Software Floating-Point Computation on Parallel Machines

by

Michael Ruogu Zhang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999
[June 1999]

Author ....
Department of Electrical Engineering and Computer Science
May 1, 1999

Certified by ....
Anant Agarwal
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Software Floating-Point Computation on Parallel Machines

by

Michael Ruogu Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis examines the ability to optimize the performance of software floating-point (FP) operations on parallel architectures. In particular, instruction level parallelism (ILP) of FP operations is explored, optimization techniques are proposed, and efficient algorithms are developed. In our method, FP operations such as FP add, are decomposed into a set of primitive integer and logic operations, such as integer adds and shifts, and the primitive operations are then scheduled on a parallel architecture. The algorithms for fast division and square root computation also enable the hardware FP unit to be clocked at a faster rate. The design and analysis of such a system is detailed and is tested on Raw, a software-exposed parallel architecture. Results show that division and square root implementations achieve reasonable performance compared to a hardware FP unit.

Thesis Supervisor: Anant Agarwal
Title: Professor

Acknowledgments

I would like to thank my thesis advisor, Professor Anant Agarwal, for giving me the opportunity and freedom to work on the Raw project. He has been a source of inspiration and direction.

I would also like to thank the members of the Raw group, especially Matt Frank, Mike Taylor, Jon Babb for their comments and suggestions; Walt Lee and Rajeev Barua for helping me getting started on rawcc and the timely bug fixes; Jason Kim for staying up many late nights with me and all the late night Chinatown runs.

Finally, my deepest love and appreciation go to my parents, who have always been there for me. Thank you so much, Mom and Dad!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most computer systems today that handle floating-point (FP) operations employ a hardware FP co-processor. FP operations are implemented through hardware to obtain high performance — software implementations are regarded as very slow. However, performing FP operations in parallel has been rarely considered. In our method, FP operations such as FP add, are decomposed into a set of primitive integer and logic operations, such as integer adds and shifts, and the primitive operations are then scheduled on a parallel architecture.

1.1 Motivation

As more and more transistors can be placed on the silicon, traditional superscalars will not be able to take advantage of the technological advances due to problems in scaling, wire length, bandwidth, and many other issues. One approach taken is to decentralize computing and storage resources [1, 2, 9]. Therefore, instruction level parallelism (ILP) and memory parallelism, as well as the scheduling of the instructions become increasingly important. The goal of this thesis is to study FP implementation in software and examine the trade-off between using software and hardware in a parallel architecture. The high level goal is to maintain high FP performance while minimizing area.

1.2 Approaches

The three central parameters in any FP unit design are latency, cycle time, and area. In order to optimize the performance in terms of these three parameters, the approaches that this thesis takes are the following:

1. Explore instruction level parallelism.
2. Compiler optimization.
3. Improve FP algorithms.
4. Provide hardware for expensive software operations.

This thesis presents the analysis, design, and implementation of a software FP unit. The basic FP operations, i.e., addition, subtraction, multiplication, division, and square-root, will be implemented.

The testbed for this software unit is Raw [1, 2, 9]. Raw is like a multiprocessor on a single chip. It is a highly parallel architecture with each processing element being a simple RISC-like chip and some on-chip reconfigurable logic. The processing elements are statically connected. A more detailed description of this architecture will be presented in Chapter 2.

Many FP units follow the IEEE 754 FP standard [6]. However, as a starting point, implementations will not fully follow this in order to leave more time for the investigation for other more interesting issues.

1.3 Organization

The following chapters detail the design of the software FP unit. They can be roughly separated into two parts. Part 1 introduces the basic FP representation and analyzes multiplication and addition by exploring parallelism in the algorithms, as well as discuss the trade-off between hardware and software through a case study. Part 2 mainly focuses on division and square root algorithms and develops two fast algorithms for division and square root. It then discuss the advantages of these two algorithms. Specifically, the chapters are organized as follows:

1. Part 1

   Chapter 2 briefly describes the Raw architecture, the target architecture of this software FP unit.

   Chapter 3 introduces the basic FP representation and, as a starting point, analyzes the parallelism in multiplication.

   Chapter 4 gives the analysis of addition and the discusses the trade-off between software and hardware. Furthermore, it presents a case study of the comparison between software and hardware implementations of the adder.

2. Part 2

   Chapter 5 presents the three major division algorithms and a comparison between them. In particular, the only parallel algorithm, Goldschmidt algorithm will be examined in detail.

   Chapter 6 presents a fast algorithm of calculating FP division in parallel, algorithm A. Comparison and advantages of algorithm A and Goldschmidt algorithm will be presented.

   Chapter 7 presents the current square-root algorithms. The parallel square-root algorithm, algorithm G, will be discussed in detail.

   Chapter 8 gives an fast alternative to calculate square-root in parallel, algorithm B. Comparison between algorithms G and B will be presented.

   Chapter 9 summarizes the result of this research.

Chapter 2

RAW Architecture

Advanced VLSI technologies enable more and more transistors to be placed on a single chip. In order to take advantage of the technology, microprocessors need to decentralize the resources and exploit instruction level and memory parallelism. Raw is a radically new approach which tries to take this advantage. This architecture is simple and easily scalable and exposes itself completely to the software so that fine-grained parallelism can be discovered and statically scheduled to improve performance [1, 2, 9].

The Raw microprocessor chip comprises a set of replicated tiles, each tile containing a simple RISC like processor, a small amount of configurable logic, and a portion of memory for instructions and data. Each tile has an associated programmable switch which connects the tiles in a wide-channel point-to-point interconnect. The compiler statically schedules multiple streams of computations. The interconnect provides register-to-register communication with very low latency and can also be statically scheduled. The compiler is thus able to schedule instruction-level parallelism across the tiles and exploit the large number of registers and memory ports. Raw also provides backup dynamic support in the form of flow control for situations in which the compiler cannot determine a precise static schedule. Figure 2-1 shows the structure of the microprocessor.

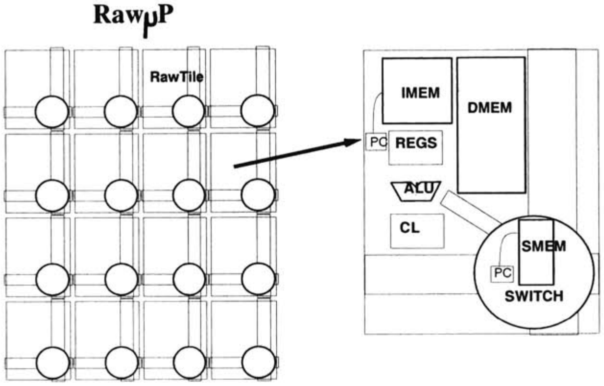

Figure 2-1: RawμP composition. Each RawμP comprises multiple tiles.

This thesis gives detailed analysis and design of how floating-point operations can take advantage of such a highly parallel architecture with configurable logic. Specifically, communication cost between the tiles will be considered. The implementations will also take advantage of the reconfigurable logic to perform some expensive software operations quickly.

Chapter 3

Floating-Point Representation and Multiplication

In this Chapter, the basics of floating-point representation is introduced. FP multiplication, the simplest of the FP operations, will also be presented and analyzed.

3.1 Floating-Point Representation

Floating-point numbers generally follow the IEEE 754 format and have several common precisions — single, double, and quadruple precisions. Certain architectures might support even higher precision. An ordinary single-precision FP number has 4 bytes and double, 8 bytes. This thesis limits its scope to single-precision FP numbers. Extension of FP algorithms for single precision to double precision is straight forward. FP numbers contains three fields — sign, mantissa, and exponent. The format is summarized in Table 3.1.

Two things in particular should be paid attention to. First of all, the exponent is offset by half of the exponent range. In single precision numbers, the exponent is offset by 127, and in double precision numbers, the exponent is offset by 511. Secondly, the mantissa bits contain only the fractional part of the value, that is, all mantissas are normalized and assume an inexplicit 1. This also means that the value of the significant will always be in the range $[1, 2)$. For example, given an decomposed single precision FP number with sign s, exponent e, and mantissa f, the actual value of the number is

	Sign	Exponent	Mantissa
Single-Precision	$< 31 >$	$< 30 : 23 >$	$< 22 : 0 >$
Double-Precision	$< 63 >$	$< 62 : 52 >$	$< 51 : 0 >$
Quad-Precision	$< 127 >$	$< 126 : 112 >$	$< 111 : 0 >$

Table 3.1: Floating-Point Number Representations

$$(-1)^s \times 1.f \times 2^{e-127}$$

In addition to the basic representations, the IEEE 754 standard specifies a variety of special exponent and fraction values in order to support the concepts of plus and minus infinity, plus and minus zero, "denormalized" numbers and "not-a-number" (NaN). Refer to Table 3.2 for the detailed representations. Special attention should be paid to denormal numbers. Denormal numbers are numbers whose values are very small thus cannot be normalized within the range of the exponent. They are supported by IEEE for gradual underflow.

Exponent	Mantissa	Value
-127	0	± 0
128	0	$\pm\infty$
128	$\neq 0$	NaN
-127	$\neq 0$	$f \times 2^{-127}$

Table 3.2: Special FP Values

In this thesis, the focus will be to explore the parallelism in the basic FP operations, as well as various optimizations that can be used to improve performance. Therefore, the IEEE 754 standard will be relaxed to leave more time and effort to explore the above. However, common exceptions such as overflow and underflow will be supported.

3.2 Floating-Point Multiplication

Among all FP operations, multiplication is the simplest. As a starting point, the basic steps involved in multiplication will be examined. Parallelism that exists in multiplication will be exploit in this section.

3.2.1 Basic Multiplication Steps

The basic steps of multiplication can be broken down into the following, assuming that numbers are given in IEEE format,

1. Unpack the operands, this step includes extraction of the sign, exponent, mantissa fields, as well as tagging on the inexplicit one for the mantissa.
2. Perform sign calculation, which is a simple `xor`.
3. Perform exponent calculation, which simply adds two exponents together.
4. Perform mantissa calculation, which is an integer multiplication.

5. Perform mantissa normalization.
6. Perform exception checks.
7. Pack results back into the IEEE format.

3.2.2 Parallelism

The parallelism in multiplication operation exists among sign calculation, exponent calculation, mantissa calculation. Figure 3-1 shows the data dependency diagram. From the data dependency digram, it is clear that it is a three-way parallelism. The darkened arrows forms the critical path of the calculation. The operations that must be performed sequentially are the significant multiplication, the normalization of the resulting significant, as well as the exponent adjustment. Theoretically, sign and exponent calculation can be placed on different processors. However, the number of cycles required to walk through the critical path is longer the the number of cycles required to calculate both sign and exponent. Therefore, it is not necessary to separate sign and exponent calculation onto different tiles.

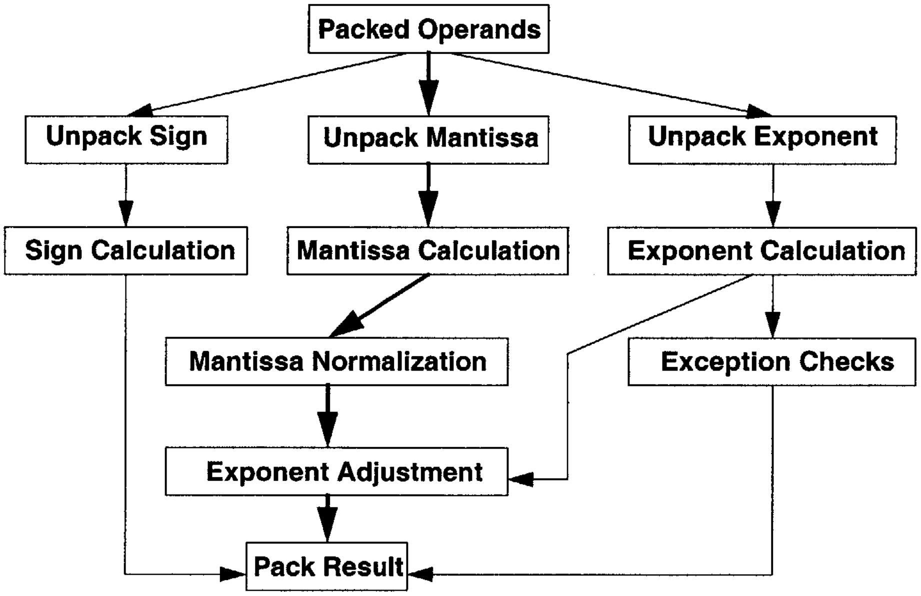

Figure 3-1: Data Dependency Diagram for Multiplication

3.2.3 Assembly Code Implementation

As described in the previous section, the parallelized multiplication can be implemented using two Raw tiles. Table 3.3 shows the trace of the execution. This implementation closely follow the algorithm described in the last section. It also takes into consideration the cost of communication. The code is describe in detail in this section. In future sections, focus will be placed on analyzing algorithms and ILPs instead of describing assembly code.

There are a few parts of the implementation should be paid attention to, the rest will be very easy to follow given the algorithm. First of all, during sending and receiving data over the static network, the instruction can compute and send/receive in one step in some cases. For example, computing sign of `a` × `b` requires four instructions, assuming `$4` contains the value of `a` and `$5` contains the value of `b`.)

```
xor $8, $4, $5
srl $8, $8, 31
sll $8, $8, 31
or $2, $2, $8
```

Theoretically, these instruction cannot be parallelized since they have direct data dependency one after another. But using communication instructions and clever scheduling, these computation can easily be hidden and thus be able to reduce the latency in half. Most branch delay slots are utilized also to reduce latency.

Code and Comments

cycle	tile 0	communication	tile 1
1	`add $csto,$4,$0`	a → tile 1	`addiu $at,$0,127`
2	`add $csto,$5,$0`	b → tile 1	`sll $4,$csti,1`
3	`xor $csto,$4,$5`	a xor b → tile 1	`sll $5,$csti,1`
4	`lui $at,32768`		`srl $8,$csti,31`
5	`sll $14,$4,8`		...
6	`or $14,$14,$at`		...
7	`sll $25,$5,8`		`srl $12,$12,24`
8	`or $25,$25,$at`		`srl $11,$11,24`
9	`multu $14,$25`		`add $11,$11,$12`
10	`lui $at,128`		`subu $11,$11,$at`
11	`lui $24,0x7F00`		`blez $10,main.UNDER`
12	...		`lui $at,32767`
13	`mfhi $15`	tile 0 ← `z.Exp`	`sll $csto,$11,23`
14	`srl $16,$15,31`	tile 0 ← `z.Sign`	`sll $csto,$8,31`
15	`bne $16,$0,main.L1`		...
16	`sll $15,$15,1`		...
17	`add $2,$csti,$at`		...
18	`blt $2,$24,main.OVER`		...
19	`srl $15,$15,9`		...
20	`or $2,$2,$15`		...
21	`or $2,$2,$csti`		...

Table 3.3: Execution Trace of Multiplication

On tile 0, first three instructions send over the data to tile 1. Instructions 4 to 8 unpack the mantissa and tags on the inexplicit 1. Instructions 9 to 13 perform the integer multiplication of the significants. Instructions 14 to 16 normalize the resulting significant. Instructions 17 to 18 performs exponent adjustment and checks for overflow exception. Instructions 19 to 21 pack the result.

On tile 1, all necessary data are received after cycle 6. Instructions 7 and 8 unpack the exponents. Instructions 9 and 10 calculate exponent. Instructions 11 and 12 check underflow exception. Lastly, resulting exponent and sign are sent to tile 0.

Performance

The above execution trace requires 21 cycles using two tiles. If multiplication is to be done sequentially, 30 cycles are required. Therefore, parallelism in multiplication actually can achieve approximately 30% speed-up. It turns out that multiplication has a good amount of parallelism compared to the rest of the operations. In the next chapter, we will examine addition as well as some of the optimizations might be useful.

Chapter 4

Floating-Point Addition and Subtraction

In this section, floating-point addition and subtraction algorithm is examined. Parallelism is explored. As a case study, at the end of this chapter, a generic hardware floating-point adder is presented and compared to the software version so that some of the issues that exist in deciding whether to choose customized hardware will become clear.

4.1 Brief Review of Addition Algorithm

Most FP addition algorithms are generically performed in the following steps. Figure 4-1 shows the flowchart of this algorithm. Assuming that operands are already unpacked.

1. Compute exponent difference `expDiff` = `a.Exp` - `b.Exp`. If `expDiff` <0 then swap `a` and `b`. If `expDiff` $=0$ and `a.Sig` $<$ `b.Sig`, also swap `a` and `b`. This step makes sure that first operand is greater than the second one.
2. If `a.Sign` $\neq$ `b.Sign`, convert `b.Sig` to its two's complement format and negate it. By always turning the second operand into negative in a subtraction, the resulting mantissa will be positive, thus eliminates the need to check for polarity after addition is performed.
3. Shift `b.Sig` to the right by `expDiff`, aligning the significants.
4. Perform `z.Sig` = `a.Sig` + `b.Sig`. Notice `z.Sig` will always be positive.
5. Count Leading Zeros in `z.Sig` and perform normalization — mantissa shift and exponent adjustment.
6. Check for Overflow/Underflow.
7. Perform rounding and pack results.

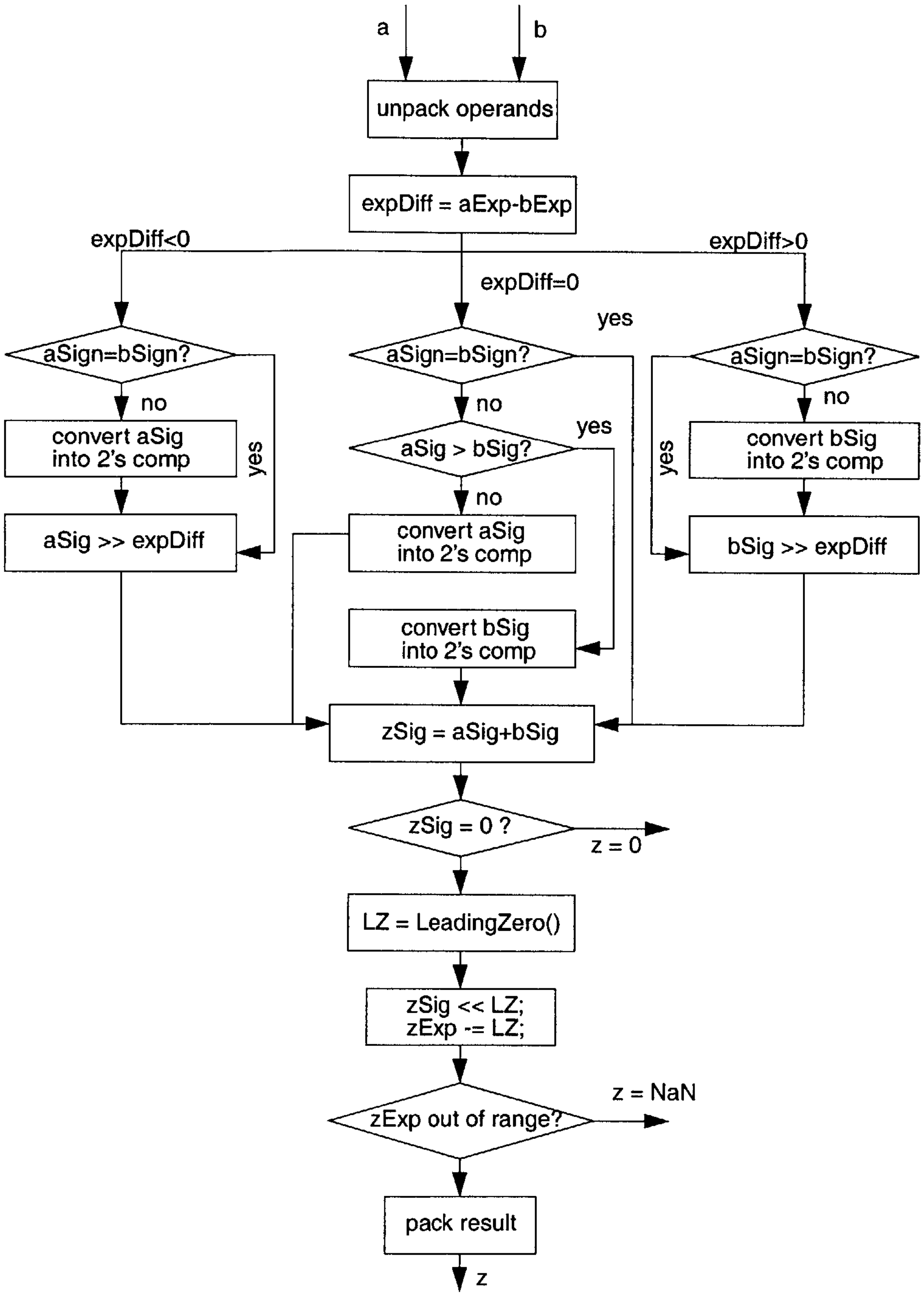

Figure 4-1: Flowchart for Floating-Point Addition

4.2 Parallelism in Floating-Point Addition

Unlike multiplication, there is little parallelism in FP addition. The data dependency diagram for the computation is shown in Figure 4-2. The dotted arrow lines are the imaginary execution traces that can be run in parallel. We notice that the computation on tile 1 is the critical path of the entire addition. Most of the work is directly on the critical path. Only a few operations can be performed in parallel. For example, sign comparison and exponent subtraction can be performed in parallel; exponent adjustment and mantissa normalization can also be performed in parallel. However, each of these operation costs at most two cycles. Weighing in the communication cost, it is actually faster and much simpler to keep computation on the same tile.

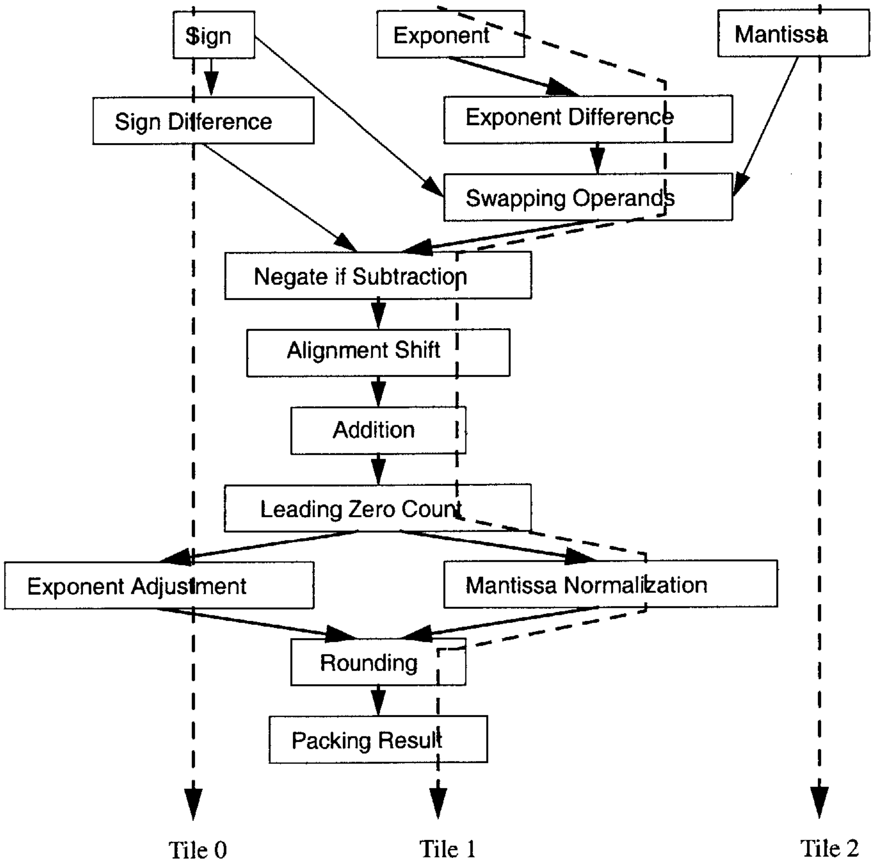

Figure 4-2: Hardware Floating-Point Adder Pipelined

4.3 Software Floating-Point Addition

Software implementation of FP addition follows the algorithm described in the Section 4.1. Refer to Figure 4-1 for the flowchart of the code. We first assume that `a` and `b` are stored in registers `$4` and `$5` in standard IEEE format.

4.3.1 Detailed Implementation

1. Unpack Operands

   The following seven instructions unpack the mantissa and tag on the inexplicit leading one. The values are right shifted by 2 bits to both avoid overflow in the adding step as well as to leave the most significant bit as the sign bit after optional conversion of operands to two's complement format.

   ```
   lui $at,8192
   sll $24,$4,9
   srl $24,$24,3
   or  $24,$24,$at
   sll $25,$5,9
   srl $25,$25,3
   or  $25,$25,$at
   ```

   The next four instructions unpacks the exponent by a left shift to shift off the sign bit followed by a right shift.

   ```
   sll $11,$4,1
   srl $11,$11,24
   sll $10,$5,1
   srl $10,$10,24
   ```

   Unpacking sign bits takes two cycles.

   ```
   srl $9,$4,31
   srl $8,$5,31
   ```

2. Compute exponent difference

   ```
   subu $12,$11,$10
   ```

3. We need to consider three cases of exponent difference, namely, `expDiff` > 0, `expDiff` $= 0$, and `expDiff` < 0. The first and the last cases are the same if we swap the operands. We choose to take two different paths in the code to avoid swapping the unpacked form of the operands which would involve the swapping of signs, significants, and exponents.

4. We will look at the path taken when `expDiff` > 0. We compare the sign of the the operands, if they are different, we know this is a subtraction and thus negate the smaller of the significants and place it in two's complement form. If signs are the same, we directly jump to shifting of the operands.

   ```
   # comparing signs
   beq   $9,$8,main.GTSIGNEQ
   # conversion to 2's complement
   lui   $at,65535
   ori   $at,$at,65535
   xor   $25,$25,$at
   addiu $25,$25,1
   ```

5. This step shifts the smaller of the significants to align the two's complement form decimal point for addition. The shift is an arithmetic shift to maintain the polarity of the significant.

   ```
   main.GTSIGNEQ:
   srav $25,$25,$12
   ```

6. Perform addition.

   ```
   addu $2,$24,$25
   ```

7. This step tests for the special case that the result is zero. The result is zero if and only if the resulting significant is zero. If so, return zero and exit the program.

   ```
   beq   $2,$0,main.ZERO
   addiu $at,$0,3
   ```

8. If the value is not zero, we need to normalize the result, i.e., shift off all the leading zeros. Currently counting leading zero is done in the reconfigurable logic. A straight forward implementation of leading zero count would cost around 20 cycles. The leading 1 is also shifted out in the same step. In the future, an ALU instruction could be implemented to perform leading zero count. The number of digits needs to be shifted off is placed in register $14.

   ```
   # version without RCL would cost ~ 20 cycles
   or    $rlo,$2,$0
   addiu $14,$rli,1
   ```

9. Perform normalization and adjust exponent. This include shifting off all the leading zeros and subtract the number of leading zeros to the exponent. The leading 1 is also shifted out in the same step.

```
sllv $2,$2,$14
sub  $14,$14,$at
sub  $11,$11,$14
```

10. Performing checks to overflow and underflow. If the exponent is out of the range of [1, 254], an overflow or underflow occurred and we return `NaN` as result.

```
slti  $10,$11,0x000000FF
slt   $12,$0,$11
and   $10,$10,$12
beq   $10,$0,main.OVERFLOW
lui   $at,32767
```

11. Lastly, pack the result into standard IEEE format.

```
srl  $2,$2,9
sll  $7,$9,31
sll  $11,$11,23
addu $11,$7,$11
addu $2,$11,2
```

4.3.2 Cycle Count Summary

The total cycle count for this path is 60 cycles. Out of the 60 cycles, 20 cycles are spent counting leading zeros, 18 cycles are spent unpacking and packing the operands and result. For the remaining cycles, 4 cycles are spent determining which path the code should take; 1 cycle is spent computing exponent difference; 5 cycles are spent checking whether the operation is actually a subtraction, and if so, negate the smaller operand and put into two's complement form; 2 cycles are spent shifting the smaller operands and adding; 7 cycles are spent checking for zero and over/underflow exception; 3 cycles are spent normalizing the resulting significant and adjusting the exponent. The cycle count can be summarized in Table 4.1.

When the exponents are equal and signs are different, mantissas are compared to decide which operand to negate. In this algorithm, the smaller one is, which produces a positive result. The cycle count for this path is 62, which includes the 2 cycles to compare mantissa of the operands.

4.4 Possible Optimization on Addition

Since there is little parallelism in FP addition, we have to look for other means of optimization. There are two major improvements that could be achieved and they will be presented in turn.

Function	Cycle Count	% of Total Cycle Count
Count Leading Zero	20	33%
Packing/Unpacking	18	30%
Exception Checks	7	12%
Computing 2's Comp.	5	8%
Choosing Path	4	7%
Normalizing Result	3	6%
Mantissa Alignment Shift	2	3%
Compute Exp. Difference	1	1%
Total Cycle Count	60	100%

Table 4.1: Cycle Breakdown of Floating-Point Addition

4.4.1 Compiler Optimization to Eliminate Pack/Unpack

From Table 4.1, we notice that the effort spent on packing and unpacking takes approximately 30% of the work. It is obvious that some of the packing and unpacking is unnecessary because the result of one FP operation will be the input of another. We will give an example to demonstrate this point.

Example: Compute the length of the hypotenuse of a right triangle with sides of length a and b:

$$H = \sqrt{a^2 + b^2}$$

This is a very common operation in scientific computation or graphics computation. Table 4.2 shows the comparison between the naive implementation and an implementation with desired optimization.

From this example, it is very clear that the naive approach is doing much wasteful work by packing and unpacking the intermediate results. Since packing and unpacking takes a significant part of the calculation, this optimization could extremely useful. The naive approach takes about 145 cycles to execute the above operations, where as the optimized version takes about 73 cycles, which is a 2X speed up.

4.4.2 Providing Special Hardware for Expensive Software Operations

A second performance improvement could be made possible if a small amount of special purpose hardware can be provided to perform operations that cost heavily in software. The count leading zero operation is a very good example of this optimization. Normally, count leading zero could take as many as 25 cycles to perform directly in software, which takes more than one third of the cycles count in addition. However, if we can implement this operation using special purpose hardware, possibly in reconfigurable logic, latency is greatly reduced.

Naive Implementation	Optimized Implementation
unpack a	unpack a
unpack a	unpack b
compute a^2	compute a^2
pack $t_1 = a^2$	compute b^2
unpack b	compute $a^2 + b^2$
unpack b	compute $H = \sqrt{a^2 + b^2}$
compute b^2	pack H
pack $t_2 = b^2$	
unpack t_1	
unpack t_2	
compute $t_3 = t_1 + t_2$	
pack t_3	
unpack t_3	
compute $H = \sqrt{t_3}$	
pack H	

Table 4.2: Effect of Optimization of Pack/Unpack: A Comparison

Other than count leading zero, exception checks and rounding are also good places to put in small amount of hardware to trade significant reduction in latency.

4.5 Case Study: Hardware FP Adder

In this section, a hardware implementation of the FP adder is analyzed to explain the difference in performance compared to software. The hardware FP adder also roughly follows the algorithm described in Section 4.1. A straight forward implementation of the hardware FP adder is shown in Figure 4-3. The components of this circuit and their function will be described in Section 4.5.1.

4.5.1 Function Description

1. Sign Comparator
   Input: `a.Sign`, `b.Sign`
   Output: `signDiff`
   Explanation: It determines whether two operands have different signs.

2. Exponent Difference
   Input: `a.Exp`, `b.Exp`
   Output: `expDiff`
   Explanation: It computes the difference between the two exponents.

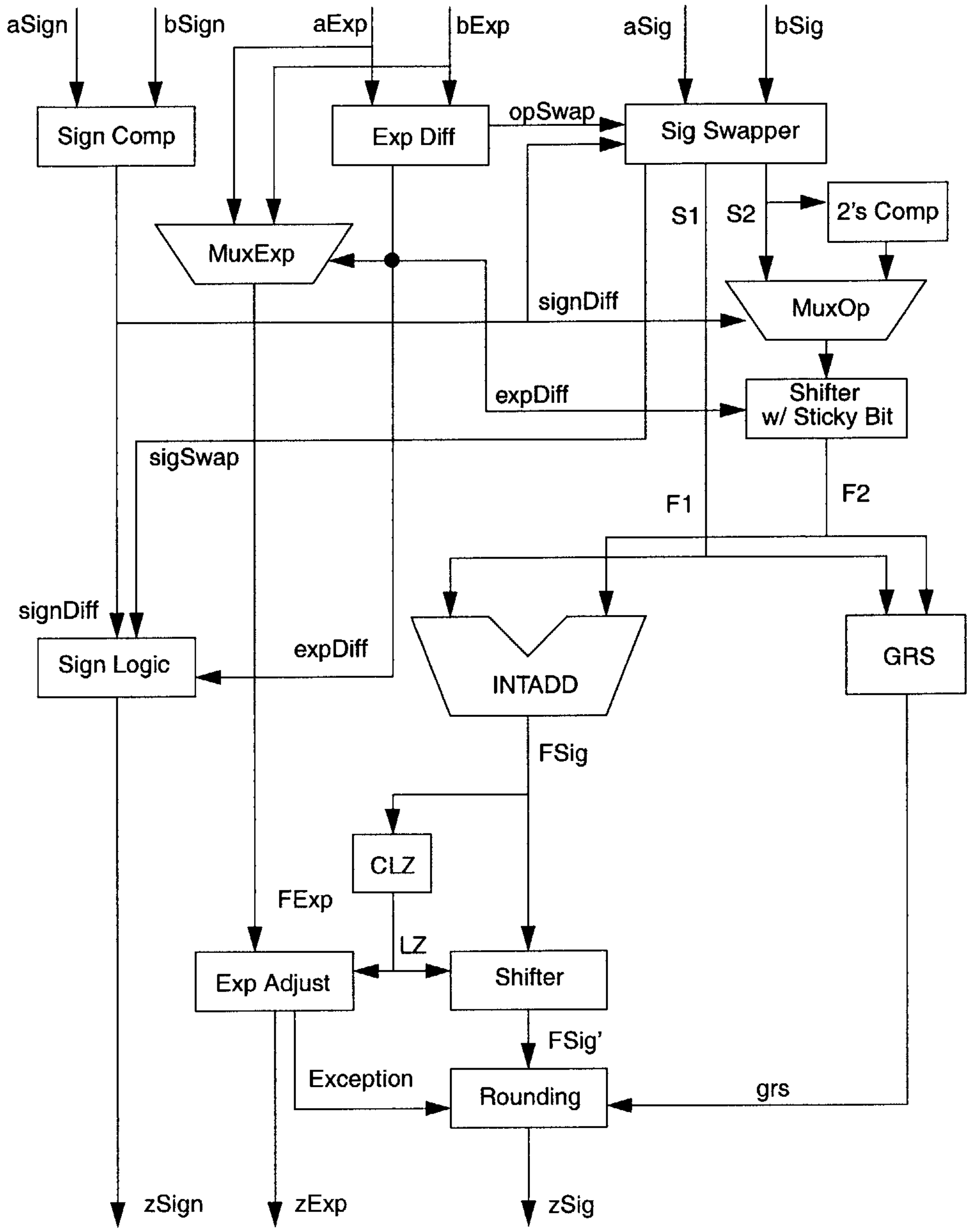

Figure 4-3: A Generic Hardware Floating-Point Adder

`expDiff`	`signDiff`	`a.Sig` vs. `b.Sig`	Swap?
< 0	X	X	Yes
> 0	X	X	No
$= 0$	Yes	`a.Sig` < `b.Sig`	Yes
$= 0$	No	X	No

Table 4.3: Rules for Swapping the Operands

3. Significant Swapper
   Input: `a.Sig`, `b.Sig`, `expDiff`, `SignDiff`
   Output: `S1`, `S2`, `sigSwap`
   Explanation: Produces `S1` and `S2` according to the following table. Sets `sigSwap` bit when the operand is swapped.

4. Sticky Shifter
   Input: `expDiff`, `S2`
   Output: `F2`
   Explanation: It shifts `F2` to the right by `expDiff` bits and retains the sticky bit.

5. Integer Adder
   Input: `F1 F2`
   Output: `FSig`
   Explanation: It adds the two inputs.

6. GRS
   Input: `F1 F2`
   Output: `grs`
   Explanation: Determines rounding information.

7. CLZ
   Input: `FSig`
   Output: `LZ`
   Explanation: It counts the number of leading zeros in the adder output.

8. Shifter
   Input: `LZ`, `FSig`
   Output: `FSig'`
   Explanation: It shifts off the leading zeros as well as the first one.

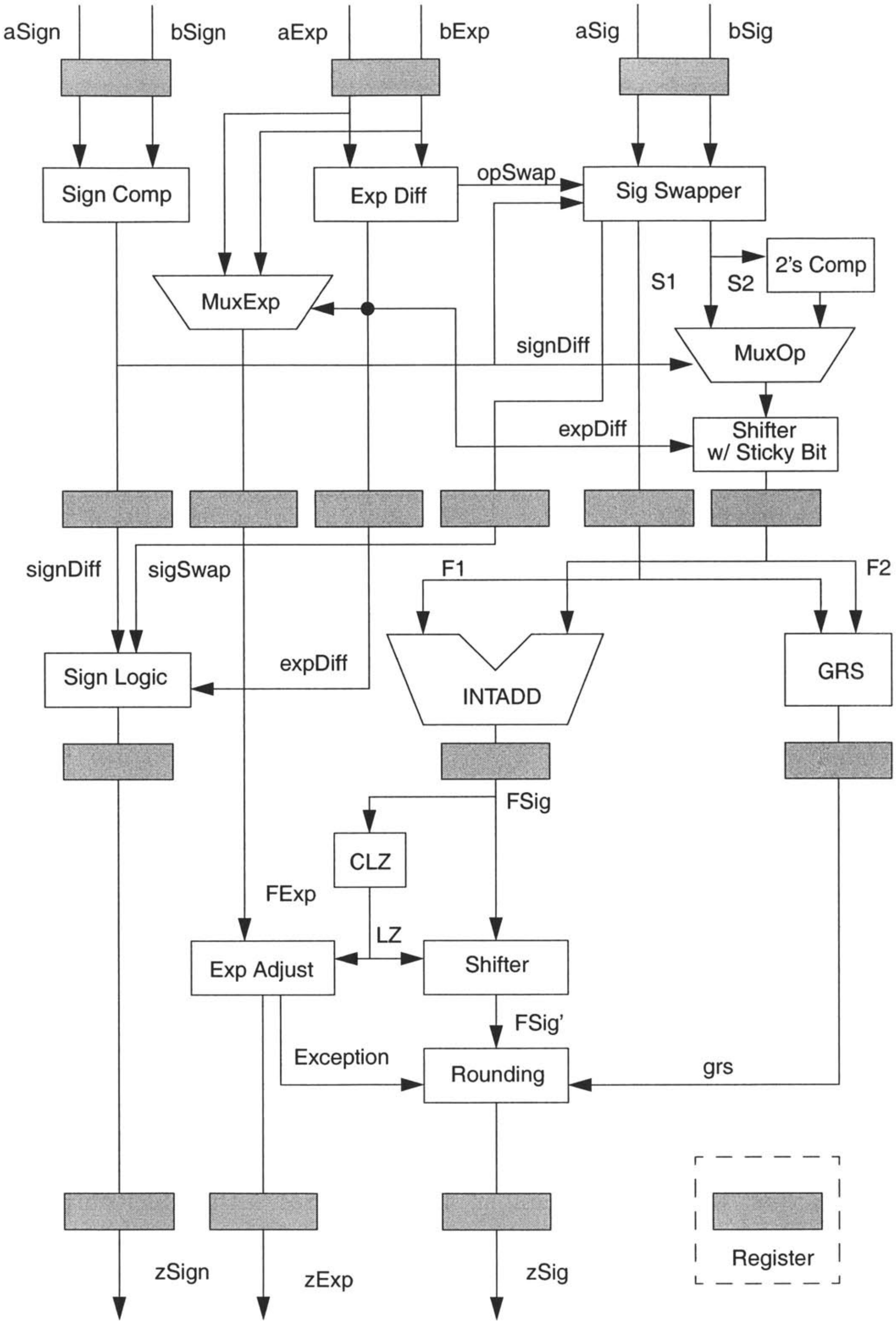

Figure 4-4: Hardware Floating-Point Adder Pipelined

9. Exponent Adjust
Input: `LZ`
Output: `zExp`, `Exception`
Explanation: It adjusts the resulting exponents according to the number of leading zeros. Exceptions are raised when exponent is out of range after shifting.

10. Sign Logic
Input: `expDiff`, `sigSwap`
Output: `z.Sign`
Explanation: It determines the sign of the result.

11. Rounder
Input: `FSig'`, `Exception`, `grs`
Output: `z.Sig`
Explanation: It does the proper rounding and produces the final significant.

4.5.2 Pipelined Adder

The adder described in the last section can be easily pipelined into three stages to increase the throughput. During the first stage, all the swapping of operands and mantissa alignment are done. During the second stage, addition is performed and rounding information is determined. The last stage does the leading zero count, exponent adjust, normalization of mantissa, as well as rounding. The pipelined block diagram is shown in Figure 4-4. Most commercial architectures have fully pipelined adders with 3 cycle of latency.

4.6 Summary

Addition is the most performed FP operations in most applications. Much research has been done to optimize the latency. After presenting the two different approaches, it is clear that performance difference between the software and hardware is a simple area-performance trade-off. Much specialized hardware is dedicated to perform addition. For example, implementing a hardware leading zero counter that uses one cycle would save around 20 cycles if this operation is done in hardware. Having a sticky shifter would save 6 cycles if there is none. Unpacking and packing which is required in software is completely unnecessary in hardware. There is also little parallelism in FP additions to take advantage of parallel machines.

Chapter 5

Floating-Point Division Algorithms

5.1 Introduction

Previous chapters show that there is still a large gap between software and hardware implementations of addition and multiplication. In most designs, multiplication and addition units are very carefully designed to have high performance since they are very frequently used. On the other hand, division and square root operation are much less frequently used since they are inherently slow operations and take much longer to execute. In the remaining portion of this thesis, focus will be placed on parallel floating-point division and square root algorithms and implementations to pursue software implementations that at least achieve comparable performances of the hardware. Since the percentage of the floating-point division and square root is small, comparable software performance makes it possible for the designers to consider not to have hardware floating-point divider and square-root units.

As the result of this research, two fast parallel algorithms are developed, one for division and one for square-root. Comparison between these two algorithms and the two existing parallel algorithms will be presented in detail.

5.2 Common Approaches To Division

There are many different approaches in calculating floating-point division. Each one might have a different underlying mathematical formulation, or different convergence rate, or different hardware implementation challenges. In this section, a few common approaches to floating-point division will be briefly described.

Most division algorithms use reciprocal approximation to compute the reciprocal of the divisor then multiply by the dividend. This section briefly describes some of these algorithms and compares the performance. In Section 5.2.1, a functional iterative algorithm, the Newton-Raphson algorithm is presented; in Section 5.2.2, a digit recurrence algorithm, the SRT algorithm is presented; in Section 5.2.4, a parallel algorithm, the Goldschmidt, algorithm is presented.

5.2.1 Newton-Raphson Algorithm

The classic Newton-Raphson algorithm is an iterative root-finding algorithm. Equation 5.1 is the recurrence equation for finding a root of $f(x)$ and is demonstrated in Figure 5-1.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{5.1}$$

The Newton-Raphson algorithm is a quadratically converging algorithm, i.e., the number of significant digits doubles during each iterations. It is commonly used if the result does not require proper rounding. In calculating $1/D$,

$$f(x) = 1/x - D. \tag{5.2}$$

Therefore, the iteration equation is

$$x_{i+1} = x_i + \frac{\frac{1}{x_i} - D}{\frac{1}{x_i^2}} = x_i(2 - D * x_i) \tag{5.3}$$

The algorithm is, for $Q = N/D$

1. *initialization step:* $P_0 = 0.75$. This is the initial guess, which is the mid-point of the range of $1/D \in (0.5, 1]$.
2. *iteration step:* $P_{i+1} = P_i(2 - D * P_i)$.
3. *final step:* $Q = P_\infty \times N$

This algorithm has a quadratic convergence rate with initial guess having two bits of accuracy. Therefore, in order to achieve single precision, four iterations are necessary, for double, five iterations, and for quadruple, six iterations.

$$2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128$$

Each iteration in this algorithm contains two multiplies, one subtraction, and one two's-complementation.

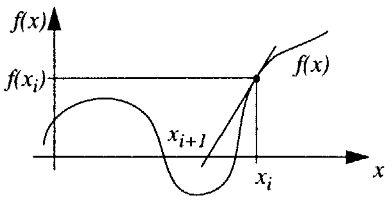

Figure 5-1: Newton's Iterative Method of Root Finding

5.2.2 SRT Algorithm

SRT division algorithm is a common digit recurrence division algorithm. It is also iterative with each iteration reduce a fixed number of quotient bits. SRT is very simple to implement but has a relatively long latency. We call a SRT algorithm *radix-r* SRT algorithm if the number of precision digits produced after each iteration is r.

The basic idea behind SRT is to use a trial and error process. We first estimate the first quotient digit and then obtain a partial remainder. If the partial remainder is negative, we successively try smaller digits until the remainder turns positive. This is also the reason why SRT has a longer latency. Higher radix SRT algorithms will need to do less iterations but each iteration will take longer. We call a SRT algorithm radix-r when each iteration produces $\lg n$ significant bits.

Assume that we are to implement a radix-r SRT algorithm for N/D. The initial look-up for the initial guess of the partial remainder is

$$P_0 = N \tag{5.4}$$

During iteration, $\lg r$ bits are produced. These bits are determined through by a quotient-digit function. It is a function of three parameters, radix, partial remainder, and divisor:

$$q_{i+1} = F(rP_i, D) \tag{5.5}$$

and to obtain the next partial remainder, the iterative formula is:

$$P_{i+1} = rP_i - q_{i+1}D \tag{5.6}$$

The final result is the sum of all the quotient bits properly shifted:

$$Q = \sum_{i=1}^{k} q_i \times r^{-i} \tag{5.7}$$

There are a few parameters we can change to gain better performance. The simplest of them is to increase the radix of the algorithm, therefore, reduces the number of iterations necessary. However, it should noted that as the radix increases, the latency of each iteration also increases because the quotient selection function becomes much more complex, therefore lengthen the critical path as well as the practicality of the hardware for the selection function becomes infeasible. Therefore, what is the best radix to use is decision which should be based on individual cases.

The most complex part of the algorithm is the quotient bits selection function. The function selects a quotient from a quotient digit set that is composed of symmetric signed-digits. For example, if we are producing $b = \lg r$ bits per iteration, the quotient set would contain:

$$Quotient - Set = \{-r+1, -r+2, \cdots, -1, 0, 1, \cdots, r-2, r-1\} \tag{5.8}$$

We will not detail the quotient selection function but rather give an example of a radix-2 SRT algorithm. For a radix-2 algorithm, the quotient selection function is:

$$q_i = 1 \tag{5.9}$$
$$q_i = 0 \tag{5.10}$$
$$q_i = \bar{1} \tag{5.11}$$

Example: Use radix-2 algorithm to compute N/D where

$$\begin{aligned} N &= 0.0011001_2 \\ &= \left(\frac{21}{128}\right)_{10} \\ D &= 0.011_2 \\ &= \left(\frac{3}{8}\right)_{10} \end{aligned}$$

1. Iteration #0:

$$\begin{aligned} P_0 &= N \\ 2P_0 &= 0.0011001 << 1 \\ &= 0.0110010 \end{aligned}$$

According to the quotient selection function, we assign $q_1 = 0$.

2. Iteration #1:

$$\begin{aligned} P_1 &= 2P_0 - q_0 D \\ &= 0.0110010 \end{aligned}$$

Thus,

$$2P_1 = 0.1100100 \tag{5.12}$$

we assign $q_2 = 1$.

3. Iteration #2:

$$\begin{aligned} P_2 &= 2P_1 - D \\ &= 0.0110100 \end{aligned}$$

$$2P_2 = 0.110100 \tag{5.13}$$

leads to $q_3 = 1$.

4. Iteration #3:

$$\begin{aligned} P_3 &= 2P_2 - D \\ &= 0.011100 \end{aligned}$$

$$2P_3 = 0.111000 \tag{5.14}$$

leads to $q_4 = 1$.

5. Iteration #4:

$$\begin{aligned} P_4 &= 2P_3 - D \\ &= 0.000000 \end{aligned}$$

Therefore, the quotient is $0.011100_2 = \left(\frac{7}{16}\right)_{10}$ and the remainder is zero.

5.2.3 Hybrid of Newton-Raphson and SRT and Use of Look-Up Table

Many practical division algorithm has been a hybrid of the Newton-Raphson and SRT algorithms. Moreover, an initial look-up table is very frequently used to give initial estimates of the reciprocal. Look-up tables usually saves one to two iterations in the beginning with a reasonable size. High radix SRT algorithms have also been heavily studied. For example, in Cyrix floating-point coprocessors, two iterations of Newton-Raphson were used with an initial look-up table, then a high-radix SRT is used to obtain the final result.

5.2.4 Goldschmidt Algorithm

After examining Newton-Raphson and SRT algorithms, it should be noticed that both algorithms are completely sequential, i.e., the data necessary to start each new iteration can only be obtained after the previous iterations is done. There is very little ILP to be explored in these algorithms. Goldschmidt algorithm, on the other hand, can be run in parallel. This algorithm was proposed by R. E. Goldschmidt [4].

The Algorithm

The basic idea of the algorithms is to keep multiplying the dividend and divisor by the same value and converge the divisor to one, thus the dividend converges to the quotient:

$$\begin{aligned} Q &= \frac{N_0}{D_0} \\ &= \frac{N_0}{D_0} \times \frac{P_1}{P_1} \\ &= \frac{N_0}{D_0} \times \frac{P_1}{P_1} \cdots \times \frac{P_i}{P_i} \\ &= \frac{N_i}{D_i} \end{aligned}$$

where $D_i \Rightarrow 1$ and P_i is the correction factor calculated in iteration i.

We now describe Goldschmidt algorithm. Assume that we are calculating N/D. First of all, an initial table look-up is used to give an rough estimate of the reciprocal of the divisor.

$$P_0 \approx \frac{1}{D} = \frac{1}{D} + E_0 \tag{5.15}$$

Term E_0 represents the error/difference between the actual value and the value obtained from the look-up table. We then multiply D by P_0 to obtain

$$\begin{aligned} D_0 &= D \times P_0 \\ &= D \times (\frac{1}{D} + E_0) \\ &= 1 + D * E_0 \\ &= 1 + X_0 \end{aligned} \tag{5.16}$$

where X_0 is the new error term and P_0 is the correction term. We then multiply the dividend by P_0 also, obtaining:

$$\begin{aligned} N_0 &= N \times P_0 \\ &= \frac{N}{D} \times (1 + X_0) \end{aligned} \tag{5.17}$$

Next step is the key step in the algorithm. The goal of this algorithm is to iteratively reduce error term X_i and when X_i is close enough to 0, N_i is the quotient. To reduce X_i, a simple equality is used: $(1+\epsilon)(1-\epsilon) = 1 - \epsilon^2$. When $\epsilon < 1$, the value reduces quadratically. Therefore, the error correction term P_{i+1} is:

$$P_{i+1} = 2 - D_i = 2 - (1 - X_i) = 1 + X_i$$

A clever way to eliminate the need of this subtraction is to use two's complementation. When the value of x is between 1 and 2, $2 - x = \bar{x}$ where $\overline{x}$ represents the two's complementation of x.

Example: Compute $2 - x$ when $x = 1.75_{10} = 1.110_2$

Solution: $2 - x = \overline{x} = 0.001_2 + 0.001_2 = 0.010_2 = 0.25_{10}$

Therefore, putting everything together, the algorithm runs as follows:

1. *Initialization Step:*

$$\begin{aligned} P_0 &\approx \frac{1}{D} \\ D_0 &= D \times X_0 \\ N_0 &= N \times X_0 \end{aligned}$$

2. *Iteration Step:*

$$\begin{aligned} P_{i+1} &= 2 - D_i \\ N_{i+1} &= N_i \times P_{i+1} \\ D_{i+1} &= D_i \times P_{i+1} \end{aligned}$$

3. $Q = N_\infty$

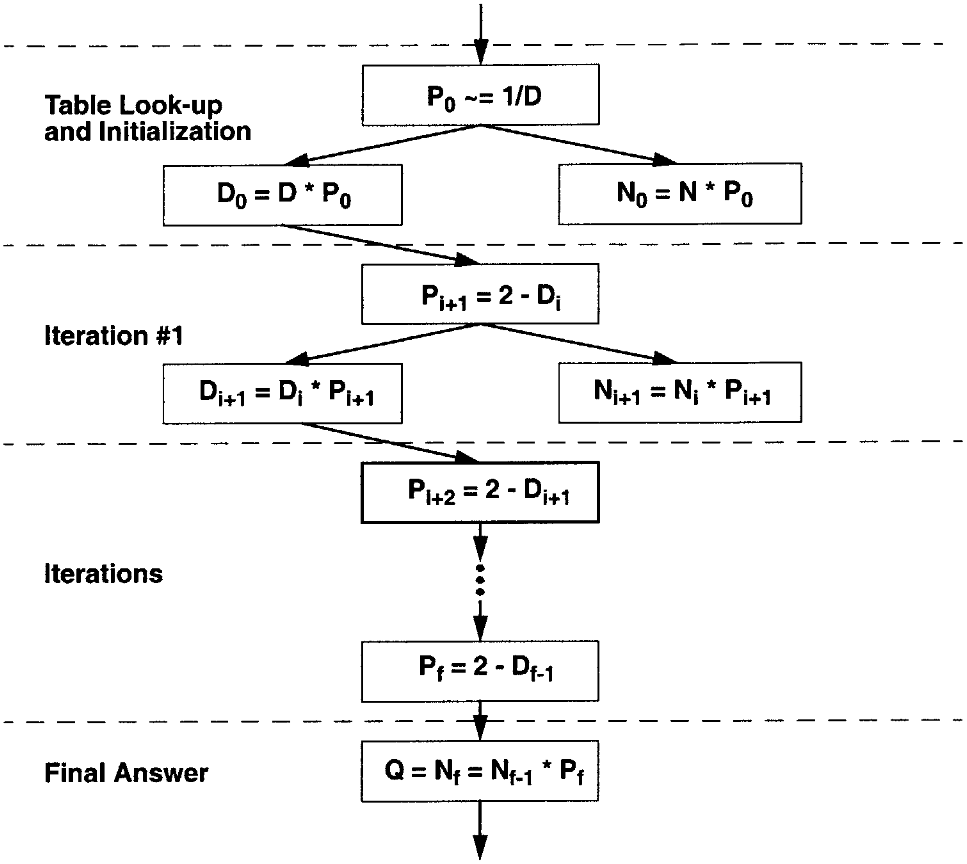

Figure 5-2: Data Dependency Diagram in Goldschmidt Algorithm

Parallelism in the Algorithm

Unlike to Newton-Raphson or SRT, Goldschmidt algorithm can be performed in parallel. In the initialization step, D_0 and N_0 can be calculated in parallel after the table look-up of P_0. In the iteration step, as soon as P_{i+1} is calculated, N_{i+1} and D_{i+1} can be computed in parallel. Refer to the data dependency diagram shown in Figure 5-2.

During each iteration, there is one two's complementation, and two multiplications. The multiplication can be done in parallel. Notice that two's complementation is performed as part of a cycle that performs multiplication. In next chapter, a new approach will be presented which reduces the critical path by eliminating the two's complementation on the critical path.

Pipelined Multiplier Implementation

Flynn et. al. presented an implementation using a two-stage pipelined multiplier for Goldschmidt algorithm as follows [8]:

Cycles:	0	1	2	3	4	5	6	7	8	9
X:	X_0		X_1		X_2					
D:		D_0	D_0	D_1	D_1	D_2	D_2			
N:			N_0	N_0	N_1	N_1	N_2	N_2	N_3	N_3

Table 5.1: Implementing Goldschmidt Algorithm

This timing diagram assumes that the multiplier has a throughput of one multiplication per cycle and latency of two cycles.

Minimum Cycle Delay

One important point which should be examined closely is how does the algorithm calculate the two's complementation. During the iteration,

$$X_{i+1} = \overline{D_i}.$$

In the timing diagram, however, D_i and X_{i+1} appears in the same cycle, e.g., cycles 2 and 4. Therefore, in cycles 2 and 4, the time required to perform the computation is at least the time required to propagate through the second stage of the multiplier and the time required to propagate through the two's complementer, which include an adder that takes both area and time.

For precisely this reason, an approach that commonly taken is to approximate the two's complementation using one's complementation which eliminates the need to propagate through the adder. However, this approach would introduce error in every iteration of the algorithm and could require larger initial look-up tables or more iterations to obtain the necessary precision.

In the next chapter, an alternative to Goldschmidt algorithm is presented which has the same performance but removes two's complementation from the critical path, thus reduces the minimum cycle delay.

Chapter 6

A Fast Parallel Division Algorithm

In this section, a new parallel algorithm will be presented. It is similar to Goldschmidt algorithm. The comparison between the two will be presented later this chapter. The key difference between this algorithm and Goldschmidt algorithm is that this algorithm does not need to compute two's complementation. Instead, it performs an addition by one at the most significant bit. We will first describe the algorithm. For convenience, let us call this algorithm Algorithm A.

6.1 Algorithm

Algorithm A is based on the following observation:

$$\begin{aligned}
\frac{1}{1-y} &= \frac{1+y}{1-y^2} && (6.1)\\
&= \frac{(1+y)(1+y^2)}{1-y^4} && (6.2)\\
&= \frac{(1+y)(1+y^2)(1+y^4)}{1-y^8} && (6.3)\\
&= \frac{(1+y)(1+y^2)(1+y^4)(1+y^8)}{1-y^{16}} && (6.4)\\
&= \cdots\cdots && (6.5)
\end{aligned}$$

Notice that the quantity $1-y^{16}$ is very small for a small y. Therefore the following is proposed — in performing a/b, shift b to fall in the range $[0.5, 1)$. and select $1-y=b$, Algorithm converges quadratically since $y < 0.5$. We will first describe the algorithm, and in Section 6.4, the algorithm will be compared to Goldschmidt to demonstrate its advantages.

The beginning of the algorithm is the same as Goldschmidt algorithm, i.e., a table look-up.

$$L \approx \frac{1}{D}$$

$$
\begin{aligned}
&= \frac{1}{D} + E_0 \\
&= \frac{1}{D}(1 + X_0)
\end{aligned}
$$

Again, term E_0 represents the error/difference between the actual value and the value obtained from the look-up table. Term X_0 is the y term in Equation 6.1, which is the actual error term that the algorithm tries to correct.

Next step would be to obtain the approximate quotient by multiplying N with K_0:

$$
\begin{aligned}
N_0 &= N * L && (6.6) \\
&= N * \frac{1}{D}(1 + X_0) && (6.7) \\
&= Q * (1 + X_0) && (6.8)
\end{aligned}
$$

In order to decrease the error term X, we need to compute $1 - X_0$. Therefore, we first compute $-X_0$.

$$
\begin{aligned}
1 - D * K_0 &= 1 - (1 + X_0) \\
&= -X_0
\end{aligned}
$$

We then add one to $-X_0$ to obtain the desired term,

$$
\begin{aligned}
N_1 &= N_0 * (1 - X_0) \\
&= Q * (1 + X_0)(1 - X_0) \\
&= Q * (1 - X_0^2) \\
&= Q * (1 - X_1)
\end{aligned}
$$

where $X_1 = X_0^2$. The error term, X_i, is decreasing quadratically.

During the next iteration, we need the term $1 + X_1$. Since we already know the value of X_0, $1+X_1$ can be easily obtained by squaring X_0 and add one to it. Therefore,

$$
\begin{aligned}
N_2 &= N_1 * (1 + X_0^2) \\
&= Q * (1 - X_1)(1 + X_1) \\
&= Q * (1 - X_1^2) \\
&= Q * (1 - X_2)
\end{aligned}
$$

Again, the error term converges quadratically. Thus, $N_\infty = Q$. For a single-precision division, 5 iterations are necessary without the use of look-up table, and 2 iterations with a reasonably sized look-up table. For double precision, one more iteration is needed.

Therefore, the entire algorithm is:

1. *Initialization Step:*

$$L \approx \frac{1}{D} \tag{6.9}$$
$$N_0 = N * L \tag{6.10}$$
$$X_0 = 1 - D * L \tag{6.11}$$

2. *Iterative Step:*

$$P_i = 1 + X_i \tag{6.12}$$
$$N_{i+1} = N_i * P_i \tag{6.13}$$
$$X_{i+1} = X_i^2 \tag{6.14}$$

3. *Final:*

$$N_\infty = Q \tag{6.15}$$

6.2 Parallelism

Similar to Goldschmidt algorithm, algorithm A also has a two-way parallelism. Refer to the data dependency diagram shown in Figure 6-1. During each iteration, one addition and two multiplications are required. There are two key differences between algorithm A and Goldschmidt algorithm which give algorithm A an edge over Goldschmidt algorithm. We will compare the two algorithms in Section 6.4.

6.3 Implementation of Division

We will use two Raw tiles to implement algorithm A. Tile 0 will be computing N_i and tile 1 will be computing X_i in Equations 6.13 and 6.14. For convenience, we will describe the task of tile 1 first.

Instructions assigned to Tile 1

1. Loading constants and wait for divisor. Divisor is stored in $5.

```
lui $7,126
lui $8,32768
add $5,$csti,$0
```

2. Check for division by zero.

```
sll $10,$5,1
beq $10,$0,main.DIVBYZERO
```

3. Computing offset in the look-up table.

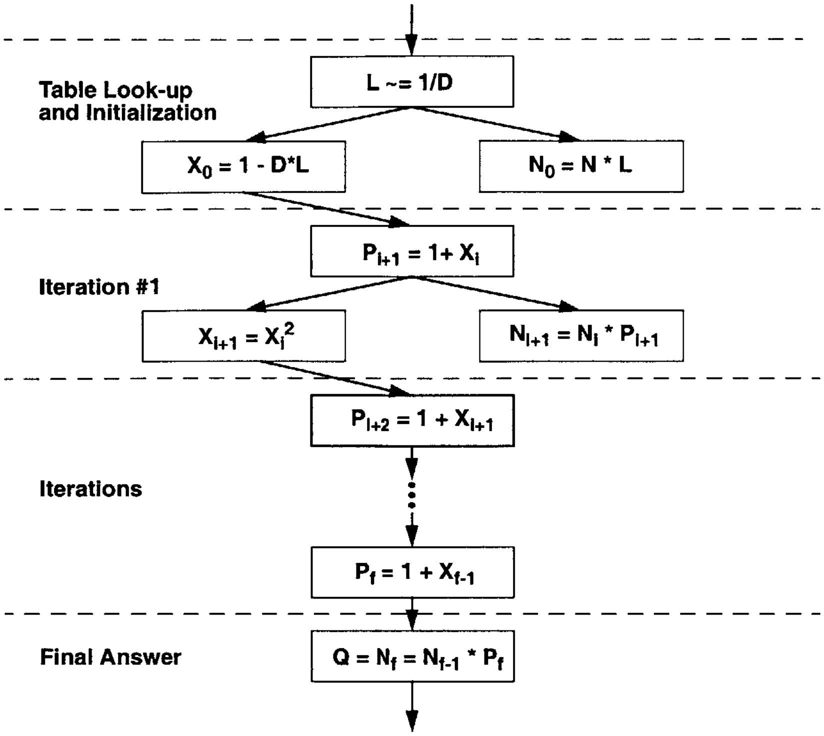

Figure 6-1: Data Dependency Diagram in New Division Algorithm

```
sll $15,$5,8
and $14,$15,$7
```

4. Unpack mantissa and do initial table look-up to obtain L in Equation 6.9.

```
lui $at,(divtable>>16)
ori $at,$at,(divtable\&0xffff)
addu $at,$at,$14
lw $4,0($at)
```

5. Send over the value of L to tile 0 and finish unpacking mantissa.

```
add $csto,$4,$0
or $15,$15,$8
```

After table look-up, we compute the initial values of X.

6. Calculation of X_0.

```
multu $15,$4
mfhi $15
srl $15,$15,1
sub $15,$at,$15
```

7. Send over the value of X_0 to tile 0. Notice the term that tile 1 needs is $1 + X_0$. This addition is hidden using the instruction to communicate between the tiles.

   ```
   add $csto,$15,$at
   ```

   By now we have finished the initialization step and ready to enter the first iteration.

8. Iteration #1, calculates $X_1 = X_0^2$.

   ```
   multu $15,$15
   mfhi $15
   ```

9. Send over X_1 to tile 0.

   ```
   sll $csto,$15,3
   ```

10. Iteration #2, calculates $X_2 = X_1^2$.

    ```
    multu $15,$15
    mfhi $15
    ```

11. Send over X_2 to tile 0.

    ```
    sll $csto,$15,7
    ```

    For single precision floating-point division, only two iterations are necessary after the initialization step. Therefore, tile 1 is done.

Now let's describe the task of tile 0, given that error term X is calculated elsewhere, tile 1, and fed into tile 0 properly. We assume that dividend N is stored in $4, and divisor D is stored in $5.

Instructions assigned to Tile 0

1. First, we route the divisor to tile 1

   ```
   add $csto,$5,$0
   ```

2. Check for division by zero

   ```
   sll $10,$5,1
   beq $10,$0,main.DIVBYZERO
   ```

3. Unpacking
   - unpack mantissa

     ```
     sll $15,$4,8
     or $15,$15,$at
     ```

   - unpack sign and exponent of N sign of N is placed in `$14` and exponent of N is placed in `$13`.

     ```
     srl $14,$4,31
     sll $13,$4,1
     srl $13,$13,24
     ```

   - unpack D. `$11` contains sign of D, `$10` contains exponent of D.

     ```
     srl $11,$5,31
     sll $10,$5,1
     srl $10,$10,24
     ```

4. We will first compute the resulting exponent and sign since tile 1 will not be able to finish the table look-up and route the result back to tile 0 yet. Resulting exponent will be stored in `$10` and resulting sign will be stored in `$11`. It also checks for underflow before computing sign.

   ```
   addiu $13,$13,127
   sub $10,$13,$10
   srl $12,$10,31
   bne $12,$0,main.UNDERFLOW
   xor $11,$14,$11
   sll $24,$11,31
   ```

5. We now enter the initialization step, calculating $N_0 = N * X_0$. Term X_0 is routed from tile 0.

   ```
   add $9,$csti,$0
   multu $15,$9
   mfhi $15
   ```

6. We are now ready for iterative step. Iteration #1 calculates $N_1 = N_0 * X_1$

   ```
   sll $9,$csti,1
   multu $15,$9
   mfhi $15
   ```

7. Iteration #2, calculates $N_2 = N_1 * X_2$.

```
add $9,$csti,$at
multu $15,$9
mfhi $9
```

Two iterations are sufficient for the accuracy required by single-precision calculations. Therefore, we enter the final steps to pack the result.

8. Normalization

```
addiu $9,$9,4
or $rlo,$9,$0
addiu $14,$rli,1
sllv $9,$9,$14
srl $2,$9,9
```

9. Check for overflow

```
subu $10,$10,$14
addiu $10,$10,3
slti $11,$10,0x000000FF
slt $12,$0,$10
and $11,$11,$12
beq $11,$0,main.OVERFLOW
```

10. Final packing

```
sll $10,$10,23
or $2,$2,$10
or $2,$2,\$24
```

The entire algorithm takes 50 cycles to execute. If packing and exception checks can be done in hardware, the latency is reduced to 36 cycles.

6.4 Comparison with Goldschmidt

From the surface, algorithm A is simply a different implementation of Goldschmidt algorithm. However, the advantages of algorithm A over Goldschmidt algorithm will be presented here. There are two key differences between the two algorithms.

1. The subtraction on Goldschmidt algorithm's critical path is replaced by an addition on algorithm F's critical path.

    - For Goldschmidt algorithm:

$$P_{i+1} = 2 - D_i \tag{6.16}$$

- For algorithm A:

$$P_{i+1} = 1 + X_i \tag{6.17}$$

2. During each iteration of Goldschmidt algorithm, two multiplications are performed,

$$\begin{aligned} N_{i+1} &= N_i * P_{i+1} \\ D_{i+1} &= D_i * P_{i+1} \end{aligned}$$

where $N_\infty = Quotient$ and $D_\infty = 1$.
However, during each iteration of algorithm A,

$$\begin{aligned} N_{i+1} &= N_i * P_i \\ X_{i+1} &= X_i^2 \end{aligned}$$

where $N_\infty = Quotient$ but unlike Goldschmidt, $X_\infty = 0$.

6.4.1 Area Comparison

The first key difference replaces the subtraction in the critical path of Goldschmidt by a addition. From the software point of view, the two variations would have the same latency. However, in actual hardware implementation, algorithm A wins.

As shown in the example on page 38, the subtraction $2 - X$ in Goldschmidt algorithm is cleverly done using a two's complement given $X < 2$. Performing a two's complement requires a inverter and a ripple-carry adder that are either 24-bit or 53-bit according the precision requirement. Refer to Figure 6-2.

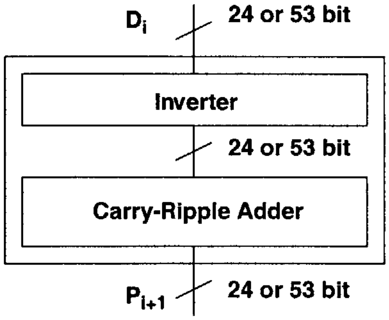

Figure 6-2: Hardware for Two's Complementation

In algorithm A, $2-X$ is replaced by $1+X$. This addition is different from the one in the two's complementation. In Goldschmidt, the one is added at the least significant bit. Therefore, a full-width adder is necessary. However, in algorithm A, the addition adds one, which is the most significant bit in the representation. Therefore, a trivial solution is available to avoid using the integer adder for it. We know that X will always be less than 1 since X is the error term and X keeps decreasing quadratically. Therefore, to perform the addition, simply hard wire the most significant bit to be 1. Refer to Figure 6-3. Thus algorithm A completely eliminates the area necessary to implement the hardware used to perform two's complementation. At the same time, it also greatly shortens the critical path. In particular, a 53-bit ripple-carry adder uses a considerable number of gates and takes a long time to finish computation.

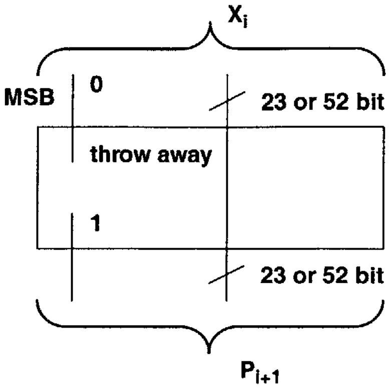

Figure 6-3: Hardware for Addition by One

Parameter Values	Delay (ns) by Power Level				Area (cells) by Power Level			
	C	E	H	J	C	E	H	J
WIDTH64	3.18	2.05	1.66	1.71	1350	1413	1686	1799
WIDTH32	2.93	1.78	1.48	1.50	606	636	768	853

Table 6.1: Two's Complement Delay and Area

Table 6.1 shows the delay and area of their two's complementer. We see that the slowest delay among the 64-bit two's complementation takes 3.18 nanoseconds, which is close one cycle on a 300 MHz machine and more than one cycle on a 333 MHz machine. Even the fastest one takes 1.71 nanoseconds which takes more than half a cycle on a 333 MHz machine. Similarly, it takes a considerable amount of area too. The area it takes is roughly the same as a leading zero counter. Algorithm A completely eliminates the need to use any of these hardware and reduces the latency directly off the critical path. Therefore, the hardware unit can be clocked at a much faster rate.

6.4.2 Latency Comparison

Another advantage of algorithm A over Goldschmidt algorithm is that one of the two multiplications during each iteration doesn't have to be a full-width multiplication. As pointed out earlier, one multiplication is to square the error term X_i. Since the error term is approaching 0, after each iteration, the number of leading zero bits are doubled.

Example: Assume that there is no initial look-up table, all input start on the same initial guess. Since the range of the mantissa is $[1, 2)$, the range of the reciprocal is $(0.5, 1]$. Therefore, the hardwired initial guess chosen is 0.75.

Assume initially that:

$$X_0 < 0.01000\cdots000_2$$

We have:

$$\begin{array}{c}
\textit{Iteration \#0: } \text{2 leading zeros} \\
\downarrow \\
\textit{Iteration \#1: } \text{4 leading zeros} \\
\downarrow \\
\textit{Iteration \#2: } \text{8 leading zeros} \\
\downarrow \\
\textit{Iteration \#3: } \text{16 leading zeros} \\
\downarrow \\
\textit{Iteration \#4: } \text{32 leading zeros} \\
\downarrow \\
\textit{Iteration \#5: } \text{64 leading zeros}
\end{array}$$

Table 6.2 shows the multiplication widths necessary during each iteration.

Iteration #	Multiplication Width		
	Single	Double	Quadruple
1	22	51	111
2	20	49	109
3	16	45	105
4	8	37	97
5	—	21	81
6	—	—	49

Table 6.2: Multiplication Widths for Algorithm A

As we can see, towards the end of the iterations, the number of non-zero significant digits decrease dramatically. Therefore, not all multiplications are required to be full-width multiplications. The latency required to perform multiplication with less bit-width are generally smaller than wider bit-width ones.

Table 6.3 shows the delay and area of a signed multiplier. Notice that during iterations 3 and 4 during a single-precision division, only 16-bit and 8-bit multiplications are necessary. According to Table 6.3, a 32-bit multiplier uses 22967 cells. However, during iteration 3, a 16-bit multiplier is necessary, which only uses 6432 cells. Furthermore, a 8-bit multiplier, which is sufficient for iteration 4, only uses 1875 cells. Algorithm A introduces significant savings for the hardware area. And by the same token, it also reduces latency.

This advantage could also have impact on high precision operations. Let's look at the following example.

Parameter Values	Delay (ns) by Power Level				Area (cells) by Power Level			
	C	E	H	J	C	E	H	J
N4_M4	5.24	4.11	3.24	3.07	554	560	633	717
N8_M8	7.00	5.20	4.25	4.01	1875	1902	2190	2470
N16_M16	9.47	6.32	5.39	4.67	6432	6491	7535	8477
N32_M32	11.72	7.87	6.71	6.19	22967	23217	26878	29919

Table 6.3: Signed Multiplier Delay and Area

Example: If quadruple precision is required and architecture only supports 64-bit multiplication.

To perform the mantissa multiplication, a 128-bit multiplier is required. However, when we only have 64-bit multipliers, we need to perform four 64-bit multiplications and three 64-bit additions. In Goldschmidt algorithm, every multiplication has to be done as described. However, in algorithm A, the number of significant bits during the last iteration is less than half of the width of the initial multiplication. Therefore, in the context of this example, only one 64-bit multiplication is necessary, considerably reduces latency.

6.4.3 Relative Error in Division Algorithms

There are three sources of error that affect division algorithms in general. First source of error is caused by the termination of the algorithm since we can not keep multiplying correction terms indefinitely. Second one comes from the truncation of the the significant bits after multiplications, i.e., we can only retain a finite number of bits. The third source of error, however, is introduced by approximating two's complementation using one's complementation in the algorithm to reduce the cycle latency. In this section, each one is briefly discussed.

Termination

The termination error is the error caused by the inability to keep multiplying correction terms forever. For example, for single precision division, 5 iterations are used, i.e., term

$$\sum_{i=3}^{\infty} 1 + y^{2^i} \tag{6.18}$$

is not multiplied to the final result. Let us give an upper bound on the value of term expressed in Equation 6.18.

The following formulas are useful in the analysis:

$$1 + x < e^x < 1 + x + x^2 \tag{6.19}$$

when x is a small number and e is the base of natural log.

we have

$$\begin{aligned} 1 &< (1+y^8)(1+y^{16})(1+y^{32})\cdots \\ &< e^{y^8} * e^{y^{16}} * e^{y^{32}} \cdots \\ &< e^{y^8+y^{16}+y^{32}+\cdots} \\ &< e^{1.1*y^8} \\ &< 1 + 1.1 * y^8 + (1.1 * y^8)^2 \\ &< 1 + 1.2 * y^8 \end{aligned}$$

The value is very close to 1. The error caused by termination is very small.

Truncation Error

After each multiplication, only a finite number of bits are retained for the next iteration. The truncated bits, even though represent a very small value, cause error. This is a common problem to both Goldschmidt algorithm and algorithm A. The error it introduces is the same for both algorithms. Therefore, extensive analysis will not be done here.

Approximating Two's Complement Using One's Complement

Since Goldschmidt algorithm uses two's complementers which requires considerable amount of area and time. In many practical applications, the two's complementation is approximated by one's complementation so that the hardware can be clocked faster. The difference is that the addition by one at the least significant bit is ignored. However, there is significant disadvantage with this approach. This approximation takes place during every iteration and the error it introduces keeps propagating into all future iterations.

Algorithm A eliminates most of the error from this source by removing the two's complementation out of the computation in all iterations except the first one.

In the following, a more detailed analysis of how much error is introduced by approximating two's complementation by one's complementation is presented. This analysis is done for algorithm A, i.e., only the first iteration creates this error. For Goldschmidt algorithm, this analysis has to be done for every iteration.

Assuming that we are performing N/D. We use the following notation:

Based on the representation, the following hold:

$$\begin{aligned} B &= N(1+y)(1+y^2)(1+y^4) \\ A &= N(1+f)(1+f^2)(1+f^4) \\ Q &= N(1+y)(1+y^2)(1+y^4)/(1-y^8) \end{aligned}$$

symbol	meaning
y	$y = 1 - D$, the error term
f	y modified using one's complement
e	the value of last bit
A	final result of the division N/D using one's complement
B	final result of the division N/D
Q	the theoretical value of N/D

We notice,

$$\begin{aligned}\frac{B}{Q} &= 1 - y^8 \\ &< 1\end{aligned}$$

thus,

$$B < Q.$$

Next, we can claim that $y = f + e$, thus we have

$$\begin{aligned} y &< f \\ 1 + y &< 1 + f \\ 1 + y^2 &< 1 + f^2 \\ &\cdots \end{aligned}$$

Therefore,

$$B > A$$

Next, let us examine the difference between A and Q.

$$\begin{aligned}\frac{A}{Q} &= \frac{(1+f)(1+f^2)(1+f^4)(1-y^8)}{(1+y)(1+y^2)(1+y^4)} \\ &= (1+f)(1+f^2)(1+f^4)(1-y) \\ &= (1+f)(1+f^2)(1+f^4)(1-(f+e)) \\ &= (1-f^8) - e(1+f)(1+f^2)(1+f^4) \\ &= \frac{B}{Q} + (y^8 - f^8) - e(1+f)(1+f^2)(1+f^4)\end{aligned}$$

Since $y = e + f$, we have $e = y - f$:

$$\frac{A}{Q} = \frac{B}{Q} + e(y+f)(y^2+f^2)(y^4+f^4) - e\left(\frac{1-f^8}{1-f}\right)$$

$$
\begin{aligned}
&> \frac{B}{Q} - e\left(\frac{1-f^8}{1-f}\right) \\
&> \frac{B}{Q} - \frac{e}{1-f}
\end{aligned}
$$

We also know that,

$$
\begin{aligned}
1 &> \frac{B}{Q} \\
&> \frac{A}{Q} \\
&> \frac{B}{Q} - \frac{e}{1-f} \\
&= 1 - y^8 - \frac{e}{1-f}
\end{aligned}
$$

Since $e << f < d << 1$, the additional relative error caused by using one's complements is bounded by $e/(1-f)$. It is on the order of e, the least significant bit, which is tolerable.

Chapter 7

Floating-Point Square-Root Operation

In this section, we will first introduce the generic square-root algorithm. Then, a parallel one which parallels Goldschmidt for division will be presented as well as the implementation on RAW and it's performance.

7.1 Generic Newton-Raphson Algorithm

Equation 5.1 gives the iterative formulation used in Newton-Raphson algorithm. A straight forward implementation of the algorithm would use

$$f(x) = x^2 - B \tag{7.1}$$

where B is the input to the square-root operation. According to Equation 5.1, the iterative formula for the naive Newton-Raphson would be

$$x_{i+1} = \frac{x_i + \frac{B}{x_i}}{2} \tag{7.2}$$

Notice that there is a division involved in every iteration which can take many cycles to complete. Therefore, a simple modification to the naive approach is to compute $1/\sqrt{B}$ and then multiply by B. In this case, the equation used for Newton-Raphson is

$$f(x) = \frac{1}{x^2} - B \tag{7.3}$$

Plugging into Equation 5.1, we obtain

$$x_{i+1} = \frac{x_i}{2}(3 - B * x_i^2) \tag{7.4}$$

Equation 7.4 is the iterative formula for the modified Newton-Raphson algorithm. Notice that the division previously existed in each iteration is removed by better formulation of the algorithm.

Algorithm N

Therefore the algorithm is described as following, for $\sqrt{B}$

1. *initialization step:* Initial table look-up:

$$K_0 \approx 1/\sqrt{B}$$

2. *iteration step:*

$$K_{i+1} = \frac{K_i * (3 - B * K_i^2)}{2} \tag{7.5}$$

3. *final step:*

$$\sqrt{B} = K_\infty \times N$$

During each iteration, there are three multiplications and one subtraction. This algorithm is completely sequential.

7.2 Parallel Square-Root Algorithm

Flynn et. al. detailed a parallel algorithm in [8], algorithm G, which we will present in this section.

7.2.1 Algorithm G

On calculating $\sqrt{N}$:

1. *initialize:*

$$\begin{aligned} r_0 &\approx \frac{1}{\sqrt{N}} && (7.6)\\ B_0 &= N && (7.7)\\ X_0 &= N && (7.8) \end{aligned}$$

2. *iterate:*

$$\begin{aligned} SQr_i &= r_i * r_i && (7.9)\\ B_{i+1} &= B_i * r_i && (7.10)\\ X_{i+1} &= X_i * SQr_i && (7.11)\\ r_{i+1} &= 1 + 0.5 * \overline{X_{i+1}^f} && (7.12) \end{aligned}$$

3. *final:*

$$\sqrt{N} = B_\infty \tag{7.13}$$

We briefly show how this algorithm works. The term X_i is the convergence factor and $X_\infty = 1.0$. Initially, table look-up gives

$$\begin{aligned} r_0 &\approx \frac{1}{\sqrt{N}} \\ &= \frac{1}{\sqrt{N}} + \epsilon \end{aligned}$$

leads to

$$\begin{aligned} SQr_0 &= \frac{1}{N} + \frac{2\epsilon}{\sqrt{N}} + \epsilon^2 \\ &\approx \frac{1}{N} + \frac{2\epsilon}{\sqrt{N}} \\ X_1 &= 1 + 2\epsilon\sqrt{N} \end{aligned}$$

The term $2\epsilon\sqrt{N}$ is the error term and it converges quadratically. Assume that $X_i = 1 - \delta$ where δ is the error term.

$$\begin{aligned} r_i &= 1 - 0.5 * \overline{X_i^f} \\ &= 1 + 0.5 * \delta \end{aligned}$$

Therefore,

$$\begin{aligned} SQr_i &= 1 + \delta + \frac{\delta^2}{4} \\ &\approx 1 + \delta \end{aligned}$$

leads to

$$X_{i+1} = 1 - \delta^2.$$

We notice that

$$\begin{aligned} X_\infty &= X_0 * SQr_0 * SQr_1 * \cdots \\ &= N * \Pi_{i=0}^{\infty} SQr_i \\ &= 1 \end{aligned}$$

Therefore,

$$\begin{aligned} \Pi_{i=0}^{\infty} SQr_i &= (\Pi_{i=0}^{\infty} r_i)^2 \\ &= \frac{1}{N} \end{aligned}$$

which means $\Pi_{i=0}^{\infty} r_i = 1/\sqrt{N}$. Finally

$$\begin{aligned} B_\infty &= B_0 * \Pi_{i=0}^{\infty} r_i \\ &= N * \Pi_{i=0}^{\infty} r_i \\ &= \sqrt{N} \end{aligned}$$

7.2.2 Parallelism

This algorithm has some parallelism that we can explore. Figure 7-1 shows the data-dependency of this algorithm. The critical path is the path along the left. On this critical path, there are two multiplications, and two subtraction.

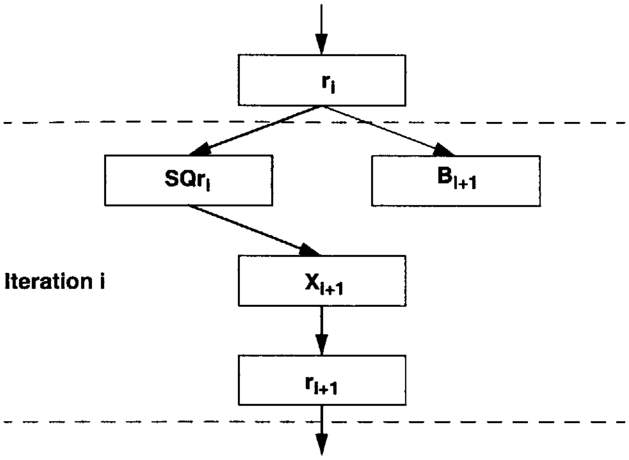

Figure 7-1: Data Dependency Diagram for Algorithm G

Using a two-stage pipelined multiplier, this algorithm can be implemented as shown in Table 7.1 [8]:

Cycles:	0	1	2	3	4	5	6	7	8	9	10
r:	r_0				r_1				r_2		
SQr:		SQr_1	SQr_1			SQr_2	SQr_2				
B:			B_1	B_1			B_2	B_2		B_3	B_3
X:				X_1	X_1			X_2	X_2		

Table 7.1: Implementing Algorithm G

In the next chapter, a different square root algorithm will be presented. It parallels algorithm A as algorithm G parallels Goldschmidt algorithm.

Chapter 8

A Fast Square Root Algorithm

In this section, a new parallel square-root algorithm will be presented. It is similar to algorithm A and has a few advantages over algorithm G in a similar fashion. For convenience, we call this the following algorithm algorithm B. Algorithm B parallels algorithm A as algorithm G parallels Goldschmidt algorithm.

8.1 Algorithm

In this section we will first introduce the basic algorithm. Section 8.1.2 shows the correctness of the basic algorithm. In Section 8.2, further improvement will be added onto the basic algorithm.

8.1.1 Basic Algorithm

The basic steps of algorithm B are as follows:

1. *Initialize:*

$$\begin{aligned} P &\approx \frac{1}{\sqrt{N}} && (8.1)\\ B_0 &= N * P && (8.2)\\ e_0 &= B_0 * P - 1 && (8.3)\\ & && (8.4) \end{aligned}$$

2. *Iterate:*

$$\begin{aligned} SQe_i &= e_i * e_i && (8.5)\\ Q_i &= \frac{e_i - 3}{4} && (8.6)\\ R_{i+1} &= 1 - \frac{e_i}{2} && (8.7) \end{aligned}$$

$$
\begin{aligned}
B_{i+1} &= B_i * R_{i+1} && (8.8)\\
e_{i+1} &= SQe_i * Q_i && (8.9)
\end{aligned}
$$

3. *Final:*

$$\sqrt{N} = B_\infty \tag{8.10}$$

This algorithm converges in a quadratic fashion since the error term. e_i, decreases as:

$$e_{i+1} = \frac{e_i^3 - 3 * e_i^2}{4} \tag{8.11}$$

Each iteration contains three multiplications, one two's complementation, two shifts and two additions. From the surface, it seems that algorithm B has a bit more work compared to algorithm G. But in Section 8.3, we will show how can these operations be performed in parallel and critical path can be reduced compared to algorithm G. Before that, the correctness of the algorithm will be presented in Section 8.1.2.

8.1.2 Correctness

Let's take a look at how this algorithm works. To best understand it, let's introduce one more variable which does not appear in the algorithm. Let $X_i = 1 + e_i$. We make the following claims:

1. Term $X_\infty = 1$ with quadratic convergence.
2. $X_{i+1} = X_i * R_i^2$.
3. $B_\infty = \sqrt{N} * \Pi_{i=0}^{\infty} R_i$

Claim 1

The first claim is trivial to verify since $X_\infty = 1 + e_\infty$ and $e_\infty = 0$.

Claim 2

Now let us verify claim 2. Assume first that,

$$
\begin{aligned}
X_i &= 1 + e_i\\
R_i &= 1 - \frac{e_i}{2}.
\end{aligned}
$$

Following the algorithm, we obtain that

$$
\begin{aligned}
X_{i+1} &= 1 + e_{i+1}\\
&= 1 + \frac{e_i^3 - 3 * e_i^2}{4}
\end{aligned}
$$

Squaring R_i gives:

$$R_i^2 = 1 - e_i + \frac{e_i^2}{4}$$

Lastly,

$$\begin{aligned}
X_{i+1} &= X_i * R_{i+1}^2 \\
&= (1 + e_i)(1 - e_i + \frac{e_i^2}{4}) \\
&= 1 + \frac{e_i^3 - 3 * e_i^2}{4} \\
&= 1 + e_{i+1}
\end{aligned}$$

Base Case

Initially, table look-up gives

$$\begin{aligned}
P &\approx \frac{1}{\sqrt{N}} \\
&= \frac{1}{\sqrt{N}} + \delta \\
B_0 &= N * P \\
&= \sqrt{N}(1 + \delta\sqrt{N}) \\
&= \sqrt{N} * R_0
\end{aligned}$$

leads to

$$\begin{aligned}
X_0 &= 1 + e_0 \\
&= 1 + 2\delta\sqrt{N} + (\delta\sqrt{N})^2 \\
&= R_0^2
\end{aligned}$$

Result

$$\begin{aligned}
B_\infty &= \sqrt{N} * \Pi_{i=0}^{\infty} R_i \\
&= \sqrt{N} * \sqrt{B_\infty} \\
&= \sqrt{N}
\end{aligned}$$

8.2 Improvement on the Basic Algorithm

A simple change in the basic algorithm could eliminate one two's complementation in each iteration except the first one. Recall that during each iteration, the calculation required are:

$$
\begin{aligned}
SQe_i &= e_i * e_i \\
Q_i &= \frac{e_i - 3}{4} \\
R_{i+1} &= 1 - \frac{e_i}{2} \\
B_{i+1} &= B_i * R_{i+1} \\
e_{i+1} &= SQe_i * Q_i
\end{aligned}
$$

Notice that at the start of every iteration, term e_i is always negative except maybe the first iteration due to the table look-up. This is because SQe_{i-1} is always positive and Q_{i-1} is always negative. However, if we modify calculation for R_i to the following,

$$Q_i = 3 + e_i 4 \tag{8.12}$$

then e_i will always be positive. Therefore, calculation for R_i can be changed to

$$R_{i+1} = 1 + \frac{e_i}{2} \tag{8.13}$$

thus eliminating the two's complementation. However, this change can be made only if term e is positive at the beginning of iteration i. Therefore, during the first iteration, we need to use the basic algorithm. To briefly justify the modification, assume that

$$
\begin{aligned}
X_i &= 1 - e_i \\
R_i &= 1 + \frac{e_i}{2}.
\end{aligned}
$$

With the modification,

$$
\begin{aligned}
X_{i+1} &= 1 - e_{i+1} \\
&= 1 - \frac{e_i^3 - 3 * e_i^2}{4}
\end{aligned}
$$

Squaring R_i gives:

$$R_i^2 = 1 + e_i + \frac{e_i^2}{4}$$

Leads to,

$$
\begin{aligned}
X_{i+1} &= X_i * R_{i+1}^2 \\
&= (1 - e_i)(1 + e_i + \frac{e_i^2}{4}) \\
&= 1 - \frac{e_i^3 + 3 * e_i^2}{4} \\
&= 1 - e_{i+1}
\end{aligned}
$$

The benefit of this modification will be shown in Section 8.4.

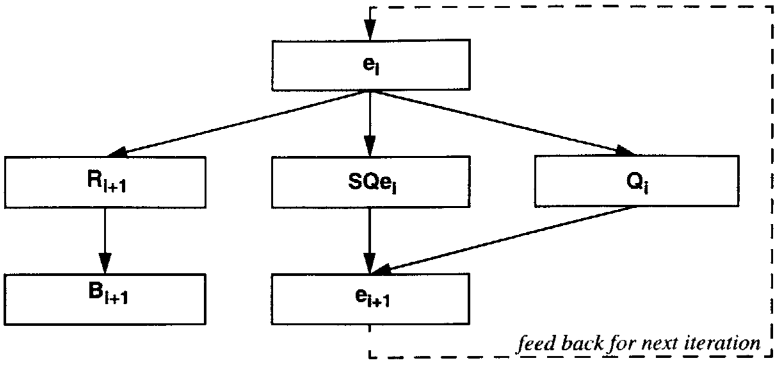

Figure 8-1: Data Dependency Diagram for Algorithm B

8.3 Parallelism

There exists parallelism in algorithm B that we can explore to speed up the calculation. As usual, let us look at the data dependency diagram, shown in Figure 8-1. We see that R's, SQe's, and Q's can all be calculated in parallel. The only true dependence is that e_{i+1} depends directly on SQe_i.

8.4 Comparison with Algorithm G

We first give the timing diagram for the algorithm, assume that we use a two-stage pipelined multiplier. The timing diagram is shown in Table 8.1. It shows that the number of cycles required for square-root is the same as algorithm G. A total of 10 cycles are required and six multiplications are performed. If the algorithm is performed serially, then it uses one less multiplication then algorithm G.

Cycles:	0	1	2	3	4	5	6	7	8	9	10
P:	P										
B:		B_0	B_0				B_1	B_1		B_2	B_2
e:				e_0	e_0			e_1	e_1		
SQe:						SQe_0	SQe_1				
R:						R_1				R_2	
Q:							Q_1				

Table 8.1: Implementing Algorithm B

There are two major advantages algorithm B possess over algorithm G. Each will be presented in the following sections.

8.4.1 Reduced Critical Path

The first key difference here is that the critical path of algorithm B is shorter compared to algorithm G. To recap, during each iteration algorithm G performs:

$$\begin{aligned} SQr_i &= r_i * r_i \\ B_{i+1} &= B_i * r_i \\ X_{i+1} &= X_i * SQr_i \\ r_{i+1} &= 1 + 0.5 * \overline{X^f_{i+1}} \end{aligned}$$

out of the four variables being updated, B is not on the critical path, i.e., the critical path contains two multiplications and a two's complementation.

$$\begin{aligned} SQr_i &= r_i * r_i \\ X_{i+1} &= X_i * SQr_i \\ r_{i+1} &= 1 + 0.5 * \overline{X^f_{i+1}} \end{aligned}$$

During each iteration of algorithm B, the algorithm performs:

$$\begin{aligned} SQe_i &= e_i * e_i \\ Q_i &= \frac{e_i - 3}{4} \\ R_{i+1} &= 1 - \frac{e_i}{2} \\ B_{i+1} &= B_i * R_{i+1} \\ e_{i+1} &= SQe_i * Q_i \end{aligned}$$

However, out of the five variables being updated. The critical path only contains

$$\begin{aligned} SQe_i &= e_i * e_i \\ e_{i+1} &= SQe_i * Q_i \end{aligned}$$

which contains only two multiplications. Compared to algorithm G, the two's complementation on the critical path of algorithm G is no longer on the critical path of algorithm B.

Notice that when calculating the last value of B, the value of R is required. In the basic algorithm presented in Section 8.1, computing R requires a two's complementation, which is in the critical path of algorithm B. However, with the modification made in Section 8.2, this two's complementation is eliminated for all iterations except the first one. Therefore the computation of R required by the last multiplication does not need to propagate through the adder.

Refer to Table 6.1 for a rough idea of the delay through a two's complementation. Therefore Floating-point square root unit implemented using algorithm B can be clocked at a much faster rate.

8.4.2 Reduced Multiplication Width

Similar to algorithm A, One of the three multiplications performed by algorithm B does not require a full multiplication, which is the step to square the error term

$$SQe_i = e_i * e_i.$$

Since error is approaching zero, this multiplication does not need all of the accuracy of a full-width multiplier. Section 6.4.2 already discussed this advantage.

8.5 Implementation

In this section, we show how algorithm B is implemented in assembly code. Again, we employ two Raw tiles since the critical path is longer than all the rest of calculation combined.

Instructions assigned to Tile 1

1. First check for negative operands by checking sign.

```
srl $8,$4,31
bne $8,$0,main.NAN
```

2. Unpack the rest of field

```
sll $9,$4,1
srl $9,$9,24
sll $10,$4,8
or $10,$10,$11
```

3. Fix the exponent. That is, if the exponent is odd, then significant is rightshifted by one and exponent is incremented. Otherwise the significant is right shifted by two and exponent adjusted accordingly. This step puts the range of the significant to [0.25,1).

```
# fix exponent
sll $24,$9,31
beq $24,$11,main.EXPNOFIX
addiu $9,$9,1
addiu $9,$9,1
j main.EXPNOFIX1
add $0,$0,$0

main.EXPNOFIX:
srl $10,$10,1
```

4. Table look-up for initial guess.

```
main.EXPNOFIX1:
# load initial guess r0 into $5
# compute offset in the table
srl $25,$10,25
sll $25,$25,2
lui $at,(sqrttable>>16)
ori $at,$at,(sqrttable&0xffff)
add $at,$at,$25
lw $5,0xFF80($at)
lui $15,65535
```

5. Initialization, calculate B_0, e_0 and send B_0 to tile 1.

```
multu $5,$10
mfhi $24

# send over B_0 to tile 1
add $csto,$0,$24

multu $24,$5
srl $11,$11,3
mfhi $25
# calculate e0
sub $25,$25,$11
```

6. Send term e_0 over to tile 1. Compute SQe_0 and R_1.

```
sll $24,$25,3
add $csto,$0,$25
mult $24,$24
subu $12,$8,$25
sll $12,$12,2
mfhi $15
```

7. Calculate e_1.

```
multu $15,$12
lui $11,32768
mfhi $25
```

8. Calculate R_1.

```
or $25,$25,$11
```

9. Calculate final value of B and exponent.

```
sll $12,$csti,3
multu $25,$12
# calculate exponent
srl $9,$9,1
add $2,$9,$13
sll $2,$2,23
mfhi $5
```

10. Packing final result.

```
sll $at,$5,4
srl $at,$at,9
or $2,$at,$$2
```

Instructions assigned to Tile 0

1. Waiting for value of B_0 and e_0. Compute R_1 using the communication instruction.

```
lui $at,16384
add $24,$0,$csti
add $25,$at,$csti
```

2. Calculate B_1 and route it to Tile 0.

```
multu $24,$25
mfhi $csto
```

The algorithm runs in 55 cycles. Without packing, it runs in 47 cycles.

Chapter 9

Summary

This thesis conducts a investigation at performing floating-point operations in parallel using software. A software floating-point unit is constructed and tested on the Raw parallel architecture. The approaches taken are to explore instruction level parallelism; using compiler optimization; improve floating-point algorithms; and provide special hardware for expensive software operations.

The four basic floating-point operations are implemented. For multiplication, a two-way parallelism existed and can be easily parallelized and achieve good speed up. For addition and subtraction, however, calculation is sequential. For both division and square-root, parallel algorithms exist.

For all operations, unnecessary packing and unpacking for intermediate results can be eliminated. As shown in Chapter 4, packing and unpacking takes around 30% of the calculation and many of them are done for intermediate results.

Providing some special hardware for expensive software operations might greatly reduce the latency. For example, If a hardware leading-zero counter can be provided, it reduces around 20 cycles on the critical path of addition. Also, sticky shifter and rounding logic can also reduce the length of the critical path but require little area. On the Raw architecture, they could be implemented in the reconfigurable logic.

For both division and square-root, improved algorithms are constructed to optimize the existing algorithms. Both algorithm A and algorithm B have better critical path than their corresponding algorithms, Goldschmidt algorithm and algorithm G. Therefore, they have shorter latency when implemented in software. Reduced critical path enables much faster clocking in hardware and reduced multiplication width reduces the area.

Table 9 provides the comparison of performance between software and hardware. A few things can be concluded from the data. First of all, it is difficult for software addition and multiplication to catch up with the performance of hardware. Both floating-point multiplication and addition are fully pipelined and have latencies of 3 cycles, where as a fast integer multiplication could take 4 cycles . Moreover, addition and multiplication are heavily used and thus require high performance.

However, the data shows that, with good implementation and appropriate optimizations, division and square-root can achieve a comparable performance as hardware. If only a very small percentage of the floating-point operations are division and

	MUL	ADD	DIV	SQRT
Hardware	3	3	10	12
Software	21	42	55	55
SW (w/o Packing)	14	25	40	50
SW (w/o Excep. Chk.)	8	21	34	48

Table 9.1: Performance Comparison

square root. It might be realistic to perform them in software whenever they appear and save the hardware division and square root unit.

Bibliography

[1] Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, and Michael Taylor. The raw compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, Stanford, CA, August 1997.

[2] Rajeev Barua, Walt Lee, Anant Agarwal, and Saman Amarasinghe. Memory bank disambiguation using modulo unrolling for raw machines. In *Proceedings of the Fifth International Conference on High Performance Computing*, Chennai, India, December 1998.

[3] William J. Dally. Micro-optimization of floating-point operations. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–289, Boston, Massachusetts, April 3–6, 1989.

[4] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, M.I.T., June 1964.

[5] John Hennessy and David Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., second edition, 1996.

[6] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[7] Stuart F. Oberman. *Design Issues in High Performance Floating Point Arithmetic Units.* PhD thesis, Stanford University, November 1996.

[8] Eric M. Schwarz and Michael J. Flynn. Using a floating-point multiplier's internals for high-radix division and square root. Technical Report CSL-TR-93-554, Stanford University, January 1993.

[9] Elliot Waingold, Michael Taylor, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Srikrishna Devabhaktuni, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: The raw machine. In *IEEE Computer*, September 1997.

[10] Shlomo Waser and Michael Flynn. *Introduction to Arithmetic for Digital System Designers.* Holt, Rinchart, and Winston, 1982.

2/26 - 57