# DESIGN ISSUES IN HIGH PERFORMANCE FLOATING POINT ARITHMETIC UNITS

**Stuart Franklin Oberman**

**Technical Report: CSL-TR-96-711**

**December 1996**

This work was supported by NSF under contract MIP93-13701.

# DESIGN ISSUES IN HIGH PERFORMANCE FLOATING POINT ARITHMETIC UNITS

by
Stuart Franklin Oberman
**Technical Report: CSL-TR-96-711**
December 1996

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-9040


## Abstract

In recent years computer applications have increased in their computational complexity. The industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay particular attention to implementation of the floating point unit, or FPU. Special purpose applications, such as high performance graphics rendering systems, have placed further demands on processors. High speed floating point hardware is a requirement to meet these increasing demands. This work examines the state-of-the-art in FPU design and proposes techniques for improving the performance and the performance/area ratio of future FPUs.

In recent FPUs, emphasis has been placed on designing ever-faster adders and multipliers, with division receiving less attention. The design space of FP dividers is large, comprising five different classes of division algorithms: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. While division is an infrequent operation even in floating point intensive applications, it is shown that ignoring its implementation can result in system performance degradation. A high performance FPU requires a fast and efficient adder, multiplier, and divider.

The design question becomes how to best implement the FPU in order to maximize performance given the constraints of silicon die area. The system performance and area impact of functional unit latency is examined for varying instruction issue rates in the context of the SPECfp92 application suite. Performance implications are investigated for shared multiplication hardware, shared square root, on-the-fly rounding and conversion and fused functional units. Due to the importance of low latency FP addition, a variable latency FP addition algorithm has been developed which improves average addition latency by 33% while maintaining single-cycle throughput. To improve the performance and area of linear converging division algorithms, an automated process is proposed for minimizing the complexity of SRT tables. To reduce the average latency of quadratically-converging division algorithms, the technique of reciprocal caching is proposed, along with a method to reduce the latency penalty for exact rounding. A combination of the proposed techniques provides a basis for future high performance floating point units.


**Key Words and Phrases:** Addition, computer arithmetic, division, floating point, rounding, variable latency

# Acknowledgments

This work would not have been possible without the help of many wonderful people.

I would like to thank my advisor, Michael Flynn, for all of his technical, financial, and personal support during my years at Stanford. He believed in my abilities from my first day at Stanford, and he has been a source of encouragement ever since.

I would like to thank my other readers Mark Horowitz and Cal Quate for their efforts to review the drafts of this dissertation. Mark Horowitz provided constructive advice and acted as a reality-check for many of my ideas, improving both the content and clarity of this research. Thanks to Martin Morf for the many discussions yielding interesting ideas.

Thanks to Nhon Quach and Grant McFarland. I learned many of the "tricks" of fast floating point from Nhon Quach, and these have influenced the direction of this research. Grant McFarland was always there to discuss a new idea or to read a draft of a paper. Thanks to the other members of Professor Flynn's group for their comments and discussions: Kevin Nowka, Hesham Altwaijry, Dan Zucker, Kevin Rudd, and Steve Fu. Thanks also to the skeptics in the group who kept me honest and often understood the subjects better than I: Gary Bewick, Andrew Zimmerman, Brian Flachs, and John Johnson.

I extend sincere gratitude to my family. To my loving wife, Joy, who sacrificed a more comfortable living, helped support us, and ultimately helped drive me to the completion of this work. To my mother, father, and sister who have always been a tremendous source of encouragement, confidence, and love. This dissertation is dedicated to them.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

A floating point number representation can simultaneously provide a large range of numbers and a high degree of precision. As a result, a portion of modern microprocessors is often dedicated to hardware for floating point computation. Previously, silicon area constraints have limited the complexity of the floating point unit, or FPU. Advances in integrated circuit fabrication technology have resulted in both smaller feature sizes and increased die areas. Together, these trends have provided a larger transistor budget to the processor designer. It has therefore become possible to implement more sophisticated arithmetic algorithms to achieve higher FPU performance.

Due to the complexity of floating point number systems, hardware implementations of floating point operations are typically slower than integer operations. Many modern computer programs, such as scientific computation, 3D graphics applications, digital signal processing, and system performance benchmarks have a high frequency of floating point operations. The performance of these applications is often limited by the speed of the floating point hardware. For these reasons, high performance FPUs are now both practical and desirable.

The IEEE 754 floating point standard [1] is the most common floating point representation used in modern microprocessors. It dictates the precisions, accuracy, and arithmetic operations that must be implemented in conforming processors. The arithmetic operations include addition, multiplication, division, and square root. The design of a high performance IEEE conforming FPU requires a fast adder, multiplier, divider, and square root unit. This research therefore investigates algorithms and implementations for designing high performance IEEE conforming floating point units.

## 1.2 Design Space

The performance and area of a functional unit depend upon circuit style, logic implementation, and choice of algorithms. The space of current circuit styles ranges from fully-static CMOS designs to hand-optimized self-timed dynamic circuits. Logic design styles range from automatically-synthesized random logic to custom, hand-selected gates. The widest selection of design choices is available at the algorithmic level, which is the focus of this dissertation.

The three primary parameters in FP functional unit design are latency, cycle time, and area. The functional unit latency is the time required to complete a computation, typically measured in machine cycles. Designs can be either *Fixed Latency (FL)* or *Variable Latency (VL)*. In a FL design, each step of the computation completes in lock-step with a system clock. Further, any given operation completes after a fixed quantity of cycles. The cycle time in a FL design is the maximum time between the input of operands from registers and the latching of new results into the next set of registers. In contrast, VL designs complete after a variable quantity of cycles. This allows a result to be returned possibly sooner than the maximum latency, reducing the average latency. They achieve their variability through either the choice of algorithm (VLA) or choice of circuit design (VLC). VLA designs operate in synchronization with a system clock. However, the total number of cycles required to complete the operation varies depending upon other factors, such as the actual values of the input operands. An example of a non-floating point VLA design is a hierarchical memory system. In such a memory system, the total latency for a LOAD operation depends upon the level in the hierarchy at which the requested data resides. VLC designs need not have any internal synchronization with the rest of the system. Instead, such a design accepts new inputs at one time, and it produces results sometime later, independent of the system clock. A self-timed circuit is an example of a VLC design. Self-timed designs use special circuits to allow for results to be generated as soon as they are available. While self-timed designs reduce the average latency, they often introduce additional complexity, including additional testing requirements and integration issues with synchronous designs.

FL designs can be fully combinational or pipelined. Pipelining is a commonly used technique for increasing the throughput of functional units. A functional unit can be divided into smaller components by introducing explicit registers in-between the components. In this way, the cycle time of the unit becomes the maximum time for any of the components to complete. By increasing the number of components, the cycle time of the unit is decreased at the expense of increasing the latency. The primary motivation for pipelining is to allow for one operation to be initiated and another to be completed in each machine cycle, with more than one operation in progress at any time. As a result, the total latency for a sequence of operations is reduced by exploiting the component-level parallelism. However, the introduction of additional registers to hold the intermediate values contributes overhead in the form of longer cycle times and area. These tradeoffs must be understood for optimal FL and VLA functional unit design.

## 1.3 Arithmetic Operations

The most frequent FP operation is addition. Conceptually, it is the simplest operation, returning either the sum or difference of two FP numbers. In practice, FP adders can be slow due to additional aligning and normalizing shifts, recomplementation, and rounding that may be required. A fast FP adder is vital to the performance of an FPU, and thus techniques to reduce the latency of FP addition are investigated in this dissertation.

Multiplication is typically the next-most frequent FP operation. As a result, high-speed multiplication is also critical to a high performance FPU. Multiplication involves the summation of several shifted partial products, each of which is a product of the multiplicand and one digit of the multiplier. Thus, three steps are required in multiplication: partial product generation, partial product reduction, and final carry-propagate-addition. In practice, the partial products may be formed by direct ANDing of the multiplier digit and the multiplicand, or they may be formed by one of many *bit scanning* algorithms such as Booth's Algorithm [2]. The partial product reduction is implemented by a series of carry-free adders, which are connected in one of many different topologies ranging from linear arrays to logarithmic trees. The organization of these reduction trees has been the subject of previous research, much of which is summarized in [3], [4], [5], and [6]. The final carry-propagate-addition is an application of integer addition, a topic independent from FPU design [7]. The remaining tradeoffs in multiplier design are: method of partial product generation, topology and circuit design of the reduction tree, and the topology and circuit design of the final carry-propagate adder. These tradeoffs involve the analysis of the delay and area of the specific circuits and layouts in a given technology. As this dissertation focuses primarily on algorithmic tradeoffs in functional unit design, an analysis of the area and performance tradeoffs of FP multipliers is not presented here. Some of these tradeoffs are presented in Al-Twaijry [3].

The least frequently occurring operations in an IEEE FPU are division and square root. However, a slow divider implementation in a high performance FPU can result in performance degradation. The theory of square root computation is very similar to that of division, and thus much of the analysis for division can also be applied to square root units. As the design space of division algorithms and implementations is much larger than that of addition and multiplication, a considerable portion of this dissertation is dedicated to analysis and algorithms for divider design.

## 1.4 Organization

The following chapters detail design issues in FPU design, a VLA algorithm for higher performance FP addition, a taxonomy of division algorithms, and several FL and VL techniques for increasing FP division performance.

Chapter 2 examines the system performance impact of functional unit latencies. It also examines the performance implications of shared multiplication hardware, shared square root, on-the-fly rounding and conversion, and fused functional units. This chapter forms a basis for the FPU performance requirements from a system perspective, and it demonstrates the importance of a lower latency FP divider.

Chapter 3 presents a case study demonstrating the use of a variable latency functional unit to achieve maximum performance. The study proposes a new floating point addition algorithm which reduces the average addition latency while maintaining single cycle throughput. This algorithm achieves higher performance by exploiting the distribution of FP addition operands.

Chapter 4 presents five major classes of division algorithms in order to clarify the design space of division algorithms and implementations. It analyzes the fundamental design tradeoffs in terms of latency, cycle time, and area.

Chapter 5 analyzes in more detail techniques to minimize the complexity of quotient-digit selection tables in SRT division implementations. The quotient-digit selection function is often the major contributor to the cycle time of SRT dividers. By minimizing the delay and area of the quotient-digit selection tables, the performance and area of the overall divider implementation is improved.

Chapter 6 presents two variable latency methods for reducing average division latency: division and reciprocal caches.

Chapter 7 clarifies the methodology for correct IEEE compliant rounding for quadratically-converging division algorithms and proposes techniques to reduce the latency penalty for exact rounding.

Chapter 8 summarizes the results of this research and suggests further areas of floating point research.

# Chapter 2

# A System Perspective

## 2.1 Introduction

In recent FPUs emphasis has been placed on designing ever-faster adders and multipliers, with division and square root receiving less attention. The typical range for addition latency is 2 to 4 cycles, and the range for multiplication is 2 to 8 cycles. In contrast, the latency for double precision division ranges from 6 to 61 cycles, and square root is often far larger [8]. Most emphasis has been placed on improving the performance of addition and multiplication. As the performance gap widened between these two operations and division, floating point algorithms and applications have been slowly rewritten to account for this gap by mitigating the use of division. Thus, current applications and benchmarks are usually written assuming that division is an inherently slow operation and should be used sparingly.

This chapter investigates in detail the relationship between FP functional unit latencies and system performance. The application suites considered for this study included the NAS Parallel Benchmarks [9], the Perfect Benchmarks [10], and the SPECfp92 [11] benchmark suite. An initial analysis of the instruction distribution determined that the SPEC benchmarks had the highest frequency of floating point operations, and they were therefore chosen as the target workload of the study to best reflect the behavior of floating point intensive applications.

These applications are used to investigate several questions regarding the implementation of floating point units [12]:

- Does a high-latency division/square root operation cause enough system performance degradation to justify dedicated hardware support?
- How well can a compiler schedule code in order to maximize the distance between floating point result production and consumption?
- What are the effects of increasing the width of instruction issue on effective division latency?
- If hardware support for division and square root unit is warranted and a multiplication-based algorithm is utilized, should the FP multiplier hardware be shared, or should a dedicated functional unit be designed?
- Should square root share the division hardware?
- What operations most frequently consume division results?
- Is on-the-fly rounding and conversion necessary?

The organization of this chapter is as follows. Section 2 describes the method of obtaining data from the applications. Section 3 presents the results of the study. Section 4 analyzes and summarizes the results.

## 2.2 System Level Study

### 2.2.1 Instrumentation

System performance was evaluated using 11 applications from the SPECfp92 benchmark suite. The applications were each compiled on a DECstation 5000 using the MIPS C and Fortran compilers at each of three levels of optimization: no optimization, O2 optimization, and O3 optimization. O2 performs common subexpression elimination, code motion, strength reduction, code scheduling, and inlining of arithmetic statement functions. O3 performs all of O2's optimizations, but it also implements loop unrolling and other code-size increasing optimizations [13]. Among other things, varying the level of compiler optimization varies the total number of executed instructions and the distance between a division operation and the use of its result. The compilers utilized the MIPS R3000 machine model for all schedules assuming double precision FP latencies of 2 cycles for addition, 5 cycles for multiplication, and 19 cycles for division.

In most traditional computer architectures, a close match exists between high-level-language semantics and machine-level instructions for floating point operations [14]. Thus, the results obtained on a given architecture are applicable to a wide range of architectures. The results presented were obtained on the MIPS architecture, primarily due to the availability of the flexible program analysis tools pixie and pixstats [15]. Pixie reads an executable file and partitions the program into its basic blocks. It then writes a new version of the executable containing extra instructions to dynamically count the number of times each basic block is executed. The benchmarks use the standard input data sets, and each executes approximately 3 billion instructions. Pixstats is then used to extract performance statistics from the instrumented applications.

### 2.2.2 Method of Analysis

To determine the effects of a floating point operation on overall system performance, the performance degradation due to the operation needs to be determined. This degradation can be expressed in terms of excess CPI, or the CPI due to the result interlock. Excess CPI is a function of the dynamic frequency of the operation, the urgency of its results, and the functional unit latency. The dynamic frequency of an operation is the number of times that a particular operation is executed in the application. The urgency of a result is measured by how soon a subsequent instruction needs to consume the result. To quantify the urgency of results, interlock distances were measured for division results. The interlock distance is the distance between the production of a division result and its consumption by a subsequent instruction. It is clear that the dynamic frequency is solely a function of the application, urgency is a function of the application and the compiler, and functional unit latency depends upon the hardware implementation. The system designer has the most control over the functional unit latency. Through careful design of the processor architecture, though, the designer has some limited influence on the urgency. Adding extra registers and providing for out-of-order instruction execution are two means by which the system designer can influence urgency.

## 2.3 Results

### 2.3.1 Instruction Mix

Figure 2.1 shows the average frequency of division and square root operations in the benchmark suite relative to the total number of floating point operations, where the applications have been compiled using O3 optimization. This figure shows that simply in terms of dynamic frequency, division and square root seem to be relatively unimportant instructions, with about 3% of the dynamic floating point instruction count due to division and only 0.33% due to square root. The most common instructions are FP multiply and add, but note that add, subtract, move, and convert operations typically use the FP adder hardware. Thus, FP multiply accounts for 37% of the instructions, and the FP adder is used for 55% of the instructions. However, in terms of latency, division can play a much larger role. By assuming a machine model of a scalar processor, where every division operation has a latency of 20 cycles and the adder and multiplier each have a 3 cycle latency, a distribution of the excess CPI due to FP stall time was formed, shown in figure 2.2. The stall time is the period during which the processor was ready to execute the next instruction, but an interlock on the result of a previous unfinished FP add, multiply, or divide instruction prevented it from continuing. This excess CPI reduces overall performance by increasing the total CPI. Here, FP division accounts for 40% of the performance degradation, FP add accounts for 42%, and multiply accounts for the remaining 18%. Therefore, the performance of division is significant to the overall system performance.

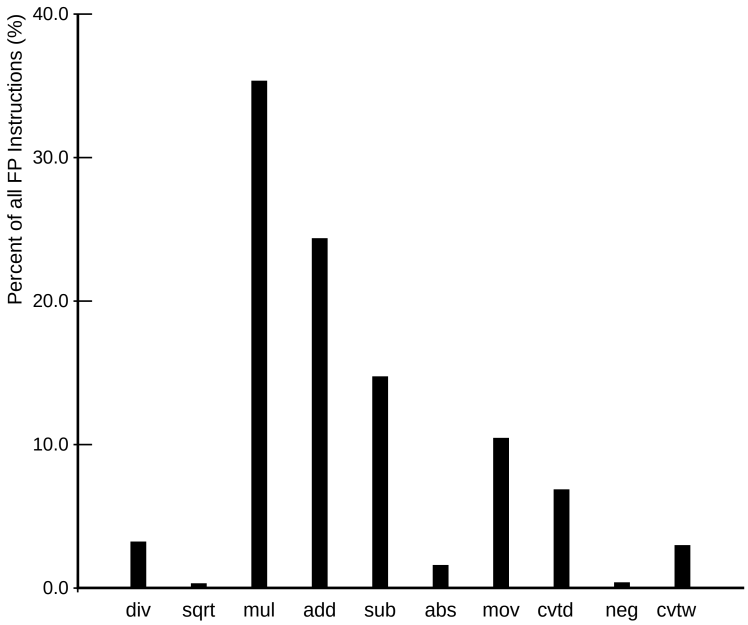


Figure 2.1: Instruction mix

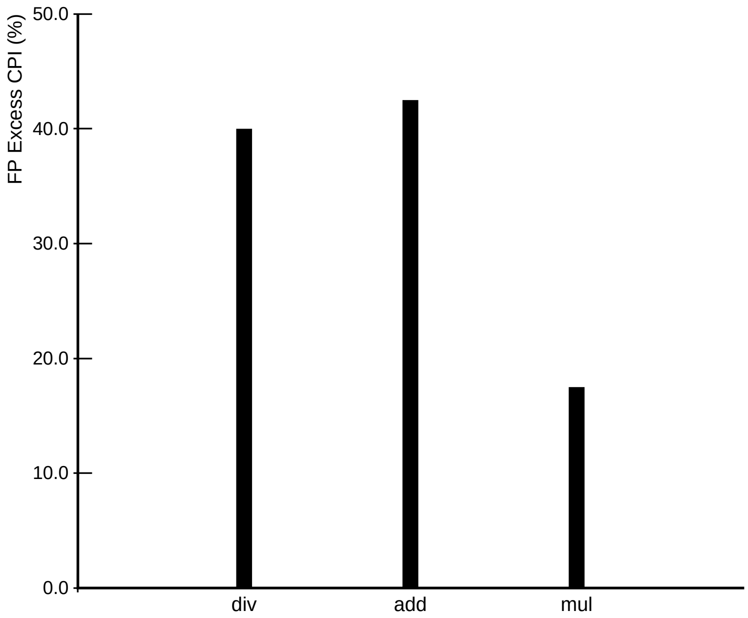


Figure 2.2: Functional unit stall time distribution

### 2.3.2 Compiler Effects

In order to analyze the impact that the compiler can have on improving system performance, the urgency of division results was measured as a function of compiler optimization level. Figure 2.3 shows a histogram of the interlock distances for division instructions at O0, as well as a graph of the cumulative interlock distance for the spice benchmark. Figure 2.4 shows the same data when compiled at O3. Figure 2.5 shows the average interlock distances for all of the applications at both O0 and O3 levels of optimization. By intelligent scheduling and loop unrolling, the compiler is able to expose instruction-level parallelism in the applications, decreasing the urgency of division results. Figure 2.5 shows that the average interlock distance can be increased by a factor of three by compiler optimization.

An average of the division interlock distances from all of the benchmarks was formed, weighted by division frequency in each benchmark. This result is also shown in figure 2.5 for the three levels of compiler optimization. In this graph, the curves represent the cumulative percentage of division instructions at each distance. The results from figure 2.5 show that the average interlock distance can be increased to only approximately 10 instructions. Even if the compiler assumed a larger latency, there is little parallelism left to exploit that could further increase the interlock distance and therefore reduce excess CPI. If the compiler scheduled assuming a low latency, the excess CPI could only increase for dividers with higher latencies than that for which the compiler scheduled. This is because the data shows the maximum parallelism available when scheduling for a latency of 19 cycles. If the compiler scheduled for a latency much less than 19 cycles, then it would not be as aggressive in its scheduling, and the interlock distances would be smaller, increasing urgency and therefore excess CPI. The results for division can be compared with those of addition and multiplication, shown in figure 2.6.

### 2.3.3 Performance and Area Tradeoffs

The excess CPI due to division is determined by summing all of the stall cycles due to division interlocks, which is the total penalty, and dividing this quantity by the

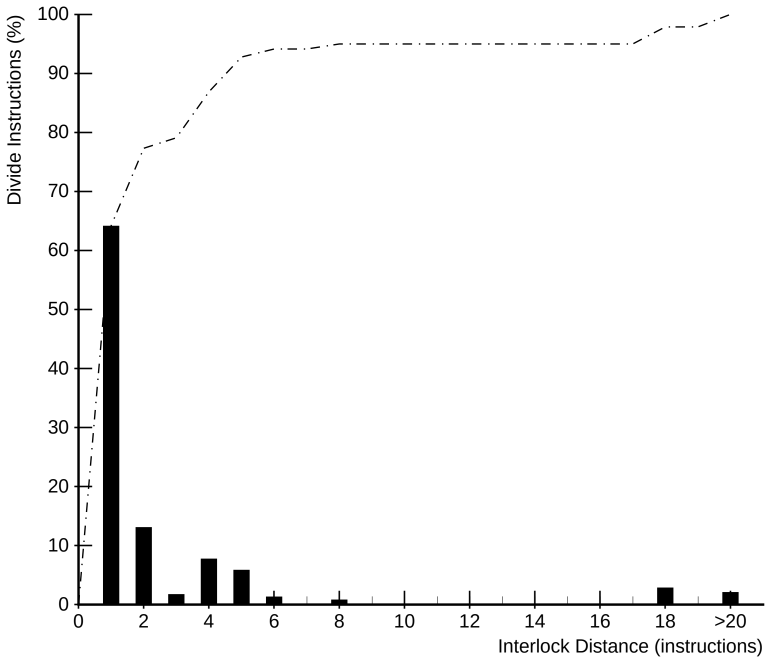


Figure 2.3: Spice with optimization O0

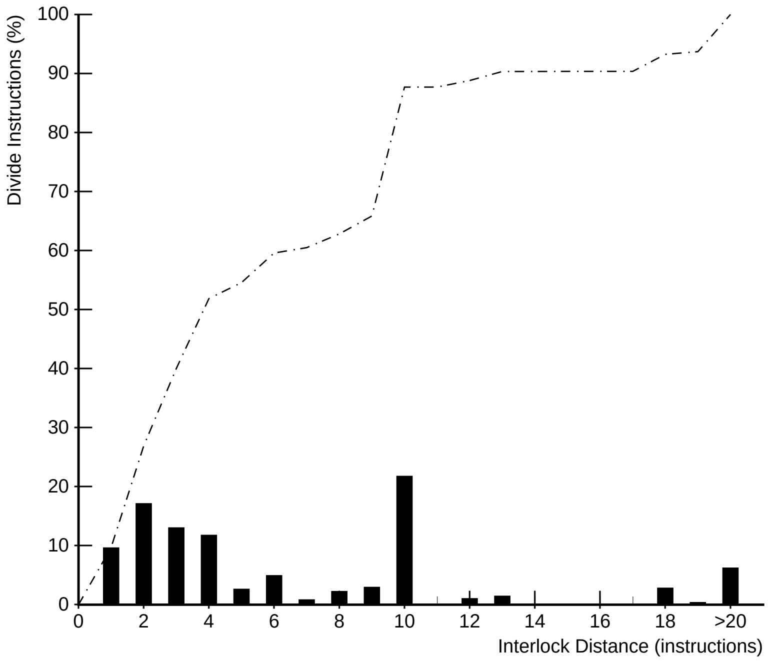


Figure 2.4: Spice with optimization O3

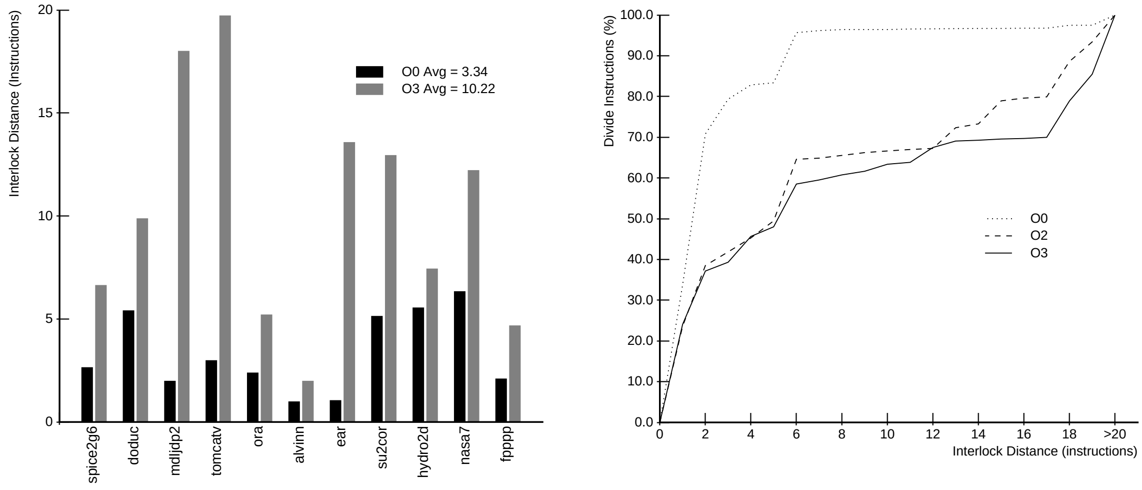


Figure 2.5: Interlock distances: by application and cumulative average

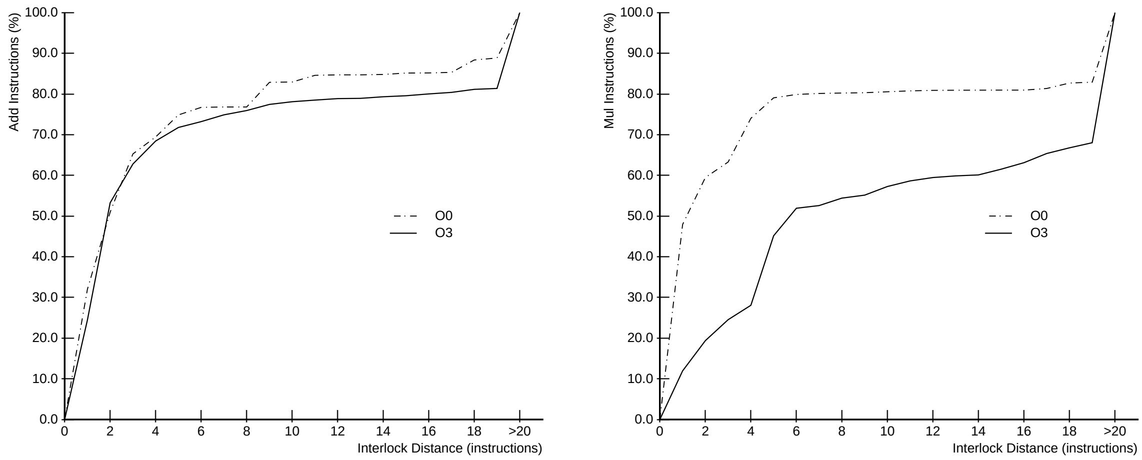


Figure 2.6: Cumulative average add and mul interlock distances

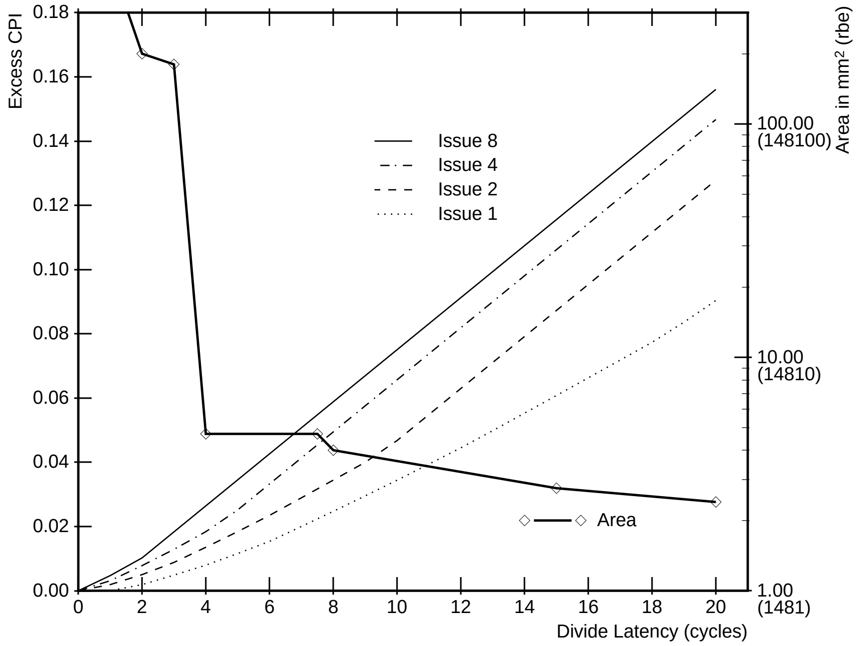


Figure 2.7: CPI and area vs division latency - low latency

total number of instructions executed, as shown in equation 2.1.

$$\mathrm{CPI}_{div} \quad = \quad \frac{\sum \text{stall time}}{\text{total instructions}} \tag{2.1}$$

The performance degradation due to division latency is displayed in figure 2.7. This graph shows how the excess CPI due to the division interlocks varies with division unit latency between 1 and 20 cycles for O3 optimization. Varying the optimization level also changed the total number of instructions executed, but left the number of division instructions executed constant. As a result, the fraction of division instructions is also a function of optimization level. While CPI due to division actually increases from O0 to O2, the overall performance at O2 and O3 increases because the total instruction count decreases. This effect is summarized in table 2.1, where the division latency is taken to be 20 cycles.

Figure 2.7 also shows the effect of increasing the number of instructions issued per cycle on excess CPI due to division. To determine the effect of varying instruction issue rate on excess CPI due to division, a model of an underlying architecture must

| Opt Level | Div Freq | Excess CPI |
|---|---|---|
| O0 | 0.33% | 0.057 |
| O2 | 0.76% | 0.093 |
| O3 | 0.79% | 0.091 |

Table 2.1: Effects of compiler optimization

be assumed. In this study, an optimal superscalar processor is assumed, such that the maximum issue rate is sustainable. This model simplifies the analysis while providing an upper bound on the performance degradation due to division. The issue rate is used to appropriately reduce the interlock distances. As the width of instruction issue increases, urgency of division data increases proportionally. In the worst case, every division result consumer could cause a stall equal to the functional unit latency. The excess CPI for the multiple issue processors is then calculated using the new interlock distances.

Figure 2.7 also shows how area increases as the functional unit latency decreases. The estimation of area is based on reported layouts from [16], [17], [18], all of which have been normalized to 1.0$\mu$m scalable CMOS layout rules. As division latencies decrease below 4 cycles, a large tradeoff must be made. Either a very large area penalty must be incurred to achieve this latency by utilizing a very high radix division algorithm, or large cycle times may result if an SRT divider is utilized. Several classes of division algorithms, including very high radix division, Newton-Raphson, and SRT division are examined in more detail in chapter 4.

In order to make the comparison of chip areas technology independent, the register bit equivalent (**rbe**) area model of Mulder [19] was used. In this model, one **rbe** equals the area of a one bit storage cell. For the purposes of this study, an **rbe** unit is referenced to a six-transistor static cell with high bandwidth, with an area of $675f^2$, where $f$ is the minimum feature size. The area required for 1 static RAM bit, as would be used used in an on-chip cache, is about 0.6 **rbe**. Since all areas are normalized to an $f = 1.0\mu m$ process, 1 $mm^2 = 1481$ **rbe**.

Figure 2.8 shows excess CPI versus division latency over a larger range of latencies.

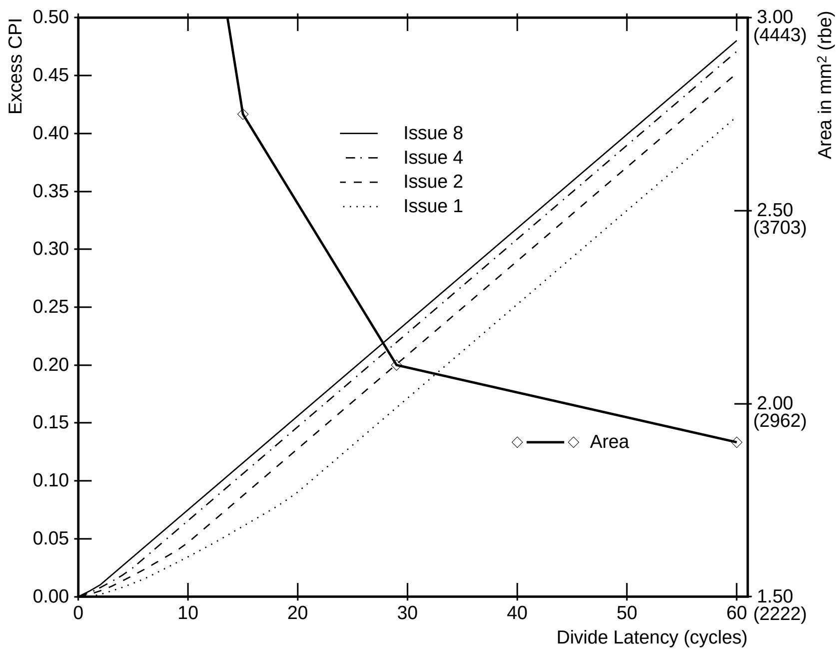


Figure 2.8: CPI and area vs division latency - full range

This graph can roughly be divided into five regions. Table 2.2 shows that inexpensive 1-bit SRT schemes use little area but can contribute in the worst case up to 0.50 CPI in wide-issue machines. Increasing the radix of SRT implementations involves an increase in area, but with a decrease in excess CPI. The region corresponding to 4-bit SRT schemes also represents the performance of typical multiplication-based division implementations, such as Newton-Raphson or series expansion [20]. The additional area required in such implementations is difficult to quantify, as implementations to date have shared existing multipliers, adding control hardware to allow for the shared functionality. At a minimum, such implementations require a starting approximation table that provides at least an 8 bit initial approximation. Such a table occupies a minimum of 2 Kbits, or 1230 **rbe**. The final region consists of very-high radix dividers of the form presented in [18] and [21]. To achieve this performance with CPI $< 0.01$, large area is required for very large look-up tables, often over 500,000 **rbe**.

To better understand the effects of division latency on the system performance of

| Divider Type | Latency (cycles) | Excess CPI | Area (rbe) |
|---|---|---|---|
| 1-Bit SRT | $> 40$ | $< 0.5$ | $< 3000$ |
| 2-Bit SRT | [20, 40] | [0.10, 0.32] | 3110 |
| 4-Bit SRT | [10, 20] | [0.04, 0.10] | 4070 |
| 8-Bit SRT and Self-Timed | [4, 10] | [0.01, 0.07] | 6665 |
| Very-High Radix | $< 4$ | $< 0.01$ | $> 100,000$ |

Table 2.2: Five regions of division latency

multiple issue processors, the excess CPI due to division can be expressed as a percentage of the base processor CPI. Figure 2.9 shows this relationship quantitatively. As instruction width increases, the degradation of system performance markedly increases. Not only does increasing the width of instruction issue reduce the average interlock distance, but the penalty for a division interlock relative to the processor issue rate dramatically increases. A slow divider in a wide-issue processor can easily reduce system performance by half.

### 2.3.4 Shared Multiplier Effects

If a multiplication-based division algorithm is chosen, such as Newton-Raphson or series expansion, it must be decided whether to use a dedicated multiplier or to share the existing multiplier hardware. The area of a well-designed 3 cycle FP multiplier is around 11 $mm^2$, again using the 1.0$\mu$m process. Adding this much area may not be always desirable. If an existing multiplier is shared, this has two effects. First, the latency through the multiplier likely increases due to the modifications necessary to support the division operation. Second, multiply operations may be stalled due to conflicts with division operations sharing the multiplier.

The effect of this on excess CPI is shown in figure 2.10. The results are based on an average of all of the applications when scheduled with O3. In all cases for a division latency less than 20 cycles, the excess CPI is less than 0.07. For reasonable

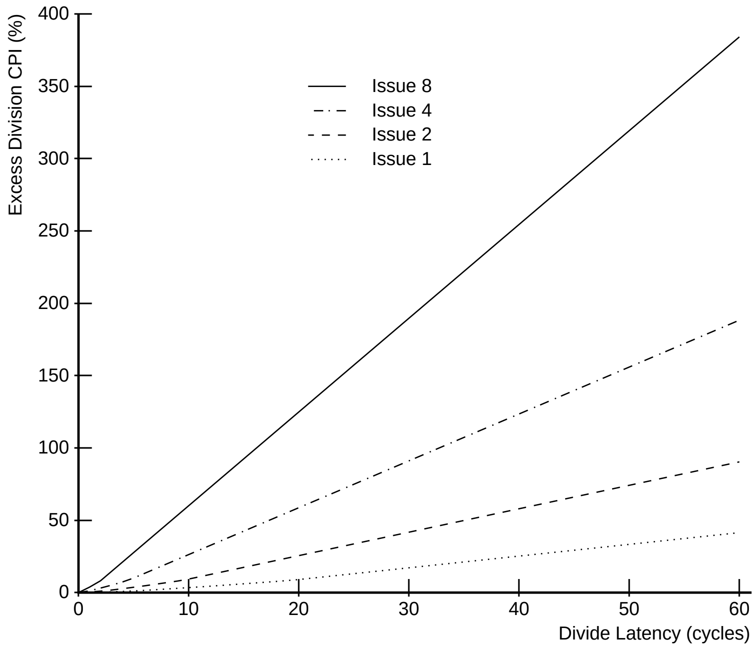


Figure 2.9: Excess CPI as a percentage of base CPI for multiple issue processors

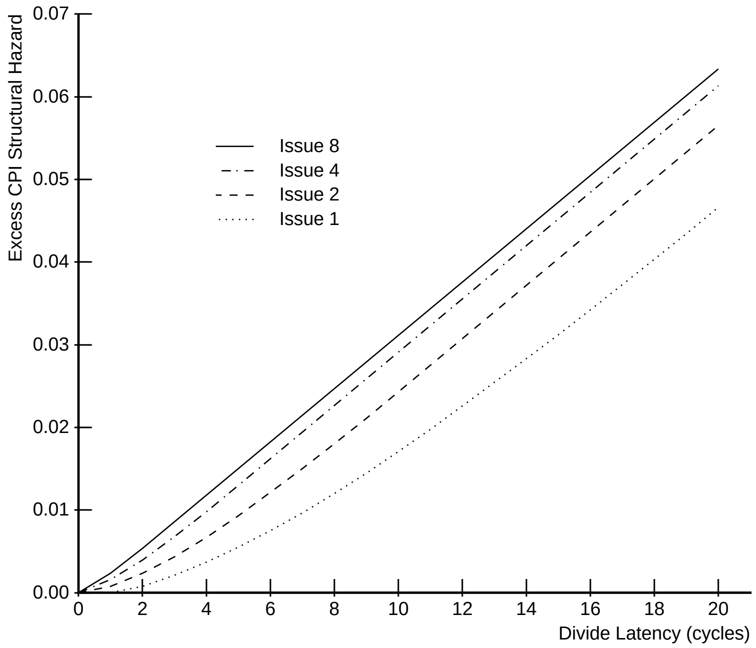


Figure 2.10: Excess CPI due to shared multiplier

implementations of multiplication-based division, with a latency of approximately 13 cycles, the actual penalty is $0.02 < \text{CPI} < 0.04$. For these applications, the penalty incurred for sharing an existing multiplier is not large due to the relatively low frequency of division operations. For special classes of applications, such as certain graphics applications, division and multiplication frequencies could be higher, requiring a separate division unit to achieve high performance.

### 2.3.5 Shared Square Root

The recurrence equation for square root is very close in form to that of division for both subtractive and multiplicative algorithms. Accordingly, division hardware can be implemented with additional functionality to perform square root computation. The design tradeoff then becomes whether a possible increase in hardware complexity and/or cycle time can be justified by an increase in overall performance.

The results of this study show that floating point square root on the average accounts for 0.087% of all executed instructions. This is a factor of 9.1 less than division. To avoid significant performance degradation due to square root, the latency of square root should be no worse than a factor 9.1 greater than division. However, any hardware implementation of square root more than likely meets this requirement. Even a simple 1 bit per iteration square root would contribute only 0.05 CPI for a scalar processor. Accordingly, these results suggest that the square root implementation does not need to have the same performance as the divider, and the sharing of division hardware is not crucial to achieving high system performance. Only if the additional area is small and the cycle time impact is negligible should division and square root share the same hardware.

### 2.3.6 On-the-fly Rounding and Conversion

In a nonrestoring division implementations such as SRT, an extra cycle is often required after the division operation completes. In SRT, the quotient is typically collected in a representation where the digits can take on both positive and negative values. Thus, at some point, all of the values must be combined and converted into

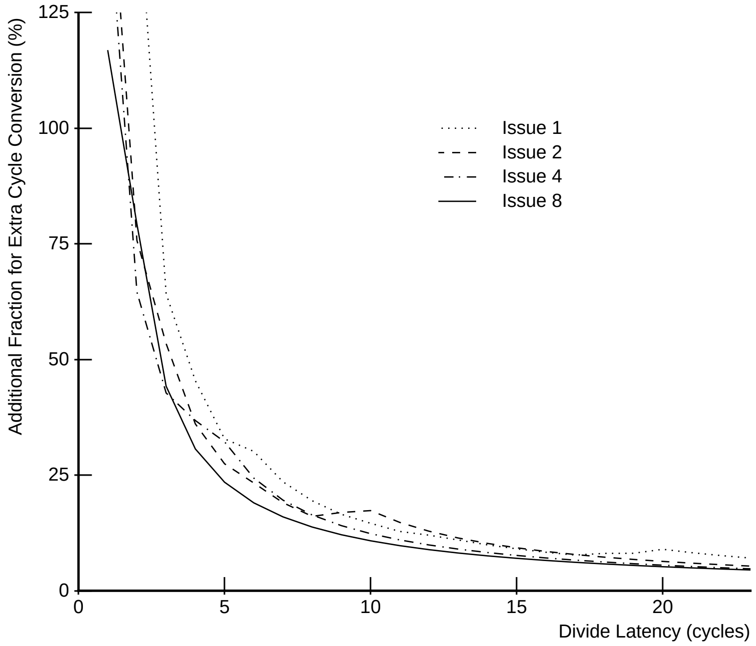


Figure 2.11: Effects of on-the-fly rounding and conversion

a standard representation. This requires a full-width addition, which can be a slow operation. To conform to the IEEE standard, it is necessary to round the result. This, too, can require a slow addition.

Techniques exist for performing this rounding and conversion "on-the-fly," and therefore the extra cycle may not be needed [22]. Because of the complexity of this scheme, the designer may not wish to add the additional required hardware. Figure 2.11 shows the performance impact of requiring an additional cycle after the division operation completes. For division latencies greater than 10 cycles, less than 20% of the total division penalty in CPI is due to the extra cycle. At very low division latencies, where the latency is less than or equal to 4 cycles, the penalty for requiring the additional cycle is obviously much larger, often greater than 50% of the total division penalty.

### 2.3.7 Consumers of Division Results

In order to reduce the effective penalty due to division, consider which operations actually use division results. Figure 2.12 is a histogram of instructions that consume division results.

This can be compared with the histogram for multiply results, shown in figure 2.13. For multiply results, the biggest users are multiply and add instructions. Since both *add.d* and *sub.d* use the FP adder, the FP adder is the consumer for nearly 50% of the multiply results. Accordingly, fused operations such as multiply-accumulate are useful. Because the multiply-add pattern occurs frequently in such applications and it does not require much more hardware than the separate functional units, fused multiply-adders are often used in modern processors.

Looking at the consumers of division results, the FP adder is the largest consumer with 27% of the results. The second biggest consumer is the store operation with 23% of the results. It is possible to overcome the penalties due to a division-store interlock, though, with other architectural implementations. A limited number of registers can require a division result to be spilled to memory through a store instruction. By either adding registers or register renaming, it may be possible to reduce the urgency due to store.

While the percentage of division results that the adder consumes is not as high as for multiply results, it is still the largest user. A designer could consider the implementation of a fused divide-add instruction to increase performance. In division implementations where on-the-fly conversion and rounding is not used, an extra addition cycle exists for this purpose. It may be possible to make this a three-way addition, with the third operand coming from a subsequent add instruction. Because this operand is known soon after the instruction is decoded, it can be sent to the the three-way adder immediately. Thus, a fused divide-add unit could provide additional performance.

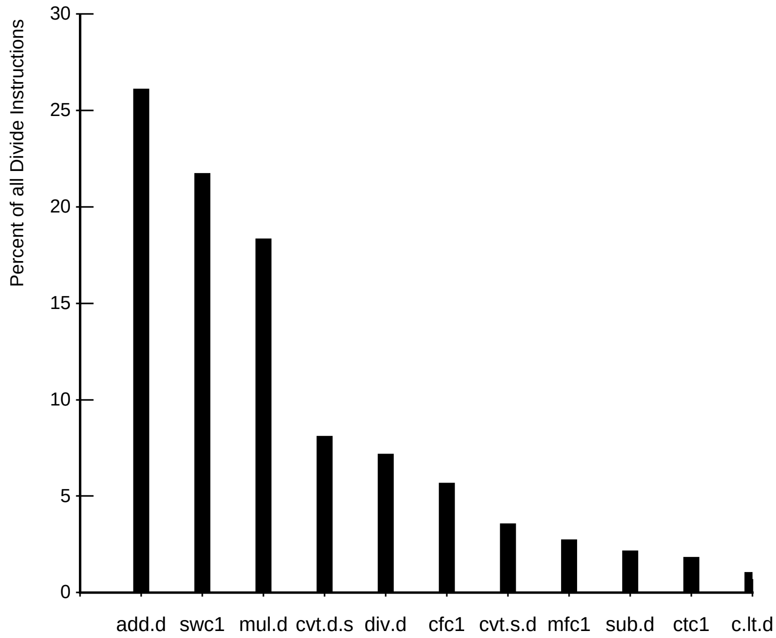


Figure 2.12: Consumers of division results

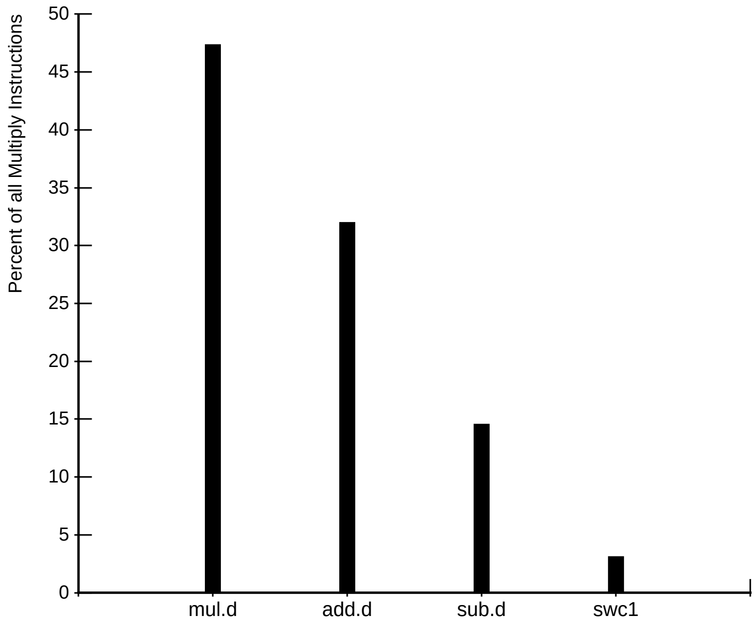


Figure 2.13: Consumers of multiply results

## 2.4 Summary

This chapter has investigated the issues of designing an FP divider in the context of an entire system. The frequency and interlock distance of division instructions in SPECfp92 benchmarks have been determined, along with other useful measurements, in order to answer several questions regarding the implementation of a floating point divider.

The data shows that for the slowest hardware divider, with a latency greater than 60 cycles, the CPI penalty can reach 0.50. To achieve good system performance, some form of hardware division is required. However, at very low divider latencies, two problems arise. The area required increases exponentially or cycle time becomes impractical. There is a knee in the area/performance curve near 10 cycles. Dividers with lower latencies do not provide significant system performance benefits, and their areas are too large to be justified.

The compiler can decrease the urgency of division results. Most of the performance gain is in performing basic compiler optimizations, at the level of O2. Only marginal improvement is gained by further optimization. The average interlock distance increases by a factor of three by using compiler optimization. Accordingly, for scalar processors, a division latency of 10 cycles or less can be tolerated.

Increasing the number of instructions issued per cycle also increases the urgency of division results. Increasing the number of instructions issued per cycle to 2 causes a 38% increase in excess CPI, increasing to 4 causes a 94% increase in excess CPI, and increasing to 8 causes a 120% increase in excess CPI. Further, as the width of instruction issue increases, the excess CPI due to division increases even faster. Wide issue machines utilize the instruction-level parallelism in applications by issuing multiple instructions every cycle. While this has the effect of decreasing the base CPI of the processor, it exposes the functional unit latencies to a greater degree and accentuates the effects of slow functional units.

In most situations, an existing FP multiplier can be shared when using a multiplication based division algorithm. The results show that for a division latency of around 13 cycles, the CPI penalty is between 0.025 and 0.040. While the CPI penalty is low when the multiplier is shared and modified to also perform division, the designer must also consider effects on the multiplier which could have an impact on cycle time.

On-the-fly rounding and conversion is not essential for all division implementations. For division latencies greater than 10 cycles, the lack of on-the-fly rounding and conversion does not account for a significant fraction of the excess CPI, and, as a result, is not required. However, for very high performance implementations where the area and complexity are already large, this method is a practical means of further reducing division latency.

Addition and store operations are the most common consumers of division results. Accordingly, the design of a fused divide-add unit is one means of achieving additional system performance.

While division is typically an infrequent operation even in floating point intensive applications, ignoring its implementation can result in system performance degradation.

# Chapter 3

# Faster FP Addition

## 3.1 Introduction

Chapter 2 shows that the most frequent FP operations are addition and subtraction, and together they account for over half of the total FP operations in typical scientific applications. Both addition and subtraction use the FP adder. Techniques to reduce the latency and increase the throughput of the FP adder have therefore been the subject of much previous research.

Due to its many serial component operations, FP addition can have a longer latency than FP multiplication. Pipelining is a commonly used method to increase the throughput of the adder, but it does not reduce the latency. Previous research has provided algorithms to reduce the latency by performing some of the operations in parallel. This parallelism is achieved at the cost of additional hardware. The minimum achievable latency using such algorithms in high clock-rate microprocessors has been three cycles, with a throughput of one cycle.

To further reduce the latency, we observe that not all of the components are needed for all input operands. Two VLA techniques are proposed to take advantage of this to reduce the average addition latency [23]. To effectively use average latency, the processor must be able to exploit a variable latency functional unit. The processor might use some form of dynamic instruction scheduling with out-of-order execution in order to use the reduced latency and achieve maximum system performance.

## 3.2 FP Addition Algorithms

FP addition comprises several individual operations. Higher performance is achieved by reducing the maximum number of serial operations in the critical path of the algorithm. The following sections summarize the results of previous research in the evolution of high-performance floating-point addition algorithms. Throughout this study, the analysis assumes IEEE double precision operands (a 64 bit word, comprising a 1 bit sign, an 11 bit biased exponent, and a 52 bit significand, with one hidden significand bit [1]).

### 3.2.1 Basic

The straightforward addition algorithm *Basic* requires the most serial operations. It has the following steps [24]:

1. Exponent subtraction: Perform subtraction of the exponents to form the absolute difference $|E_a - E_b| = d$.

2. Alignment: Right shift the significand of the smaller operand by $d$ bits. The larger exponent is denoted $E_f$.

3. Significand addition: Perform addition or subtraction according to the effective operation, which is a function of the opcode and the signs of the operands.

4. Conversion: Convert the significand result, when negative, to a sign-magnitude representation. The conversion requires a two's complement operation, including an addition step.

5. Leading-one detection: Determine the amount of left shift needed in the case of subtraction yielding cancellation. For addition, determine whether or not a 1 bit right is required. Priority encode (PENC) the result to drive the normalizing shifter.

6. Normalization: Normalize the significand and update $E_f$ appropriately.

7. Rounding: Round the final result by conditionally adding 1 ulp as required by the IEEE standard. If rounding causes an overflow, perform a 1 bit right shift and increment $E_f$.

The latency of this algorithm is large, due to its many long length components. It contains two full-length shifts, in steps 2 and 6. It also contains three full-length significand additions, in steps 3, 4 and 7.

### 3.2.2 Two-Path

Several improvements can be made to *Basic* in order to reduce its total latency. These improvements come typically at the cost of adding additional hardware. These improvements are based on noting certain characteristics of FP addition/subtraction computation:

1. The sign of the exponent difference determines which of the two operands is larger. By swapping the operands such that the smaller operand is always subtracted from the larger operand, the conversion in step 4 is eliminated in all cases except for equal exponents. In the case of equal exponents, it is possible that the result of step 3 may be negative. Only in this event could a conversion step be required. Because there would be no initial aligning shift, the result after subtraction would be exact and there will be no rounding. Thus, the conversion addition in step 4 and the rounding addition in step 7 become mutually exclusive by appropriately swapping the operands. This eliminates one of the three carry-propagate addition delays.

2. In the case of effective addition, there is never any cancellation of the results. Accordingly, only one full-length shift, an initial aligning shift, can ever be needed. For subtraction, two cases need to be distinguished. First, when the exponent difference $d > 1$, a full-length aligning shift may be needed. However, the result never requires more than a 1 bit left shift. Similarly if $d \leq 1$, no full-length aligning shift is necessary, but a full-length normalizing shift may be required in the case of subtraction. In this case, the 1 bit aligning shift and the

conditional swap can be predicted from the low-order two bits of the exponents, reducing the latency of this path. Thus, the full-length alignment shift and the full-length normalizing shift are mutually exclusive, and only one such shift need ever appear on the critical path. These two cases can be denoted *CLOSE* for $d \leq 1$, and *FAR* for $d > 1$, where each path comprises only one full-length shift [25].

3. Rather than using leading-one-detection after the completion of the significand addition, it is possible to predict the number of leading zeros in the result directly from the input operands. This leading-one-prediction (LOP) can therefore proceed in parallel with the significand addition using specialized hardware [26],[27].

An improved adder takes advantage of these three cases. It implements the significand datapath in two parts: the *CLOSE* path and *FAR* path. At a minimum, the cost for this added performance is an additional significand adder and a multiplexor to select between the two paths for the final result. Adders based on this algorithm have been used in several commercial designs [28],[29],[30]. A block diagram of the improved *Two Path* algorithm is shown in figure 3.1.

### 3.2.3 Pipelining

To increase the throughput of the adder, a standard technique is to pipeline the unit such that each pipeline stage comprises the smallest possible atomic operation. While an FP addition may require several cycles to return a result, a new operation can begin each cycle, providing maximum throughput. Figure 3.1 shows how the adder is typically divided in a pipelined implementation. It is clear that this algorithm fits well into a four cycle pipeline for a high-speed processor with a cycle time between 10 and 20 gates. The limiting factors on the cycle time are the delay of the significand adder (SigAdd) in the second and third stages, and the delay of the final stage to select the true result and drive it onto a result bus. The first stage has the least amount of computation; the *FAR* path has the delay of at least one 11 bit adder and two multiplexors, while the *CLOSE* path has only the delay of the 2 bit exponent

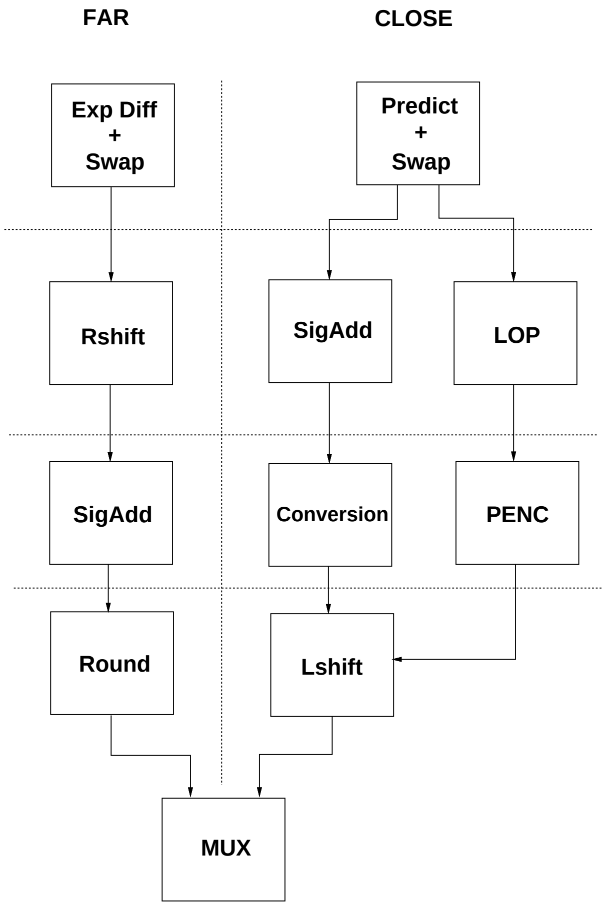


Figure 3.1: Two path algorithm

prediction logic and one multiplexor. Due to the large atomic operations in the second stage, the full-length shifter and significand adder, it is unlikely that the two stages can be merged, requiring four distinct pipeline stages.

When the cycle time of the processor is significantly larger than that required for the FP adder, it is possible to combine pipeline stages, reducing the overall latency in machine cycles but leaving the latency in time relatively constant. Commercial superscalar processors, such as Sun UltraSparc [31], often have larger cycle times, resulting in a reduced FP addition latency in machine cycles when using the *Two Path* algorithm. In contrast, superpipelined processors, such as DEC Alpha [32], have shorter cycle times and have at least a four cycle FP addition latency. For the rest of this study, it is assumed that the FP adder cycle time is limited to contain only one large atomic operation, such that the pipelined implementation of *Two Path* requires four stages.

### 3.2.4 Combined Rounding

A further optimization can be made to the *Two Path* algorithm to reduce the number of serial operations. This optimization is based upon the realization that the rounding step occurs very late in the computation, and it only modifies the result by a small amount. By precomputing all possible required results in advance, rounding and conversion can be reduced to the selection of the correct result, as described by Quach [33],[34]. Specifically, for the IEEE *round to nearest* (RN) rounding mode, the computation of $A + B$ and $A + B + 1$ is sufficient to account for all possible rounding and conversion possibilities. Incorporating this optimization into *Two Path* requires that each significand adder compute both *sum* and *sum+1*, typically through the use of a compound adder (ComAdd). A compound adder is a special adder that computes *sum* and *sum+1* while sharing some internal hardware to reduce the size. Selection of the true result is accomplished by analyzing the rounding bits, and then selecting either of the two results. The rounding bits are the sign, LSB, guard, and sticky bits. This optimization removes one significand addition step. For pipelined implementations, this can reduce the number of pipeline stages from four to three. The cost of this improvement is that the significand adders in both paths must be modified to produce both *sum* and *sum+1*.

For the two directed IEEE rounding modes *round to positive* and *minus infinity* (RP and RM), it is also necessary to compute $A + B + 2$. The rounding addition of 1 ulp may cause an overflow, requiring a 1 bit normalizing right-shift. This is not a problem in the case of RN, as the guard bit must be 1 for rounding to be required. Accordingly, the addition of 1 ulp will be added to the guard bit, causing a carry-out into the next most significant bit which, after normalization, is the LSB. However, for the directed rounding modes, the guard bit need not be 1. Thus, the explicit addition *sum+2* is required for correct rounding in the event of overflow requiring a 1 bit normalizing right shift. In [33], it is proposed to use a row of half-adders above the

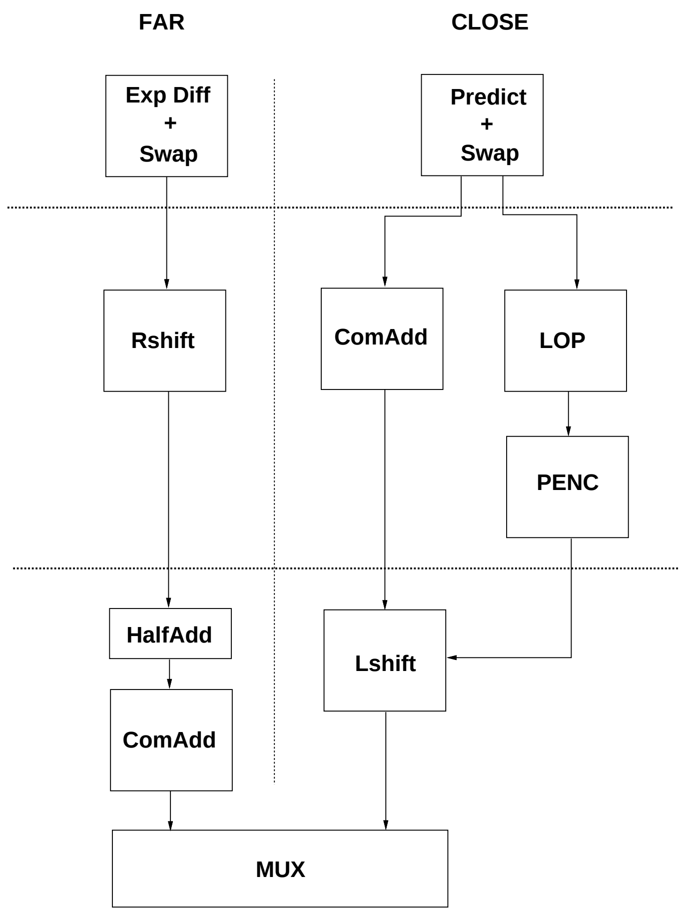


Figure 3.2: Three cycle pipelined adder with combined rounding

*FAR* path significand adder. These adders allow for the conditional pre-addition of the additional ulp to produce *sum+2*. In the Intel i860 floating-point adder [35],[36], an additional significand adder is used in the third stage. One adder computes *sum* or *sum+1* assuming that there is no carry-out. The additional adder computes the same results assuming that a carry-out occurs. This method is faster than Quach [33], as it does not introduce any additional delay into the critical path. However, it requires duplication of the entire significand adder in the third stage. A block diagram of the three cycle *Combined Rounding* algorithm based on Quach is shown in figure 3.2. The critical path in this implementation is in the third stage consisting of the delays of the half-adder, compound adder, multiplexor, and drivers.

## 3.3 Variable Latency Algorithm

From figure 3.2, the long latency operation in the first cycle occurs in the *FAR* path. It contains hardware to compute the absolute difference of two exponents and to conditionally swap the exponents. Depending upon the FP representation used within the FPU, the exponents are either 11 bits for IEEE double precision or 15 bits for extended precision. As previously stated, the minimum latency in this path comprises the delay of an 11 bit adder and two multiplexors. The *CLOSE* path, in contrast, has relatively little computation. A few gates are required to inspect the low-order 2 bits of the exponents to determine whether or not to swap the operands, and a multiplexor is required to perform the swap. Thus, the *CLOSE* path is faster than the *FAR* path by a minimum of

$$\Delta t_d \geq t_{mux} + (t_{add11} - t_{2bit})$$

### 3.3.1 Two Cycle

Rather than letting the *CLOSE* path hardware sit idle during the first cycle, it is possible to take advantage of the duplicated hardware and initiate *CLOSE* path computation one cycle earlier. This is accomplished by moving both the second and third stage *CLOSE* path hardware up to their preceding stages. Since the first stage in the *CLOSE* path completes very early relative to the *FAR* path, the addition of the second stage hardware need not result in an increase in cycle time. For example, in the implementation of the DEC Alpha 21164 FP adder [37], the first cycle of the *CLOSE* path includes exponent prediction and swapping logic along with the significand carry-propagate-adder. In contrast, the first cycle of the *FAR* path contains only the exponent difference hardware and swapping logic. However, the DEC adder requires a constant 4 cycles for both *CLOSE* and *FAR* paths, 3 cycles to compute the result and 1 cycle for driving the result out of the functional unit.

The operation of the proposed algorithm is as follows. Both paths begin speculative execution in the first cycle. At the end of the first cycle, the true exponent difference is known from the *FAR* path. If the exponent difference dictates that the

*FAR* path is the correct path, then computation continues in that path for two more cycles, for a total latency of three cycles. However, if the *CLOSE* path is chosen, then computation continues for one more cycle, with the result available after a total of two cycles. While the maximum latency of the adder remains three cycles, the average latency is reduced due to the faster *CLOSE* path. If the *CLOSE* path is a frequent path, then a considerable reduction in the average latency can be achieved. A block diagram of the *Two Cycle* algorithm is shown in figure 3.3.

Since a result can be placed on the output bus in either stage 2 or stage 3, some logic is required to control the tri-state buffer in the second stage to ensure that it only gates a result when there is no result to be gated in stage 3. In the case of a collision with a pending result in stage 3, the stage 2 result is simply piped into stage 3. While this has the effect of increasing the *CLOSE* path latency to three cycles in these instances, it does not affect throughput. As only a single operation is initiated every cycle, it is possible to retire a result every cycle.

The frequency of collisions depends upon the actual processor micro-architecture as well as the program. Worst case collisions would result from a stream of consecutive addition operations which alternate in their usage of the *CLOSE* and *FAR* paths. The distance between consecutive operations depends upon the issue-width of the processor and the number of functional units.

Scheduling the use of the results of an adder implementing *Two Cycle* is not complicated. At the end of the first cycle, the *FAR* path hardware has determined the true exponent difference, and thus the correct path is known. Therefore, a signal can be generated at that time to inform the scheduler whether the result is available at the end of one more cycle or two more cycles. In certain cases, one cycle may be sufficient to allow for the proper scheduling of a result in a dynamically-scheduled processor.

### 3.3.2 One Cycle

Further reductions in the latency of the *CLOSE* path can be made after certain observations. First, the normalizing left shift in the second cycle is not required for

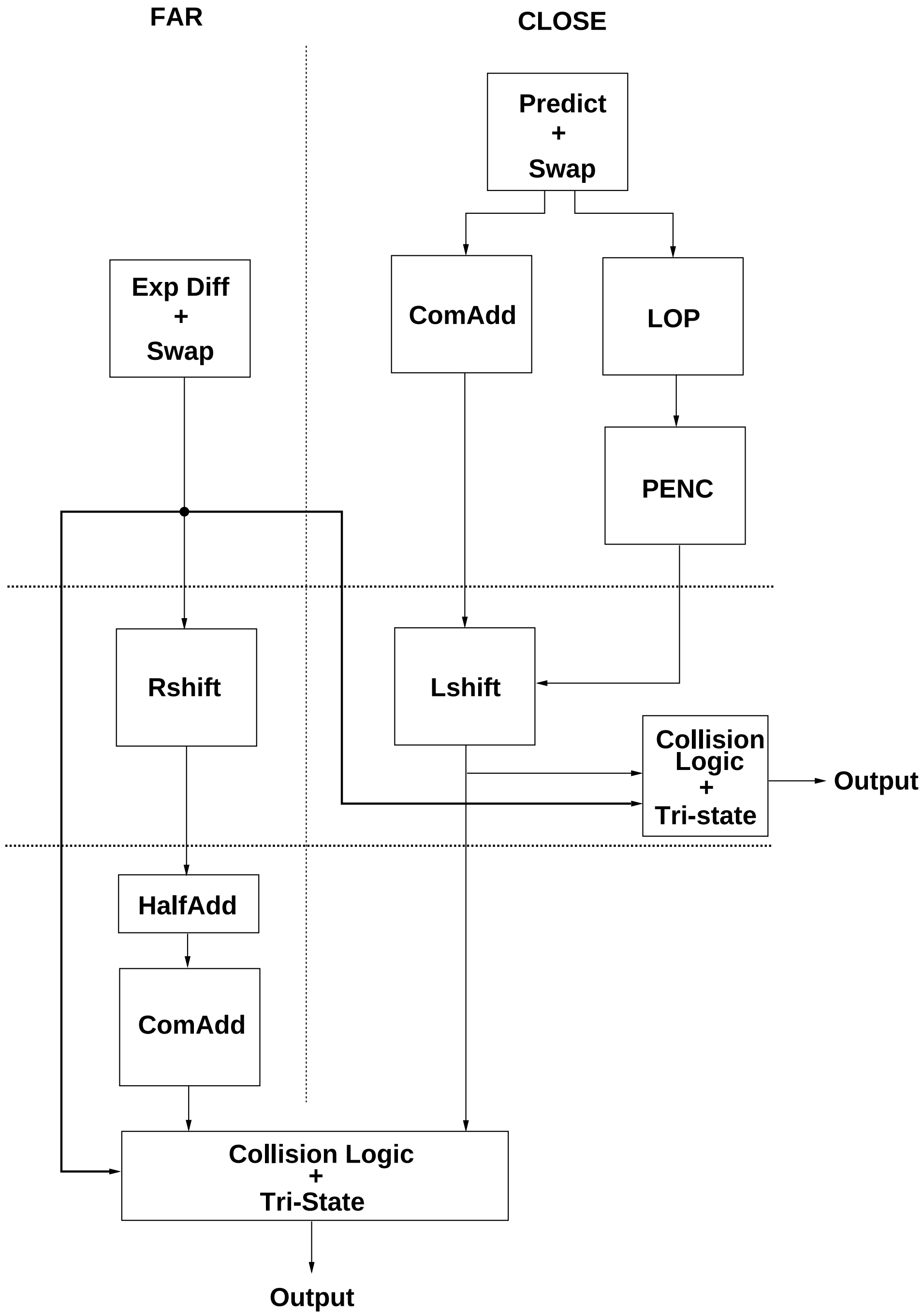


Figure 3.3: Two or three cycle variable latency adder

all operations. A normalizing left shift can only be required if the effective operation is subtraction. Since additions never need a left shift, addition operations in the *CLOSE* path can complete in the first cycle. Second, in the case of effective subtractions, small normalizing shifts, such as $d \leq 2$, can be separated from longer shifts. While longer shifts still require the second cycle to pass through the full-length shifter, short shifts can be completed in the first cycle through the addition of a separate small multiplexor. Both of these cases have a latency of only one cycle, with little or no impact on cycle time. If these cases occur frequently, the average latency is reduced. A block diagram of this adder is shown in figure 3.4.

The *One Cycle* algorithm allows a result to be gated onto the output bus in any of the three stages. As in the *Two Cycle* algorithm, additional control for the tri-state buffers is required to ensure that only one result is gated onto the bus in any cycle. In the case of a collision with a pending result in any of the other two stages, the earlier results are simply piped into their subsequent stages. This guarantees the correct FIFO ordering on the results. While the average latency may increase due to collisions, throughput is not affected.

Scheduling the use of the results from a *One Cycle* adder is somewhat more complicated than for *Two Cycle*. In general, the instruction scheduling hardware needs some advance notice to schedule the use of a result for another functional unit. It may not be sufficient for this notice to arrive at the same time as the data. Thus, an additional mechanism may be required to determine as soon as possible before the end of the first cycle whether the result will complete either 1) in the first cycle or 2) the second or third cycles. A proposed method is as follows. First, quickly determine whether the correct path is the *CLOSE* or *FAR* path from the absolute difference of the exponents. If all bits of the difference except for the LSB are 0, then the absolute difference is either 0 or 1 depending upon the LSB, and the correct path is the *CLOSE* path. To detect this situation fast, an additional small leading-one-predictor is used in parallel with the exponent adder in the *FAR* path to generate a CLOSE/FAR signal. This signal is very fast, as it does not depend on exactly where the leading one is, only if it is in a position greater than the LSB.

Predicting in the first cycle whether or not a *CLOSE* path operation can complete

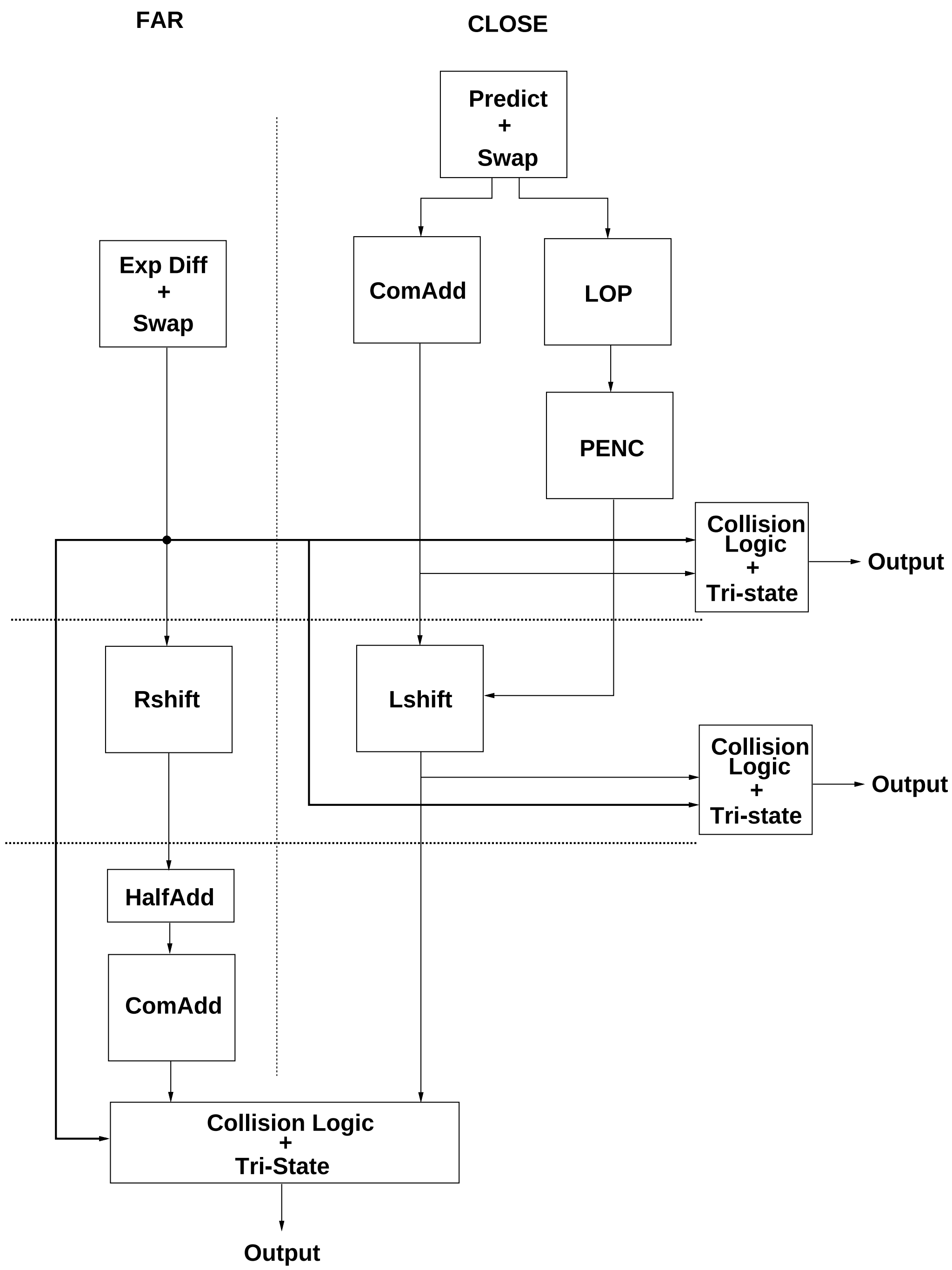


Figure 3.4: One, two, or three cycle variable latency adder

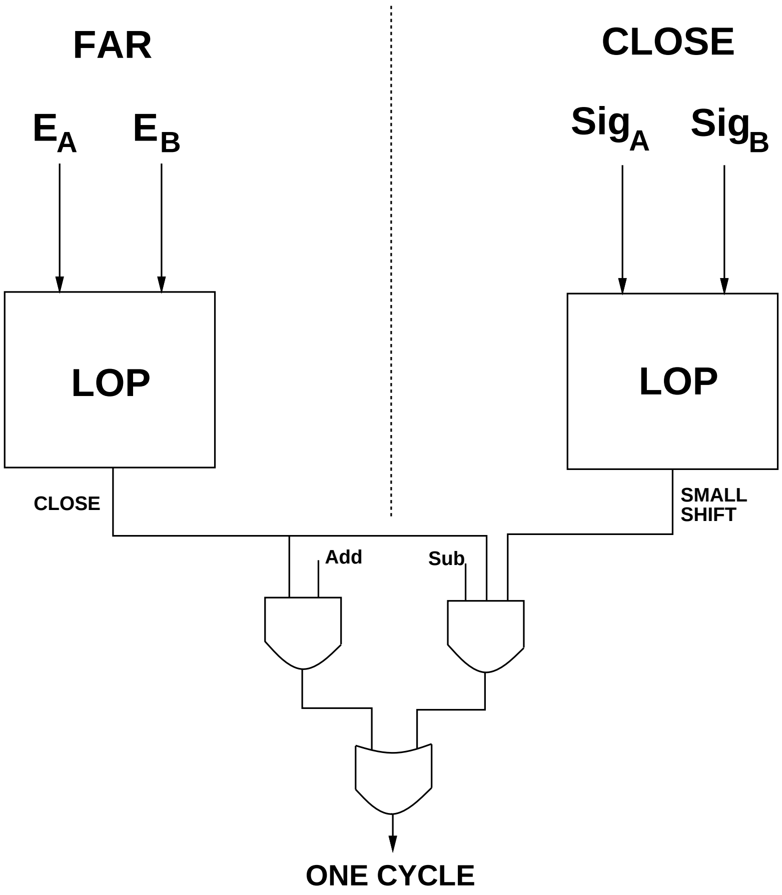


Figure 3.5: Additional hardware for one cycle operation prediction

in one or two cycles may require additional hardware. Effective additions require no other information than the CLOSE/FAR signal, as all *CLOSE* path effective additions can complete in the first cycle. In the case of effective subtractions, an additional specialized leading-one-predictor can be included in the significand portion of the *CLOSE* path to predict quickly whether the leading one is in any of the high order three bits. If it is in these bits, then it generates a one cycle signal; otherwise, it generates a two cycle signal. A block diagram of the additional hardware required for early prediction of one cycle operations is shown in figure 3.5. An implementation of this early prediction hardware should produce a one cycle signal in less than 8 gate delays, or about half a cycle.

## 3.4 Performance Results

To demonstrate the effectiveness of these two algorithms in reducing the average latency, the algorithms were simulated using operands from actual applications. The data for the study was acquired using the ATOM instrumentation system [38]. ATOM was used to instrument 10 applications from the SPECfp92 [11] benchmark suite. These applications were then executed on a DEC Alpha 3000/500 workstation. The benchmarks used the standard input data sets, and each executed approximately 3 billion instructions. All double precision floating-point addition and subtraction operations were instrumented. The operands from each operation were used as input to a custom FP adder simulator. The simulator recorded the effective operation, exponent difference, and normalizing distance for each set of operands.

Figure 3.6 is a histogram of the exponent differences for the observed operands, and it also is a graph of the cumulative frequency of operations for each exponent difference. This figure shows the distribution of the lengths of the initial aligning shifts. Note that 57% of the operations are in the *FAR* path with $E_d > 1$, while 43% are in the *CLOSE* path. A comparison with a different study of floating point addition operands [39] on a much different architecture using different applications provides validation for these results. In that study over 30 years ago, six problems were traced on an IBM 704, tracking the aligning and normalizing shift distances. There 45% of the operands required aligning right shifts of 0 or 1 bit, while 55% required more than a 1 bit right shift. The similarity in the results suggests a fundamental distribution of floating point addition operands in scientific applications.

An implementation of the *Two Cycle* algorithm utilizes the two cycle path 43% of the time with a performance of:

$$\begin{aligned} \text{Average Latency} &= 3 \times (.57) + 2 \times (.43) = 2.57 \text{ cycles} \\ \text{Speedup} &= 3/2.57 = 1.17 \end{aligned}$$

Thus, an implementation of the *Two Cycle* algorithm has a speedup in average addition latency of 1.17, with little or no effect on cycle time.

Implementations of the *One Cycle* algorithm reduce the average latency even further. An analysis of the effective operations in the *CLOSE* path shows that the total

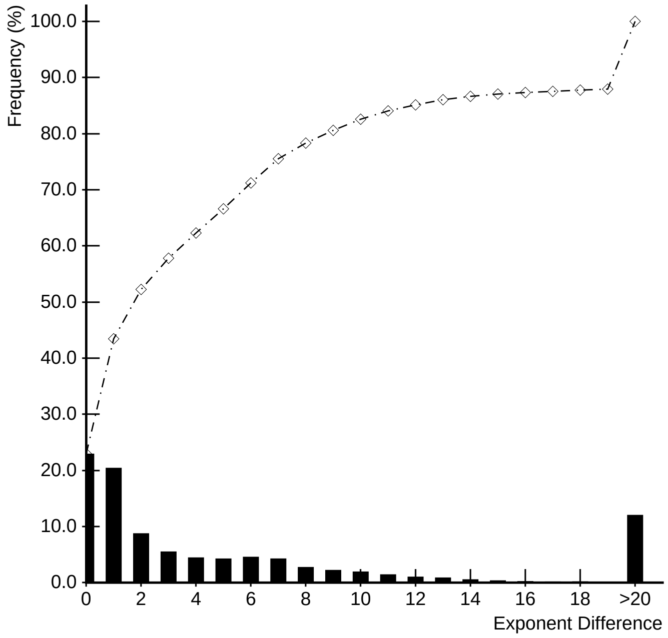


Figure 3.6: Histogram of exponent difference

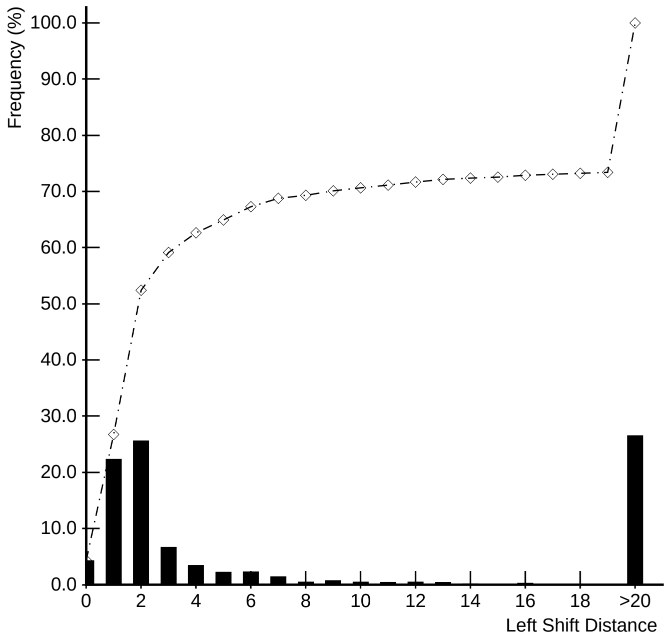


Figure 3.7: Histogram of normalizing shift distance

of 43% can be broken down into 20% effective addition and 23% effective subtraction. As effective additions do not require any normalization in the close path, they complete in the first cycle. An implementation allowing effective addition to complete in the first cycle is referred to as *adds*, and has the following performance:

$$\begin{aligned} \text{Average Latency} &= 3 \times (.57) + 2 \times (.23) + 1 \times (.20) = 2.37 \text{ cycles} \\ \text{Speedup} &= 3/2.37 = 1.27 \end{aligned}$$

Thus, *adds* reduces the average latency to 2.37 cycles, for a speedup of 1.27.

Figure 3.7 is a histogram of the normalizing left shift distances for effective subtractions in the *CLOSE* path. From figure 3.7, the majority of the normalizing shifts occur for distances of less than three bits. Only 4.4% of the effective subtractions in the *CLOSE* path require no normalizing shift. However, 22.4% of the subtractions require a 1 bit normalizing left shift, and 25.7% of the subtractions require a 2 bit normalizing left shift. In total, 52.5% of the *CLOSE* path subtractions require a left shift less than or equal to 2 bits. The inclusion of separate hardware to handle these frequent short shifts provides a performance gain.

Three implementations of the *One Cycle* algorithm could be used to exploit this behavior. They are denoted *subs0*, *subs1*, and *subs2*, which allow completion in the first cycle for effective subtractions with maximum normalizing shift distances of 0, 1, and 2 bits respectively. The most aggressive implementation *subs2* has the following performance:

$$\begin{aligned} \text{Average Latency} &= 3 \times (.57) + 2 \times (.11) + 1 \times (.32) = 2.25 \text{ cycles} \\ \text{Speedup} &= 3/2.25 = 1.33 \end{aligned}$$

Allowing all effective additions and those effective subtractions with normalizing shift distances of 0, 1, and 2 bits to complete in the first cycle reduces the average latency to 2.25 cycles, for a speedup of 1.33.

The performance of the proposed techniques is summarized in figure 3.8. For each technique, the average latency is shown, along with the speedup provided over the base *Two Path* FP adder with a fixed latency of three cycles.

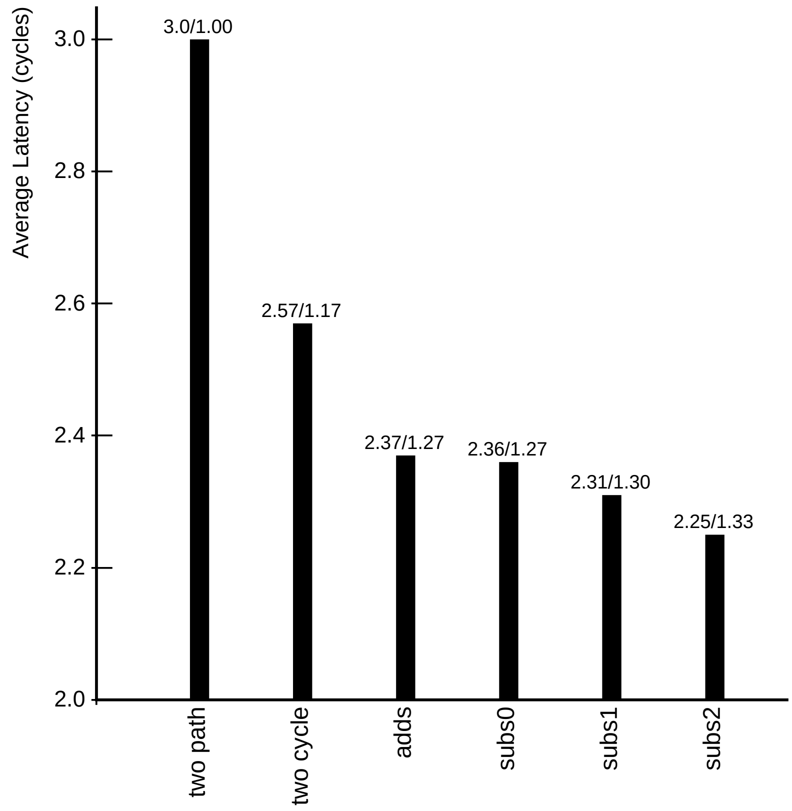


Figure 3.8: Performance summary of proposed techniques

## 3.5 Summary

There are two techniques for reducing the average latency of FP addition. Previous research has shown techniques to guarantee a maximum latency of three cycles in high clock-rate processors. Additional performance can be achieved in dynamic instruction scheduling processors by exploiting the distribution of operands that use the *CLOSE* path. It has been shown that 43% of the operands in the SPECfp92 applications use the *CLOSE* path, resulting in a speedup of 1.17 for the *Two Cycle* algorithm. By allowing effective additions in the *CLOSE* path to complete in the first cycle, a speedup of 1.27 is achieved. For even higher performance, an implementation of the *One Cycle* algorithm achieves a speedup of 1.33 by allowing effective subtractions requiring very small normalizing shifts to complete in the first cycle. These techniques do not add significant hardware, nor do they impact cycle time. They provide a reduction in average latency while maintaining single cycle throughput.

# Chapter 4

# Division Algorithms

## 4.1 Introduction

Many algorithms have been developed for implementing division in hardware. These algorithms differ in many aspects, including quotient convergence rate, fundamental hardware primitives, and mathematical formulations. In this chapter, a taxonomy of division algorithms is presented which divides the classes based upon the differences in the hardware operations used in their implementations, such as multiplication, subtraction, and table look-up.

Division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency [40]. Many practical division algorithms are not pure forms of a particular class, but rather are combinations of multiple classes. For example, a high performance algorithm may use a table look-up to gain an accurate initial approximation to the reciprocal, use a functional iteration algorithm to converge quadratically to the quotient, and complete in variable time using a variable latency technique.

In the past, others have presented summaries of specific classes of division algorithms and implementations. The most common implementation of digit recurrence division in modern processors has been named *SRT* division by Freiman [41], taking its name from the initials of Sweeney, Robertson [42] and Tocher [43], who discovered the algorithm independently in approximately the same time period. Two fundamental works on division by digit recurrence are Atkins [44], which is the first major analysis of SRT algorithms, and Tan [45], which derives and presents the theory of high-radix SRT division and an analytic method of implementing SRT look-up tables. Ercegovac and Lang [22] is a comprehensive treatment of division by digit recurrence. The theory and methodology of division by functional iteration is described in detail in Flynn [46]. Soderquist [47] presents a survey of division by digit recurrence and functional iteration along with performance and area tradeoffs in divider and square root design in the context of a specialized application.

This chapter synthesizes the fundamental aspects of these and other works, in order to clarify the division design space. The five classes of division algorithms are presented and analyzed in terms of the three major design parameters: latency in system clock cycles, cycle time, and area. Other issues related to the implementation of division in actual systems are also presented. The majority of the discussion is devoted to division. The theory of square root computation is an extension of the theory of division. As shown in chapter 2, square root operations occur nearly ten times less frequently than division in typical scientific applications, suggesting that fast square root implementations are not crucial to achieving high system performance. However, most of the analyses and conclusions for division can also be applied to the design of square root units.

## 4.2 Digit Recurrence Algorithms

The simplest and most widely implemented class of division algorithms is digit recurrence. Digit recurrence algorithms retire a fixed number of quotient bits in every iteration. Implementations of digit recurrence algorithms are typically of low complexity, utilize small area, and have relatively large latencies. The fundamental choices in the design of a digit recurrence divider are the radix, the allowed quotient digits, and the representation of the partial remainder. The radix determines how many bits of quotient are retired in an iteration, which fixes the division latency. Larger radices can reduce the latency, but increase the time for each iteration. Judicious choice of the allowed quotient digits can reduce the time for each iteration, but with a corresponding increase in complexity and hardware. Similarly, different representations of the partial remainder can reduce iteration time, with corresponding increases in complexity.

Various techniques have been proposed for further increasing division performance, including staging of simple low-radix stages, overlapping sections of one stage with another stage, and prescaling the input operands. All of these methods introduce tradeoffs in the time/area design space. This section introduces the principles of digit recurrence division, along with an analysis of methods for increasing the performance of digit recurrence implementations.

### 4.2.1 Definitions

Digit recurrence algorithms use subtractive methods to calculate quotients one digit per iteration. *SRT division* is the name of the most common form of digit recurrence division algorithm. The input operands are assumed to be represented in a normalized floating point format with fractional significands of $n$ binary digits in sign-magnitude representation. The algorithms presented here are applied only to the magnitudes of the significands of the input operands. Techniques for computing the resulting exponent and sign are straightforward. As mentioned earlier, the IEEE standard defines single and double precision formats, where $n$=24 for single precision and $n$=53 for double precision. The significand consists of a normalized quantity, with an explicit or implicit leading bit to the left of the implied binary point, and the magnitude of the significand is in the range [1,2). To simplify the presentation, this analysis assumes fractional quotients normalized to the range [0.5,1).

The quotient is defined to comprise $k$ radix-$r$ digits with

$$r = 2^b \tag{4.1}$$

$$k = \frac{n}{b} \tag{4.2}$$

where a division algorithm that retires $b$ bits of quotient in each iteration is said to be a radix-$r$ algorithm. Such an algorithm requires $k$ iterations to compute the final $n$ bit result. For a fractional quotient, 1 unit in the last place (ulp) $= r^{-n}$. Note that the radix of the algorithm need not be the same as that of the floating point representation nor the underlying physical implementation. Rather, they are independent quantities. For typical microprocessors that are IEEE 754 conforming, both the physical implementation and the floating point format are radix-2.

The following recurrence is used in every iteration of the SRT algorithm:

$$rP_0 = dividend \tag{4.3}$$

$$P_{j+1} = rP_j - q_{j+1} divisor \tag{4.4}$$

where $P_j$ is the partial remainder, or residual, at iteration $j$. In each iteration, one digit of the quotient is determined by the quotient-digit selection function:

$$q_{j+1} = SEL(rP_j, divisor) \tag{4.5}$$

The final quotient after $k$ iterations is then

$$q = \sum_{j=1}^{k} q_j r^{-j} \tag{4.6}$$

The *remainder* is computed from the final residual by:

$$\text{remainder} = \begin{cases} P_n & \text{if } P_n \geq 0 \\ P_n + \text{divisor} & \text{if } P_n < 0 \end{cases}$$

Furthermore, the quotient has to be adjusted when $P_n < 0$ by subtracting 1 ulp.

### 4.2.2 Implementation of Basic Scheme

A block diagram of a practical implementation of the basic SRT recurrence is shown in figure 4.1. The critical path of the implementation is shown by the dotted line. From equations (4.4) and (4.5), each iteration of the recurrence comprises the following steps:

- Determine next quotient digit $q_{j+1}$ by the quotient-digit selection function, a look-up table typically implemented as a PLA or combinational logic.

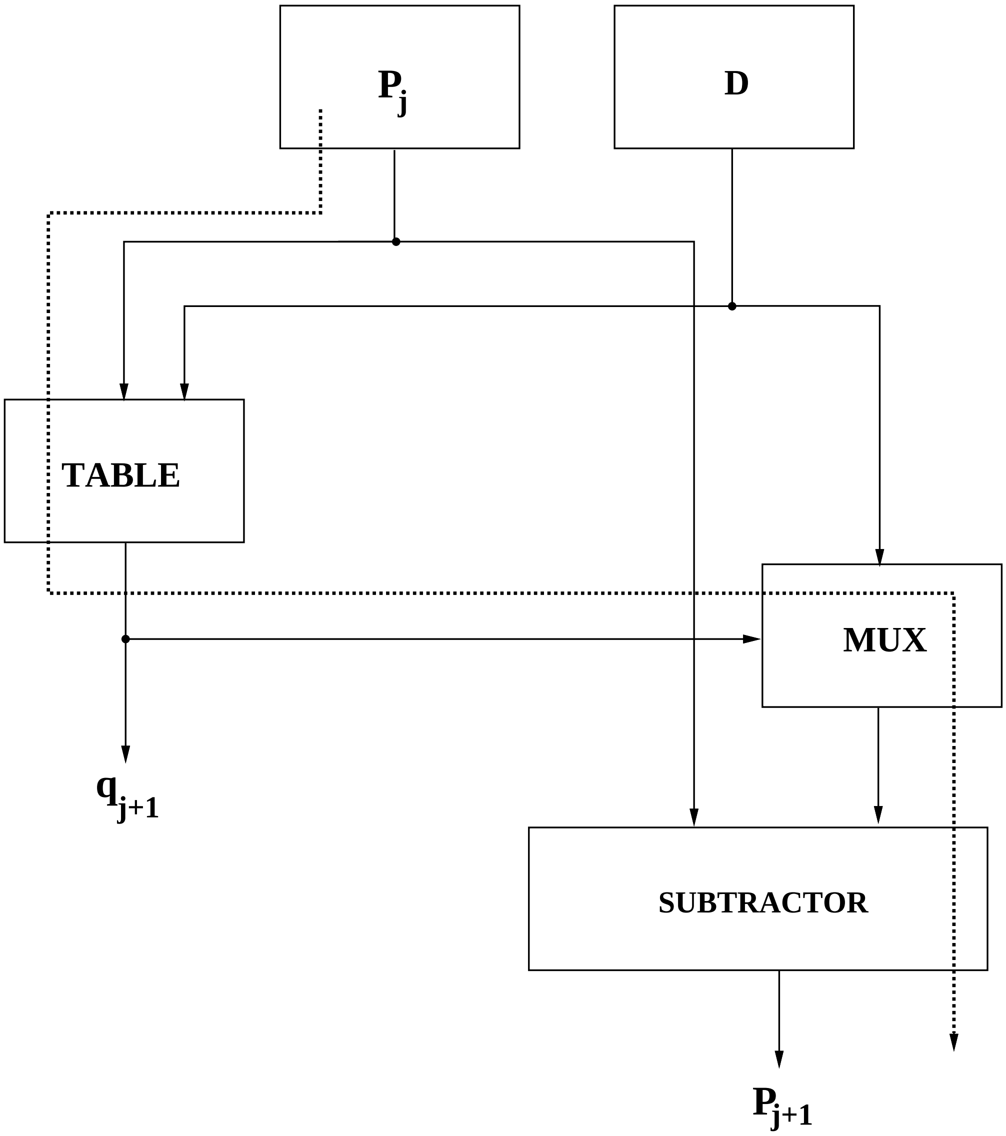


Figure 4.1: Basic SRT Divider Topology

- Generate the product $q_{j+1}\times$ *divisor*.
- Subtract $q_{j+1}\times$ *divisor* from the shifted partial remainder $r \times P_j$

Each of these components contributes to the overall cost and performance of the algorithm. Depending on certain parameters of the algorithm, the execution time and corresponding cost can vary widely.

### Choice of Radix

The fundamental method of decreasing the overall latency (in machine cycles) of the algorithm is to increase the radix $r$ of the algorithm. The radix is chosen to be a power of 2. The product of the radix and the partial remainder can then be formed by shifting. Assuming the same quotient precision, the number of iterations of the algorithm required to compute the quotient is reduced by a factor $f$ when the radix is increased from $r$ to $r^f$. For example, a radix 4 algorithm retires 2 bits of quotient in every iteration. Increasing to a radix 16 algorithm allows for retiring 4 bits in every iteration, halving the latency. This reduction does not come for free. As the radix increases, the quotient-digit selection becomes more complicated. It can be seen from figure 4.1 that quotient selection is on the critical path of the basic algorithm. The cycle time of the divider is defined as the minimum time to complete this critical path. While the number of cycles may have been reduced due to the increased radix, the time per cycle may have increased. As a result, the total time required to compute an $n$ bit quotient is not reduced as expected. Additionally, the generation of all required divisor multiples may become impractical or infeasible for higher radices. Thus, these two factors can offset some or possibly all of the performance gained by increasing the radix.

### Choice of Quotient Digit Set

In digit recurrence algorithms, some range of digits is decided upon for the allowed values of the quotient in each iteration. The simplest case is where, for radix $r$, there are exactly $r$ allowed values of the quotient. However, to increase the performance of the algorithm, we use a *redundant digit set.* Such a digit set can be composed of symmetric signed-digit consecutive integers, where the maximum digit is $a$. The digit set is made redundant by having more than $r$ digits in the set. In particular,

$$q_j \in \mathcal{D}_a = \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a-1, a\}$$

Thus, to make a digit set redundant, it must contain more than $r$ consecutive integer values including zero, and thus $a$ must satisfy

$$a \geq \lceil r/2 \rceil$$

The redundancy of a digit set is determined by the value of the redundancy factor $\rho$, which is defined as

$$\rho \;=\; \frac{a}{r-1}, \quad \rho > \frac{1}{2}$$

Typically, signed-digit representations have $a < r - 1$. When $a = \lceil \frac{r}{2} \rceil$, the representation is called *minimally redundant*, while that with $a = r - 1$ is called *maximally redundant*, with $\rho = 1$. A representation is known as *non-redundant* if $a = (r-1)/2$, while a representation where $a > r-1$ is called *over-redundant.* For the next residual $P_{j+1}$ to be bounded when a redundant quotient digit set is used, the value of the quotient digit must be chosen such that

$$|P_{j+1}| \;<\; \rho \times divisor$$

The design tradeoff can be noted from this discussion. By using a large number of allowed quotient digits $a$, and thus a large value for $\rho$, the complexity and latency of the quotient selection function can be reduced. However, choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Multiples that are powers of two can be formed by simply shifting. If a multiple is required that is not a power of two (e.g. three), an additional operation such as addition may also be required. This can add to the complexity and latency of generating the divisor multiple. The complexity of the quotient selection function and that of generating multiples of the divisor must be balanced.

After the redundancy factor $\rho$ is chosen, it is possible to derive the quotient selection function. A *containment condition* determines the selection intervals. A

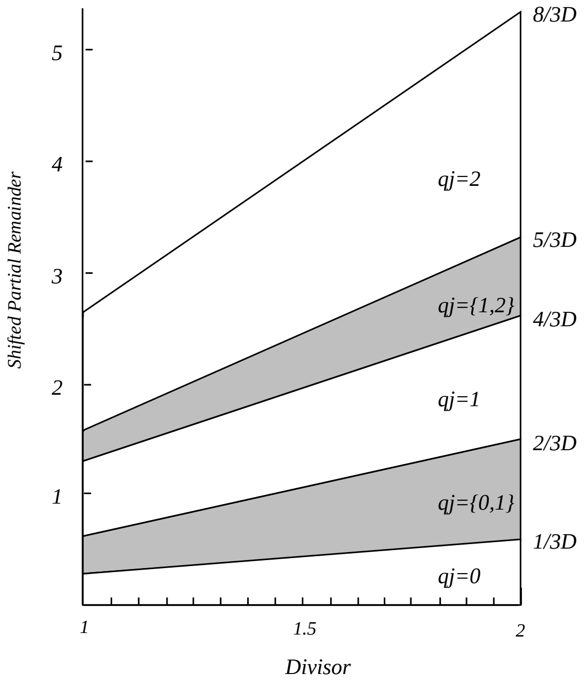


Figure 4.2: P-D diagram for radix-4

selection interval is the region in which a particular quotient digit can be chosen. These expressions are given by

$$U_k = (\rho + k)d \qquad L_k = (-\rho + k)d$$

where $U_k$ ($L_k$) is the largest (smallest) value of $rP_j$ such that it is possible for $q_{j+1} = k$ to be chosen and still keep the next partial remainder bounded. The *P-D diagram* is a useful visual tool when designing a quotient-digit selection function. It plots the shifted partial remainder vs. the divisor. The selection interval bounds $U_k$ and $L_k$ are drawn as lines starting at the origin with slope $\rho + k$ and $-\rho + k$, respectively. A P-D diagram is shown in figure 4.2 with $r = 4$ and $a = 2$. The shaded regions are the overlap regions where more than one quotient digit may be selected.

**Residual Representation**

The residual can be represented in either of two different forms, either *redundant* or *nonredundant* forms. Conventional two's complement representation is an example of a *nonredundant* form, while carry-save two's complement representation is an example of a *redundant* form. Each iteration requires a subtraction to form the next residual. If this residual is in a nonredundant form, then this operation requires a full-width adder requiring carry propagation, increasing the cycle time. If the residual is computed in a redundant form, a carry-free adder, such as a carry-save adder (CSA), can be used in the recurrence, minimizing the cycle time. However, the quotient-digit selection, which is a function of the shifted residual, becomes more complex. Additionally, twice as many registers are required to store the residual between iterations. Finally, if the remainder is required from the divider, the last residual has to be converted to a conventional representation. At a minimum, it is necessary to determine the sign of the final remainder in order to implement a possible quotient correction step, as discussed previously.

**Quotient-Digit Selection Function**

Critical to the performance of a divider is the efficient implementation of the quotient selection function. If a redundant representation is chosen for the residual, the residual is not exactly known, and neither is the exact next quotient digit. However, by using a redundant quotient digit set, the residual does not need to be known exactly to select the next quotient digit. It is only necessary to know the exact range of the residual in figure 4.2. The selection function is realized by approximating the residual $P_j$ and *divisor* to compute $q_{j+1}$. This is typically done by means of a small look-up table. The challenge in the design is deciding how many bits of $P_j$ and *divisor* are needed, while simultaneously minimizing the complexity of the table. Chapter 5 presents a methodology for performing this analysis along with several techniques for minimizing table complexity.

### 4.2.3 Increasing Performance

Several techniques have been reported for improving the performance of SRT division including [48], [49], [50], [51], [52], [53], [16], [54], [55], [56], [17]. Some of these approaches are discussed below.

### Simple Staging

In order to retire more bits of quotient in every cycle, a simple low radix divider can be replicated many times to form a higher radix divider, as shown in figure 4.3. In this implementation, the critical path is equal to:

$$t_{iter} \quad = \quad t_{reg} + 2(t_{qsel} + t_{qDsel} + t_{CSA}) \tag{4.7}$$

One advantage of unrolling the iterations by duplicating the lower-radix hardware is that the contribution of register overhead to total delay is reduced. The more the iterations are unrolled, the less of an impact register overhead has on total delay. However, the added area due to each stage and the increase in cycle time must be carefully considered.

In general, the implementation of divider hardware can range from totally sequential, as in the case of a single stage of hardware, to fully combinational, where the hardware is replicated enough such that the entire quotient can be determined combinationally in hardware. For totally or highly sequential implementations, the hardware requirements are small, saving chip area. This also leads to very fast cycle times, but the radix is typically low. Hardware replication can yield a very low latency in clock cycles due to the high radix but can occupy a large amount of chip area and have unacceptably slow cycle times.

One alternative to hardware replication to reduce division latency is to clock the divider at a faster frequency than the system clock. In the HP PA-7100, the very low cycle time of the radix-4 divider compared with the system clock allows it to retire 4 bits of quotient every machine cycle, effectively becoming a radix-16 divider [57]. In the succeeding generation HP PA-8000 processor, due to a higher system clock frequency, the divider is clocked at the same frequency as the rest of the CPU, increasing the latency (in cycles) by a factor of two [58].

The radix-16 divider in the AMD 29050 microprocessor [59] is another example of achieving higher radix by clocking a lower radix core at a higher frequency. In this implementation, a maximally-redundant radix-4 stage is clocked at twice the system clock frequency to form a radix-16 divider. Two dynamic short CPAs are used in front of the quotient selection logic, such that in one clock phase the first

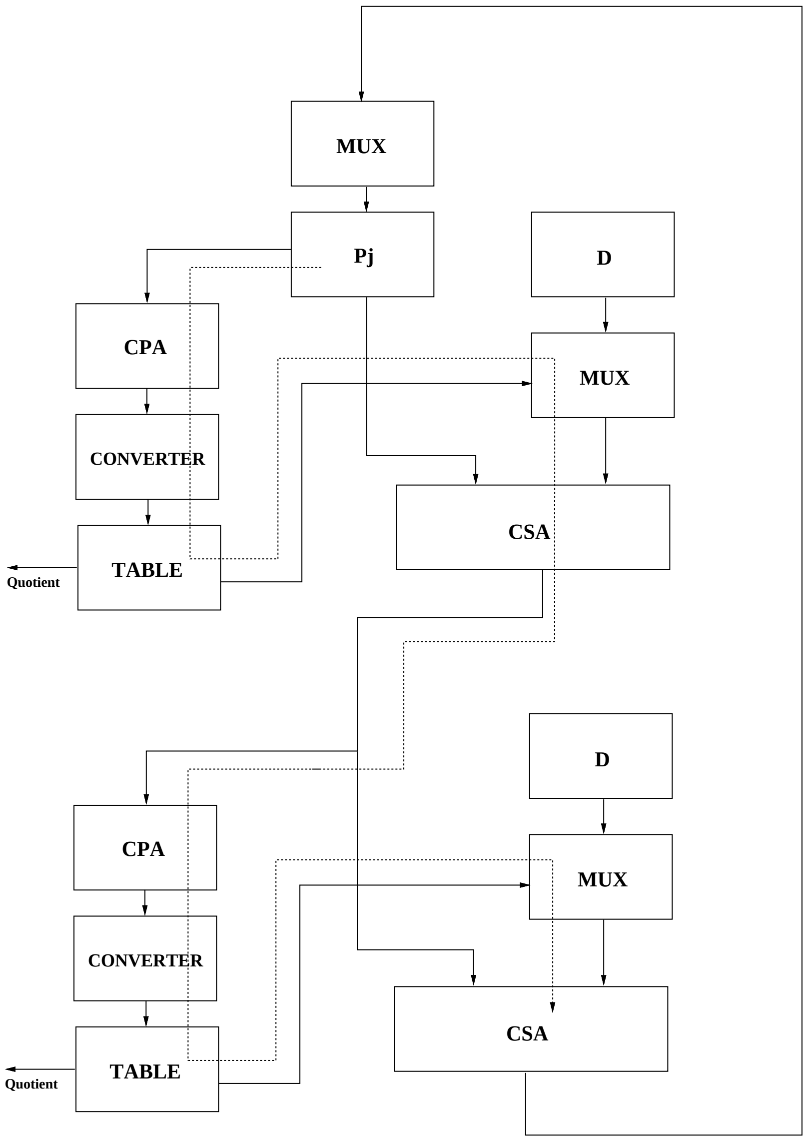


Figure 4.3: Higher radix using hardware replication

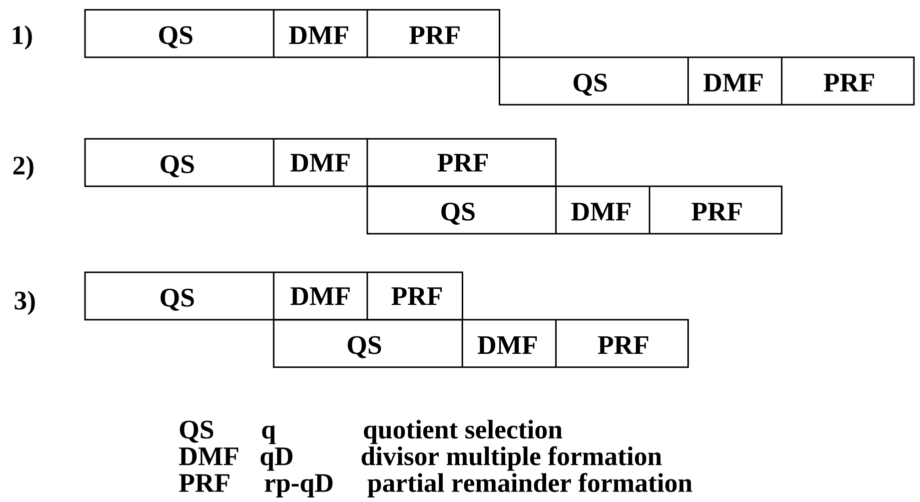


Figure 4.4: Three methods of overlapping division components

CPA evaluates and the second CPA precharges, while in the other clock phase the first CPA precharges and the second CPA evaluates. In this way, one radix-4 iteration is completed in each phase of the clock, for a total of two iterations per clock cycle.

### Overlapping Execution

It is possible to overlap or pipeline the components of the division step in order to reduce the cycle time of the division step [56]. This is illustrated in figure 4.4. The standard approach is represented in this figure by approach 1. Here, each quotient selection is dependent on the previous partial remainder, and this defines the cycle time. Depending upon the relative delays of the three components, approaches 2 or 3 may be more desirable. Approach 2 is appropriate when the overlap is dominated by partial remainder formation time. This would be the case when the partial remainder is not kept in a redundant form. Approach 3 is appropriate when the overlap is dominated by quotient selection, as is the case when a redundant partial remainder is used.

### Overlapping Quotient Selection

To avoid the increase in cycle time that results from staging radix-r segments together in forming higher radix dividers, some additional quotient computation can proceed in parallel [22],[49],[56]. The quotient-digit selection of stage $j+2$ is overlapped with the quotient-digit selection of stage $j+1$, as shown in figure 4.5. This is accomplished by calculating an estimate of the next partial remainder and the quotient-digit selection for $q_{j+2}$ conditionally for all $2a+1$ values of the previous quotient digit $q_{j+1}$, where $a$ is the maximum allowed quotient digit. Once the true value of $q_{j+1}$ is known, it selects the correct value of $q_{j+2}$. As can be seen from figure 4.5, the critical path is equal to:

$$t_{iter} \quad = \quad t_{reg} + t_{qsel} + t_{qDsel} + 2t_{CSA} + t_{mux(data)} \tag{4.8}$$

Accordingly, comparing the simple staging of two stages with the overlapped quotient selection method for staging, the critical path has been reduced by

$$\Delta t_{iter} \quad = \quad t_{qsel} + t_{qDsel} - t_{mux(data)} \tag{4.9}$$

This is a reduction of slightly more than the delay of one stage of quotient-digit selection, at the cost of replicating $2a+1$ quotient-digit selection functions. This scheme has diminishing returns when overlapping more than two stages. Each additional stage requires the calculation of an additional factor $(2a+1)$ of quotient-digit values. Thus the $k$th additional stage requires $(2a+1)^k$ replicated quotient-selection functions. Because of this exponential growth in hardware, only very small values of $k$ are feasible in practice.

Prabhu [60] discusses a radix-8 shared square root design that utilizes overlapping quotient selection in the Sun UltraSPARC microprocessor. In this implementation, three radix-2 stages are cascaded to form a radix-8 divider. The second stage conditionally computes all three possible quotient digits of the the first stage, and the third stage computes all three possible quotient digits of the second stage. In the worst case, this involves replication of three quotient-selection blocks for the second stage and nine blocks for the third stage. However, by recognizing that two of the

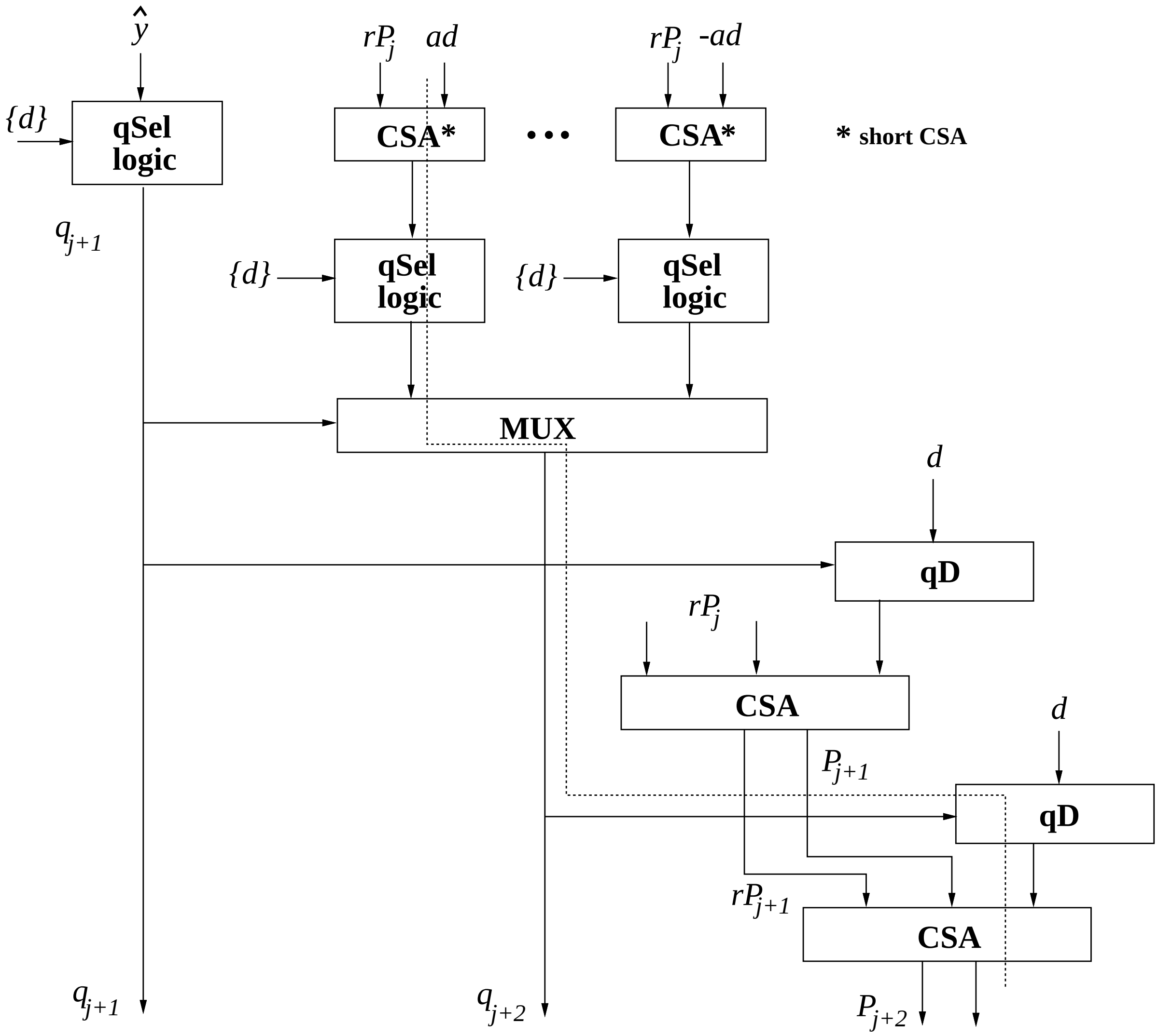


Figure 4.5: Higher radix by overlapping quotient selection

nine blocks conditionally compute the identical quotient bits as another two blocks, only seven are needed.

### Overlapping Remainder Computation

A further optimization that can be implemented is the overlapping of the partial remainder computation, also used in the UltraSPARC. By replicating the hardware to compute the next partial remainder for each possible quotient digit, the latency of the recurrence is greatly reduced.

Oberman [16] and Quach [54] report optimizations for radix-4 implementations. For radix-4, it might seem that because of the five possible next quotient digits, five copies of partial remainder computation hardware are required. However, in the design of quotient-selection logic, the sign of the next quotient digit is known in advance, as it is just the sign of the previous partial remainder. This reduces the number of copies of partial remainder computation hardware to three: 0, $\pm 1$, and $\pm 2$. However, from an analysis of a radix-4 quotient-digit selection table, the boundary between quotient digits 0 and 1 is readily determined. To take advantage of this, the quotient digits are encoded as:

$$\begin{aligned} q(-2) &= Sq_2 \\ q(-1) &= Sq_1\overline{q}_2 \\ q(0) &= \overline{q}_1\overline{q}_2 \\ q(1) &= \overline{S}q_1\overline{q}_2 \\ q(2) &= \overline{S}q_2 \end{aligned}$$

In this way, the number of copies of partial remainder computation hardware can be reduced to two: 0 or $\pm 1$, and $\pm 2$. A block diagram of a radix-4 divider with overlapped remainder computation is shown in figure 4.6. The choice of 0 or $\pm 1$ is made by $q_1$ early, after only a few gate delays, by selecting the proper input of a multiplexor. Similarly, $q_2$ selects a multiplexor to choose which of the two banks of hardware is the correct one, either the 0 or $\pm 1$ bank, or the $\pm 2$ bank. The critical path of the divider becomes: $\max(tq_1, t_{CSA}) + 2t_{mux} + t_{shortCPA} + t_{reg}$. Thus, at the

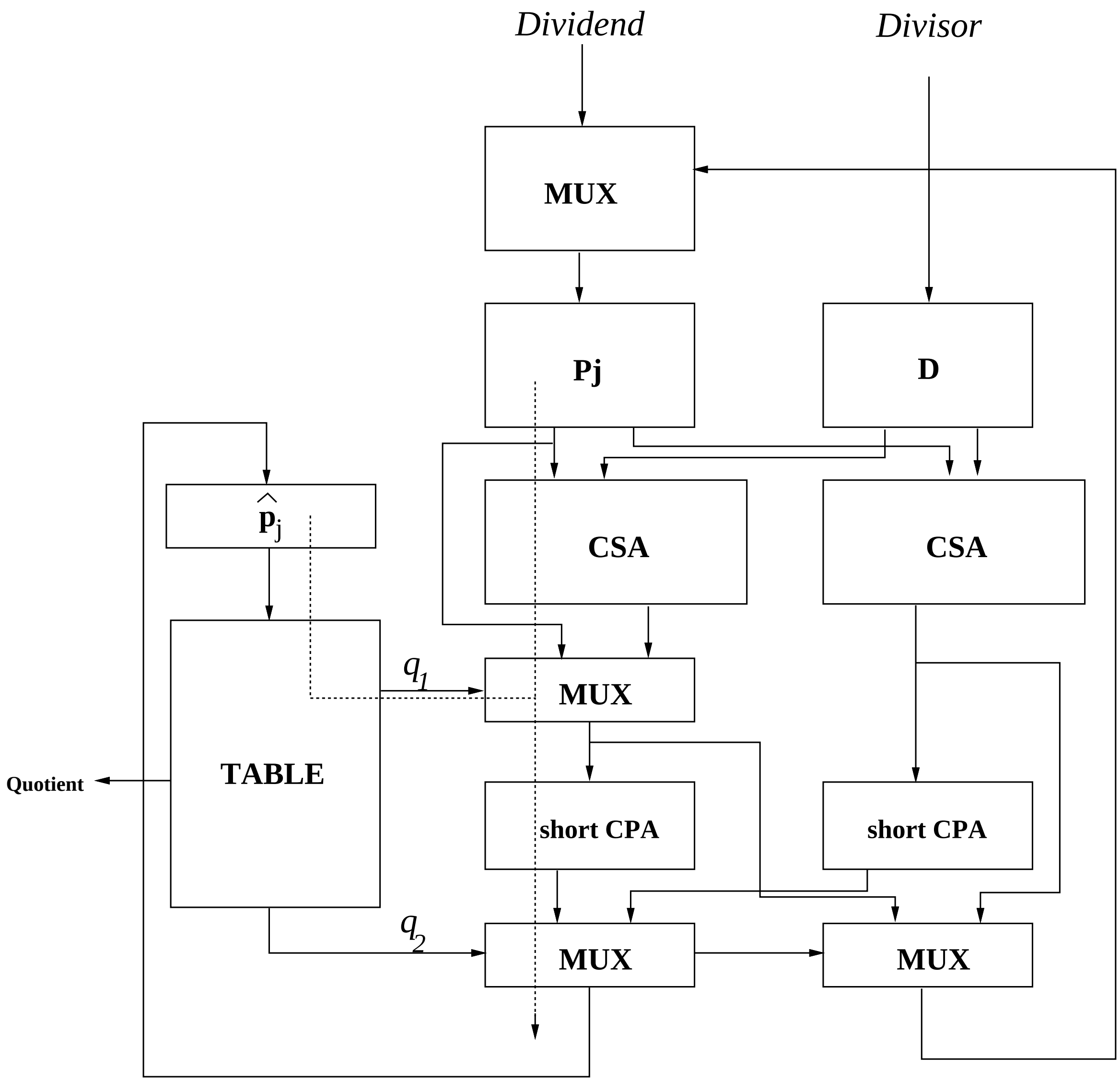


Figure 4.6: Radix-4 with overlapped remainder computation

expense of duplicating the remainder computation hardware once, the cycle time of the standard radix-4 divider is nearly halved.

### Range Reduction

Higher radix dividers can be designed by partitioning the implementation into lower radix segments, which are cascaded together. Unlike simple staging, in this scheme there is no shifting of the partial remainder between segments. Multiplication by the radix $r$ is performed only between iterations of the step, but not between segments. The individual segments reduce the range of the partial remainder so that it is usable by the remaining segments [22],[49].

A radix-8 divider can be designed using a cascade of a radix-2 segment and a radix-4 segment. In this implementation the quotient digit sets are given by:

$$q_{j+1} = q^h + q^l \qquad q^h \in \{-4, 0, 4\}, \quad q^l \in \{-2, -1, 0, 1, 2\}$$

However, the resulting radix-8 digit set is given by:

$$q_{j+1} = \{-6, \ldots, 6\} \quad \rho = 6/7$$

When designing the quotient-digit selection hardware for both $q_h$ and $q_l$, it should be realized that these are not standard radix-2 and radix-4 implementations, since the bounds on the step are set by the requirements for the radix-8 digit set. Additionally, the quotient-digit selections can be overlapped as discussed previously to reduce the cycle time. In the worst case, this overlapping involves two short CSAs, two short CPAs, and three instances of the radix-4 quotient-digit selection logic. However, to distinguish the choices of $q^h = 4$ and $q^h = -4$, an estimate of the sign of the partial remainder is required, which can be done with only three bits of the carry-save representation of the partial remainder. Then, both $q^h = 4$ and $q^h = -4$ can share the same CSA, CPA and quotient-digit selection logic by muxing the input values. This overall reduction in hardware has the effect of increasing the cycle time by the delay of the sign detection logic and a mux.

The critical path for generating $q^l$ is given by:

$$t_{iter} = t_{reg} + t_{signest} + t_{mux} + t_{CSA} + t_{shortCPA} + t_{qlsel} + t_{mux(data)} \tag{4.10}$$

In order to form $P_{j+1}$, $q^l$ is used to select the proper divisor multiple which is then subtracted from the partial remainder from the radix-2 segment. The additional delay to form $P_{j+1}$ is a mux select delay and a CSA. For increased performance, it is possible to precompute all partial remainders in parallel and use $q^l$ to select the correct result. This reduces the additional delay after $q^l$ to only a mux delay.

### Operands Scaling

In higher radix dividers the cycle time is generally determined by quotient-digit selection. The complexity of quotient-digit selection increases exponentially for increasing radix. To decrease cycle time, it may be desirable to reduce the complexity of the quotient-digit selection function using techniques beyond those presented, at the cost of adding additional cycles to the algorithm. From an analysis of a quotient-digit selection function the maximum overlap between allowed quotient digits occurs for the largest value of the divisor. Assuming a normalized divisor in the range $1/2 \leq d < 1$, the greatest amount of overlap occurs close to $d = 1$. To take advantage of this overlap, the divisor can be restricted to a range close to 1 by *prescaling* the divisor [48],[61]. In order that the quotient be preserved, either the dividend also must be prescaled, or else the quotient must be postscaled. In the case of prescaling, if the true remainder is required after the computation, postscaling is required. The dividend and the divisor are prescaled by a factor $M$ so that the scaled divisor $z$ is

$$1 - \alpha \leq z = Md \leq 1 + \beta \tag{4.11}$$

where $\alpha$ and $\beta$ are chosen in order to provide the same scaling factor for all divisor intervals and to ensure that the quotient-digit selection is independent of the divisor. The initial partial remainder is the scaled dividend: the smaller the range of $z$, the simpler the quotient-digit selection function. However, shrinking the range of $z$ becomes more complex for smaller ranges. Thus, a design tradeoff exists between these two constraints.

By restricting the divisor to a range near 1, the quotient-digit selection function becomes independent of the actual divisor value, and thus is simpler to implement. The radix-4 implementation reported by Ercegovac [48] uses 6 digits of the redundant partial remainder as inputs to the quotient-digit selection function. This function assimilates the 6 input digits in a CPA, and the 6 bit result is used to consult a look-up table to provide the next quotient-digit. The scaling operation uses a 3-operand adder. If a CSA is already used for the division recurrence, no additional CSAs are required and the scalings proceed in sequence. To determine the scaling factor for each operand, a small table yields the proper factors to add or subtract in the CSA to determine the scaled operand. Thus, prescaling requires a minimum of two additional cycles to the overall latency; one to scale the divisor, and one to assimilate the divisor in a carry-propagate adder. The dividend is scaled in parallel with the divisor assimilation, and it can be used directly in redundant form as the initial partial remainder.

Enhancements to the basic prescaling algorithms have been reported by Montuschi [52] and Srinivas [55]. Montuschi uses an over-redundant digit set combination with operand prescaling. The proposed radix-4 implementation uses the over-redundant digit set $\{\pm 4,\ \pm 3,\ \pm 2,\ \pm 1,\ 0\}$. The quotient-digit selection function uses a truncated redundant partial remainder that is in the range $[-6, 6]$, requiring four digits of the partial remainder as input. A 4-bit CPA is used to assimilate the four most significant digits of the partial remainder and to add a 1 in the least significant position. The resulting 4 bits in two's complement form represent the next quotient digit. The formation of the $\pm 3d$ divisor multiple is an added complication, and the solution is to split the quotient digit into two separate stages, one with digit set $\{0, \pm 4\}$ and one with $\{0, \pm 1, -2\}$. This is the same methodology used in the range reduction techniques previously presented. Thus, the use of a redundant digit set simplifies the quotient-digit selection from requiring 6 bits of input to only 4 bits.

Srinivas [55] reports an implementation of prescaling with a maximally redundant digit set. This implementation represents the partial remainder in radix-2 digits $\{-1, 0, +1\}$ rather than carry-save form. Each radix-2 digit is represented by 2 bits. Accordingly, the quotient-selection function uses only 3 digits of the radix-2 encoded partial remainder. The resulting quotient digits produced by this algorithm belong to the maximally redundant digit set $\{-3, \cdots, +3\}$. This simpler quotient-digit selection function decreases the cycle time relative to a regular redundant digit set with prescaling implementation. Srinivas reports a 1.21 speedup over Ercegovac's regular redundant digit set implementation, and a 1.10 speedup over Montuschi's over-redundant digit set implementation, using $n = 53$ IEEE double precision mantissas. However, due to the larger than regular redundant digit sets in the implementations of both Montuschi and Srinivas, each requires hardware to generate the $\pm 3d$ divisor multiple, which in these implementations results in requiring an additional 53 CSAs.

### 4.2.4 Quotient Conversion

As presented so far, the quotient has been collected in a redundant form, such that the positive values have been stored in one register, and the negative values in another. At the conclusion of the division computation, an additional cycle is required to assimilate these two registers into a single quotient value using a carry-propagate adder for the subtraction. However, it is possible to convert the quotient digits as they are produced such that an extra addition cycle is not required. This scheme is known as on-the-fly conversion [62].

In on-the-fly conversion, two forms of the quotient are kept in separate registers throughout the iterations, $Q_k$ and $QM_k$. $QM_k$ is defined to be equal to $Q_k - r^{-k}$. The values of these two registers for step $k+1$ are defined by:

$$Q_{k+1} = \begin{cases} (Q_k, q_{k+1}) & \text{if } q_{k+1} \geq 0 \\ (QM_k, (r - |q_{k+1}|)) & \text{if } q_{k+1} < 0 \end{cases}$$

and

$$QM_{k+1} = \begin{cases} (Q_k, (q_{k+1} - 1)) & \text{if } q_{k+1} > 0 \\ (QM_k, (r - |q_{k+1}| - 1)) & \text{if } q_{k+1} \leq 0 \end{cases}$$

with the initial conditions that $Q_0 = QM_0 = 0$. From these conditions on the values of $Q_k$ and $QM_k$, it can be seen that all of the additions can be implemented with concatenations. As a result, there is no carry or borrow propagation required. As every quotient digit is formed, each of these two registers is updated appropriately, either through register swapping or concatenation.

### 4.2.5 Rounding

For floating point representations such as the IEEE 754 standard, provisions for rounding are required. Traditionally, this is accomplished by computing an extra guard digit in the quotient and examining the final remainder. One ulp is conditionally added based on the rounding mode selected and these two values. The disadvantages in the traditional approach are that 1) the remainder may be negative and require a restoration step, and 2) the the addition of one ulp may require a full carry-propagate-addition. Accordingly, support for rounding can be expensive, both in terms of area and performance.

The previously described on-the-fly conversion can be extended to perform on-the-fly rounding [63]. This technique requires keeping a third version of the quotient at all times $QP_k$, where $QP_k = Q_k + r^{-k}$. The values of this register for step $k+1$ is defined by:

$$QP_{k+1} = \begin{cases} (QP_k, 0) & \text{if } q_{k+1} = r-1 \\ (Q_k, (q_{k+1}+1)) & \text{if } -1 \leq q_{k+1} \leq r-2 \\ (QM_k, (r - |q_{k+1}| + 1)) & \text{if } q_{k+1} < -1 \end{cases}$$

Correct rounding requires the computation of the sign of the final remainder and the determination of whether the final remainder is exactly zero. Sign detection logic can require some form of carry-propagation detection network, such as in standard carry-lookahead adders, while zero-remainder detection can require the logical ORing of the assimilated final remainder. Faster techniques for computing the sign and zero-remainder condition are presented in [22]. The final quotient is appropriately selected from the three available versions.

## 4.3 Functional Iteration

Unlike digit recurrence division, division by functional iteration utilizes multiplication as the fundamental operation. The primary difficulty with subtractive division is the linear convergence to the quotient. Multiplicative division algorithms are able to take advantage of high-speed multipliers to converge to a result quadratically. Rather than retiring a fixed number of quotients bits in every cycle, multiplication-based algorithms are able to double the number of correct quotient bits in every iteration. However, the tradeoff between the two classes is not only latency in terms of the number of iterations, but also the length of each iteration in cycles. Additionally, if the divider shares an existing multiplier, the performance ramifications on regular multiplication operations must be considered. It is shown in chapter 2 that in typical floating point applications, the performance degradation due to a shared multiplier is small. Accordingly, if area must be minimized, an existing multiplier may be shared with the division unit with only minimal system performance degradation. This section presents the algorithms used in multiplication-based division, both of which are related to the Newton-Raphson equation.

### 4.3.1 Newton-Raphson

Division can be written as the product of the dividend and the reciprocal of the divisor, or

$$Q = a/b = a \times (1/b), \tag{4.12}$$

where $Q$ is the quotient, $a$ is the dividend, and $b$ is the divisor. In this case, the challenge becomes how to efficiently compute the reciprocal of the divisor. In the Newton-Raphson algorithm, a *priming function* is chosen which has a root at the reciprocal [46]. In general, there are many root targets that could be used, including $\frac{1}{b}$, $\frac{1}{b^2}$, $\frac{a}{b}$, and $1-\frac{1}{b}$. The choice of which root target to use is arbitrary. The selection is made based on convenience of the iterative form, its convergence rate, its lack of divisions, and the overhead involved when using a target root other than the true quotient.

The most widely used target root is the divisor reciprocal $\frac{1}{b}$, which is the root of the priming function:

$$f(X) \quad = \quad 1/X - b = 0. \tag{4.13}$$

The well-known quadratically converging Newton-Raphson equation is given by:

$$x_{i+1} \quad = \quad x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.14}$$

The Newton-Raphson equation of (4.14) is then applied to (4.13), and this iteration is then used to find an approximation to the reciprocal:

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i + \frac{(1/X_i - b)}{(1/X_i^2)} = X_i \times (2 - b \times X_i) \qquad (4.15)$$

The corresponding error term is given by

$$\epsilon_{i+1} = \epsilon_i^2(b)$$

and thus the error in the reciprocal decreases quadratically after each iteration. As can be seen from (4.15), each iteration involves two multiplications and a subtraction. The subtraction is equivalent to forming the two's complement and is commonly replaced by it. Thus, two dependent multiplications and one two's complement operation are performed each iteration. The final quotient is obtained by multiplying the computed reciprocal with the dividend.

Rather than performing a complete two's complement operation at the end of each iteration to form $(2 - b \times X_i)$, it is possible instead to simply implement the one's complement of $b \times X_i$, as was done in the IBM 360/91 [64] and the Astronautics ZS-1 [65]. This only adds a small amount error, as the one's and two's complement operations differ only by 1 ulp. The one's complement avoids the latency penalty of carry-propagation across the entire result of the iteration, replacing it by a simple inverter delay.

While the number of operations per iteration is constant, the number of iterations required to obtain the reciprocal accurate to a particular number of bits is a function of the accuracy of the initial approximation $X_0$. By using a more accurate starting approximation, the total number of iterations required can be reduced. To achieve 53 bits of precision for the final reciprocal starting with only 1 bit, the algorithm will require 6 iterations:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 53$$

By using a more accurate starting approximation, for example 8 bits, the latency can be reduced to 3 iterations. By using at least 14 bits, the latency could be further reduced to only 2 iterations. Section 4.5 explores in more detail the use of look-up tables to increase performance.

### 4.3.2 Series Expansion

A different method of deriving a division iteration is based on a series expansion. A name sometimes given to this method is *Goldschmidt's algorithm* [66]. Consider the familiar Taylor series expansion of a function $g(y)$ at a point $p$,

$$g(y) = g(p) + (y-p)g'(p) + \frac{(y-p)^2}{2!}g''(p) + \cdots + \frac{(y-p)^n}{n!}g^{(n)}(p) + \cdots \quad (4.16)$$

In the case of division, it is desired to find the expansion of the reciprocal of the divisor, such that

$$q = \frac{a}{b} = a \times g(y), \quad (4.17)$$

where $g(y)$ can be computed by an efficient iterative method. A straightforward approach might be to choose $g(y)$ equal to $1/y$ with $p = 1$, and then to evaluate the series. However, it is computationally easier to let $g(y) = 1/(1+y)$ with $p = 0$, which is just the Maclaurin series. Then, the function is

$$g(y) = \frac{1}{1+y} = 1 - y + y^2 - y^3 + y^4 - \cdots. \quad (4.18)$$

So that $g(y)$ is equal to 1/b, the substitution $y = b - 1$ must be made, where b is bit normalized such that $0.5 \leq b < 1$, and thus $|Y| \leq 0.5$. Then, the quotient can be written as

$$q = a \times \frac{1}{1 + (b-1)} = a \times \frac{1}{1+y} = a \times (1 - y + y^2 - y^3 + \cdots)$$

which, in factored form, can be written as

$$q = a \times [(1-y)(1+y^2)(1+y^4)(1+y^8)\cdots]. \quad (4.19)$$

This expansion can be implemented iteratively as follows. An approximate quotient can be written as

$$q_i = \frac{N_i}{D_i}$$

where $N_i$ and $D_i$ are iterative refinements of the numerator and denominator after step $i$ of the algorithm. By forcing $D_i$ to converge toward 1, $N_i$ converges toward $q$.

Effectively, each iteration of the algorithm provides a correction term $(1+y^{2^i})$ to the quotient, generating the expansion of (4.19).

Initially, let $N_0 = a$ and $D_0 = b$. To reduce the number of iterations, $a$ and $b$ should both be prescaled by a more accurate approximation of the reciprocal, and then the algorithm should be run on the scaled $a'$ and $b'$. For the first iteration, let $N_1 = R_0 \times N_0$ and $D_1 = R_0 \times D_0$, where $R_0 = 1 - y = 2 - b$, or simply the two's complement of the divisor. Then,

$$D_1 = D_0 \times R_0 = b \times (1-y) = (1+y)(1-y) = 1 - y^2.$$

Similarly,

$$N_1 = N_0 \times R_0 = a \times (1-y).$$

For the next iteration, let $R_1 = 2 - D_1$, the two's complement of the new denominator. From this,

$$\begin{aligned} R_1 &= 2 - D_1 = 2 - (1-y^2) = 1 + y^2 \\ N_2 &= N_1 \times R_1 = a \times [(1-y)(1+y^2)] \\ D_2 &= D_1 \times R_1 = (1-y^2)(1+y^2) = (1-y^4) \end{aligned}$$

Continuing, a general relationship can be developed, such that each step of the iteration involves two multiplications

$$N_{i+1} = N_i \times R_i \quad \text{and} \quad D_{i+1} = D_i \times R_i \tag{4.20}$$

and a two's complement operation,

$$R_{i+1} = 2 - D_{i+1} \tag{4.21}$$

After $i$ steps,

$$N_i = a \times [(1-y)(1+y^2)(1+y^4)\cdots(1+y^{2^i})] \tag{4.22}$$

$$D_i = (1-y^{2^i}) \tag{4.23}$$

Accordingly, $N$ converges quadratically toward $q$ and $D$ converges toward 1. This can be seen in the similarity between the formation of $N_i$ in (4.22) and the series expansion of $q$ in (4.19). So long as $b$ is normalized in the range $0.5 \leq b < 1$, then $y < 1$, each correction factor $(1 + y^{2i})$ doubles the precision of the quotient. This process continues as shown iteratively until the desired accuracy of $q$ is obtained.

Consider the iterations for division. A comparison of equation (4.22) using the substitution $y = b - 1$ with equation (4.15) using $X_0 = 1$ shows that the results are identical iteration for iteration. Thus, the series expansion is mathematically identical to the Newton-Raphson iteration for $X_0 = 1$. Additionally, each algorithm can benefit from a more accurate starting approximation of the reciprocal of the divisor to reduce the number of required iterations. However, the implementations are not exactly the same. Newton-Raphson converges to a reciprocal, and then multiplies by the dividend to compute the quotient, whereas the series expansion first prescales the numerator and the denominator by the starting approximation and then converges directly to the quotient. Thus, the series expansion requires the overhead of one more multiplication operation as compared to Newton-Raphson. Each iteration in both algorithms comprises two multiplications and a two's complement operation. From (4.15), the multiplications in Newton-Raphson are dependent operations. In the series expansion iteration, the two multiplications of the numerator and denominator are independent operations and may occur concurrently. As a result, a series expansion implementation can take advantage of a pipelined multiplier to obtain higher performance in the form of lower latency per operation. In both iterations, unused cycles in the multiplier can be used to allow for more than one division operation to proceed concurrently. Specifically, a pipelined multiplier with a throughput of 1 and a latency of $l$ can have $l$ divisions operating simultaneously, each initiated at 1 per cycle. A performance enhancement that can be used for both iterations is to perform early computations in reduced precision. This is reasonable, because the early computations do not generate many correct bits. As the iterations continue, quadratically larger amounts of precision are required in the computation.

In practice, dividers based on functional iteration have used both versions. The Newton-Raphson algorithm was used in the Astronautics ZS-1 [65], Intel i860 [67], and the IBM RS/6000 [68]. The series expansion was used in the IBM 360/91 [64] and TMS390C602A [69]. Latencies for such dividers range from 11 cycles to more than 16 cycles, depending upon the precision of the initial approximation and the latency and throughput of the floating-point multiplier.

## 4.4 Very High Radix Algorithms

Digit recurrence algorithms are readily applicable to low radix division and square root implementations. As the radix increases, the quotient-digit selection hardware and divisor multiple process become more complex, increasing cycle time, area or both. To achieve very high radix division with acceptable cycle time, area, and means for precise rounding, we use a variant of the digit recurrence algorithms, with simpler quotient-digit selection hardware. The term "very high radix" applies roughly to dividers which retire more than 10 bits of quotient in every iteration. The very high radix algorithms are similar in that they use multiplication for divisor multiple formation and look-up tables to obtain an initial approximation to the reciprocal. They differ in the number and type of operations used in each iteration and the technique used for quotient-digit selection.

### 4.4.1 Accurate Quotient Approximations

In the high radix algorithm proposed by Wong [18], truncated versions of the normalized dividend $X$ and divisor $Y$ are used, denoted $X_h$ and $Y_h$. $X_h$ is defined as the high-order $m+1$ bits of $X$ extended with 0's to get a $n$-bit number. Similarly, $Y_h$ is defined as the high order $m$ bits of $Y$ extended with 1's to get a $n$-bit number. From these definitions, $X_h$ is always less than or equal to $X$ and $Y_h$ is always greater than or equal to $Y$. This implies that $1/Y_h$ is always less than or equal to $1/Y$, and therefore $X_h/Y_h$ is always less than or equal to $X/Y$.

The algorithm is as follows:

1. Initially, set the estimated quotient $Q$ and the variable $j$ to 0. Then, get an approximation of $1/Y_h$ from a look-up table, using the top $m$ bits of $Y$, returning an $m$ bit approximation. However, only $m-1$ bits are actually required to

index into the table, as the guaranteed leading one can be assumed. In parallel, perform the multiplication $X_h \times Y$.

2. Scale both the truncated divisor and the previously formed product by the reciprocal approximation. This involves two multiplications in parallel for maximum performance,

$$(1/Y_h) \times Y \quad \text{and} \quad (1/Y_h) \times (X_h \times Y)$$

The product $(1/Y_h) \times Y = Y'$ is invariant across the iterations, and therefore only needs to be performed once. Subsequent iterations use only one multiplication:

$$Y' \times P_h,$$

where $P_h$ is the current truncated partial remainder. The product $P_h \times 1/Y_h$ can be viewed as the next quotient digit, while $(P_h \times 1/Y_h) \times Y$ is the effective divisor multiple formation.

3. Perform the general recurrence to obtain the next partial remainder:

$$P' = P - P_h \times (1/Y_h) \times Y, \tag{4.24}$$

where $P_0 =$ X. Since all products have already been formed, this step only involves a subtraction.

4. Compute the new quotient as

$$\begin{aligned} Q' &= Q + (P_h/Y_h) \times (1/2^j) \\ &= Q + P_h \times (1/Y_h) \times (1/2^j) \end{aligned} \tag{4.25}$$

The new quotient is then developed by forming the product $P_h \times (1/Y_h)$ and adding the shifted result to the old quotient $Q$.

5. The new partial remainder $P'$ is normalized by left-shifting to remove any leading 0's. It can be shown that the algorithm guarantees $m-2$ leading 0's. The shift index $j$ is revised by $j' = j + m - 2$.

6. All variables are adjusted such that $j = j'$, $Q = Q'$, and $P = P'$.

7. Repeat steps 2 through 6 of the algorithm until $j \geq q$.

8. After the completion of all iterations, the top $n$ bits of $Q$ form the true quotient. Similarly, the final remainder is formed by right-shifting $P$ by $j - q$ bits. This remainder, though, assumes the use of the entire value of $Q$ as the quotient. If only the top $n$ bits of $Q$ are used as the quotient, then the final remainder is calculated by adding $Q_l \times Y$ to $P$, where $Q_l$ comprises the low order bits of $Q$ after the top $n$ bits.

This basic algorithm reduces the partial remainder $P$ by $m - 2$ bits every iteration. Accordingly, an $n$ bit quotient requires $\lceil n/(m-2) \rceil$ iterations.

Wong also proposes an advanced version of this algorithm [18] using the same iteration steps as in the basic algorithm presented earlier. However, in step 1, while $1/Y_h$ is obtained from a look-up table using the leading $m$ bits of $Y$, in parallel approximations for higher order terms are obtained from additional look-up tables, all indexed using the leading $m$ bits of $Y$. These additional tables have word widths of $b_i$ given by

$$b_i \;=\; (m \times t - t) + \lceil \log_2 t \rceil - (m \times i - m - i) \tag{4.26}$$

where $t$ is the number of terms of the series used, and thus the number of look-up tables. The value of $t$ must be at least 2, but all subsequent terms are optional. The advanced version reduces $P'$ by $m \times t - t - 1$ bits per iteration, and therefore the algorithm requires $\lceil n/(m \times t - t - 1) \rceil$ iterations.

As in SRT implementations, both versions of the algorithm can benefit by storing the partial remainder $P$ in a redundant representation. However, before any of the multiplications using $P$ as an operand take place, the top $m + 3$ bits of $P$ must be carry-assimilated for the basic method, and the top $m + 5$ bits of $P$ must be carry-assimilated for the advanced method. Similarly, the quotient $Q$ can be kept in a redundant form until the final iteration. After the final iteration, full carry-propagate additions must be performed to calculate $Q$ and $P$ in normal, non-redundant form.

The hardware required for this algorithm is as follows. At least one look-up table is required of size $2^{m-1}m$ bits. Three multipliers are required: one multiplier with carry assimilation of size $(m+1) \times n$ for the initial multiplications by the divisor $Y$, one carry-save multiplier with accumulation of size $(m+1) \times (n+m)$ for the iterations, and one carry-save multiplier of size $(m+1) \times m$ to compute the quotient segments. One carry-save adder is required to accumulate the quotient in each iteration. Two carry-propagate adders are required: one short adder at least of size $m+3$ bits to assimilate the most significant bits of the partial remainder $P$, and one adder of size $n+m$ to assimilate the final quotient.

A slower implementation of this algorithm might use the basic method with $m = 11$. The single look-up table would have $2^{11-1} = 1024$ entries, each 11 bits wide, for a total of 11K bits in the table, with a resulting latency of 9 cycles. A faster implementation using the advanced method with $m = 15$ and $t = 2$ would require a total table size of 736K bits, but with a latency of only 5 cycles. Thus, at the expense of several multipliers, adders, and two large look-up tables, the latency of division can be greatly reduced using this algorithm. In general, the algorithm requires at most $\lceil n/(m-2) \rceil + 3$ cycles.

### 4.4.2 Short Reciprocal

The Cyrix 83D87 arithmetic coprocessor utilizes a short reciprocal algorithm similar to the accurate quotient approximation method to obtain a radix $2^{17}$ divider [70],[71]. Instead of having several multipliers of different sizes, the Cyrix divider has a single 18x69 rectangular multiplier with an additional adder port that can perform a fused multiply/add. It can, therefore, also act as a 19x69 multiplier. Otherwise, the general algorithm is nearly identical to Wong:

1. Initially, an estimate of the reciprocal $1/Y_h$ is obtained from a look-up table. In the Cyrix implementation, this approximation is of low precision. This approximation is refined through two iterations of the Newton-Raphson algorithm to achieve a 19 bit approximation. This method decreases the size of the look-up table at the expense of additional latency. Also, this approximation is chosen to be intentionally larger than the true reciprocal by an amount no greater than $2^{-18}$. This differs from the accurate quotient method where the approximation

is chosen to be intentionally smaller than the true reciprocal.

2. Perform the recurrence

$$P' = P - P_h \times (1/Y_h) \times Y \quad (4.27)$$

$$Q' = Q + P_h \times (1/Y_h) \times (1/2^j) \quad (4.28)$$

where $P_0$ is the dividend $X$. In this implementation, the two multiplications of (4.27) need to be performed separately in each iteration. One multiplication is required to compute $P_h \times (1/Y_h)$, and a subsequent multiply/add is required to multiply by Y and accumulate the new partial remainder. The product $P_h \times (1/Y_h)$ is a 19 bit high radix quotient digit. The multiplication by $Y$ forms the divisor multiple required for subtraction. However, the multiplication $P_h \times (1/Y_h)$ required in (4.28) can be reused from the result computed for (4.27). Only one multiplication is required in the accurate quotient method because the product $(1/Y_h) \times Y$ is computed once at the beginning in full precision, and can be reused on every iteration. The Cyrix multiplier only produces limited precision results, 19 bits, and thus the multiplication by $Y$ is repeated at every iteration. Because of the specially chosen 19 bit short reciprocal, along with the 19 bit quotient digit and 18 bit accumulated partial remainder, this scheme guarantees that 17 bits of quotient are retired in every iteration.

3. After the iterations, one additional cycle is required for rounding and post-correction. Unlike the accurate quotient method, on-the-fly conversion of the quotient digits is possible, as there is no overlapping of the quotient segments between iterations.

Thus, the short reciprocal algorithm is very similar to the accurate quotient algorithm. One difference is the method for generating the short reciprocal. However, either method could be used in both algorithms. The use of Newton-Raphson to increase the precision of a smaller initial approximation is chosen merely to reduce the size of the look-up table. The fundamental difference between the two methods is Cyrix's choice of a single rectangular fused multiplier/add unit with assimilation to perform all core operations. While this eliminates a majority of the hardware required in the accurate quotient method, it increases the iteration length from one multiplication to two due to the truncated results.

The short reciprocal unit can generate double precision results in 15 cycles: 6 cycles to generate the initial approximation by Newton-Raphson, 4 iterations with 2 cycles per iteration, and one cycle for postcorrection and rounding. With a larger table, the initial approximation can be obtained in as little as 1 cycle, reducing the total cycle count to 10 cycles. The radix of $2^{17}$ was chosen due to the target format of IEEE double extended precision, where $n = 64$. This divider can generate double extended precision quotients as well as double precision in 10 cycles. In general, this algorithm requires at least $2\lceil n/b \rceil + 2$ cycles.

### 4.4.3 Rounding and Prescaling

Ercegovac and Lang [21] report a high radix division algorithm which involves obtaining an accurate initial approximation of the reciprocal, scaling both the dividend and divider by this approximation, and then performing multiple iterations of quotient-selection by rounding and partial remainder reduction by multiplication and subtraction. By retiring $b$ bits of quotient in every iteration, it is a radix $2^b$ algorithm. The algorithm is as follows to compute $X/Y$:

1. Obtain an accurate approximation of the reciprocal from a table to form the scaling factor $M$. Rather than using a constant piecewise approximation, this method uses the previously presented technique of linear approximation to the reciprocal.

2. Scale $Y$ by the scaling factor $M$. This involves the carry-save multiplication of the $b + 6$ bit value $M$ and the $n$ bit operand $Y$ to form the $n + b + 5$ bit scaled quantity $Y \times M$.

3. Scale $X$ by the scaling factor $M$, yielding an $n + b + 5$ bit quantity $X \times M$. This multiplication along with the multiplication of step 2 both can share the $(b + 6) \times (n + b + 5)$ multiplier used in the iterations. In parallel, the scaled

divisor $M \times Y$ is assimilated. This involves an $(n + b + 5)$ bit carry-propagate adder.

4. Determine the next quotient digit, needed for the general recurrence:

$$P_{j+1} = rP_j - q_{j+1}(M \times Y) \tag{4.29}$$

where $P_0 = M \times X$. In this scheme, the choice of scaling factor allows for quotient-digit selection to be implemented simply by rounding. Specifically, the next quotient digit is obtained by rounding the shifted partial remainder in carry-save form to the second fractional bit. This can be done using a short carry-save adder and a small amount of additional logic. The quotient-digit obtained through this rounding is in carry-save form, with one additional bit in the least-significant place. This quotient-digit is first recoded into a radix-4 signed-digit set (-2 to +3), then that result is recoded to a radix-4 signed-digit set (-2 to +2). The result of quotient-digit selection by rounding requires $2(b+1)$ bits.

5. Perform the multiplication $q_{j+1} \times z$, where $z$ is the scaled divisor $M \times Y$, then subtract the result from $rP_j$. This can be performed in one step by a fused multiply/add unit.

6. Perform postcorrection and any required rounding. As discussed previously, postcorrection requires at a minimum sign detection of the last partial remainder and the correction of the quotient.

Throughout the iterations, on-the-fly quotient conversion is used.

The latency of the algorithm in cycles can be calculated as follows. At least one cycle is required to form the linear approximation $M$. One cycle is required to scale $Y$, and an additional cycle is required to scale $X$. $\lceil n/b \rceil$ cycles are needed for the iterations. Finally, one cycle is needed for the postcorrection and rounding. Therefore, the total number of cycles is given by

$$Cycles = \lceil n/b \rceil + 4$$

The hardware required for this algorithm is similar to the Cyrix implementation. One look-up table is required of size $2^{\lfloor b/2 \rfloor}(2b+11)$ bits to store the coefficients of the linear approximation. A $(b+6) \times (b+6)$ carry-save fused multiply/add unit is needed to generate the scaling factor $M$. One fused multiply/add unit is required of size $(b+6) \times (n+b+5)$ to perform the two scalings and the iterations. A recoder unit is necessary to recode both $M$ and $q_{j+1}$ to radix-4. Finally, combinational logic and a short CSA are required to implement the quotient-digit selection by rounding.

## 4.5 Look-up Tables

Functional iteration and very high radix division implementations can both benefit from a more accurate initial reciprocal approximation. Further, when a only low-precision quotient is required, it may be sufficient to use a look-up table to provide the result without the subsequent use of a refining algorithm. Methods for forming a starting approximation include direct approximations and linear approximations. More recently, partial product arrays have been proposed as methods for generating starting approximations while reusing existing hardware.

### 4.5.1 Direct Approximations

For modern division implementations, the most common method of generating starting approximations is through a look-up table. Such a table is typically implemented in the form of a ROM or a PLA. An advantage of look-up tables is that they are fast, since no arithmetic calculations need be performed. The disadvantage is that a look-up table's size grows exponentially with each bit of added accuracy. Accordingly, a tradeoff exists between the precision of the table and its size.

To index into a reciprocal table, it is assumed that the operand is IEEE normalized $1.0 \leq b < 2$. Given such a normalized operand, $k$ bits of the truncated operand are used to index into a table providing $m$ bits after the leading bit in the $m+1$ bit fraction reciprocal approximation $R_0$ which has the range $0.5 < recip \leq 1$. The total size of such a reciprocal table is $2^k m$ bits. The truncated operand is represented as $1.b'_1b'_2\cdots b'_k$, and the output reciprocal approximation is $0.1b'_1b'_2\cdots b'_m$. The only exception to this is when the input operand is exactly 1, in which case the output reciprocal should also be exactly 1. In this case, separate hardware is used to detect this. All input values have a leading-one that can be implied and does not index into the table. Similarly, all output values have a leading-one that is not explicitly stored in the table. The design of a reciprocal table starts with a specification for the minimum accuracy of the table, often expressed in bits. This value dictates the number of bits ($k$) used in the truncated estimate of $b$ as well as the minimum number of bits in each table entry ($m$). A common method of designing the look-up table is to implement a piecewise-constant approximation of the reciprocal function. In this case, the approximation for each entry is found by taking the reciprocal of the mid-point between $1.b'_1b'_2\cdots b'_k$ and its successor, where the mid-point is $1.b'_1b'_2\cdots b'_k1$. The reciprocal of the mid-point is rounded by adding $2^{-(m+2)}$ and then truncating the result to $m+1$ bits, producing a result of the form $0.1b'_1b'_2\cdots b'_m$. Thus, the approximation to the reciprocal is found for each entry $i$ in the table from:

$$R_i = \left\lfloor 2^{m+1} \times \left( \frac{1}{\hat{b} + 2^{-(k+1)}} + 2^{-(m+2)} \right) \right\rfloor / 2^{m+1} \tag{4.30}$$

where $\hat{b} = 1.b'_1b'_2\cdots b'_k$.

Das Sarma [72] has shown that the piecewise-constant approximation method for generating reciprocal look-up tables minimizes the maximum relative error in the final result. The maximum relative error in the reciprocal estimate $\epsilon_r$ for a $k$-bits-in $k$-bits-out reciprocal table is bounded by:

$$|\epsilon_r| = \left| R_0 - \frac{1}{b} \right| \leq 1.5 \times 2^{-(k+1)} \tag{4.31}$$

and thus the table guarantees a minimum precision of $k + 0.415$ bits. It is also shown that with $m = k + g$, where $g$ is the number of output guard bits, the maximum relative error is bounded by:

$$|\epsilon_r| \leq 2^{-(k+1)} \times \left( 1 + \frac{1}{2^{g+1}} \right) \tag{4.32}$$

Thus, the precision of a $k$-bits-in, $(k+g)$-bits-out reciprocal table for $k \geq 2$, $g \geq 0$, is at least $k + 1 - \log_2(1 + \frac{1}{2^{g+1}})$. As a result, generated tables with one, two, and three guard bits on the output are guaranteed precision of at least $k + 0.678$ bits, $k + 0.830$ bits, and $k + 0.912$ bits respectively.

In a more recent study, Das Sarma [73] describes bipartite reciprocal tables. These tables use separate table lookup of the positive and negative portions of a reciprocal value in borrow-save form, but with no additional multiply/accumulate operation required. Instead, it is assumed that the output of the table is used as input to a multiplier for subsequent operations. In this case, it is sufficient that the table produce output in a redundant form that is efficiently recoded for use by the multiplier. Thus, the required output rounding can be implemented in conjunction with multiplier recoding for little additional cost in terms of complexity or delay. This method is a form of linear interpolation on the reciprocal which allows for the use of significantly smaller tables. These bipartite tables are 2 to 4 times smaller than 4-9 bit conventional reciprocal tables. For 10-16 bit tables, bipartite tables can be 4 to more than 16 times smaller than conventional implementations.

### 4.5.2 Linear Approximations

Rather than using a constant approximation to the reciprocal, it is possible to use a linear or polynomial approximation. A polynomial approximation is expressed in the form of a truncated series:

$$P(b) = C_0 + C_1 b + C_2 b^2 + C_3 b^3 + \cdots \tag{4.33}$$

To get a first order or linear approximation, the coefficients $C_0$ and $C_1$ are stored in a look-up table, and a multiplication and an addition are required. As an example, a linear function is chosen such as

$$P(b) = -C_1 \times b + C_0 \tag{4.34}$$

in order to approximate $1/b$ [74]. The two coefficients $C_0$ and $C_1$ are read from two look-up tables, each using the $k$ most significant bits of $b$ to index into the tables and each returning $m$ bits. The total error of the linear approximation $\epsilon_{la}$ is the error due to indexing with only $k$ bits plus the truncation error due to only storing $m$ bit entries in each of the tables. It is shown in [74] that the coefficients should be chosen in order to minimize the maximium relative error, which is then

$$|\epsilon_{la}| \quad < \quad 2^{-(2k+3)} + 2^{-m} \tag{4.35}$$

Setting $m = 2k+3$ yields $|\epsilon_{la}| < 2^{-(2k+2)}$, and thus guaranteeing a precision of $2k+2$ bits. The total size required for the tables is $2^k \times m \times 2$ bits, and a $m \times m$ bit multiply/accumulate unit is required.

Schulte [75] proposes methods for selecting constant and linear approximations which minimize the absolute error of the final result for Newton-Raphson implementations. Minimizing the maximum relative error in an initial approximation minimizes the maximum relative error in the final result. However, the initial approximation which minimizes the maximum absolute error of the final result depends on the number of iterations of the algorithm. For constant and linear approximations, they present the tradeoff between $n$, the number of iterations, $k$, the number of bits used as input to the table, and the effects on the absolute error of the final result. In general, linear approximations guarantee more accuracy than constant approximations, but they require additional operations which may affect total delay and area.

Ito [74] proposes an improved initial approximation method similar to a linear approximation. A modified linear function

$$A_1 \times (2\hat{b} + 2^{-k} - b) + A_0 \tag{4.36}$$

is used instead of $-C_1 \times b + C_0$ for the approximation to $1/b$. In this way, appropriate table entries are chosen for $A_0$ and $A_1$. The total table size required is $2^k \times (3k+5)$ bits, and the method guarantees a precision of $2.5k$ bits.

### 4.5.3 Partial Product Arrays

An alternative to look-up tables for starting approximation is the use of partial product arrays [76],[77]. A partial product array can be derived which sums to an approximation of the reciprocal operation. Such an array is similar to the partial product array of a multiplier. As a result, an existing floating-point multiplier can be used to perform the summation.

A multiplier used to implement IEEE double precision numbers involves 53 rows of 53 elements per row. This entails a large array of 2809 elements. If Booth encoding is used in the multiplier, the bits of the partial products are recoded, decreasing the number of rows in the array by half. A Booth multiplier typically has only 27 rows in the partial product array. A multiplier sums all of these boolean elements to form the product. However, each boolean element of the array can be replaced by a generalized boolean element. By back-solving the partial product array, it can be determined what elements are required to generate the appropriate function approximation. These elements are chosen carefully to provide a high-precision approximation and reduce maximum error. This can be viewed as analogous to the choosing of coefficients for a polynomial approximation. In this way, a partial product array is generated which reuses existing hardware to generate a high-precision approximation.

In the case of the reciprocal function, a 17 digit approximation can be chosen which utilizes 18 columns of a 53 row array. Less than 20% of the array is actually used. However, the implementation is restricted by the height of the array, which is the number of rows. The additional hardware for the multiplier is 484 boolean elements. It has been shown that such a function will yield a minimum of 12.003 correct bits, with an average of 15.18 correct bits. An equivalent ROM look-up table that generates 12 bits would require about 39 times more area. If a Booth multiplier is used with only 27 rows, a different implementation can be used. This version uses only 175 boolean elements. It generates an average of 12.71 correct bits and 9.17 bits in the worst case. This is about 9 times smaller than an equivalent ROM look-up table.

## 4.6 Variable Latency Algorithms

Digit recurrence and very high radix algorithms all retire a fixed number of quotient bits in every iteration, while algorithms based on functional iteration retire an increasing number of bits every iteration. However, all of these algorithms complete in a fixed number of cycles. This section discusses several VLA methods for implementing dividers that compute results in a variable amount of time. The DEC Alpha

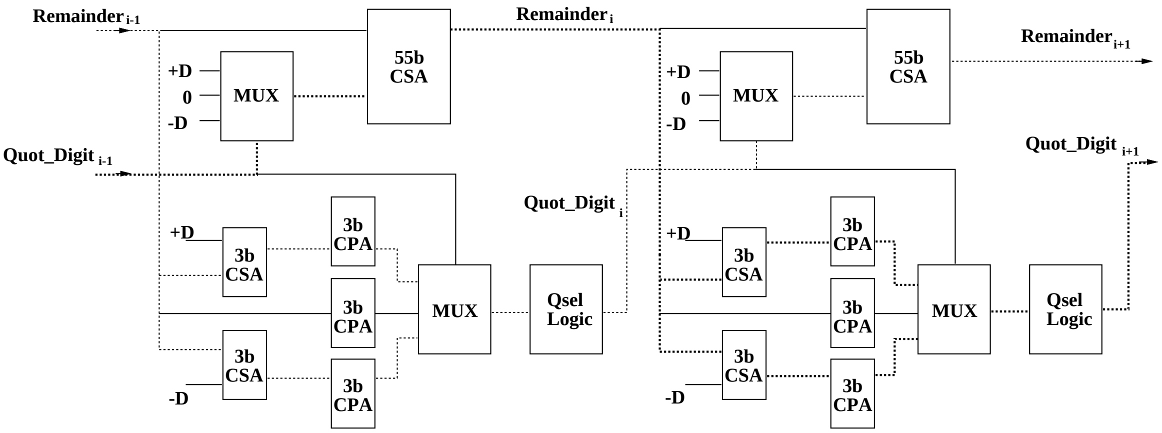


Figure 4.7: Two stages of self-timed divider

21164 [32] uses a simple normalizing non-restoring division algorithm, which is a predecessor to fixed-latency SRT division. Whenever a consecutive sequence of zeros or ones is detected in the partial remainder, a similar number of quotient bits are also set to zero, all within the same cycle. In [32], it is reported that an average of 2.4 quotient bits are retired per cycle using this algorithm.

This section presents three additional techniques for reducing the average latency of division computation. These techniques take advantage of the fact that the computation for certain operands can be completed sooner than others, or reused from a previous computation. Reducing the worst case latency of a divider requires that all computations made using the functional unit will complete in less than a certain amount of time. In some cases, modern processors are able to use the results from functional units as soon as they are available. Providing a result as soon as it is ready can therefore increase overall system performance.

### 4.6.1 Self-Timing

A recent SRT implementation returning quotients with variable latency is reported by Williams [17]. This implementation differs from conventional designs in that it uses self-timing and dynamic logic to increase the divider's performance. It comprises five cascaded radix-2 stages as shown in figure 4.7. Because it uses self-timing, no explicit registers are required to store the intermediate remainder. Accordingly, the critical path does not contain delay due to partial remainder register overhead. The adjacent stages overlap their computation by replicating the CPAs for each possible quotient digit from the previous stage. This allows each CPA to begin operation before the actual quotient digit arrives at a multiplexor to choose the correct branch. Two of the three CPAs in each stage are preceded by CSAs to speculatively compute a truncated version of $P_{i+1} - D$, $P_{i+1} + D$, and $P_{i+1}$. This overlapping of the execution between neighboring stages allows the delay through a stage to be the average, rather than the sum, of the propagation delays through the remainder and quotient-digit selection paths. This is illustrated in figure 4.7 by the two different drawn paths. The self-timing of the data path dynamically ensures that data always flow through the minimal critical path. This divider, implemented in a 1.2$\mu$m CMOS technology, is able to produce a 54-b result in 45 to 160ns, depending upon the particular data operands. The Hal SPARC V9 microprocessor, the Sparc64, also implements a version of this self-timed divider, producing IEEE double precision results in about 7 cycles [78].

### 4.6.2 Result Caches

In typical applications, the input operands for one calculation are often the same as those in a previous calculation. For example, in matrix inversion, each entry of the matrix must be divided by the determinant. By recognizing that such redundant behavior exists in applications, it is possible to take advantage of this fact and decrease the effective latency of computations.

Richardson [79] presents the technique of result caching as a means of decreasing the latency of otherwise high-latency operation, such as division. This technique exploits the redundant nature of certain computations by trading execution time for increased memory storage. Once a computation is calculated, it is stored in a *result cache*. When a targeted operation is issued by the processor, access to the result cache is initiated simultaneously. If the cache access results in a hit, then that result is used, and the operation is halted. If the access misses in the cache, then the operation writes the result into the cache upon completion. Various sized direct-mapped result caches were simulated which stored the results of double precision multiplies, divides, and square roots. The applications surveyed included several from the SPEC92 and Perfect Club benchmark suites. Significant reductions in latency were obtained in these benchmarks by the use of a result cache. However, the standard deviation of the resulting latencies across the applications was large. Chapter 6 investigates in more detail the use of caches to reduce average division latency.

### 4.6.3 Speculation of Quotient Digits

A method for implementing an SRT divider that retires a variable number of quotient bits every cycle is reported by Cortadella [80]. The goal of this algorithm is to use a simpler quotient-digit selection function than would normally be possible for a given radix by using fewer bits of the partial remainder and divisor than are specified for the given radix and quotient digit set. This new function does not give the correct quotient digit all of the time. When an incorrect speculation occurs, at least one iteration is needed to fix the incorrect quotient digit. However, if the probability of a correct digit is high enough, then the resulting lower cycle time due to the simple selection function offsets the increase in the number of iterations required.

Several different variations of this implementation were considered for different radices and base divider configurations. A radix-64 implementation was considered which could retire up to 6 bits per iteration. It was 30% faster in delay per bit than the fastest conventional implementation of the same base datapath, which was a radix-8 divider using segments. However, because of the duplication of the quotient-selection logic for speculation, it required about 44% more area than a conventional implementation. A radix-16 implementation, retiring a maximum of 4 bits per cycle, using the same radix-8 datapath was about 10% faster than a conventional radix-8 divider, with an area reduction of 25%.

## 4.7 Comparison

Five classes of division algorithms have been presented. In SRT division, to reduce division latency, more bits need to be retired in every cycle. However, directly increasing the radix can greatly increase the cycle time and the complexity of divisor multiple formation. The alternative is to stage lower radix stages together to form higher radix dividers, by simple staging or possibly overlapping one or both of the quotient selection logic and partial remainder computation hardware. All of these alternatives lead to an increase in area, complexity and potentially cycle time. Given the continued industry demand for ever-lower cycle times, any increase must be managed.

Higher degrees of redundancy in the quotient digit set and operand prescaling are the two primary means of further reducing the recurrence cycle time. These two methods can be combined for an even greater reduction. For radix-4 division with operand prescaling, an over-redundant digit set can reduce the number of partial remainder bits required for quotient selection from 6 to 4. Choosing a maximally redundant set and a radix-2 encoding for the partial remainder can reduce the number of partial remainder bits required for quotient selection down to 3. However, each of these enhancements requires additional area and complexity for the implementation that must be considered. Due to the cycle time constraints and area budgets of modern processors, SRT dividers are realistically limited to retiring fewer than 10 bits per cycle. However, a digit recurrence divider is an effective means of implementing a low cost division unit which operates in parallel with the rest of a processor.

Very high radix dividers are used when it is desired to retire more than 10 bits per cycle. The primary difference between the presented algorithms are the number and width of multipliers used. These have obvious effects on the latency of the algorithm and the size of the implementation. In the accurate quotient approximations and short-reciprocal algorithms, the next quotient digit is formed by a multiplication $P_h \times (1/Y_h)$ in each iteration. Because the Cyrix implementation only has one rectangular multiply/add unit, each iteration must perform this multiplication in series: first this product is formed as the next quotient digit, then the result is multiplied by $Y$ and subtracted from the current partial remainder to form the next partial remainder, for a total of two multiplications. The accurate quotient approximations method computes $Y' = Y \times (1/Y_h)$ once at the beginning in full precision, and is able to used the result in every iteration. Each iteration still requires two multiplications, but these can be performed in parallel: $P_h \times Y'$ to form the next partial remainder, and $P_h \times (1/Y_h)$ to form the next quotient digit.

The rounding and prescaling algorithm, on the other hand, does not require a separate multiplication to form the next quotient digit. Instead, by scaling *both* the dividend and divisor by the initial reciprocal approximation, the quotient-digit selection function can be implemented by simple rounding logic directly from the redundant partial remainder. Each iteration only requires one multiplication, reducing the area required compared with the accurate quotient approximations algorithm, and decreasing the latency compared with the Cyrix short-reciprocal algorithm. However, because both input operands are prescaled, the final remainder is not directly usable. If a remainder is required, it must be postscaled. Overall, the rounding and prescaling algorithm achieves the lowest latency and cycle time with a reasonable area, while the Cyrix short-reciprocal algorithm achieves the smallest area.

Self-timing, result caches, bit-skipping, and quotient digit speculation have been shown to be effective methods for reducing the average latency of division computation. A reciprocal cache is an efficient way to reduce the latency for division algorithms that first calculate a reciprocal. While reciprocal caches do require additional area, they require less than that required by much larger initial approximation look-up tables, while providing a better reduction in latency. Self-timed implementations use circuit techniques to generate results in variable time. The disadvantage of self-timing is the complexity in the clocking, circuits, and testing required for correct operation. Quotient digit speculation is one example of reducing the complexity of SRT quotient-digit selection logic for higher radix implementations.

Both the Newton-Raphson and series expansion iterations are effective means of implementing faster division. For both iterations, the cycle time is limited by two multiplications. In the Newton-Raphson iteration, these multiplications are dependent and must proceed in series, while in the series expansion, these multiplications

| Algorithm | Iteration Time | Latency (cycles) | Approximate Area |
|---|---|---|---|
| SRT | Table + MUX + sub | $\lceil \frac{n}{r} \rceil$ + scale | Qsel table + CSA |
| Newton-Raphson | 2 serial mults | $(2\lceil \log_2 \frac{n}{i} \rceil + 1)t_{mul} + 1$ | multiplier + table + control |
| series expansion | 1 mult[1] | $(\lceil \log_2 \frac{n}{i} \rceil + 1)t_{mul} + 2,\ t_{mul} > 1$<br>$2\lceil \log_2 \frac{n}{i} \rceil + 3,\ t_{mul} = 1$ | multiplier + table + control |
| accurate quotient approx | 1 mult | $(\lceil \frac{n}{i} \rceil + 1)t_{mul}$ | 3 multipliers + table + control |
| short reciprocal | 2 serial mults | $2\lceil \frac{n}{i} \rceil t_{mul} + 1$ | short multiplier + table + control |
| round/prescale | 1 mult | $(\lceil \frac{n}{i} \rceil + 2)t_{mul} + 1$ | multiplier + table + control |

Table 4.1: Summary of algorithms

may proceed in parallel. To reduce the latency of the iterations, an accurate initial approximation can be used. This introduces a tradeoff between additional chip area for a look-up table and the latency of the divider. An alternative to a look-up table is the use of a partial product array, possibly by reusing an existing floating-point multiplier. Instead of requiring additional area, such an implementation could increase the cycle time through the multiplier. The primary advantage of division by functional iteration is the quadratic convergence to the quotient. Functional iteration does not readily provide a final remainder. Accordingly, correct rounding for functional iteration implementations is difficult. When a latency is required lower than can be provided by an SRT implementation, functional iteration is currently the primary alternative. It provides a way to achieve lower latencies without seriously impacting the cycle time of the processor and without a large amount of additional hardware.

A summary of the algorithms from these classes is shown in table 4.1.

[1]When a pipelined multiplier is used, the delay per iteration is $t_{mul}$, but one cycle is required at the end of the iterations to drain the pipeline.

In this table, $n$ is the number of bits in the input operands, $i$ is the number of bits of accuracy from an initial approximation, and $t_{mul}$ is the latency of the fused multiply/accumulate unit in cycles. None of the latencies include additional time required for rounding or normalization.

Table 4.2 provides a rough evaluation of the effects of algorithm, operand length, width of initial approximation, and multiplier latency on division latency. All operands are IEEE double precision mantissas, with $n = 53$. It is assumed that table look-ups for initial approximations require 1 cycle. The SRT latencies are separate from the others in that they do not depend on multiplier latency, and they are only a function of the radix of the algorithm for the purpose of this table. For the multiplication-based division algorithms, latencies are shown for multiplier/accumulate latencies of 1, 2 and 3 cycles. Additionally, latencies are shown for pipelined as well as unpipelined multipliers. A pipelined unit can begin a new computation every cycle, while an unpipelined unit can only begin after the previous computation has completed.

From table 4.2, the advanced version of the accurate quotient approximations algorithm provides the lowest latency. However, the area requirement for this implementation is tremendous, as it requires at least a 736K bits look-up table and three multipliers. For realistic implementations, with $t_{mul} = 2$ or $t_{mul} = 3$ and $i = 8$, the lowest latency is achieved through a series expansion implementation. However, all of the multiplication-based implementations are very close in performance. This analysis shows the extreme dependence of division latency on the multiplier's latency and throughput. A factor of three difference in multiplier latency can result in nearly a factor of three difference in division latency for several of the implementations.

It is difficult for an SRT implementation to perform better than the multiplication-based implementations due to infeasibility of high radix at similar cycle times. However, through the use of scaling and higher redundancy, it may be possible to implement a radix-256 SRT divider that is competitive with the multiplication-based schemes in terms of cycle time and latency. The use of variable latency techniques, such as self-timing, can provide further means for matching the performance of the multiplication-based schemes, without the difficulty in rounding that is intrinsic to the functional iteration implementations.

| Algorithm | Radix | Latency (cycles) |
|---|---|---|
| SRT | 4 | 27 |
| | 8 | 18 |
| | 16 | 14 |
| | 256 | 7 |

| Algorithm | Pipelined | Initial Approx (bits) | Latency (cycles) | | |
|---|---|---|---|---|---|
| | | | $t_{mul} = 1$ | $t_{mul} = 2$ | $t_{mul} = 3$ |
| Newton-Raphson | either | $i = 8$ | 8 | 15 | 22 |
| | either | $i = 16$ | 6 | 11 | 16 |
| series expansion | no | $i = 8$ | 9 | 17 | 25 |
| | no | $i = 16$ | 7 | 13 | 19 |
| | yes | $i = 8$ | 9 | 10 | 14 |
| | yes | $i = 16$ | 7 | 8 | 11 |
| accurate quotient approximations | either | $i = 8$ (basic) | 8 | 16 | 24 |
| | either | $i = 16$ (basic) | 5 | 10 | 15 |
| | either | $i = 13$ and $t = 2$ (adv) | 3 | 6 | 9 |
| short reciprocal | either | $i = 8$ | 15 | 29 | |
| | either | $i = 16$ | 9 | 17 | |
| round/prescale | no | $i = 8$ | 10 | 19 | 28 |
| | no | $i = 16$ | 7 | 13 | 19 |
| | yes | $i = 8$ | 10 | 18 | 26 |
| | yes | $i = 16$ | 7 | 10 | 14 |

Table 4.2: Latencies for different configurations

## 4.8 Summary

In this chapter, the five major classes of division algorithms have been presented. The classes are determined by the differences in the fundamental operations used in the hardware implementations of the algorithms. The simplest and most common class found in the majority of modern processors that have hardware division support is digit recurrence, specifically SRT. Recent commercial SRT implementations have included radix 2, radix 4, and radix 8. These implementations have been chosen in part because they operate in parallel with the rest of the floating point hardware and do not create contention for other units. Additionally, for small radices, it has been possible to meet the tight cycle-time requirements of high performance processors without requiring large amounts of die area. The disadvantage of these SRT implementations is their relatively high latency, as they only retire 1-3 bits of result per cycle. As processors continue to seek to provide an ever-increasing amount of system performance, it becomes necessary to reduce the latency of all functional units, including division.

An alternative to SRT implementations is functional iteration, with the series expansion implementation the most common form. The latency of this implementation is directly coupled to the latency and throughput of the multiplier and the accuracy of the initial approximation. The analysis presented shows that a series expansion implementation provides the lowest latency for reasonable areas and multiplier latencies. Latency is reduced in this implementation through the use of a reordering of the operations in the Newton-Raphson iteration in order to exploit single-cycle throughput of pipelined multipliers. This has the disadvantage that the multiplier is completely occupied until completion of the division iterations. In contrast, the Newton-Raphson iteration itself, with its serial multiplications, has a higher latency. However, if a pipelined multiplier is used throughout the iterations, more than one division operation can proceed in parallel. For implementations with high division throughput requirements, both iterations provide a means for trading latency for throughput.

If minimizing area is of primary importance, then such an implementation typically shares an existing floating-point multiplier. This has the effect of creating additional contention for the multiplier, although this effect is minimal in many applications. An alternative is to provide an additional multiplier, dedicated for division. This can be an acceptable tradeoff if a large quantity of area is available and maximum performance is desired for highly parallel division/multiplication applications, such as graphics and 3D rendering applications. The main disadvantage with functional iteration is the lack of remainder and the corresponding difficulty in rounding.

Very high radix algorithms are an attractive means of achieving low latency while also providing a true remainder. The only commercial implementation of a very high radix algorithm is the Cyrix short-reciprocal unit. This implementation makes efficient use of a single rectangular multiply/add unit to achieve lower latency than most SRT implementations while still providing a remainder. Further reductions in latency could be possible by using a full-width multiplier, as in the rounding and prescaling algorithm.

The Hal Sparc64 self-timed divider and the DEC Alpha 21164 divider are the only commercial implementations that generate quotients with variable latency depending upon the input operands. Reciprocal caches have been shown to be an effective means of reducing the latency of division for implementations that generate a reciprocal. Quotient digit speculation is also a reasonable method for reducing SRT division latency.

# Chapter 5

# Faster SRT Dividers

## 5.1 Introduction

The most common division implementation in commercial microprocessors is SRT division. There are many performance and area tradeoffs when designing an SRT divider. One metric for comparison of different designs is the minimum required truncations of the divisor and partial remainder for quotient-digit selection. Atkins [44] and Robertson [42] provide such analyses of the divisor and partial remainder precisions required for quotient-digit selection. Burgess and Williams [81] present in more detail allowable truncations for divisors and both carry-save and borrow-save partial remainders. However, a more detailed comparison of quotient-digit selection complexity between different designs requires more information than input precision.

This chapter analyzes in detail the effects of algorithm radix, redundancy, divisor and partial remainder precision, and truncation error on the complexity of the resulting table. Complexity is measured by the number of product terms in the final logic equations, and the delay and area of standard-cell implementations of the tables. These metrics are obtained by an automated design flow using the specifications for the quotient-digit selection table as input, a Gray-coded PLA as an intermediate representation, and an LSI Logic 500K standard-cell implementation as the output. This chapter also examines the effects of additional techniques such as table-folding and longer external carry-assimilating adders on table complexity. Using the methodology presented, it is possible to automatically generate optimized high radix quotient-digit selection tables.

## 5.2 SRT Division

As discussed in chapter 3, SRT dividers comprise the following steps in each iteration of the recurrence:

- Determine next quotient-digit $q_{j+1}$ by the quotient-digit selection function
- Generate the product $q_{j+1} \times divisor$
- Subtract $q_{j+1} \times divisor$ from $r \times P_j$ to form the next partial remainder

Each of these components can contribute to the overall cost and performance of the algorithm. To reduce the time for partial remainder computation, intermediate partial remainders are often stored in a redundant representation, either carry-save or signed digit form. Then, the partial remainder computation requires only a full adder delay, rather than a full width carry-propagate addition. In a standard SRT implementation, the largest contributor to cycle time is the first step, quotient-digit selection. The rest of this chapter is concerned with the quotient-digit selection component.

## 5.3 Implementing SRT Tables

### 5.3.1 Divisor and Partial Remainder Estimates

To reduce the size and complexity of the quotient-digit selection table for a given choice of radix $r$ and maximum allowable quotient digit $a$, it is desirable to use as input to the table estimates of the divisor and shifted partial remainder which have fewer bits than the true values. In IEEE floating point, the input operands are in the range $1 \leq D < 2$. Thus, a leading integer one can be assumed for all divisors, and the table only requires fractional divisor bits to make a quotient-digit selection. The shifted partial remainder, though, requires both integer and fractional bits as inputs to the table. The shifted partial remainder $rP_j$ and divisor $d$ can be approximated by estimates $r\hat{P}_j$ and $\hat{d}$ using the $c$ most significant bits of $rP_j$ and the $\delta$ most significant bits of $d$. The $c$ bits in the truncated estimate $r\hat{P}_j$ can be divided into $i$ integer bits and $f$ fractional bits, such that $c = i + f$. The table can take as input the partial remainder estimate directly in redundant form, or it can use the output of a short carry-assimilating adder that converts the redundant partial remainder estimate to a nonredundant representation. The use of an external short adder reduces the complexity of the table implementation, as the number of partial remainder input bits are halved. However, the delay of the quotient-digit selection function increases by the delay of the adder.

It is not possible to determine the optimal choices of $\delta$ and $f$ analytically, as several factors are involved in making these choices. However, it is possible to determine a lower bound on $\delta$ using the continuity condition and the fact that the next partial remainder must remain bounded:

$$2^{-\delta} \leq \frac{2\rho - 1}{a - \rho} \tag{5.1}$$

$$\delta \geq \left\lceil -\log_2 \frac{2\rho - 1}{a - \rho} \right\rceil \tag{5.2}$$

Because the divisor has a leading one, only the leading $b = \delta - 1$ fractional bits are required as input to the table. The next quotient-digit can then be selected by using these estimates to index into a $2^{b+c}$ entry lookup table, implemented either as a PLA or random logic.

Assuming a nonredundant two's complement partial remainder, the estimates have nonnegative truncation errors $\epsilon_d$ and $\epsilon_p$ for the divisor and shifted partial remainder estimates respectively, where

$$\epsilon_d \leq 2^{-b} - 2^{-n+1} \approx 2^{-b} \tag{5.3}$$

$$\epsilon_p \leq 2^{-f} - 2^{-n+1} \approx 2^{-f} \tag{5.4}$$

Thus, the maximum truncation error for both the divisor and the nonredundant shifted partial remainder estimates is strictly less than 1 ulp.

For a redundant two's complement partial remainder, the truncation error depends upon the representation. For a carry-save representation, the sum and carry estimates each has nonnegative truncation error $\epsilon_p$, assuming that both the sum and carry estimates are represented by the $c$ most significant bits of their true values. The resulting estimate $r\hat{P}_{j(cs)}$ has truncation error

$$\epsilon_{p(cs)} \leq 2 \times (2^{-f} - 2^{-n+1}) \approx 2^{-f+1} \tag{5.5}$$

Thus, the maximum truncation error for an estimate of a carry-save shifted partial remainder is strictly less than 2 ulps.

From this discussion, the number of integer bits $i$ in $r\hat{P}_j$ can be determined analytically. Using the general recurrence for SRT division, the maximum shifted partial remainder is given by

$$rP_{j(max)} = r\rho d_{max} \tag{5.6}$$

For IEEE operands,

$$d_{max} = 2 - 2^{-n+1} \tag{5.7}$$

For a carry-save two's complement representation of the partial remainder, the truncation error is always nonnegative, and therefore the maximum estimate of the partial remainder is

$$r\hat{P}_{j(max)} = \left\lfloor r \times \rho \times (2 - 2^{-n+1}) \times 2^f \right\rfloor / 2^f \tag{5.8}$$

The minimum estimate of the partial remainder is

$$r\hat{P}_{j(min)} = \left\lceil -(r \times \rho \times (2 - 2^{-n+1}) - \epsilon_{p(cs)}) \times 2^f \right\rceil / 2^f \tag{5.9}$$

Accordingly, $i$ can be determined from

$$r\hat{P}_{j(max)} - r\hat{P}_{j(min)} + 1 \leq 2^i \tag{5.10}$$

$$i \geq \left\lceil \log_2(r\hat{P}_{j(max)} - r\hat{P}_{j(min)} + 1) \right\rceil \tag{5.11}$$

### 5.3.2 Uncertainty Regions

By using a redundant quotient-digit set, it is possible to correctly choose the next quotient-digit even when using the truncated estimates $r\hat{P}_j$ and $\hat{d}$. Due to the truncation error in the estimates, each entry in the quotient-digit selection table has an uncertainty region associated with it. For each entry, it is necessary for all combinations of all possible values represented by the estimates $r\hat{P}_j$ and $\hat{d}$ to lie in the same selection interval. For a carry-save representation of the shifted partial remainder, this involves calculating the maximum and minimum ratios of the shifted partial remainder and divisor, and ensuring that these ratios both lie in the same selection interval:

$$ratio_{max} = \begin{cases} \frac{r\hat{P}_j + \epsilon_{p(cs)}}{\hat{d}} & \text{if } P_j \geq 0 \\ \frac{r\hat{P}_j}{\hat{d}} & \text{if } P_j < 0 \end{cases} \tag{5.12}$$

$$ratio_{min} = \begin{cases} \frac{r\hat{P}_j}{\hat{d} + \epsilon_d} & \text{if } P_j \geq 0 \\ \frac{r\hat{P}_j + \epsilon_{p(cs)}}{\hat{d} + \epsilon_d} & \text{if } P_j < 0 \end{cases} \tag{5.13}$$

If an uncertainty region is too large, the maximum and minimum ratios may span more than one selection interval, requiring one table entry to return more than one quotient-digit. This would signify that the estimate of the divisor and/or the shifted partial remainder has too much truncation error. Figure 5.1 shows several uncertainty regions in a radix 4 P-D plot. Each uncertainty region is represented by a rectangle whose height and width is a function of the divisor and partial remainder truncation errors. The value of $ratio_{max}$ corresponds to the upper left corner of the rectangle, while $ratio_{min}$ corresponds to the lower right corner. In this figure, the four valid uncertainty regions include a portion of an overlap region. Further, the lower right uncertainty region is fully contained within an overlap region, allowing the entry corresponding to that uncertainty region to take on the quotient-digits of either 0 or 1. The other three valid uncertainty regions may take on only a single quotient-digit. The upper left uncertainty region spans more than an entire overlap region, signifying that the corresponding table entry, and as a result the entire table, is not valid. To determine the valid values of $b$ and $f$ for a given $r$ and $a$, it is necessary to calculate the uncertainty regions for all $2^{b+i+f}$ entries in the table. If all uncertainty regions

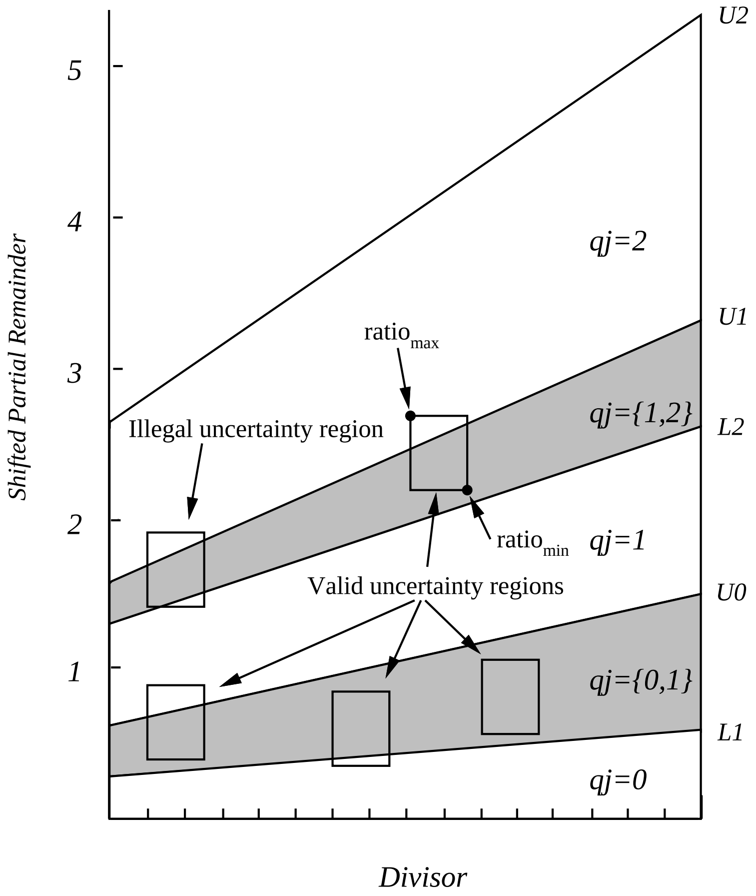


Figure 5.1: Uncertainty regions due to divisor and partial remainder estimates

are valid for given choices of $b$, $i$, and $f$, then they are valid choices.

### 5.3.3 Reducing Table Complexity

The size of the table implementation can be reduced nearly in half by folding the table entries as suggested in Fandrianto [82]. Folding involves the conversion of the two's complement representation of $r\hat{P}_j$ to signed-magnitude, allowing the same table entries to be used for both positive and negative values of $r\hat{P}_j$. This reduction does not come for free. First, it requires additional logic outside of the table, such as a row of XOR gates, to perform the representation conversion, adding external delay to the quotient digit selection process. Second, it may place further restrictions on the table design process. When a carry-save representation is used for $rP_j$ and a truncated estimate $r\hat{P}_j$ is used to consult the table, the truncation error is always nonnegative, resulting in an asymmetrical table. To guarantee the symmetry required for folding, additional terms must be added to the table, resulting in a less than optimal implementation.

A complexity-reducing technique proposed in this study is to minimize $\epsilon_p$. When using an external carry-assimilating adder for a truncated two's complement carry-save partial remainder estimate, the maximum error $\epsilon_{p(cs)}$ is approximately $2^{-f+1}$. This error can be further reduced by using $g$ fractional bits of redundant partial remainder as input to the external adder, where $g > f$, but only using the most significant $f$ fractional output bits of the adder as input to the table. The maximum error in the output of the adder is

$$\epsilon_{p(adder)} = 2^{-g} + 2^{-g} - 2^{-n+1} \tag{5.14}$$

Then, by using $f$ bits of the adder output, the maximum error for the input to the table is

$$\begin{aligned} \epsilon_{p(cs)} &= 2^{-f} - 2^{-g} + \epsilon_{p(adder)} \\ &= 2^{-f} + 2^{-g} - 2^{-n+1} \approx 2^{-f} + 2^{-g} \end{aligned} \tag{5.15}$$

For the case $g = f$, the error remains approximately $2^{-f+1}$. However, by increasing $g$, the error $\epsilon_{p(cs)}$ is reduced, converging towards the error for a nonredundant partial remainder which is approximately $2^{-f}$. Reducing the truncation error $\epsilon_{p(cs)}$ decreases the height of the uncertainty region in the PD diagram. This has the effect of allowing more of the entries' uncertainty regions to fully fit within overlap regions, increasing the flexibility in the logic minimization process, and ultimately reducing the complexity of the final table. A block diagram illustrating the various components of an SRT divider is shown in figure 5.2.

## 5.4 Experimental Methodology

The focus of this chapter is to quantitatively study the relationship between the parameters of the tables $r$, $a$, $b$, $i$, $f$, and $g$, and the complexity of the tables. Complexity is measured by the number of product terms in the switching equations for the table and by the delay and area of combinational logic implementations of these equations. A carry-save two's complement representation is used for the partial remainder in all tables. The design flow used to automatically generate quotient-digit selection tables in random logic is shown in figure 5.3.

### 5.4.1 TableGen

The program *TableGen* performs the analytical aspects of the quotient-digit selection table design. This program takes the table parameters as input and produces the unminimized PLA entries necessary to implement the table. First, all of the uncertainty regions for all entries in the table are computed. *TableGen* determines whether or not the choice of input parameters results in a valid table design. If the table is valid, it then computes the allowable quotient-digits for each entry in the table, based upon the size of the uncertainty region. The allowable quotient-digits are then written in PLA form for all $2^{i+b+f}$ possible shifted partial remainder and divisor pairs.

To allow for the greatest reduction in the complexity of the table implementations, we use a Gray code to encode an entry's allowable quotient-digits. In a Gray encoding, neighboring values only differ in one bit position [83]. This allows for the efficient representation of multiple allowable quotient-digits in the PLA output, while still

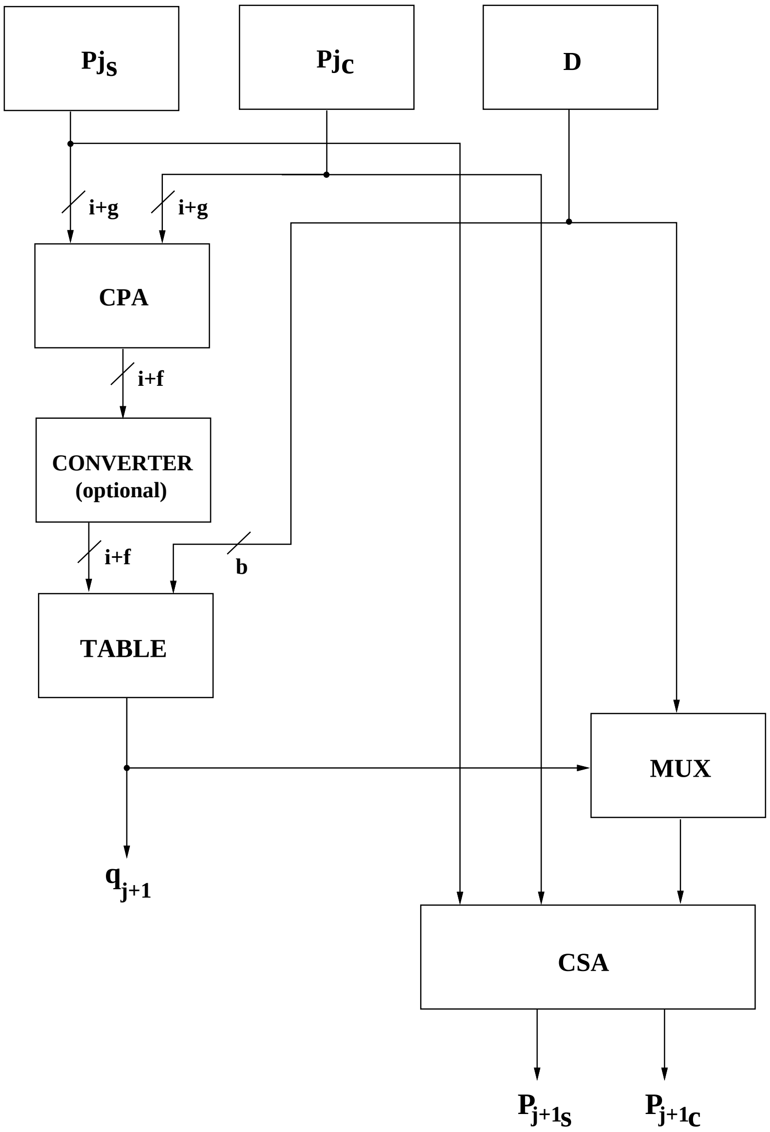


Figure 5.2: Components of an SRT divider

**Table Spec** $\xrightarrow[\textit{TableGen}]{}$ **PLA Entries** $\xrightarrow[\textit{Espresso}]{}$ **Logic Equations** $\xrightarrow[\textit{Synopsys}]{}$ **Standard Cell Netlist**

Figure 5.3: Design flow

| Allowable Digits | Encoding |
|---|---|
| 0 | 00 |
| 1 | 01 |
| 0 or 1 | 0x |
| 2 | 11 |
| 1 or 2 | x1 |
| 3 | 10 |
| 2 or 3 | 1x |

Table 5.1: Gray encoding for maximally redundant radix 4

only requiring $\lceil \log_2 r \rceil$ bits. The Gray-coding of the digits is recommended to ensure that given a choice between two allowable quotient-digits in an overlap region, the optimal choice can be automatically determined that results in the least complex logic equations.

Accordingly, there are $\lceil \log_2 r \rceil$ outputs of the table which are the bits of the encoded quotient-digit. Table entries where $ratio_{min}$ and $ratio_{max}$ are both greater than $\rho \times r$ are unreachable entries. Thus, their outputs are set to *don't care*. An example of Gray-coding for $r = 4$ and $a = 3$ is shown in table 5.1. In this example, a value of $x$ implies a *don't care*. Because the table stores digits in encoded form, all tables in this study require an explicit decoder to recover the true quotient-digit and select the appropriate divisor multiple. This results in the addition of a decoder delay into the critical path. However, since all tables in this study use the same encoding, this is a constant delay that only grows as the log of the radix. Alternatively, the quotient-digits could be stored in an unencoded form, removing the need for the decoder. Optimal logic minimization becomes a much more difficult problem for unencoded digits.

Espresso is used to perform logic minimization on the output PLA. This produces a minimized PLA and the logic equations representing the PLA in sum of products form. The number of product terms in these expressions is one metric for the complexity of the tables. To verify the correctness of the tables, an SRT divider was simulated using a DEC Alpha 3000/500 with the minimized logic equations as the quotient-digit selection function. After each table was generated, the equations were incorporated into the simulator. Several thousand random and directed IEEE double precision vectors were used as input to the simulator, and the computed quotients for each table were compared with the computed results from the Alpha's internal FPU.

### 5.4.2 Table Synthesis

To quantify the performance and area of random logic implementations of the tables, each table was synthesized using a standard-cell library. The Synopsys Design Compiler [84] was used to map the logic equations describing each table to an LSI Logic 500K 0.5$\mu$m standard-cell library [85]. In the mapping stage, low flattening and medium mapping effort options were used. However, area was always sacrificed to reduce the latency. In order to minimize the true critical path of the tables, the input constraints included the late arrival time of all partial remainder inputs due to an external carry-assimilating adder.

Area and delay estimates were obtained from a Design Compiler pre-layout report. Each delay includes intrinsic gate delay, estimated interconnect wire delay, and the input load of subsequent gates. All delays were measured using nominal conditions. Each area measurement includes both cell and estimated routing area. The delay and area are reported relative to those for a base radix 4 table, which is Gray-coded with $i = 3$, $f = 3$, $g = 3$, and $d = 3$. The base table requires 43 cells and has a delay of 1.47ns.

The results are presented in tables 5.2 through 5.6. The complexity of the tables is measured by the number of product terms, the relative delay, and the relative area for each configuration. The terms result contains both the number of product terms for each output of the table, as well as the total number of terms in the table. For a given radix $r$, there are exactly $n_{out} = \log_2 r$ outputs of the table. Accordingly, this column first lists the number of terms in each of the $n_{out}$ outputs. Because there is usually some internal sharing of product terms, the last number, which is the total number of unique terms required to implement the table, is typically less than the sum of the terms for the individual outputs. The reported delay is the worst-case delay of any of the $n_{out}$ outputs, typically corresponding to the output which has the highest number of product terms.

## 5.5 Results

### 5.5.1 Same Radix Tradeoffs

| Description | a | b | i | f | g | Terms | Relative Table Delay | Relative Table Area |
|---|---|---|---|---|---|---|---|---|
| Baseline | 2 | 3 | 4 | 3 | 3 | 19,8,25 | 1.00 | 1.00 |
| | 2 | 3 | 4 | 3 | 4 | 17,8,23 | 0.88 | 0.93 |
| | 2 | 3 | 4 | 3 | 5 | 17,8,23 | 0.84 | 0.91 |
| | 2 | 3 | 4 | 2 | 52 | 14,6,18 | 0.77 | 0.77 |
| Folded | 2 | 3 | 4 | 3 | 4 | 13,5,17 | 0.79 (1.30) | 0.65 (0.85) |
| Folded + t-bit conv | 2 | 3 | 4 | 3 | 4 | 12,4,16 | 0.71 (1.58) | 0.57 (0.83) |
| Line Encode | 2 | 3 | 4 | 3 | 4 | 17,7,23 | 0.84 | 0.86 |
| Choose Highest | 2 | 3 | 4 | 3 | 4 | 22,13,33 | 1.05 | 1.23 |
| Max Red | 3 | 2 | 5 | 1 | 1 | 6,9,14 | 0.80 | 0.54 |
| | 3 | 2 | 5 | 1 | 52 | 3,6,9 | 0.63 | 0.30 |

Note: b, i, f, and g are the bit field sizes shown in figure 5.2.

Table 5.2: Radix 4 Tradeoffs

Table 5.2 shows the results for various radix 4 configurations. The parameters varied in this table are 1) the number of $g$ bits vs $f$ bits, 2) folding, 3) method of choosing values in the overlap regions, and 4) the amount of redundancy. The first entries examine the effects of using $g$ bits of the redundant partial remainder into the short CPA outside of the table, while only using $f$ bits of the adder output as input to the table. Simply extending the CPA by one bit reduces the delay by 12% and area by 7%. Extending the CPA by two bits reduces the delay by 16% and area by 9%. In the limit, where $g = n$ and a full-mantissa carry propagate addition is required, the delay and area are both reduced by 23%. Increasing the width of the CPA by as little as one or two bits can reduce the complexity of the table.

The next two entries demonstrate the effects of using folded tables. For both tables, it is assumed that $f = 3$ and $g = 4$, which matches the format of the table suggested in Fandrianto [82]. The use of only a two's complement to sign-magnitude converter yields the first folded table, which achieves an additional delay reduction of 10% and an additional area reduction of 30% over the $f = 3$ $g = 4$ table without folding. This introduces a serial delay of an XOR gate external to the delay of the table. When the sign-magnitude converter is combined with a "t-bit converter", which further constrains the range of input values to the table, the table delay is reduced relative to the simple folded table by an additional 10%, and the area is reduced by an additional 12%. This converter introduces the serial delay of an OR gate external to the table. These results show that folding can reduce the size and delay of the table. However, the delay of the required external gates must be considered for the overall design. If the sum of the XOR and OR gate delays is less than 29% of the base table delay, table folding can result in a net decrease in delay.

The parenthesized values for these two entries represent the delay and area of table when the required external gates are included in the calculation. When the XOR gates are implemented in this standard cell technology, they account for about 66% of the unfolded table delay. This is in part due to the more complex functionality of the XOR gate relative to the other utilized gates. It is also due to the high fanout of the sign of the partial remainder which must drive the select signals of 5 XOR gates. The total delay of the folded table with the row of XOR gates is 49% greater than the unfolded table, but the area is reduced by 8%. When the "t-bit converter" is implemented through the addition of a row of OR gates driven my the MSB of the ones-complemented partial remainder, the total delay increases by 80% over the unfolded table, but the area is reduced by 10%. Thus, in this technology, the use of either table-folding technique results in higher overall delay, as the added delay

| a | b | i | f | g | Terms | Relative Table Delay | Relative Table Area |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 4 | 0 | 0 | 3 | 0.35 | 0.07 |

Table 5.3: Radix 2

from the external gates is greater than the corresponding reduction in table delay. However, these techniques do allow for a small reduction in total area.

Different encodings of the quotient-digits can change the complexity of the tables. The lower bound for delay and area of the table is achieved when each boundary between two consecutive quotient-digits is individually encoded. The recovery of the unencoded quotient-digit may require a large external decoder. When using such a "line" encoding scheme, again with $f = 3$ and $g = 4$, the delay and area are reduced by 5% and 8% respectively relative to the base Gray-coded table, also with $f = 3$ and $g = 4$. However, the external decoder delay grows linearly with increasing $a$ for line encoding, while only growing as the log of $a$ for Gray-coding. Another common encoding scheme always uses the highest digit whenever a choice is available between two consecutive quotient-digits. This is represented in the table as "choose highest" encoding. While simplifying the table generating process, this method increases the resulting table delay by 19% and area by 32% over the base $f = 3$ $g = 4$ table. Thus, this study shows that Gray-coding of the quotient-digits achieves delays and areas approaching the lower bound of line encoding, while requiring less complex external decoders.

The redundancy of the digit set has an impact on table complexity. The final entries in the table are for maximally redundant radix 4 tables, with $a = 3$. For an implementation with $f = g = 1$, the delay and area are reduced by 20% and 46% respectively. When $g$ increases to $n = 52$, requiring a full mantissa-width CPA, the delay is further reduced by 21% and the area by 44%. If the hardware is available to generate the 3x divisor multiple, the iteration time can be reduced by over 20%, due to the reduction in table complexity and length of the external short CPA.

| a | b | i | f | g | Terms | Relative Table Delay | Relative Table Area |
|---|---|---|---|---|---|---|---|
| 4 | 7 | 5 | 4 | 4 | 137,59,114,292 | 1.85 | 10.2 |
| | 7 | 5 | 4 | 5 | 111,48,94,240 | 1.76 | 8.80 |
| | 6 | 5 | 5 | 5 | 110,50,94,240 | 1.70 | 8.85 |
| | 5 | 5 | 6 | 6 | 104,50,85,221 | 1.67 | 8.21 |
| 5 | 5 | 5 | 3 | 3 | 42,19,69,122 | 1.45 | 5.21 |
| | 4 | 5 | 4 | 4 | 35,14,57,103 | 1.47 | 4.05 |
| 6 | 5 | 5 | 2 | 2 | 26,35,39,92 | 1.46 | 4.82 |
| | 4 | 5 | 3 | 3 | 24,33,38,88 | 1.46 | 4.46 |
| | 3 | 5 | 5 | 5 | 27,29,33,78 | 1.36 | 4.03 |
| 7 | 6 | 6 | 1 | 1 | 16,23,43,76 | 1.46 | 3.90 |
| | 3 | 6 | 2 | 2 | 15,21,35,64 | 1.41 | 3.53 |

Note: b, i, f, and g are the bit field sizes shown in figure 5.2.
Delay and Area are relative to Baseline in table 5.2.

Table 5.4: Radix 8

Table 5.3 shows the complexity of a basic radix 2 table. This table can be implemented by a single three-input gate, as it only has 3 PLA terms, which can be contrasted with the 25 terms in the baseline radix 4 table. The resulting delay is 65% less than the base radix 4 table, while 93% less area is required. Accordingly, the radix 2 table is 2.86 times faster than the base radix 4 table, and 2.40 times faster than a $g = 5$ radix 4 table.

### 5.5.2 Higher Radix

Tables 5.4, 5.5, and 5.6 show the complexity for tables that directly implement radix 8, 16, and 32 respectively. The allowable choices of $i$, $b$, and $f$ determined in this study correspond with the results presented in [81] for radix 8 and 16. In this study, the allowed operand truncations are extended to radix 32. For radix 16 and radix 32, the minimally redundant configurations required 20 or more inputs to the table. Due to computational constraints, table optimization was limited to configurations containing fewer than 20 inputs. Those configurations where optimization was infeasible are denoted with a dagger in the tables.

| a | b | i | f | g | Terms | Relative Table Delay | Relative Table Area |
|---|---|---|---|---|---|---|---|
| 8 | 11 | 6 | 5 | 5 | † | † | † |
| | 8 | 6 | 6 | 6 | † | † | † |
| | 7 | 6 | 8 | 8 | † | † | † |
| 9 | 7 | 6 | 4 | 4 | 198,98,176,481,871 | 2.56 | 32.9 |
| | 6 | 6 | 5 | 5 | 170,81,154,410,745 | 2.50 | 29.0 |
| 10 | 7 | 6 | 3 | 3 | 120,58,231,280,616 | 2.37 | 24.8 |
| | 6 | 6 | 4 | 4 | 105,57,191,240,530 | 2.32 | 21.4 |
| | 5 | 6 | 5 | 5 | 96,49,183,227,497 | 2.24 | 21.1 |
| 11 | 6 | 6 | 3 | 3 | 80,44,159,258,484 | 2.21 | 21.1 |
| | 5 | 6 | 4 | 4 | 68,39,142,208,418 | 2.10 | 18.9 |
| 12 | 7 | 6 | 2 | 2 | 79,138,144,228,481 | 2.17 | 25.0 |
| | 6 | 6 | 3 | 3 | 66,112,119,172,373 | 2.09 | 19.1 |
| 13 | 6 | 6 | 2 | 2 | 60,105,105,207,393 | 2.14 | 20.4 |
| | 5 | 6 | 3 | 3 | 62,104,99,186,344 | 2.07 | 18.5 |
| 14 | 5 | 6 | 2 | 2 | 48,101,136,169,383 | 2.14 | 20.2 |
| | 4 | 6 | 6 | 6 | 54,92,110,139,310 | 2.06 | 16.5 |
| 15 | 9 | 7 | 1 | 1 | 47,76,135,193,383 | 2.16 | 19.2 |
| | 5 | 7 | 2 | 2 | 39,69,99,136,281 | 2.03 | 15.4 |
| | 4 | 7 | 3 | 3 | 42,61,93,125,261 | 1.94 | 14.8 |

Note: b, i, f, and g are the bit field sizes shown in figure 5.2.
Delay and Area are relative to Baseline in table 5.2.

Table 5.5: Radix 16

For a given choice of radix and redundancy, there exists more than one possible table design. As discussed previously, a minimum number of divisor estimate bits is required as input for a given configuration. This corresponds to a maximum number of partial remainder bits that need be used. However, it is possible to trade an increase in divisor bits for a reduction in the number of partial remainder bits. This might initially seem desirable, as the partial remainder bits must first be assimilated in an external adder, adding to the overall iteration time. By using a fewer number of partial remainder bits in the table, the external adder can be smaller, reducing the external delay. However, for carry-save partial remainders, the maximum partial

| a | b | i | f | g | Terms | Relative Table Delay | Relative Table Area |
|---|---|---|---|---|---|---|---|
| 17 | 9 | 7 | 5 | 5 | † | † | † |
| | 8 | 7 | 6 | 6 | † | † | † |
| 18 | 9 | 7 | 4 | 4 | † | † | † |
| | 8 | 7 | 5 | 5 | † | † | † |
| | 7 | 7 | 7 | 7 | † | † | † |
| 19 | 8 | 7 | 4 | 4 | 351,208,352,945,1633,3119 | 4.56 | 141 |
| | 7 | 7 | 5 | 5 | 309,191,308,860,1445,2767 | 4.18 | 79.1 |
| 20 | 9 | 7 | 3 | 3 | 312,164,660,891,1527,3218 | 4.69 | 144 |
| | 7 | 7 | 4 | 4 | 257,156,531,727,1215,2592 | 4.24 | 106 |
| 21 | 8 | 7 | 3 | 3 | 237,128,507,649,1274,2499 | 4.13 | 73.2 |
| | 7 | 7 | 4 | 4 | 206,118,424,541,1060,2099 | 4.06 | 94.8 |
| | 6 | 7 | 6 | 6 | 180,108,366,466,912,1826 | 3.91 | 80.5 |
| 22 | 7 | 7 | 3 | 3 | 192,119,450,733,1032,2127 | 4.31 | 101 |
| | 6 | 7 | 5 | 5 | 178,90,356,596,794,1696 | 4.07 | 80.9 |
| 23 | 7 | 7 | 3 | 3 | 158,85,365,607,946,1865 | 4.11 | 92.5 |
| | 6 | 7 | 4 | 4 | 140,84,316,527,814,1621 | 3.86 | 78.6 |
| 24 | 9 | 7 | 2 | 2 | 207,408,421,678,1073,2243 | 4.62 | 92.5 |
| | 7 | 7 | 3 | 3 | 147,327,314,491,780,1678 | 3.63 | 86.8 |
| | 6 | 7 | 4 | 4 | 142,286,277,458,699,1497 | 3.61 | 73.3 |
| 25 | 8 | 7 | 2 | 2 | 185,330,318,543,985,1978 | 4.36 | 99.6 |
| | 6 | 7 | 3 | 3 | 144,252,258,425,747,1534 | 3.70 | 76.3 |
| 26 | 8 | 7 | 2 | 2 | 154,291,299,607,838,1783 | 3.95 | 91.6 |
| | 6 | 7 | 3 | 3 | 123,249,235,505,687,1475 | 3.73 | 75.1 |
| 27 | 7 | 7 | 2 | 2 | 141,263,266,535,798,1620 | 3.73 | 86.4 |
| | 6 | 7 | 3 | 3 | 126,221,227,439,661,1377 | 3.63 | 73.7 |
| 28 | 7 | 7 | 2 | 2 | 146,233,359,494,717,1578 | 3.86 | 82.0 |
| | 6 | 7 | 3 | 3 | 134,218,322,413,599,1327 | 3.76 | 74.4 |
| 29 | 7 | 7 | 2 | 2 | 226,233,342,431,697,1480 | 3.75 | 82.1 |
| | 6 | 7 | 3 | 3 | 116,188,259,349,573,1241 | 3.73 | 69.8 |
| 30 | 6 | 7 | 2 | 2 | 145,220,318,461,656,1433 | 4.14 | 76.0 |
| | 5 | 7 | 7 | 7 | 165,184,252,351,505,1143 | 3.78 | 61.0 |
| 31 | 11 | 8 | 1 | 1 | † | † | † |
| | 6 | 8 | 2 | 2 | 89,170,241,399,555,1158 | 4.05 | 61.3 |
| | 5 | 8 | 4 | 4 | 86,152,219,333,486,1021 | 3.52 | 57.2 |

Table 5.6: Radix 32

remainder truncation error $\epsilon_{p(cs)}$ is greater than the maximum divisor truncation error $\epsilon_d$. By trading-off fewer partial remainder bits for more divisor bits, the height of the uncertainty region increases at approximately twice the rate at which the width of the region decreases. As a result, the overall uncertainty region area increases as fewer partial remainder bits are used. This result is presented in tables 5.4, 5.5, and 5.6. For any given choice of radix and redundancy, the use of the maximum number of divisor bits and minimum number of partial remainder bits results in the largest number of total product terms, and typically the largest delay and area. As the number of divisor bits is reduced and the number of partial remainder bits increased, the number of product terms, the delay, and the area are all typically reduced.

This study confirms that as the radix increases, the complexity of the tables also increases. Fitting the average area at a given radix to a curve across the various radices determines that the area increases geometrically with increasing radix:

$$\text{Area} = .1R^2 \tag{5.16}$$

for radix $R$, where this area is the table area relative to that of the base radix 4 divider. Similar analysis of average table delay demonstrates that table delay increases only linearly with an increase in the number of bits retired per cycle. Increasing the radix of the algorithm reduces the total number of iterations required to compute the quotient. However, to realize an overall reduction in the total latency as measured in time, the delay per iteration must not increase at the same rate. Thus, to realize the performance advantage of a higher radix divider, it is desirable for the delay of the look-up table to increase at no more than the rate of increase of the number of bits retired per cycle.

For radix 8, the delay is on the average about 1.5 times that of the base radix 4 table. However, it can require up to 10 times as much area. While radix 16 tables have about 2 times the base delay, they can require up to 32 times the area. In the case of radix 32, it was not even possible to achieve a delay of 2.5 times the base delay, the maximum desired delay, with actual delays between 3.5 and 4.7. The area required for radix 32 ranges from 57 to 141 times the base area. These results show that radix 16 and 32 are clearly impractical design choices, even ignoring practical implementation limitations such as generating all divisor multiples. This study shows that it is possible to design radix 8 tables with reasonable delay and area; a minimally-redundant radix 8 table is demonstrated to be a practical design choice.

## 5.6 Summary

This chapter has proposed a methodology for generating quotient-digit selection tables from a table specification through an automated design flow. Using this process, performance and area tradeoffs of quotient selection tables in SRT dividers have been presented for several table configurations. The use of Gray-coding is shown to be a simple yet effective method that allows automatically determining optimal choices of quotient-digits which reduce table complexity.

Short external carry-assimilating adders are necessary to convert redundant partial remainders to a non-redundant form. By extending the length of these adders by as little as one or two bits, table complexity can be further reduced. The conventional wisdom for SRT table specification is that the length of the partial remainder estimate should be reduced at the expense of increasing the length of the divisor estimate in order to reduce the width, and thus the delay, of the external adder. However, this study shows that such a choice also increases the size and delay of the table, mitigating the performance gain provided by the narrower adder. Accordingly, the overall iteration time is not reduced through such a tradeoff.

As the radix increases, the table delay increases linearly. However, the area increases quadratically with increasing radix. This fact, combined with the difficulty in generating all of the required divisor multiples for radix 8 and higher, limits practical table implementations to radix 2 and radix 4.

# Chapter 6

# Division and Reciprocal Caches

## 6.1 Introduction

Computer applications often use the same input operands as those in a previous calculation. In matrix inversion, for example, each entry must be divided by the determinant. By recognizing and taking advantage of this redundant behavior, it is possible to decrease the effective latency of computations.

Richardson [79] discusses the technique of result caching as a means of decreasing the latency of otherwise high-latency operations, such as division. This technique exploits the redundant nature of certain computations by trading execution time for increased memory storage. Once a result is calculated, it is stored in a result cache. When an operation is initiated, the result cache can be simultaneously accessed to check for a previous instance of the computation. If the previous computation result is found, the result is available immediately from the cache. Otherwise, the operation continues in the functional unit, and the result is written into the cache upon completion of the computation.

This chapter investigates two techniques for decreasing the average latency of floating-point division. Both techniques are based on recurring or redundant computations that can be found in applications. When the same divide calculation is performed on multiple occasions, it is possible to store and later reuse a previous result without having to repeat the lengthy computation. For multiplication-based division implementations, the reciprocal can be reused rather than the quotient, increasing the likelihood of the computation being redundant [86]. Additionally, due to the similarity between division and square root computation, the quantity of redundant square root computation is investigated.

## 6.2 Reciprocal Caches

### 6.2.1 Iterative Division

As discussed in chapter 4, division can be implemented in hardware using the following relationship:

$$Q = \frac{a}{b} = a \times (\frac{1}{b}),$$

where $Q$ is the quotient, $a$ is the dividend, and $b$ is the divisor. Certain algorithms, such as the Newton-Raphson and series expansion iterations, are used to evaluate the reciprocal [46]. Whereas Newton-Raphson converges to a reciprocal and then multiplies by the dividend to compute the quotient, Goldschmidt's algorithm typically prescales both the dividend (a) and the denominator (b) by an approximation of the reciprocal and converges directly to the quotient. In this study, we use a modified implementation of the series expansion or Goldschmidt's algorithm to converge to the reciprocal. As the desired result is the reciprocal, only the divisor need be prescaled by the initial reciprocal approximation. The initial numerator is the reciprocal approximation itself. Higher performance can be achieved by using a higher precision starting approximation. Due to the quadratic convergence of these iterative algorithms, the computation of 53-bit double precision quotients using an 8-bit initial approximation table requires 3 iterations, while a 16-bit table requires only 2 iterations. This results in a tradeoff between area required for the initial approximation table and the latency of the algorithm, as discussed in chapter 4. In this study, the additional tradeoff is presented between larger initial approximation tables and cache storage for redundant reciprocals.

### 6.2.2 Experimental Methodology

To obtain the data for the study, ATOM [38] was used to instrument several applications from the SPECfp92 [11] and NAS [9] benchmark suites. These applications were then executed on a DEC Alpha 3000/500 workstation. All double precision floating-point division operations were instrumented. For division, the exponent is handled in parallel with the mantissa calculation. Accordingly, the quotient mantissa is independent of the input operands' exponents.

For reciprocal caches, the cache tag is the concatenation of the divisor mantissa and a valid bit, for a total of 53 bits. Because the leading one is implied for the mantissas, only 52 bits per mantissa need be stored. The cache data is the double precision reciprocal mantissa, with implied leading one, and the guard, round, and sticky bits for a total of 55 bits. These extra bits are required to allow for correct rounding on subsequent uses of the same reciprocal, with possibly different rounding modes. The total storage required for each entry is therefore 108 bits.

When a division operation is initiated, the reciprocal cache is simultaneously accessed to check for a previous instance of the reciprocal. If the result is found, the reciprocal is returned and multiplied by the dividend to form the quotient. Otherwise, the operation continues in the divider, and upon computation of the reciprocal the result is written into the cache.

### 6.2.3 Performance

Reciprocal cache hit rates were first measured assuming an infinite cache. These results are shown in figure 6.1. The average hit rate of all 11 applications is 81.7%, with a standard deviation of 27.7%. From figure 6.1, it can be seen that the application tomcatv is unusual in that it has no reciprocal reuse, as demonstrated by its 0% hit rate. When tomcatv is excluded, the average hit rate is 89.8%, and the standard deviation is only 6.2%. Fully-associative reciprocal caches of finite size were then simulated, and the resulting hit rates are shown in figure 6.2. Figure 6.2 shows that most of the redundant reciprocal computation is captured by a 128 entry cache, with a hit rate of 77.1%.

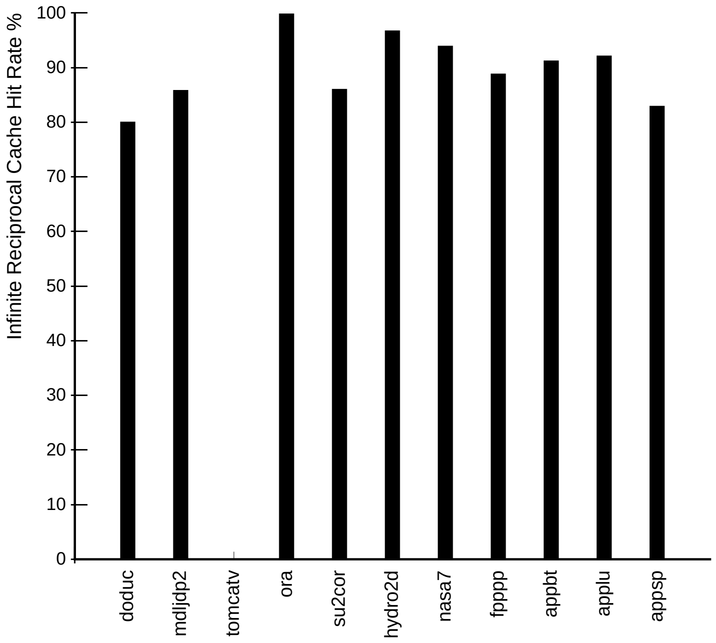


Figure 6.1: Hit rates for infinite reciprocal caches

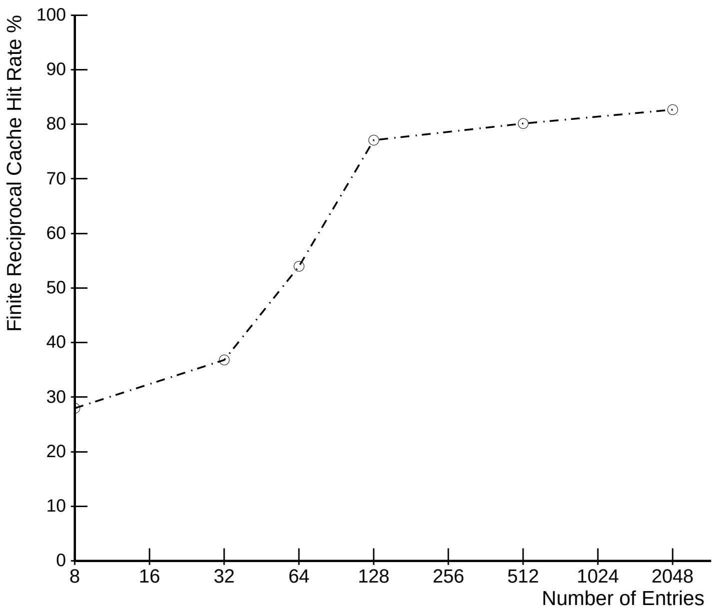


Figure 6.2: Hit rates for finite reciprocal caches

| ROM Size | Cache Entries | Latency (cycles) | Extra Area (bits) |
|---|---|---|---|
| 8-bit | 0 | 10 | 0 |
| 16-bit | 0 | 8 | 1,046,528 |
| 8-bit | 8 | 9.48 | 864 |
| 8-bit | 32 | 8.69 | 3,456 |
| 8-bit | 64 | 7.14 | 6,912 |
| 8-bit | 128 | 5.06 | 13,824 |
| 8-bit | 512 | 4.79 | 55,296 |
| 8-bit | 2048 | 4.56 | 221,184 |

Table 6.1: Performance/area tradeoffs for reciprocal caches

To determine the effect of reciprocal caches on overall system performance, the effective latency of division is calculated for several iterative divider configurations. For this analysis, the comparison is made with respect to the modified implementation of Goldschmidt's algorithm discussed previously. It is assumed that a pipelined multiplier is present with a latency of 2 cycles and a throughput of 1 cycle.

The latency for a division operation can be calculated as follows. An initial approximation table look-up is assumed to take 1 cycle. The initial prescaling of the dividend and the divisor requires 2 cycles. Each iteration of the algorithm requires 2 cycles for the 2 overlapped multiplications. The final result is available after an additional cycle to drain the multiplier pipeline. Thus, a base 8-bit Goldschmidt implementation without a cache requires 10 cycles to compute the quotient. Two cases arise for a scheme using a reciprocal cache. A hit in the cache has an effective latency of only 3 cycles: 1 cycle to return the reciprocal and 2 to perform the multiplication by the dividend. A miss in the cache suffers the base 10 cycle latency plus an additional 2 cycles to multiply the reciprocal by the dividend, as per the modified Goldschmidt implementation. While the modified Goldschmidt implementation does not explicitly require a multiplication to prescale the initial numerator, this does not result in a reduction in latency. When a pipelined multiplier is used, the latency of this multiplication is completely overlapped with the other operations. The results of this analysis are shown in table 6.1.

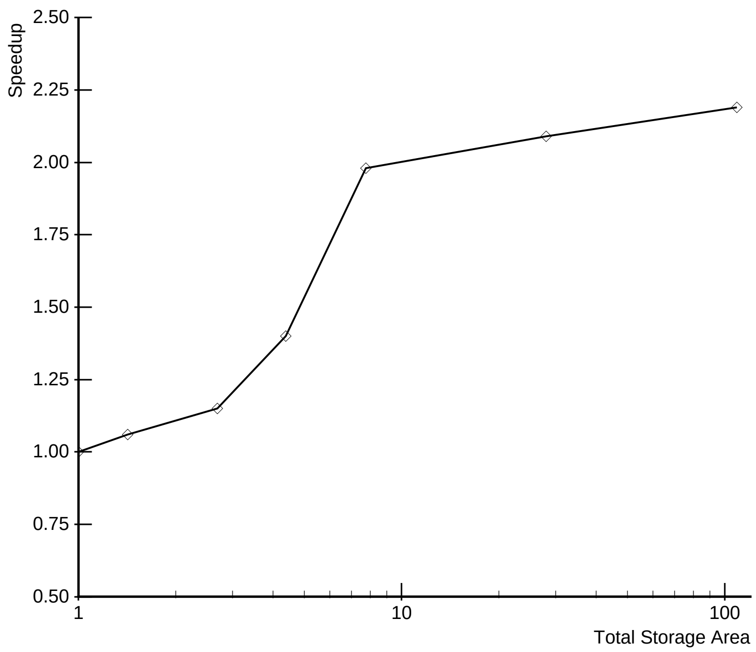


Figure 6.3: Speedup from reciprocal caches

Figure 6.3 shows the performance of the different cache sizes relative to an 8-bit initial approximation table implementation. Here, the speedups are measured against the total storage area required, expressed as a factor of the 8-bit initial approximation table size, which is 2048 bits. This graph demonstrates that when the total storage is approximately eight-times that of an 8-bit implementation with no cache, a reciprocal cache can provide a significant increase in division performance, achieving approximately a two-times speedup. When the total area exceeds eight-times the base area, the marginal increase in performance does not justify the increase in area. A reciprocal cache implementation can be compared to the use of a 16-bit initial approximation table, with a total storage of 1M bits. This yields an area factor of 512, with a speedup of only 1.25. The use of various table compression techniques could reduce this storage requirement. However, the best case speedup with no reciprocal cache and requiring 2 iterations is still 1.25.

## 6.3 Division Caches

An alternative to a reciprocal cache to reduce division latency is a division cache. A division cache can be used for any form of divider implementation, regardless of the choice of algorithm. For a division cache, the tag is larger than that of a reciprocal cache, as it comprises the concatenation of the dividend and divisor mantissas, and a valid bit, forming 105 bits. Accordingly, the total storage required for each division cache entry is 160 bits. The functionality of the division cache is similar to that of the reciprocal cache. When a division operation is initiated, the division cache can be simultaneously accessed to check for a previous instance of the exact dividend/divisor pair. If the result is found, the correct quotient is returned. Otherwise, the operation continues in the divider, and upon computation of the quotient, the result is written into the cache. The number of computations reusing both operands at best will be equal to and will be typically less than the number reusing only the same divisor. However, reciprocal caches restrict the form of algorithm used to compute the quotient, while division caches allow any divider implementation.

Hit rates were measured for each of the applications assuming an infinite division cache. The average hit rate was found to be 57.1%, with a standard deviation of 36.5%. When analyzing only those applications that exhibited some redundant computation, excluding tomcatv and su2cor, the average hit rate is 69.8%, with a standard deviation of 25.8%. Thus, the quantity of redundant division in the applications compared with redundant reciprocals was lower and more variant.

Fully-associative finite division caches were simulated, and the resulting hit rates are shown in figure 6.4, along with the hit rates of the reciprocal caches with the same number of entries. The results of figure 6.4 demonstrate a knee near a division cache of 128 entries, with an average hit rate of 60.9%. In general, the shape of the reciprocal cache hit rate tracks that of the division cache. For the same number of entries, though, the reciprocal cache hit rate is larger than that of the division cache by about 15%. Thus, the quantity of redundant division in the applications compared with redundant reciprocals was lower and more variant. Additionally, a division cache requires approximately 50% more area than a reciprocal cache with the same number of entries. Further performance and efficiency analysis of division caches is presented in [87].

## 6.4 Square Root Caches

The implementation of square root often shares the same hardware used for division computation. A variation of Goldschmidt's algorithm can be used to converge to the square root of an operand [24]. Thus, the question arises as to the quantity of redundant square root computation available in applications. Because both the reciprocal and square root operations are unary, they can share the same cache for their results.

In a similar experiment for square root, hit rates were measured for finite shared reciprocal/square root caches, where both reciprocals and square roots reside in the same cache. The results are shown in figure 6.5. The shared cache results show that for reasonable cache sizes, the square root result hit rates are low, about 50% or less. Although the frequency of square root was about 10 times less than division, the inclusion of square root results caused interference with the reciprocal results. This had the effect of decreasing the reciprocal hit rates, especially in the cases of 64 and 128 entries. Thus, this study suggests that square root computations should not be stored in either a dedicated square root cache or a shared reciprocal cache, due to the low and highly variant hit rate of square root and the resulting reduction in reciprocal hit rate.

## 6.5 Summary

This chapter has shown that redundant division computation exists in many applications. Both division caches and reciprocal caches can be used to exploit this redundant behavior. For high performance implementations, where a multiplication-based algorithm is used, the inclusion of a reciprocal cache is an efficient means of increasing performance. In this scenario, too, a division cache could be used. However, the high standard deviation of a division cache's hit rates compared with that of a reciprocal

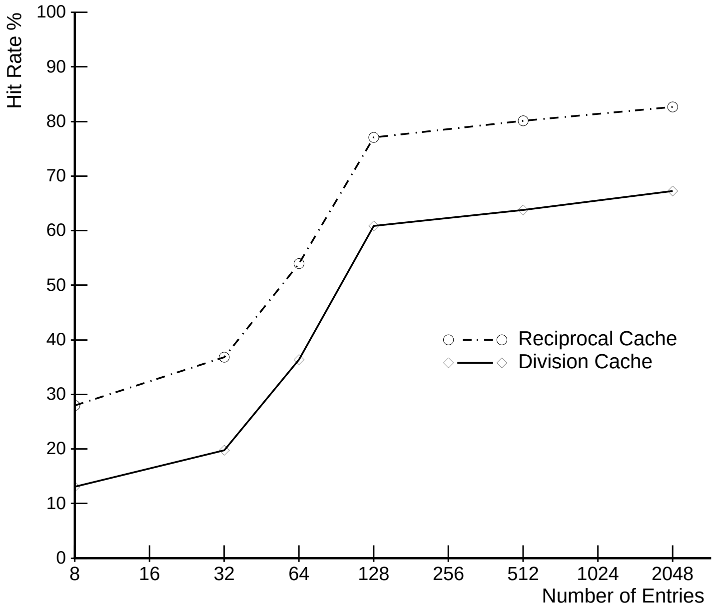


Figure 6.4: Hit rates for division caches

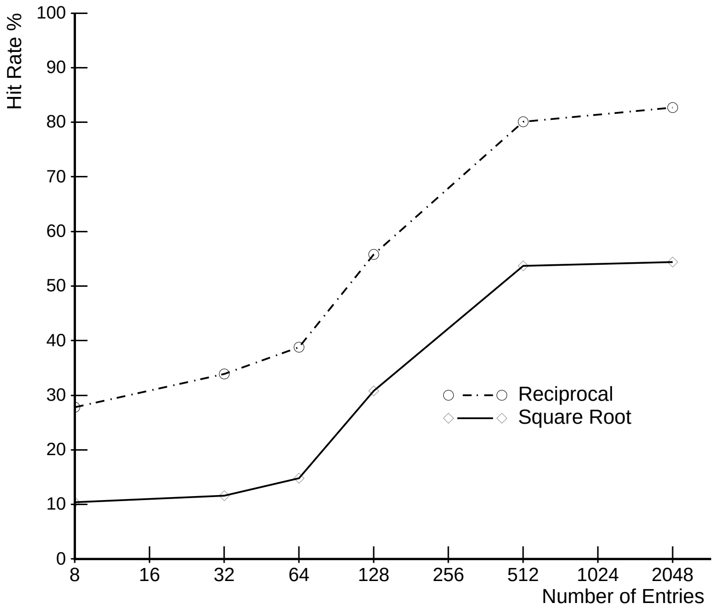


Figure 6.5: Hit rates for reciprocal/square root caches

cache argues against its usage and for the use of a reciprocal cache. Additionally, the analysis has shown that these applications do not contain a consistently large quantity of redundant square root computation. Thus, the caching of square root results as a means for increasing overall performance is not recommended.

The primary alternative previously to decrease latency of multiplication-based division algorithms has been to reduce the number of iterations by increasing the size of the initial approximation table. This chapter demonstrates that a reciprocal cache can often be an effective alternative to large reciprocal tables. The inclusion of a reasonably sized reciprocal cache can consistently provide a significant reduction in division latency.

# Chapter 7

# Fast Rounding

## 7.1 Introduction

Linear converging division algorithms retire a fixed number of quotient digits in each iteration. After each iteration, a partial remainder is available. At the conclusion of the iterations, the quotient is available, as is the final remainder. By noting the sign and magnitude of the final remainder, it is possible to adjust the quotient appropriately by 1 ulp to obtain an exactly rounded result that complies with the IEEE floating-point standard. In contrast, both Newton-Raphson and Goldschmidt's algorithms produce a quotient, but with no final remainder. For exact rounding of the quotient, it is typically necessary to use an additional multiplication of the quotient and the divisor and then to subtract the product from the dividend to form the final remainder. Accordingly, quadratically-converging algorithms can incur a latency penalty of one multiplication and a subtraction in order to produce IEEE exactly rounded quotients.

Previous implementations of quadratically-converging dividers have demonstrated various techniques of achieving close-to-exact rounding as well as exact rounding. However, all implementations with exactly rounded quotients have suffered from a rounding penalty. In this chapter, an extension of a VLA technique presented by Schwarz [88] is proposed which further reduces the frequency of final remainder calculations required by increasing the precision of the quotient. For those cases where a final remainder calculation is required, a technique is proposed which reduces the full-width subtraction to combinational logic operating on one bit of the dividend, one bit of the back multiplication product, and the sticky bit from the multiplier.

## 7.2 IEEE Rounding

The IEEE floating point representation describes two different formats: single and double precision. The standard also suggests the use of extended precision formats, but their use is not mandatory. The most common format used in modern processors is double precision, which comprises a 1-bit sign, an 11-bit biased exponent, and a 52-bit significand with one hidden significand bit, for a total of a 64 bits. A significand is a normalized number $M$, such that $1 \leq M < 2$. The standard includes four rounding modes: RN, RZ, RM, and RP. RN is unbiased rounding to nearest, rounding to even in the case of a tie. It guarantees that the final result has at most $\pm 0.5$ ulp error due to rounding. RZ is simple truncation. The two directed rounding modes RM and RP are round towards minus infinity and round towards plus infinity respectively. Exactly rounded results must be computable for all four rounding modes. The result generated by an operation according to any of the four rounding modes must be the machine number which is identical to an intermediate result that is correct to infinite precision and is then rounded according to the same rounding mode.

The significand immediately before rounding has the format as given in figure 7.1. In this figure, $L$ is the LSB before rounding, $G$ is the guard bit, and $S$ is the sticky


Figure 7.1: Significand format before rounding

bit. The guard bit is one bit less significant than is strictly required by the precision of the format. The sticky bit is essentially a flag, noting the existence of any bits in the result less significant than the guard bit. It is the logical OR of all of the less significant bits in the result.

To implement rounding for each of the rounding modes, an action table listing the appropriate procedure for all combinations of $L$, $G$, and $S$ can be written. An example action table for RN is shown in table 7.1. The rightmost column of the action table

| L | G | S | Action | A |
|---|---|---|---|---|
| X | 0 | 0 | Exact result. No rounding. | 0 |
| X | 0 | 1 | Inexact result, but is correctly rounded. | 0 |
| 0 | 1 | 0 | Tie case with even significand, so correctly rounded. | 0 |
| 1 | 1 | 0 | Tie case with odd significand, so round to nearest even. | 1 |
| X | 1 | 1 | Round to nearest. | 1 |

Table 7.1: Action table for RN rounding mode

dictates whether the result should be rounded. Rounding is accomplished by adding $A$ to $L$ to obtain the correct machine number. Such a table can be implemented simply in random logic. Similar tables can be written for the other three rounding modes.

For division, the complication is the determination of the sticky bit. This determination requires knowledge of the magnitude of the final remainder. Since division by functional iteration does not directly provide the remainder, the design challenge is how to gain the information in the remainder while incurring as minimal a latency penalty as possible.

## 7.3 Division by Functional Iteration

Techniques for performing division by functional iteration have been presented in chapter 4. However, the Newton-Raphson and Goldschmidt algorithms were presented assuming that the multipliers used within the iterations were full-precision. For a complete analysis of the design space as it relates to exact rounding, it is necessary to consider other multiplier precisions. A performance enhancement that can be used for both algorithms is to perform early computations in reduced precision. This is acceptable, because the early computations do not generate many correct bits. As the iterations continue, quadratically larger amounts of precision are required in the computation. However, this has an effect on the precision of the final quotient approximation. Consider the series expansion algorithm. For $n$ bit input operands, so long as all multiplications are at least $2n$ bits wide, then

$$D_{i+1} = 0.11 \cdots xxx$$

approaching 1 from below. Similarly $N_{i+1}$ approaches the quotient from below. Accordingly the final $n$ bit result can have at most a 1 ulp error which satisfies:

$$0 \leq \epsilon_t < 2^{-n}$$

and therefore the error in the final $n$ bit result $Q'$ is satisfied by:

$$0 \leq Q - Q' < 2^{-n} \tag{7.1}$$

where $Q$ is the infinitely precise result. Should either of the two iteration products

$$D_{i+1} = D_i \times R_i$$

or

$$N_{i+1} = N_i \times R_i$$

be computed with multipliers which return only the top $k$ bits, where $k < 2n$, rounding error is added into the approximation. If the multiplications return only the top $k$ bits of precision in the result, then

$$R_{it} = 2 - D_{it}$$

This induces rounding error $\epsilon_r$ in $D_{i+1}$ that satisfies

$$|\epsilon_r| < 2^{-(k+1)}$$

Because of the resulting error in $D_{i+1}$, it converges to 1, but it converges from either below or above rather than from strictly below.

## 7.4 Previously Implemented Techniques

There have been three main techniques used in previous implementations to compute rounded results when using division by functional iteration. The IBM 360/91 implemented division using Goldschmidt's algorithm [64]. In this implementation, 10 extra bits of precision in the quotient were computed. A hot-one was added in the LSB of the guard bits. If all of the 10 guard bits were ones, then the quotient was rounded up. This implementation had the advantage of the fastest achievable rounding, as it did not require any additional operations after the completion of the iterations. However, while the results could be considered "somewhat round-to-nearest," they were definitely not IEEE compliant, and there was no concept of exact rounding in this implementation.

Another implemented method requires a datapath twice as wide as the final result, and it is the method used to implement division in the IBM RS/6000 [68]. The quotient is computed to a little more than twice the precision of the final quotient, and then the extended result is rounded to the final precision. An explanation of this procedure is as follows. Consider that the dividend $X$ and the divisor $Y$ are both normalized and represented by $b$ bits, and the final quotient $Q = X/Y$ is represented by $b$ bits. Note that the exact halfway quotient can not occur when dividing two $b$ bit normalized numbers. For an exact halfway case, the quotient would be represented by exactly a $b+1$ bit number with both its MSB and LSB equal to 1, and thus having exactly $b-1$ bits between its most significant and least significant 1's. The product of such a number with any non-zero finite binary number must also have the same property, and thus the dividend must have this property. But, the dividend is defined to be a normalized $b$ bit number, and thus can it can have a maximum of $b-2$ bits between its most significant and least significant 1's.

To obtain $b$ significant bits of the quotient, $b$ bits are computed if the first quotient bit is 1, and $b+1$ bits if the first quotient bit is 0. At this point, because the exact halfway case can not occur, rounding can proceed based solely on the values of the next quotient bit and the sticky bit. The sticky bit is 0 if the remainder at this point is exactly zero. If any bit of the remainder is 1, then the sticky bit is 1. Let $R_0$ be the value of the remainder after this computation, assuming the first bit is 1:

$$X = Q_0 \times Y + R_0, \quad \text{with } R_0 < 2^{-b}$$

Then, compute another $b$ bits of quotient, denoted $Q_1$.

$$R_0 = Q_1 \times Y + R_1, \quad \text{with } R_1 < 2^{-2b}$$

$Q_1$ is less than $2^{-b}$, with an accuracy of $2^{-2b}$, and Y is normalized to be accurate to $2^{-b}$. Accordingly if $Q_1 = 0$, then $R_0 = R_1$. But, $R_0$ can equal $R_1$ if and only if $R_0 = R_1 = 0$. This is because $R_0 < 2^{-b}$ and $R_1 < 2^{-2b}$ and $Y$ is a $b$ bit quantity. Similarly, if $Q_1 \neq 0$, then the remainder $R_0$ can not equal 0. The computation proceeds in the same manner if the first quotient bit is 0, except that $b + 1$ bits will have been computed for $Q_0$. By computing at most $2b + 1$ bits, the sticky bit can be determined, and the quotient can be correctly rounded.

The RS/6000 implementation uses its fused multiply-accumulate for all of the operations to guarantee accuracy greater than $2n$ bits throughout the iterations. After the completion of the additional iteration,

$$Q' \;=\; \text{estimate of } Q = \frac{a}{b} \text{ accurate to } 2n \text{ bits}$$

A remainder is calculated as

$$R \;=\; a - b \times Q' \tag{7.2}$$

A rounded quotient is then computed as

$$Q'' \;=\; Q' + R \times b \tag{7.3}$$

where the final multiply-accumulate is carried in the desired rounding mode, providing the exactly rounded result. The principal disadvantage of this method is that it requires one additional full iteration of the algorithm, and it requires a datapath at least two times larger than is required for non-rounded results.

A third approach is that which was used in the TI 8847 FPU [69]. In this scheme, the quotient is also computed with some extra precision, but less than twice the desired final quotient width. To determine the sticky bit, the final remainder is directly computed from:

$$\begin{aligned} Q &= \frac{a}{b} - R \\ R &= a - b \times Q \end{aligned}$$

It is not necessary to compute the actual magnitude of the remainder; rather, its relationship to zero is required. In the worst case, a full-width subtraction may be used to form the true remainder $R$. Assuming sufficient precision is used throughout the iterations such that all intermediate multiplications are at least $2n$ bits wide for $n$ bit input operands, the computed quotient is less than or equal to the infinitely precise quotient. Accordingly, the sticky bit is zero if the remainder is zero and one if it is nonzero. If truncated multiplications are used in the intermediate iterations, then the computed quotient converges toward the exactly rounded result, but it may be either above or below it. In this case, the sign of the remainder is also required to detect the position of the quotient estimate relative to the true quotient. Thus, to support exact rounding using this method, the latency of the algorithm increases by at least the multiplication delay necessary to form $Q \times b$, and possibly by a full-width subtraction delay as well as zero-detection and sign-detection logic on the final remainder. In the TI 8847, the relationship of the quotient estimate and the true quotient is determined using combinational logic on 6 bits of both $a$ and $Q \times b$.

## 7.5 Reducing the Frequency of Remainder Computations

### 7.5.1 Basic Rounding

To simplify the discussion and analysis of the rounding techniques in the rest of this chapter, it is assumed that the input operands are normalized significands in the range [0.5,1), rather than the IEEE range of [1,2). The analysis is equivalent under both conditions and there is no loss of generality. Accordingly, 1 ulp for such a normalized $n$ bit number is $2^{-n}$. The basic rounding technique is as follows. Depending upon the widths of the multiplications and the precision of the initial approximation, there are three different cases that need to be considered.

**Case 1:**

The multiplications used in the iterations return at least $2n$ bit results for $n$ bit operands. Sufficient iterations of the algorithm are then implemented such that the quotient is accurate to $n+1$ bits with an error strictly less than 1 ulp of this $n+1$ bit quantity. As the final result only has $n$ bits, the quotient estimate has an error strictly less than +0.5 ulp, and this estimate satisfies:

$$0 \leq Q - Q' < 2^{-(n+1)} \tag{7.4}$$

The steps to correctly round this quotient estimate are:

- Add $2^{-(n+2)}$ to $Q'$.
- Truncate the transformed $Q'$ to $n+1$ bits to form $Q''$. $Q''$ will then have strictly less than $\pm 0.5$ ulp error.
- Form the remainder $R = a - b \times Q''$, which is an $n$ bit by $(n+1)$ bit product.
- By observing the sign and magnitude of $R$ and bit $n+1$, the guard bit, all IEEE rounding modes can be implemented by choosing either $Q''$, $Q'' + 2^{-n}$, or $Q'' - 2^{-n}$.

After the addition in the first step, $Q'$ satisfies:

$$-2^{-(n+2)} \leq Q - (Q' + 2^{-(n+2)}) < 2^{-(n+2)} \tag{7.5}$$

Truncation in the second step induces an error satisfying

$$0 \leq \epsilon_t < 2^{-(n+1)} \tag{7.6}$$

after which the result $Q''$ satisfies

$$-2^{-(n+2)} \leq Q - Q'' < 2^{-(n+1)} \tag{7.7}$$

Thus, the result can have an error of [-0.25,+0.5) ulp. This can be rewritten as a looser but equivalent error bound of (-0.5,+0.5) ulp. Accordingly, the observation

| Guard Bit | Remainder | RN | RP (+/-) | RM (+/-) | RZ |
|---|---|---|---|---|---|
| 0 | =0 | **trunc** | trunc | trunc | trunc |
| 0 | - | **trunc** | trunc/dec | dec/trunc | dec |
| 0 | + | **trunc** | inc/trunc | trunc/inc | trunc |
| 1 | = 0 | — | — | — | — |
| 1 | - | trunc | **inc/trunc** | **trunc/inc** | **trunc** |
| 1 | + | inc | **inc/trunc** | **trunc/inc** | **trunc** |

Table 7.2: Action table for basic method

of the guard bit and the sign and equality to zero of the remainder are sufficient to exactly round the quotient. The rounding in the last step is accomplished by conditionally incrementing or decrementing $L$. The action table for correctly rounding $Q''$ is shown in table 7.2. For the directed rounding modes RP and RM, the actual action may depend upon the sign of the quotient estimate. Those entries that contain two operations such as *pos/neg* are for the sign of the final result itself being positive and negative respectively. As discussed earlier, the exact halfway case can not occur in division, and thus the table row with $G = 1$ and $R = 0$ has no entries.

**Case 2:**

A similar methodology can be used should the estimate converge to the infinitely precise value from either above or below. Such a situation occurs when multipliers are used in the iterations returning fewer than $2n$ bits, adding either truncation or rounding error to the estimate, and when the two's complement operation is replaced by the simpler one's complement, also adding a fixed error to the estimate. This added error causes convergence to the infinitely precise quotient from either side rather than strictly from below. Assuming that at least $n+2$ bits of quotient are computed with sufficient iterations to guarantee an accuracy of $n+1$ bits as before, then the estimate $Q'$ can have at most a $\pm 0.5$ ulp error, and it satisfies

$$-2^{-(n+1)} < Q - Q' < 2^{-(n+1)} \tag{7.8}$$

due to the convergence to the quotient from above or below. This is not sufficient precision for rounding using the guard bit. Instead, the estimate must be accurate to

$n+2$ bits, requiring at least 2 additional bits, rather than 1, to be computed in the iterations. Then, the estimate satisfies

$$-2^{-(n+2)} < Q - Q' < 2^{-(n+2)} \tag{7.9}$$

In this case, the addition of $2^{-(n+2)}$ is again performed

$$-2^{-(n+1)} < Q - (Q' + 2^{-(n+2)}) < 0 \tag{7.10}$$

and after truncation to $n+1$ bits forms $Q''$

$$-2^{-(n+1)} < Q - Q'' < 2^{-(n+1)} \tag{7.11}$$

After these adjustments, rounding proceeds in the same manner as discussed previously using table 7.2.

**Case 3:**

A final possibility is that due to large errors in the initial approximation, the bounds on the error in the quotient estimate are inclusive rather than exclusive. Assuming as before that at least $n+2$ bits of quotient are computed with an accuracy of $n+2$ bits, then the estimate satisfies

$$-2^{-(n+2)} \leq Q - Q' \leq 2^{-(n+2)} \tag{7.12}$$

The addition of $2^{-(n+2)}$ to $Q'$ yields

$$-2^{-(n+1)} \leq Q - (Q' + 2^{-(n+2)}) \leq 0 \tag{7.13}$$

After truncation to $n+1$ bits, the truncation error $\epsilon_t$ causes $Q''$ to satisfy

$$-2^{-(n+1)} \leq Q - Q'' < 2^{-(n+1)} \tag{7.14}$$

Due to the lower inclusive point, it is not possible to round directly as before. To allow the same rounding methodology, it is necessary to force this bound to be exclusive rather than inclusive. To do this, it is necessary that the accuracy of the original quotient estimate $Q'$ have more than $n+2$ bits of accuracy. As an example, if $Q'$ has $n+3$ bits of accuracy using at least $n+3$ bits of quotient, then it will satisfy

$$-2^{-(n+3)} \leq Q - Q' \leq 2^{-(n+3)} \tag{7.15}$$

The addition of $2^{-(n+2)}$ to $Q'$ forms

$$-2^{-(n+3)} - 2^{-(n+2)} \leq Q - (Q' + 2^{-(n+2)}) \leq 2^{-(n+3)} - 2^{-(n+2)} \tag{7.16}$$

and after truncation to $n+1$ bits forming $Q''$

$$-2^{-(n+3)} - 2^{-(n+2)} \leq Q - Q'' < 2^{-(n+1)} \tag{7.17}$$

which clearly satisfies

$$-2^{-(n+1)} < Q - Q'' < 2^{-(n+1)} \tag{7.18}$$

after which rounding may proceed using table 7.2. This analysis shows that in this case it is necessary for the quotient estimate to have an accuracy of strictly greater than $n+2$ bits.

### 7.5.2 Faster Rounding

By observing the entries in table 7.2, it can be seen that for any of the rounding modes, half of the column's entries are identical. These identical entries are shown in bold type in the table. Accordingly, for all rounding modes, only half of the entries require knowledge of the remainder itself. As an example, for RN, if $G = 0$, then the correct action is truncation, regardless of the sign or magnitude of the remainder. However, if $G = 1$, then the sign of the remainder is needed to determine whether truncation or incrementing is the correct rounding action. Similarly, for RP, if $G = 1$, then the correct rounding action is to increment if the sign of the quotient is positive, and truncation if negative. Again, no computation of the remainder is required. These results are similar to those reported in [88].

Implementation of such a variable latency divider is as follows. The iterations for computing the quotient estimate $Q'$ are carried to at least $n+2$ bits assuming an error bound of

$$0 \leq Q - Q' < 2^{-(n+1)}$$

As previously discussed, other error bounds on the quotient estimate can be tolerated by employing more accuracy in the results. Thus, the technique presented can be easily modified to handle other error bounds. The quantity $2^{-(n+2)}$ is then added to $Q'$. This can be done either in a dedicated additional addition step, or, for higher performance, as part of a fused multiply-accumulate operation in the last iteration of the division algorithm. Depending upon the rounding mode and the value of $G$, it may be possible to round immediately. Otherwise, it may be necessary to perform the back-multiplication and subtraction to form the final remainder, and to observe its magnitude and sign in order to begin rounding. Thus, assuming a uniform distribution of quotients, in half of the cases a back-multiplication and subtraction is not required, reducing the total division latency.

### 7.5.3 Higher Performance

The previously discussed technique requires the computation of one guard bit, and in so doing, allows for the removal of the back-multiplication and subtraction in about half of the computations. This method can be extended as follows. Consider that at least $n+3$ bits of quotient estimate are computed such that there are two guard bits with an error in this estimate of at most 1 ulp. This estimate $Q'$ then satisfies:

$$0 \leq Q - Q' < 2^{-(n+2)} \tag{7.19}$$

The preliminary steps to correctly round this quotient estimate are similar to the previous technique:

- Add $2^{-(n+3)}$ to $Q'$.
- Truncate the transformed $Q'$ to $n+2$ bits to form $Q''$. $Q''$ will then have at most $\pm 0.25$ ulp error.

The action table for correctly rounding this quotient estimate $Q''$ is shown in table 7.3. From this table, for each rounding mode, only in 1 out of the 4 possible guard bit combinations is a back-multiplication and subtraction needed, as denoted by the bold-faced operations. In all of the other guard bit combinations, the guard bits themselves along with the sign of the final result are sufficient for exact rounding.

These results can be generalized to the use of $m$ guard bits, with $m \geq 1$:

| Guard Bits | Remainder | RN | RP (+/-) | RM (+/-) | RZ |
|---|---|---|---|---|---|
| 00 | =0 | trunc | trunc | trunc | trunc |
| 00 | - | trunc | **trunc/dec** | **dec/trunc** | **dec** |
| 00 | + | trunc | **inc/trunc** | **trunc/inc** | **trunc** |
| 01 | =0 | trunc | inc/trunc | trunc/inc | trunc |
| 01 | - | trunc | inc/trunc | trunc/inc | trunc |
| 01 | + | trunc | inc/trunc | trunc/inc | trunc |
| 10 | = 0 | — | — | — | — |
| 10 | - | **trunc** | inc/trunc | trunc/inc | trunc |
| 10 | + | **inc** | inc/trunc | trunc/inc | trunc |
| 11 | = 0 | inc | inc/trunc | trunc/inc | trunc |
| 11 | - | inc | inc/trunc | trunc/inc | trunc |
| 11 | + | inc | inc/trunc | trunc/inc | trunc |

Table 7.3: Action table using two guard bits

- Add $2^{-(n+m+1)}$ to $Q'$.
- Truncate the transformed $Q'$ to $n+m$ bits to form $Q''$. $Q''$ will then have at most $\pm 2^{-m}$ ulp error.
- In parallel, observe the guard bits and begin computation of the remainder $R = a - b \times Q''$.
- If the guard bits are such that the sign and magnitude of the remainder are required, wait until the remainder is computed and then round. Otherwise, begin rounding immediately.

After the conversion in the first two steps, $Q''$ satisfies:

$$-2^{-(n+m)} < Q - Q'' < 2^{-(n+m)} \tag{7.20}$$

By inspecting the $m$ guard bits, a back-multiplication and subtraction are required for only $2^{-m}$ of all cases. Specifically, the RN mode needs the computation to check the position around the mid-point between two machine numbers. The other three modes use the guard bits to check the position around one of the two machine numbers themselves. The examination of the $m$ guard bits dictates the appropriate action in all of the other cases.

## 7.6 Faster Magnitude Comparison

For those cases where it is necessary to have information regarding the remainder, it is not necessary to perform the complete calculation of the remainder. Rather, the essential elements are the sign of the remainder and whether the remainder is exactly zero. Stated slightly differently, what is required is whether the remainder is greater than, less than, or exactly equal to zero. The design challenge becomes how to compute this information in less time than that required for an $n$ bit multiplication, subtraction, and subsequent zero-detection logic.

Recall that in the worst case the remainder can be computed from:

$$a - b \times Q'' = R$$

Since the remainder's relationship to zero is the information desired, the equation can be rewritten as:

$$a - b \times Q'' \stackrel{?}{=} 0$$

or

$$a \stackrel{?}{=} b \times Q''$$

Clearly, the back multiplication of $b \times Q''$ is required for this comparison, and this multiplication should be carried in RZ mode, with no effective rounding. However, to reduce the latency penalty, it may be possible to remove the full-width subtraction, replacing it with simpler logic. The remaining question is the number of bits of both $a$ and $b \times Q''$ that are required in this comparison.

Recall from (7.20) that the maximum error in $Q''$ is $\pm 2^{-(n+m)}$. Therefore, the maximum error in the product $b \times Q''$ with respect to $a$ can be derived as follows:

$$b_{max} = 1 - 2^{-n} \tag{7.21}$$

| $X_{lsb}$ | $Y_{lsb}$ | Error in $Y$ | Sign of $X - Y$ |
|---|---|---|---|
| 0 | 0 | $< .01111\cdots$ | - |
| 0 | 1 | $> .01111\cdots$ | + |
| 1 | 0 | $> .01111\cdots$ | + |
| 1 | 1 | $< .01111\cdots$ | - |

Table 7.4: Sign prediction

$$\begin{aligned} error_{max} &= b_{max} \times Q'' \\ &= (1 - 2^{-n}) \times (\pm 2^{-(n+m)}) \end{aligned} \tag{7.22}$$

or

$$-2^{-(n+m)} + 2^{-(2n+1)} < error < 2^{-(n+m)} - 2^{-(2n+1)} \tag{7.23}$$

So long as the number of guard bits $m$ is greater than or equal to 1, then the absolute value of the error in the product $b \times Q''$ is strictly less than 0.5 ulp. Accordingly, the sign of the difference between $a$ and $b \times Q''$ can be exactly predicted by only examining the LSB of $a$ and the LSB of $b \times Q''$.

To demonstrate how this prediction can be implemented, consider the entries in table 7.4. In this table, let $X = a$ and $Y = b \times Q''$, each with $n$ bits. From this table, it is clear that the sign of the difference can be written as:

$$Sign = X_{lsb} \text{ XNOR } Y_{lsb} \tag{7.24}$$

Thus, the sign of the difference can be computed by using one gate, rather than requiring a complete full-width carry-propagate addition. This hardware is sufficient to handle the RN rounding mode which only requires the sign of the remainder and no information about the magnitude itself. Again, this is because RN only requires remainder information when trying to determine on which side of the mid-point between two machine numbers the true quotient lies. As the exact halfway case can not occur, exact equality to zero of the remainder need not be checked.

For the other three rounding modes, RP, RM, and RZ, remainder information is used to detect if the true quotient is less than, greater than, or exactly equal to a machine number. Rather than using additional hardware to detect remainder equality to zero, it is proposed to reuse existing hardware in the FP multiplier. For proper implementation of IEEE rounding for FP multiplication, most FP multipliers use dedicated sticky-bit logic. The sticky-bit of the multiplier is a flag signifying whether all bits below the LSB of the product are zero. In the context of the product of the back-multiplication of $b \times Q''$, this sticky-bit signals whether the product is exactly equal to $a$, and thus whether the remainder is exactly zero.

For those cases in RP, RM, and RZ requiring remainder information, the product $b \times Q''$ is computed using RZ, and the LSB of $a$ and $b \times Q''$ are observed, along with the sticky-bit from the multiplier. After using equation (7.24) to determine the sign, the following switching expressions can be used:

$$(b \times Q'' == a) = Sign \text{ AND } \overline{Sticky} \tag{7.25}$$

$$(b \times Q'' > a) = Sign \text{ AND } Sticky \tag{7.26}$$

$$(b \times Q'' < a) = \overline{Sign} \tag{7.27}$$

## 7.7 Summary

In division by functional iteration, extensions to VLA techniques for reducing the rounding penalty have been proposed. By using $m$ bits of extra precision in the adjusted quotient estimate, a back-multiplication and subtraction are required for only $2^{-m}$ of all cases, reducing the average latency for exactly-rounded quotients formed using functional iteration. Further, a technique has been presented which reduces the subtraction in the remainder formation to very simple combinational logic using the LSB's of the dividend and the back product of the quotient estimate and the divisor, along with the sticky bit from the multiplier. The combination of these techniques allows for increased division performance in processors which can exploit a VLA functional unit.

# Chapter 8

# Conclusion

## 8.1 Summary

High performance floating point units require high throughput, low latency functional units. For maximum system performance, the FPU must include a high performance FP adder, multiplier, and divider. As computer applications continue to increase their floating point instruction content, the demand for higher performance FPUs continues to increase.

The most frequent operation in FP intensive applications is addition. We have proposed a VLA technique for FP addition to reduce the average latency by exploiting the distribution of operands. The realization that not all operands require all of the components in the addition dataflow yields a speedup of as much 1.33 in average addition performance for high clock-rate microprocessors.

In spite of the low division frequency in FP applications, high latency dividers have a significant effect on system performance. In order to keep the CPI penalty due to division low, double precision division latency should be designed to be substantially below 60 cycles, a typical performance of some of today's simpler microprocessors. For scalar processors, a target of 10 cycle division latency is suggested. As the number of instructions issued per cycle increases, this target decreases, as the effects of a slow divider are accentuated by the increase in the urgency for division results. Depending upon the desired design-point on the performance/area curve, either SRT division or functional iteration may be appropriate.

SRT division is a good choice when higher latency division is acceptable. However, in spite of the apparent simplicity of SRT dividers, the SRT algorithm contains sufficient complexity to warrant continued investigation into next-quotient-digit table design, as demonstrated by the infamous flaw in the Intel SRT divider [89]. In this dissertation, a methodology has been proposed to reduce the latency of SRT quotient-digit selection tables, and thus increase SRT divider performance. Through the use of Gray-encoding and slightly longer external carry-assimilating adders, SRT table complexity can be reduced in terms of delay and area. While directly increasing the radix of an SRT divider reduces the total number of iterations required, the time per iteration must not also increase if an overall increase in performance is to be achieved. Due to the increase in next-quotient-digit table delay and area for higher radices, the individual stages in realistic SRT divider implementations are likely limited to radix 2 or 4. Further research on the application of aggressive circuit techniques to low radix dividers may yet allow low radix stages to be competitive in terms of latency.

For implementations where higher performance is required, division by functional iteration is the primary alternative. Both the Newton-Raphson and Goldschmidt iterations converge quadratically to the quotient. This is important for current long formats, such as double precision (53 bits) and double extended precision (64 bits), but it is especially important for possible future precisions, such as quad precision (113 bits). Goldschmidt's algorithm achieves lower latency by reordering the operations and exploiting pipelined multipliers. Both iterations can order the operations in order to exploit unused cycles in a pipelined multiplier allowing multiple division operations to be in progress simultaneously. This is particularly important for special-purpose applications, such as 3D graphics, that have a high throughput requirement.

We have proposed two VLA techniques to further increase the performance of division by functional iteration. First, reciprocal caches are often effective in reducing average division latency by exploiting redundant reciprocal computations. A reasonably-sized reciprocal cache in conjunction with a Goldschmidt divider can achieve a two-times speedup in average latency. However, while many applications contain a large quantity of redundant reciprocals, there exists those which do not. Further investigation may be required to analyze the underlying distributions in more detail. Second, while functional iteration leads to fast division implementations, these implementations suffer from the lack of a readily-available final remainder and the corresponding difficulty and latency penalty for exact rounding. We have therefore shown that by computing extra guard bits in the quotient the frequency of the latency penalty can be reduced quadratically in proportion to the added precision. The latency penalty itself can be reduced by replacing the required carry-propagate-addition with a comparison of two bits.

This dissertation demonstrates that VL functional units provide a means for achieving higher performance than can be obtained through FL-only implementations. The next generation of performance-oriented processors require flexible and robust micro-architectures to best exploit the performance achievable with VLA and VLC FPUs. As superscalar processors become more complex and move to higher widths of instruction issue, it becomes even more imperative that processors incorporate VLA functional units due to the increased exposure of the latencies of individual FP functional units.

Bringing together many of the themes of this dissertation, we suggest one possible organization of such a high performance FPU in figure 8.1. The pipelined FP adder in this FPU uses the proposed VLA addition algorithm. A pipelined VLA FP multiplier is shown that could be possibly designed along the lines of the VLA addition algorithm of chapter 3, although further research is required to determine the implementation details. Further, the multiplier allows for either full-width operands or two sets of half-width operands. Such an arrangement facilitates data-parallel operations desirable for many graphics applications. Division is computed through the Newton-Raphson iteration. This allows for more than one division operation to be in progress simultaneously, providing higher division throughput. Dedicated hardware is included to provide a very accurate initial reciprocal approximation. A small reciprocal cache returns frequently-computed reciprocals at a much lower latency. VLA exact rounding is supported in this implementation by using a multiplier with wider precision than is strictly required in order to compute several extra guard bits in the pre-rounded quotient estimate. These guard bits greatly increase the probability that the quotient can be exactly rounded without an additional back-multiplication.

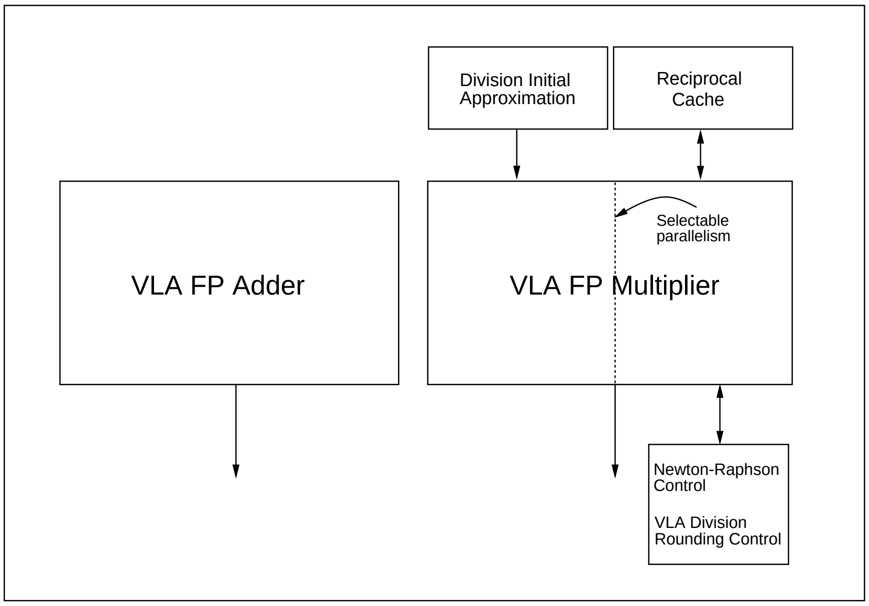


Figure 8.1: Example FPU implementation

## 8.2 Future Work

There are many aspects of floating point unit design left to be investigated. Several design issues in division remain unsolved. Currently, the computation of exactly rounded quotients still has a finite probability of incurring a latency penalty when implementing division by functional iteration. While it has been shown how this probability can be greatly reduced, it can not be eliminated for all operands. Further research would investigate techniques for computing exactly rounded quotients with no additional latency penalty for all combinations of operands.

The field of approximation theory as it relates to reciprocal table design continues to be an active area of research. Higher performance division is possible by having more accurate initial reciprocal and reciprocal square root approximations. The tables that provide these approximations still remain large. It should be possible to design even smaller tables with greater accuracy, without the addition of significant delay.

The trend in floating point formats continues to change. For 3D graphics applications, single precision operations typically provide sufficient range and precision. In contrast, scientific applications demand increasing amounts of precision, spawning the development of quad precision formats of 113 bits and more. The implications of both of these extremes on functional unit design are very important. The tradeoffs for these cases are most likely different than of those for the current standard of double precision. Investigating and understanding these tradeoffs is fundamental for designing future FPUs.

# Bibliography


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

[2] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[3] H. Altwaijry and M. Flynn, "Performance/area tradeoffs in Booth multipliers," Technical Report No. CSL-TR-95-684, Computer Systems Laboratory, Stanford University, November 1995.

[4] G. Bewick and M. J. Flynn, "Binary multiplication using partially redundant multiples," Technical Report No. CSL-TR-92-528, Computer Systems Laboratory, Stanford University, June 1992.

[5] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, March 1996.

[6] A. R. Omondi, *Computer Arithmetic Systems*, Prentice Hall, 1994.

[7] N. T. Quach and M. J. Flynn, "High-speed addition in CMOS," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1612–1615, December 1992.

[8] *Microprocessor Report*, various issues, 1994-96.

[9] NAS Parallel Benchmarks 8/91.

[10] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer performance evaluation and the perfect benchmarks," in *International Conference on Supercomputing*, June 1990, pp. 254–266.

[11] SPEC Benchmark Suite Release 2/92.

[12] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *to appear in IEEE Transactions on Computers*, 1997.

[13] DEC Fortran Language Reference Manual, 1992.

[14] J. C. Huck and M. J. Flynn, *Analyzing Computer Architectures*, IEEE Computer Society Press, Washington, D.C., 1989.

[15] M. D. Smith, "Tracing with pixie," Technical Report No. CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November 1991.

[16] S. Oberman, N. Quach, and M. Flynn, "The design and implementation of a high-performance floating-point divider," Technical Report No. CSL-TR-94-599, Computer Systems Laboratory, Stanford University, January 1994.

[17] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, November 1991.

[18] D. Wong and M. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 981–995, August 1992.

[19] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 2, pp. 98–105, February 1991.

[20] M. Darley, B. Kronlage, D. Bural, B. Churchill, D. Pulling, P. Wang, R. Iwamoto, and L. Yang, "The TMS390C602A floating-point coprocessor for Sparc systems," *IEEE Micro*, vol. 10, no. 3, pp. 36–47, June 1990.

[21] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very high radix division with selection by rounding and prescaling," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 909–918, August 1994.

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

[23] S. F. Oberman and M. J. Flynn, "A variable latency pipelined floating-point adder," in *Proceedings of Euro-Par'96, Springer LNCS vol. 1124*, August 1996, pp. 183–192.

[24] S. Waser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Holt, Rinehart, and Winston, 1982.

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

[26] E. Hokenek and R. K. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 71–77, January 1990.

[27] N. T. Quach and M. J. Flynn, "Leading one prediction - implementation, generalization, and application," Technical Report No. CSL-TR-91-463, Computer Systems Laboratory, Stanford University, March 1991.

[28] B. J. Benschneider et al., "A pipelined 50-Mhz CMOS 64-bit floating-point arithmetic processor," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1317–1323, October 1989.

[29] M. Birman, A. Samuels, G. Chu, T. Chuk, L. Hu, J. McLeod, and J. Barnes, "Developing the WTL 3170/3171 Sparc floating-point co-processors," *IEEE Micro*, vol. 10, no. 1, pp. 55–63, February 1990.

[30] P. Y. Lu, A. Jain, J. Kung, and P. H. Ang, "A 32-mflop 32b CMOS floating-point processor," in *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, 1988, pp. 28–29.

[31] D. Greenley et al., "UltraSPARC: the next generation superscalar 64-bit SPARC," in *Digest of Papers. COMPCON 95*, March 1995, pp. 442–451.

[32] P. Bannon and J. Keller, "Internal architecture of Alpha 21164 microprocessor," in *Digest of Papers COMPCON '95*, March 1995, pp. 79–87.

[33] N. T. Quach and M. J. Flynn, "An improved algorithm for high-speed floating-point addition," Technical Report No. CSL-TR-90-442, Computer Systems Laboratory, Stanford University, August 1990.

[34] N. Quach and M. Flynn, "Design and implementation of the SNAP floating-point adder," Technical Report No. CSL-TR-91-501, Computer Systems Laboratory, Stanford University, December 1991.

[35] L. Kohn and S. W. Fu, "A 1,000,000 transistor microprocessor," in *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, 1989, pp. 54–55.

[36] H. P. Sit, M. R. Nofal, and S. Kimn, "An 80 MFLOPS floating-point engine in the Intel i860 processor," in *Digest of Technical Papers, IEEE International Conference on Computer Design*, 1989, pp. 374–379.

[37] J. A. Kowaleski, G. M. Wolrich, T. C. Fischer, R. J. Dupcak, P. L. Kroesen, T. Pham, and A. Olesin, "A dual execution pipelined floating-point CMOS processor," in *Digest of Technical Papers, IEEE International Solid-State Circuits Conference*, 1996, pp. 358–359.

[38] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 196–205.

[39] D. W. Sweeney, "An analysis of floating-point addition," *IBM Systems Journal*, vol. 4, pp. 31–42, 1965.

[40] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *to appear in IEEE Transactions on Computers*, 1997.

[41] C. V. Freiman, "Statistical analysis of certain binary division algorithms," *IRE Proceedings*, vol. 49, pp. 91–103, 1961.

[42] J. E. Robertson, "A new class of digital division methods," *IRE Transactions on Electronic Computers*, vol. EC-7, no. 3, pp. 88–92, September 1958.

[43] K. D. Tocher, "Techniques of Multiplication and Division for Automatic Binary Computers," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 11, Pt. 3, pp. 364–384, 1958.

[44] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Transactions on Computers*, vol. C-17, no. 10, pp. 925–934, October 1968.

[45] K. G. Tan, "The theory and implementation of high-radix division," in *Proceedings of the 4th IEEE Symposium on Computer Arithmetic*, June 1978, pp. 154–163.

[46] M. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, no. 8, pp. 702–706, August 1970.

[47] P. Soderquist and M. Leeser, "An area/performance comparison of subtractive and multiplicative divide/square root implementations," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 132–139.

[48] M. D. Ercegovac and T. Lang, "Simple radix-4 division with operands scaling," *IEEE Transactions on Computers*, vol. C-39, no. 9, pp. 1204–1207, September 1990.

[49] J. Fandrianto, "Algorithm for high-speed shared radix 8 division and radix 8 square root," in *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, July 1989, pp. 68–75.

[50] S. E. McQuillan, J. V. McCanny, and R. Hamill, "New algorithms and VLSI architectures for SRT division and square root," in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, July 1993, pp. 80–86.

[51] P. Montuschi and L. Ciminiera, "Reducing iteration time when result digit is zero for radix 2 SRT division and square root with redundant remainders," *IEEE Transactions on Computers*, vol. 42, no. 2, pp. 239–246, February 1993.

[52] P. Montuschi and L. Ciminiera, "Over-redundant digit sets and the design of digit-by-digit division units," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 269–277, March 1994.

[53] P. Montuschi and L. Ciminiera, "Radix-8 division with over-redundant digit set," *Journal of VLSI Signal Processing*, vol. 7, no. 3, pp. 259–270, May 1994.

[54] N. Quach and M. Flynn, "A radix-64 floating-point divider," Technical Report No. CSL-TR-92-529, Computer Systems Laboratory, Stanford University, June 1992.

[55] H. Srinivas and K. Parhi, "A fast radix-4 division algorithm and its architecture," *IEEE Transactions on Computers*, vol. 44, no. 6, pp. 826–831, June 1995.

[56] G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," in *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, June 1985, pp. 64–71.

[57] T. Asprey, G. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter, "Performance features of the PA7100 microprocessor," *IEEE Micro*, vol. 13, no. 3, pp. 22–35, June 1993.

[58] D. Hunt, "Advanced performance features of the 64-bit PA-8000," in *Digest of Papers COMPCON '95*, March 1995, pp. 123–128.

[59] T. Lynch, S. McIntyre, K. Tseng, S. Shaw, and T. Hurson, "High speed divider with square root capability," U.S. Patent No. 5,128,891, 1992.

[60] J. A. Prabhu and G. B. Zyner, "167 MHz Radix-8 floating point divide and square root using overlapped radix-2 stages," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995.

[61] A. Svoboda, "An algorithm for division," *Information Processing Machines*, vol. 9, pp. 29–34, 1963.

[62] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Transactions on Computers*, vol. C-36, no. 7, pp. 895–897, July 1987.

[63] M. D. Ercegovac and T. Lang, "On-the-fly rounding," *IEEE Transactions on Computers*, vol. 41, no. 12, pp. 1497–1503, December 1992.

[64] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-point execution unit," *IBM Journal of Research and Development*, vol. 11, pp. 34–53, January 1967.

[65] D. L. Fowler and J. E. Smith, "An accurate, high speed implementation of division by reciprocal approximation," in *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, September 1989, pp. 60–67.

[66] R. E. Goldschmidt, "Applications of division by convergence," M.S. thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1964.

[67] Intel, i860 64-bit microprocessor programmer's reference manual, 1989.

[68] P. W. Markstein, "Computation of elementary function on the IBM RISC System/6000 processor," *IBM Journal of Research and Development*, pp. 111–119, January 1990.

[69] H. M. Darley et al., "Floating Point / Integer Processor with Divide and Square Root Functions," U.S. Patent No. 4,878,190, 1989.

[70] W. S. Briggs and D. W. Matula, "A 17x69 Bit multiply and add unit with redundant binary feedback and single cycle latency," in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, July 1993, pp. 163–170.

[71] D. Matula, "Highly parallel divide and square root algorithms for a new generation floating point processor," in *SCAN-89, International Symposium on Scientific Computing, Computer Arithmetic, and Numeric Validation*, October 1989.

[72] D. DasSarma and D. Matula, "Measuring the accuracy of ROM reciprocal tables," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 932–940, August 1994.

[73] D. DasSarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 12–25.

[74] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, July 1995, pp. 2–9.

[75] M. J. Schulte et al., "Optimal initial approximations for the Newton-Raphson division algorithm," *Computing*, vol. 53, pp. 233–242, 1994.

[76] E. Schwarz, "High-radix algorithms for high-order arithmetic operations," Technical Report No. CSL-TR-93-559, Computer Systems Laboratory, Stanford University, January 1993.

[77] E. Schwarz and M. Flynn, "Hardware starting approximation for the square root operation," in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, July 1993, pp. 103–111.

[78] T. Williams, N. Patkar, and G. Shen, "SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1215–1226, November 1995.

[79] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, July 1993, pp. 220–227.

[80] J. Cortadella and T. Lang, "High-radix division and square root with speculation," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 919–931, August 1994.

[81] N. Burgess and T. Williams, "Choices of operand truncation in the SRT division algorithm," *IEEE Transactions on Computers*, vol. 44, no. 7, pp. 933–937, July 1995.

[82] J. Fandrianto, "Algorithm for high-speed shared radix 4 division and radix 4 square root," in *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, May 1987, pp. 73–79.

[83] A. Barna and D. Porat, *Integrated Circuits in Digital Electronics*, John Wiley and Sons, 1973.

[84] Synopsys Design Compiler version v3.2b, 1995.

[85] LSI Logic lcb500k standard-cell library, 1994.

[86] S. F. Oberman and M. J. Flynn, "Reducing division latency with reciprocal caches," *Reliable Computing*, vol. 2, no. 2, pp. 147–153, April 1996.

[87] S. F. Oberman and M. J. Flynn, "On division and reciprocal caches," Technical Report No. CSL-TR-95-666, Computer Systems Laboratory, Stanford University, April 1995.

[88] E. Schwarz, "Rounding for quadratically converging algorithms for division and square root," in *Proceedings of 29th Asilomar Conf. on Signals, Systems, and Computers*, October 1995, pp. 600–603.

[89] H. P. Sharangpani and M. L. Barton, "Statistical analysis of floating point flaw in the pentium processor," Intel Corporation, November 1994.