IEEE Compliant Double-Precision FPU and 64-bit ALU with Variable Latency Integer Divider

Ryan D. Williams

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
IEEE Compliant Double-Precision FPU and 64-bit ALU

with Variable Latency Integer Divider

by

Ryan D. Williams

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Dr. Kenneth W. Hsu
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
June 2007

Approved By:

__________________________
Dr. Kenneth W. Hsu
Primary Advisor – R.I.T. Dept. of Computer Engineering

__________________________
Dr. Roy Melton
Secondary Advisor – R.I.T. Dept. of Computer Engineering

__________________________
Dr. Dhireesha Kudithipudi
Secondary Advisor – R.I.T. Dept. of Computer Engineering
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: IEEE Compliant Double-Precision FPU and 64-Bit ALU with Variable Latency Integer Divider

I, Ryan D. Williams, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Ryan D. Williams

Date
Dedication

To Emileigh, without whose patience and support I never would have been able to complete this work, and to my parents for their continued support.
Acknowledgements

I would like to thank Dr. Kenneth W. Hsu for lending me his support, knowledge, and enthusiasm, and the members of my committee, Dr. Dhireesha Kudithipudi and Dr. Roy Melton, for their help and suggestions.
Abstract

Together the arithmetic logic unit (ALU) and floating-point unit (FPU) perform all of the mathematical and logic operations of computer processors. Because they are used so prominently, they fall in the critical path of the central processing unit – often becoming the bottleneck, or limiting factor for performance. As such, the design of a high-speed ALU and FPU is vital to creating a processor capable of performing up to the demanding standards of today’s computer users.

In this paper, both a 64-bit ALU and a 64-bit FPU are designed based on the reduced instruction set computer architecture. The ALU performs the four basic mathematical operations – addition, subtraction, multiplication and division – in both unsigned and two’s complement format, basic logic operations and shifting. The division algorithm is a novel approach, using a comparison multiples based SRT divider to create a variable latency integer divider. The floating-point unit performs the double-precision floating-point operations add, subtract, multiply and divide, in accordance with the IEEE 754 standard for number representation and rounding.

The ALU and FPU were implemented in VHDL, simulated in ModelSim, and constrained and synthesized using Synopsys Design Compiler (2006.06). They were synthesized using TSMC 0.13μm CMOS technology. The timing, power and area synthesis results were recorded, and, where applicable, compared to those of the corresponding DesignWare components. The ALU synthesis reported an area of 122,215 gates, a power of 384 mW, and a delay of 2.89 ns – a frequency of 346 MHz. The FPU synthesis reported an area 84,440 gates, a delay of 2.82 ns and an operating frequency of 355 MHz. It has a maximum dynamic power of 153.9 mW.
Table of Contents

Abstract v

List of Figures x

List of Tables xiii

Glossary xv

1. Introduction 1

2. ALU Overview 4

2.1 Select MIPS Instructions 4

3. Integer Adder/Subtractor 7

3.1 Parallel Prefix Adders 8

3.1.1 Traditional Equations 8

3.1.2 Ling Equations 9

3.2 PPA Tree Structures 10

3.2.1 Kogge-Stone 11

3.2.2 Brent-Kung 12

3.2.3 Han-Carlson 13

3.2.4 Comparison of Radix-2 PPAs 15

3.2.5 Radix-4 Kogge-Stone 16

3.3 Hybrid Adder 17
5.4 Chapter Summary 55

6. ALU Synthesis Results 56

6.1 Adder Synthesis 56

6.2 Multiplier Synthesis 59

6.3 Divider Synthesis 60

6.4 ALU Synthesis 61

7. Floating-Point Unit Overview 64

7.1 The IEEE 754 Standard 64

7.2 FPU Instructions 67

8. Floating-Point Adder/Subtractor 68

8.1 Common FP Adder/Subtractor Optimizations 70

8.1.1 Use of Compound Adders 70

8.1.2 Parallel Paths 70

8.1.3 One’s Complement Significand Negation 70

8.1.4 Leading Zero Approximation 71

8.1.5 Reduction of Rounding Modes 71

8.2 The SE FP Addition/Subtraction 72

8.2.1 Near Path 73

8.2.2 Far Path – First Cycle 80

8.2.3 Far Path – Second Cycle 83

8.3 Chapter Summary 92
List of Figures

3.1 64-Bit Kogge-Stone Adder 11
3.2 64-Bit Kogge-Stone Adder with Carry-in 12
3.3 64-Bit Brent-Kung Adder 13
3.4 64-Bit Han-Carlson Adder 14
3.5 64-Bit Han-Carlson Adder with Carry-in 15
3.6 64-Bit Radix-4 Han-Carlson Adder 16
3.7 64-Bit Hybrid Han-Carlson Carry-Select Adder 18
3.8 Adder/Subtractor 20
4.1 4-Bit Unsigned and Signed Multiplication Examples 22
4.2 Shift and Add Multiplier Circuit 23
4.3 3:2 and 4:3 Wallace Trees with 7 Inputs 24
4.4 4:2 Compressor Stage 25
4.5 Booth-Wallace Multiplier 28
5.1 PD Plot 38
5.2 Comparison Multiples Generator 46
5.3 Comparator 46
5.4 BSDA - Binary Signed Digit Carry-Free Adder 47
5.5  Sign Detector  
5.6  QDS Output Encoding  
5.7  QDS Logic  
5.8  Adjust Unit for Integer Divider  
5.9  SRT Division Block Diagram  
5.10  Operand Conditioning  
6.1  HC/CS Adder Schematic from Synthesis  
6.2  Universal Multiplier from Synthesis  
6.3  Universal Divider from Synthesis  
6.4  ALU from Synthesis  
6.5  Barrel Shift Left from Synthesis  
7.1  Double-Precision IEEE 754 Format  
8.1  Block Diagram of Floating-Point Adder/Subtractor  
8.2  High-Level SE FP Adder/Subtractor Block Diagram  
8.3  SE Adder Near Path  
8.4  PN Recoding Circuit  
8.5  Binary Priority Encoder  
8.6  Area Optimized Unary Priority Encoder  
8.7  BPENC Implemented Using Recursive Structure  
8.8  SE Adder Far Path – First Cycle
List of Tables

2.1 ALU Operations 5

3.1 Comparison of Radix-2 PPAs for Theoretical Area and Delay 15

4.1 Modified Booth Encoding 26

5.1 BSD Conversions 35

5.2 Redundant Digit Sets 36

5.3 QDS Function 38

5.4 Upper and Lower Bounds for r=4, a=2 BSD Set 41

5.5 BSDA Outputs 47

5.6 QDS Output Encoding 48

5.7 Compression Process for l ∈ {0,1,2} 51

6.1 Comparison of Adder Synthesis Results 57

7.1 Double Precision IEEE 754 FP Values 66

7.2 Possible Implementation of IEEE Rounding Modes 66

7.3 FPU Operations 67

8.1 Compound Adder Inputs Based on Exponents 75

8.2 Reduce Rounding Modes 83

8.3 RNRI to INJ Conversion 86
8.4 Truth Tables for CRS Generation 89

10.1 FP Compression Process for \( l \in \{0,1,2\} \) 104

10.2 Analysis of Initial q Values with Respect to x and d 107

10.3 CRN Rounding Rules 109

10.4 CRN Rounding 110
# Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>BPENC</td>
<td>Binary priority encoder</td>
</tr>
<tr>
<td>BSD</td>
<td>Binary Signed-Digit</td>
</tr>
<tr>
<td>CFA</td>
<td>Carry-Free Adder</td>
</tr>
<tr>
<td>CSA</td>
<td>Carry-Save Adder</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary metal oxide semiconductor</td>
</tr>
<tr>
<td>CRN</td>
<td>Convert, Round, and Normalize</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute for Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>LSB</td>
<td>Least significant bit</td>
</tr>
<tr>
<td>MIPS</td>
<td>Millions of instructions per second</td>
</tr>
<tr>
<td>MSB</td>
<td>most significant bit</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>QDS</td>
<td>Quotient Digit Selection</td>
</tr>
<tr>
<td>PENC</td>
<td>Priority encoder</td>
</tr>
<tr>
<td>PG</td>
<td>Propagate and Generate</td>
</tr>
<tr>
<td>PP</td>
<td>Partial product</td>
</tr>
<tr>
<td>PPA</td>
<td>Parallel prefix adder – an adder with a prefix stage to convert the A and B inputs into propagate and generate signals, and uses a tree structure to compute the carries in parallel</td>
</tr>
<tr>
<td>RI</td>
<td>Round to infinity</td>
</tr>
<tr>
<td><strong>RISC</strong></td>
<td>Reduced instruction set computer</td>
</tr>
<tr>
<td><strong>RM</strong></td>
<td>Round to minus infinity</td>
</tr>
<tr>
<td><strong>RN</strong></td>
<td>Round to nearest</td>
</tr>
<tr>
<td><strong>RNE</strong></td>
<td>Round to nearest (Even)</td>
</tr>
<tr>
<td><strong>RP</strong></td>
<td>Round to positive infinity</td>
</tr>
<tr>
<td><strong>RZ</strong></td>
<td>Round to zero</td>
</tr>
<tr>
<td><strong>TSMC</strong></td>
<td>Taiwan Semiconductor Manufacturing Company</td>
</tr>
<tr>
<td><strong>ULP</strong></td>
<td>Unit of least precision – used interchangeably with LSB</td>
</tr>
<tr>
<td><strong>UPENC</strong></td>
<td>Unary priority encoder</td>
</tr>
<tr>
<td><strong>VHDL</strong></td>
<td>VHSIC Hardware Descriptive Language</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

Computing pioneer John von Neumann proposed the idea of a separate block for arithmetic and logic operations in 1945 [43]. The idea was that dividing the architecture of a central processing unit into separate functional blocks would allow designers to focus on the specific problems associated with the creation of a single group of specialized circuitry, without worrying about the remainder of the architecture. This is a vital concept in processor design, and especially in the work undertaken in the research and implementation of work presented in this paper. This paper focuses on the design of the arithmetic logic unit (ALU) and floating-point unit (FPU), which perform integer mathematical and logic operations, and decimal mathematical operations, respectively. The research of this paper was undertaken with the notion that it will be combined with theses on an instruction control unit and a cache hierarchy to create a functional processor based on the MIPS64 architecture.

The end goal of this thesis is the design, VHDL implementation and synthesis of the arithmetic logic unit and floating-point unit of a 64-bit reduced instruction set computer (RISC) processor. The RISC architecture, and specifically MIPS64, was chosen because it requires the implementation of relatively few instructions, while allowing for the same functionality of more complicated instruction sets. The MIPS64 instruction set was chosen because it is a well defined, fairly prevalent architecture in the vein of the RISC ideal.

The ALU and FPU are two of the most time critical components in a processor, so effort was made to reduce the delay as much as possible. There is always a tradeoff between speed, area, and power, with an increase in speed usually coming at the expense
of an increase in area and power usage. All three factors are limited by the technology used, and the type of logic implemented. The TSMC process used in this paper is highly scalable but is not known for its speed. Most modern processors utilize silicon on inductor (SOI) technology that allows for faster switching of transistors, and thus circuits. Dynamic logic typically performs faster than static logic, but uses more power. The TSMC 0.13μm library that was utilized for synthesis with Synopsys uses only static logic. As a result, it cannot be expected that the implemented design will be of comparable speed to commercial processors. Nor will they approach the same speeds as the implementations from the IEEE papers that they are based upon, if more modern technologies were utilized in those papers [26, 44, 49].

This remainder of paper is divided into two parts – one each for the ALU and FPU. The first part describes the implementation of the functions of the arithmetic logic unit. Integer adders, multipliers, and dividers are discussed in detail, with the basic logic operations – AND, OR, XOR, etc. - not focused upon, as they are simple to implement. The second part of the paper describes the floating-point unit. Implementations of double-precision adder/subtractors, multipliers, and dividers that conform to the IEEE 754 standard are presented.
Part I

Arithmetic Logic Unit

This section presents the implementation of a 64-Bit arithmetic logic unit based on the MIPS 64 instruction set. The arithmetic logic unit supports signed and unsigned addition, subtraction, multiplication and division, as well as arithmetic and logical shifting, and basic logic operations.
Chapter 2  ALU Overview

The arithmetic logic unit (ALU) is a fundamental component of the central processing unit (CPU) of a computer. As the name implies, the ALU handles all of the integer arithmetic operations – addition, subtraction, multiplication, and division – as well as the logic operations – OR, AND, XOR, and inversion – and operand shifting operations performed by the CPU. The ability to perform all of these operations results in the ALU being one of the most complex circuits in the CPU.

The concept of a separate unit designed specifically for arithmetic and logic was proposed by John von Neumann in 1945. Von Neumann stated that since a computer must always perform the basic mathematical and logic operations, it is “reasonable that [the computer] should contain specialized organs for these operations [43].” Implementing a logic block designed specifically for mathematical and logic operations has allowed designers to create many design improvements over the past 60 years, greatly reducing the delay caused by the mathematical operations, and thus allowing for much higher frequencies.

While one ALU is necessary for a computer, more is better. The use of multiplier ALU units allows multiple instructions to be executed in parallel, greatly increasing throughput for the processor. This comes at the cost of area, but with ever-decreasing transistor sizes, the area is typically sacrificed for the additional performance that having multiple arithmetic logic units offers.
2.1 Selected MIPS Instructions

The designed ALU takes a clock signal, two 64-bit operands, and a 4-bit opcode as inputs. The outputs of the ALU are a 128-bit signal spread across two 64-bit registers, and a status signal that denotes errors such as overflow, underflow, and divide-by-zero. The instructions implemented for the ALU are a subsection of the MIPS64 instruction set architecture (ISA) that deals with 64-bit arithmetic and logic operations [28]. A complete list of the implemented instructions can be seen in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>OP Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Operations</td>
<td>0 0 0 0</td>
<td>A + B</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1</td>
<td>A − B</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0</td>
<td>A * B</td>
</tr>
<tr>
<td></td>
<td>0 0 1 1</td>
<td>A / B</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0</td>
<td>A + B</td>
</tr>
<tr>
<td></td>
<td>0 1 0 1</td>
<td>A − B</td>
</tr>
<tr>
<td></td>
<td>0 1 1 0</td>
<td>A * B</td>
</tr>
<tr>
<td></td>
<td>0 1 1 1</td>
<td>A / B</td>
</tr>
<tr>
<td>Logic Operations</td>
<td>1 0 0 0</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1</td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>1 0 1 0</td>
<td>XOR</td>
</tr>
<tr>
<td></td>
<td>1 0 1 1</td>
<td>NOR</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0</td>
<td>INV</td>
</tr>
<tr>
<td></td>
<td>1 1 0 1</td>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td></td>
<td>1 1 1 0</td>
<td>Logical Shift Left</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1</td>
<td>Logical Shift Right</td>
</tr>
</tbody>
</table>

As shown in Table 1, all four of the arithmetic operations are implemented in both unsigned and signed (two’s complement) format. Similarly, the right shift is implemented both arithmetically – where the operation is sign extended as it is shifted – and logically – where 0’s are inserted into the bit positions that are shifted out. The shift
operations are performed on operand A, with the least significant six bits of B indicating
the number of bits to shift A. All of the shift operations, the addition, subtraction and
division operations output to the lowest 64-bits of the output, with the most significant
64-bits filled with 0's. The multiplication operation produces a 128-bit output that
utilizes both registers.

The logical operations implement a series of standard logic operations on the
operands at the bit level. The AND operation produces a 1 at output bit \( i \) only if \( A_i \) and
\( B_i \) are both equal to 1. The OR operation produces a 1 at output bit \( i \) if \( A_i \) or \( B_i \) is equal
to 1. The XOR operation produces a 1 at output bit \( i \) if either \( A_i \) or \( B_i \) is equal to 1, but
not both. The NOR operation is the opposite of the OR operation. A 0 is inserted at the
bit position if the operation conditions are not met. All four of the operations use only
the lowest 64 output bits, filling the higher bits with 0s. The INV operation inverts each
bit of both A and B. The inversion of B is output to the most significant 64-bits, while
the inversion of A is output to the least significant 64-bits. Using a combination of these
operations, any logic operation can be implemented.

The design of the arithmetic portion of the ALU is further broken down into the
universal adder/subtractor, universal multiplier, and universal divider to simplify
implementation. The selection of the proper functionality by the ALU is then selected
using multiplexers controlled by the opcode. The three arithmetic blocks that make up
the majority of the complexity of the ALU are described in Chapters 3, 4, and 5,
respectively.
Chapter 3   Integer Addition/Subtraction

Fast addition is extremely important in many digital systems. Adders are utilized in the critical path of address generation units, floating-point units and arithmetic logic units. As such, the characteristics of the adder are particularly important factors in the size, speed, and power efficiency of microprocessors.

The adder speed is critical to the throughput of a microprocessor, as it is involved in some way or another with every mathematical operation, in addition to its function in address generation. However, designing purely for speed can result in a design that uses a large amount of power. While this may be acceptable in devices that use power outlets, it can greatly reduce the battery life of portable devices. Similarly, fast designs are typically large designs, and the size of the adder influences the overall design size.

To implement high-speed addition, parallel prefix adders are commonly used. These adders utilize tree structures to compute the carries of each bit in parallel, reducing the time required for sum generation to a logarithmic function rather than a linear function. These carry trees are the critical features of parallel prefix adders both in terms of speed and area. As such, the trade-offs of the tree structures must be determined to select an appropriate adder for a design.

In this paper, two common types of tree structures – Kogge-Stone and Han-Carlson – are compared in terms of theoretical delay and area. A third design is then introduced which offers a cross between a parallel prefix adder and a carry-select adder. The three designs are implemented in VHDL, and synthesized in standard 0.13μm TSMC CMOS using Synopsys Design Compiler (2006.06). The designs are compared for power, timing and area.
3.1 Parallel Prefix Adders

3.1.1. Traditional Equations

All parallel prefix adders can be divided into three stages. The prefix generation stage takes in the addition inputs ‘A’ and ‘B’, where

$$A = \{a_{n-1}a_{n-2}...a_0\}$$

and

$$B = \{b_{n-1}b_{n-2}...b_0\},$$

and converts them into propagate and generate signals using the following computations:

$$g_i(A, B) = a_i \cdot b_i$$

$$p_i(A, B) = a_i \oplus b_i$$

The carry generation stage then uses the propagate and generate signals to form the carries. As their names imply, the propagate signals are used to transmit a carry-in into a carry-out, while the generate signals are able to create a carry-out regardless of the value of the carry-in. The propagates and generates can be grouped together to form a tree. In these groups, the values represent a larger number of bits per signal, but maintain the same properties. The method in which the grouping occurs, specifically the radix – the number of p,g pairs per operation – and the sparseness – the number of bits skipped for every bit used in the tree structure – are the cause of the variation in speed and area of the adders. The radix-2 groupings are as follows:

$$P_{1:0} = p_1 \cdot p_0$$

$$G_{1:0} = g_1 + p_1 \cdot g_0$$

The recursive merge operation is expressed in one of the two following ways, where the second (carry) equation is essentially a subsection of the first (merge) equation, with the propagate logic removed to reduce the area.
\[(G, P) \circ (G', P') = (G + P \cdot G', P \cdot P')\]

\[C_{i+1} = g_i + (p_i \cdot C_i)\]

The final stage is sum generation. In this step, the sum of the A+B is computed as

\[S_i = p_i \oplus C_i,\]

where \(C_i = G_{i-1.0}\).

### 3.1.2 Ling Equations

In 1981, Ling published a paper detailing how to create a faster parallel prefix adder, regardless of the tree structure used in for the design [17]. The speed increase is accomplished by replacing the propagate signal with

\[t_i(A, B) = a_i + b_i.\]

Since \(g_i\) and \(t_i\) both cover the case when \(a_i = b_i = 1\), the Ling equations produce what is called a pseudo-carry, which represents a carry-in or a carry-out of a bit position, rather than the carry-out signal that the traditional equations produce.

\[H_i = G_i + G_{i-1}\]

\[H_i = g_i + t_{i-1} \cdot H_{i-1}\]

In addition to using OR for the propagation operation, which is less costly than an XOR operation, the pseudo-carry further reduces the costs of the tree because \(t_i \cdot g_i = g_i\), resulting in one less logic gate per operation. For example:

\[G_{30} = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0\]

in the tradition equations becomes
\[ H_{30} = g_3 + g_2 + t_2 \cdot g_1 + t_2 \cdot t_1 \cdot g_0 \]

in the Ling equations. The propagate equations for the merge operations are essentially the same as the traditional equations, with

\[ P_{1:0} = p_1 \cdot p_0 \]

in the traditional equations becoming

\[ T_{1:0} = t_0 \cdot t_{-1} \]

in the Ling equations.

This efficiency in the prefix and carry generation stages outweighs the slower sum generation stage implemented with the Ling equations. The sum stage is more complex due to the fact that the carry bit needs to be derived from the pseudo carry. The carry can be derived as

\[ C_i = t_i \cdot H_{i:0} \]

resulting in a sum of

\[ S_i = p_i \oplus (t_i \cdot H_{i:0}) \]  [49].

In spite of the increased sum delay, analysis shows that the Ling equations always outperform the traditional equations [22].

### 3.2 PPA Tree Structures

The design of the carry tree is the critical component in creating a trade-off between speed and area. There has been a large amount of research done in this area, resulting in many different structures. The structures presented in this section are the most commonly used, regular structures.
3.2.1 Kogge-Stone

In 1973, Kogge and Stone introduced a densely packed, regular tree structure that uses the minimum possible number of stages, allowing for the highest possible theoretical speed of any parallel prefix adder [1]. This tree – shown in Figure 3.1 without a carry-in bit – has often been utilized due to its high speed, regular layout and limited fan-out – each output drives at most two merge operations after buffering.

![Figure 3.1: 64-Bit Kogge-Stone Adder](image)

While the Kogge-Stone adder offers high speed, it requires a large amount of area to implement due to the large number of carry and merge operations. The merge operations are depicted by the shaded dots in the figures, while the white dots represent the carry operations. The white and shaded squares represent the propagate and generate (pg) logic, and sum computations respectively.
Another problem with the Kogge-Stone adder is that it is ill adapted to handle a carry-in. In order to add the carry-in to the tree, a specialized pg generation or carry operation must be created to integrate the carry-in into the tree. Figure 3.2 shows the Kogge-Stone tree adapted for a carry-in. Note the additional carry operation, and that the second carry dot in the first stage includes 3 inputs, both of which are required to propagate the carry-in. The 3 input carry increases the delay of the tree by increasing the number of gates in the logic for the dot to: \( (G_{10} \circ Cin) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot Cin \).

![Figure 3.2: 64-Bit Kogge-Stone Adder with Carry-in](image)

### 3.2.2 Brent-Kung

The Brent-Kung adder [1] was introduced in 1982 and takes the opposite approach of the Kogge-Stone adder. Rather than attempting to create the fastest parallel prefix adder, Brent and Kung tried to create the smallest. The Brent-Kung adder has a sparseness of 2 – alternating columns are skipped until the last stage – trading extra tree stages for fewer merge operations.
While the additional stages of the Brent-Kung adder preclude it from being included in a discussion of high-speed adders, and thus not seriously considered for use in the ALU designed in this paper, it is an important precursor to the Han-Carlson adder discussed in the following section.

![Diagram of 64-Bit Brent-Kung Adder](image)

**Figure 3.3: 64-Bit Brent-Kung Adder**

### 3.2.3 Han-Carlson

The Han-Carlson adder [19] was introduced in 1987. It combines the philosophies behind the Kogge-Stone and Brent-Kung adders – in fact, Parhami [39] refers to it as a hybrid Brent-Kung/Kogge-Stone adder, rather than a Han-Carlson. The Han-Carlson adder utilizes the merge pattern of the Kogge-Stone adder, but only performs the operations on the odd number bits, giving it a sparseness of 2 like the Brent-Kung Tree. The result is a structure that requires one additional stage to generate the carries than does the Kogge-Stone, but is significantly smaller in area. This tradeoff of area and speed make the Han-Carlson adder a popular choice for recent designs.
Like the Kogge-Stone adder, the Han-Carlson adder can be converted to handle a carry-in by altering the carry operation in the first stage of the tree to handle three inputs. However, the sparseness of the Han-Carlson adder allows for a better design to handle the carry-in than does the denser Kogge-Stone adder. To accommodate the carry-in, A and B can be shifted left along the tree by one bit, inserting the carry-in where the least significant bits of A and B were positioned. In this way, the sum can be computed in the same number of stages, with the addition of a single carry dot to compute the carry-out bit. This presents an optimal solution to the implementation of the carry-in because it only alters the final stage of the tree and does not add any delay due to the fact that the carry operation added to the tree is the same as the other operations implemented in the final stage. The Han-Carlson adder modified for a carry-in can be seen in Figure 3.5.
3.2.4 Comparison of Radix-2 PPAs

As can be seen in Table 3.1, the Kogge-Stone adder is theoretically the fastest and largest for any size inputs, the Brent-Kung is the smallest and slowest, and the Han-Carlson presents a tradeoff between size and speed. This comparison only takes the number of logic operations into consideration, ignoring the effects that interconnect delays may have upon the adders. In the table, k represents the operand width in terms of bits. As will be demonstrated in subsequent discussions, wiring delays, fan-outs and capacitances can greatly affect the synthesis and actual results achieved by the adders.

Table 3.1: Comparison of Radix-2 PPAs for Theoretical Area and Delay [18]

<table>
<thead>
<tr>
<th>Factor</th>
<th>Brent Kung</th>
<th>Kogge Stone</th>
<th>Han Carlson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>$2 \cdot \log_2 k - 2$</td>
<td>$\log_2 k$</td>
<td>$\log_2 k + 1$</td>
</tr>
<tr>
<td>Area</td>
<td>$2k \cdot 2 \cdot \log_2 k$</td>
<td>$k \cdot \log_2 k \cdot (k-1)$</td>
<td>$k/2 \cdot \log_2 k$</td>
</tr>
</tbody>
</table>
3.2.5 **Radix-4 Han-Carlson Adder**

As mentioned previously, increasing the radix of the tree can reduce the number of stages in an adder tree. The radix is equal to the number of maximum number of inputs that the merge operations have. As such, an increase in the radix causes an increase in the amount of logic required to process the merge operations, so radices are typically limited to 2, 4, or 8 so that the amount of logic in each operation does not become overly large.

As can be seen in Figure 3.6, the increase in radix also increases the fan-out of each merge operation. This increased load can result in a slower operation time depending on the type of logic and process used to create the adder. It is reported in [49] that using an SOI process, radix-4 adders perform better than radix-2 adders; however Synopsys synthesis results conducted in TSMC 0.13µm static CMOS report that the radix-2 implementations perform at higher speeds, in spite of having three more stages of carry and merge operations.

![Figure 3.6: 64-Bit Radix-4 Han-Carlson Adder](image)
3.3 Hybrid Adder

Recently, adder designers have focused on combining parallel prefix adders with carry select adders to produce faster designs [22, 49]. Carry select adders generate the possible sums for each bit in parallel with the computation of the carries. The carries are then used to select the correct sums. When this approach is combined with the fast carry generation of the parallel prefix adders, it can result in a design that is faster than a parallel prefix adder.

The speed increase is dependant upon the parallel prefix structure used. If a Kogge-Stone design is utilized, the sum select multiplexers replace the sum generation stage of the adder, resulting in a similar speed, with a larger area. If a Han-Carlson structure is utilized, the sum-select multiplexers can replace both the final stage of merge operations, and the sum generation, using each of the n/2 carries generated to select two sum bits. This results in an adder that has the same number of logic levels as the Kogge-Stone carry-select adder, but is significantly smaller.

To implement the Han-Carlson carry-select adder, the Ling Han-Carlson tree is used, with the log2k+1 stage removed. The sum pre-computation [49] for the odd bits and 0 is

\[ S_i^0 = a_i \oplus b_i \]
\[ S_i^1 = a_i \oplus b_i \oplus (a_{i-1} + b_{i-1}) \]

and the even sum pre-computes are computed as

\[ S_i^0 = a_i \oplus b_i \oplus (a_{i-1} \cdot b_{i-1}) \]
\[ S_i^1 = a_i \oplus b_i \oplus [a_{i-1} \cdot b_{i-1} + (a_{i-1} + b_{i-1})(a_{i-2} \cdot b_{i-2})] \]
where the superscript signifies the even carry-out values used for sum selection. Each carry-out is used to select both the subsequent odd sum and the following even sum.

After comparing the synthesis results for the size, speed, and power of the various adders discussed in this chapter, this hybrid Han-Carlson carry-select design was selected to implement the 64-bit adder/subtractor, as it has the highest speed, while still having a smaller area than the Kogge-Stone adder. The synthesis results can be viewed in Chapter 6, where the ALU synthesis results are discussed.

![Figure 3.7: 64-Bit Hybrid Han-Carlson Carry-Select Adder](image)

### 3.4 Subtraction Using Addition

As any elementary school child knows, addition and subtraction are essentially the same operation. Subtraction merely inverts the sign of the second operand. Using this knowledge, it becomes simple to implement integer subtraction using any type integer adder.
To alter the adder to be able to handle both integer addition and subtraction, logic must be established to perform two’s complement conversion on the B operand if subtraction is selected, and to leave B untouched if addition is selected. This can be done in one of two ways: using an inverter and a multiplexer on each bit, or an XOR gate with one bit tied to a control, and the other tied to the corresponding bit of B. As a multiplexer and an XOR gate have similar delays using the static CMOS in the TSMC process, the XOR gate method is used for the inversion to remove the additional logic necessary for the alternative method.

The inversion of the B operand results in the one’s complement of B. However, modern computers typically operate using two’s complement format for signed numbers. To convert the one’s complement value into a two’s complement value, 1 must be added to the least significant bit. This can be accomplished during the addition/subtraction by adjusting the carry-in bit accordingly. Since the MIPS instructions do not include a carry-in, the carry-in can be tied directly to the control bit that selects between addition and subtraction. In this method the control bit will be ‘1’ if subtraction is selected, causing the inversion of the B operand, and the addition of a 1 to the least significant bit via the carry-in – completing the conversion for subtraction.

3.5 Signed and Unsigned Addition/Subtraction

Neither addition nor subtraction requires any modifications to switch between signed and unsigned operation when two’s complement format is used for the signed representation. The difference lies in the way that the results are interpreted, with the MSB representing the sign in signed format, and the MSB in unsigned format.
3.6 Addition/Subtraction Exceptions

Integer addition and subtraction also lend themselves nicely to error detection. According to the MIPS ISA, the only exceptions that need to be detected for either are overflows for unsigned operations. These can be detected directly from the carry-out of the adder, with no additional logic required.

![Add/Subtractor Diagram](image)

**Figure 3.8: Adder/Subtractor**

3.7 Summary

This chapter provides an overview for some of the basic types of parallel prefix adders, presenting strengths and weaknesses for each. An adder combining the strengths of the Han-Carlson parallel prefix structure, with those of the carry-select adder is then introduced as a higher speed addition architecture. This Han-Carlson/Carry-select hybrid is then leveraged into performing both unsigned and signed addition and subtraction through a simple modification of the B operand, and the carry-in. The carry-out of the adder is used to detect the overflow condition, as specified in the MIPS ISA.
Chapter 4  Integer Multiplication

Multiplication, like addition, is a heavily used operation that figures prominently in many types of operations. Among many other uses, multiplication is used in signal processing and scientific applications. It is also a common basis for division. With such wide-ranging applications, multiplication has been a heavily researched area of digital design, with many different types of multipliers and optimizations proposed.

Integer multipliers can be implemented in a variety of ways. Typical implementations range from the slow, but small, shift and add sequential multipliers to the larger and faster tree multipliers. Regardless of the implementation used, integer multiplication is the only mathematical operation explored in this paper that creates a 2n-bit output from n-bit inputs. In this chapter shift and add, and tree multipliers are described, and a universal (signed/unsigned) integer multiplier is designed for use in the ALU.

4.1 Multiplier Types

4.1.1  Shift and Add

Shift and add multipliers are the simplest and easiest to implement types of multipliers. The basic unsigned radix-2 shift and add multiplier is the binary equivalent of the multiplication method taught in grade school. The A input is multiplied by each bit of B in turn to create each of the n partial products, where n is the number of bits of B. Each partial product is then shifted left by the bit position of B from which it was created, with the shifted bits and the unused significant bits filled with 0’s. The partial products
are added together to form the product. In this implementation, one partial product is generated and added to the running product per clock cycle. To expand this implementation to handle signed notation, each partial product must be sign extended to 2n bits, rather than grounding the unused bits. 4-bit examples of both signed and unsigned are presented in Figure 4.1.

![Figure 4.1: 4-Bit Unsigned and Signed Multiplication Examples](image)

This implementation is fairly straight forward, requiring only an n-bit adder, a 2n-bit register and a multiplexer. The multiplier is initially loaded into the least significant n-bits of the register. The least significant bit of the register controls the multiplexer, selecting the multiplicand if 1 and ground if 0. The most significant n-bits of the partial product are then added to the output of the multiplexer, and the resulting sum and carry-in are loaded into the most significant n+1 bits of the partial product register. Bits n down to 1 of the partial product are loaded into the lowest n-1 bits of the register, resulting in a shift of the multiplier. This method takes n clock cycles, but allows for a high frequency and small area.
Although the radix-2 implementation of the shift and add multiplier is the simplest type of multiplier to implement, it is unsuitable to high speed circuits due to the large latency that the large number of required clock cycles creates. This problem can be partially rectified by increasing the effective radix of the multiplier by recoding the operands to reduce the number of partial products. The most common method of doing this is through Booth encoding, which is discussed in detail later in this chapter.

4.1.2 Tree Multipliers

In a tree multiplier, the partial products are computed and added in parallel. The tree multipliers reduce the number of partial products by a set factor each stage, and can execute multiple stages per clock cycle. Typical trees utilize 3:2 or 4:2 compressors, which reduce the number of parallel products to $2/3$ and $1/2$, respectively, at each stage of the tree other than the first. While these circuits result in multipliers with greatly reduced...
latency, the circuitry required to compute the partial products in parallel, and to perform the tree compressions results in a much larger circuit.

The traditional trees – Wallace [46] and Dadda [6] – were introduced in the mid 1960’s, and used a series of full adders and half adders to sum the partial products. Since these circuits consist of three inputs and two outputs, the tree structures were irregular. To make the structures more regular, compressor structures with a larger number of inputs were introduced. In this paper, a simple 4:2 compressor is utilized. As can be seen in Figure 4.3, this structure allows for a more regular tree structure by cutting the number of partial products in half at each stage of the tree. Since the outputs of two partial product summations become the inputs of the following summation, the 4:2 becomes simpler to map and layout than the 3:2 tree.

Figure 4.3: 3:2 and 4:2 Wallace Trees with 7 inputs
Each of the carry-save adders (CSA) in the 3:2 Wallace tree is a series of full adders, producing the carry and sum outputs. The 4:2 CSAs in the 4:2 Wallace tree also produce the carry and save outputs for each bit using a 4:2 compressor. As the compressor unit makes up the majority of the Wallace tree multiplier, it has been a widely researched topic [2, 20, 24, 29]. The logic for the Nagamatsu [29] 4:2 compressor is shown in Figure 4.4.

![4:2 Compressor Stage](image)

**Figure 4.4: 4:2 Compressor Stage [3, 29]**

### 4.2 Booth Encoding

Independent of the type of multiplier used, the simplest way to speed up the multiplier is to reduce the number of partial products that must be added together. This can be done through recoding methods, the most famous of which was developed by Booth in 1951 [3].

Booth deduced that by analyzing one of the two operands for strings of 1’s and 0’s, it could be recoded so that it might produce fewer partial products. Booth did this by analyzing groups of two consecutive bits. If the bits were the same value, the partial
product was a string of 0’s. If the most significant of the two bits was 0 and the other bit was 1, it was noted that a string of ones was ending and the other operand (shifted accordingly) was used for the partial product. If the most significant of the two bits was 1 and the other bit was 0, a string of 1’s was starting and the two’s complement of the other operand was used as the partial product. To ensure that the least significant bit is not seen as a continuation of a string of 1’s, a 0 is appended to the end of the analyzed operand. This method has the potential to remove a large number of partial products if the 0’s are merely shifted over, and not added. However, the reduction of partial products is entirely dependent upon the operand being analyzed, and if the operand alternates 0’s and 1’s, this method provides no benefit.

To create a recoding method that is always beneficial, Booth encoding can be extended to analyze three consecutive bits, with the index of the middle bit incremented by two so that the middle of the group is only analyzed once. This method forces the multiplier into radix-4 format, and reduces the number of partial products by half.

**Table 4.1: Modified-Booth Encoding [4]**

<table>
<thead>
<tr>
<th>$b_{i+1}$</th>
<th>$b_i$</th>
<th>$b_{i-1}$</th>
<th>PP Value</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Midstring of 0s</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>End string of 1s</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A</td>
<td>Single 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2*A</td>
<td>End string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-2*A</td>
<td>Begin string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-A</td>
<td>Single 0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-A</td>
<td>Begin string of 1s</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Midstring of 1s</td>
</tr>
</tbody>
</table>
It should be noted that Booth encoding is inherently signed. To produce an unsigned product, the analyzed operand must be buffered with a 0 at the most significant bit, potentially creating the need for an additional partial product [15]. This bit must be a sign extension of the MSB if the same circuit is to be used for signed multiplication as well. Thus, in order to achieve a universal multiplier, the mode and MSB are ANDed together for each operand and concatenated with the operand to form an n+1 bit extended operand.

4.3 Booth-Wallace Multiplier

In order to produce a high-speed multiplier, it is common to combine modified Booth encoding with a Wallace tree structure [3, 25, 27]. The modified Booth encoder reduces the number of partial products from 64 to 32 (65 to 33 when the universal multiplier is implemented) and the Wallace tree is used to add the partial products quickly.

It is this Booth-Wallace approach that is used for the universal multiplier implemented for the ALU in this paper. To prepare the operands for the Booth encoding, B is sign extended two bits and a 0 is appended. A is sign extended one bit. The sign extensions are necessary to switch between signed and unsigned multiplication, while the appended 0 on B prevents the initial group from being marked as a continuation of a string of 1s. The modified A and B values are thereby denoted as,

\[
A_{\text{ext}} = \text{SE} \& A \\
B_{\text{ext}} = \text{SE} \& \text{SE} \& B \& 0
\]
where the '& ' symbol represents a concatenation operation. The sign extensions here are not true sign extensions, but are instead defined as $SE = MSB \cdot S$, where S is 1 if a signed operation is being performed, and 0 otherwise. This forces the MSB to 0 for unsigned operations, and extends the sign for signed operations.

The 67-bit extended B operand is then divided into groups of 3 bits, with the end bits of the group overlapping, to create 33 partial products. The additional partial product creates a non-symmetrical tree, which results in an additional stage. The outputs of the tree are added together to form the 128-bit product. No exceptions must be handled for integer multiplication. Figure 4.5 shows the structure of the Booth-Wallace Multiplier that was designed in this paper.

![Booth-Wallace Multiplier](image)

**Figure 4.5: Booth-Wallace Multiplier**
4.4 Summary

In this chapter, a universal 64-bit by 64-bit integer multiplier was designed. The multiplier uses a Wallace tree to provide a regular structure to add the partial products in parallel. A modified Booth encoder is used to reduce the number of partial products by half, increasing the radix from two to four. To allow for both signed and unsigned operation, A and B must extended by a bit, introducing an additional partial product, bringing the final tally to 33. Integer multiplication does not produce any observable exceptions.
Chapter 5  Integer Division

Division is the least used of the four basic arithmetic operations. As such, it has been least researched of the four operations, and remains the most difficult operation to implement efficiently. Although many different high-speed, high-precision mathematical algorithms have been developed in the past 50 years, few are suitable for implementation in VLSI, and those that are require a large clock cycle per iteration or many clock cycles to implement. As such, division remains the limiting factor in ALU performance.

5.1 Types of Dividers

Oberman and Flynn [35] group division algorithms into five classes: digit recurrence, functional iteration, very high radix, table lookup and variable latency. Digit recurrence and functional iteration are the predominate types of multipliers, although the implementations of each typically incorporate aspects of the other three groups.

5.1.1 Functional Iteration

Functional iteration is the division equivalent of a tree multiplier. Division by functional iteration uses a multiplier for the fundamental operator. As a result, the quotient is converged upon quadratically - the number of correct bits is doubled each iteration, much as it is halved each clock cycle of a tree multiplier. While the functional iteration method requires fewer clock cycles than a linear approach, the use of a multiplier as the fundamental operation greatly increases the required clock period, as the multiplier has a significantly longer delay than the adders that serve as the fundamental operators of the digit recurrence algorithm.
The basis for the functional iteration method can be seen by expressing the quotient \( Q \) as the product of the dividend \( x \) and reciprocal of the divisor \( d \):

\[
Q = \frac{x}{d} = x \cdot \frac{1}{d}.
\]

As the dividend is provided, the difficulty in this method is computing the reciprocal of the divisor. It is the method of obtaining the reciprocal that varies between functional iteration algorithms. The most common types of functional dividers are the Newton-Raphson and Goldschmidt algorithms.

The Newton-Raphson divider gets its name from the fact that it is based on the Newton-Raphson convergence algorithm

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}
\]

which is combined with the priming function

\[
f(X) = \frac{1}{X} - d = 0
\]

to form the approximation equation

\[
X_{i+1} = X_i \cdot (2 - d \cdot X_i).
\]

The corresponding error is given by the equation

\[
\varepsilon_{i+1} = \varepsilon_i^2 \cdot (d).
\]

From the equations, it can be seen that each iteration of the Newton-Raphson divider requires two multiplications and a subtraction.

The Goldschmidt algorithm uses a Maclaurin series expansion to approximate the reciprocal of the divisor to similar effect. The mathematics behind this approach can be seen in [1].
5.1.2 Digit Recurrence

Digit recurrence is the simplest and most widely implemented class of division algorithms. It utilizes a shift and subtract approach that is similar to the shift and add multiplication algorithm introduced in Chapter 4. A fixed number of quotient bits are retired per iteration, with the number of bits retired dependent upon the radix of the implementation. The basic recurrence algorithms are also similar to shift and add multiplication in that they require a small amount of area to implement, but the large number of required clock cycles results in a large latency. The radix of the algorithms can be increased to reduce the overall latency, at the expense of requiring a larger area and longer clock cycle. In fact, the possible variations in digit recurrence division are numerous enough to warrant entire textbooks on the subject. Ercegovac and Lang [10] have created such a textbook, which presents a comprehensive analysis of digit recurrence algorithms, and is recommended for those who require further information on the topic.

Digit recurrence division is defined by the equations:

\[ x = q \cdot d + rem \]

\[ |rem| < |d| \cdot ulp \]

\[ sign(rem) = sign(x) \]

where the dividend \( x \), and the divisor \( d \) are the operands, \( q \) is the quotient, and \( rem \) is the remainder. In integer division, the unit of least precision (ULP) is 1, as there are no fractional bits. Digit recurrence assumes that the operands \( x \) and \( d \) are normalized at the time of implementation.
Using the number of bits to be retired per iteration (b), and the number of bits in the operands (n = 64 for integer division), the radix (r), and number of iterations (k) are defined as

\[ r = 2^b, \quad k = \frac{n}{b}. \]

The following equation is the critical recurrence used in each iteration of the algorithm:

\[ P_{j+1} = r \cdot P_j - d \cdot q_{j+1} \]

where \( P_j \) is the partial remainder (residual) at iteration \( j \), and \( q_j \) is the quotient bits as determined by the quotient digit selection algorithm. As digit recurrence is a subtractive algorithm, it is necessary to use the initial condition of

\[ r \cdot P_0 = x. \]

The quotient digit selection (QDS) algorithm can be implemented in a number of ways, so for simplicity’s sake, it is written as

\[ q_{j+1} = SEL(r \cdot P_j, d). \]

Using the QDS values, it is possible to determine the quotient after the \( j \)th iteration using the following equation. It should be noted that the final quotient occurs when \( j = k \).

\[ q = \sum_{j=1}^{j} q_j \cdot r^{-j} \]

The remainder is defined as

\[ \text{rem} = \begin{cases} P_k & \text{if } P_k \geq 0 \\ P_k + d - ulp & \text{if } P_k < 0. \end{cases} \]
Digit recurrence algorithms can be divided into two groups: restoring and non-
restoring. Restoring division supports only non-negative values of \( q_j \), and must revert to
the last state before continuing with the next iteration if a negative \( q_j \) is formed. This
reversion takes time and logic. In restoring division, the multiples of \( d \) act as the
separation point for the values of \( q_j \). In radix-2 format, \( q_j \) is defined as
\[
q_{j+1} = \begin{cases} 
0, & \text{if } rP_k < d \\
1, & \text{if } d < rP_k < 2d.
\end{cases}
\]

To eliminate the reversion step, non-restoring division utilizes both positive and
negative values of \( q_j \). By using both positive and negative values for \( q_j \), one of the
separation points can be 0, simplifying the QDS selection.
\[
q_{j+1} = \begin{cases} 
0, & \text{if } rP_k < d \\
1, & \text{if } d < rP_k < 2d.
\end{cases}
\]

The quotient digit selection algorithms can be further reduced through the use of
redundant digits sets. Redundant digit sets have more digits than the digit set that they
represent. So a redundant digit set for a radix-2 circuit would have 3 or more numbers.
Redundancy allows for more separation points, allowing for simpler QDS functions. For
simplicity, binary signed digit (BSD) representation is typically used for the redundant
digit set. BSD is a symmetrical digit set that is defined by the following rules:
\[
\{a, a-1, \ldots, 1, 0, 1, \ldots, a-1, a\} \quad \text{where } n = -n, \text{ and } \frac{r}{2} \leq a \leq r - 1
\]

Numbers expressed in BSD format are represented by two arrays of bits. One
array represents the positive bits, while the other represents the negative bits. BSD
format can be converted into binary by subtracting the negative bits from the positive
bits. Using this methodology, each bit of the BSD number can be decoded as seen in Table 4. The sign of the BSD number is decided by the most significant bit where the positive and negative values differ, with a larger negative value resulting in an underflow, and creating an automatic sign extension. BSD format also has the advantage of being invertible in zero time – without logic – simply by switching the positive and negative arrays.

### Table 5.1: BSD Conversion

<table>
<thead>
<tr>
<th>BSDpos</th>
<th>BSDneg</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Each signed digit set is measured by a redundancy factor $p$, where

$$\frac{1}{2} < p = \frac{a}{r-1} \leq 1.$$ 

A greater redundancy results in more overlap in the regions covered by the digits, allowing for greater leniency in selecting a separation point. It also results in more complex logic to select the appropriate value.

Some sample SD sets for radix-2, radix-4 and radix-8 implementations can be seen in the following table. In this table, maximally redundant means that $a = r-1$, and represents the largest SD set, while minimally redundant means that $a = 0.5*r$, and represents the minimal SD set.
Table 5.2: Redundant Digit Sets [31]

<table>
<thead>
<tr>
<th>$r$</th>
<th>$a$</th>
<th>SD set</th>
<th>$\rho$</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>${0, 1}$</td>
<td>1</td>
<td>Maximally and Minimally redundant</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>${0, 1, 2}$</td>
<td>$\frac{2}{5}$</td>
<td>Minimally redundant</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>${0, 1, 2, 3}$</td>
<td>1</td>
<td>Maximally redundant</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>${0, 1, 2, 3, 4}$</td>
<td>$\frac{4}{5}$</td>
<td>Over redundant</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>${0, 1, 2, 3}$</td>
<td>$\frac{2}{7}$</td>
<td>Non-redundant</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>${0, 1, 2, 3, 4}$</td>
<td>$\frac{4}{7}$</td>
<td>Minimally redundant</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>${0, 1, 2, 3, 4, 5, 6, 7}$</td>
<td>1</td>
<td>Maximally redundant</td>
</tr>
</tbody>
</table>

5.2 SRT Division by Comparison Multiples

The most common implementation of digit recurrence division in modern processors is SRT division, named for Sweeny [5], Robertson [40], and Tocher [45], who independently developed a method of performing non-restoring division around the same time. They proposed the inclusion of 0 into the signed digit set. This allows some iterations to be reduced to shifting, reducing latency in asynchronous implementations.

$$ q_{j+1} = \begin{cases} 
1, & \text{if } rP_k < -d \\
0, & \text{if } -d \leq rP_k < d \\
1, & \text{if } rP_k \geq d 
\end{cases} $$

SRT division is easily expandable to higher-radix division by implementing more complex QDS functions. There are three basic methods of implementing the QDS function. The selection intervals method uses Robertson’s diagram to graphically determine the precision required for $rP_j$ and $d$ to choose $q_{j+1}$ from a lookup table. The selection constants method is the most commonly implemented QDS algorithm. It also uses a lookup table, but forms separation points based on the value calculations using $r$ and $p$, rather than graphically. Information about these methods can be found in [10].
The third method, comparison multiples, is used in the divider implemented in this paper, and will be described in the following section.

5.2.1 *QDS by Comparison Multiples*

Although most processors implement SRT division using the selection constants method, a comparison multiples approach is used for the dividers in this paper. With this approach, the residuals are compared to multiples of d to determine separation points for the values of $q_{j+1}$. The residuals are compared to each of the separation points in parallel to determine which region they fall into and thus which value of $q_{j+1}$ to output. Ercegovac and Lang [9] report that this method is complicated due to the need for multiples and comparisons; however it is shown in [31] that this method is a reasonable alternative to selection constants.

Each value of $q_{j+1}$ is bounded by an upper and lower limit as shown in the equation

$$d(l - p) \leq rP_j \leq d(l + p)$$

where $l \in \{a, a-1, ..., 1, 0, 1, ..., a-1, a\}$. When redundant digit sets like BSD are used, the upper bound of $q_{j+1} = l$ overlaps the lower bound of $q_{j+1} = l+1$. It is in this overlap region that a multiple of d is selected to act as a separation point between $l$ and $l+1$. 
Using the bounds for $rP_j$, $q_{j+1}$ can be defined as:

$$q_{j+1} = \frac{1}{\pi^*} \frac{rP_j}{1 - \rho}$$

The overlap between values allows for more error in defining the separation points, allowing for truncated multiples to be used for the comparisons. The truncated multiples, reduced to c bits, allow for smaller and faster circuits for the determination of the comparison multiples. Defining $M_k$ as $M_k = A_k d$, where $A_k$ is a rational number, and using the symmetry $M_{-k+1} = -M_k$, where $k \in \{1, \ldots, a-1, a\}$, the number of comparison multiples can be reduced to $a$, and $q_{j+1}$ can be rewritten as

---

Table 5.3: QDS Function

<table>
<thead>
<tr>
<th>$q_{j+1}$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$rP_j \leq -d(1 - \rho)$</td>
</tr>
<tr>
<td>$\frac{1}{\pi} - 1$</td>
<td>$-d(1 + \rho) \leq rP_j \leq -d(1 - \rho)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>0</td>
<td>$-d\rho \leq rP_j \leq d\rho$</td>
</tr>
<tr>
<td>1</td>
<td>$d(1 - \rho) \leq rP_j \leq d(1 + \rho)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$a - 1$</td>
<td>$d(a - 1 - \rho) \leq rP_j \leq d(a - 1 + \rho)$</td>
</tr>
<tr>
<td>$a$</td>
<td>$d(a - \rho) \leq rP_j$</td>
</tr>
</tbody>
</table>
The number of bits required for the truncated comparison multiples was determined in [7] as follows. It should be noted that this method was derived for a floating-point multiplier normalized so that \( \frac{1}{2} \leq d < 1 \). However, these equations can be easily adapted to integer division by assuming that \( d \) has been shifted so that the most significant non-zero bit is the first bit of a fraction.

Truncating a BSD number \( X \) to \( t \) bits results in the inequality

\[
\{ X \}_t - 2^{-t} < X < \{ X \}_t + 2^{-t},
\]

which when combined with the inequalities used in the QDS function changes to

\[
M_t - 2^{-c} < \{ M_t \}_c \leq \{ rP_j \}_c < rP_j + 2^{-c}.
\]

Removing the middle statements in the inequality results in

\[
M_t - 2^{-c+1} < rP_j.
\]

As \( q_{j+1} = l \), adding the recurrence subtraction \( -ld \) results in an inequality involving the subsequent residual.
\[ M_i - 2^{-c+1} - ld < P_{i+1} \]

In order for the division to converge,

\[-pd \leq P_{i+1} < pd,\]

so

\[-pd \leq M_i - 2^{-c+1} - ld\]

must be met. Reworking this equation, the lower bounds on \( M_i \) is derived as

\[ d(l - p) + 2^{-c+1} \leq M_i. \]

Similarly, the upper bounds on \( M_i \) is derived as

\[ M_{i+1} \leq d(l + p) - 2^{-c}. \]

Combining the two bounds on \( M_i \) gives the inequality:

\[ d(l - p) + 2^{-c+1} \leq M_i \leq d(l + p - 1) - 2^{-c}, \]

which must be maintained for the truncation to be a viable substitution for the complete number. Eliminating the middle condition and reworking the equation gives

\[ 2^c > \frac{3}{d(2p-1)}. \]

Since \( d \geq \frac{1}{2} \) in the shifted form, the final equation for minimum number of bits in the truncated comparison multiples is based on the redundancy factor as

\[ 2^c > \frac{6}{2p-1}. \]

The QDS function must also allow for the maximum possible overflow of the additions. This number of bits is defined as \( e \), and is based on the radix and number of integer bits \( i \) coming into the QDS function as \( e = \log_2 r + i \).
5.3 Implementation of SRT Division by Comparison Multiples

5.3.1 QDS by Comparison Multiples

Before beginning the design of the hardware for the divider, the radix and digit set that will be used must be determined. As the Booth-Wallace multiplier in Chapter 4 was implemented in radix-4 format, the radix for the divider has also been chosen to be 4. To minimize the amount of hardware, the BSD set was chosen such that $a = 2$, resulting in a redundancy factor of $p = \frac{2}{3}$. Using the redundancy factor, the truncation precisions are determined to be $c = 5$ and $e = 2$, as the number of integer bits into the QDS function is said to be 0 after the theoretical shifting.

Table 5.4: Upper and Lower Bounds for $r=4$, $a=2$ BSD Set [35]

<table>
<thead>
<tr>
<th>$L$</th>
<th>$L_l = d(l-p)$</th>
<th>$U_l = d(l+p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>$-(8/3)*d$</td>
<td>$-(4/3)*d$</td>
</tr>
<tr>
<td>-1</td>
<td>$-(5/3)*d$</td>
<td>$-(1/3)*d$</td>
</tr>
<tr>
<td>0</td>
<td>$-(2/3)*d$</td>
<td>$(2/3)*d$</td>
</tr>
<tr>
<td>1</td>
<td>$(1/3)*d$</td>
<td>$(5/3)*d$</td>
</tr>
<tr>
<td>2</td>
<td>$(4/3)*d$</td>
<td>$(8/3)*d$</td>
</tr>
</tbody>
</table>

With the chosen digit set, the upper ($U_l$) and lower ($L_l$) bounds of $l$ are given according to the above table. Analyzing the lower bound of $l$ with the upper bound of $l-1$, it can be seen that due to symmetry, only two separation points – and thus two comparison multiples – need to be derived. The multiples are the overlap between $l = 0$ and $l = 1$, and between $l = 1$ and $l = 2$. 
In order for the comparison multiples method to be feasible, the comparison multiples must be constructed strictly of powers of two of d. This is due to the fact that powers of two can be implemented via simple shifting of the d value. If the comparison multiples cannot be constructed of powers of two, the derivation of the multiples becomes too costly for hardware implementation. For this reason the comparison multiples are chosen as \( M_1 = 0.5 \cdot d \) and \( M_2 = 1.5 \cdot d \). \( M_2 \) can be defined as \( M_2 = d + 0.5d \) or \( M_2 = 2d - 0.5d \). These cases must be analyzed to ensure that they conform to the inequalities described previously.

Firstly, the binary (for d) and BSD (for the residuals) truncation conditions must be described.

\[
0.5d - 2^{-5} < \{0.5d\}_s \leq 0.5d \\
d - 2^{-5} < \{d\}_s \leq d \\
2d - 2^{-5} < \{2d\}_s \leq 2d \\
4P_j - 2^{-5} < \{4P_j\}_s < 4P_j + 2^{-5}
\]

**\( M_1 = 0.5d \)**

It can be shown that \( M_1 = 0.5d \) conforms to the inequalities by analyzing the two cases that \( M_1 \) borders: \( q_{j+1} = 0 \) and \( q_{j+1} = 1 \).

For \( q_{j+1} = 0 \):

\[
\min(\{rP_j\}_s) = 4P_j - 2^{-5} < \{4P_j\}_s < \{M_1\}_s < \frac{1}{2}d = \max(\{M_1\}_s)
\]

Since \( q_{j+1} = 0 \), \( P_{j+1} = rP_j \), and the previous equation can be rewritten as
\[ P_{j+1} = 4P_j < \frac{1}{2} d + 2^{-5}. \]

When \( P_{j+1} \) is bounded for convergence as \( -pd \leq P_{j+1} < pd \) \( d \) can be derived as

\[ d \geq \frac{3}{16}. \]

Since \( d \geq \frac{1}{2} \), this condition is always met, so \( M_1 = 0.5d \) can be used for \( q_{j+1} = 0 \).

For \( q_{j+1} = 1 \):

\[ \min(\{M_1\}_5) = \frac{1}{2} d - 2^{-5} < \{M_1\}_5 < \{4P_j\} < 4P_j + 2^5 = \max(\{rP_j\}_5) \]

Since \( q_{j+1} = 1 \), \( P_{j+1} = rP_j - d \), and the previous equation can be rewritten as

\[ \frac{1}{2} d - 2^{-4} < 4P_j - d = P_{j+1}. \]

When \( P_{j+1} \) is bounded for convergence, \( d \) can be derived as

\[ d \geq \frac{3}{8}. \]

Both \( q_{j+1} = 0 \) and \( q_{j+1} = 1 \) are always correct, so \( M_1 = \frac{1}{2} d \) is a valid comparison multiple.

\( M_2 = d + 0.5d \)

\( M_2 \) is the border between: \( q_{j+1} = 1 \) and \( q_{j+1} = 2 \).

For \( q_{j+1} = 1 \):

\[ \min(\{rP_j\}_5) = 4P_j - 2^{-5} < \{4P_j\}_5 < \{M_1\} < d + \frac{1}{2} d = \max(\{M_1\}_5) \]
Since \( q_{j+1} = 1 \), \( P_{j+1} = rP_j - d \), and the previous equation can be rewritten as

\[ P_{j+1} = 4P_j - d < \frac{1}{2} d + 2^{-5}. \]

When \( P_{j+1} \) is bounded for convergence, \( d \) can be derived as

\[ d \geq \frac{3}{16}. \]

For \( q_{j+1} = 2 \):

\[ \min(M_1_5) = d - 2^{-5} + \frac{1}{2} d - 2^{-5} < M_1_5 < M_1 < 4P_j < M_1_5 < \max(rP_j_5). \]

Since \( q_{j+1} = 2 \), \( P_{j+1} = rP_j - 2d \), and the previous equation can be rewritten as

\[ -\frac{1}{2} d - 2^{-4} - 2^{-5} < 4P_j - 2d = P_{j+1}. \]

When \( P_{j+1} \) is bounded for convergence, \( d \) can be derived as

\[ d \geq \frac{9}{16}. \]

As \( d \geq \frac{1}{2} \), \( q_{j+1} = 2 \) is not correct for all cases and \( M_2 = d + \frac{1}{2} d \) is not a valid comparison multiple.

\( M_2 = 2d - 0.5d \)

\( M_2 \) is the border between: \( q_{j+1} = 1 \) and \( q_{j+1} = 2 \).

For \( q_{j+1} = 1 \):

\[ \min(rP_j_5) = 4P_j - 2^{-5} < M_1 < 2d - \frac{1}{2} d = \max(M_1_5) \]
Since \( q_{j+1} = 1 \), \( P_{j+1} = rP_j - d \), and the previous equation can be rewritten as
\[
P_{j+1} = 4P_j - d < \frac{1}{2} d + 2^{-5}.
\]

When \( P_{j+1} \) is bounded for convergence, \( d \) can be derived as
\[
d \geq \frac{3}{16}.
\]

For \( q_{j+1} = 2 \):
\[
\min(\{M_1\}_5) = 2d - 2^{-5} - (\frac{1}{2} d - 2^{-5}) < \{M_1\}_5 < \{4P_j\} < 4P_j + 2^5 = \max(\{rP_j\}_5)
\]

Since \( q_{j+1} = 2 \), \( P_{j+1} = rP_j - 2d \), and the previous equation can be rewritten as
\[
-\frac{1}{2} d - 2^{-5} < 4P_j - 2d = P_{j+1}.
\]

When \( P_{j+1} \) is bounded for convergence, \( d \) can be derived as
\[
d \geq \frac{3}{16}.
\]

Since \( d \geq \frac{1}{2} \), both \( q_{j+1} = 1 \) and \( q_{j+1} = 2 \) are always correct, so \( M_2 = 2d - \frac{1}{2} d \) is a valid comparison multiple.

The implementation of the comparison multiples generator is rather straightforward. The multiples 0.5\( d \) and 2\( d \) are derived in zero time by shifting \( d \) using wires. A subtractor constructed from inverters and a Han-Carlson adder is used to subtract 0.5\( d \) from 2\( d \). Since the values for \( d \) will not change from iteration to iteration, the generation of the comparison multiples is a one time per division delay.
To compare the binary comparison multiples with the BSD residual $P_j$, a three input comparator using a carry-free BSD adder was constructed,

$$a^+ + b = 2s^+ - s^-.$$

where the BSDA is constructed using full-adders, with the A input being the positive residual vector, the B input being the negative residual vector, and the carry-ins being the XNOR of the comparison multiple and the sign of $q_{j+1}$, where the BSDA produces the equation

$$a^+_i - a^-_i + b_i = 2s^+_{i+1} - s^-_i.$$
Table 5.5: BSDA Outputs [31]

<table>
<thead>
<tr>
<th>( b_i )</th>
<th>( \overline{1} )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((s_{i+1}^+, s_{i}^-) = (0, 1))</td>
<td>((s_{i+1}^+, s_{i}^-) = (0, 0))</td>
<td>((s_{i+1}^+, s_{i}^-) = (1, 1))</td>
</tr>
<tr>
<td>1</td>
<td>((s_{i+1}^+, s_{i}^-) = (0, 0))</td>
<td>((s_{i+1}^+, s_{i}^-) = (1, 1))</td>
<td>((s_{i+1}^+, s_{i}^-) = (1, 0))</td>
</tr>
</tbody>
</table>

In this manner, the BSD residuals can be compared with the binary comparison multiples, without the necessity of propagating a carry-bit. As a result the comparison can be completed in less time than a typical binary subtractor,

![Figure 5.4: BSDA - Binary Signed Digit Carry-Free Adder](image)

so the comparator produces

\[
rP_j^+ - rP_j^- - M_j = rP_j - M_j = P_i^+ - P_i^- .
\]

The signs of the comparator outputs must then be determined so that they may be used to find the proper value of \( q_{j+1} \). To determine the sign of the comparator outputs, the inverse of the carry-out of \( P_i^+ - P_i^- \) was calculated. To perform the calculation, a Han-Carlson adder with the sum stage removed was used.
The final stage of the QDS circuit was the encoding of the 2-bit unsigned $q_{j+1}$ output using the signs of the two comparators and the sign of $q_{j+1}$, which were computed the previous stage. The QDS output was determined as seen in Table 5.6.

<table>
<thead>
<tr>
<th>$S_{M_1}$</th>
<th>$S_{M_2}$</th>
<th>$\text{Sign}(q_{j+1})$</th>
<th>$\text{Mag}(q_{j+1}) = q_1q_0$</th>
<th>$q_{j+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>2</td>
</tr>
</tbody>
</table>

Analyzing the table, it can be seen that the $q_{j+1}$ can be generated using simple XNOR gates.

Figure 5.6: QDS Output Encoding
Combining these components, the complete QDS circuit can be seen above. It consists of the comparison multiple generator to create the two comparison multiples. Two comparators compare the residual, shifted by the radix, with the comparison multiples. Two sign detectors then determine if the comparison multiple was larger or smaller than the shifted residual. The final outputs are determined from the sign of the QDS output – calculated the previous cycle, as described in the next section – and the output of the sign detectors.

### 5.3.2 Residual Computation

With the quotient digit selection function implemented, it is possible to implement the recurrence function $P_{j+1} = r \cdot P_j - d \cdot q_{j+1}$ that is at the heart of the SRT division. The implementation of $rP_j = 4P_j$ is a simple matter of appending two grounds to the end of the residuals. The two bit shift acts as a multiplication by four, and takes zero time.
Similarly, since the values of \( q_{j+1} \in \{-2, -1, 0, 1, 2\} \), which are all multiples of two, the values for \( d \cdot q_{j+1} \) can all be calculated by shifting \( d \). To obtain a value of \( 2d \), one ground is appended, and to obtain \( d \), no action is taken. If \( d \) is zero, \( P_{j+1} = r \cdot P_j - 0 \cdot q_{j+1} = r \cdot P_j \), so the \( d \) component can be ignored entirely.

Examining the recurrence equation reveals the familiar formula

\[
BSD - BIN = BSD^+ - BSD^- - BIN. 
\]

This is the exact same equation used for the QDS comparators and, in fact, the same comparator hardware – extended to support the bit lengths of shifted residual – can be used to calculate the possible residuals for the next cycle.

With the possible residuals for the following cycle computed for all \( q_{j+1} \) values, the comparator outputs must be reduced from 69 bits to 66, and the signs of each must be calculated. The sign detector is an extended version of the sign detector used for the QDS function. It simply uses a Han-Carlson tree to determine the carry-out of \( BSD^+ - BSD^- \) and inverts it.

The bit reduction utilizes the convergence condition – \( pd \leq P_{j+1} < pd \) – to perform the compression. Given that \( p = \frac{2}{3} \), \( \frac{1}{2} \leq d < 1 \), and \(-\frac{2}{3} < P_{j+1} < \frac{2}{3}\), this means that the integer portion of the comparator outputs can be removed without affecting the value of the residuals. Furthermore, [31] has demonstrated that the most significant two bits of the compressed residual can be defined as seen in Table 10.
Table 5.7: Compression Process for \( l \in \{0,1,2\} \)

<table>
<thead>
<tr>
<th>( P_i&lt;64:64&gt; )</th>
<th>( A.B.C.X.Y )</th>
<th>( P_i&lt;65:64&gt; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000.XY</td>
<td>.XY</td>
<td></td>
</tr>
<tr>
<td>001.0i</td>
<td>.11</td>
<td></td>
</tr>
<tr>
<td>001.1iX</td>
<td>.1X</td>
<td></td>
</tr>
<tr>
<td>001.0i1</td>
<td>.\Pi</td>
<td></td>
</tr>
<tr>
<td>001.1iX</td>
<td>.\Pi</td>
<td></td>
</tr>
<tr>
<td>011.0i</td>
<td>.11</td>
<td></td>
</tr>
<tr>
<td>011.1iX</td>
<td>.1X</td>
<td></td>
</tr>
<tr>
<td>011.0i1</td>
<td>.\Pi</td>
<td></td>
</tr>
<tr>
<td>011.1iX</td>
<td>.\Pi</td>
<td></td>
</tr>
<tr>
<td>111.0i</td>
<td>.11</td>
<td></td>
</tr>
<tr>
<td>111.1iX</td>
<td>.1X</td>
<td></td>
</tr>
<tr>
<td>111.0i1</td>
<td>.\Pi</td>
<td></td>
</tr>
<tr>
<td>111.1iX</td>
<td>.\Pi</td>
<td></td>
</tr>
</tbody>
</table>

The encoding of the 65\textsuperscript{th} and 66\textsuperscript{th} residual bits can be seen in the following figure. The lowest 64 bits of the residual remain the same.

![Diagram](image)

**Figure 5.8: Adjust Unit for Integer Divider**

With possible residuals calculated, they can be stored in registers for the following cycle. The \( q_j \) value can then be used to select the residual \( P_j \) and sign of \( q_{j+1} \) using multiplexers.
5.3.3 Quotient Conversion

The final component in the division algorithm is the conversion of the quotient from a BSD number to a binary number. The simple method of converting the numbers is to wait until all iterations of the division have been completed, and then to subtract the negative quotient array from the positive array. However, rounding in floating-point division requires the converted quotient, so using this method could necessitate an additional clock cycle. For this reason, an on-the-fly rounding was described by Ercegovac and Lang in [11] and expanded upon in [9] and [10]. Since the integer divider is derived from the floating-point divider, it also utilizes this on-the-fly rounding method.

This method utilizes two n+2 bit shift registers $Q$ and $Q_m$, where $Q_m$ is always equal to $Q-1$. Each clock the quotient bits are appended to shift registers as:

$$
Q_m[j + 1] = \begin{cases} 
(Q[j],(q_{j+1} -1)) & \text{if } q_{j+1} > 0 \\
(Qm[j],(r-|q_{j+1}| -1)) & \text{if } q_{j+1} \leq 0 
\end{cases}
$$

$$
Q[j + 1] = \begin{cases} 
(Q[j],(q_{j+1})) & \text{if } q_{j+1} \geq 0 \\
(Qm[j],(r-|q_{j+1}|)) & \text{if } q_{j+1} < 0 
\end{cases}
$$

Where (a,b) is a concatenation, $Q[0] = 0$ and $Q_m[0]$ is undefined. Due to the assumption that $\frac{1}{2} \leq d < 1$ and $\frac{1}{2} \leq x < 1$, $q_1$ is always positive so it is not necessary to initialize $Q_m$.

To automate the divider to detect when the division has finished, $Q[0]$ can be initialized to all 1’s except for the LSB, which should be 0. Using this method, the proper number of iterations can be detected when the 0 bit is shifted out of the Q register. At this point, Q register can be used to determine the quotient. If the MSB of Q is 1, the output is $Q(n+1:2)$, otherwise it is $Q(n:1)$. Once again, this normalization shows the algorithms roots in floating-point format.
5.3.4 Adaptation for Universal Integer Division

Unlike the mantissas used in the floating-point division algorithm, the number of bits following the leading one of the integer dividend and divider is not constant. For this reason, the operands must be conditioned before they are used in the SRT divider.

Firstly, as the divider handles both signed and unsigned division, a method for handling both must be implemented. The simplest method, and the method implemented here, is to take the two’s complement of the operand if the MSB is 1 and signed division is chosen. This results in all values being positive, so all values can be handled in the same manner.

To shift the operands so that they are 64-bits with the MSB = 1, the operands – or their two’s complements – are fed into leading-zeros detectors. This determines the number leading zeros of x (LZX) and d (LZD), and thus the number of bits that the operand must be shifted left. X and LZX, and d and LZD are then fed into barrel shifters,
and x and d are shifted so that there are no leading zeros. The shifted values are then put into registers, and used as the x and d inputs to the SRT divider on the following clock.

This shift of x and d has the effect of changing the number of iterations required to determine q. While q would be perfectly accurate if simply allowed to run for 33 clock cycles, this wastes valuable ALU cycles and reduces the throughput. To alter the algorithm so that the minimum number of cycles is used, Q[0] is first shifted left by LZX. This value is entered into a register, as it takes the same amount of time to determine as the x and d values. At the start of the next clock cycle, it is additionally shifted by $n - 2 - LZD$ to account for shift of d that occurred. Taking advantage of the fact that the first QDS output is not appended to shift registers until the beginning of the following cycle, this value is then inserted as the initial Q register value: Q[0].

![Figure 5.10: Operand Conditioning](image-url)
The leading zero detectors can also be utilized to detect when an input operand is zero. If either operand is 0, the division is completed. In the case of x being 0, 0 is outputted as the quotient. If d is 0, a divide-by-zero exception is triggered.

When the quotient has been produced by the SRT division circuit, provided that both operands are non-zero, the quotient can be modified accordingly. If either x or d was negative, and signed division is being used, then the two’s complement of the quotient is formed. Otherwise, the quotient remains unchanged. This value is the final quotient for the divider.

5.4 Summary

The universal integer divider described in this chapter is based upon the SRT digit-recurrence floating-point divider described in [31], and containing elements of variable latency and high-radix division. The FP algorithm requires that the inputs be normalized. Maintaining this condition requires an additional cycle of operand conditioning to take the two’s complement of negative numbers if signed division is selected, and to shift the operands so that the most significant bit is 1. This allows the divider to require a variable number of cycles to execute.

The divider uses a radix-4 quotient digit selection function by comparison multiples to reduce the maximum number of iterations by half. This reduces the maximum number of iterations to 34, including the conditioning cycle. The minimum number of cycles, is 0 if either operand is 0 or if d is 1.
Chapter 6  ALU Synthesis Results

Each of the arithmetic blocks described previously were implemented in the VHDL hardware descriptive language. The designs were constructed primarily of basic logic gates, multiplexers and flip-flops using a structural architecture format. Test benches were then designed and used to verify the functionality of the blocks using ModelSim. Upon successful verification of the VHDL implementation, each arithmetic block was constrained and synthesized using Synopsys Design Compile (2006.06). The designs were synthesized using a TSMC 0.13μm static CMOS process. The designs were synthesized with a focus on speed, with area and power trade-offs considered, but secondary. The delay, area, and power of each were recorded.

6.1 Adder Synthesis

As the integer adder is such a vital part of all of the arithmetic components discussed in this paper, special emphasis was placed on determining the trade-offs between the different types of adders. First, radix-2 and radix-4 implementations of the Kogge-Stone parallel prefix adder were synthesized to determine which radix produces faster circuits with the TSMC process. As can be seen in Table 10, the radix-2 implementation demonstrated faster speeds, presumably due to the fan-out requirements of the radix-4 adder.

With the radix of the adder determined, the Han-Carlson and Han-Carlson/Carry-select adders were synthesized. These were compared to each other, as well as the two Kogge-Stone implementations. It was observed that the Han-Carlson performed slightly better than the corresponding Kogge-Stone, with the hybrid adder outperforming the rest.
Finally, to obtain a baseline against which to compare the performances of the adders being tested, an adder implementation from the DesignWare library was synthesized under identical conditions. The DesignWare libraries consist of highly optimized VHDL and Verilog implementations of circuits created by the Synopsys team. While the components implemented in the library are not identical to those implemented in this paper, they can, at least, be synthesized using the same technology. As comparable circuits in IEEE papers have been implemented in different sizes, and with different technologies, the DW libraries provide the best opportunity to make a fair comparison of the implemented circuits against other implementations. The DW adder that was synthesized was a variation on the Brent-Kung, modified to improve performance, without a large increase in area. This is the fastest implementation offered by DesignWare – another carry-look ahead variant may be chosen, but it is both slower, and larger.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Area (gates)</th>
<th>Power (mW)</th>
<th>Delay (ns)</th>
<th>Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS-R2</td>
<td>Worst Case</td>
<td>3093</td>
<td>16.58</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>Best Case</td>
<td>34.11</td>
<td>0.44</td>
<td>2,273</td>
</tr>
<tr>
<td>KS-R4</td>
<td>Worst Case</td>
<td>3007</td>
<td>17.96</td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td>Best Case</td>
<td>39.26</td>
<td>0.52</td>
<td>1,923</td>
</tr>
<tr>
<td>HC</td>
<td>Worst Case</td>
<td>2406</td>
<td>16.23</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>Best Case</td>
<td>66.07</td>
<td>0.44</td>
<td>2,273</td>
</tr>
<tr>
<td>HC/CS</td>
<td>Worst Case</td>
<td>2564</td>
<td>18.38</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>Best Case</td>
<td>35.35</td>
<td>0.4</td>
<td>2,500</td>
</tr>
<tr>
<td>DW BK</td>
<td>Worst Case</td>
<td>2092</td>
<td>11.62</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>Best Case</td>
<td>24.54</td>
<td>0.52</td>
<td>1,923</td>
</tr>
</tbody>
</table>
As can be seen in the preceding table, the Han-Carlson/Carry-select hybrid is the fastest of five adders that were analyzed, though, as expected, the increased performance comes at the cost of an increase in size over the strictly PPA Han-Carlson implementation. The radix-4 Kogge-Stone performed the worst due to increased fan-out, but did offer a decrease in area over the radix-2 Kogge-Stone implementation. The DesignWare adder was the smallest and used the least power, but was also slower than the radix-2 adders that were implemented – as expected with a Brent-Kung solution.

As this design is focused primarily on speed, the Han-Carlson/Carry-select adder is the best solution. The schematic that was produced by Synopsys Design Vision can be seen in Figure 6.1.
6.2 Multiplier Synthesis

As with the adders, the DesignWare libraries provide the best opportunity to
gauge the effectiveness of the multiplier design. The DesignWare libraries offer several
Booth-Wallace multipliers, each requiring a different number of clock cycles – the
number of rows of flip-flops can be selected to determine the number of required cycles.
In this manner, the DW multiplier is extremely similar to the multiplier described
previously in its flexibility.

As the integer divider has a fixed iteration, it is the limiting factor in the ALU.
The number of stages in the multiplier can then be adjusted so that the fewest possible
stages are used, while keeping the cycle time less than that of the divider. The
synthesized universal Booth-Wallace multiplier can be seen in Figure 6.2.

Figure 6.2: Universal Booth-Wallace Multiplier from Synthesis
Using four stages – with registers after the Booth encoder, and each two stages of the Wallace tree – the custom Booth-Wallace multiplier synthesis produced an area of 95649 gates, at a power dissipation of 345 mW and a delay of 2.04 ns – a speed of 490 MHz – under worst case conditions. Using the same conditions and constraints, the DW implementation required an area 52,167 gates, a maximum power of 254.72 mW, and a delay of 2.03 ns. While the optimized DW multiplier utilized less power and area, the custom design operates at nearly the same speed.

6.3 Divider Synthesis

Unfortunately, the DesignWare libraries do not offer a clocked divider. As such, there is no good benchmark with which to compare the universal divider. The reported divider area was 20925 gates and the power was 49.64 mW. The delay was 2.72 ns, for a frequency of 368 MHz.
6.4 ALU Synthesis

The synthesis of the completed ALU yielded an area of 122,215 gates, a power of 384 mW, and a delay of 2.89 ns – a frequency of 346 MHz. As expected, the critical path of the ALU is in the divider – from the output of the partial residual registers, through the computation of the next partial residuals. The multiplier dominated the area.

As with most large designs, the schematic produced by Synopsys is sufficiently large and dense to be dominated by wiring - making it nearly impossible to see any of the logic. For this reason, the ALU synthesis results diagram includes an inset to show the locations of two of the arithmetic blocks. The inset includes four blocks – the universal multiplier (UM), universal divider (UD), left shifter (BL) and right shifter (BR). The adder/subtractor and logic blocks are locate below the subset.

Figure 6.4: ALU from Synthesis
While the implementation of the XOR, OR, AND, and inverse logic functions are trivial, the implementation of the shifting operations are more complex. They required the used of a series of multiplexers to implement. The multiplexers use the shift control to determine whether to output a portion of the input array, or a constant. Using several multi-bit multiplexers, a barrel shifter is formed that can shift an n-bit number up to n bits in roughly the same amount of time that it takes to perform an n-bit addition – a one clock cycle operation. A Synopsys schematic of the left barrel shifter is included below.

Figure 6.5: Barrel Shift Left from Synthesis
Part II

Floating-Point Unit

This section presents the implementation of an IEEE compliant double-precision floating-point unit based on the MIPS 64 instruction set. The floating-point unit supports double precision addition, subtraction, multiplication and division, and conforms to the IEEE 754 standard for number representation and rounding.
Chapter 7  Floating-Point Unit Overview

Floating-Point Units (FPU) are the hardware components that handle decimal mathematical operations in the CPU. Like the ALU, the FPU implements the four basic mathematical operations – addition, subtraction, multiplication, and division – the difference being the number representation scheme utilized. An ALU handles integer values, represented in binary numbers. This means that the entire 64-bits of the bit vector represent the portion of a number to the left of the decimal point. An FPU deals with both the integer and fraction portions of numbers. As there is no way to slide a decimal point into the bit vectors to tell the computer what is the integer and what is the fraction, the operands must be divided into sections representing the sign, exponent, and mantissa of the number.

7.1 The IEEE 754 Standard

At the dawn of the computing age, chaos ruled the floating-point world. That is to say, there was no standard way of implementing a floating-point operation. Each platform implemented their own floating-point rules, with different ranges and precisions - different mantissa and exponent lengths, and methods of implementing exponent biases. As a result, programs written for one platform could not be easily ported to another platform.

In 1977, the companies of Silicon Valley formed a committee to rectify this problem under the banner of IEEE p754 [42]. Among the represented companies were Intel, National Semiconductor, Zilog, Motorola, IBM, DEC, CDC and Cray. After years
of debate and compromise – with many companies trying to persuade the committee to use their standards so that they did not have to change implement new formats – the IEEE 754-1985 [21] was adopted.

The IEEE 754 standard implemented a floating-point algorithm more complex than any to that day. While most companies implemented one type of rounding, the IEEE standard supports four modes: round to nearest (RN), round to zero (RZ), round to infinity (RP), and round to negative infinity (RN). The standard supports single-precision 32-bit numbers, and double-precision 64-bit numbers. As would be expected, double-precision offers a larger range (11 exponent bits compared to 8) and greater accuracy (52 fraction bits compared to 23) than the single-precision. As they operate in the same manner and the focus of the work presented in this paper is on 64-bit inputs, only the double precision format will be explored.

![Figure 7.1: Double-Precision IEEE 754 Format](image)

The decimal value of normalized FP numbers in IEEE 754 format is represented as \(-1^{\text{sign}} \times 2^e \times 1.f\). A complete list of the possible values for double precision numbers can be seen in the following table. The difference between single and double precision format lies in the exponent bias. For double precision it is 1023, while the single precision bias is 127. The bias allows for values both with and without integer digits.
### Table 7.1: Double Precision IEEE 754 FP Values

<table>
<thead>
<tr>
<th></th>
<th>Exponent (e)</th>
<th>Fraction (f)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>000000000000</td>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td>Denormalized</td>
<td>000000000000</td>
<td>nonzero</td>
<td>(-1^{sign} \times 2^{-bias+1} \times 0.f)</td>
</tr>
<tr>
<td>Normalized</td>
<td>0 &lt; e &lt; 2047</td>
<td>any</td>
<td>(-1^{sign} \times 2^{-bias} \times 1.f)</td>
</tr>
<tr>
<td>Infinity</td>
<td>111111111111</td>
<td>zero</td>
<td>Infinity</td>
</tr>
<tr>
<td>NAN</td>
<td>111111111111</td>
<td>nonzero</td>
<td>Not a number</td>
</tr>
</tbody>
</table>

IEEE rounding supports four modes that can be compressed into three rounding modes dependent upon the sign of the result. The round to nearest mode (RN) rounds the infinitely precise result to the nearest ULP. In the case where the infinite precision is exactly midway between the possible values, the result is rounded to the value with the ULP of 0. The round to positive infinity mode (RP) rounds the infinite precision number to the smallest value greater than or equal to itself. The round to negative infinity mode (RM) rounds the infinite precision number to the largest value less than or equal to itself. The round to zero (RZ) mode rounds the infinite precision value to the smallest value greater than or equal to itself if the number is negative, or the largest value less than or equal to itself if the number is positive.

Analyzing these modes, it can be seen that the rounding modes possess a certain amount of redundancy. As such, the IEEE rounding can be implemented without the RM and RP modes, as suggested by Quach, et al in 1991 [36]. These modes are replaced by a round to infinity mode that rounds away from zero, regardless of the sign of the number.

### Table 7.2: Possible Implementation of IEEE Rounding Modes [37]

<table>
<thead>
<tr>
<th>IEEE Rounding Mode</th>
<th>Positive Number</th>
<th>Negative Number</th>
<th>Treated As</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN</td>
<td></td>
<td></td>
<td>RN</td>
</tr>
<tr>
<td>RP</td>
<td>RI</td>
<td>RZ</td>
<td></td>
</tr>
<tr>
<td>RM</td>
<td>RZ</td>
<td>RI</td>
<td></td>
</tr>
<tr>
<td>RZ</td>
<td></td>
<td>RZ</td>
<td></td>
</tr>
</tbody>
</table>
7.2 FPU Instructions

The instructions implemented for the floating-point unit are add, subtract, multiply and divide. The floating-point addition, subtraction and multiplication operations support all four IEEE rounding modes. The FP division implements round to nearest rounding only, as it is common to sacrifice the remaining three rounding modes in order to decrease the division latency.

Table 7.3: FPU Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>OP Code</th>
<th>Rounding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>00</td>
<td>00</td>
<td>DP Addition with RZ Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>DP Addition with RP Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>DP Addition with RM Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>DP Addition with RN Rounding</td>
</tr>
<tr>
<td>Subtraction</td>
<td>01</td>
<td>00</td>
<td>DP Subtraction with RZ Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>DP Subtraction with RP Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>DP Subtraction with RM Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>DP Subtraction with RN Rounding</td>
</tr>
<tr>
<td>Multiplication</td>
<td>10</td>
<td>00</td>
<td>DP Multiplication with RZ Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01</td>
<td>DP Multiplication with RP Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>DP Multiplication with RM Rounding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>DP Multiplication with RN Rounding</td>
</tr>
<tr>
<td>Division</td>
<td>11</td>
<td>-</td>
<td>DP Division with RN Rounding</td>
</tr>
</tbody>
</table>
Chapter 8  Floating-Point Adder/Subtractor

Just as integer addition/subtraction is the most used ALU operation, FP addition/subtraction is the most utilized floating-point operation. Oberman and Flynn [34] report that FP adder is used for 55% of all floating-point instructions. Because of this, many methods of implementing floating-point division have been researched and implemented. Although the approaches can be implemented in vastly different manners in an attempt to optimize the circuitry, all methods implement some variation of the block diagram shown in Figure 8.1.

![Figure 8.1: Block Diagram of Floating-Point Adder/Subtractor](image)
The unpacking stage in this diagram involves separating the sign, exponent, and significant for each operand. This includes reinstating the hidden 1 for normalized number, and the hidden 0 for denormalized numbers. If a different number formatting is used in the adder, the number conversion occurs here as well.

The difference between the exponents is used to determine the amount of right shifting necessary to align the smaller operand with the larger operand. It is also utilized to determine which operand is larger.

To reduce the costs of additional logic, the alignment of the mantissas is limited to only one operand, requiring only one shift operand. This requires that it be permissible to swap the operands so that the one with the smallest magnitude may be fed into the shifter. To lessen the effects of the swap, it is implemented in the same block that performs selective two’s complement in preparation for effective subtraction.

After the addition/subtraction of the aligned significands, the values sum/difference has a magnitude in the range \([0,4)\). This result must be normalized to be in the range of \([1,2)\) to conform to the IEEE standard. If the result is in the range of \([2,4)\), it must be normalized by shifting one bit to the right. If the result is in the range of \([0,1)\), it must be shifted one bit to the left for normalization. The exponent must be adjusted in accordance to the normalization shift – an increase of 1 for a shift right, a decrease of 1 for a shift left.

After normalization, IEEE rounding is performed in accordance with the standard described in the previous chapter. If the normalized result is rounded upward, it has the potential to cause an overflow to 2, which must then be normalized with another shift right, and an increment in the exponent. This result can then converted to IEEE format.
8.1 Common FP Adder/Subtractor Optimizations

8.1.1 Use of Compound Adders

Compound adders compute \( a+b+cin \) and \( a+b+cin+1 \) in parallel. Using compound adders to add the aligned significands allows the rounding decision to be computed in parallel. The round decision is then used to select between the sum and sum+1. This replaces an adder with a multiplexer, reducing both delay and area.

8.1.2 Parallel Paths

The FP-adder pipeline can be partitioned into two parallel paths that work under different assumptions, as described by Farmwald [6]. Each path can be optimized for its specific purpose, and disregard some of the steps required for the alternative path. The most common method of portioning the paths is according to exponent difference. For small exponent differences (\( A_{\exp} - B_{\exp} = \{-1,0,1\} \)), a near path is defined. For larger exponent differences, a far path is defined.

8.1.3 One’s Complement Significand Negation

The one’s complement of a number is formed by inverting the bits of the significand, while conversion to the two’s complement format requires the use of inverters and an adder, as the two’s complement is equal to the one’s complement + ULP. The conversion of one’s complement to two’s complement format can be accounted for by adjusting the ULP of a subsequent calculation. For example, using a compound adder, the a+b+1 output accounts for the missing ULP of the two’s complement.
8.1.4 Leading Zero Approximation

For the near path, effective subtraction may result in a sum/difference that will require normalization via left shifting. The number of bits that the shift will require can be approximated by analyzing the number of leading zeros of a recoding of the mantissas. Since it requires only the mantissas, and not an exact value for the sum/difference, the approximate number of leading zeros can be derived in parallel to the addition/subtraction operation taking place via the compound adder. The exact number of leading zeros can then be selected simultaneously to the selection of the correct sum/difference using multiplexers. This eliminates several logic delays, allowing the correct normalization to be performed immediately up on the selection of the correct sum/difference. The exact process for approximating the leading zeros varies between FP adder/subtractor implementations.

8.1.5 Reduction of Rounding Modes

Using the compressed rounding, as described in the previous section, allows for simpler rounding implementation. There are many proposed rounding algorithms that improve upon the naïve implementation described previously [13, 36, 37, 48]. Those that offer the highest speeds tend to use this compression from four down to three rounding modes. The benefits of this process are not difficult to see. The compression logic can be implemented in parallel to the computation of the sum/difference, so it does not introduce additional delay into the system. The amount of rounding logic, and the logic used to select between rounding modes may then be reduced to accommodate for one less mode, reducing both area and delay.
8.2 The SE FP Adder/Subtractor

As there are so many variations on the floating-point adder/subtractor – each company has its own variation(s) – only the one implemented for this paper will be discussed. The FP adder/subtractor used for the FPU described in this paper is based on the Seidel-Even (SE) FP adder [41], but has been modified to allow for denormalized numbers, which were not supported the original. The formulae behind the SE adder are proven in [47].

The SE adder utilizes a two-path parallel path architecture, and takes two clock cycles to execute. The N path is similar to the near path described previously, assuming that \( A_{exp} - B_{exp} = \{-1,0,1\} \) and that effective subtraction is taking place. Effective subtraction means that the sign of the A operand and the output could possibly have differing signs depending upon the respective sizes of A and B:

\[
SEFF = sign(A) \oplus sign(B) \oplus is\_sub
\]

Where is_sub is 1 if subtraction is being performed and sign(A) and sign(B) are the sign bits of the A and B operands.

This parallel path method has the advantages described previously. The paths can be optimized for a single purpose, and the hardware can be smaller and faster. This partitioning of logic offers the additional benefit of only requiring one path – the N path – to perform subtraction. By limiting the near path to effective subtraction, the significand is guaranteed to be of smaller magnitude than the larger of the inputs. As such, the rounding operation can be removed from this path. The use of a single subtraction path is a method utilized by both AMD [33] and SUN [16] in recent processor implementations.
8.2.1 Near Path

To determine which of the operands is the larger and which is the smaller, the least significant two bits of the exponent of B are subtracted from the least significant two bits of the exponent of A. The two-bit output of this subtraction is used as the selection inputs for a group of multiplexers that determine the inputs for a compound adder. If the difference is “00”, the exponents are the same, so the inputs for the adder are the mantissa of A and the inverse of the mantissa of B. If the difference is “11”, the exponent of B is one larger than the exponent of A, so the mantissa of B is shifted by two, and used as an input along with the inverse of the mantissa of A. If “01” is the difference, the exponent of A is one larger, so the mantissa of A is shifted by two, and used as an input along with the inverse of the mantissa of B.
Figure 8.3: SE Adder Near Path
Table 8.1: Compound Adder Inputs Based on Exponents

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>FA + ~FB</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>1 1</td>
<td>2FB + ~FA</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
<td>~FB + FA</td>
</tr>
<tr>
<td>0 1</td>
<td>1 1</td>
<td>1 1</td>
<td>2FB + ~FB</td>
</tr>
<tr>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td>2FB + ~FA</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>2FB + ~FA</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td>FA + ~FB</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>0 1</td>
<td>2FB + ~FA</td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>1 0</td>
<td>~FB + FA</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>FA + ~FB</td>
</tr>
</tbody>
</table>

The case where the difference is “10” can be disregarded, as the difference of two violates the assumptions for the near path, so the far path will be used. Similarly, cases where the exponents of A and B differ at a larger bit position than those analyzed violate the near path conditions and fall into the far path. The shift by two of the larger operand if one exponent is one ULP larger than the other exponent acts to align the significands, ensuring that correct values are subtracted.

The compound adder used to perform the subtraction uses the lazy one’s complement, where the addition of the ULP required to convert the inverse of the lesser significand to two’s complement format is not added into the calculation (i.e. by carry in). Instead, it is accounted for by the structure of the compound adder itself. This adder is specially designed to accommodate for the fact that the operands should be switched if EA and EB are equivalent, and FB is greater than FA. This combined with the selection
and alignment logic described previously ensures that the operation $|A| - |B|$ is performed if $A > B$, and $|B| - |A|$ is performed if $B > A$.

Rather than merely producing the carries, as done for the integer adder described in Chapter 2, the propagation groups, and the original propagation signals from the prefix logic are also utilized in this compound adder. The carries and propagation groups produced by the Han-Carlson tree, can be OR’d together to form an incremented carry-group. The incremented carry-group can then be XOR’d together with propagation signals from the prefix logic to create the two’s complement subtraction $|A| + |B| + 1$. An XNOR of the propagation signals with the carries produces the inverse of the addition of the inputs: $|A| + |B|$. The resulting output of the compound adder is defined as:

$$\text{abs}(|A| - |B|) = \begin{cases} 
|A| + |B| + 1 & \text{if } |A| - |B| > 0 \\
|A| + |B| & \text{if } |A| - |B| \leq 0
\end{cases}$$

Using this method for the compound adder allows both $|A| - |B|$ and $|B| - |A|$ to be computed simultaneously. Using the most significant bit of the $|B| - |A| = |A| + |B|$ output, the correct orientation can be determined. If that value is 1, then $|B| > |A|$, so $|B| - |A|$ must be selected. Otherwise, $|A| - |B|$ is selected.

The number of leading zeros of the differences are predicted in parallel with the compound subtractions. As explained previously, this allows for an immediate normalization upon the selection of the proper subtraction orientation. To perform the leading zeros prediction, the subtraction is approximated using PN-recoding, as described in [10], [11]. This produces sum and carry outputs, which are XOR’d together. The
output of the XOR is then OR’d with the exponent of B, decoded into one-hot format, with an exponent value of 0 decoded into a 64-bit value of 1. This limits the number of leading zeros so that denormalized numbers may be included in the calculations. These values are then fed into priority encoders such that one priority encoder (PENC55 in the diagram) takes an additional 0, creating an output of 1 greater than the other priority encoder. If $|B| > |A|$, then the output of PENC55 is used as the leading zeros count. The output of PENC54 is used otherwise.

To implement the PN-recoding, two levels of half-adders are used. The holes created by the half-adders – carry[LSB] and sum[MSB] – are filled with 0’s to align the outputs, as the MSB of the carry should be two times greater than the MSB of the sum.

![PN Recoding Circuit](image)

*Figure 8.4: PN Recoding Circuit*
The priority encoder (PENC) counts the number of leading ones in a binary string, and expresses that number in binary format. In order to do this, it is constructed of a simpler type of priority encoder that uses a unary format. The unary output is then converted into one-hot format, and subsequently encoded into binary format.

The n-bit binary input is encoded into an n-bit unary output in accordance with the following rule, with the input $x[0:n-1]$ and the output $y[0:n-1]$.

$$y[i] = OR(x[0:i])$$

To reduce the area of the unary priority encoder (UPENC), an $n/2$ input unary with a level of OR gates before and after the UPENC to incorporate the remaining inputs.
The difference logic is a simple array of XOR gates that compares each bit to the previous bit. In this manner, the location of the change from 0 to 1 is detected, and the unary format is converted to one-hot. The one-hot format is then padded with 0’s so that it becomes a power of two in length, and converted into binary format by way of an encoder. A zero detector is utilized to ensure that the input is nonzero, but this is unnecessary with the addition of the decoder and OR logic added for the denormalized numbers.

As both the UPENC and encoder circuit grow non-linearly in area, the size of the binary priority encoder (BPENC) can be reduced through the use of a divide and conquer implementation. The recursive n-input BPENC uses two \( n/2 \) input BPENC circuits as shown in the following figure.

**Figure 8.6: Area Optimized Unary Priority Encoder**
With both the leading zeros and the mantissa difference calculated, the difference can be normalized by way of left shifting to remove the leading zeros created by the subtraction. The difference is shifted left by the number of leading zeros predicted through the approximation. Final normalization is then adapted using a multiplexer with the most significant bit (bit 53) of the shifted difference used for the selection logic. If the MSB is 1, then the output is bits 53 down to 1 of the shifted difference, otherwise it is 52 down to 0.

8.2.2 Far Path – First Cycle

The far path encompasses all cases not covered by the near path. That is to say, all effective addition operations and subtraction operations where the exponents differ by more than a ULP utilize the far path. The far path is divided into two parts – one for each clock cycle.
The first cycle of the far path determines which operand is the larger, and shifts the smaller accordingly to accommodate the difference in magnitude. It also converts the four IEEE rounding modes into the compressed three rounding mode format utilized by the rounding algorithm and determines if the exponent difference is sufficiently large to require the far path.

The difference in magnitude is determined using a 1’s complement subtraction of EB from EA. To accommodate for the use of 1’s complement instead of 2’s complement format, the value of the effective subtraction bit is subtracted from the larger of the exponents. Which exponent is larger is determined by the most significant bit of the exponent subtraction. If it is 1, the magnitude of B is larger than that of A.
The significands are chosen through a series of shifts and multiplexers such that

\[ \text{flp} = \text{fl} \cdot 2^{\text{seff}} \]

\[ \text{fsopa} = \begin{cases} \text{fs} \cdot 2^{-\delta \text{lim}} & \text{if } \text{seff} = 0 \\ - \text{fs} \cdot 2^{-\delta \text{lim}+1} & \text{otherwise} \end{cases} \]

The exponent subtraction result \( \delta \) can be limited such that a shift need not exceed the next power of two larger than the number of bits of the mantissa. This is true due to the fact that a difference of greater than 53 in the exponent will result in a shift of the smaller mantissa such that the smaller mantissa has no effect upon the larger mantissa when they are added together.

\[ \delta \text{lim} = \min\{\delta, 2^q\} \]

\[ q \geq \log_2 p + 2 = \log_2 55 \rightarrow q = 6 \]

As a result, only the least significant six bits of the difference, XOR’d with the seventh bit need to be analyzed to determine a shift in the smaller operand. The seventh bit of \( \delta \) is used to determine whether A or B is larger in magnitude when the exponent difference is limited to the least significant six bits. A 0 at \( \delta[6] \) shows that the exponent of A is larger than that of B. A 1 at this position shows that the exponent of B is larger than that of A. The inversion if B is larger allows the lower six bits to be used as the shift regardless of which operand is larger.

The most significant bit of \( \delta \) is also XOR’d with \( \delta[11:6] \) and then OR’d together to determine if the \( \delta \text{lim} \) condition has been met. If so, the most significant 65 bits of are padded with the effective subtraction bit, which is a faster operation when performed directly, than the variable length shifting performed otherwise. This is_big condition can
be OR’d the shift bits to determine the size of \( \delta \), and produce a component of the path selection algorithm.

\[
is_{-r1} = (|\delta| \geq 2)
\]

The rounding compression reduces the four standard IEEE rounding modes into the three effective modes described previously. The compression requires only AND gates, inverters and a single OR gate. As it is produced concurrently with the significand conditioning, compression adds no additional delays to the circuit. The sign used to determine the sign of the output is selected using \( \delta[11] \), as done for the large and small mantissa values.

<table>
<thead>
<tr>
<th>IEEE Rounding Mode</th>
<th>Reduced Rounding Mode – RNRI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Positive Number</td>
</tr>
<tr>
<td>RZ</td>
<td>00</td>
</tr>
<tr>
<td>RP</td>
<td>01</td>
</tr>
<tr>
<td>RM</td>
<td>10</td>
</tr>
<tr>
<td>RN</td>
<td>11</td>
</tr>
</tbody>
</table>

### 8.2.3 Far Path – Second Cycle

The second cycle of the far path adds the mantissas, rounds the result, and determines if an overflow exception has occurred. By implementing the rounding in parallel with the compound adder used to create the sum of the mantissas, the latency of the adder is significantly reduced over the naïve rounding method.
To accommodate the desire to perform the rounding in parallel with the mantissa addition, the mantissa of the larger operand (FLP) and the most significant 54 bits of the smaller operand's mantissa are run through a half-adder before the compound adder. This compression creates the "carry" input necessary for the rounding algorithm as the least significant bit of the sum output. The half-adder also has the benefit of reducing the compound adder inputs from 54 bits to 53 bits, requiring a few gates less hardware to implement. No carry is generated to bit 54, as the sum will be less than four.
The compound adder uses the same format as the compound adder in the near path. The only difference being that the XNORs are replace with XORs so that the far path adder forms $A+B$ and $A+B+1$ rather than $\overline{A}+\overline{B}$ and $A+B+1$ as formed in the near path. The use of the compound adder here produces both possible mantissas. The larger of which is used if the rounding requires that non-LSB bits of the mantissa be changed. Each of the compound adder outputs is normalized using its most significant bit to control a one bit shift operation.

Naïve rounding implementations perform the addition/subtraction of the shifted mantissa values to get a sum/difference value of $\sigma$. Three values are analyzed for rounding. The carry bit “$C$” is the at the bit position of the least significant bit of FLP. The round bit “$R$” is the most significant bit not affected by FLP. The sticky bit “$S$” represents all bits of $\sigma$. Due to the fact that the smaller operand may have been shifted by up to 65 bits, while the larger operand requires only a one bit shift, it can be observed that the least significant 64 bits of $\sigma$ smaller than $C$. To acquire the sticky bit, the least significant 63 bits of $\sigma$ are OR’d together. The rounding bit is equal to $\sigma[63]$. The carry bit is equal to $\sigma[64]$.

![Figure 8.10: CRS Generation in Naïve FP Addition/Subtraction Rounding](image-url)
The rounding algorithm used in the SE Adder is a variation of rounding by injection [13], where an injection value is added to the FLP and FSOPA addition. The injection is defined as follows, where the $2^{-52}$ bit position occurs at FLP[0] and FSOPA[64], and $2^{-53}$ bit position occurs at FSOPA[63]. This results in the injection being added to the lower part of the FSOPA – directly influencing the R and S values respectively.

\[
INJ = \begin{cases} 
0 & \text{if } RZ \\
2^{-53} & \text{if } RN \\
2^{-52} - ULP & \text{if } RI 
\end{cases}
\]

However, due to the fact that the R and S bits rely only on the smaller operand FSOPA, the injections can be added directly to the R and S bits. This eliminates the need to extend the injection for RI all the way to the ULP, reducing a 64-bit adder to a 2-bit adder. The resulting 2-bit injection can be derived from the RN and RI three mode rounding using the a simple XOR gate of RN and RI for INJ[1], and using RI as INJ[0].

<table>
<thead>
<tr>
<th>RN</th>
<th>RI</th>
<th>INJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

The addition of the injection is fairly straightforward when effective addition is being implemented. As FSOPA is being added to FLP, there is no two’s complement to account for, so R can be read directly from FSOPA[63], and S can be derived from an OR tree of the lowest 63 bits of FSOPA. The injection, truncated to two bits, can then be added to R and S to create a new R and S, and a carry-out.
Since the full addition of the FLP and FSOPA is not required for the rounding, as only the LSB of the FLP is being used, the S bit must be modified to account for the fact that addition and subtraction would affect the FSOPA differently. While the FSOPA is already negated if effective subtraction is being performed, the increment of the ULP required for 2’s complement subtraction is not implemented. If the lower part of the FSOPA is all ones, the addition of the ULP results in an overflow that must be accounted for. To accommodate this fact, the least significant 63 bits of the FSOPA are XOR’d with the effective subtraction bit. This has the effect of inverting the least significant 63 bits of FSOPA if effective subtraction is performed, and leaving them alone otherwise. As a result, the S bit can be used to denote both itself, and the carry into the R position – represented as the inverse of S. If the lower part of FSOPA is all ones an increment would result in an overflow. This is shown in the hardware as the inversion of all ones results in an S bit of 0. The inverse of S then shows the carry-in of the R position as 1. If the lower part of FSOPA is not all ones, no overflow results from an increment, so the XOR results in an S bit of 1 and no carry-in into the R position is required.
Figure 8.12: CRS Creation for Effective Subtraction

Since the derivation of S requires an array of XOR gates followed by an OR tree, it takes significantly longer to generate than C, R or the injection. For this reason, two sets of CRS outputs are created – one each for the cases when S is 0 and when S is 1. The generated value for S is then used to select which CRS to use for the rounding implementation. In this representation, C denotes a carry-in into the \( \sigma[0] \) bit position.

Figure 8.13: CRS Circuit
The CRS circuit is derived from the following truth tables, which were in turn derived from an analysis of CRS creation diagrams for effective subtraction and addition displayed previously. The tables are divided into the cases where \( S = 0 \) and \( S = 1 \), to accommodate the division of the circuitry. The cases where \( \text{INJ} \) is "00" were omitted from the truth tables as it should never occur. The accompanying optimized CRS equations were derived using Karnaugh maps.

Table 8.4: Truth Tables for CRS Generation

<table>
<thead>
<tr>
<th>( R' )</th>
<th>SEFF</th>
<th>INJ</th>
<th>CRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

\[
C = (\text{Seff} \times R') + (\text{Seff} \times \text{INJ}(1)) + (\text{INJ}(1) \times R')
\]

\[
R = R' \text{xor} \text{Seff} \text{xor} \text{INJ}(1)
\]

\[
S = \text{INJ}(0)
\]

<table>
<thead>
<tr>
<th>( R' )</th>
<th>SEFF</th>
<th>INJ</th>
<th>CRS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>011</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00</td>
<td>011</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

\[
C = (\text{INJ}(1) \times \text{INJ}(0)) + (\text{INJ}(1) \times R')
\]

\[
R = R' \text{xor} \text{INJ}(1) \text{xor} \text{INJ}(0)
\]

\[
S = \text{not}(\text{INJ}(0))
\]
The least significant bit of the half-adder sum described previously accounts for the remaining portion of the carry bit. When it is XOR’d with the C bit from the CRS, the actual carry bit (σ[64]) is created. This, combined with the R, S, and RN bits, is used to define the rounding modifiers OVF and NOVF, used for overflows and non-overflows of the compound addition of FLP and FSOPA.

When there is no overflow in the compound addition, the least significant bit of the mantissa formed by the XORing of the C and least significant bit of the sum output of the half-adder array is valid for all cases except for one. The round to nearest algorithm implemented rounds upward if the result is exactly halfway between two representable numbers. The IEEE round to nearest algorithm rounds to the nearest even number in this case, always pulling the ULP down. To accommodate this difference, the halfway condition must be detected and used to create a mask to alter the output LSB. Without the injection, the halfway condition occurs when R is 1 and S is 0. With the injection for RN, the halfway condition occurs when R is 0 and S is 0. As the mask only affects the output when the halfway condition is met and RN rounding is used, the no overflow mask is defined as

\[ NOVF = R + S + \overline{RN} . \]

This produces a 0 if the conditions are met and a 1 otherwise. When NOVF and the output of \( XSUM[0] \oplus C \) are AND’d together, they produce the correct LSB for the case when no overflow has occurred.

Similarly, the case when an overflow occurs must also be altered for round to nearest halfway condition. Without the injection, the halfway condition for this case comes when \( XSUM[0] \oplus C \) is 1, R is 0, and S is 0. With the injection, the halfway
condition occurs when \( XSUM[0] \oplus C \) is 1, \( R \) is 1 and \( S \) is 0. The mask can then be created as

\[
OVF = (XSUM[0] \oplus C) + \overline{R} + S + \overline{RN}.
\]

This mask must then be applied to the LSBs of both outputs of the compound adder in order to provide the correctly rounded values of each. The most significant bits of the compound adder outputs are then used to select from the overflow and non-overflow least significant bits.

\[\text{Figure 8.14: Injection Based Rounding}\]

To choose between the straight sum and incremented sums of the compound adder, as well as the corresponding rounded least significant bits, an increment circuit is implemented along with the rounding circuit. The increment circuit is divided into two
paths – one each for the overflow and no overflow conditions. The overflow is equal to most significant bit of the non-incremented compound adder output (FOPSUM[52]). For the non-overflow case, when FOPSUM[52] is 0 or the round to zero rounding mode is selected, the FOPSUM[1] bit needs to be incremented only if both the C and XSum[0] are 1.

For the case when FOPSUM[52] is 1 and round to infinity or round to nearest is selected, the path needs to be further divided according to the rounding mode. If the round to infinity mode is selected, the increment takes place if the value is greater than a truncation to 53 bits, so the increment occurs if C or XSum[0] is 1. If round to nearest is selected, the increment will only take place if the value is greater than the halfway condition. This occurs when $R + C + XSum[0] \geq 2$, with XSum[0] and C each independently representing the halfway condition.

\[
INC = \begin{cases} 
  C \cdot XSum[0] & \text{if } \overline{FOPSUM[52]} \text{ or } RZ \\
  C + XSum[0] & \text{if } FOPSUM[52] \text{ and } RI \\
  R \cdot C + C \cdot XSum[0] + R \cdot XSum[0] & \text{if } FOPSUM[52] \text{ and } RN 
\end{cases}
\]

### 8.3 Summary

In this chapter, a floating-point adder/subtractor based on that of Seidel and Even was presented. The design uses the two-path architecture, dividing the operations by exponent difference and effective operation. Subtraction is only performed if the exponent difference is 0 or 1, and effective subtraction is performed – the “near” path. Rounding by injection is only performed on the “far” path. The original design was altered to accommodate denormalized numbers, and to fix an error in the SE paper that resulted in an exponent difference of 2 being in the “near” path.
Chapter 9  Floating-Point Multiplication

Floating-point multiplication has nearly as many far-reaching applications as floating-point addition/subtraction. Oberman and Flynn [34] report that floating-point multiplication accounts for 37% of all floating-point operations. As a result, it is important to implement an efficient multiplier design. The major variation in floating-point multiplier design stems from the method used of implementing the unsigned multiplier, and the corresponding rounding.

![Generic FP Multiplier Block Diagram](image)

Figure 9.1: Generic FP Multiplier Block Diagram
It is important to note that the exponent biases must be accounted for during the exponent addition. As each exponent includes a bias, straight addition would result in a product exponent offset by an additional bias.

\[(A_{Exp} + bias) + (B_{Exp} + bias) = (P_{Exp} + bias) + bias\]

This also makes the prospects of an exponent overflow exception much more likely. A simple subtraction of one bias value from either exponent input, or the resulting exponent will resolve this issue.

9.1 FP Multiplier using Booth-Wallace

The multiplier can be implemented using either the shift and add sequential or the parallel implementations as described in Chapter 4. While the shift and add implementation requires less hardware, reducing area, it requires more clock cycles, and thus increases latency. For that reason, the Booth-Wallace architecture is once again utilized in this multiplier design.

Structurally, the Booth-Wallace multipliers implemented for the floating-point and integer dividers are very similar. The major difference stems from the fact that the floating-point multiplier does not need to support both signed and unsigned multiplication. The multiplier used in the floating-point needs to support only unsigned numbers, as the mantissas represent 53-bit unsigned number, despite the fact that the mantissa of each normalized number begins with a leading one.

As the mantissas with the hidden bit added are 53-bit, two bits must be added to B to prepare it for use in the Booth encoder. A 0 is appended as the LSB so that the initial partial product is not mistakenly encoded as belonging to the middle of a string of 1s. For
B to be divided into groups of three, it must have an odd number of bits, so the 54-bit extended B must be padded with an additional 0. To ensure that the initial value of A is represented in unsigned format, A is also padded with a 0.

\[
A_{\text{ext}} = 0 \& A \\
B_{\text{ext}} = 0 \& B \& 0
\]

The radix-4 modified Booth encoding is exactly the same as that used in the 64-bit integer multiplier, the functionality of which may be seen in Table 3. The Booth encoding results in 27 partial products, resulting in an asymmetric Wallace tree. The 27-input Wallace tree consists of two less compressor blocks than the symmetrical 32-input tree, while maintaining the same four-level design.

Figure 9.2: 53-Bit Booth-Wallace Tree for FP Multiplication
The 128-bit Han-Carlson adder that performed the final summation of the Wallace tree outputs in the integer multiplier is replaced with a compound adder and rounding logic. This allows for both the incremented and straight sums that are the possible results of the rounding and the rounded LSB to be computed simultaneously, in much the same way as done in the floating-point adder.

9.2 Rounding by Injection

While any number of rounding methods can be employed for the multiplier, including those mentioned previously [13, 36, 37, 48], the rounding by injection method [13] introduced for the floating-point adder/subtractor is also utilized for the floating-point multiplier. However, as the algorithm was developed for multiplicative rounding, it lends itself to be both much more integrated and elegant in the multiplier instantiation.

The multiplicative rounding of the outputs of a tree structure differs from the additive rounding of the shifted mantissas in two simple ways. Firstly, as the addition algorithm allows for a 6-bit shift control, the smaller operand may be 118 bits in length, while the maximum bit size of the output of the partial products tree is twice that of the inputs, or 106 bits – 108 bits including the leading zeros that result from the unsigned buffering of A. This accounts for the fact that both operands are normalized in the range of [1,2), so can never attain a value of 4. The second difference, which has more of an impact on the rounding algorithm, is that both operands can exceed the 53-bit mantissa size, and not just one as with the addition algorithm. For addition, the larger operand is shifted a maximum of one bit, so the lower bits are zeroed, allowing for simpler handling of the rounding algorithm that creates the carry, round and sticky bits. This is not true for
the multiplication algorithm, where the Wallace tree produces a carry and a sum, both of which are 106 bits in length. This means that the lower bits must also be added together, a step which was unnecessary in the additive rounding algorithm. The carry, round, and sticky bits for the multiplier rounding scheme are defined the same as they were for the addition/subtraction rounding scheme.

![Figure 9.3: CRS Generation in Naive FP Multiplication Rounding](image)

The injections used in floating-point multiplication are defined the same as those used in the floating-point adder:

\[
INJ = \begin{cases} 
0 & \text{if } RZ \\
2^{-53} & \text{if } RN \\
2^{-52} - ULP & \text{if } RI 
\end{cases}
\]

The primary difference between the two methods is that the injection cannot be reduced to a two bit number due to the fact that both Wallace tree outputs have the same number of bits, and both affect the CRS. The Wallace tree structure still does allow for a different injection optimization though. Examining the Wallace tree structure for the 53-bit multiplier, it can be observed that there is one input that is unaccounted for. In the previous diagram, it was zeroed so as not disrupt the addition of the partial products, but
it is possible to insert the injection value here. By injecting it directly into the Wallace
tree, no additional addition hardware is required, and the sum and carry outputs already
include injection. The injection is selected from the RNRI three mode rounding
compression, using multiplexers.

As the injection is already included in the Wallace tree outputs, the computation
of the CRS bits is derived by a simple addition of the lower 52 bits of the Wallace tree
outputs, which produces the C as the carry out, and the R as the MSB. The remaining 51
outputs bits are OR’d together to create S.

Apart from the CRS generation, the multiplicative and additive rounding exactly
are the same. A compound adder is used to create the sum and the incremented sum after
a row of half adders finds sum at the C position without the carry from the previous bit
(XSum[0]), while the possible least significant bits are formed by the rounding block.
The rounding block also determines if the incremented sum or straight sum is used for the
product using the C, R and XSum[0] bits. The increment decision logic is the same as
that used for the floating-point adder/subtractor.

\[
INC = \begin{cases} 
C \cdot XSum[0] & \text{if } FOPSUM[52] \text{ or RZ} \\
C + XSum[0] & \text{if } FOPSUM[52] \text{ and RI} \\
R \cdot C + C \cdot XSum[0] + R \cdot XSum[0] & \text{if } FOPSUM[52] \text{ and RN} 
\end{cases}
\]
9.3 Exponent Calculation and Overflow Exception

As mentioned previously, the unbiased exponent of the product is equal to the sum of the unbiased exponents of the operands. The result may have to be incremented if the product of the mantissas produces a value greater than or equal to two, which must subsequently be normalized into the range of [1,2). The addition of the exponents has the possibility of creating an overflow condition, where the carry-out of the addition is equal to 1. This is means that the product of the operands cannot be expressed in double-precision format, and an exception flag must be thrown.
To account for the additional bias, and to detect the overflow condition, the bias is removed from the A operand using an 11-bit adder. The two’s complement of the bias – “10000000001” – is added to A. The sum is equal to the exponent with the bias removed, and the carry-out denotes whether the two’s complement of the value must be taken. If the carry-out is 1, the exponent minus the bias is positive (or zero), and the sum denotes the difference. If the carry-out is 0, the exponent minus the bias is negative, so the exponent is negative, and the absolute value of the exponent can be found by taking the two’s complement of the sum.

The exponent of A, with the bias removed, is then added to the exponent of B, with the bias, using a compound adder. This creates both possible exponents. The carries created can then be used in conjunction with the carry from A exponent – bias calculation to determine if an overflow has occurred for each of the two possible exponents. If both carries are 1, then an overflow has occurred – the resulting exponent plus the bias is equal to or greater than 2048 – and the exception must be flagged. If both carries are 0, the resulting value is denormalized. If the carries differ, the resulting exponent plus bias are between 0 and 2047. These are the acceptable exponent values. The overflow flags can therefore be detected by an XNOR of the carries.

### 9.4 Summary

This chapter presents a floating-point multiplier using a Booth encoder and a Wallace tree for partial product reduction. IEEE rounding is implemented using injection based rounding. Overflow exceptions are detected and flagged.
Chapter 10  Floating-Point Division

Oberman and Flynn [34] report that while division accounts for only 3% of floating-point operations, it accounts for 40% of the latency. This assumes a twenty clock cycle latency for division, and a three clock cycle latency for multiplication and addition. The large number of result of the large number of cycles required for a division operation to be implemented, as demonstrated with the integer divider.

![Diagram of Generic FP Division Block Diagram](image_url)

**Figure 10.1: Generic FP Division Block Diagram**
As with the floating-point multiplier, the exponent biases in the division must be accounted for. In floating-point division, exponent of the dividend is subtracted from that of the divisor. A straight subtraction results in an exponent without the IEEE bias.

\[(A\text{Exp} + \text{bias}) - (B\text{Exp} + \text{bias}) = Q\text{Exp}\]

The bias must be added to the quotient exponent to conform to the IEEE standard.

### 10.1 FP Divider Using SRT by Comparison Multiples

The divider used in the floating-point division algorithm is a variation of the SRT non-restoring digit recurrence algorithm proposed independently by Sweeney [5], Robertson [40], and Tocher [45]. The benefits of using the SRT divider were discussed in Chapter 5. While it requires more clock cycles to implement than functional dividers, it has a smaller area and operates at a higher frequency.

The quotient digit selection algorithm used for the SRT divider was implemented using the comparison multiples method described in [10] and [31]. The integer implementation has a few key differences from this floating-point implementation from which it was derived. Firstly, as the floating-point divider need only handle one type of multiplication, and not be able to perform both signed and unsigned, the operand conditioning – two’s complement and shifting – that were implemented in the integer divider are not found in the floating-point divider. The removal of the operand shifting is done with the assumption that normalized numbers will have a leading one, and not leading zeroes. This means that the FP divider has a fixed latency, unlike the variable latency integer divider. The other key difference is that the floating-point rounding
algorithm requires three shift registers, as opposed to the two register method utilized in the integer divider.

As the algorithm used for the integer divider was adapted from the floating-point divider, all of the equations derived in Chapter 5 are applicable here. The radix-4 QDS function is instituted once again, with a redundancy factor of $p = \frac{2}{3}$, and a signed digit set of $\{\overline{2}, 1, 0, 1, 2\}$. Thus, the quotient digit selection function is defined as:

$$q_{j+1} = \begin{cases} 
2, & \text{if } \{rP_j\}_c < 0 \text{ and } \{rP_j\}_c < -\{M_2\}_c \\
1, & \text{if } \{rP_j\}_c < 0 \text{ and } -\{M_2\}_c \leq \{rP_j\}_c < -\{M_1\}_c \\
0, & \text{if } \{rP_j\}_c < 0 \text{ and } -\{M_1\}_c \leq \{rP_j\}_c \\
0, & \text{if } \{rP_j\}_c \geq 0 \text{ and } \{rP_j\}_c < \{M_1\}_c \\
1, & \text{if } \{rP_j\}_c \geq 0 \text{ and } \{M_1\}_c \leq \{rP_j\}_c < \{M_2\}_c \\
2, & \text{if } \{rP_j\}_c \geq 0 \text{ and } \{M_2\}_c \leq \{rP_j\}_c 
\end{cases}$$

Utilizing the truncation inequalities determined in Chapter 5,

$$2^c > \frac{6}{2p-1}, \quad e = \log_2 r + i,$$

in conjunction with the fact that the dividend and divisor are shifted such that they fall into the range of $[0.5, 1)$, the minimum number of fraction digits required after truncation, $c$, is determined to be 5. The minimum number of integer digits is 2. This proves to be the same as in the QDS function of integer divider, in spite of the differences in the number of bits used to represent the numbers. As a result, the QDS function used for the floating-point divider can be used with absolutely no changes in the integer divider – or visa versa.

Similarly, much of the remaining divider hardware can be used in both dividers with little or no modification. The multiplexers need only be shortened so that they 55
bits, rather than 66 bits. The comparators for the partial residual computations can also be used with a simple alteration of bit length. The adjust unit that compresses the partial residual into a 55-bit format is then formed by removing the least significant 11 bits from the integer adjust unit. The resulting operation can be seen in the following Table, with the hardware implementation following immediately.

Table 10.1: FP Compression Process for \( I \in \{0,1,2\} \)

<table>
<thead>
<tr>
<th>( w'_{57:53} = ABCXY )</th>
<th>( w_{54:53} = xy )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000.XY</td>
<td>XY</td>
</tr>
<tr>
<td>001.0I</td>
<td>.11</td>
</tr>
<tr>
<td>001.1X</td>
<td>.1X</td>
</tr>
<tr>
<td>00T.0I</td>
<td>.T</td>
</tr>
<tr>
<td>00T.1X</td>
<td>.Tx</td>
</tr>
<tr>
<td>01T.0I</td>
<td>.11</td>
</tr>
<tr>
<td>01T.1X</td>
<td>.1X</td>
</tr>
<tr>
<td>01I.0I</td>
<td>.T</td>
</tr>
<tr>
<td>01I.1X</td>
<td>.Ix</td>
</tr>
<tr>
<td>11T.0I</td>
<td>.11</td>
</tr>
<tr>
<td>11T.1X</td>
<td>.1X</td>
</tr>
<tr>
<td>11I.0I</td>
<td>.T</td>
</tr>
<tr>
<td>11I.1X</td>
<td>.Ix</td>
</tr>
</tbody>
</table>

Figure 10.2: Adjust Unit for Floating-Point Divider
Using binary signed digit representation, the binary representation of a number $A$ is $A^+ - A^-$, so the sign of the number is determined from the subtraction by the first bit position where $A_i^+ \neq A_i^-$. Using the two's complement representation of the negative BSD value produces a sign extended value which then allows the carry-out of the subtraction to indicate the inverse of the sign of $A$. After the inversion of $A^-$, the sign detectors for the partial residuals are composed of the parallel prefix creation – propagate and generate formation – and carry-tree of the Han-Carlson adder. To reduce logic, and subsequently area, only the most significant carry is outputted, and any dot operations that do not contribute to this were removed. The carry-out bit of each sign detector is then run through an invert to find the sign of each of the possible partial residuals. The following figure depicts the modified Han-Carlson tree used for the sign detectors.

Figure 10.3: Floating-Point PR Sign Detector
The largest difference – apart from the lack of operand conditioning – between floating-point and integer dividers occur in the block where the BSD to binary conversion and rounding takes place. The floating-point divider uses the on-the-fly conversion and rounding scheme introduced in [2], and refined in [3] and [4]. The on-the-fly summation is similar to that implemented by the integer divider, except that an a third shift register is introduced that stores the running total plus 1. The Q register maintains an on-the-fly conversion of quotient by concatenating the q values produced each cycle with itself (if positive or zero) or the Qm register (if negative). The Qm register maintains Q – 1 ULP, allowing for negative values to be added to the total without the need to perform a literal subtraction of one ULP before the concatenation. The newly added Qp register maintains Q + 1 ULP to similar effect. Although the choice of a redundancy factor (p) of $\frac{2}{3}$ makes Qp unnecessary for on-the-fly conversion, it is used for rounding. The registers are updated according to the following equations.

$$Qm[j + 1] = \begin{cases} (Q[j], (q_{j+1} - 1)) & \text{if } q_{j+1} > 0 \\ (Qm[j], (r - |q_{j+1}| - 1)) & \text{if } q_{j+1} \leq 0 \end{cases}$$

$$Q[j + 1] = \begin{cases} (Q[j], (q_{j+1})) & \text{if } q_{j+1} \geq 0 \\ (Qm[j], (r - |q_{j+1}|)) & \text{if } q_{j+1} < 0 \end{cases}$$

$$Qp[j + 1] = \begin{cases} (Q[j], (q_{j+1} + 1)) & \text{if } -1 \leq q_{j+1} \leq r - 2 \\ (Qm[j], (r - |q_{j+1}| + 1)) & \text{if } q_{j+1} < -1 \end{cases}$$

In these equations, (x,y) is a concatenation operation. It may be observed that Qp is not covered for all r – Qp becomes itself, shifted left two bits if $q_{j+1}$ is r-1 (3 in this case), however, as the possible $q_{j+1}$ values are {-2,-1,0,1,2} due to the choice of
redundancy factor, so this case is eliminated. These equations also assume that the
registers are 54 bits wide and initialized to zero. A simple modification to allow Q to be
initialized such that the least significant bit is 0 and all other bits are 1, allows
conversion/rounding block to become an indicator of when the proper number of shifts
has q values have been added. Using this method, when the initial 0 overflows to bit 54,
27 two-bit q values have been added to the registers – ensuring that necessary 53 bits of
the normalized quotient have been produced.

The $q_{j+1}$ value is determined by comparison rP with the comparison multiples of
d. As the initial rP value is equal to x, an analysis of the possible values for x and d,
which fall into the range of $[0.5,1)$, and the resulting comparison multiples – 1.5d and
0.5d – results in a $q_{j+1}$ of “01” or “10”, as shown in Table 10.2.

<table>
<thead>
<tr>
<th>x, d Values</th>
<th>x ~1, d ~ 1</th>
<th>x ~1, d = .5</th>
<th>x = .5, d ~ 1</th>
<th>x = d = .5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x vs Comp Mults</td>
<td>.5d &lt; x &lt; 1.5d</td>
<td>x &gt; 1.5d</td>
<td>1.5d &gt; x &gt; .5d</td>
<td>.5d &lt; x &lt; 1.5d</td>
</tr>
<tr>
<td>Resulting q</td>
<td>qj+1 = 1</td>
<td>qj+1 = 2</td>
<td>qj+1 = 1</td>
<td>qj+1 = 1</td>
</tr>
</tbody>
</table>

However, a $q_0$ will be followed by a negative q, to bring the result below 2. So
the quotient after 27 q values have been added – and the overflow to stop the
incrementing of the shift registers – will have a leading one at bit 52 or 51. Combining
this with $q_{28}$ for LSB calculations, gives a 56 bit quotient, allowing for a rounded result.

$$
\begin{array}{cccccccc}
\text{bit number} & 55 & 54 & 53 & 51 & 2 & 1 & 0 \\
q[28] = & 0 & 1 & x & x & \cdots & x & x \\
\end{array}
$$

$$
\begin{array}{cccccccc}
\text{bit number} & 55 & 54 & 53 & 52 & 2 & 1 & 0 \\
q[28] = & 0 & 0 & 1 & x & \cdots & x & x \\
\end{array}
$$

107
In the above representation, the 0 that was initially at the LSB of Q would be at bit 56. Each cycle, two bits are shifted out of the shift register. The least significant of the two bits shifted out each cycle is caught in an overflow bit – depicted above as bit 54. When the overflow bit is 0, the shift registers are disabled to prevent quotient bits from being shifted out in subsequent cycles. It is for this reason that the initialization routine fills all but the LSB with 1s – so that the LSB may act as an indicator that the proper number of cycles has passed. With the shift registers frozen, the entire 53-bit quotient – if the MSB (bit 53 in the above representation) is 1, the LSB is excluded, otherwise bits 52 through 0 are used – can be used in conjunction with the following (n+1) quotient to determine the rounded quotient.

* parallel load with wired left shift

![Diagram of FP Conversion and Rounding](image)

Figure 10.4: FP Conversion and Rounding [5]
Due to the fact that division has the potential to produce irregular fractions, the precision required to implement the round to infinity or round to negative infinity IEEE rounding modes, could theoretically approach an infinite number of decimal places. As a result, it is common for floating-point dividers to implement only the round to nearest (even) rounding mode, as is the case with this divider. The rounded LSB for the quotient in the round to nearest mode can be determined using only the least significant bit of the pre-rounding quotient and the bit position immediately to the right of that.

Since the quotient values are in BSD format, they have the potential to be negative values, so the rounding is slightly more complex than it sounds. A negative value for $q_{28}$ could require that the Qm register be used. Similarly, if the rounding will result in an overflow, the Qp register is used. The following table, shows the rounding rules for the CRN unit.

<table>
<thead>
<tr>
<th>Table 10.3: CRN Rounding Rules [5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>53-bit normalised and rounded $q$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$q_{28}$</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$(Q[27]&lt;51:0&gt;, 0)$</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
</tr>
</tbody>
</table>

| 53-bit normalised and rounded $q$ |

<table>
<thead>
<tr>
<th>$q_{28}$</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(Q[27]&lt;51:0&gt;, 0)$</td>
<td>$(Q[27]&lt;51:0&gt;, 0)$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$(Q[27]&lt;51:0&gt;, 0)$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$(Q[27]&lt;51:0&gt;, 1)$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
<td>$Q[27]&lt;52:0&gt;$</td>
</tr>
</tbody>
</table>
To determine whether the to use the normal \( Q \), the incremented \( Q_p \), or the decremented \( Q_n \), two control signals are used. These signals, \( s_0 \) and \( s_1 \) are used to control multiplexers which select the proper \( Q \). Simultaneously, the rounded LSB bit is computed as \( u \). The values for \( u \), \( s_0 \), and \( s_1 \) can be derived from the following table, which takes into account the most significant bits of \( Q \) and \( Q_m \), as well as \( q_{28} \), and the sign of the partial residual that results from the \( q_{28} \) calculation.

<table>
<thead>
<tr>
<th>( q_{28} )</th>
<th>sign</th>
<th>( Q[27]&lt;52&gt; )</th>
<th>( QM[27]&lt;52&gt; )</th>
<th>( u )</th>
<th>( s_1 )</th>
<th>( s_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2 = 111 )</td>
<td>0</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( 2 = 111 )</td>
<td>0</td>
<td>( \times )</td>
<td>1</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( 2 = 111 )</td>
<td>1</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( 2 = 111 )</td>
<td>1</td>
<td>( \times )</td>
<td>1</td>
<td>( \times )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( 1 = 110 )</td>
<td>0</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( 1 = 110 )</td>
<td>0</td>
<td>( \times )</td>
<td>1</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( 1 = 110 )</td>
<td>1</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( 1 = 110 )</td>
<td>1</td>
<td>( \times )</td>
<td>1</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 = ( \times 00 )</td>
<td>0</td>
<td>0</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 = ( \times 00 )</td>
<td>0</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 = ( \times 00 )</td>
<td>1</td>
<td>0</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 = ( \times 00 )</td>
<td>1</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 = 010</td>
<td>0</td>
<td>0</td>
<td>( \times )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 = 010</td>
<td>0</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 = 010</td>
<td>1</td>
<td>0</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 = 010</td>
<td>1</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 = 011</td>
<td>0</td>
<td>0</td>
<td>( \times )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 = 011</td>
<td>0</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2 = 011</td>
<td>1</td>
<td>0</td>
<td>( \times )</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 = 011</td>
<td>1</td>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Using this table, and some 8x8 Karnaugh maps, the equations for $u$, $s_1$, and $s_0$ can be derived using the six inputs. Using $Sq$, $q_1$ and $q_0$ to denote the sign, middle and LSB bits of $q_{28}$, respectively, these equations were derived as:

$$u = q_1 \cdot \left( q_0 + \left[ (Sq \oplus Sign) \right] \right)$$

$$s_1 = Qm[52] \cdot (Sign + q_0) \cdot (Sq \cdot q_1)$$

$$s_0 = Qm[52] \cdot (Sign + q_0) \cdot (Sq \cdot q_1) + Sq \cdot q_0 \cdot Sign \cdot Q[52]$$

Using these equations, the rounding and normalization portion of the CRN can be implemented as follows.

![Diagram](image_url)

**Figure 10.5: FP Divider Rounding and Normalization Logic**
Using the components described previously in this section, the mantissa divider can be constructed. With the exception of the CRN block, it is structurally very similar to the divider described in Chapter 5 – without the operand conditioning.

![Figure 10.6: FP Divider Block Diagram](image)

### 10.2 Exponent Calculation and Overflow Exception

As mentioned previously, the unbiased exponent of the quotient is equal to the difference between the unbiased exponents of the operands. Assuming the normalized mantissa range of [1,2), the possible quotient range is (0.5, 2). As the range of [1,2) is normalized, any quotient less than one must be shifted left, and the exponent must be decremented. The subtraction of the exponents has the possibility of creating an overflow condition, when the unbiased exponent of the divisor is negative. An exception flag must be triggered in this event, but the flag need not be differentiated from the divide-by-zero exception.
When the exponent subtraction of A-B occurs is processed, it has the effect of removing the bias from the resulting exponent. The bias must then be added back into the resulting exponent so that it conforms to the IEEE 754 standard. To form both possible exponents – the biased subtraction, and the biased subtraction minus one used if the quotient from the mantissa division is less than one – a compound subtractor is used. The compound subtractor performs the compound addition of the exponent of A and the inverse of the exponent of B. By not including the addition of the ULP for proper two’s complement subtraction, the result of the addition becomes Aexp-Bexp-1 for the sum output, and Aexp-Bexp for the incremented sum output. The bias is then added to each of the possible exponents, and the same bit used to select whether or not to use the rounding bit, u, is used to select the exponent to output.

The overflow condition occurs only when the absolute value of the divisor - $|2^{B_{\exp-bias}} \cdot 1.f|$ - is less than one, and thus results in a quotient that is larger than the absolute value of the dividend. More specifically, the overflow occurs when the dividend exponent minus the divisor exponent plus the bias results in a value greater than the 10-bit exponent can hold (2047). The detection of the overflow condition lends itself easily from the exponent calculation. Firstly, the overflow will only occur if $A > B$ - if $A \leq B$ and $B < 1$, the resulting quotient will be in the range of $(A, 1]$, and will not overflow. If this condition is met, the subtraction of the exponents will results in a carry-out of 1. Secondly, for an overflow to occur, the unbiased quotient exponent plus the bias must create a carry-out. The overflow condition can therefore be detected using a simple AND of the carry-outs of the compound subtraction and subsequent bias additions. This
produces two possible overflow bits – one for $A_{\text{exp}}-B_{\text{exp}}$ and one for $A_{\text{exp}}-B_{\text{exp}}-1$.

The proper overflow bit is chosen along with its corresponding exponent.

### 10.3 Summary

This chapter describes the creation of a double-precision divider following the IEEE 754 standard and supporting round-to-nearest (even) rounding. The SRT non-restoring digit recurrence algorithm is implemented using a radix-4 quotient digit selection function by comparison multiples, with on-the-fly digit conversion, rounding and normalization. The divider detects and triggers an exception bit for the overflow and divide-by-zero exceptions as specified in the MIPS64 ISA.
Chapter 11  FPU Synthesis Results

Each of the floating-point blocks was implemented in VHDL. Test benches were developed in ModelSim, and executed in order to ensure that the functions performed as expected. The VHDL code was then synthesized under worst case conditions using Synopsys Design Compiler (2006.06) and the power, area, and delay were observed. Post synthesis verification was then done on the synthesized circuits. Unlike the integer adder and divider, there are no DesignWare libraries against which to compare the FP implementations, so the FP blocks were compared to the ALU in terms of speed to ensure that no component required a longer clock period than the universal integer divider.

11.1 FP Adder/Subtractor Synthesis

The floating-point adder/subtractor requires two cycles to execute an instruction but is fully pipelined such that one instruction may be issued per cycle, without affecting the operation issued the previous cycle – giving it a maximum throughput of one operation per cycle. When synthesized, the reported area was 16490 gates, with a maximum power draw of 92 mW. At a latency of 2.67 ns – a frequency of 375 MHz – the FP adder/subtractor is slightly faster than the integer divider. The schematic produced by Synopsys can be seen in the following figure. Due to the lack of logic outside of the R, N, and path selection blocks, the top view of the design is relatively uncluttered.
11.2 FP Multiplier Synthesis

The floating-point Booth-Wallace multiplier utilizes a four clock cycle structure that is divided very similarly to the integer multiplier. The three sets of registers are place after the Booth encoding, and after each two stages of the Wallace tree. Unlike the integer multiplier, the critical path of the floating-point multiplier lies not in the Wallace-tree stages, but in the final addition and rounding. As a result, the FP multiplier has a longer delay than the integer multiplier, at 2.32 ns – an operating frequency of 431 MHz – but is still faster than the integer divider. The floating-point multiplier has an area of 55,891 gates and a maximum dynamic power of 197.6 mW.
11.2 FP Divider Synthesis

The floating-point divider compares favorably to the integer divider in terms of both size and speed. Since the critical path of each is the same portion of the circuit – the creation of the new partial residual – and floating-point divider uses fewer bits to perform the calculations, the floating-point divider is slightly faster than the integer divider at a speed of 392 MHz – a delay of 2.55 ns. The lack of operand conditioning in the floating-point divider, combined with the smaller number of bits to be processed by the actual divider circuit, counter the addition of the rounding and exponent circuitry to result in an area of 11,879 gates. The maximum dynamic power of the floating-point divider was reported as 2.315 mW. Division is the only operation that is iterative, and thus does not allow for pipelining. As with the ALU, multiple FPUs can be included to increase throughput.
11.4 FPU Synthesis

The completed floating point unit added utilized the three floating-point arithmetic designs, and added a small amount of logic to switch between which is being used. As a result, the floating point unit is slightly slower than the divider with a delay of 2.82 ns, and an operating frequency of 355 MHz. The area of the FPU is 84,440 gates. It has a maximum dynamic power of 153.9 mW. As one instruction can be completed per cycle when executing addition or multiplication instructions, the maximum number of floating-point operations per second (FLOPS) is 355 million.

The schematic for the floating-point unit, as taken from Synopsys, primarily shows the multiplexer logic, as that was much of the top level of VHDL code. The floating-point adder, multiplier and divider in the circuit have been labeled.
Figure 11.4: FPU from Synthesis
Chapter 12  Conclusions and Future Work

Both a 64-bit arithmetic logic unit and a double-precision floating-point unit were designed, modeled in VHDL and tested using ModelSim. The ALU performs arithmetic and logical shifts, logic operations including XOR, NOR, OR, AND, and INV, as well as signed and unsigned integer addition, subtraction, multiplication and division. The addition/subtraction was implemented using a 64-bit Han-Carlson/carry-select hybrid adder that operates at higher speeds than standard parallel prefix adders. The multiplier was implemented using an signed/unsigned Booth-Wallace structure. Division was implemented using a novel approach that combined a comparison multiples SRT divider with operand conditioning to perform both signed and unsigned division at variable latency. The ability to perform variable latency division reduces throughput for all cases where the (positive) operands have leading zeros.

The floating-point unit performs double-precision addition, subtraction and multiplication with all four IEEE rounding modes supported. FP addition and subtraction were implemented using a dual path adder based on the Seidel-Even adder [41]. The multiplier used a Booth-Wallace structure with rounding by injection. The FP divider utilized an SRT digit recurrence algorithm with quotient digit selection by comparison multiples.

All designs were constrained and synthesized using Synopsys Design Compiler (2006.06). The delay, area, and power were recorded. The ALU synthesis reported an area of 122,215 gates, a power of 384 mW, and a delay of 2.89 ns – a frequency of 346 MHz. At one operation per second for addition, this results in 346 million instructions per second. The FPU synthesis reported an area 84,440 gates, a delay of 2.82 ns and an
operating frequency of 355 MHz or 355 MFLOPS when pipelining is considered. It has a maximum dynamic power of 153.9 mW.

While the components were designed for speed, and compared favorably with the corresponding DesignWare components, the exhibited speeds of around 350 MHz under worst case conditions, or around 1 GHz for best case conditions, are nowhere near the speeds produced by commercial processors. Much of this can be attributed to the speed differences between the mixed (static and dynamic) logic used in industry and the static logic produced by synthesis with Synopsys. The SOI CMOS that is used in industry has also been shown to provide significant speed improvements of the bulk CMOS used here.

Future work should include rewriting the VHDL code in Verilog. Verilog provides a lower level implementation and also allows for transistor modeling. The transistor modeling can be used to create dynamic logic to increase speed, however transistor level synthesis is not currently supported by Synopsys. A concerted effort to implement the designs in both SOI and dynamic logic would result in a product that could be compared to industry products. An examination into compressor structures should also be undertaken to allow for multipliers that are smaller in terms of both delay and area.
Bibliography


18. Gunawan, S., Hsu, K., “Design and Analysis of 64-bit Han Carlson Adder in 0.18μm CMOS,” submitted to ACM, April, 2004.


28. MIPS Technologies, MIPS64 Architecture For Programmers Volume II: The MIPS64 Instruction Set, Revision 0.95, March 2001.


