Three operand addition is typically implemented by a carry save addition followed by a carry propagate addition. Figure 4.3a directly implements

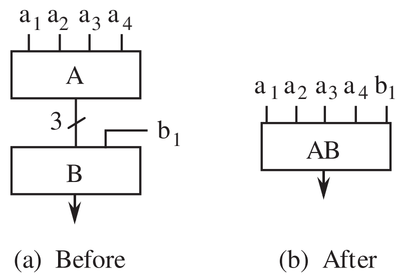


Figure 4.2: Example of Merging

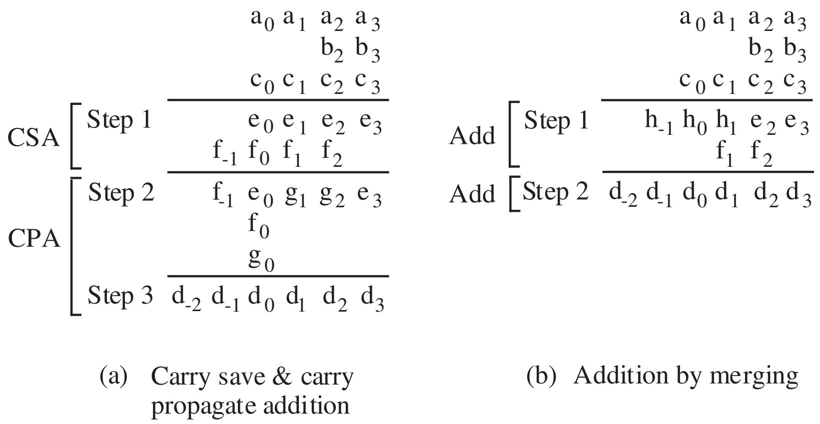


Figure 4.3: Three Operand Addition Example

these steps without using the fast carry-propagate adder of the XC4010 (FCPA). In Fig. 4.3a:

(i)	$2f_{i-1} + e_i = a_i + c_i; \quad i = 0, 1$	(1 CLB, 2-input functions)
(ii)	$2f_{i-1} + e_i = a_i + b_i + c_i; \quad i = 2, 3$	(1 CLB, 3-input functions)
(iii)	$(4g_0 + 2g_1 + g_2)2^{-2} = \sum_{i=1}^{2}(e_i + f_i)2^{-i}$	($1\frac{1}{2}$ CLBs, 4-input functions)
(iv)	$d_3 = e_3$	(0 CLBs)
(v)	$d_2 = g_2$	(0 CLBs)
(vi)	$d_1 = g_1$	(0 CLBs)

(vii) $4d_{-2} + 2d_{-1} + d_0 = 2f_{-1} + f_0 + e_0 + g_0$ ($1\frac{1}{2}$ CLBs, 4-input functions)

Use of the FCPA in the example of Fig. 4.3a would result in a comparable delay.

Figure 4.3b implements the same addition but merges the carry save and carry propagate addition steps. In the resulting design, neither of the two steps are explicitly carry save (CSA) or carry propagate adders. Rather, the new addition steps are defined as:

(i) $2f_{i-1} + e_i = a_i + b_i + c_i; \quad i = 2, 3$ (1 CLB, 3-input fns.)

(ii) $(4h_{-1} + 2h_0 + h_1)2^{-1} = \sum_{i=0}^{i=1}(a_i + c_i)2^{-i}$ ($1\frac{1}{2}$ CLBs, 4-inp. fns.)

(iii) $d_3 = e_3$ (0 CLBs)

(iv) $2d_1 + d_2 = (2h_1 + 2f_1 + e_2 + f_2) \bmod 4$ (1 CLB, 4-input fns.)

(v) $d_0 = (\lfloor h_0 + (h_1 + f_1)2^{-1} + (e_2 + f_2)2^{-2} \rfloor)\mathrm{mod}2$ ($\frac{1}{2}$ CLB, 5-input fn.)
(d_0 needs table H and either table F or G of Fig. 1.2)

(vi) $d_{-1} = (\lfloor (\sum_{i=-1}^{1} h_i 2^{-i-1}) + f_1 2^{-2} + (e_2 + f_2)2^{-3} \rfloor)\mathrm{mod}\ 2$ (1 CLB, 6-input fn.)

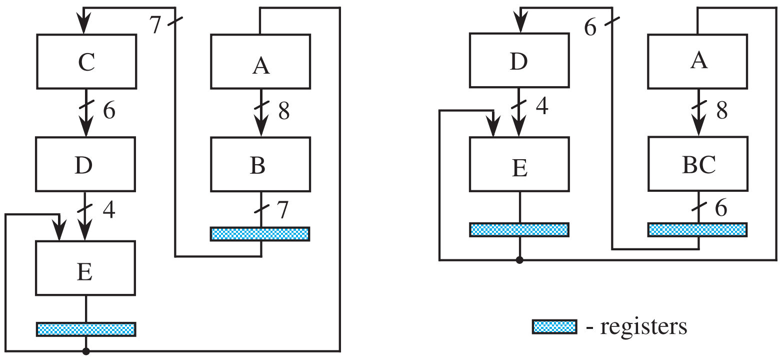
(vii) $d_{-2} = \lfloor \left(\sum_{i=-1}^{1} h_i 2^{-i-2}\right) + f_1 2^{-3} + (e_2 + f_2)2^{-4} \rfloor$ (1 CLB, 6-input fn.)

Merging avoids underutilization of a CLB's functionality. For d_{-2} and d_{-1}, merging lowers the logic delay to two CLB levels at the cost of one logic block in area. Each new 6-input function requires an entire XC4010 CLB whereas each 4-input (or fewer input) function occupies only half a CLB. The implementation of Fig. 4.3a thus requires 5 CLBs with a delay of 3 4-input logic levels while Fig. 4.3b requires 6 CLBs with a delay of one 4-input and one 5-input logic level.

4.2.2 Input Reduction Strategies

Although merging efficiently compacts a specific arrangement of algorithm steps, another approach is needed to create efficient variations of the steps. The presented mapping strategy creates variations by strategically placing the registers relative to the algorithm steps and by developing efficient encoding schemes for the given algorithm.

4.2.2.1 Register Placement

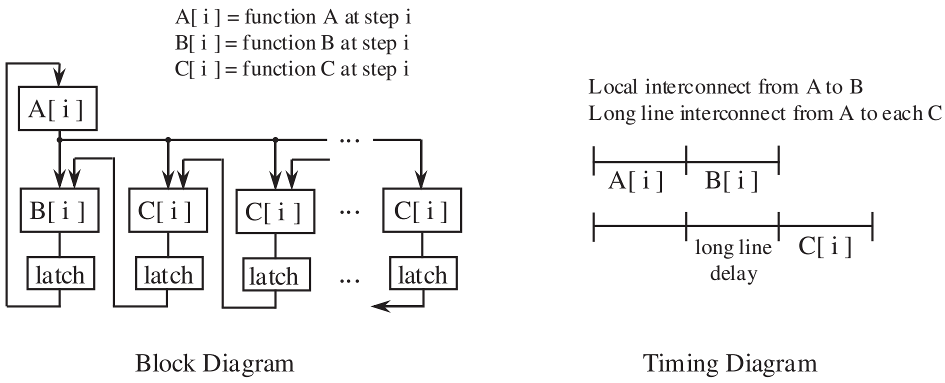


(a) Before register repositioning (b) After register repositioning

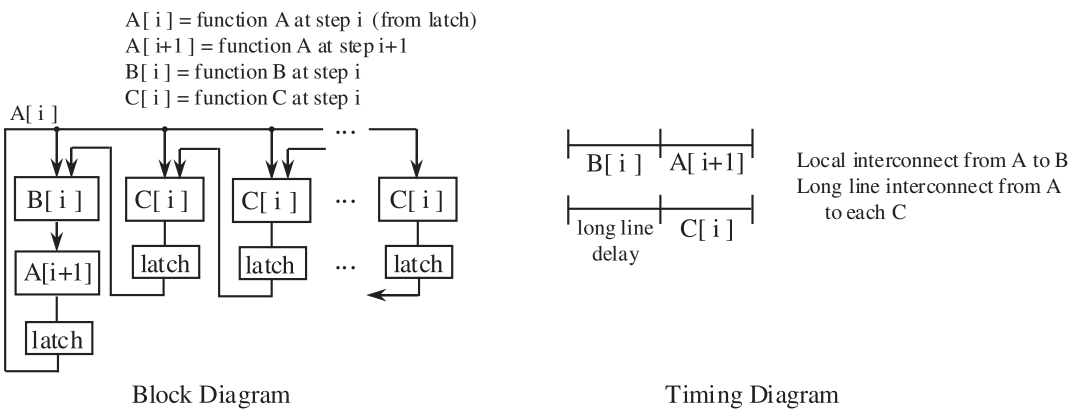
Figure 4.4: Register Repositioning with Merging

A significant delay and input reduction results from the strategic placement of registers relative to the algorithm steps. Register repositioning is similar to a retiming [LeRS83, MiYi90, MSBS90]. When a significant difference in delay exists between circuit paths, a repositioning in the form of variable precomputation balances the logic timings and reduces the delay of the critical path. In the example of Figure 4.4, register repositioning lowers the delay by providing a merging opportunity as well as a reduction in the number of function inputs. In another example (Fig. 4.5), register repositioning removes a long line delay equivalent to one logic level from the critical path.

With repositioning of the registers at the algorithm level, the computation of some bits can be eliminated. Since the precomputation from register repositioning causes a reduction in number of inputs, a decrease in number of computed outputs can result. The subsequent algorithm variation can feature reduced area and/or reduced delay. For example, Figure 4.6a shows an iterative addition where $w[j]$ (w at step j) is added to $d[j]$ to form $w[j+1]$. For this example, assume that the implementation can use only 5-input lookup tables. In Figure 4.6a each bit of $w[j+1]$ is generated from three input bits and requires one lookup table. Figure 4.6b then shows the same addition but with a precomputation of $\sum_{i=k;\,k\,\mathrm{even}}^{k+1}(s_i+c_i)2^{-i}$, before the next iteration. The subsequent reduction in number of bits enables the design of Figure 4.6c where each bit $c_{i;\,i\,\mathrm{even}}$ is unnecessary and is no longer computed. The final algorithm variation of Figure 4.6c features the same logic depth as Figure 4.6a but requires 25% fewer lookup tables.



(a) Latch before computing $A[i+1]$



(b) Latch after computing $A[i+1]$

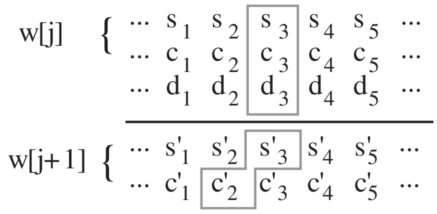
Figure 4.5: Register Repositioning Example

4.2.2.2 Efficient Encodings

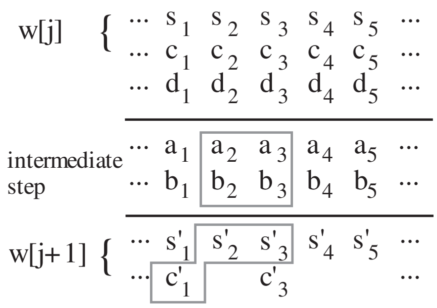
Efficient encodings also reduce the number of function inputs and decrease logic depth. For example, in SRT division [Rob58] which is discussed in detail in Chapter 5, $-q_{j+1}d$ is typically implemented as:

$$\begin{array}{cccccccc} D_{-2} & D_{-1} & D_0 & . & D_1 & \ldots & D_n \\ & & & & & & c_{in} \end{array}$$

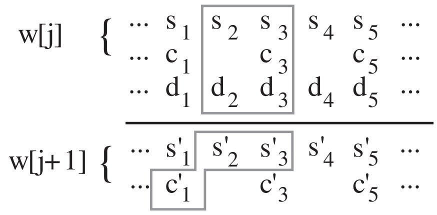
where $\sum_{i=-2}^{n} D_i 2^{-i}$ is the ones' complement value of $-q_{j+1}d$ and $((\sum_{i=-2}^{n} D_i) + c_{in})$ forms the two's complement of $-q_{j+1}d$. Due to the limited range of d, the bits of



(a) Addition by carry-save adders



(b) Addition with further computation



(c) Addition with input reduction

Figure 4.6: Algorithm Variation Example with Register Repositioning

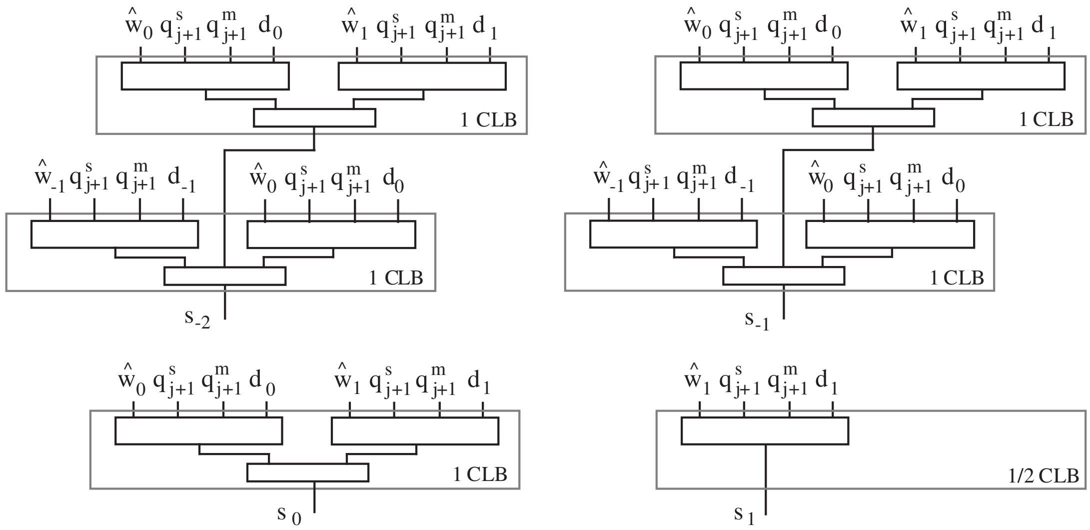
D_i, $i \leq 1$, are deducible from q_{j+1} where:

$$D_{-2}D_{-1}D_0.D_1 = \begin{cases} 111.0 & \text{if } q_{j+1} = 1 \quad (q^s_{j+1} = 0,\ q^m_{j+1} = 1) \\ 000.0 & \text{if } q_{j+1} = 0 \quad (q^s_{j+1} = 0,\ q^m_{j+1} = 0) \\ 000.1 & \text{if } q_{j+1} = -1 \quad (q^s_{j+1} = 1,\ q^m_{j+1} = 1) \end{cases}$$

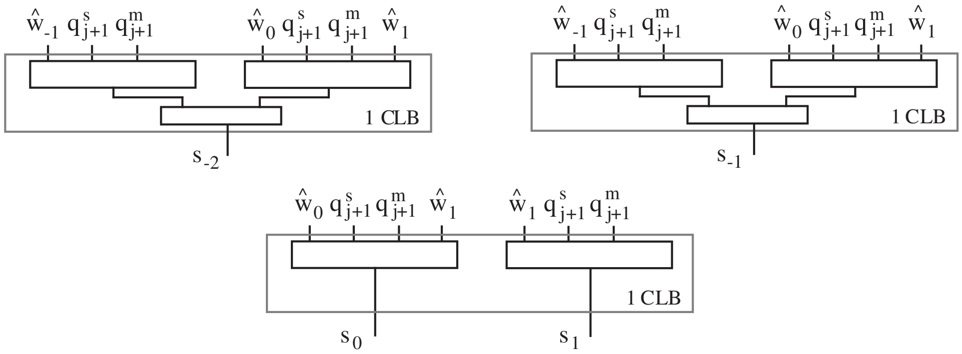
Hence, q_{j+1} serves as an encoding for D_i, $i \leq 1$, and explicit computations of D_i, $i \leq 1$, are eliminated. Figure 4.7 shows how this encoding reduces the logic delay of $\hat{w}_i + D_i$, $-1 \leq i \leq 1$, from a depth of 2 logic levels to 1 logic level. The encoding also reduces the area in the example of Figure 4.7 from $5\frac{1}{2}$ CLBs to 3 CLBs.

$$\begin{array}{r} \hat{w}_{-1}\,\hat{w}_0\,.\,\hat{w}_1 \\ D_{-1}\,D_0\,.\,D_1 \\ \hline s_{-2}\;s_{-1}\;s_0\,.\,s_1 \end{array} \qquad \text{where } D_i = f\,(q^s_{j+1},\ q^m_{j+1}, d_i)$$

(a) Addition example



(b) Implementation before encoding



(c) Implementation after encoding

Figure 4.7: Encoding Example

The most desirable encodings have fewer bits than the number of inputs to a lookup table. Such encodings not only save logic steps as in the previous example, but they also provide merging opportunities. Larger encodings of more bits than lookup table inputs will require some decoding to map to a lookup table, and this can offset the logic delay savings generated by the code.

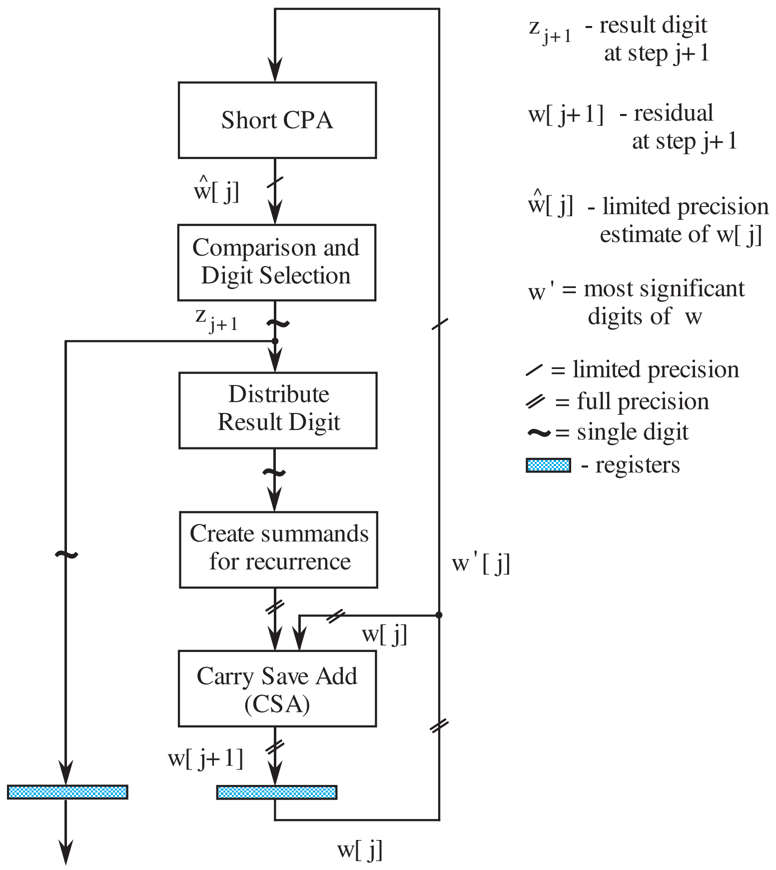


Figure 4.8: Simple Model for Conventional Selection

4.3 Result Digit Prediction

This section demonstrates that result digit prediction for radix-2 can be implemented by register repositioning [LoEr92, LoEr93b]. Figure 4.8 shows a simple model of the selection process. After selecting a result digit, added terms are created according to the algorithm's recurrence equation. Data is then reduced via a carry-save add (CSA) and short carry-propagate add (CPA) for the next selection. In Figure 4.8, however, the reduction process is separated across two different cycles — in the first cycle the added terms and residual are combined into two vectors with a carry-save adder and in the second cycle the data is further combined into one vector with a carry-propagate adder. The separation of the CSA and CPA across cycles prevents merging; as a result, these two steps (CSA and CPA) must be implemented separately. By repositioning the registers as in Figure 4.9, the reduction process is recombined into one cycle. Merging opportunities, such as in Figures 4.3 and 4.7, thus arise for better performance designs.

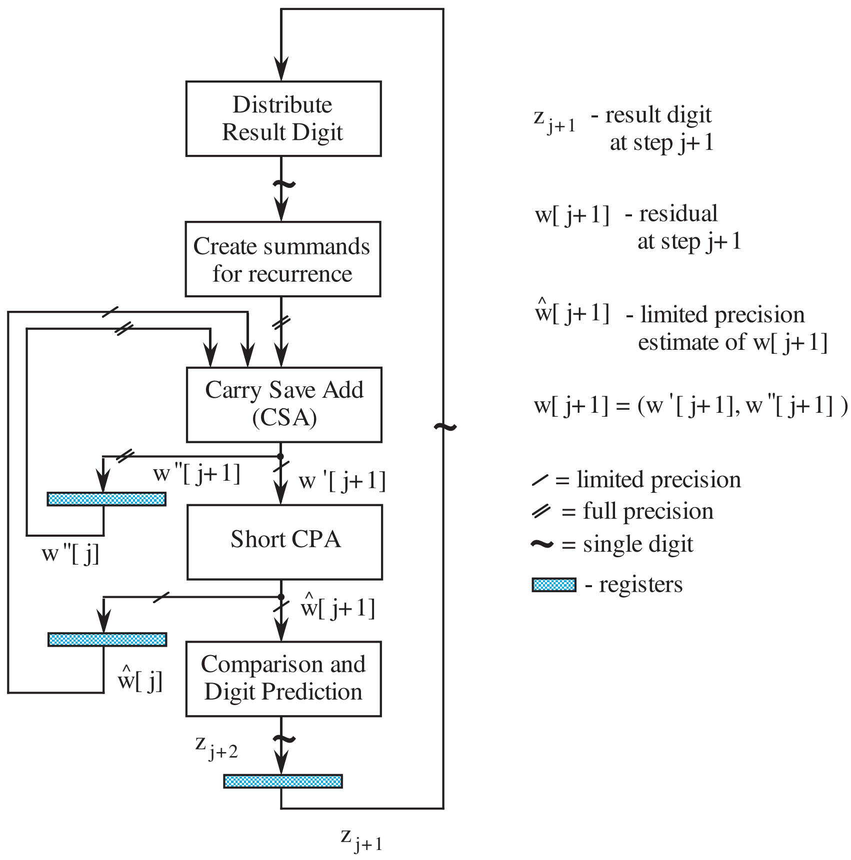


Figure 4.9: Simple Model with Result Digit Prediction

In Figure 4.9, the result digit for step $j+1$ is precomputed in step j. This precomputation was first used in the quotient digit prediction algorithm of [ErLa85]. Most prediction algorithms (e.g., [ErLa85] and [ErLa89]) modify the selection function constants to obtain the prediction. Alternatively with register repositioning, the comparison constants for selection remain the same; the new opportunities for encoding and merging are exploited to lower the delay of the critical path.

Repositioning the registers for result digit prediction maintains the same limited precision selection as in the conventional selection function. From Figure 4.8 and also specifying the arithmetic scaling (shifts) that typically occur in the digit-recurrence algorithms, a radix-2 recurrence is generally of the form:

$$w[j+1] = 2w[j] + g(z_{j+1}, d) \tag{4.3}$$

where d is an operand in the digit-recurrence operation, z_{j+1} is the result digit for step $j+1$, and $g(z_{j+1}, d)$ denotes the function generating the summands for the recurrence. The selection function for the result digit is then defined as:

$$z_{j+1} = f(2w[j]) \tag{4.4}$$

where $f()$ performs the comparison and selection based on a limited precision estimate of $2w[j]$, denoted as $2\hat{w}[j]$. In the conventional selection of Figure 4.8 and Equation 4.4, z_{j+2} is selected based on the latched data of $w[j+1]$. With result digit prediction, however, z_{j+2} is chosen as a function of the latched $w[j]$ data. From Equations 4.3 and 4.4, the computation of z_{j+2} is as follows where the first expression represents conventional selection while the second expression represents result digit prediction:

$$\begin{aligned} z_{j+2} &= f(2w[j+1]) && (4.5) \\ &= f(4w[j] + 2g(z_{j+1}, d)) && (4.6) \end{aligned}$$

Result digit prediction thus performs the selection based on the summed components of $w[j+1]$. The equivalent summand precision to create the limited precision estimate in the conventional scheme also creates the selection function inputs for result digit prediction. Hence result digit prediction uses the same limited precision estimate and comparisons as in the conventional selection scheme. This equivalence is also demonstrated in [MoCi94]. The difference between conventional selection and result digit prediction consists of the ordering of the algorithm steps within each cycle; for result digit prediction, the ordering provides merging opportunities for the CSA and CPA where these opportunities are not apparent in the conventional scheme of Figure 4.8. As shown by examples in the previous section (Section 4.2), merging of a CSA and CPA can create new functions of fewer input bits that can result in a lower logic delay. Further reductions in logic delay can also be achieved by using the residual estimate, $\hat{w}[j]$, instead of a sum and carry representation for the most significant bits of $w[j]$, and using z_{j+1} as a simple encoding of a few of the most significant bits of $\hat{w}[j+1]$.

Like the quotient digit prediction algorithms of [ErLa85, ErLa89] etc., the register repositioned prediction and mapping scheme can implement in parallel result digit selection and updating of the residual vectors for the next selection.

This parallelism is shown in the mapping of SRT division [LoEr93b] (Chapter 5) and radix-2 square root [LoEr93a] (Chapter 6). The parallelism from result digit prediction helps to prevent the lower significant bits of the data path from entering the critical path. In the original model of Figure 4.8, result digit distribution occurs in the middle of the cycle, immediately prior to the only two steps affecting both the data path bits as well as the bits involved in the selection function. Therefore, if the implementation of these two steps and result digit distribution has a larger delay in the data path than in the selection function, the data path will be part of the critical path. For example, broadcasting has a larger delay than local routing and will cause the data path to enter the critical path. The LSA scheme of Chapter 2 eliminates broadcasting, but its potential overhead (maximum of twice the combinational logic delay of the data path) may also cause the data path to enter the critical path when using the model of Figure 4.9. By using result digit prediction, the data path logic and selection logic can proceed simultaneously. The LSA will thus avoid being part of the critical path when its extra step is less than the delay of the short CPA and comparisons. In the mapping method presented in the following section, result digit prediction is an important step.

4.4 Method for Applying the Mapping Strategies

The presented mapping method combines the three techniques of merging, encoding, and register repositioning (including result digit prediction) to create efficient implementations. The process iteratively reduces the logic delay of the critical path until the mapping cycles generate no further latency reductions. An iterative process is employed because the critical path may involve different logic functions after each pass through the mapping. Within the mapping method, each cycle consists of:

Step 1. Strategically reposition the registers to reduce the number of combinational logic inputs in the critical path. Perform repositioning for result digit prediction. Consider moving the registers to allow precomputation of data and merging of the last and first algorithm steps (e.g., Fig. 4.4).

Step 2. After repositioning, search for encodings that will reduce the number of inputs to the combinational logic. Eliminate bits that do not need to be explicitly computed as a result of the encodings.

Step 3. Exploit merging opportunities.

Step 4. After completing the mapping cycles, implement area optimizations that do not alter the critical path.

Each mapping cycle more closely balances the circuit path delays to improve a design's performance. Step 1 creates a design variation relative to the sequential logic. Through this step, large fanout delays are transferred to non-critical paths; this transfer would not be possible from combinational logic optimizations alone. Step 1 also provides opportunities for combinational logic optimizations in the critical path by decreasing the number of function inputs. Step 2 continues reduction in the number of function inputs by attempting encodings at intermediate logic levels as well as input and output stages of the critical path. Finally, Step 3 attempts to merge serial functions of decreased inputs into new functions of lower logic depth.

The proposed strategies and mapping method improve the performance of digit-recurrence implementations as explained in the following chapters. In the next chapter, the method is detailed for the development of two SRT division designs. The better of these designs is then extended in the succeeding chapter for radix-2 square root.

CHAPTER 5

SRT Division and the Xilinx XC4010

To illustrate the mapping method of the previous chapter, a radix-2 digit recurrence division algorithm (SRT) is mapped to the Xilinx XC4010 lookup table based FPGA. Two design alternatives are explored, one that explicitly uses the XC4010 fast carry propagate adder hardware and one that does not use this feature [LoEr92, LoEr93b]. For comparison, both designs assume 24-bit operands and conventional broadcasting in the data path. Conversion of these designs for a linear sequential array scheme is as shown in Figure 2.12 of Chapter 2.

5.1 Notation

The following notation is utilized throughout this chapter. Additional notation is defined as needed for specific designs. In general, the digit labels have the format a_i where i indicates that the digit is of weight 2^{-i}.

d	divisor
z_{j+1}	quotient digit of weight $2^{-(j+1)}$; $z_{j+1} \in \{-1, 0, 1\}$
z	quotient $= \sum_{i=1}^{n} z_i 2^{-i}$
z_{j+1}^{s}	sign bit in the representation of z_{j+1} in sign/magnitude form
z_{j+1}^{m}	magnitude bit in the representation of z_{j+1} in sign/magnitude form
$w[j]$	residual at step j; $w[j]$ in two's complement carry save representation
$2\hat{w}[j]$	limited precision estimate of $2w[j]$
$\hat{w}_i$	bit of weight 2^{-i} in $2\hat{w}[j]$
s	sum vector in the carry save representation of $2w[j]$
c	carry vector in the carry save representation of $2w[j]$
s_i	bit of weight 2^{-i} in the sum vector, s
c_i	bit of weight 2^{-i} in the carry vector, c

s'_i bit of weight 2^{-i} in the new sum vector s' after assimilation of some bits

D multiple of the divisor; a ones' complement representation of $-z_{j+1}d$

D_i bit of weight 2^{-i} in D

c_{in} carry-in bit which combines with D to form the two's complement representation of $-z_{j+1}d$; $-z_{j+1}d$ (two's complement) $= (\sum_{i=1}^{n} D_i 2^{-i}) + c_{in}2^{-n}$

5.2 Division Overview

This chapter uses the SRT digit-recurrence division algorithm [Rob58, ErLa94] where the scaled residual, $2w[j]$, is stored in redundant (carry save) form. One radix-2 digit of the quotient is produced in redundant form at each iteration such that the total error of the quotient at each iteration j is less than 2^{-j}. The SRT algorithm is reviewed next with a block diagram given in Figure 5.1.

Recurrence:

$$w[j+1] = 2w[j] - dz_{j+1}$$

where

$w[0]$ — dividend; d — divisor

$1/2 \leq d < 1, \quad |w[0]| < d$

Iteration cycle:

Step 1. One digit arithmetic left-shift of $w[j]$ to form $2w[j]$
(No delay — implemented by appropriate routing)

Step 2. Determine z_{j+1} by the quotient digit selection function.

Step 3. Obtain $-z_{j+1}d$

Step 4. Compute $w[j+1]$ by adding $-z_{j+1}d$ to $2w[j]$
(carry save addition)

Quotient digit selection function (where $2\hat{w}[j]$ is a 4-bit estimate of $2w[j]$):

$$z_{j+1} = \begin{cases} 1 & \text{if } 0 \leq 2\hat{w}[j] \leq 3/2 \\ 0 & \text{if } -1/2 = 2\hat{w}[j] \\ -1 & \text{if } -5/2 \leq 2\hat{w}[j] \leq -1 \end{cases}$$

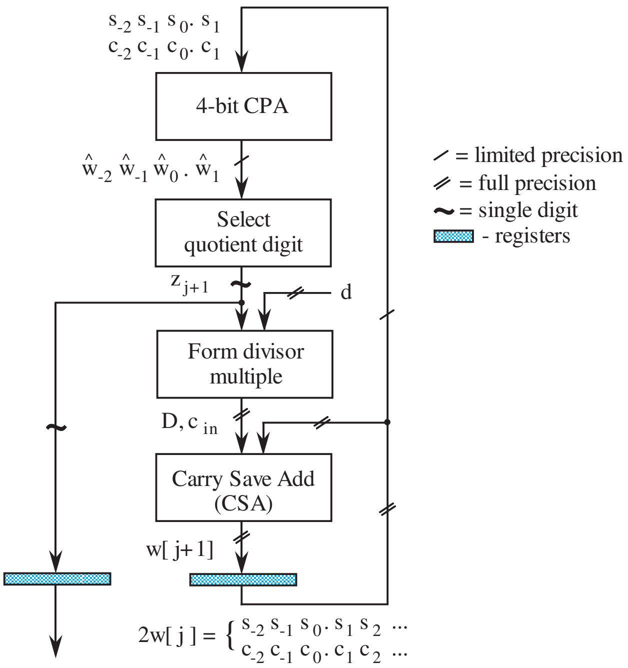
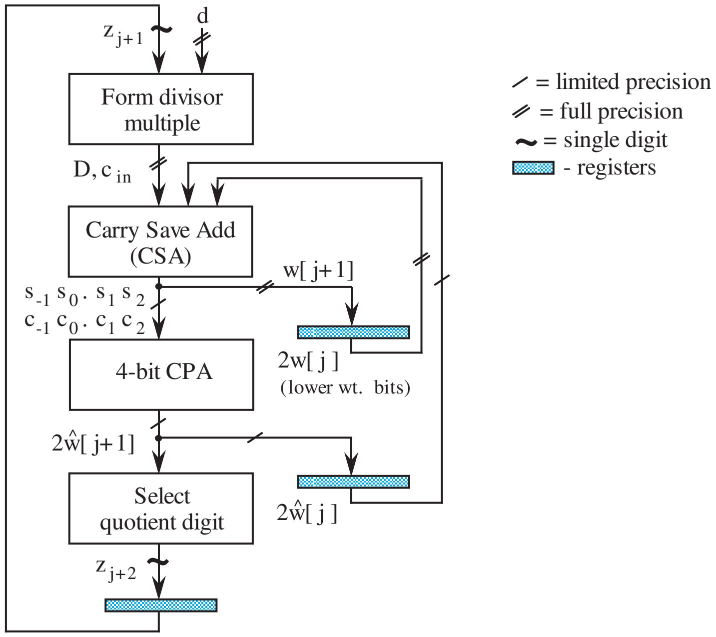


Figure 5.1: Block Diagram of the SRT Division Algorithm

5.3 Division Design with the Fast CPA

To illustrate the mapping process, the mapping method of Chapter 4 is applied to two SRT division designs. Both designs assume 24-bit operands where the long line delay is equivalent to the delay of a CLB. In the first design, the XC4010 fast carry propagate adder hardware explicitly generates $2\hat{w}[j]$ and performs any carry propagate addition steps that arise as a result of the mapping. The mapping strategies implement the remaining logic.

On applying the mapping method, the first cycle precomputes the quotient digit to remove its fanout delay from the critical path. The computation of z_{j+2} during the jth step is known as quotient digit prediction [ErLa89] [ErLa94]. With quotient digit prediction, $2\hat{w}[j+1]$ is precomputed and stored instead of s_i and c_i, $-2 \leq i \leq 1$ (Fig. 5.2); the bits D_{-2}, c_{-2}, and s_{-2} then become unnecessary and are eliminated.



(a) Block diagram

$$\left.\begin{array}{ccl} z^s_{j+1} & z^m_{j+1} & \hat{w}_{-1}\hat{w}_0 . \hat{w}_1\ s_2\ s_3\ s_4\ \ldots \\ & & \qquad .\quad c_2\ c_3\ c_4\ \ldots \end{array}\right\}\ 2w[\,j\,]$$

(b) Registers for the implementation

Figure 5.2: SRT Division with Quotient Digit Prediction

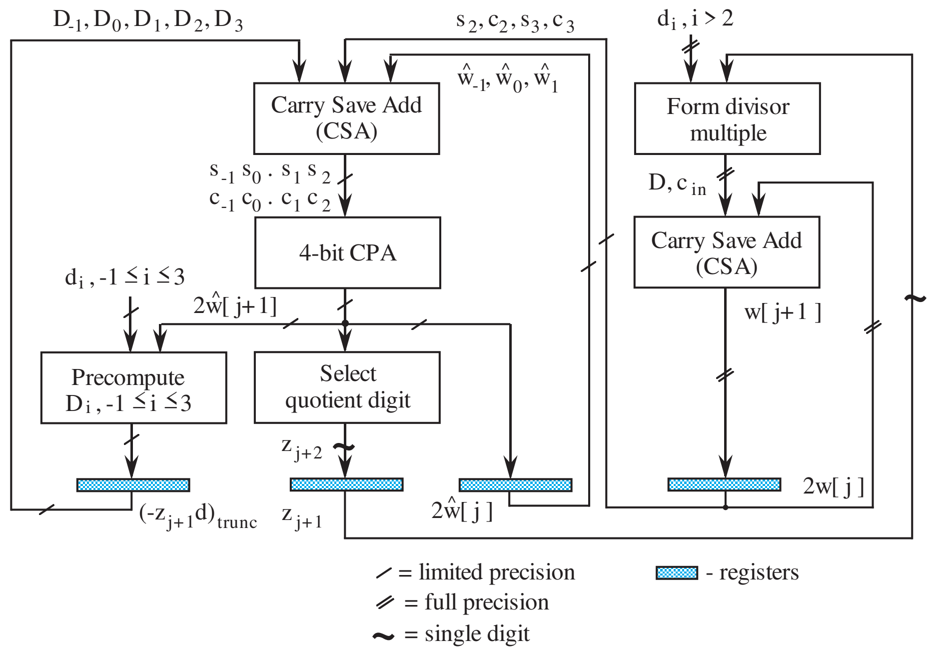
At the second iteration (Fig. 5.3), D_i, $-1 \le i \le 3$, are precomputed to merge the selection of z_{j+2} with the computation of D_i, $-1 \le i \le 3$. The explicit generation of D_{-1}, D_0, D_1, D_2, D_3, and z_{j+2} are performed in parallel. Each of the functions computing D_{-1}, D_0, D_1, D_2 and D_3 implicitly derives z_{j+2} where $D_i = f(g(\hat{w}_{-2}, \hat{w}_{-1}, \hat{w}_0, \hat{w}_1), d_i)$ and $g(\hat{w}_{-2}, \hat{w}_{-1}, \hat{w}_0, \hat{w}_1)$ represents z_{j+2}. Generating each D_i requires 1 CLB at a total cost of 2.5 CLBs for precomputing $D_i, -1 \le i \le 3$. After the precomputation, the steps in the critical path are:

(i) carry save addition to generate $w[j+1]$,

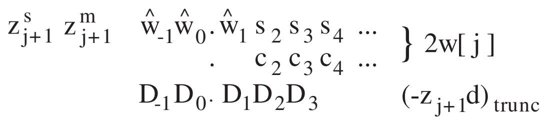
(ii) 4-bit Fast Carry-Propagate Adder (FCPA) to form $2\hat{w}[j+1]$, and

(iii) simultaneous computation of z_{j+2} and D_i, $-1 \le i \le 3$.

In the third iteration (Fig. 5.4) c_2, c_3, and s_i, $-1 \le i \le 3$, are assimilated into s'_i where $s' = \sum_{i=-1}^{3}(s_i + c_i)2^{-i}$. This precomputation creates a more compact



(a) Block diagram

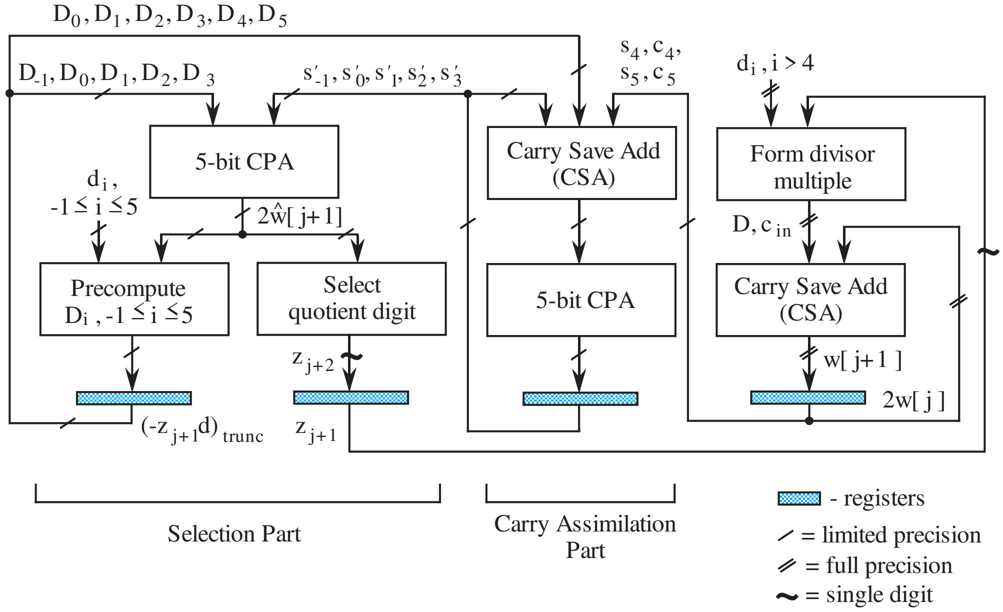


(b) Registers for the implementation

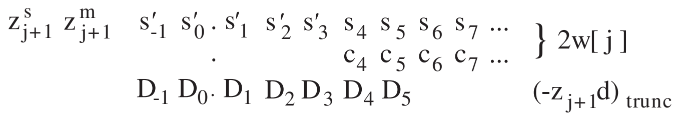
Figure 5.3: SRT Division with Quotient Digit Prediction and Merging of z_{j+2} and $(-z_{j+2}d)_{trunc}$

representation of $2w[j]$ so that z_{j+2} and D_i, $-1 \leq i \leq 3$, become functions of fewer input bits. The bits of D_4 and D_5 are also precomputed to eliminate the CSA from the critical path. Thus, the computation of z_{j+2} and D_i, $-1 \leq i \leq 5$, begins directly with a 5-bit CPA. The implementation for the most significant bits is therefore divided into two parallel parts: (i) the selection part which utilizes a 5-bit FCPA to evaluate z_{j+2} and D_i, $-1 \leq i \leq 5$, and (ii) the carry assimilation part which utilizes a 5-bit FCPA to generate s'_i, $-1 \leq i \leq 3$.

The resulting XC4010 speed grade -5 implementation for single precision operands (mantissa only) requires 60 CLBs and has a delay of 26.6 ns per quotient digit. The 5-bit FCPA uses 49% of the iteration delay where the remaining cycle time is



(a) Block diagram



(b) Registers for the implementation

Figure 5.4: SRT Division with the Fast CPA

shared by one CLB stage, latching, and interconnection time. One signed quotient digit is produced per iteration. To convert the result to conventional form, an on-the-fly converter [ErLa87] can be additionally implemented. The converter would operate simultaneously with the divider, and the fully converted result would be produced within one cycle of the completed division.

5.4 Division Design without the Fast CPA

As in the previous design, algorithm steps that are implemented with the FCPA cannot be merged with other steps in the design. This section explores the merging and input reduction opportunities that are possible when not using the FCPA.

Continuing from quotient digit prediction (Fig. 5.2), step 2 of the method employs the encoding scheme of Section 4.2.2.2 for D_{-1}, D_0, D_1. The encoding saves implementation area because D_i is no longer explicitly computed. Additional

$\hat{w}_{-2}\hat{w}_{-1}$	$\hat{w}_0\hat{w}_1$	$z^s_{j+1}z^m_{j+1}$
1 1	1 1	0 0
1 1	1 0	1 0
1 1	0 1	1 0
1 1	0 0	1 0
1 0	1 1	1 0
0 0	0 0	0 1
0 0	0 1	0 1
0 0	1 0	0 1
0 0	1 1	0 1

Table 5.1: Residual Estimate Values and z_{j+1} for SRT Division

encoding represents $\hat{w}_{-1}$ with z_{j+1}, $\hat{w}_0$, and $\hat{w}_1$ (Table 5.1) to eliminate the explicit generation of $\hat{w}_{-1}$. The subsequent input reduction enables merging of three steps of Fig. 5.2 consisting of (i) computation of D_i, $-1 \leq i \leq 3$, (ii) CSA to form s_i and c_i, $-1 \leq i \leq 2$, and (iii) 4-bit CPA, to create two new merged steps (Fig. 5.5) of reduced delay.

Figure 5.6 shows the combinational logic with encoding and merging where:

(i) $$(4a_{-1} + 2a_0 + a_1)2^{-2} = f(z^s_{j+1}, z^m_{j+1}, \hat{w}_0, \hat{w}_1) = \textstyle\sum_{i=-1}^{1}(\hat{w}_i + D_i)2^{-i}$$ ($1\frac{1}{2}$ CLBs, 4-input fns.)

(ii) $$2b_1 + a_2 = f(s_2, c_2, z^s_{j+1}, z^m_{j+1}, d_2) = s_2 + c_2 + D_2$$ ($1\frac{1}{2}$ CLBs, 5-input fns.)

(iii) $$2b_2 + a_3 = f(s_3, c_3, z^s_{j+1}, z^m_{j+1}, d_3) = s_3 + c_3 + D_3$$ ($1\frac{1}{2}$ CLBs, 5-input fns.)

(iv) $$(4e_0 + 2e_1 + e_2)2^{-2} = \textstyle\sum_{i=1}^{2}(a_i + b_i)2^{-i}$$ ($1\frac{1}{2}$ CLBs, 4-input fns.)

(v) $$\hat{w}''_0 = e_1$$ (0 CLBs)

(vi) $$\hat{w}''_1 = e_2$$ (0 CLBs)

(vii) $$(z^s_{j+2}, z^m_{j+2}) = f(a_{-1}, a_0, e_0, e_1, e_2)$$ (2 CLBs, 5-input fns.)

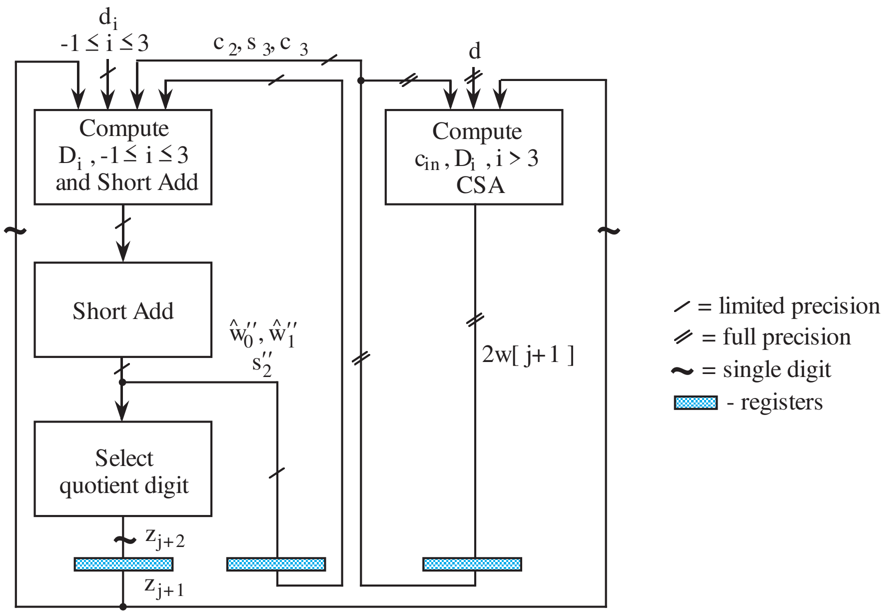


Figure 5.5: Block Diagram with Quotient Digit Prediction and Elimination of Bits

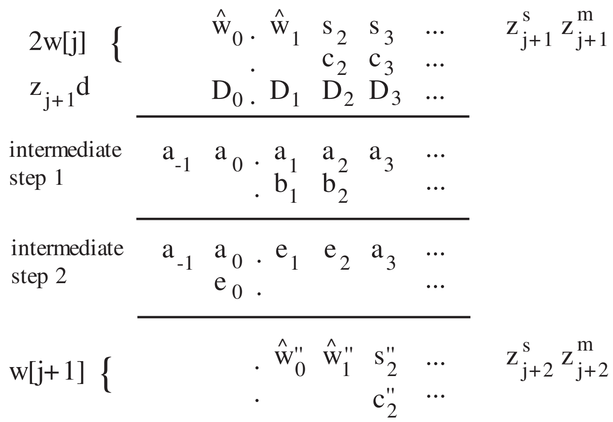


Figure 5.6: Logic with Quotient Digit Prediction and Bit Elimination

(viii) $(s''_i, c''_{i-1})(i \geq 3) = (a_{i+1}, b_i)$
$\quad = f(s_{i+1}, c_{i+1}, z^s_{j+1}, z^m_{j+1}, d_{i+1})$ (2 CLBs per s_i, c_{i-1} pair, 5-input functions)

In Figure 5.6, three logic levels are needed because the function to compute z_{j+2}, where $z_{j+2} = f(a_{-1}, a_0, a_1, a_2, b_1, b_2)$, cannot be computed in a single CLB.

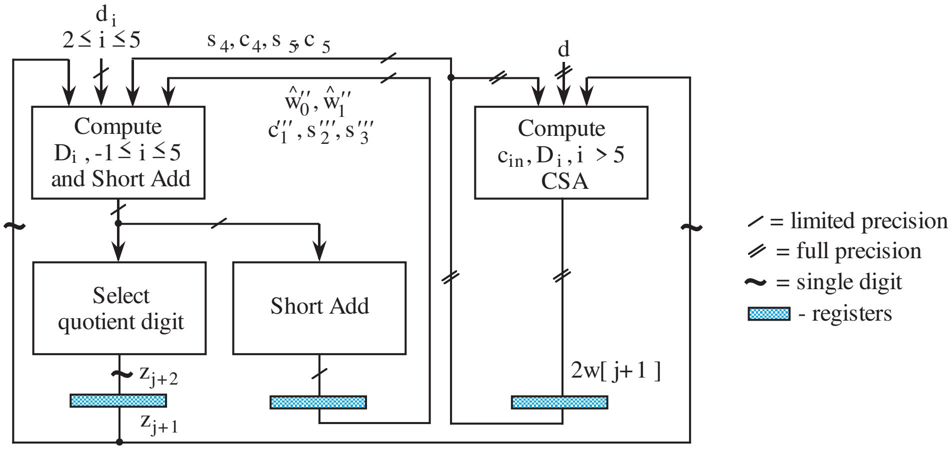


Figure 5.7: Block Diagram with the Mapping Strategies

2w[j] {	$\hat{w}_0$	.	$\hat{w}_1$	s_2	s_3	s_4	s_5	...	z^s_{j+1}
		.	c_1			c_4	c_5	...	z^m_{j+1}
$z_{j+1}d$	D_0	.	D_1	D_2	D_3	D_4	D_5	...	
Intermediate Step 1		.	m_1	k_2	k_3	p_4			$F_1 F_2$
		.	k_1		p_3	u_4			
w[j+1] {		.	$\hat{w}''_0$	$\hat{w}''_1$	s'''_2	s'''_3		...	z^s_{j+2}
		.		c''_1				...	z^m_{j+2}

Figure 5.8: SRT Division with the Mapping Strategies

During the next iteration, input reduction reduces the logic to two levels. The registers are repositioned to precompute $c_1 s'_2 s'_3 = \sum_{i=2}^{3}(s_i + c_i)2^{-i}$, and the four bits, s_2, s_3, c_2, and c_3, are replaced with their three bit sum, $c_1 s'_2 s'_3$. An encoding ($F_1 F_2$ of Fig. 5.8) also reduces the number of bits in the computation of $2\hat{w}[j+1]$. As a result, the addition and selection steps of Fig. 5.5 are merged (Fig. 5.7) so that the cycle time decreases to two CLB levels (Fig. 5.8). In Figure 5.8:

(i) $$m_1 = f(\hat{w}_1, c_1, z^s_{j+1}, z^m_{j+1}) = (\hat{w}_1 + c_1 + D_1) \bmod 2$$ ($\frac{1}{2}$ CLB, 4-input fn.)

(ii) $$(4k_1 + 2k_2 + k_3)2^{-3} = f(s_2, s_3, z^s_{j+1}, z^m_{j+1}, d_2, d_3) = \sum_{i=2}^{3}(s_i + D_i)2^{-i}$$ ($2\frac{1}{2}$ CLBs, 2 6-input fns., 1 4-input fn.)

(iii) $2p_3 + p_4 = f(s_4, c_4, z^s_{j+1}, z^m_{j+1}, d_4)$ $\quad$ ($1\frac{1}{2}$ CLBs, 5-input fns.)

$$= (s_4 + c_4 + D_4)$$

(iv) $u_4 = f(s_5, c_5, z^s_{j+1}, z^m_{j+1}, d_5)$ $\quad$ (1 CLB, 5-input fn.)

$$= \lfloor (s_5 + c_5 + D_5)2^{-1} \rfloor$$

(v) $(F_1, F_2) = f(z^s_{j+1}, z^m_{j+1}, \hat{w}_0, \hat{w}_1, c_i)$ $\quad$ (2 CLBs, 5-input fns.)
representing an encoding of $c_1 2^{-1} + \sum_{i=-1}^{1}(\hat{w}_i + D_i)2^{-i}$.
Letting $(4h_{-1} + 2h_0 + h_1)2^{-i} = c_1 2^{-1} + \sum_{i=-1}^{1}(\hat{w}_i + D_i)2^{-i}$:
$F_1 F_2 =$

- 00 if $h_{-1}h_0h_1 = 000$ or 001 where
  - $z_{j+2} = 1$
- 01 if $h_{-1}h_0h_1 = 111$ where
  - $z_{j+2} = 0$ if $k_1 = 0$, $k_2 = 1$
  - $z_{j+2} = 1$ if $k_1 = 1$
  - $z_{j+2} = -1$ otherwise
- 10 if $h_{-1}h_0h_1 = 110$ where
  - $z_{j+2} = 0$ if $k_1 = 1$, $k_2 = 1$
  - $z_{j+2} = -1$ otherwise
- 11 if $h_{-1}h_0h_1 = 100$ or 101 where
  - $z_{j+2} = -1$

(vi) $(4c_1''' + 2s_2''' + 2s_3''')$ $\quad$ ($1\frac{1}{2}$ CLBs, 4-input fns.)

$$= 2k_3 + 2p_3 + p_4 + u_4$$

(vii) $\hat{w}_0'' = (m_1 + k_1) \bmod 2$ $\quad$ ($\frac{1}{2}$ CLB, 2-input fn.)

(viii) $\hat{w}_1'' = k_2$ $\quad$ (0 CLBs)

(ix) $(z^s_{j+2}, z^m_{j+2}) = f(F_1, F_2, k_1, k_2)$ $\quad$ (1 CLB, 4-input fns.)

(x) (s_i''', c_{i-1}'''), $i \geq 3$, $\quad$ (2 CLBs per s_i, c_{i-1} pair, 5-input fns.)

$$= f(s_{i+1}, c_{i+1}, z^s_{j+1}, z^m_{j+1}, d_{i+1})$$

The implementation of Figure 5.8 for single precision operands (mantissas only) requires 64 CLBs with a delay of 19.0 ns per quotient digit produced. A comparison of this result with the previous design is presented in Section 5.6.

5.5 Initialization

Operand initialization contributes another set of inputs to the combinational logic. Since this function occurs only once during the division operation, it is undesirable for the initialization logic to increase the iteration delay. Ideally, data controlled set-reset (SR) flip-flop control lines as well as D-flip flop capabilities would simplify register initialization. However, the XC4010 does not provide this feature.

The SRT implementations presented here resolve the initialization problem by entering the data through the divisor register [LoEr93b]. The divisor register is initially assigned the dividend value, and the predicted quotient is set to -1. At the initialization cycle the combinational logic adds the contents of the divisor register to the pre-zeroed sum and carry vectors and stores the result (the dividend) in the sum register. The divisor register is then changed to the true divisor value for the remaining division iterations. With this technique, the initialization logic is separated from the division iteration logic at a cost of one additional cycle for initialization.

For the more optimized second SRT implementation (without the FCPA), the logic assumes that $\frac{1}{2} \leq$ divisor < 1. Hence, this also forces the initialized dividend to be positive. The initial values are thus:

(i) divisor register $\leftarrow \frac{1}{2} + \frac{1}{2}$dividend

(ii) $z_0 \leftarrow -1$ (for initialization only; actual z_0 is always implied as 0)

(iii) sum register $\leftarrow -1$ (two's complement)

(iv) carry register $\leftarrow \frac{1}{2}$

With respect to the SRT algorithm with carry save addition in [ErLa94], this implementation performs $\frac{1}{2}$dividend $\div$ divisor. The first computed quotient digit is of weight 1.

5.6 Implementation Results

5.6.1 Single Precision under a Broadcasting Scheme

For the SRT designs of Sections 5.3 and 5.4, Table 5.2 shows the XC4010 implementation delay per output digit and the required number of CLBs for single precision operands (24-bit mantissa only) when a broadcasting scheme is employed. The second design (without the FCPA) demonstrates a 28% improvement in performance over the first. The critical paths of both implementations consist of two algorithm steps, but the latency of the FCPA increases the first design's total cy-

Design	Speed Grade -5 Delay Per Quotient Digit	No. CLBs
SRT with Fast CPA	26.6 ns	60
SRT without Fast CPA	19.0 ns	64
SRT Baseline Estimate	40.0 ns	57

Table 5.2: Division Implementation Comparisons

cle time to three logic levels. No further optimizations seem feasible for the first design because the FCPA behaves as a single hardware unit.

By not utilizing the FCPA, there are a greater number of merging opportunities. Two logic levels generate $2\hat{w}[j+1]$ as in the first design, but merging is possible for an intermediate step of $2\hat{w}[j+1]$ and the logic for z_{j+2}. As a result, the critical path reduces to a total of two logic levels. The second design does not precompute D_i because the increase in number of combinational logic input bits prevents merging with the computation of $2\hat{w}[j+1]$.

In the second design (Section 5.4), area is traded for speed improvements in the lower weighted bits. Since the long line fanout delay of z_{j+1} is equivalent to one logic level, merging is required in the lower weighted bits to maintain a critical path of two logic levels. The merged steps consist of (i) the computation of D_i and (ii) the carry save addition to form s_i and c_i, $i \geq 4$. This merging necessitates two CLBs for each s_i c_{i-1} pair where each s_i and c_{i-1} implicitly computes the value of D_i. In the first design, merging in the lower weighted bits is unnecessary because the logic delay for these bits is less than the critical path. For the first design, the value of each D_i is explicitly generated in $\frac{1}{2}$ CLB so that the pair, s_i and c_{i-1}, are generated in 1 CLB. Thus, the total logic for D_i, s_i and c_{i-1} requires only $1\frac{1}{2}$ CLBs in the first design as compared to 2 CLBs in the second.

For the most significant bits, the mapping reduces the area of the second design more than the first. Through merging and elimination of bits, the second design utilizes only 12 CLBs for the selection logic (Fig. 5.8) whereas the first design requires 18 CLBs for the selection and carry assimilation parts of Figure 5.4.

Table 5.2 also provides a division baseline estimate for comparison with the two design studies. The division baseline utilizes the original SRT algorithm of Figure 5.1. It assumes that each algorithm step is implemented separately and that the XC4010 fast CPA hardware computes the estimate of the residual. A comparison shows that the second design achieves a doubling in performance over the baseline while the first design achieves a 33% performance improvement. With regard to area, the baseline requires fewer total logic blocks; the baseline functions utilize the 4-input functionality of the XC4010 CLBs where each CLB can implement two of these functions.

In each of the implementations and baseline estimate, the block count includes the logic to perform the iteration cycles, the logic for initialization, and the registers for the operands. The output is serially available in signed digit form. The division designs do not include an on-the-fly converter [ErLa87] to convert the output to two's complement form. A counter must also be created to identify completion when the desired number of output bits are produced.

5.6.2 Higher Precision with the LSA

Both designs of Sections 5.3 and 5.4 assume single precision mantissas where the long line fanout delay of z_{j+1} is equivalent to a logic level. Quotient digit prediction successfully transfers the long line delay to non-critical paths only because the critical and non-critical paths differ by at least one logic level. However, as higher precision dividers are needed, the fanout delay of z_{j+1} grows and eventually determines the critical path. For the second design of Section 5.4, the fanout delay of z_{j+1} enters the critical path with precisions larger than 24 bits. A linear sequential array (LSA) [Erce84] avoids the large fanout delay by providing pipelining rather than the large fanout broadcasting of z_{j+1}.

As shown in Chapter 2, Figure 2.12 provides an LSA module for the data path of SRT division. The modules may be attached to the division design at any point where the sum and carry bits of the residual are computed and latched. Hence, the LSA implementation serves as an extension and/or replacement of the lower weighted residual bits of the existing design.

An implementation of the linear sequential array requires two logic levels, each of a 4-input lookup table delay. Fanout of data between stages of the pipeline is small and locally distributed to achieve low interconnection times. As a result, the LSA's logic delay is less than the critical path for either design of Section 5.3 or 5.4. Due to its pipelined nature, LSA stages and hence precision may be added freely without increasing the critical path of the divider. The LSA is also applicable to other technologies and has an iteration delay of approximately two full adders.

Table 5.3 presents extrapolated performance results for the LSA divider and compares them with both a high precision software implementation on the Cray

Divider Design	Operand Precision (number of bits)	Computation Delay
LSA Divider Modules	334	6.4 μs
Cray software (vectorized) [BuWa89]	334	116 μs
Cray software (no vectorization) [BuWa89]	334	139 μs
LSA Divider Modules	24	475 ns
TMS390S10 microSPARC [TI92]	24	400 ns
LSA Divider Modules	56	1.1 μs
TMS390S10 microSPARC [TI92]	56	700 ns

Table 5.3: Performance Comparisons of Various Dividers

[BuWa89] and also a general-purpose microprocessor (TMS390S10 microSPARC [TI92]). Table 5.3 demonstrates that the LSA design is comparable to the microprocessor in performance but can easily outperform a software implementation.

Regarding area, each 4-bit LSA module requires 9.5 CLBs for implementation. This includes overhead for the operand registers as well as initialization logic. For the second SRT design (Section 5.4) without the linear sequential array, the lower weighted residual bits require 10 CLBs for every 4 bits of precision, including initialization overhead and operand registers. Similarly, in the first SRT design (Section 5.3) where area is optimized (traded for logic delay) in the lower weighted bits, 8 CLBs are needed for every 4 bits of precision. Hence, the linear sequential array provides additional precision at a cost of 25% more logic blocks than an area optimized design. For the higher performance design in which block count must be sacrificed for speed, the LSA actually improves the area requirements.

CHAPTER 6

Square Root and the Xilinx XC4010

As an extension to the SRT divider, this chapter develops a radix-2 square root design. The design consists of three module types – a selection logic module for the selection function (Section 6.3), an LSA module to extend the precision in 4-bit increments and an interface module to span multiple chips (Section 6.4) [LoEr93a, LoEr94d]. Section 6.5 presents the implementation results.

6.1 Notation

The following notation is utilized throughout the chapter. Additional notation are defined as needed. In general, the digit labels have the format d_i where i indicates that the digit is of weight 2^{-i}.

z_j	square root digit of weight 2^{-j}; $z_j \in \{-1, 0, 1\}$
$S[j]$	computed square root value as of step j where $S[j] = \sum_{i=1}^{j} z_i 2^{-i}$
z_j^s	sign bit in the representation of z_j in sign and magnitude form
z_j^m	magnitude bit in the representation of z_j in sign and magnitude form
$w[j]$	residual at step j; $w[j]$ in two's complement carry save representation
s	sum vector in the carry save representation of $2w[j]$
c	carry vector in the carry save representation of $2w[j]$
s_i	bit of weight 2^{-i} in the sum vector, s
c_i	bit of weight 2^{-i} in the carry vector, c
$T[j] =$	$-S[j-1]z_j - z_j^2 2^{-(j+1)}$
$T_i[j]$	bit of weight 2^{-i} in $T[j]$

z_j	Selection Range
1	$0 \leq 2\hat{w}[j-1] \leq \frac{3}{2}$
0	$-\frac{1}{2} = 2\hat{w}[j-1]$
-1	$-\frac{5}{2} \leq 2\hat{w}[j-1] \leq -1$

Table 6.1: Square Root Digit Selection Function

6.2 Square Root Overview

An overview of the square root algorithm [ErLa91] is described in Figure 6.1. The square root digit selection function is based on a carry save adder implementation where $w[j]$ is represented by the sum and carry vectors s and c. The digit selection function (Table 6.1) [ErLa94] assigns a value based on a limited precision comparison of an estimate of $2w[j-1]$, defined as $2\hat{w}[j-1]$, with a series of selection constants. With assimilation of the carries, 3 integer bits and 1 fraction bit represent $2\hat{w}[j-1]$.

Recurrence:
$w[j] = 2w[j-1] - S[j-1]z_j - z_j^2 2^{-(j+1)}$
where
$w[j]$ = residual at step j, $w[0] = 1/2$ operand
$1/4 \leq$ operand < 1
z_j = signed square root digit of weight 2^{-j}, $z_j \in \{-1, 0, 1\}$
$S[j-1]$ = square root at step $j-1 = \sum_{i=1}^{j-1} z_i 2^{-i}$

Iteration Cycle:
Step 1. One digit arithmetic left-shift of $w[j-1]$ to form $2w[j-1]$
Step 2. Determine z_j by the square root digit selection function
Step 3. Obtain $T[j] = -S[j-1]z_j - z_j^2 2^{-(j+1)}$
Step 4. Add $T[j]$ to $2w[j-1]$ to compute $w[j]$

Figure 6.1. Square Root Recurrence and Iterations

To obtain a two's complement representation of $T[j]$, the representation of $S[j-1]$ is first converted from signed-digit to two's complement form. The original description of the algorithm proposes an on-the-fly converter in which two cases

z_j	$A[j]$	$B[j]$
1	$(A, 0, 1, 1_{j+2}, 0, \ldots, 0)$	$(\bar{A}, 0, 1, 1_{j+2}, 0, \ldots, 0)$
0	$(A, 1, 1, 1_{j+2}, 0, \ldots, 0)$	$(B, 1, 1, 1_{j+2}, 0, \ldots, 0)$
-1	$(\bar{B}, 0, 1, 1_{j+2}, 0, \ldots, 0)$	$(B, 0, 1, 1_{j+2}, 0, \ldots, 0)$

Table 6.2: Updating Rules for $A[j]$ and $B[j]$ [ErLa91]

of $T[j]$, when $z_j = 1$ and $z_j = -1$, are maintained at each iteration. Letting $A[j-1] = -S[j-1] - 2^{-(j+1)}$ and $B[j-1] = S[j-1] - 2^{-(j+1)}$, the value of $T[j]$ is multiplexed from $A[j-1]$, $B[j-1]$, or 0 depending on the value of z_j. Table 6.2 shows the original algorithm's updating rules for $A[j]$ and $B[j]$ at the bit level. Based on the value of z_j, the values of $A_i[j]$ and $B_i[j]$ (the ith bits of $A[j]$ and $B[j]$, respectively) are the complemented or uncomplemented value of $A_i[j-1]$ or $B_i[j-1]$ when $i < j$. A later section shows that a variation of these rules is necessary to minimize the iteration delay of an FPGA implementation.

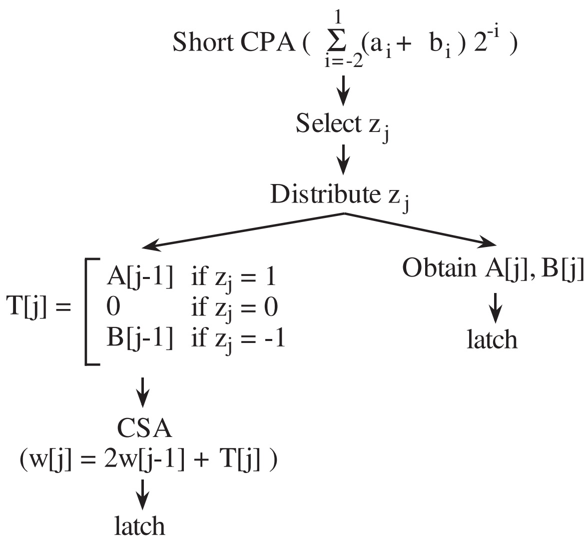
6.3 The Selection Module

The selection module contains the critical path of the variable precision design. In this section the process for efficiently mapping the square root selection logic to the XC4010 is presented.

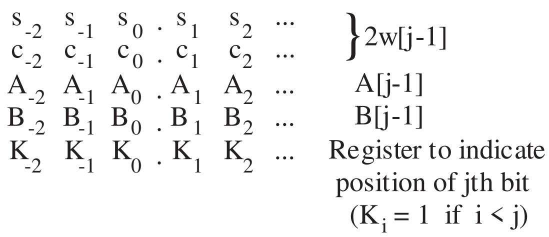
6.3.1 Creating an Efficient Mapping

For the defined square root algorithm, the algorithm steps and input bit configuration are shown in Figure 6.2. Typically the registers are placed after the carry save addition so that the sum and carry vectors (s and c) are saved for the next iteration. Table 6.3 shows the number of input bits to generate $s[j]$, $c[j]$, $A[j]$, $B[j]$, $K[j]$, and z_j for this register placement.

The first step in the selection function mapping rearranges the registers for result digit prediction. With prediction, selection is performed simultaneously with the computation of the residual so that the lower weighted bits in the data path are removed from the critical path. The repositioning replaces the bits s_i and c_i for $i \leq 1$ with the precomputed estimate of the residual ($\hat{w}_{-2}$, $\hat{w}_{-1}$, $\hat{w}_0$, and $\hat{w}_1$). Thus, the number of bits assimilated in the precomputation of z_j is reduced.



(a) Algorithm steps



(b) Registers for the Implementation

Figure 6.2: Original Square Root Algorithm Steps and Registers

Further input reduction is achieved as in the XC4010 SRT division implementation as described in the previous chapters and [LoEr92] [LoEr93b]. To summarize the additional changes:

1. $\hat{w}_{-2}$ and $\hat{w}_{-1}$ are eliminated because they can be deduced from z_j, $\hat{w}_0$ and $\hat{w}_1$ as shown in Table 5.1, and

2. register repositioning precomputes $\sum_{i=2}^{3}(s_i[j] + c_i[j])2^{-i}$ during step $j-1$, thus forming $c_1[j]$, $s_2[j]$, and $s_3[j]$ and eliminating $c_2[j]$ and $c_3[j]$.

With respect to encodings, the two-bit value of z_j uniquely represents T_{-2}, T_{-1}, T_0, and T_1. The operand range, recurrence equation, and selection function cause

Variable	No. of Bits (Original Algorithm)	No. of Bits (With Optimization)
s_{-2}, s_{-1}, c_{-2}, c_{-1}	8	-
s_0	8	-
$\hat{w}_0$	-	10
c_0	12	-
s_1	12	-
$\hat{w}_1$	-	8
s_2, c_1	12	-
s'_2, c'_1	-	13
s_3	12	-
s'_3	-	10
c_2, c_3	12	-
$s_i(i > 3)$	12	6
$c_i(i > 3)$	12	6
A_i, B_i	14	8
K_i	2	2
z_j	8	-
z_{j+1}	-	11

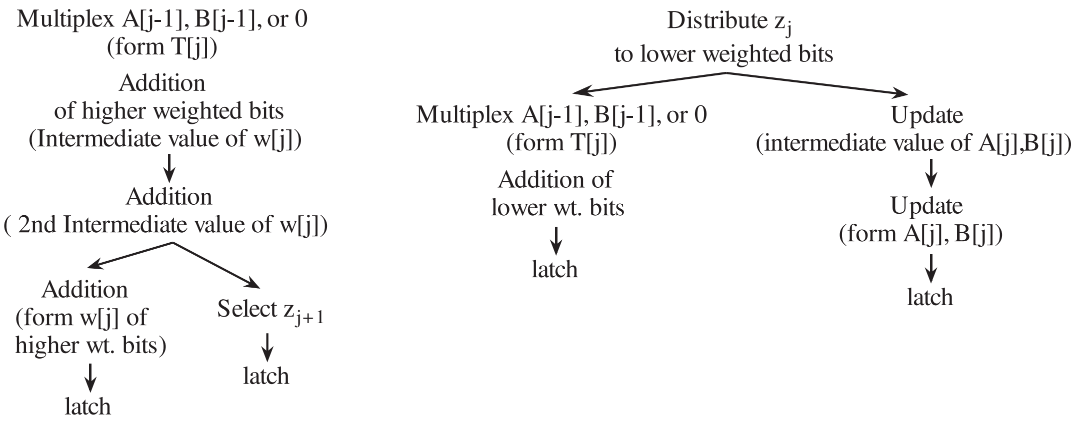
Table 6.3: Number of Input Bits to Compute Each Variable

$z_1 = z_2 = 1$ so that $|T[j]| > 1/2$ when $z_j = 1$ or -1 and $j > 1$. As a result, the four most significant bits of $T[j]$ can be encoded by z_j as follows:

for $j > 1$: if $z_j = 1$ then $T_{-2}T_{-1}T_0T_1 = 1110$
if $z_j = 0$ then $T_{-2}T_{-1}T_0T_1 = 0000$
if $z_j = -1$ then $T_{-2}T_{-1}T_0T_1 = 0001$
for $j = 1$: $T_{-2}T_{-1}T_0T_1 = 1111$
(incorporated into the initial condition)

The algorithm steps and register definition as applied to square root are shown in Figure 6.3. Table 6.3 summarizes the square root input reductions. The number of bits to compute z_{j+1}, s'_2, c'_1, and $\hat{w}_0$ increases after the change, but the maximum number of combinational logic input bits is reduced overall. With the reduction, the computation of A_i and B_i is removed from the critical path.

In Figure 6.3, three CLB levels are required for each iteration of the imple-



(a) Algorithm steps

z_j^s	z_j^m	$\hat{w}_0$	.	$\hat{w}_1$	s'_2	s'_3	s_4	s_5	...
			.	c'_1			c_4	c_5	...
			.		A_2	A_3	A_4	A_5	...
			.		B_2	B_3	B_4	B_5	...
			.		K_2	K_3	K_4	K_5	...

(b) Registers for the Implementation

Figure 6.3: Square Root after Input Reduction

mentation. A reduced delay necessitates further input reduction on the selection function part as well as on the computation of $A[j]$ and $B[j]$. For the updating of $A[j]$ and $B[j]$, a more compact encoding achieves the necessary input reduction. The proposed scheme is a variation of those in [ErLa91] and [CS252a]. The method, described as follows, is derived by examining the relative two's complement bit representations of $S[j-1]$, $B[j-1]$, and $A[j-1]$ (Table 6.4). For $j > 1$ in Table 6.4, let $S_p[j-1]$ denote the least significant nonzero bit of $S[j-1]$. Let p[j-1] denote the position of this bit at step $j-1$. Since $\frac{1}{2} \leq S[j-1]$, $p[j-1]$ is always greater than or equal to 1. From Table 6.4, $B[j-1]$ and $A[j-1]$ are then simply described as:

$$B_i[j-1] = \begin{cases} S_i[j-1] & \text{if } i < p[j-1] \text{ or } i > j+1 \\ \bar{S}_i[j-1] & \text{if } p[j-1] \leq i \leq j+1 \end{cases}$$

$$A_i[j-1] = \begin{cases} S_i[j-1] & \text{if } i > j+1 \\ \bar{S}_i[j-1] & \text{otherwise} \end{cases}$$

	Bit Representation
$S[j-1]$	$000.1d_2d_3\ldots d_{p-1}1_p00\ldots 0_{j-1}0_j0_{j+1}00\ldots$
$B[j-1]$	$000.1d_2d_3\ldots d_{p-1}0_p11\ldots 1_{j-1}1_j1_{j+1}00\ldots$
$A[j-1]$	$111.0\bar{d}_2\bar{d}_3\ldots \bar{d}_{p-1}0_p11\ldots 1_{j-1}1_j1_{j+1}00\ldots$

Table 6.4: Two's Complement Bit Representations [LoEr93a] ©1993 IEEE

z_j	$p[j]$	$B_i[j]$
1	j	$B_i[j-1]$ if $i < p[j-1]$ or $i = j+1$ or $i > j+2$ $\bar{B}_i[j-1]$ otherwise
0	$p[j-1]$	$B_i[j-1]$ if $i \neq j+2$ $\bar{B}_i[j-1]$ otherwise
-1	j	$B_i[j-1]$ if $i < j$ or $i = j+1$ or $i > j+2$ $\bar{B}_i[j-1]$ otherwise

Table 6.5: The Computation of $B_i[j]$

Table 6.5 shows the bit representation of $B[j]$ with respect to $B[j-1]$, and Table 6.6 defines the value of each bit $B_i[j]$ as a function of j, $B_i[j-1]$, and $p[j-1]$. Both Tables 6.5 and 6.6 show four regions defining $B[j]$ relative to $B[j-1]$:

1. $i = j+2$ in which $B_i[j] = \bar{B}_i[j-1]$
2. $i = j$ in which $B_i[j] = B_i[j-1]$ if $z_j = 0$, and $B_i[j] = \bar{B}_i[j-1]$ if $z_j = -1$ or 1
3. $p[j-1] \leq i < j$ in which $B_i[j] = B_i[j-1]$ if $z_j = 0$ or -1, and $B_i[j] = \bar{B}_i[j-1]$ if $z_j = 1$
4. $i < p[j-1]$ or $i = j+1$ or $i > j+2$ in which $B_i[j] = B_i[j-1]$

	Bit Representation
$B[j-1]$	$000.1d_2d_3\dots d_{p-1}0_p11\dots 1_{j-1}1_j1_{j+1}0_{j+2}00\dots$
$B[j]$ when $z_j = 1$	$000.1d_2d_3\dots d_{p-1}1_p00\dots 0_{j-1}0_j1_{j+1}1_{j+2}00\dots$
$B[j]$ when $z_j = 0$	$000.1d_2d_3\dots d_{p-1}0_p11\dots 1_{j-1}1_j1_{j+1}1_{j+2}00\dots$
$B[j]$ when $z_j = -1$	$000.1d_2d_3\dots d_{p-1}0_p11\dots 1_{j-1}0_j1_{j+1}1_{j+2}00\dots$

Table 6.6: Two's Complement Bit Representations of $B[j] = f(B[j-1], z_j)$

Two bits minimally specify the region for each bit of $B_i[j]$. Thus two new vectors, E and H, replace the original control vector K. In the original algorithm, four bits of K identify the bit positions j, $j+1$, and $j+2$. The new B, E, and H are specified by the following boolean expressions:

1. $f_1 = (E_i[j-1]\bar{H}_i[j-1]) + (E_i[j-1]H_i[j-1]z_j^m) + (\bar{E}_i[j-1]H_i[j-1]\bar{z}_j^s z_j^m)$
2. $B_i[j] = f_1 \oplus B_i[j-1]$
3. $E_i[j] = E_{i-1}[j-1]$
4. $H_i[j] = (H_i[j-1]\bar{H}_{i+1}[j-1]) + (\bar{H}_i[j-1]H_{i-1}[j-1]) + (H_i[j-1]\bar{z}_j^m)$

When $j = 1$:

1. $B_2[0] = 1$ and $B_i[0] = 0$, $(i > 2)$
2. $E_1[0] = E_3[0] = 1$ and $E_i[0] = 0$, $(i \neq 1, 3)$
3. $H_1[0] = 1$ and $H_i[0] = 0$, $(i \neq 1)$

Apparent from Table 6.4, $A[j-1]$ is easily derived from $B[j-1]$:

$$A_i[j-1] = \begin{cases} B_i[j-1] & \text{if } i \geq p[j-1] \\ \bar{B}_i[j-1] & \text{otherwise} \end{cases}$$

Therefore one additional vector, I, identifies the two regions for the bits $A_i[j-1]$. Maintaining the vector I results in a simpler function with fewer required input bits than maintaining the vector A (which would be similar to the function for B).

Variable	No. of Bits Before Encoding	No. of Bits After Encoding
B_i	8	5
A_i	8	-
I_i	-	4
H_i	-	4
E_i	-	1
K_i	2	-

Table 6.7: Number of Input Bits to Compute Each Variable Before and After Encoding [LoEr93a] ©1993 IEEE

Hence, the implementation of I requires fewer CLBs. The corresponding boolean expression for I is:

$$I_i[j] = (H_i[j-1]\bar{H}_{i+1}[j-1]) + I_i[j-1](\bar{H}_i[j-1] + \bar{z}_j^m)$$

where $I_i[0] = 1$ for all i. The resulting input reduction with the new encoding is shown in Table 6.7. Although this encoding creates one additional vector as compared to the original algorithm [ErLa91], all of the associated vectors can be computed in one logic level and require fewer CLBs to implement. With the XC4010, a single CLB maps two bits of the new control vector, either (1) $B_i[j]$ and $H_i[j]$ or (2) two bits of $I_i[j]$. The vector E does not require any additional CLBs because it maps to those having utilized lookup tables but unutilized flip flops.

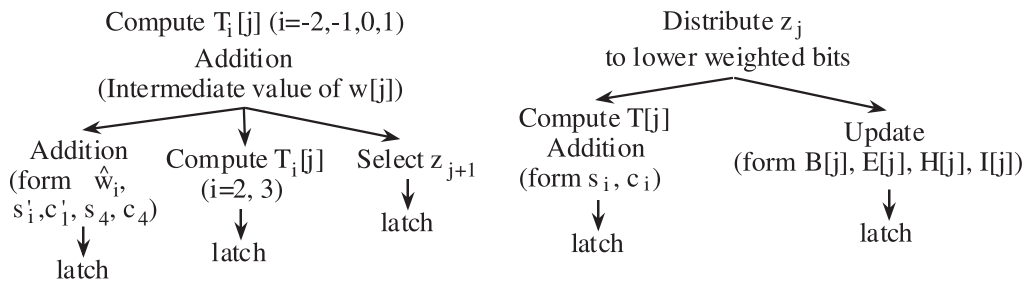
The overall iteration delay is further reduced to two CLB levels by performing input reduction on z_j and the most significant bits of the residual. Repositioning the registers and precomputing $T_2[j+1]$ and $T_3[j+1]$ in step j achieves this goal. The resulting input reduction is summarized in Table 6.8, and the final implementation description is shown in Figure 6.4.

6.4 LSA Modules for Square Root

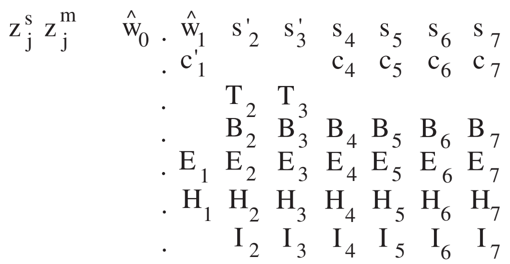
For the data path, the LSA conversion algorithm generates Figure 6.5 where the superscripts denote each variable's computational step. Each LSA module encompasses four contiguous weights in the residual bits. The modules may be attached to the square root design at any point where the sum and carry bits (s_i and c_i) of

Variable	No. of Bits Before Precomputing	No. of Bits After Precomputing
$\hat{w}_0$	10	8
$\hat{w}_1$	8	4
s'_2, c'_1	13	12
z_j	11	9

Table 6.8: Number of Input Bits to Compute Each Variable Before and After Precomputing T_2 and T_3



(a) Algorithm steps



(b) Registers for the Implementation

Figure 6.4: Square Root Selection Module after Final Input Reduction

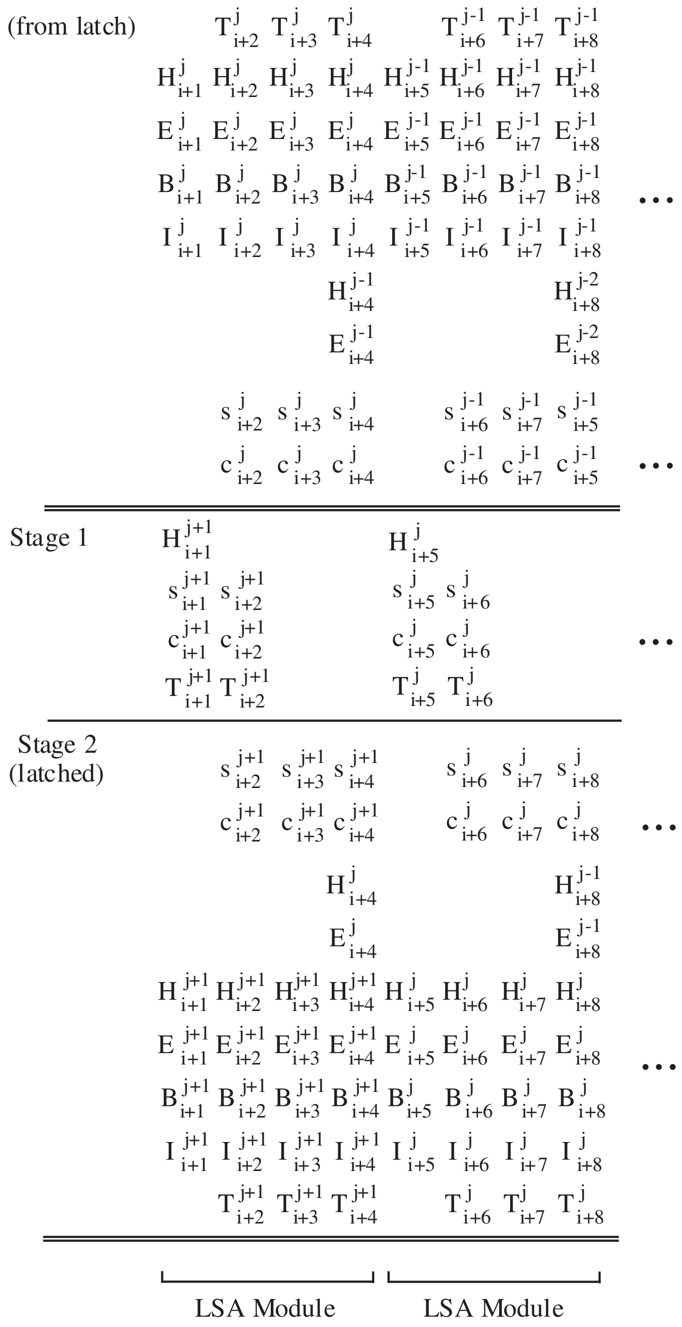


Figure 6.5: Square Root LSA After Conversion

the residual are computed and latched. Hence, the LSA implementation serves as an extension and/or replacement of the lower weighted residual bits of an existing broadcasting design. Figure 6.6 shows the transfer of data between modules as defined with the LSA conversion algorithm. After further optimizations as discussed in Section 2.3.3 and Chapter 4, Figure 6.7 shows the resulting transfer of data with reduced number of bits passed between modules. Figure 6.8 then shows the details of two appended LSA modules. Each module in Figure 6.8 requires four fewer latches than in the design of Figure 6.7.

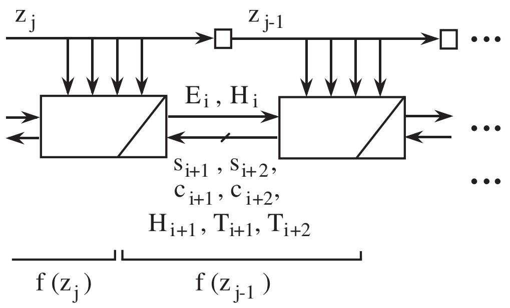


Figure 6.6: Modules After LSA Conversion

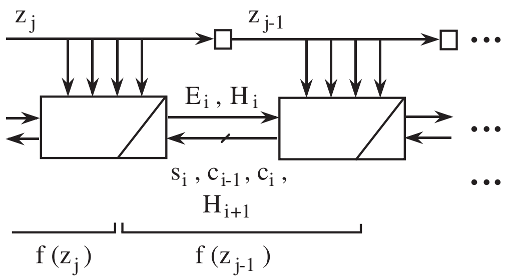


Figure 6.7: LSA Modules After Optimizations

An implementation of the linear sequential array requires two logic levels, one 5-input lookup table and one 4-input lookup table. Fanout of data between stages of the pipeline is small and locally distributed to achieve low interconnection times. Since the selection module requires two logic levels consisting of two 5-input lookup tables, the delay of the LSA is less than the computation delay of the selection. Therefore, due to its pipelined nature, LSA stages and hence precision may be added freely within an FPGA chip without increasing the throughput delay of the square root design. High precision implementations then experience the same cycle

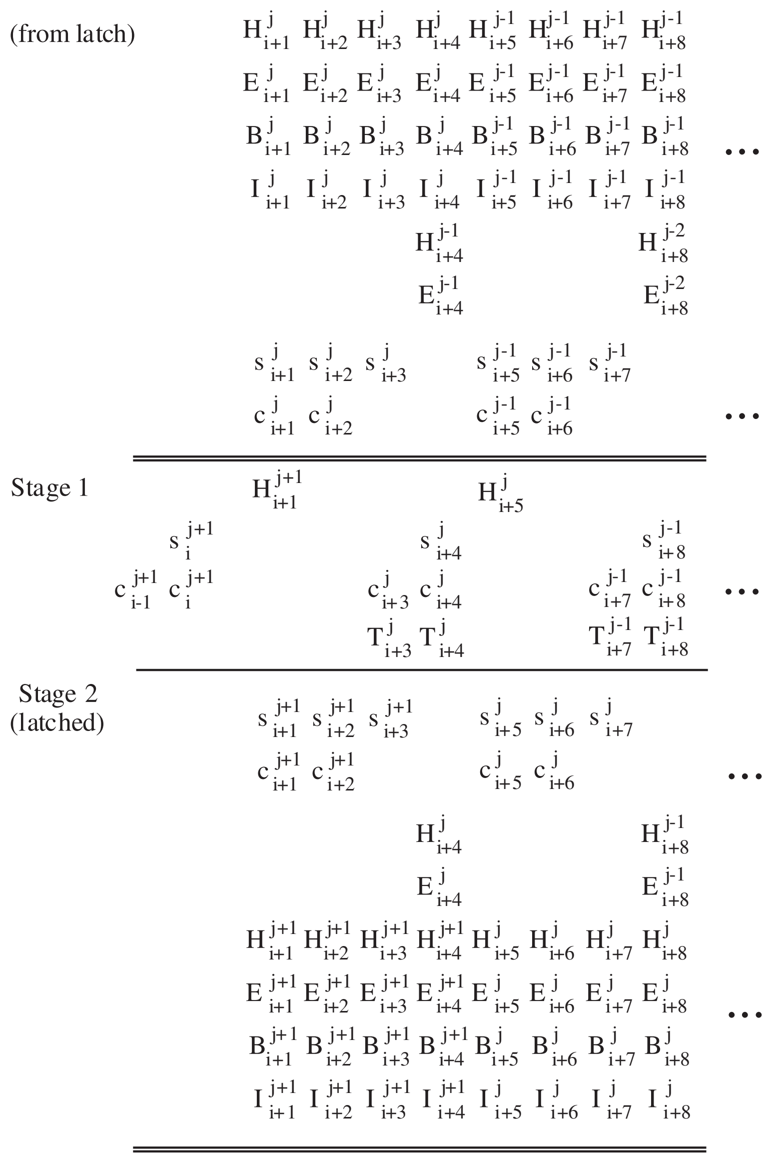


Figure 6.8: The LSA Design After Optimizations

time as lower precision designs. Figure 6.9 shows an LSA chip interface module for the square root design.

Regarding area, the implementation of an LSA module's 4 bits of added precision requires 16.5 CLBs. This includes overhead for the operand registers as well as initialization logic. For a conventional broadcasting scheme without the linear sequential array, the lower weighted bits of the residual require 14 CLBs for every 4 bits of precision, including initialization overhead and operand registers.

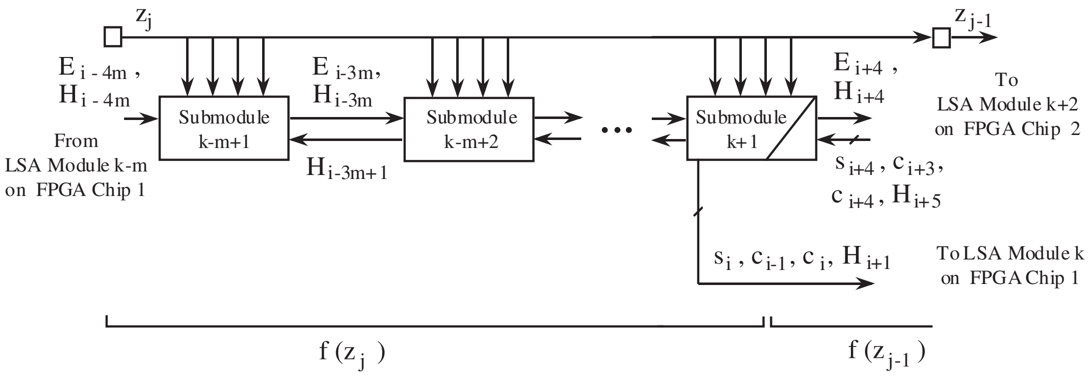


Figure 6.9: The LSA Chip Interface Module for Square Root

Variable precision within an FPGA thus costs 18% more in data path area over a conventional broadcasting scheme.

6.5 Implementation Results

The variable precision square root design is implemented with the XC4010 speed grade -5. The selection logic has been tested in a single precision (24-bit mantissa only) design where the lower weighted bits are implemented with conventional broadcasting of the square root digits. One LSA module was appended to this design at the 2^{-20} weighted bit (replacing bits of weight 2^{-21} to 2^{-24}) and verified not to exceed the critical path delay of the selection logic.

The selection module is compared to an equivalent baseline estimate (Table 6.9) for the original steps of Figure 6.1. The estimate makes three assumptions: (1) a selection module of 7 fractional bits is created, (2) each algorithm step of Figure 1 is implemented separately, and (3) the CPA is implemented with the XC4010 fast CPA hardware. In each of the implementations of Table 6.9, the block count includes both the logic to perform the iteration cycles and the registers for the most significant 7 bits of the operands. In each of the cases, the output is serially available in signed digit form. The resulting comparison shows that the proposed optimization improves the performance by 38% over a baseline estimate. By creating compact encoding schemes, the optimizations can also reduce the number of logic blocks by 40%.

In a comparison of the LSA modules with the conventional design (Table 6.9), the LSA requires 18% more CLBs for implementation. Due to this additional cost, the LSA modules are most efficiently appended to the single precision con-

Module	No. of CLBs	Speed Grade -5 Cycle Time
Variable Precision Selection Module (7 fractional bits)	38	22.3 ns
Baseline Estimate (7 fractional bits)	63	36.0 ns
Variable Precision LSA Module (4 bits of precision)	16.5	–
Conventional Design Broadcasting (4 fractional bits)	14	–

Table 6.9: Timing and Area Comparisons

Square Root Design	Operand Precision (number of bits)	Computation Delay
LSA Square Root Modules	24	558 ns
TMS390S10 microSPARC [TI92]	24	740 ns
LSA Square Root Modules	56	1.3 μs
TMS390S10 microSPARC [TI92]	56	1.3 μs

Table 6.10: Performance Comparisons for the LSA Square Root Design

ventional design rather than replacing the lower weighted bits of the conventional design. The conventional design on the XC4010 speed grade -5 cannot exceed single precision without increasing the cycle time.

Extrapolated performance results for the square root LSA design are presented in Table 6.10. Table 6.10 shows that the LSA square root performance is comparable to that of the Texas Instruments TMS390S10 microSPARC [TI92] general purpose microprocessor. Applications needing higher precisions are often relegated to software implementations [Bren76, BuWa89, Kana88]. Table 6.11 summarizes the extrapolated LSA results with high precision software implementations on the Cray 2 [BuWa89]. The extrapolated LSA results assume use of the LSA chip inter-

Operation	Operand Precision (number of bits)	Computation Delay
LSA Square Root	334	7.5 μs
LSA Division	334	6.4 μs
LSA Multiplication	334	2.2 μs
Cray division software [BuWa89] (vectorized)	334	116 μs
Cray division software [BuWa89] (no vectorization)	334	139 μs
Cray multiplication software [BuWa89] (vectorized)	334	55 μs
Cray multiplication software [BuWa89] (no vectorization)	334	232 μs

Table 6.11: Performance Comparisons of High Precision Operations

face modules so that operation latency is independent of interchip communication delays. For the Cray, the reported timings are the averaged results for a single operation when 10^6 operations of two 100-decimal digit operands are performed. To create a fair comparison, normalization time is excluded. Cray performance both with and without vectorization are given.

Although both the LSA and conventional broadcasting designs produce one signed square root output digit per iteration, only the conventional design obtains a two's complement representation simply by computing $S = B[n] \oplus I[n]$ after n iterations where n is the precision of the output. Due to pipelining, $B[n]$ and $I[n]$ are derived at different times for the various LSA modules. A total of m additional iterations are needed for all modules to generate $B[n]$ and $I[n]$ where m is the number of LSA modules in the design.

Additional required hardware that is not included in the block count of Table 6.9 are a counter to identify completion and the final *xor* logic and latching scheme for the two's complement result. Future work can include the development of a separate on-the-fly converter [ErLa87] to convert the signed digit output to two's complement with fewer additional iterations.

CHAPTER 7

Summary and Future Work

With flexibility and rapid configurability, field programmable gate arrays offer a hardware alternative to application-specific integrated circuits (ASICs). FPGA boards such as PAM [BeRV93], Splash [ArBD92], Virtual Computer [Cass93], etc., can attach to existing computer systems to provide customizable coprocessors. Thus applications that cannot afford to develop an ASIC due to development time or cost constraints may find a hardware solution in FPGAs.

This dissertation has explored variable-precision digit-recurrence arithmetic with lookup-table based FPGAs. Digit-recurrence algorithms have typically used broadcasting in the data path, but broadcasting delays cause the cycle time to grow as a function of precision. By applying a bit-level systolic array approach, large fanout broadcasting can be avoided. Cycle time thus becomes independent of precision.

Bit-level systolic arrays (e.g., [MCMW87, IrOw89]) feature processing modules that compute only one bit-slice or digit-slice at each cycle. For computing each bit slice, the necessary input data is always available from nearest neighboring modules at the beginning of the cycle. One problem with the single bit-slice/digit-slice approach is that algorithms having localized but not nearest-neighbor dependencies do not meet the bit-level systolic design requirements. Also, due to pipelining, neighboring modules operate on different algorithm steps; thus a module may require data that other modules have not yet computed. This dissertation presents the linear sequential array to address the above difficulties with bit-level systolic arrays. The LSA extends the definition of a processing module to allow multiple bit slices. The number of bit slices in a module is minimally defined to satisfy all localized data dependencies without limitation to dependencies on nearest neighbor bit slices. Since all bit slices within an LSA module operate on the same algorithm step, the internal structure of the module uses very small fanout broadcasting rather than pipelining. The LSA is therefore a hybrid bit-level systolic array in which the global design is a linear bit-level systolic array, but in which the modules internally use a conventional broadcasting scheme. The broadcasting delays within a module are comparable to local routing delays since the result digit is passed to only a few bit slices.

An LSA's data path logic delay is at most twice that of a conventional broadcasting scheme. As shown by the multiplier design, LSA variations can feature the same data path delay found in a broadcasting scheme. By also using result digit prediction, the LSA can proceed in parallel with the digit selection process. The combination of LSA and result digit prediction generally prevents the LSA modules from entering the critical path because the data path usually requires only a small and simple subset of functions found in the selection process (multiple generation and redundant addition). Regarding area, the amount of combinational logic for the LSA is comparable to a conventional broadcasting approach, but due to pipelining the LSA requires more latches.

A majority of current bit-level systolic designs are two-dimensional structures [MCMW87, IrOw89, MQMC92, DaFe92]. Linear configurations are more desirable than two-dimensional designs since they require less area and are more rapidly constructed. For division and square root, however, bit-level systolic designs have been two-dimensional structures [MQMC92, DaFe92] due to the difficultly in satisfying linear array constraints for these algorithms [IrOw87]. Now, with the LSA approach, linear systolic array designs have been developed for division and square root. LSA designs can be created for any digit-recurrence algorithm featuring localized data dependencies. An LSA can either replace or attach to an existing broadcasting data path design.

In summary of the LSA research, this dissertation develops:

1. a generalization of linear sequential arrays for any digit-recurrence arithmetic data path having localized data dependencies,

2. an algorithm to convert any broadcasting scheme in digit-recurrence arithmetic to an LSA approach, and

3. extensions for the LSA to avoid performance degradation from interchip communication in multiple chip LSA designs.

The LSA is applicable to any technology. While it has been applied to FPGAs for this study, it can be useful in ASIC implementations. For example, multi-chip modules (MCMs) can be used to build very high precision arithmetic chips where the LSA chip interface module compensates for communication delays between die on the MCM.

With the LSA to avoid the broadcasting delay problem, the selection function determines the critical path. Digit-recurrence selection functions are characterized by a serial reduction of data, generally including a carry-propagate addition and bit-wise comparison. For the selection function of digit-recurrence algorithms, this study presents:

1. mapping strategies and a mapping method to implement selection functions with lookup-table based FPGAs, and
2. a simple approach to result digit prediction for radix-2 digit-recurrence algorithms via register repositioning.

The mapping strategies improve performance through preliminary data reductions in parallel with the critical path. By precomputations and substitution of functions, data dependencies can be moved across bit positions to balance the delay of combinational logic paths. Repositioning the registers within an algorithm achieves these precomputations. By also using encodings, a function may have fewer input bits than the lookup-table input size so that several algorithm steps may be merged into one logic level. Simple analysis of the selection function's behavior over successive algorithm steps can reveal additional encoding possibilities. An example of algorithm analysis and the creation of encodings for improved implementation is demonstrated with the square root implementation of Chapter 6.

Result digit prediction provides further merging opportunities by allowing the data reduction process for each algorithm step (CSA, CPA, and comparison) to occur within the same cycle. In most digit-recurrence algorithms, each cycle consists of the following sequence of combinational logic steps: CPA, comparison, CSA. With this ordering of steps, merging is difficult because a comparison generally requires known values for the most significant bits first; hence, the carries need to be completely propagated through the addition. Also, the carry-save adder requires knowledge of the new added term which is dependent on the result of the comparison. By implementing result digit prediction, the ordering of combinational logic steps is rearranged so that the CSA is immediately followed by the CPA. Merging opportunities can then be found within these two steps; several examples of merging with a CSA and CPA are provided in Chapter 4. Further merging possibilities are offered by using the predicted result digit as an encoding to reduce the number of bits in the representation of the residual estimate. Prediction also assists in removing the data path from the critical path because digit selection can proceed simultaneously with updating of the data path.

With the mapping method, encodings and function substitutions reduce the number of bits within the algorithm. Examples are given in Chapters 4, 5, and 6. Variations of the algorithm steps are then created for efficient implementation.

Using both the LSA technique and mapping methods, designs have been developed with the Xilinx XC4010 for radix-16 multiplication, SRT division, and radix-2 square root. These designs perform comparably to a general purpose microprocessor for single and double precision. Extrapolated performance results for high precision show that the presented designs exceed the performance of equivalent software implementations on a Cray by at least a factor of 18.

7.1 Future Work

A continuation of this study can create FPGA designs for on-line arithmetic. With on-line arithmetic [Erce77, Erce84], operations can be pipelined to rapidly compute more complex functions. On-line algorithms have similar characteristics to digit-recurrence algorithms in which a limited precision selection function computes each result digit and uses that digit in a feedback process. The mapping method presented in this dissertation is thus applicable to the on-line approach. From the parallelization of operations, an on-line design can feature better performance than conventional arithmetic for a series of operations.

Libraries of arithmetic selection functions and associated LSA modules can allow the rapid development of application-specific architectures on FPGAs. Implementation of the desired functions and precisions can then be created by appending the appropriate modules from the library.

Future work can also formulate a generalized mapping theory for arithmetic. This theory can then be applied to a computer-aided design (CAD) tool. Current CAD tools are designed for random logic and do not take advantage of the regular characteristics within algorithm classes (e.g., digit-recurrence arithmetic). A CAD tool specifically for digit-recurrence arithmetic would not only include optimizations of boolean expressions, but would also perform a simple analysis of the selection function. Through algorithm analysis, encodings schemes can be found that are not directly apparent from the boolean expressions. An example of such encodings is demonstrated in the square root algorithm analysis of Chapter 6, where the boolean expressions defining $B[j]$ and $A[j]$ are changed for efficient implementation.

Technology mapping is also part of the computer-aided design process. Thus, new mapping strategies for arithmetic can be created for other FPGA technologies such as the multiplexor-based Actel chips [Act94].

For the LSA approach, further work can update it from a systolic scheme with a global clock to a wavefront array scheme [KLJH87] with self-timing. With a wavefront approach, all large fanout broadcasting, including the clock signal, can be eliminated.

The multiple bit slice modules of the LSA can also be generalized for other systolic configurations, such as two-dimensional arrays and complex recurrences.

References

[Act94] Actel, "FPGA Data Book and Design Guide," 1994.

[ArBD92] J.M. Arnold, D.A. Buell, E. Davis, "Splash 2," Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 316-322.

[Aviz61] A. Avižienis, "Signed Digit Number Representations for Fast Parallel Arithmetic," IRE Trans. Electron. Comput., Vol. EC-10, No. 3, Sept. 1961, pp. 389-400.

[Aviz62] A. Avižienis, "On Flexible Implementation of Digital Computer Arithmetic," Proc. IFIP Congress '62, Aug. 27 - Sept. 1, 1962, Munich, pp. 664-670.

[Aviz64] A. Avižienis, "Binary-Compatible Signed-Digit Arithmetic," Proc. Fall Joint Computer Conference, 1964, pp. 663-672.

[Ber89] P. Bertin, et al., "Introduction to Programmable Active Memories," DEC Research Report No. 3, Paris Research Laboratory, 1989.

[BeRV93] P. Bertin, D. Roncin, J. Vuillemin, "Programmable Active Memories: a Performance Assessment," DEC Research Report No. 24, Paris Research Laboratory, 1993.

[BrEW89] R.H. Brackert Jr., M.D. Ercegovac, A.N. Willson Jr., "Design of an On-Line Multiply-Add Module for Recursive Digital Filters," Proc. 9th IEEE Symposium on Computer Arithmetic, Sept. 6-8, 1989, Santa Monica, CA, pp. 34-41.

[Bren76] R. Brent, "Fast Multiple-Precision Zero-Finding Methods and the Complexity of Elementary Function Evaluation," Journal of the ACM, Vol. 23, No. 2, April 1976, pp. 242-251.

[Booth51] A.D. Booth, "A Signed Binary Multiplication Technique,", Quart. Journ. Mech. Appl. Math., Vol. 4, Part 2, 1951, pp. 236-240.

[BuWa89] D.A. Buell, R.L. Ward, "A Multiprecise Integer Arithmetic Package," Journal of Supercomputing, 3:89-107, 1989.

[CaLu88] J.R. Cavallaro, F.T. Luk, "CORDIC Arithmetic for an SVD Processor," Journal of Parallel and Distributed Computing, Vol. 5, No. 3, June 1988, pp. 271-290.

[Cass93] S. Casselman, "Virtual Computing and the Virtual Computer," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, April 5-7, 1993, Napa Valley, CA, pp. 43-48.

[ChME93] C.-J. Chou, S. Mohanakrishnan, J. B. Evans, "FPGA Implementation of Digital Filters," Proc. 4th Int'l Conference on Signal Processing Applications and Technology, Sept. 28 - Oct. 1, 1993, Santa Clara, CA, pp. 80-88.

[CS252a] M.D. Ercegovac, T. Lang, UCLA Computer Science Department CS252a Course Notes, 1988.

[DaFe92] H. Dawid, G. Fettweis, "Bit-Level Systolic Carry-Save Array Division," Proc. IEEE Global Telecommunications Conference (GLOBECOM '92), Dec. 6-9, 1992, Orlando, FL, pp. 484-488.

[Desp74] A.M. Despain, "Fourier Transform Computers using CORDIC Iterations," IEEE Trans. on Computers, Vol. C-23, No. 10, Oct. 1974, pp. 993-1001.

[DuMu93] J. Duprat, J.-M. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," IEEE Trans. on Computers, Vol. 42, No. 2, Feb. 1993, pp. 168-178.

[Erce77] M.D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer," IEEE Trans. on Computers, Vol. C-26, No. 7, July 1977, pp. 667-680.

[Erce84] M.D. Ercegovac, "On-Line Arithmetic: An Overview," Proc. SPIE Vol. 495 – Real Time Signal Processing VII, 1984, pp. 86-92.

[ErLa85] M.D. Ercegovac, T. Lang, "A Division Algorithm with Prediction of Quotient Digits," Proc. 7th IEEE Symposium on Computer Arithmetic, 1985, pp. 51-56.

[ErLa87] M.D. Ercegovac, T. Lang, "On-the-fly Conversion of Redundant into Conventional Representations," IEEE Trans. on Computers, Vol. C-36, No. 7, July 1987, pp. 895-897.

[ErLa89] M.D. Ercegovac, T. Lang, "Fast Radix-2 Division with Quotient Digit Prediction," Journal of VLSI Signal Processing, Vol. 1, 1989, pp. 169-180.

[ErLa91] M.D. Ercegovac, T. Lang, "Module to Perform Multiplication, Division, and Square Root in Systolic Arrays for Matrix Computations," Journal of Parallel and Distributed Computing, Vol. 11, 1991, pp. 212-221.

[ErLa94] M.D. Ercegovac, T. Lang, *Digit-Recurrence Algorithms and Implementations for Division and Square Root*, Kluwer Academic Publishers, Boston, 1994.

[Fern93] J.S. Fernando, *Design Alternatives for Recursive Digital Filters Using On-Line Arithmetic*, Ph.D Dissertation, University of California, Los Angeles, 1993.

[Foun88] T.J. Fountain, "The Use of Linear Arrays for Image Processing," Proc. Int'l Conference on Systolic Arrays, May 25-27, 1988, San Diego, CA, pp. 183-192.

[FoWa87] J.A.B. Fortes, B.W. Wah, "Systolic Arrays – From Concept to Implementation," IEEE Computer, Vol. 20, No. 7, July 1987, pp. 12-17.

[FrRV91] R. Francis, J. Rose, Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," Proc. 28th ACM/IEEE Design Automation Conference, June 17-21, 1991, San Francisco, CA, pp. 227-233.

[Hu92] Y.H. Hu, "CORDIC-based VLSI Architectures for Digital Signal Processing," IEEE Signal Processing Magazine, Vol. 9, No. 3, July 1992, pp. 16-35.

[Hwan79] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley and Sons, 1979.

[IrOw87] M.J. Irwin, R.M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial," IEEE Computer, Vol. 20, No. 4, April 1987, pp. 61-73.

[IrOw89] M.J. Irwin, R.M. Owens, "Digit Serial Systolic VLSI Architectures," Proc. Int'l Conference on Systolic Arrays, 1989, Killarney, Ireland, pp. 215-224.

[Kana88] Y. Kanada, "Vectorization of Multiple-Precision Arithmetic Program and 201,326,000 Decimal Digits of Pi Calculation," Proceedings of Supercomputing 88, Vol. II, Science and Applications, pp. 117-128.

[Korn93] P. Kornerup, "High-Radix Modular Multiplication for Cryptosystems," Proc. 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 277-283.

[KuLa84] H.T. Kung, M.S. Lam, "Fault-Tolerance and Two-Level Pipelining in VLSI Systolic Arrays," Proc. Conference on Advanced Research in VLSI, Jan 23-25, 1984, MIT, Cambridge, MA, pp. 74-83.

[KuLe79] H.T. Kung, C.E. Leiserson, "Systolic Arrays (for VLSI)," Sparce Matrix Proc., I.S. Duff and G.W. Stewart (eds.), SIAM, 1979, pp. 256-282.

[KLJH87] S.Y. Kung, S.C. Lo, S.N. Jean, J.N. Hwang, "Wavefront Array Processors – Concept to Implementation," IEEE Computer, Vol. 20, No. 7, July 1987, pp. 18-33.

[LeRS83] C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimizing Synchronous Circuitry by Retiming," Third Caltech Conference on Very Large Scale Integration, 1983, pp. 87-116.

[LeSa83] C.E. Leiserson, J.B. Saxe, "Optimizing Synchronous Systems," Journal of VLSI and Computer Systems, Vol. 1, No. 1, 1983, pp. 41-67.

[LiWa85] G.-J. Li, B.W. Wah, "The Design of Optimal Systolic Arrays," IEEE Trans. on Computers, Vol. C-34, No. 1, Jan. 1985, pp. 66-77.

[LoEr92] M.E. Louie, M.D. Ercegovac, "Mapping Division Algorithms to Field Programmable Gate Arrays," Proc. 26th Asilomar Conference on Signals, Systems, and Computers, Oct. 26-28, 1992, Pacific Grove, CA, pp. 371-375.

[LoEr93a] M.E. Louie, M.D. Ercegovac, "A Digit-Recurrence Square Root Implementation for Field Programmable Gate Arrays," Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, April 5-7, 1993, Napa Valley, CA, pp. 178-183.

[LoEr93b] M.E. Louie, M.D. Ercegovac, "On Digit-Recurrence Division Implementations for Field Programmable Gate Arrays," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 202-209.

[LoEr94a] M.E. Louie, M.D. Ercegovac, "A Variable Precision Multiplier for Field Programmable Gate Arrays," 2nd Int'l ACM/SigDA Workshop on Field Programmable Gate Arrays, Feb. 13-15, 1994, Berkeley, CA.

[LoEr94b] M.E. Louie, M.D. Ercegovac, "Linear Sequential Arrays: Pipelining Arithmetic Data Paths," Technical Report CSD-940010, Computer Science Dept., University of California, Los Angeles, March 1994.

[LoEr94c] M.E. Louie, M.D. Ercegovac, "Implementing Division with Field Programmable Gate Arrays," to appear in the Journal of VLSI Signal Processing (special issue on computer arithmetic), vol. 7, no. 3, 1994, pp. 271-285.

[LoEr94d] M.E. Louie, M.D. Ercegovac, "A Variable Precision Square Root Implementation for Field Programmable Gate Arrays," to appear in the Journal of Supercomputing, 1994.

[Lyon76] R.F. Lyon, "Two's Complement Pipeline Multipliers," IEEE Trans. on Communications, Vol COM-24, No. 4, April 1976, pp. 418-425.

[Megs92] G.M. Megson, *An Introduction to Systolic Algorithm Design*, Oxford University Press, New York, 1992.

[MCMK90] J.V. McCanny, J.G. McWhirter, S.-Y. Kung, "The Use of Data Dependence Graphs in the Design of Bit-Level Systolic Arrays," IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol. 38, No. 5, May 1990, pp. 787-793.

[MCMW82] J.V. McCanny, J.G. McWhirter, "Implementation of Signal Processing Functions Using 1-Bit Systolic Arrays," Electronics Letters, Vol. 18, No. 6, 1982, pp. 241-243.

[MCMW86] J.V. McCanny, J.G. McWhirter, "The Derivation and Utilisation of Bit Level Systolic Array Architectures," Proc. 1st Int'l Workshop on Systolic Arrays, July 2-4, 1986, Oxford, pp. 47-59.

[MCMW87] J.V. McCanny, J.G. McWhirter, "Some Systolic Array Developments in the United Kingdom," IEEE Computer, Vol. 20, No. 7, July 1987, pp. 51-63.

[MQMC92] S.E. McQuillan, J.V. McCanny, "VLSI Module for High-Performance Multiply, Square Root and Divide," IEE Proceedings-E, Vol. 139, No. 6, Nov. 1992, pp. 505-510.

[MiYi90] G. De Micheli, R. Yip, "Logic Transformations for Synchronous Logic Synthesis," Proc. of the Hawaii Int'l Conference on System Sciences, Jan. 2-5, 1990, Kailua-Kona, Hawaii, pp. 407-416.

[Mont85] P.L. Montgomery, "Modular Multiplication without Trial Division," Mathematics of Computation, Vol. 44, No. 170, 1985, pp. 519-521.

[MoCi94] P. Montuschi, L. Ciminiera, "Radix-2 Division with Quotient Digit Prediction without Prescaling," Proc. of the Hawaii Int'l Conference on System Sciences, Jan. 1994.

[MSBS90] S. Malik, E.M. Sentovich, R.K. Brayton, A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques," Proc. of the Hawaii Int'l Conference on System Sciences, Jan. 2-5, 1990.

[Rob58] J.E. Robertson,"A New Class of Digital Division Methods," IRE Trans. on Electronic Computers, Sept. 1958, pp 88-92.

[ShBV91] M. Shand, P. Bertin, J. Vuillemin, "Hardware Speedups in Long Integer Multiplication," Computer Architecture News, 19(1):106-114, 1991.

[ShVu93] M. Shand, J. Vuillemin, "Fast Implementations of RSA Cryptography," Proc. of the 11th IEEE Symposium on Computer Arithmetic, June 29 - July 2, 1993, Windsor, Ontario, Canada, pp. 252-259.

[TI92] Texas Instruments TMS390S10 microSPARC Reference Guide, Nov. 1992.

[Tu90] P. K.-G. Tu "On-Line Arithmetic Algorithms for Efficient Implementation," Ph.D. Dissertation, Technical Report CSD-900029, Computer Science Department, University of California, Los Angeles, Sept. 1990.

[Van92] D.E. Van den Bout, J.N. Morris, D. Thomae, et al., "An FPGA-based, Reconfigurable System," IEEE Design & Test of Computers, Sept. 1992, pp. 21-30.

[Walt71] J.S. Walther, "A Unified Algorithm for Elementary Functions," Proc. AFIPS Spring Joint Computer Conference, 1971, pp. 379-385.

[WoSh88] A. Wolfe, J.P. Shen, "Flexible Processors: A Promising Application-Specific Processor Design Approach," Proc. of the 21st Annual Workshop on Microprogramming and Microarchitecture, Nov.30-Dec.2, 1988, San Diego, CA, pp.30-39.

[XACT] Xilinx XACT4000 Design Implementation Software.

[Xil91] Xilinx, *The Programmable Gate Array Data Book*, 1991.

[Xil92] Xilinx, "XC4000 Logic Cell Array Family - Technical Data," 1992.