“a + b” ARITHMETIC — THEORY AND IMPLEMENTATION

A Thesis Presented to
The Faculty of the Russ College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

By
SenthilKumar Manickavasagam
March, 1996
Acknowledgements

I sincerely thank Dr. Janusz Starzyk for his guidance and energy. He was more of a mentor than a guide. Without him this work would have been immature. Thank You Sir.

Special thanks to Mr. Simon Hoare, who gave me the SPARC code from Texas Instruments, Inc. Many thanks to professors Dr. Robert Judd, Dr. Robert Curtis and Dr. Larry Snyder for their support.

I like to thank my parents for bringing me to this level. My very dear grandfather for his selflessness and being there always. Also, Shyamala for her patience and encouragement.

Finally, many thanks to all my teachers and professors for making me what I am now. Thank you all.
Acknowledgements

1 Introduction
   1.1 Organization
   1.2 Convention Used in This Thesis

2 Background
   2.1 Addition
   2.2 Multiplication
   2.3 Floating Point

3 "a + b" Arithmetic
   3.1 The "a + b" Representation
      3.1.1 Addition
      3.1.2 Subtraction
      3.1.3 Scanning Algorithm for Multiplication
   3.2 "a + b" Floating Point
      3.2.1 Addition
      3.2.2 Multiplication
4 VLSI consideration for Multiplication in “a + b” representation 26
4.1 CSA and MCSA ........................................................................ 26
4.2 OCMCSA ................................................................................. 30
4.3 Estimation .................................................................................. 32

5 SPARC 40
5.1 Integer Unit ............................................................................... 40
5.1.1 Registers ............................................................................... 41
5.1.2 Instructions ............................................................................ 46
5.2 Floating Point Unit (FPU) .......................................................... 53
5.3 Coprocessor Unit (CP) ............................................................... 54
5.4 Data Formats ............................................................................. 54
5.5 Memory Model ........................................................................... 55
5.6 Reference Memory Manager Unit .............................................. 57

6 VHDL 59
6.1 VHDL Designs ........................................................................... 60
6.1.1 Library .................................................................................. 60
6.1.2 Package ................................................................................ 60
6.1.3 Entity .................................................................................... 61
6.1.4 Architecture ......................................................................... 61
6.1.5 Configuration ........................................................................ 62
### CONTENTS

9 Simulation Environment 104

9.1 System V Application Binary Interface .......................... 105

9.2 SPARC Processor Specific .......................... 105

9.2.1 Functions ........................................ 106

9.3 File Format ........................................ 114

9.3.1 Executable File Format ........................................ 115

9.3.2 Input File Format - indata ........................................ 124

9.4 Mentor Environment ........................................ 125

9.5 Tools - elf2memimg ........................................ 125

9.6 Process to create the indata ........................................ 127

9.7 Other helpful tools ........................................ 129

9.7.1 decipherlib.c ........................................ 129

9.7.2 deciphermemimg.c ........................................ 129

9.7.3 decipher.c ........................................ 130

9.7.4 elfgetscn.c ........................................ 130

9.7.5 getanddec1.c ........................................ 131

9.7.6 printhdrs.c ........................................ 131

10 Simulation 133

10.1 Sample A ........................................ 133

10.1.1 Simulation by SPARC ........................................ 135
CONTENTS

10.1.2 Simulation by SPARCAB .................................. 136

10.2 Sample B ......................................................... 136

10.2.1 Simulation by SPARC ....................................... 138

10.2.2 Simulation by SPARCAB ................................... 139

10.2.3 Comparison ..................................................... 139

11 Conclusion .......................................................... 142

11.1 Future Work ..................................................... 143

Bibliography .......................................................... 144
List of Tables

3.1 Two-bit recoding .................................................. 14
3.2 Three-bit recoding .................................................. 15
3.3 Four-bit recoding ................................................... 16
3.4 Two-bit recoding for “a + b” (k = 1) ............................. 17
3.5 Three-bit recoding for “a + b” (k = 2) ............................ 18

4.1 Time - area estimate of 64 bits multiplier .......................... 33

5.1 Window addressing ................................................... 42
5.2 op field coding in SPARC instructions .............................. 49
5.3 op2 field coding in SPARC instructions ............................. 49

9.1 .Q.cmp return values ................................................. 108
9.2 e.type values in a ELF header ...................................... 118
9.3 Typical program header segments .................................. 121
List of Figures

3.1 “a + b” Full adder ........................................ 10
3.2 Window Addition ........................................ 19
4.1 Ordinary CSA ........................................ 27
4.2 Modified CSA ........................................ 28
4.3 Description of Blocks ................................. 29
4.4 OCMCSA ................................................ 31
4.5 64-bit Multiplication add array mapped to OCMCSA .. 32
4.6 General Structure of “a + b” Multiplier .............. 35
4.7 Modified CSA With the Final Storage ................ 36
4.8 Experiment Process ..................................... 38
5.1 Circular Stack of Overlapping Windows .............. 42
5.2 Fields of the Processor Status Register ............ 43
5.3 TBR Fields ............................................. 45
5.4 Instruction Formats .................................... 47
5.5 Data Formats ........................................... 56
5.6 SPARC IU and MMU .................................. 58
6.1 Simulation Cycle ....................................... 66
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>The Model</td>
<td>68</td>
</tr>
<tr>
<td>9.1</td>
<td>A Function’s Window Registers</td>
<td>107</td>
</tr>
<tr>
<td>9.2</td>
<td>Object File Format</td>
<td>115</td>
</tr>
<tr>
<td>9.3</td>
<td>Input File Format</td>
<td>124</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Something that has always been a high priority in the minds of those designing a new arithmetic processor is "speed." Traditionally this need had been satisfied by high speed devices and better algorithms. But, this thesis introduces a new concept in computer arithmetic aimed in satisfying the same need. The study describes algorithms for addition, subtraction and multiplication. It also discusses a parallel hardware arithmetic for the proposed algorithms. The architecture is capable of performing additions in constant time and multiplication, in less time than in the best known architectures. The internal representation of the numbers is called "a + b" representation and requires two memory words.

By using "a + b" representation internally in a processor, arithmetic operations and hence, the performance can be improved. As a proof of this concept, a generic SPARC (Scalable Processor ARCHitecture) was modified to use "a + b" arithmetic internally and the timings were compared with the original. The second part of this thesis describes this process. The generic SPARC was written and is currently distributed by Texas Instruments, Inc.
CHAPTER 1. INTRODUCTION

1.1 Organization

This thesis is organized in two parts:

1. Part 1: The “a + b” arithmetic.


The first part gives the background and explains the “a + b” arithmetic. Also in this part, are chapters that explain the floating point arithmetic, the algorithms for “a + b” arithmetic and some VLSI considerations for the multiplier architecture.

In the second part, an implementation of SPARC is described with the help of a frame work. Prior to that are brief introductions to SPARC and VHDL which are essential for understanding the frame work.

1.2 Convention Used in This Thesis

The following fonts are used in this thesis:

var is used for variables.

proc is used for procedures and like.

pros is used for processes.

sig is used for signal names and like.
inst is used for instructions.

This font convention is strictly followed in this thesis to enhance the understanding of the material.
Part I: The "a + b" Arithmetic

Chapter 2

Background

Before proceeding to explain the "a + b" arithmetic, this chapter presents a glimpse of various studies conducted in the area of arithmetic addition and multiplication. Also presented in this chapter are comparisons to "a + b" arithmetic.

2.1 Addition

A study conducted by Burks et al.[5] concluded with the statement that binary addition requires $\log_2 n$ gate delays ($n$ is the word-length) for carry propagation. Another detailed study by Reitwienser[16] showed that addition must take $1 + \log_2 n$ gate delays for the worst case carry propagation. The "Carry Look Ahead" adder used by MacSorley[10] also achieved $O(\log_2 n)$ carry propagation delay complying with the study by Burks et al. Sklansky[19] achieved similar delay $(2 + 2\log_2 2(n + 1))$ using conditional sum logic. With the advent of VLSI technology, and a need for regular layout, several new algorithms were proposed. The delay were of $O(n)$ for a design by Atkins et al.[2] and for another design by Brent et al.[4]. $O(\log_2 n)$ was achieved by Ngai[13]. In [9], the arrival time of the carry signal for MSB was $O(\log_4 N)$. As
Chapter 3

"a + b" Arithmetic

Usually the binary numbers are represented by a single string of 0s and 1s. This way of representing numbers causes long delays to complete arithmetic operations. In what follows, a new method of representing numbers is presented. By representing numbers in this new way it was found that arithmetic operations can be completed in shorter time, invariant on the number of bits in a binary representation.

3.1 The "a + b" Representation

In the new method a binary number is represented as a sum of two numbers. This is "a + b" representation. For example,

\[
9_{10} = 1001_2 = 0110_2 + 0011_2 = 1000_2 + 0001_2
\]  

(3.1)

Although this representation is not unique, the advantage is that, addition can be carried out in constant time. Also, note that this representation is commutative due to the commutative property of addition. If the numbers are stored in memory, in "a + b" representation, both "a" and "b" part will have the same address, and the
increase in time to read a long word from memory is minimal, compared to reading “a” and “b” parts separately.

Both positive and negative numbers can be represented in “a + b”. For representing negative numbers both the “a” and “b” parts are represented in 1’s or 2’s complement, with the MSB of each part being the sign bit. In the following example, the numbers are represented by five bit, 2’s complement form with the left most bit as the sign bit:

\[-9_{10} = 11001_2 + 11101_2 = -6_{10} + -3_{10}\]  \hspace{1cm} (3.2)

### 3.1.1 Addition

Suppose, we need to add \(9_{10} + 3_{10} = 12_{10}\), then in the “a + b” representation, this addition is carried out as follows:

Let us assume that numbers are stored in their “a + b” representation, e.g.

\[9_{10} = 0110_2 + 0011_2\]  \hspace{1cm} (3.3)

and

\[3_{10} = 0010_2 + 0001_2\]  \hspace{1cm} (3.4)

The addition is carried out in two steps.

1. In the first step, “a” and “b” parts of the first number and “a” part are added
CHAPTER 3. "a + b" ARITHMETIC

to produce a temporary sum and carry.

\[
\begin{array}{c}
0110_2 \\
0011_2 \\
0010_2 \\
\hline
\text{Temp. Sum} & 0111_2 \\
\text{Temp. Carry} & 0100_2
\end{array}
\]

2. In the next step, the "b" part of the addend is added to the temporary result to produce the final result.

\[
\begin{array}{c}
0111_2 \\
0100_2 \\
\hline
0001_2 \\
\text{sum} & 0010_2 \Rightarrow 2_{10} \\
\text{carry} & 1010_2 \Rightarrow 10_{10}
\end{array}
\]

As a result of this addition we obtain \(0010_2 + 1010_2\) which is a correct "a + b" representation of the result. The sum is stored as the "a" part and the carry as the "b" part.

The full adder for "a + b" arithmetic can be represented as shown in Fig.3.1.

Thus the delay in this type of parallel addition is equal to two full adder delays.
irrespective of the word length. Though the addition is not complete, the "half-cooked" result can be used for further calculations without wasting time for carry propagation. Not all the results are to be displayed to the user. Hence, it is enough to propagate the carry, only when the results are to be displayed. This can be taken care of by a display circuit. By doing this addition at the time of display, we would not lose much time, as the output devices are slow compared to the adder circuit that is required for the final addition. Thus, the lower bound for addition as formulated by Burks et al.[5] is "in a way" overcome by this "$a + b$" representation.

Figure 3.1: "$a + b$" Full adder
3.1.2 Subtraction

The first step in subtraction is negating the subtractant. As mentioned before (refer Section 3.1), the negative numbers are represented in the one's or two's complement form. Finding the one's complement is the same as finding logical "not" function. The "not" function can be achieved in a constant delay similar to addition time. The algorithm for one's complement of a number represented by \( a + b \) is as listed below:

1. bitwise complement \( a \) and \( b \)

2. partial add and shift carry 1 bit (discard the MSB of carry)

3. add 1

For example, if \( A = 9_{10} = 0110_2 + 0011_2 \) then, \( \bar{A} = 0110 \) (the actual result). In the first step, we write one's complements

\[
\begin{align*}
1001 \\
1100
\end{align*}
\]
then, in the second step, the two numbers are partially added and the carry is ignored:

\[
\begin{array}{c}
0101 \\
10000 \\
\hline \\
0101 \\
0000
\end{array}
\]

(3.6)

and in the third step, one is added:

\[
\begin{array}{c}
0101 \\
0000 \\
0001 \\
\hline \\
0100 \\
0010
\end{array}
\]

(3.7)

which after final addition gives the actual result — the one's complement.
For 2’s complement we need to add 2 instead of 1 in the final step. Consider the
same number: The first two steps remain the same. In the third step,

\[
\begin{array}{c}
0101 \\
0000 \\
0010 \\
0111 \\
0000 \\
\end{array}
\]

which is the 2’s compliment of 9.

Once the negative numbers are represented in the one’s or two’s complement
form, subtraction is performed like addition. Since addition and “not” are performed
in parallel, subtraction becomes a parallel operation and time taken is invariant on
the word length.

3.1.3 Scanning Algorithm for Multiplication

The multiplication can be carried out using a modified Booth’s algorithm[3]. The
Booth’s algorithm looks at 2 bits of the multiplier number \(X\), and depending on the
particular combination at hand, it does one of the following three operations:

1. If \(x_ix_{i+1} = 00\) or \(x_ix_{i+1} = 11\) it then shifts the existing sum of partial products
   one bit to the right.
2. If \( x_i x_{i+1} = 01 \) it then adds the multiplicand \( Y \) to the existing sum of partial products and then shifts the result one bit to the right.

3. If \( x_i x_{i+1} = 10 \) it then subtracts the multiplicand \( Y \) from the existing sum of partial products and then shifts the result one bit to the right.

<table>
<thead>
<tr>
<th>Pair Value</th>
<th>Signed Digit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>+1</td>
</tr>
<tr>
<td>10</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.1: Two-bit recoding

The process starts by appending a 0 to the right of the LSB of the multiplier \( X \) (denoted by \( x_{-1} \)) and then looking at binary pairs beginning with \( x_{-1} \). This scheme is called "look forward approach." On the other hand if the process starts by appending the zero to the left of MSB, the scheme is called "look back approach" (a slight modification is done in the three operations). Adjacent pairs share one bit so that in every iteration only one bit retires. Note, that the required multiples of the multiplicand are \( +Y \) and \( -Y \) which are readily available. In the original Booth algorithm "n" iterations are required for a multiplier \( X \) with word size \( n \). MacSorley[10] proposed a modified Booth's algorithm, in which a triplet of bits instead of a pair was looked at in every iteration. This requires a total of \( n/2 \) iterations with two bits retiring, and two product bits resulting from every step. A multibit recoding algorithm for signed
two's complement binary numbers was presented and proved by Sam et al. [17]. In their work, a signed-digit (SD) number system [8] in a radix $r$ based on a redundant set of signed digits is used. The multibit recoding algorithm follows similar steps as the Booth algorithm. To ease the recoding, look-up tables can be formed for different window sizes. Results are given for 2, 3, and 4 bit recoding in Tables 3.1, 3.2 and 3.3, respectively. Note the requirement of larger multiples of the multiplicand for wider windows. This is quite expensive. The even multiples of multiplicand $Y$ can easily be obtained by just a left shift, whereas the odd multiples call for an extra addition.

A very similar algorithm is followed for “$a + b$” representation. Here, instead of considering 2, 3, 4 or 5 bits in a window, 4, 6, 8 or 10 bits are considered in a window, of which one half belongs to the “$a$” part and the other half to “$b$”. For instance, consider a window which contains $k + 1$ bits of both “$a$” and “$b$” terms of

<table>
<thead>
<tr>
<th>Triplet Value</th>
<th>Signed Digit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>+1</td>
</tr>
<tr>
<td>010</td>
<td>+1</td>
</tr>
<tr>
<td>011</td>
<td>+2</td>
</tr>
<tr>
<td>100</td>
<td>-2</td>
</tr>
<tr>
<td>101</td>
<td>-1</td>
</tr>
<tr>
<td>110</td>
<td>-1</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2: Three-bit recoding
CHAPTER 3. "a + b" ARITHMETIC

Table 3.3: Four-bit recoding

<table>
<thead>
<tr>
<th>Quadruplet Value</th>
<th>Signed Digit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+1</td>
</tr>
<tr>
<td>0011</td>
<td>+2</td>
</tr>
<tr>
<td>0100</td>
<td>+2</td>
</tr>
<tr>
<td>0101</td>
<td>+3</td>
</tr>
<tr>
<td>0110</td>
<td>+3</td>
</tr>
<tr>
<td>0111</td>
<td>+4</td>
</tr>
<tr>
<td>1000</td>
<td>-4</td>
</tr>
<tr>
<td>1001</td>
<td>-3</td>
</tr>
<tr>
<td>1010</td>
<td>-3</td>
</tr>
<tr>
<td>1011</td>
<td>-2</td>
</tr>
<tr>
<td>1100</td>
<td>-2</td>
</tr>
<tr>
<td>1101</td>
<td>-1</td>
</tr>
<tr>
<td>1110</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
</tr>
</tbody>
</table>

The right-most column overlaps the previous window\(^1\) (\(k^{th}\) bit position in the previous window). The procedure can be stated as follows:

\[ a_k \ldots a_4 a_3 a_2 a_1 a_0 + b_k \ldots b_4 b_3 b_2 b_1 b_0 \]  

\[ (3.9) \]

writing "a" part over the "b" part consider the window

\[ a_k \ldots a_4 a_3 a_2 a_1 a_0 \]

\[ b_k \ldots b_4 b_3 b_2 b_1 b_0 \]

\[ (3.10) \]

\(^1\)For a 5-bit window, \(k = 4\)
CHAPTER 3. “a + b” ARITHMETIC

<table>
<thead>
<tr>
<th>Pair Value</th>
<th>Multiples of Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>0</td>
</tr>
<tr>
<td>00 01</td>
<td>1</td>
</tr>
<tr>
<td>00 10</td>
<td>-1</td>
</tr>
<tr>
<td>00 11</td>
<td>0</td>
</tr>
<tr>
<td>01 01</td>
<td>2</td>
</tr>
<tr>
<td>01 10</td>
<td>0</td>
</tr>
<tr>
<td>01 11</td>
<td>1</td>
</tr>
<tr>
<td>10 10</td>
<td>-2</td>
</tr>
<tr>
<td>10 11</td>
<td>-1</td>
</tr>
<tr>
<td>11 11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.4: Two-bit recoding for “a + b” (k = 1)

1. Starting from the rightmost column of the window consider the weights\(^2\) of \(a_i\) and \(b_i\) as equal to 1, 1, 2, 4, 8, ..., \(2^{k-2}\) and \(-2^{k-1}\) respectively.

2. Add weighted columns within the window.

3. The result determines a multiple of multipicand shifted by \(k(i - 1)\) bits to the left (i.e. multiplied by \(2^{k(i-1)}\)), which has to be added to the partial product obtained by adding multiples of multipicand determined by other window positions.

This window addition can be realized using simple counters or a dedicated logic.

The required logic is to count the number of ones in each column, weight them, and do a final addition to find the multiple of the multipicand, as shown in Fig.3.2. The larger the window used, the smaller the number of multiples of multipicand needed.

\(^2\)i stands for the \(i^{th}\) window and it ranges from 1 to nwindows
<table>
<thead>
<tr>
<th>Triplet Value</th>
<th>Multiples of Multiplicand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 000</td>
<td>0</td>
</tr>
<tr>
<td>000 001</td>
<td>1</td>
</tr>
<tr>
<td>000 010</td>
<td>1</td>
</tr>
<tr>
<td>000 011</td>
<td>2</td>
</tr>
<tr>
<td>000 100</td>
<td>-2</td>
</tr>
<tr>
<td>000 101</td>
<td>-1</td>
</tr>
<tr>
<td>000 110</td>
<td>-1</td>
</tr>
<tr>
<td>000 111</td>
<td>0</td>
</tr>
<tr>
<td>001 001</td>
<td>2</td>
</tr>
<tr>
<td>001 010</td>
<td>2</td>
</tr>
<tr>
<td>001 011</td>
<td>3</td>
</tr>
<tr>
<td>001 100</td>
<td>-1</td>
</tr>
<tr>
<td>001 101</td>
<td>0</td>
</tr>
<tr>
<td>001 110</td>
<td>0</td>
</tr>
<tr>
<td>001 111</td>
<td>1</td>
</tr>
<tr>
<td>010 010</td>
<td>2</td>
</tr>
<tr>
<td>010 011</td>
<td>3</td>
</tr>
<tr>
<td>010 100</td>
<td>-1</td>
</tr>
<tr>
<td>010 101</td>
<td>0</td>
</tr>
<tr>
<td>010 110</td>
<td>0</td>
</tr>
<tr>
<td>010 111</td>
<td>1</td>
</tr>
<tr>
<td>011 011</td>
<td>4</td>
</tr>
<tr>
<td>011 100</td>
<td>0</td>
</tr>
<tr>
<td>011 101</td>
<td>1</td>
</tr>
<tr>
<td>011 110</td>
<td>1</td>
</tr>
<tr>
<td>011 111</td>
<td>2</td>
</tr>
<tr>
<td>100 100</td>
<td>-4</td>
</tr>
<tr>
<td>100 101</td>
<td>-3</td>
</tr>
<tr>
<td>100 110</td>
<td>-3</td>
</tr>
<tr>
<td>100 111</td>
<td>-2</td>
</tr>
<tr>
<td>101 101</td>
<td>-2</td>
</tr>
<tr>
<td>101 110</td>
<td>-2</td>
</tr>
<tr>
<td>101 111</td>
<td>-1</td>
</tr>
<tr>
<td>110 110</td>
<td>-2</td>
</tr>
<tr>
<td>110 111</td>
<td>-1</td>
</tr>
<tr>
<td>111 111</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.5: Three-bit recoding for "a + b" (k = 2)
CHAPTER 3. "a + b" ARITHMETIC

Figure 3.2: Window Addition

Multiples of Multiplicand
to obtain the final product. In general the number of multiples of the multiplicand to be added can be given as:

\[
\text{Number of Multiples} = \left\lceil \frac{n + 1}{k - 1} \right\rceil
\]  

(3.11)

where, \( n \) is word size and \( k \) is window size. However, there is a tradeoff between reduction in the number of multiples of the multiplicand and increase in the computational cost and hardware required to produce and deliver different multiples of multiplicand. It can be shown that with respect to the area and time delay product for VLSI realization of 64 bit multiplication the optimum window is five bits wide, i.e., we scan 10 bits (5 of "a" 5 of "b"). Discussion on the effect of the window size on the performance and the architecture of multiplier is related to VLSI layout realization, and is carried out in Chapter 4.

As it was stated for the SD representation it is easy to form the look up tables for recoding for the multiplication in the "a + b" representation. These are given in Table 3.4 and Table 3.5 for \( k = 1 \) and \( k = 2 \) respectively. Not all combinations are given because of the commutative property of the "a + b" representation.

Let's consider few examples.
Example 1

To multiply two numbers $A$ and $B$, where $A = 3_{10} = 0010_2 + 0001_2$ and $B = 9_{10} = 0110_2 + 0011_2$, use scanning of $B = 9_{10}$ with a 3 bit window. The “$a$” and “$b$” parts of $B$ are written one over the other:

\[
\begin{array}{c|c}
01 & 10 \\
00 & 11 \\
\end{array}
\]

and two windows used with one bit from the previous bit position give (we assume that the LSBs are preceded with zeros)

\[
\begin{align*}
011_2 & \quad 100_2 \\
001_2 & \quad 110_2
\end{align*}
\]  
(3.12)

From Table 3.5, it can be seen that the first window (left) causes addition of $-3A$, and the second addition of $+3A$. First, $-3A$ must be added to the product (initially zero). Then $+3A$ is shifted by 2 bits (in general, $k(i-1)$ bits) to the left and added to the previous result (equivalent of adding $12A$). Thus the combined result represents $9A$. This special parallel arithmetic is the most advantageous for large word-length operations (e.g. 64 bits or more).
Example 2

Consider an example of 16-bit multiplication with the five bit scanning window. Let the multiplicand be $51467_{10}$ represented in "a + b" as:

$$51467_{10} = 7662_{16} + 52a9_{16} \quad (3.13)$$

The scanning windows and the respective multiples they generate are:

00100 $\equiv$ (20) \quad (3.14)

10010 $\equiv$ (90)

\[ -8 + 2 + 1 = -5 \]

01100 $\equiv$ (60) \quad (3.15)

10101 $\equiv$ (a1)

\[ -8 + 4 + 4 + 1 = 1 \]
CHAPTER 3. "a + b" ARITHMETIC

\[ \begin{align*}
01100 & \equiv (60) \quad (3.16) \\
00101 & \equiv (21) \\
\hline
4 + 4 + 1 & = 9 \\
\hline
01110 & \equiv (70) \quad (3.17) \\
01010 & \equiv (50) \\
\hline
8 + 2 + 2 & = 12 \\
\hline
\end{align*} \]

The addition of the multiples can be shown as:

\[ -5 + 1 \times 2^{4(2-1)} + 9 \times 2^{4(3-1)} + 12 \times 2^{4(4-1)} \]

\[ = -5 + 16 + 2304 + 49152 \]

\[ = 51467 \]
3.2 “a + b” Floating Point

The “a + b” floating point representation is an extension of the “a + b” integer representation. The Floating point numbers are represented as shown below:

\[(a + b) \times 2^{c+d}\]  \hspace{1cm} (3.20)

3.2.1 Addition

The first step which is needed for the addition of two floating point numbers say

\[A = (a_1 + b_1) \times 2^{(c_1+d_1)}\]
\[B = (a_2 + b_2) \times 2^{(c_2+d_2)}\]  \hspace{1cm} (3.21)

is to make the exponents equal. First the exponents are added using “a + b” addition in order to determine the necessary mantissa transformation.

\[
c_2 + d_2 - c_1 + d_1 = e + f\]  \hspace{1cm} (3.22)

If \(e > 0\) then \(a_1\) and \(b_1\) are shifted right by \(e\) bits else shifted left by \(e\) bits. Similarly for \(f\). After these transformations, both numbers are equal in their exponents, and
their shifted mantissas can be added as explained in the Section 3.1.1.

\[(a_1 + b_1 + a_2 + b_2) \times 2^{(e_2 + d_2)}\]  \hspace{1cm} (3.23)

### 3.2.2 Multiplication

The multiplication of floating point numbers requires \(a + b\) addition of exponents and multiplication of the mantissas. The final result is obtained from

\[(a_1 + b_1) \times (a_2 + b_2) \times 2^{(e_1 + d_1 + e_2 + d_2 + \Delta)}\]  \hspace{1cm} (3.24)

In the above equation \(\Delta\) is to account for the final stripping of leading zeros. The final stripping is done to maintain a more accurate answer and avoid storing unnecessary leading zeros. For example:

\[0.5 \times 10^{-1} \times 0.2 \times 10^{-2} = \]
\[0.01 \times 10^{-3}\]  \hspace{1cm} (3.25)

In the above example instead of storing \(0.01 \times 10^{-3}\) we can store \(1 \times 10^{-5}\). This step of removing leading zeros is called stripping.
Chapter 4

VLSI consideration for Multiplication in \( a + b \) representation

The previous chapter described the algorithms for addition, subtraction and multiplication. This chapter describes the constraints for laying out a multiplier that would use the algorithm delineated in Chapter 3. Later in this chapter, a regular layout for the multiplier is also suggested.

4.1 CSA and MCSA

Usually, the multiples of the multiplicands are added using a Carry Save Adder (CSA). The shifting and adding cells for a CSA tree is as shown in Fig.4.1. The standard CSA tree has an irregular structure, which is a disadvantage for VLSI implementation. To avoid this, a regular structure as shown in Fig.4.2 is used. In the figure each block is a full-adder of full word-length (i.e., if word length is 64 bits then each block has 64 full adders). Each column in the figure processes different multiples of multiplicand\((M_1, M_2, \ldots, M_{32})\) to be added as required by scanning algorithm (refer page 16). Partial results obtained from each column are combined employing a small CSA tree. The
CHAPTER 4. VLSI Consideration

Figure 4.1: Ordinary CSA
Figure 4.2: Modified CSA
last stage denoted by "storage and final addition" is the stage that increases the delay by three (due to two full adders with feedback as shown in Fig. 4.3).

The number of multiples of multiplicand generated by the scanning algorithm depends on the window size used, and the word-length. This number equals the number of inputs to a CSA tree. To compute a standard and modified CSA tree, let us notice first that the total number of full-adders does not depend on the configuration of the CSA tree and is equal to the number of inputs (i.e. the number of multiples of multiplicand to be added). The standard CSA tree has the smallest delay. On the other hand the layout of such a tree is difficult due to high irregularity of the
structure. However, in the modified CSA, the linear array part takes all the required inputs and can be laid out easily.

For a 64 bit multiplier with 4 bit scanning, the number of inputs needed is 22 (refer Equation 3.11). Comparing the optimum tree configurations with at least 22 inputs, which are shown in Fig.4.1 and Fig.4.2, we see that the delay in the modified CSA is 11 full adder delays versus 10 full adder delays in the regular CSA tree. Final accumulation of the product takes place in the adder at the base of CSA tree. This adder delivers on its two outputs an “a + b” representation of the result. Different multiples of multiplicand have to be delivered to the CSA tree, each shifted by the number of bits which correspond to the position of window, which was used to generate a given multiple. In the modified CSA tree, the final addition has to be carried out after the corresponding bits are brought together. This involves routing of the bits which may be away from each other by a number of adders. For instance in a 64-bit multiplier this routing occupies a large area.

4.2 OCMCSA

Though the MCSA has a regular layout the routing at the base is very irregular. To avoid the excess routing area, it is proposed to use overlapping columns. In the Overlapping Column Modified CSA (OCMCSA) architecture, the sum and the carry are sent diagonally down as shown in Fig.4.4. The layout shown is at bit level (m_j in
Figure 4.4: OCMCSA
the figure represents, $i^{th}$ bit of the $j^{th}$ multiplicand). This makes the columns of the modified CSA overlap each other, hence OCMCSA. This can be mapped to the 64-bit multiple-add-parallelogram as shown in Fig.4.5. The dark vertical lines in Fig.4.5 represent a column in the bit-slice layout. The maximum feed-through in this layout is four.

OCMCSA has a regular structure, and therefore a VLSI layout can be easily generated. The result of multiplication, which is twice the word-length, is to be taken out from both sides of OCMCSA area.

### 4.3 Estimation

Table 4.1 shows an estimated area and delay in different realizations of the multiplier structure for a 64-bit multiplier. Two modifying factors of the design are considered:

1. A number of scanning bits in the scanning algorithm

2. A number of columns in the linear array part of the modified CSA tree.
### Table 4.1: Time - area estimate of 64 bits multiplier

<table>
<thead>
<tr>
<th>Scan Size</th>
<th>CSA Array</th>
<th>Full Adder</th>
<th>Total Area</th>
<th>Scan Delay</th>
<th>CSA Delay</th>
<th>Total Delay</th>
<th>Time Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>Cols</td>
<td>Rows</td>
<td>Cols</td>
<td>Area</td>
<td>Rows</td>
<td>Cols</td>
<td>Area</td>
</tr>
<tr>
<td>1</td>
<td>14</td>
<td>8</td>
<td>252</td>
<td>260</td>
<td>0</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>6</td>
<td>220</td>
<td>226</td>
<td>0</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>4</td>
<td>182</td>
<td>186</td>
<td>0</td>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>1</td>
<td>62</td>
<td>2</td>
<td>149</td>
<td>151</td>
<td>0</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>8</td>
<td>115</td>
<td>131</td>
<td>2</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>6</td>
<td>108</td>
<td>120</td>
<td>2</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>4</td>
<td>90</td>
<td>98</td>
<td>2</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>2</td>
<td>81</td>
<td>85</td>
<td>2</td>
<td>35</td>
<td>2975</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>130</td>
<td>150</td>
<td>2</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
<td>83</td>
<td>99</td>
<td>2</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>4</td>
<td>68</td>
<td>80</td>
<td>2</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>2</td>
<td>64</td>
<td>72</td>
<td>2</td>
<td>24</td>
<td>1728</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
<td>79</td>
<td>107</td>
<td>4</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>70</td>
<td>94</td>
<td>4</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>60</td>
<td>80</td>
<td>4</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>2</td>
<td>59</td>
<td>75</td>
<td>4</td>
<td>20</td>
<td>1500</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>8</td>
<td>74</td>
<td>118</td>
<td>4</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
<td>74</td>
<td>114</td>
<td>4</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>72</td>
<td>108</td>
<td>4</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>2</td>
<td>68</td>
<td>100</td>
<td>4</td>
<td>17</td>
<td>1700</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>8</td>
<td>56</td>
<td>132</td>
<td>4</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>77</td>
<td>149</td>
<td>4</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>93</td>
<td>161</td>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>2</td>
<td>94</td>
<td>158</td>
<td>4</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>
In Table 4.1, the area is calculated in terms of a single bit full adder area taken as a unit. The delay is calculated in full adder delays. This table helps to choose an optimum window size. It can be seen from the table that the optimum window size for 64 bit multiplier in "a + b" representation is 10 which corresponds to $k = 4$. Note a significant drop in time-area product when the number of scanning bits is increased from 1 to 3. Then, the optimum time-area product as a function of the number of scanning bits levels off. So, in spite of having minimum time-area product at $k = 4$, a smaller window ($k = 3$) may be chosen for the reason of a simpler layout. The general structure of the multiplier is shown in Fig.4.6. In the "a + b" multiplication without clock (only $A$, $B$ and result registers are clocked) the multiplication is obtained in a single clock period. In such a case, CSA portion of the multiplier together with the storage and final addition stage is realized by a standard CSA tree. The clock (or the control timer for the multiplier) is set to allow for the complete signal propagation from the registers $A$ and $B$ to the Result register. The resulting multiplier would appear as shown in Fig.4.7.
CHAPTER 4. VLSI Consideration

Figure 4.6: General Structure of "a + b" Multiplier
Figure 4.7: Modified CSA With the Final Storage
In the preceding chapters, the "a + b" representation and supporting algorithms were presented. It was claimed that additions and like operations were faster, if "a + b" representation was chosen. So, according to the claims, if a processor uses "a + b" representation, it should be faster. To test this claim, a generic architecture was modified to use "a + b" representation. External to the processor, the environment never changes. Just the internal of the processor will be changed to use the "a + b" representation. If the new processor takes less time to complete the same task as the original processor, then the concept would be proven.

The test processor chosen for this purpose was a generic SPARC\textsuperscript{1} processor. SPARC's units are quite independent of each other which facilitated the modifications. SPARC is a pipelined RISC architecture which facilitated the time tracking for each instruction. SPARC is an open architecture that is maintained by The SPARC International and hence was not a proprietary one. All the above factors favored the decision to use SPARC as the test platform. A generic SPARC's framework was already written and distributed by Texas Instruments, Inc. As part of this experiment, this framework was further expanded to create two models of SPARC. They were then tested.

\textsuperscript{1}SPARC is a registered trademark of SPARC International, Menlo Park, California
Figure 4.8: Experiment Process

test.c → cc → test.o → ln → test (statically linked) → elf2memimg → indata → outdata1 → model (sparc) → outdata2

model (sparcab)
The process which was followed is depicted in Fig.4.8. A regular C program is compiled using the generic cc compiler. The output is used by ln to create a "statically linked file." This is used to generate the memory image of the process. This can now be used by the models to create the output memory image. Comparing the output memory image and timing information, both the models can be evaluated. The following chapters explain the two models, and the steps in the process in more detail. It is important to reiterate that the external of the processor, meaning the external of the model, will not be modified. This implies that both the models implement the same architecture. There are no extra instructions introduced. Any such modification would alter the result.

To understand the models and the experiment in its entirety, an understanding of SPARC and VHDL are mandatory. For this reason, and for the completeness of this thesis, the next two chapters touch upon SPARC and VHDL. Following these are descriptions of the framework and the experiment.
Chapter 5

SPARC

SPARC is an acronym for Scalable Processor ARChitecture. It was derived from Reduced Instruction Set Computer (RISC) architecture. SPARC is an architecturally driven standard enforcing binary compatibility between various implementations. SPARC was designed with a goal to facilitate the optimizing compilers. It was formulated by Sun Microsystems in 1985 based on RISC I & II from University of California at Berkeley (refer [25] and [26]). It is comprised of three main units: Integer Unit (IU), Floating Point Unit (FPU) and an optional Coprocessor (CP). There is also a recommended Memory Management Unit (MMU). The SPARC ISA (Instruction Set Architecture) does not mandate one particular implementation of all these units, they can be from different vendors.

5.1 Integer Unit

The Integer Unit (IU) is the main unit of SPARC. It has general-purpose registers and executes the arithmetic instructions and computes memory addresses for loads and stores. IU maintains the program counters and dictates the execution of FPU and CP.
5.1.1 Registers

The IU has in general a large number of registers, usually referred to as *Register File*. The number of \( r \) registers vary from 40 to 520, depending on the implementation. The registers are 32-bits wide. The registers are divided into two groups: a group of 8 **global** registers and the rest as a circular stack of 2 to 32 sets of 16 registers each, known as **register windows**. At any given time an instruction has access to the 8 global registers and a register window of 24 registers. The 24 registers are divided into 8 **in** registers, 8 **local** registers and 8 **out** registers. The current window into register file is dictated by the **Current Window Pointer** (CWP). Each window shares its **ins** and **outs** with adjacent windows. A “SAVE” increments the CWP \((CWP+1) \mod NWINDOWS\), which causes the 8 **out** registers of the current window to become the 8 **in** registers for the next window. A “RESTORE” decrements the CWP \((CWP-1) \mod NWINDOWS\), and hence the 8 **in** registers of the current window become the 8 **out** registers of the previous window\[25\] (refer Fig.5.1). This mechanism facilitates the parameter passing in function calls. The registers are addressed as shown in Table.5.1. The global register \( r0 \) is an exception. When this register is read, it always returns a 0 and any value written to it is ignored.

Apart from the general-purpose registers, IU has several control/status registers:

1. Processor Status Register (PSR)

---

\(^1\)NWINDOWS is number of windows
CHAPTER 5. SPARC

Figure 5.1: Circular Stack of Overlapping Windows

<table>
<thead>
<tr>
<th>Windowed Register Address</th>
<th>r Register Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>local[0]−local[7]</td>
<td>r[16]−r[23]</td>
</tr>
</tbody>
</table>

Table 5.1: Window addressing
CHAPTER 5. SPARC

5. Processor State Register

The PSR is a 32-bit register which contains various fields. This register can be modified by instructions like SAVE, RESTORE, Ticc, RETT and instructions that modify the condition flags. Privileged instructions, RDPSR and WRPSR, are also provided. These instructions read and write to the PSR directly. The various fields of the PSR are shown in Fig. 5.2.

impl: This field is hardwired to indicate the implementation or class of the implementations of the architecture.
**CHAPTER 5. SPARC**

**ver:** This field is either hardwired or written to identify the one or more particular versions.

**icc:** These are condition codes that are modified by the arithmetic or logical instructions whose names end with letters **icc.** The bits are:

- **n:** Bit 23, set if the result of the operation was negative. 1 indicates negative.
- **z:** Bit 22 is set if the result of the last operation was zero. 1 indicates zero.
- **v:** If one, there had been an overflow. This is bit 21.
- **c:** Bit 20, indicates if there was a carry or borrow.

Not all instructions affect these flags.

**reserved:** These bits are reserved. **RDPSR** (read PSR) returns zero in this field and **WRPSR** (write PSR) ignores this field.

**EC:** This bit is 1 if coprocessor is present and is enabled.

**EF:** This bit is 1 if floating point unit is present and is enabled.

**PIL:** This identifies the trap level above which the processor will accept.

**S:** This bit, if 1 indicates the supervisor mode and zero indicates user mode.

**PS:** This bit has the value of the S bit at the time of the most recent trap.

**ET:** If 1, enables the trap. When zero, the traps are ignored.
CWP: This is a counter that identifies the current window in the register file.

5.1.1.2 Window Invalid Mask

This is register can be written only in supervisor mode (S bit set). The individual bits are set by SAVE and unset by RESTORE. An active state of a window $n$ is indicated by $\text{WIM}[n]$ being set. This is a 32 bit register and hence allows implementations of upto 32 windows.

5.1.1.3 Trap Base Register

This register provides the address to which the control will be transferred when a trap occurs. This has three fields as shown in Fig.5.3.

TBA: This has 20 most significant bits of the trap table address.

$tt$: This provides an offset into the table.

$\text{zero}$: The last 4 bits are always zero.

The TBA field must be written by the supervisor software. When a trap occurs, a eight bit value that uniquely identifies the trap is written into the $tt$ field by the hardware. And this address is used to transfer the control.
5.1.1.4 Multiply/Divide Register (Y Register)

This 32 bit register, holds the most significant bits of the result after multiplication. While in division it holds the MSBs of the dividend. Apart from that, it may hold the intermediate result.

5.1.1.5 Program Counter

PC holds the 32-bit address of the current instruction being executed by the IU.

5.1.1.6 Ancillary State Register

There are 32 Ancillary registers reserved for future use and are not supposed to be used by any software. This is an implementation dependent feature.

5.1.1.7 IU Deferred-Trap Queue

This helps in resuming the deferred trap. This is also implementation dependent.

5.1.2 Instructions

All SPARC instructions are 32-bits long. The instructions can only have any of the three formats shown in Fig.5.4[26]. In Fig.5.4:

a: The annul bit is used in branch instructions to control the execution of a delay instruction.
### Figure 5.4: Instruction Formats

**CALL**

<table>
<thead>
<tr>
<th>Format 1</th>
<th>op</th>
<th>disp30</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**SETHI and Bicc**

<table>
<thead>
<tr>
<th>Format 2</th>
<th>op</th>
<th>rd</th>
<th>op2</th>
<th>imm22</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 25 24 22 21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>a</th>
<th>cond</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 25 24 22 21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Other Instructions**

<table>
<thead>
<tr>
<th>Format 3</th>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=0</th>
<th>asi</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 25 24 19 18 14 13 12 5 4 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i=1</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 25 24 19 18 14 13 12 5 4 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>opf</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 25 24 19 18 12 5 4 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
asi: The address space identifier is an eight bit field used in load/store instructions.

cond: This field identifies the various condition codes for branch instructions.

disp22: This field is sign extended to give the full 32-bit address for control transfer in branch instructions.

disp30: Shifting this field to the left by two bit positions gives address for control transfer in call instructions.

i: \( i = 0 \) implies that the second operand in an arithmetic instruction is a register, while \( i = 1 \) implies that the second operand is an immediate value.

imm22: This field is the 22-bit constant used in \texttt{SETHI} instructions.

op: The op field identifies the format of the instruction. The various values are shown in Table.5.2.

op2: The various values are shown in Table.5.3.

op3: This field distinguishes the various arithmetic, logical and shift instructions.

opf: This is the opcode for the floating point processor and the coprocessor.

rd: Identifies the destination register.

rs1: Identifies the first operand register.

rs2: Identifies the second operand register, if any.
CHAPTER 5. SPARC

<table>
<thead>
<tr>
<th>op Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Bi cc, FBfcc, CB ccc, SETHI</td>
</tr>
<tr>
<td>01</td>
<td>Call</td>
</tr>
<tr>
<td>10</td>
<td>Other (arithmetic, logical, shifting etc.)</td>
</tr>
<tr>
<td>11</td>
<td>Other</td>
</tr>
</tbody>
</table>

Table 5.2: op field coding in SPARC instructions

<table>
<thead>
<tr>
<th>op2 Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UNIMPlemented</td>
</tr>
<tr>
<td>001</td>
<td>UNIMPlemented</td>
</tr>
<tr>
<td>010</td>
<td>Bi cc</td>
</tr>
<tr>
<td>011</td>
<td>UNIMPlemented</td>
</tr>
<tr>
<td>100</td>
<td>SETHI</td>
</tr>
<tr>
<td>101</td>
<td>UNIMPlemented</td>
</tr>
<tr>
<td>110</td>
<td>FBfcc</td>
</tr>
<tr>
<td>111</td>
<td>CB ccc</td>
</tr>
</tbody>
</table>

Table 5.3: op2 field coding in SPARC instructions

**simm13**: This 13-bit immediate value will be sign extended to form the second operand.

The SPARC instructions fall into six basic categories:

1. Load/Store
2. Arithmetic/logical/shift
3. Control Transfer
4. Read/Write control register
5. Floating Point operate
6. Coprocessor operate
5.1.2.1 Load/Store

The load/store instructions use two registers or a register and a signed 13-bit immediate value to calculate the address. ASI² is always appended to the address to distinguish between USER and SUPERVISOR mode. The destination can be an r register (within IU), f register (an FPU register) or a coprocessor register. The are versions to load and store 16-bit, 32-bit and 64-bit data. But, the addresses have to be aligned on 2-byte boundaries, 4-byte boundaries and 8-byte boundaries respectively. The address produced is the address of the most significant byte. There are also versions to load from or store at alternate space. These are privileged instructions and are allowed only in supervisor mode.

5.1.2.2 Arithmetic/Logical/Shift

These instructions use two operands and the result is always put in a register. The operands can be contents of two registers or a register and an immediate value. But, one instruction SETHI uses an immediate value to set the 30 most significant bits of a register.

The multiplication instructions UMUL (unsigned multiplication), SMUL (signed multiplication) etc are all optional. There is “Multiply Step” instruction, MULSec that does a single step of a multiplication. The 64bit product is stored in two registers:

²Address Space Identifier
MSBs in a register and LSBs in Y register. \texttt{MULScc} takes in two source registers \((r[rs1] \text{ and } r[rs2])\) and a destination register \((r[rd])\) as parameters. The operations performed can be listed as follows[25]:

1. A 32-bit value is computed by shifting \(r[rs2]\) right by one bit with \texttt{norv}, which is the right sign of the previous partial product.

2. If the LSB of Y is 1, then the shifted value is added, else 0 is added.

3. The above sum is written to \(r[rd]\).

4. The condition codes, especially, n and v are updated according to the addition performed above.

5. The Y register is shifted right by one bit, with the LSB of the unshifted \(r[rs1]\) replacing the MSB of Y.

There are also instructions to do tagged arithmetic. These instructions assume the least two bits as "tags." The overflow bit is set if there was an arithmetic overflow or if any of the operand’s tag bits are nonzero.

\subsection*{5.1.2.3 Control Transfer}

The control transfer instructions affect the program counter and hence alter the program flow. These instructions are: CALL, branch (Bicc), JMPL and Trap instructions. Since SPARC is a pipelined architecture, it can delay the transfer of control
to the next instruction. The instruction following the control transfer instruction, called delay instruction, is always fetched. But the annul bit in the control transfer instruction controls the execution of the delay instruction (refer Fig.5.4). The branch and call instructions use an immediate value to transfer the control. The JMPL instruction uses an register indirect and the trap instruction uses a table (refer Fig.5.3).

5.1.2.4 Read/Write control register

These instructions read and write the control registers: PSR, Y, TBR, WIM, ASR. The valid bit positions are alone updated. The invalid positions return zero when read. These instructions, except RDY and WRY are mostly privileged instructions; they can be updated only in supervisor mode (refer Fig.5.2). They are implementation dependent.

5.1.2.5 Floating Point/Coprocessor Operate

These instructions are register-to-register instructions. They execute a function with one or two source registers and the result is put in a register. The functions are identified by the opcodes.

A listing of all the SPARC instructions and their opcodes can be found in [25] and [26].
5.2 Floating Point Unit (FPU)

The FPU specified is used to do floating point calculations. It has thirty two 32-bit \( f \) registers. Pairs of these registers can be used to hold a double-precision floating point value. They must be an even-odd pair. Four continues registers can hold a quad-precision floating point unit. Thus the FPU can hold a maximum of 8 quad precision or 16 double precision or 32 single precision values.

The floating point instructions mainly use a register to compute a value based on a single register or two registers. Data can be moved directly between \( f \) registers and memory, but the address is calculated by the IU. The loading of the registers waits till all the fpop instructions to finish and then the data is moved.

The floating point instructions can be grouped into the following groups:

1. Conversion Instructions — These instructions convert an integer to single, double or quad precision floating point value or vice versa. Also there are instructions to convert from one precision to the other.

2. Move instructions — These instructions move data between the \( f \) registers itself.

3. Add and Subtract instructions

4. Multiply and Divide instructions

5. Compare Instructions
The floating point data formats confirm to the IEEE Standard for Binary Floating-point Arithmetic, ANSI/IEEE Standard 754-1985. If the FPU is not present, EF bit in PSR will be zero and execution of any fpop instructions causes the fp_disabled trap to be executed. Even if the FPU is present, the software is expected to emulate those instructions not present in the hardware and when FPU is not present, the software must emulate all the FPU instructions.

5.3 Coprocessor Unit (CP)

The CP is implementation dependent. The number of registers is also implementation dependent. In general for every FPU load/store instruction there is an analogous CP instruction. The Cpop instructions are defined by the implementation and hence the configuration.

If the CP is absent EC bit in PSR is zero, and execution of any CP instruction causes an cp_disabled trap.

5.4 Data Formats

SPARC uses three data formats:

1. Signed Integer — 8, 16, 32 and 64 bits

2. Unsigned Integer — 8, 16, 32 and 64 bits
3. Floating Point — 32, 64 and 128 bits

The various format widths are:

1. Byte — 8 bits
2. Halfword — 16 bits
3. Word — 32 bits
4. Tagged Word — 32 bits
5. Doubleword — 64 bits
6. Quadword — 128 bits

SPARC is a big endian architecture — the address of a word, half-word or a double word, is always the address of the most significant byte.

5.5 Memory Model

SPARC just defines semantics of memory operations such as load/store instructions. The model applies to both uniprocessors and shared memory multiprocessors. The standard memory model is called Total Store Ordering (TSO). This model guarantees that the store, FLUSH, and atomic load-store instructions of all processors execute serially in single order called “memory order.” The stores, FLUSHes and loads issued by a processor are stored in FIFO buffer called Store Buffer. This buffer is also used
**Signed Integer**
- **Byte**
  - 7:6:0
- **Halfword**
  - 15:14:0
- **Word**
  - 31:30:0
- **Double Word (SD0)**
  - 31:30
  - signed integer[62:32]
- **Double Word (SD1)**
  - 31:30
  - signed integer[31:0]

**Unsigned Integer**
- **Byte**
  - 7:0
- **Halfword**
  - 15:0
- **Word**
  - 15:0
- **Tagged Word**
  - 31
- **Double Word (UD0)**
  - 31:30
  - unsigned integer[62:32]
- **Double Word (UD1)**
  - 31:30
  - unsigned integer[31:0]

**Floating-point**
- **Single**
  - 31:30
  - exp[7:0]
  - fraction[22:0]
- **Double**
  - 31:30
  - exp[10:0]
  - fraction[51:32]
- **Quad**
  - 31:30
  - exp[14:0]
  - fraction[111:96]

---

**Figure 5.5: Data Formats**
as a cache; in the event of a load the buffer is checked for a store and if present, the most recent value is returned.

The Partial Store Ordering (PSO) guarantees that the store, FLUSH, and atomic load/store execute serially in a single order called memory order. Unlike TSO, PSO does not guarantee the execution of these instructions in the order issued by the processor.

All SPARC processors are expected to implement TSO and may or may not support PSO.

5.6 Reference Memory Manager Unit

The reference Memory Manager Unit suggests an management architecture for use with SPARC processors. The design is such that, it can support a large number of processes running on a single processor. The design also considers a single chip implementation of the MMU and at the same time offer a wide variety of features. Some of the features offered by the MMU are:

1. 32-bit virtual address.
2. 36-bit physical address.
4. Support a sparse address space with a 3-level map.
5. Support a large linear mappings.


7. Hardware miss processing.

The MMU is usually placed between the IU and the memory (Fig. 5.6) and does the address translation between the two. A SPARC Reference MMU provides three primary functions[25]:

1. Translate the virtual address of each running process into a physical address.

   This mapping is done in units of 4K-byte pages. Any virtual page can be mapped into any physical page.

2. Provide memory protection. A process cannot read or write to an address space of another process.

3. Implement virtual memory. The page table tracks the pages that are in main memory and when a memory location that is not in memory is addressed, a page fault is signalled.
Chapter 6

VHDL

This chapter is just an introduction to VHDL to aid the understanding of the chapters that follow. VHDL is an acronym for VHSIC Hardware Definition Language and VHSIC stands for Very High Speed Integration Circuit. This was developed as a result of The Department of Defense's (DOD) initiative. Some of the features of the VHDL are:

1. It can be used for documenting hardware designs.

2. The language itself is generic and allows various levels of abstraction.

3. The design description can be compiled to be used by a simulator and hence can be tested.

4. The compiled code can be fed to a synthesizing tool to get the actual hardware.

This section serves as an introduction to help explain the framework that follows. For further reading refer [1], [11] and [15].
6.1 VHDL Designs

VHDL has the following building blocks:

1. Library
2. Package
3. Entity
4. Architecture
5. Configuration

6.1.1 Library

When a VHDL code is compiled, the results are put in the Library. The simulation tools, the synthesis tools and like use the library to take their input from. This also allows sharing and modular development of a complex design.

6.1.2 Package

This is another level of hierarchy within a library. The packages contain constant declarations, sub-program declarations, type declarations and other related information. The packages provide a way of extending VHDL languages through STANDARD packages, vendor specific packages and user developed packages. Package is a compilable unit and once compiled into the library, all the utilities, declarations, etc., can
be called by using the "USE" statement.

6.1.3 Entity

This is the next level of hierarchy next to package. The entity specifies the input/output connections. The specifications also include the directions. This way, the internal design is separated from the external interface. Once an entity is compiled, it can be used as a component in another design.

6.1.4 Architecture

This defines the internals of an entity. This also defines the function of the entity. There are three styles to describe an entity:

Behavioral: This is a sequential description of a process.

Dataflow: Which implies a structure and behavior. Mostly implements a state diagram.

Structural: This uses several components and describes interconnection between them.

There can be more than one architectural description of an entity.
6.1.5 Configuration

This ties a particular architecture to an entity. This allows the designer to test various
designs of an entity.

6.2 Data types

VHDL is a strongly typed language. This assists the designers to catch errors early
in the development cycle. VHDL has a number of data types and operators. The
STANDARD package provides the behavioral data types and operators, while the
IEEE 1164 package provides the simulation types and operators. Basic data types
provided are: character, string, bit, bit_vector, std_logic, std_logic_vector, boolean,
real, integer and time. VHDL also provides ways to expand these types by means of
advanced data types. They are:

1. Enumerated types: This allows user defined types and sub types which are
   ranges.

2. Composited types: This comprises of arrays and records.

3. Other predefined types like File and Lines.
6.3 Statements

VHDL statements can be grouped into *concurrent statements* and *sequential statements*. Some statements can be used in both the groups. The concurrent statements are executed in parallel by the simulator. One of the primary concurrent statements is PROCESS. Within a process block, the individual statements are executed sequentially. The sequential statements can be:

1. ASSERT statement
2. CASE statement
3. EXIT statement
4. IF statement
5. LOOP statement
6. NEXT statement
7. WAIT statement
8. Sub-Programs and Functions
9. Variable assignment and sequential signal assignment statements

IF-THEN-ELSE, CASE, LOOP and NEXT statements are usually found in any programming language. The WAIT statement controls the suspension of execution of a
process. The processes can call functions and procedures.

There are special functions called *resolution* functions which are called to resolve values. The most common situation would be a bus. Multiple signals are connected to the bus. But at any given time, the bus can have only one value. To do so, this function is called to "resolve" the situation and output one value.

Concurrent statements are executed asynchronously with respect to one another, and at the same simulated time. The concurrent statements are:

1. Process statement

2. Concurrent Signal assignment

3. Conditional Signal assignment

4. Selected Signal assignment

5. Concurrent procedure call

6. BLOCK statement

The blocks help create an hierarchy in concurrent sections of an architecture.

### 6.4 Signals

Signals provide communication between processes. Signals also help in wiring components together. Signals assignments are not done immediately and are scheduled.
Within a process, a signal assignment is never executed until a **WAIT** statement. They act like global variables, but have very unusual behavior. The signal assignments can be explained with the help of simulation cycle (Section 6.5).

### 6.5 Simulation

The present generation of VHDL simulators use *two-list* algorithms as opposed to *one-list* algorithms used by earlier generation. The two-list algorithms allow the following kind of assignments:

\[
A <= B;
\]

\[
B <= A;
\]

With zero delay, A and B exchange values as opposed to both A and B having the same old value of B. While simulating, the values of A and B are retrieved and scheduled for assignment and executed after a WAIT statement. The zero delay events are executed using a fictitious time “delta time.” The execution of a WAIT statement causes the simulator to enter the simulation cycle. At *enter* time, the signal assignments with previously computed values and scheduled values are executed. Now the cycle is in *middle* time. If any process become active, they are activated. Variable assignments are executed. All the right hand side values are computed and if need be, scheduled.
At this time the cycle returns to begin. This step is increment of delta time. But the simulation time is not incremented. The cycle continues until there is no change of values. At which time, the simulation cycle is at leave time. At this point the simulation cycle time is incremented.
Chapter 7

The Framework

Texas Instruments, Inc., implemented SPARC framework in VHDL and used it to test its processor. This section describes the framework. Much work was done on the framework to port it to work for Mentor Tools. It was further enhanced to stay close to the SPARC standard and as well reflect the simulation environment. Also considerations were taken so that the “a + b” architecture can be easily accommodated. In the previous chapters an overview of SPARC and VHDL were presented. The materials presented in them will be referenced henceforth. Also refer to the typographical standard adapted in this thesis (Section 1.2).

7.1 The Model

The model implemented in the frame work is best described in Fig.7.1. The processor is directly connected to the Memory Management Unit. The MMU manages the cache and the main memory. The address bus, data bus are all implemented as signals connect the two modules. In the sections that follow both the modules are explained in detail.
7.1.1 Signals

SPARC does not define any signals or outputs that are interfaces to the outside world. Since SPARC is an Instruction Set Architecture, the signals are not part of the definition; they are implementation dependent.

For a better understanding of the model, the signals which are the interface between the modules, are examined here.

7.1.1.1 abus

This is the address bus. The SPARC address is 32 bit. This is virtual address. The MMU converts the virtual address to the physical memory address.

7.1.1.2 cbwe

This is Cache Byte Write Enable, a four bit active low signal. This signal is usually generated by the MMU for the cache ram. In this implementation it is generated by the Integer Unit. It is generated based on the size signal and abus (1:0). This determines which byte of the word is written to the cache: cbwe (3) controls the
least significant byte while \texttt{cbwe (0)} controls the most significant byte.

7.1.1.3 clock

This usually an input, but in this implementation this is generated by the IU and is an output.

7.1.1.4 dbus

The databus is usually 32 bit wide. This is usually a bidirectional bus. But to ease the implementation this is only output.

7.1.1.5 dbusi

This is the input counterpart of the \texttt{dbus}.

7.1.1.6 dxfer

This is data transfer which determines the type of data requested by the IU: instruction or data. This is asserted during the address cycles for instruction fetches.

7.1.1.7 end_reached

This signal is specific to this implementation. This signals the MMU to dump the memory to a file and end the simulation.
7.1.1.8 inst

This is instruction fetch. This usually an output of IU and asserted when a new instruction is fetched. In this implementation this is an output from MMU and input to IU. This is asserted when an instruction is put on the data bus by the MMU.

7.1.1.9 mao

Memory Address Output signal is an input to the integer unit and asserted by MMU under the MHOLD condition. This causes the IU to place all the signals including the address, asi, size, rd, we, and dxfer again. The values of these signals should be the same one that was one cycle before the cache-miss.

7.1.1.10 mds

Memory data strobe is an active low input asserted by the MMU. This tells the processor that the data for which the cache miss occurred is now on the data bus. This can be asserted only when the mhold is active.

7.1.1.11 mhold

Memory hold is an active low signal also asserted by the MMU. When mhold is asserted, the pipe is frozen in IU. This signal is used by the memory to signal the cache miss.
7.1.1.12  rd

This is read access signal, another output from IU. When asserted signifies that the current memory access is read. For write access, rd is pulled low.

7.1.1.13  size

This is a two bit signal. This is an output that identifies the size of the data being transmitted during a fetch cycle. The values denote whether a byte, halfword, word or a double word is transmitted.

7.1.1.14  we

Write enable is an active low signal. This is also an output that is asserted by the IU to signify that the data to be written to memory is on the data bus.

7.2  Organization

In the previous section, a high level model of the implementation and the signals between the modules were delineated. In what follows the implementation code is explained. After modifications, the framework is laid out in five files:

1. datasent.vhd

2. memsent.vhd
CHAPTER 7. THE FRAMEWORK

3. opcodes.vhd

4. sparcsent.vhd

5. systemsent.vhd

7.2.1 datasent.vhd

SPARC is a 32-bit alu. To provide the basic capabilities like addition, subtraction, logical functions, and other operations on 32-bit vectors, separate functions have to be written. The framework was extended to supply many valuable functions. These functions also provide a single point to evaluate the systems performance. On synthesis these functions can be converted to logical gates or a VLSI layout.

Following is a short description of all these functions.

7.2.1.1 "+"

This function is overloading the "+" operator. This adds two 32-bit words. Also, there are versions for adding an integer with a 32-bit word.

7.2.1.2 

These are overloaded for subtraction, multiplication and integer division of two 32-bit words. There are also versions for doing these operations with integers.
7.2.1.3 bitval \( (x) \)

This function converts an integer \( (x) \) to a 2's complement 32-bit word.

7.2.1.4 busfw \( (x) \)

This is a resolution function to resolve values on buses. This returns a non-zero word, if any exits, else returns a zero. It also returns zero, if there are no values assigned or, if all the assigned values are zero.

7.2.1.5 byteval \( (x) \)

This function converts an integer \( (x) \) to a byte.

7.2.1.6 gt \( (x, y) \)

This function takes two 32-bit words, \( x \) and \( y \); compares them and returns a “true,” if \( x \) is greater than \( y \). This function is overloaded to take an integer value as \( x \) and \( y \).

7.2.1.7 gte \( (x, y) \)

Same as gt. This implements greater than or equal to.

7.2.1.8 int \( (x) \)

Converts a bit vector (series of bits; a basic VHDL data type) to an integer.
7.2.1.9  intval (x)

This function converts a 32-bit word (x) to an integer.

7.2.1.10  lt (x,y)

This is similar to gt but implements less than.

7.2.1.11  lte (x,y)

This is similar to gt but implements less than or equal to.

7.2.1.12  shift_left_logical (data1,shcnt)

This shifts the bits of the word (data1) to the left by shcnt positions.

7.2.1.13  shift_right_arithmetic (data1,shcnt)

This shifts the bits of data1 to right by shcnt positions. The “overflowing” bits are wrapped around; in other words, this does a circular right shift.

7.2.1.14  shift_right_logical (data1,shcnt)

Same as shift_left_logical, but instead of shifting the bits of data1 to the left, they are shifted to the right.
7.2.1.15  sign_extend (x, y)

Extends a bit_vector to a 32 bit word (x), treating the y\textsuperscript{th} bit as the sign bit. The sign bit is used for padding.

7.2.1.16  wirednor (x)

Resolution function that returns '0' if any of the arguments are zero else returns a '1'.

7.2.1.17  wiredor (x)

Also a resolution function returning '1' if any of the arguments are '1' else returns a zero.

7.2.1.18  zero_extend (x)

Extends a bit_vector (x) to a 32-bit word by padding with zeros.

Apart from the functions, datasent.vhd also has some type definitions. They are:

1. type busarg is an array (integer range <>) of word.

2. type double32 is an array (1 downto 0) of word32.

3. type or_vector is an array (integer range <>) of wiredor_bit.

4. type size_type is an array (1 downto 0) of wiredor_bit.
5. subtype byte is a bit vector (7 downto 0).

6. subtype wiredor_bit is wiredor bit. To read any variable of this type, the resolution function wiredor will be called.

7. subtype word is word32.

8. subtype word32 is a bit vector (31 downto 0).

In the above list, <> stands for open ranges, a feature of VHDL.

7.2.2 memsent.vhd

This file defines the memory subsystem. Also in this file is the description of a primitive memory manager, one of the main units in SPARC (refer section 5.6).

The Memory implemented is a large array of bytes. Since a UNIX process is seen always loaded in memory in the range 65656 and 656560 (refer section 9.3), the physical memory is an array of bytes with the same range.

The cache implemented is a read-through and a write-through cache. That is:

1. If the word needed is available in the cache, it read from the cache. If the word is not in cache, the a memory fetch is done to bring the word from memory to cache, and then put on the data bus. This is read-through cache.

2. When a word is to be written to memory, both the memory and cache are updated at the same time. This is write-through cache.
There are two procedures: load and unload. These two procedures load and unload the memory. The format of the files that they load from and unload to are described in Section 9.3.2. The load procedure is called during the first clock cycle and the unload procedure is called when the end_reached signal goes high. This signal is controlled by the clock process of the processor. This is more like a core dump — an exact image of the memory is written to a file. The unload procedure is valid iff the cache is a write-through cache.

The memory system is itself modeled by a single VHDL process — meminterface.

The actions performed by the process can be listed as follows:

1. When the simulation starts, init would be initialized to false and if it is false, load is called.

2. unload is called when end_reached goes high and the simulation is stopped.

3. The process then waits for clock to change.

4. If the clock is high then the memory responds to a read:

   (a) cachemiss by default is false and mhold is high (refer section 7.1.1).

   (b) The cacheaddress is calculated from addr. If the tag of the cache (cacheaddress) is same as addr then an hit has occurred; else a miss.

   (c) On a hit, the cache (cacheaddress).data is written to dbusi.

   (d) On a miss there are several steps the MMU has to carry-out.
i. The \texttt{mhold} is brought low which would cause the pipe in IU to freeze.

ii. \texttt{mao} is also brought low.

iii. In the next clock cycle, \texttt{mao} is brought high. This strobing of \texttt{mao} would cause the address and other signals to be placed again (refer section 7.1.1).

iv. In the next clock cycle, the data is read into cache and also placed on the data bus, \texttt{dbus}. \texttt{mao} is brought low. \texttt{mds} is made active to indicate the availability of data to IU.

v. In the next clock cycle, the signals \texttt{mhold}, \texttt{mao} and \texttt{mds} are brought back to their normal values.

5. When \texttt{clock} is low, the MMU responds to a write:

(a) Like in the read cycle, the \texttt{cacheaddress} is calculated.

(b) There is no check for cache miss or a hit. Depending on \texttt{size}, the appropriate cache word and \texttt{ram} is updated.

To compile the memory datasent.vhd has to be compiled first into the library. The \texttt{meminterface} process can be simulated separately, but the responses to the signals (like putting \texttt{addr} etc. for \texttt{mao} strobing) warrants some automation. To lower the implementation cost for MMU, some complicated state machine and confusing variables are introduced in the code. The reference MMU recommended by SPARC
(Section 5.6) has many other functionalities. These are not implemented in this model and are out of scope for this thesis.

7.2.3 opcodes.vhd

This file can be described as collection of all the constants in all SPARC instructions. It associates a SPARC pneumatic to the various ops (op1, op2 and op3) and cond\(^1\) of a SPARC instruction. These pneumonics are used in sparc sent.vhd and they make the code more readable.

opcodes.vhd has to be compiled into the working library before sparc sen t.vhd.

7.2.4 sparc sent.vhd

This file contains the bulk of the code for the implementation. This describes the integer unit of the implementation. SPARC has register files and processor state registers (refer section 5). The register file is implemented as two arrays:

1. The global registers as an eight element array.

2. The rest as array of \(n_{\text{registers}}\) elements.

The processor state registers are implemented as separate signals.

The IU is implemented by means of several processes and signals interconnecting these processes. Apart from the processes, there are some procedures as well. First,
CHAPTER 7. THE FRAMEWORK

let us look at all the processes at a high level. Then we shall look into some of the important procedures. These procedures carry out certain tasks and some of them are local to the processes. We shall then have a closer look at some core processes.

The instructions flow in a pipeline. At various stages in the pipeline certain actions are performed for the instruction. The main pipeline stages in order are:

1. Fetch — Fetches the instruction.

2. Decode — Decodes the instruction. The operands are read and when needed the next instruction address is calculated, for example, branch and jump instructions.

3. Execute — Executes the instruction. The results are saved in temporary registers.

4. Write — Writes the results to the destination register.

All the stages execute in parallel and hence the maximum throughput can be one instruction per cycle.

In SPARC almost all instructions are single-cycle instructions, in that all the instructions are executed in one clock cycle. But there are some multi-cycle instructions that extend beyond one cycle, like load/store. The processor is usually faster than memory, hence any interaction with memory may take more than one cycle. To cope for this load dependency, an internal iop is introduced. This can be considered as a
filler. When this passes through the various stages in the pipe, the stages ignore this (acts like a NOP).

7.2.4.1 saveit

This process saves the state of the busses, so that on mhold the address of the memory access that caused mhold can be returned to the bus temporarily. Also, after the mhold, the current value has to be put back. The process waits on abus, i.e., the process is activated only on changes in abus. This reads the value of abus, size, rd and dxfer into current_abus, current_size, current_rd and current_dxfer respectively. The values that were put in previous cycle are transferred to previous_abus, previous_size, previous_rd and previous_dxfer respectively. By this the values are saved for two cycles.

7.2.4.2 clkgen

Generates clock signal. Apart from that, it also asserts end_reached signal when “end of simulation address” is reached. This uses a global signal end_addr. This is set by fetch on init. This address is compared to the pc in every clock cycle. If they match then end_reached is asserted. For the clock, the NOT of clock is TRANSPORT\textsuperscript{ed} every PERIOD/2 seconds.

\textsuperscript{2}TRANSPORT is feature of VHDL. This is well explained in [23]
7.2.4.3 restore

Temporarily restores the previous value of the busses in order to do cache misses. The mao and mds signals, set in the memory unit, control whether the current address or the previous address is on the bus (refer section 7.1.1). This occurs when the processor is stalled with mhold. When mao is asserted, the oldest value that was stored in saveit is put back. When mds is asserted the current value (values stored in current_abus, current_size, current_rd and current_dxfer) is put back.

7.2.4.4 fetch

This implements the fetch stage in the pipeline. It fetches the next instruction from memory. This process outputs the address. This is explained in detail later in Section 7.2.4.19.

7.2.4.5 decode

This implements the decode stage in the pipeline and hence the instruction is decoded by this process and instruction’s operands are read. The opcode and operands are passed in buffer registers to execute. The decode is explained more clearly later in Section 7.2.4.20.
7.2.4.6 **execute**

Executes and saves results in temporary words. This process also asserts the required signals while executing `load` and `store` instructions. This is also examined more closely later in this chapter in Section 7.2.4.21.

7.2.4.7 **write**

This implements the write stage in the pipeline. This writes the temporary values calculated by the execute stage to registers. On start of simulation, this reads in the value of the stack from the input file. This is assigned to register 14 which is the stack pointer (refer section 9.2). This process calls the procedure `wreg` to write the result to the register. The destination register is given by `destination_buffer` and the result is in `result_buffer`. The `destination_buffer` is calculated by `decode` and the result is calculated by `execute` and passed on through `result_buffer`.

7.2.4.8 **rreg (gpr, address)**

Returns the content of a register. This also takes care of the Current Window Pointer (CWP) when reading the register. The register specified must be in the range 0 – 31. If the address is in the range 0 – 7, then the word requested is in the global register array `g`. Else, based on the `cwp`, the correct register is calculated as:

\[
\text{register address} = \{(\text{given address} - 8) + (\text{cwp} \times 16 - 8)\} \mod \text{nregisters} \quad (7.1)
\]
The above equation ensures the circular stack of registers.

7.2.4.9  \textit{wreg (gpr, address, data)}

The Write counter-part of \textit{rreg}. This writes a word into the specified register. The address is calculated the same way as for \textit{rreg}.

7.2.4.10 \textit{unimplementedinstruction (op,op3,op2, i)}

This checks for illegal instruction and stops the simulation. It is a straight forward boolean expression.

7.2.4.11 \textit{eval\_cond\_code (z,n,v,v,code)}

This is local to the process \textit{decode}. This function evaluates the flags in PSR for the specified condition code of a branch instruction. Returns a true if the condition is satisfied. The conditions are explained and specified in [25].

7.2.4.12 \textit{set\_add\_cc (result)}

Sets the condition flags, N,Z,V and C for an addition. This is called by ADDcc, ADDXcc etc., to set the flags after the operation. This procedure is local to \textit{execute}. N equals \texttt{result (31)}. Z is one if \texttt{result} is zero. V is set, if both the operands are positive (MSB is 0) and the result is negative (MSBs are zero) or vice versa. C is set if both the operands are negative or if the result is positive and either of the operand
is negative. After setting the flags, bcc_set is asserted. This is used by decode.

7.2.4.13  set_sub_cc (result)

Sets the condition flags, N,Z,V and C for a subtraction. This is called by SUBcc, SUBXcc etc., to set the flags after the operation. This procedure is local to execute. The setting of N and Z are similar to addition(set_add_cc). The V flag is set if first operand is positive, the second operand is negative and the result is negative or the first operand is negative, the second operand is positive and the result is negative. C is set if the first operand is negative and second operand positive or if the result is negative and neither of the operands negative. Again after setting the flags, bcc_set is asserted, so that it can be used in decode.

7.2.4.14  set_logical_cc (result)

Sets the condition flags, N,Z,V and C for a logical operation. This is called by ANDcc, ORcc, etc., to set the flags after the logical operation. This procedure is local to execute. The V and C are always set to zero. The N and Z are set the same way as for addition(set_add_cc).

7.2.4.15  extract_byte (addr, dat)

Extracts the correct byte when a load instruction is called. This is primarily called to load the data from the data bus. This procedure is local to execute.
7.2.4.16 perform_load ()

This procedure operates on the iop. The load instruction takes more than one cycle. This procedure makes sure that during the first cycle, the address is on the address bus and during the next cycle data is taken off the dbus. This procedure is local to execute. After setting the flags, bcc_set is asserted. This is used by decode.

7.2.4.17 perform_store ()

This procedure ensures the signals are set properly for the store cycle, especially, cbwe. This makes sure the address and data are put on the first cycle of the operation and during the second cycle gives up the address and data busses. This procedure is local to execute.

7.2.4.18 loadstack (st)

This loads the value of the stack register (st) from the input file (refer section 9.3.2). This is local to the process write.

Next, we shall look at each of the process in detail. Some of the processes interact with each other by means of signals. These signals will be mentioned where appropriate.
7.2.4.19  **fetch** — In detail

The main purpose of this process is to implement the fetch cycle in a SPARC pipeline.

**fetch** will not execute if:

1. A cache miss has occurred and **mhold** has been asserted by an external source.

2. A load dependency\(^3\) has occurred, and the decoder has a load dependency iop instruction. This is indicated by the **op3_buffer** signal.

3. A **branch** instruction, a **jmp1** instruction, or a **call** instruction is being decoded and is going to preempt the use of **abus**. This is indicated by the **transfer**\(^4\) set by the **decode** process. Setting **transfer** signal to true, stops **fetch** from incrementing the **pc** and putting that on **abus**.

When **clock** = 0, the address is placed on the address bus. And, when **clock** = 1, the data corresponding to the address that was placed in the previous cycle is read.

On initialization, **fetch** reads the starting address of the program to simulate. This is directly assigned to **pc**. Apart from this, the **end_addr** is also read from the input file (refer section 9.3.2 for the file format). The exit address is used by **clock** to assert the **end_reached** signal.

\(^3\)Refer page 80

\(^4\)**transfer sig** is used to tell **fetch** that the value in **transfer** is correct and is the latest. This is set each time **transfer** is set. This an workaround to prevent using attributes that are not available in all VHDL implementations.
7.2.4.20 *decode* — In detail

The instruction is decoded in this stage of the pipeline and its operands are read. The operations that *decode* performs can be listed as follows:

1. *fetch* passes the instruction that it read through *fetch_buffer*.

2. For multi-cycle instructions like *load* instructions, *iops* are introduced. When executing the *iops*, the instruction fetched is put in a two element array.

3. If the two element array is empty then the fetched instruction is decoded else the top of the array is used.

4. The instruction is torn down to different fields as described in Section 5.1.2.

5. The instruction is then checked by *unimplementedinstruction*. If it is an unimplemented instruction, the simulation is stopped.

6. If any instruction that affect *cwp* (Current Window Pointer), like *save*, *restore*, etc, then *decode* waits on *cwp_set*.

7. If the current instruction is a branch instruction, the following steps are executed:

   (a) If the instruction in *execute* is modifying the flags, then *decode* waits until that has been done, before determining if the branch is taken. This is signalled by *bcc_set*. 
(b) The condition is evaluated by eval_cond_code.

(c) If the branch has to be taken, then the target address is calculated based on the annul bit of the branch instruction. The target instruction’s address is then written to the pc.

(d) If the annul bit is set, then annul is rotated through two signals, annul_d and annul_f i.e.,

\[
\text{annul}_d \leq \text{annul}_f;
\]
\[
\text{annul}_f \leq \text{annul};
\]

Refer Section 6.5 to see how this assignment work in VHDL. annul_d is used by execute to annul an instruction. Thus the annulling of next instruction is accomplished.

8. For sethi the disp22 is read into operand1 which will be later passed to execute.

9. For arithmetic instructions the operands are determined. If the operands are being written by the write stage at the time that decode tries to read them, then they are copied from the write's buffer register. If an operand is the result of an instruction currently executing, then a buffer register is set to signal execute to use the result from the previous instruction for that operand.
10. The **jmpl** instructions are similar to **branch** instructions except the operands may be registers or immediate values:

(a) **fetch** is stopped using the signal **transfer**.

(b) The operands are determined. Here too the routing from **write**'s buffer occurs as in arithmetic instructions.

(c) The address is calculated and is put in the **abus** directly.

(d) The address + 4 is assigned to **pc** so that the jump is effected.

(e) Other signals **size**, **rd** and **we** are set to the required values for an instruction fetch.

11. The **load** and **store** are multi-cycle instructions. The **decode** process is the one that takes care of this special need:

(a) The operands are calculated in the same way as for other instructions, including the re-routing from **write**'s buffer.

(b) **maxversion** is set number of extra cycles needed.

(c) The variables **next_iop** and **next_i** keep track of **iops** introduced in the pipe. Also, there is a variable **iop** that is set to true, if the current cycle is an **iop** cycle.

(d) If **next_i.version** is less than the **maxversion**, then the **next_iop** is set to true and **next_i.version** is incremented. The variable **next_iop** is tested in
the beginning. If it is true (set in the previous cycle) then the instruction is put in the two element array and iop is set. Thus setting next_iop ensures the introduction of iop in the next cycle.

(e) If next_i.version equals maxvresion, then next_iop is set to false. In the next cycle the instruction will be taken from the two element array.

(f) When an iop is introduced the current instruction is stored in a buffer, next_i.data. This is used when an iop cycle is in effect.

12. For a call instruction, the destination address is calculated and is put on the address bus directly. The address +4 is assigned to the pc.

13. All the above happen during the clock zero phase. In clock one phase (clock = 1), the signals are assigned to their respective buffers, like, operand1 is assigned to operand1_buff. exec uses operand1_buff to read the value; thus the value is sent to the execute stage.

7.2.4.21 execute

Process execute takes the operands from the decode and based on the instruction calculates the result. Its operation can be listed as follows:

1. If annul_d is set, nothing happens in the clock zero phase.

2. The operands are collected in op1buff and op2_buff.
3. Memory misalignments are checked next. This should never happen. But is a checkpoint.

4. The rest are nested case statements based on op1, op2 and op3 of the instruction. They are passed on from decode through op1_buffer, op2_buffer and op3_buffer.

5. If it is a load instruction, perform_load is called.

6. If it is a store instruction, perform_store is called.

7. For other instructions, the required operation is carried out. For example, for addcc, the operands are added and the result is put in result. Then set_add_cc is called to set the flags.

8. During the clock one phase, the result is passed to write through result_buffer.

9. If the current instruction is a load instruction, then the data is taken from the dbusi.

10. valid_result is set whenever the value of result is valid and is current. This is used by write.

7.2.5 systemsent.vhd

This interfaces the IU and MMU described in sparcsent.vhd and memsent.vhd. Apart from putting these two modules together, it also assigns the default values of some
CHAPTER 7. THE FRAMEWORK

generic signals that are used in these files. It also assigns the default values to the signals (refer section 7.1.1).

To reiterate the order of compilation is:

1. datasent.vhd
2. memsent.vhd
3. opcodes.vhd
4. sparc.sent.vhd
5. systemsent.vhd

Once compiled into the working library the simulation tool can be used to simulate the implementation (See Chapter 10).
Chapter 8

Changes to the Frame Work

This chapter describes all the changes that were done to the framework described in the previous chapter to use 

\[ a + b \] arithmetic. The changes were done primarily to \texttt{datasent.vhd} and \texttt{sparcab.vhd}. \texttt{opcodes.vhd} was never touched — it shouldn’t be as both the models must execute the same set of opcodes. Very minor modifications have to be done to \texttt{memab.vhd} and \texttt{systemab.vhd} so that they use new \texttt{dataab.vhd} and \texttt{sparcab.vhd}. There were no changes done to the \texttt{meminterface} process. In what follows, we shall see the changes done to the files and hence the modifications to the model to create the SPARC processor that uses 

\[ a + b \] arithmetic.

8.1 dataab.vhd

A new record definition \texttt{abword} is added to \texttt{dataab.vhd}. This as well some supporting functions that convert an \texttt{abword} to \texttt{word} and reverse have been added. Also included are versions of the functions that use \texttt{abword} type operands apart from those that use \texttt{word}. In short, \texttt{dataab.vhd} is a superset of functions compared to \texttt{datasent.vhd}.

\texttt{abword} is a record that has two \texttt{words} as elements. They accounts for the “\(a\)” and “\(b\)” parts of “\(a + b\)”. There is a third record \texttt{bzero} which would be set if “\(b\)” is
zero. The purpose of this bit is explained later in this chapter.

8.1.1 Functions

8.1.1.1 abval (x)

This function converts an integer (x) to abword. This function converts the integer (x) to word using bitval and assigns that to “a” of abword. The “b” part gets zero. This function is overloaded to convert a word to abword. For the second version, there is no need to call bitval (see Section 7.2.1.3).

8.1.1.2 wabval (x)

This function converts an abword (x) to word. If the “b” is zero, then it just returns the “a” else, adds the two parts and returns the result. This is referred to as compaction. The compaction is done only if bzero is not set. Thus saves time by not doing the compaction when “b” part is zero.

8.1.1.3 iabval (x)

This function converts an abword to an integer. This calls wabval to convert the abword to word and then calls intval to convert the word to an integer.
8.1.1.4  shift_right_arithmetic (data1,shcnt)

This shifts an abword (data1) right by shcnt. There is shift_right_arithmetic in datasent.vhd (refer Section 7.2.1.13) that operates on a word. This additional version shifts an abword right by shcnt. This calls the word version of shift_right_arithmetic to shift the “a” part and “a” part. Then it sets the bzero if “b” part is zero.

8.1.1.5  shift_left_logical (data1,shcnt)

Similar to shift_right_arithmetic in that this is a second version and has a word counterpart. This also calls the word counterpart to do the shift and then sets the bzero flag.

8.1.1.6  shift_right_logical (data1,shcnt)

Similar to shift_left_logical but this shifts the abword right. This also calls the word counterpart to do the actual shift.

8.1.1.7  “+” (a,b)

This is an over loading of the operator “+”. This adds two abwords and the result is also an abword. The addition is carried out the same way as explained in Section 3.1.1. There are two loops to signify the two full adder additions. In the first loop, the “a” and “b” parts of the first operand (a) and “a” of the second operand (b) are added. In the second loop, the “a” and “b” parts of the temporary result and “b” of
the second operand are added. The \texttt{bzero} flag is set while the addition takes place. There are versions to add a \texttt{abword} and an integer. All these call the \texttt{abval} to convert the integer to \texttt{abword} and then call the \texttt{abword} version to add the two \texttt{abwords} (refer 149).

\textbf{8.1.1.8} \hspace{1em} \texttt{two\_complement} \hspace{0.5em} (x)

This calculates the two's compliment of an \texttt{abword} (x) as delineated in Section 3.1.2 (refer 149).

\textbf{8.1.1.9} \hspace{1em} "-" \hspace{0.5em} (a,b)

This calls the \texttt{two\_complement} to calculate the 2's complement of the subtractant (b) and then does the regular addition (refer 149).

\textbf{8.1.2} \hspace{1em} \textbf{More about addition and subtraction}

These functions are key to the new model. These functions are direct implementations of the algorithms delineated in sections 3.1.1 and 3.1.2. The timings can't be put in the procedures as there are no signals involved. Though the time taken to do addition is constant, the compaction time is not constant (because compaction is done only if \texttt{bzero} flag is set). Also when the actual addition happens, the result is put into a variable local to the process. So due to the design of the model, the timings can't be incorporated in the code. But, there is a great deal of information that is obtained.
These procedures are proof that the algorithm works. All we have do then is to hand analyze the maximum time required for the operations and compare it with the best known algorithm for the regular arithmetic. This idea would be revisited in Chapter 10 where the experiments are described with both the models and the results are analyzed.

8.2 sparcab.vhd

The changes that were done to the model are very intricate. Almost every line of the code is affected. The external interface to the model are untouched. So the new model appears the same to the outside world. To differentiate the two models, the new model is named sparcab.

The new model makes use of the same framework. The basic model of the framework: several processes interconnected by signals remains the same. The main focus of the change are to the internal variables of the processes and hence the signals that run between the processes. Some of these high level changes can be listed as follows:

1. The registers are changed from word to abword.

2. All the signals that carry the operands between the stages (or processes) are changed from word to abword. Some of the notable ones are:

   (a) The operand buffers: operand1_buff, operand2_buff, destination_buff
(b) The result buffers: \texttt{result$\_buff}$, \texttt{ex$\_result$

3. Procedure \texttt{rreg} now returns an abword and procedure \texttt{wreg} takes an abword.

4. Two new procedures \texttt{reqoperand bzero} and \texttt{reqresultbzero} are added. Their role and functions are explained later.

8.2.1 Processes

Regarding the processes their functionality remains the same. Note that the "$a + b$" arithmetic defined, gives algorithms only for arithmetic operations. Since the registers are modified to hold abwords the logical functions need extra work. The extra work is to do compaction (add both the parts and do the carry over). But the time required for compaction has to be accounted for. The way this is done, is to introduce \texttt{iops} when needed. This is done by the \texttt{decode stage} and is handled by the rest.

The processes \texttt{saveit}, \texttt{clkgen}, \texttt{restore} and \texttt{write} received the ripple effect of changing the signals to abword. Apart from that, there are no major changes to the algorithm of the processes. Other processes have been changed.

8.2.1.1 fetch

The fetch is not stalled for no more reasons other than those listed under Section 7.2.4.19. The signal \texttt{invalid$_\texttt{fetch}$} is now set if there are \texttt{iops} in the \texttt{execute stage}. This should never happen but is a precaution. This is to prevent \texttt{fetch} from reading
an invalid state of the abus and assume that to be an instruction. The only reason, this happens is that there are speed incompatibilities between the processor and memory.

8.2.1.2 decode

The decode process does the same steps listed under section 7.2.4.20. In addition to the steps listed there, it does the following:

1. There are some alu instructions, that require the operands with the “b” part zero (example, logical instructions like and, or) and there are instructions that require the result to compacted (example, instructions that affect flags like addc, subc). For these instructions, the iops have to be introduced. This is done as follows:

(a) First the instruction is tested to see if it requires the result to be compacted. This is done using the procedure reqresultbzero. reqresultbzero returns true if the instruction needs the result to be compacted, based on op, opl and op2 of the instruction. The procedure is a nested case statement and hence can be synthesized easily.

(b) The instruction is then tested to see if it requires the operands to be compacted. This is done using a procedure reqoperandbzero. This procedure is similar to reqresultbzero. It returns true if the instruction needs the
operands to be compacted based on op, op1 and op2 of the instruction.

(c) Both the aforementioned conditions should not be true for the same instruction. This assumption is valid because, though there are logical instructions that affect flags, the operands are compacted for them before the execution of the instruction and hence should never have the result with a non zero "b" part. If in any case this condition occurs, the simulation is halted.

(d) If any of the above conditions become true, then an iop is introduced. This is done in a similar way as for multi-cycle instructions by setting iop and nexti. Apart from these there is also another variable that is set: alu_iop. This is to keep track of the iops that are introduced because of the above mentioned conditions.

2. When loading the immediate values of the instructions, the procedure abval is used.

3. When calculating the address for jmpl instruction compaction is done. This instruction does not need iops as the cycle is itself stalled (till the operands are available).

4. The same applies for branch instructions.
5. For load instructions, the address is put in the first clock cycle of exec and hence the compaction can be done by decode. Also the memory is usually slower and is an valid assumption.

6. In the clock one phase one additional signal is passed on to its buffer companion: alu_iop is assigned to alu_iop_buff.

8.2.1.3 execute

The execute stage’s functionality is extended in a way. This stage apart from performing the operations listed under Section 7.2.4.21, it does the following:

1. For load and store instructions, the addresses have to be compacted. These are done in perform_load and perform_store respectively. The first step in these procedures is the compaction and afterwards there are no changes in their algorithms.

2. For those logical instructions that require the operands to be zero, the compaction is done in the first cycle and the operation is carried out in the next cycle.

3. For those instructions that require the result to be zero, the operation is done in the first cycle and the compaction is done in the next cycle. The flags are set in the second cycle.
4. For these multi-cycle instructions, as before valid\_result is used to prevent other stages from reading the incorrect result.

5. For MULScc, the flags are not updated as the N and V will not be affected. They are affected in the original version (refer Section 5.1.2.2) because there is a possibility of an overflow. Since in “a + b” we have an extra word, the overflow never occurs. There is change in shifting. If Y’s “a” or “b” has a non zero LSB, then shift is by one bit position. If both are non zero, the shift is by two bit positions.

There is only a minor change in the clock one phase of the exec stage: it uses abval to convert the values from the data bus (words) to abword.

The model has to be compiled in the same way as before and in the same order. The simulation of this model is explained in Chapter 10.
Chapter 9

Simulation Environment

This chapter describes the simulation environment and the experiment requirements. The fundamental need for experimentation with the model is an error free SPARC code that carries out some meaningful task. This would make the results of the experiment more valuable. The best source of an error free opcode is a SUN SPARC Station\(^1\). As the processor is SPARC, the codes executed by the SPARC station should be compatible with the models described in the previous chapters. But the code is not available easily. It has to be extracted from the executable produced by the compiler. The compiler chosen for this purpose was the one for Solaris 2.3 Operating System (Solaris 2.3 and higher are SUN's implementation UNIX System V and Release 4). The reasons for doing so are:

1. The format of the file are standardized and published by UNIX Press.

2. The VHDL simulators and other tools were available in Solaris 2.3 environment.

Hence using the same environment for extracting the code was the right choice.

\(^1\)Registered trademark of SUN Microsystems, Inc.
We shall now look into the “Application Binary Interface” specification for System V explained in [27].

9.1 System V Application Binary Interface

AT&T released UNIX System V Release 4.0 (SVR4) and with that a document was released that defines a system interface. This is for compiled application programs. This is System V Application Binary Interface or ABI. The specification has two parts: A generic part and a processor-specific part. The generic part specifies all those interface that are common across all implementations of SVR4 while the processor-specific specifies that are specific to a hardware architecture. In this section we would visit some of the details that pertains to the extraction of code for experimentation. A much detailed description can be found in [27]. [28] delineate SPARC-specific ABI specifications.

9.2 SPARC Processor Specific

In this section we shall highlight some of the processor specific information that influences our code extraction process and understanding of the code extracted.

To avoid any confusion, the ABI specifies the various data type lengths. In SPARC, a byte is 8-bits long; this includes both signed and unsigned byte. A signed or unsigned char is 8-bits long. The half word is 16-bits long. That would include
short integers (signed and unsigned). Integers (unsigned and signed), long integers
and pointers are all 32-bits long. A word is 32-bits long.

The ABI SPARC supplement also specifies how registers and stack frame would
be used in a function and in calling a function. A function’s window registers are
shown in Fig.9.1. In the figure, %fp stands for frame pointer and %sp stands for stack
pointer. The stack pointer is the limit of the current stack frame. The stack pointer
is always decremented by an amount. The usual function prologue would be save
%sp,-80,%sp. The value is determined by the number of parameters for the function
and like. The frame pointer is the address of the previous functions stack pointer. So
the function has both ends of the frame.

9.2.1 Functions

There are certain functions in the library that are used in a SPARC ABI. These
functions are integral part of SPARC SVR4 system. Most of these functions are
equivalent to some SPARC instructions. These instructions are optional instructions
and are available only in certain implementations. To guarantee the availability of
the instructions, the compiler always uses these functions. For example, only the
mulsec instruction is implemented in the model and is guaranteed to exist in all
SPARC implementations. But fmul is not. When actual instructions are available
these are replaced by corresponding SPARC instructions. This does a lot good for
the experimentation. When the compiler generates the code ( in the way described
### Table of Window Registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong></td>
<td>%i7</td>
<td>%r31 return address -8</td>
</tr>
<tr>
<td></td>
<td>%fp,</td>
<td>%i6  %r30 frame pointer</td>
</tr>
<tr>
<td></td>
<td>%i5</td>
<td>%r29 incoming param 5</td>
</tr>
<tr>
<td></td>
<td>%i4</td>
<td>%r28 incoming param 4</td>
</tr>
<tr>
<td></td>
<td>%i3</td>
<td>%r27 incoming param 3</td>
</tr>
<tr>
<td></td>
<td>%i2</td>
<td>%r26 incoming param 2</td>
</tr>
<tr>
<td></td>
<td>%i1</td>
<td>%r25 incoming param 1</td>
</tr>
<tr>
<td></td>
<td>%i0</td>
<td>%r24 incoming param 0 and outgoing return value</td>
</tr>
<tr>
<td>local</td>
<td>%l7</td>
<td>%r23 local 7</td>
</tr>
<tr>
<td></td>
<td>%l6</td>
<td>%r22 local 6</td>
</tr>
<tr>
<td></td>
<td>%l5</td>
<td>%r21 local 5</td>
</tr>
<tr>
<td></td>
<td>%l4</td>
<td>%r20 local 4</td>
</tr>
<tr>
<td></td>
<td>%l3</td>
<td>%r19 local 3</td>
</tr>
<tr>
<td></td>
<td>%l2</td>
<td>%r18 local 2</td>
</tr>
<tr>
<td></td>
<td>%l1</td>
<td>%r17 local 1</td>
</tr>
<tr>
<td></td>
<td>%l0</td>
<td>%r16 local 0</td>
</tr>
<tr>
<td>out</td>
<td>%o7</td>
<td>%r15 address of call instruction, temporary value</td>
</tr>
<tr>
<td></td>
<td>%sp,</td>
<td>%o6  %r14 stack pointer</td>
</tr>
<tr>
<td></td>
<td>%o5</td>
<td>%r13 outgoing param 5</td>
</tr>
<tr>
<td></td>
<td>%o4</td>
<td>%r12 outgoing param 4</td>
</tr>
<tr>
<td></td>
<td>%o3</td>
<td>%r11 outgoing param 3</td>
</tr>
<tr>
<td></td>
<td>%o2</td>
<td>%r10 outgoing param 2</td>
</tr>
<tr>
<td></td>
<td>%o1</td>
<td>%r9 outgoing param 1</td>
</tr>
<tr>
<td></td>
<td>%o0</td>
<td>%r8 outgoing param 0 and incoming return value</td>
</tr>
<tr>
<td>global</td>
<td>%g7</td>
<td>%r7 global 7 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td>%g6</td>
<td>%r6 global 6 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td>%g5</td>
<td>%r5 global 5 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td>%g4</td>
<td>%r4 global 4 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td>%g3</td>
<td>%r3 global 3 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td>%g2</td>
<td>%r2 global 2 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td>%g1</td>
<td>%r1 global 1</td>
</tr>
<tr>
<td></td>
<td>%g0</td>
<td>%r0 0</td>
</tr>
<tr>
<td>floating-point</td>
<td>%f31</td>
<td>Floating-point value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>%f0</td>
<td>Floating-point value</td>
</tr>
<tr>
<td>special</td>
<td>%y</td>
<td>Y register</td>
</tr>
</tbody>
</table>

---

**Figure 9.1:** A Function's Window Registers
CHAPTER 9. SIMULATION ENVIRONMENT

<table>
<thead>
<tr>
<th>Relation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a equal b</td>
<td>0</td>
</tr>
<tr>
<td>a less than b</td>
<td>1</td>
</tr>
<tr>
<td>a greater than b</td>
<td>2</td>
</tr>
<tr>
<td>a unordered with respect to b</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 9.1: _Q_cmp return values

later), these routines are called and code for these routines get copied as well. The
mul is used to multiply two numbers and is available in the library. Thus by using
this library function, which is optimized gives us a valuable comparison between the
two models.

The following sections describe these functions at a high level. There are also
specifications about function calls, a process's initialization, a function's epilogue and
prologue, coding examples, and others included in the SVR4 ABI and are out of scope
for this thesis. Interested readers are urged to see [28] and [7].

9.2.1.1 long double _Q.add (long double a, long double b)

This adds two quad-precision floating point numbers. This is equivalent to the SPARC
instruction faddq.

9.2.1.2 int _Q_cmp (long double a, long double b)

This compares two quad-precision values and returns an integer as shown in Table.9.1.
This implements the fcmp instruction.
9.2.1.3 int _Q_cmpe (long double a, long double b)

Same as _Q_cmp and implements instruction fcmpq.

9.2.1.4 long double _Q_div (long double a, long double b)

Implements fdivq instruction. Returns the value of a/b.

9.2.1.5 long double _Q dtoq (long double a, long double b)

Implements fdtoq instruction.

9.2.1.6 int _Q_feq (long double a, long double b)

Implements fcmpeq instruction. This compares two quad-precision values and returns an integer value if they are equal, else a zero.

9.2.1.7 int _Q_fge (long double a, long double b)

This compares two quad-precision values and returns a non-zero value if a is greater than or equal to b, else a zero. This mimics some conditions of the instruction fcmpeq.

9.2.1.8 int _Q_fgt (long double a, long double b)

This compares two quad-precision values and returns a non-zero value if a is greater than b, else a zero. This mimics some of fcmpeq instruction.
9.2.1.9  int _Q_fle (long double a, long double b)

This compares two quad-precision values and returns a non-zero value if a is less than or equal to b, else a zero. This mimics some of _fcmpeq instruction.

9.2.1.10  int _Q_flt (long double a, long double b)

This compares two quad-precision values and returns a non-zero value if a is less than b, else a zero. This mimics some conditions of the instruction _fcmpeq.

9.2.1.11  int _Q_fne (long double a, long double b)

This mimics a condition of _fcmpq instruction. This compares two quad-precision values and returns a non-zero value if a is not equal to b, else a zero.

9.2.1.12  long double _Q_itoq (int a)

This converts an integer (a) to a quad-precision value (return value), equivalent to _fitoq.

9.2.1.13  long double _Q_mul (long double a, long double b)

This produces the result of a × b same as _fmulq instruction. The result is in quad-precision.
9.2.1.14  long double _Q_neg (long double a)

Computes \(-a\). Same as fnegs.

9.2.1.15  double _Q_qtod (long double a)

Converts a quad-precision (a) to a double precision (return value) and corresponds to fqtod.

9.2.1.16  int _Q_qtoi (long double a)

Corresponds to fqtoi and converts a quad-precision (a) to 32-bit integer (return value).

9.2.1.17  float _Q_qtos (long double a)

Converts a quad-precision (a) to a single precision (return value) and corresponds to fqtos.

9.2.1.18  unsigned int _Q_qtou (long double a)

Converts a quad-precision to an unsigned integer.

9.2.1.19  long double _Q_sqrt (long double a)

Computes the square root of a; equivalent to fsqrtq.
9.2.1.20  long double _Q_stoq (long double a)

Reverse counter-part of _Q_qtos and corresponds to fstoq in that it converts a single precision to a quad-precision.

9.2.1.21  long double _Q_sub (long double a, long double b)

This computes $a - b$. Computation is in quad-precision and corresponds to fsubq.

9.2.1.22  long double _Q_utoq (long double a)

Reverse counter-part of _Q_qtou in that it converts a unsigned integer to a quad-precision.

9.2.1.23  int .div (int a, int b)

Computes $a/b$. This is a signed integer division.

9.2.1.24  unsigned int _dtou (double a)

This converts a double-precision to an unsigned number and discards any fractional part.

9.2.1.25  unsigned int _ftou (float a)

Converts a single precision argument to an unsigned integer. Here again the fractional part is discarded.
CHAPTER 9. SIMULATION ENVIRONMENT

9.2.1.26 \texttt{int .mul (int a, int b)}

This computes $a \times b$. This is a signed integer multiplication.

9.2.1.27 \texttt{int .rem (int a, int b)}

This computes the signed integer remainder or $a/b$. If $b$ is zero, then a trap is generated.

9.2.1.28 \texttt{.stret1, .stret2, .stret4 and .stret8}

They should not be called directly. These are called to within routines to copy structures etc and correct alignments.

9.2.1.29 \texttt{unsigned .udiv (unsigned a, unsigned b)}

This performs the unsigned integer division $a/b$ and generates a trap when $b$ is zero.

9.2.1.30 \texttt{unsigned .umul (unsigned a, unsigned b)}

This is the unsigned counterpart of \texttt{.mul}. This computes the unsigned product of $a \times b$.

9.2.1.31 \texttt{unsigned .urem (unsigned a, unsigned b)}

This computes the unsigned integer remainder of $a/b$. 
9.3 File Format

Included in the specification is the object file format. This is called Executable and Linking Format (ELF). There are three main types of object files:

**Relocatable File**: These type of files have code that have to be linked with other object files to create an executable file.

**Executable File**: This holds the executable code that can be loaded into memory for execution. This can be used by exec\(^2\) to create the program’s process image.

**Shared Object File**: This holds code for certain objects. But the a link editor (like ld) must read this file and link it with other loadable object files to create another object file. The dynamic linker then reads this file and links with more object files and creates the process image.

Both the Relocatable file and Shared Object File require some kind of an editor to create the executable file. For extracting the code for our experiment we need an executable file since the other types of files are incomplete. Only in the executable file the code for standard procedures like multiplication, division and other mathematical functions are included. The steps to create such a file is detailed below. Also delineated are the ways to identify an executable file.

The object file format and process image of the file is shown in Fig.9.2

\(^2\)exec is a UNIX procedure that is part of the kernel. There are several steps accomplished by this procedure and is specified in the UNIX documents. The format of the file is described later.
The linking view described, is the format of relocatable and shared object files. This would be format of the files read by link editor and the dynamic editors. These editors produce a file that has the process image that has the format of the executable file also described in Fig.9.2. This is the format from which the code for the experiment has to be extracted. The format described supports various processors with 8-bit and 32-bit architectures. The format is also extensible. For the purpose of the experiment, we are more concerned about the Executable File Format and its implications to SPARC architectures. In the following sections the executable file format is described in greater detail and where appropriate the SPARC dependent features are also described.

9.3.1 Executable File Format

The executable file has an ELF header at the very beginning. The process image to be loaded are broken into segments and is described by program header table entries.
The program header table follows the ELF header table (refer Fig.9.2). Within the segments are sections. The various sections are described by the section header table. There are C libraries provided to manipulate the various header tables. They are part of a standard SVR4 distribution.

9.3.1.1 ELF Header

The ELF header resides at the very beginning of the file and holds the map for the rest of the file. The ELF header can be best described with the help of a C structure.

```c
#define EI_IDENT 16

typedef struct{
    unsigned char e_ident[EI_IDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32.Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
} ElfHeader;
```
Elf32_Half e_phnum;
Elf32_Half e_shentsize;
Elf32_Half e_shnum;
Elf32_Half e_shstrndx;
}

In the above structure, Elf_Word, Elf32.Addr and ELF.Off are 4 bytes. Elf_Half is 2 bytes long. An unsigned char is a byte long. Here a byte is 8-bits long. The members of the structure can be thought of as lined up one after the other. The definition is such that the boundaries are considered so that there aren’t any conflict with the underlying architecture.

**e_ident** is the initial bytes that mark the file. This field is sixteen characters long.

1. The first four characters are ‘0x7f’, ‘E’, ‘L’ and ‘F’. They are for file identification.

2. The fifth character identifies the file class. For SPARC it has to be 1. This specifies that this file supports 32 bit machines and 4 gigabytes of address space.

3. The sixth character must be 2 which specifies that 2’s complement values are used and the most significant byte occupies the lowest address.

4. The seventh value is same as e_version and is explained below.


<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET_NONE</td>
<td>0</td>
<td>No file type</td>
</tr>
<tr>
<td>ET_REL</td>
<td>1</td>
<td>Relocatable File</td>
</tr>
<tr>
<td>ET_EXEC</td>
<td>2</td>
<td>Executable File</td>
</tr>
<tr>
<td>ET_DYN</td>
<td>3</td>
<td>Shared Object File</td>
</tr>
<tr>
<td>ET_CORE</td>
<td>4</td>
<td>Core File</td>
</tr>
<tr>
<td>ET_LOPROC</td>
<td>0xff00</td>
<td>Processor Specific</td>
</tr>
<tr>
<td>ET_HIPROC</td>
<td>0xffff</td>
<td>Processor Specific</td>
</tr>
</tbody>
</table>

Table 9.2: e_type values in a ELF header

5. The eighth character marks the beginning of the unused bytes in eIdent.

At present it is zero.

6. The rest are all set to zero.

The index and the values are defined in the file sys/elf.h. This is part of the standard distribution of a SVR4.

e_type This member identifies the object file type. It has to be one of the values shown in Table.9.2. For our purpose it has to be 2.

e_machine This has to have a value of 2 which specifies that the required architecture is SPARC.

e_version This has to have a value of 1 which specifies that this is current.

e_entry This specifies the entry address for the program. As soon as the program is loaded into the memory, the control is first transferred to this address.

e_phoff This member holds the offset of the program header header table.
CHAPTER 9. SIMULATION ENVIRONMENT

**e.shoff** This gives the offset of the section header table.

**e.flags** This holds processor specific flags. For SPARC this is unused and set to zero.

**e.ehsize** gives the size of the ELF header.

**e.phentsize** gives the size of a program header table entry.

**e.phnum** specifies the number of entries in the program header table.

**e.shentsize** gives the size of a section header table entry.

**e.shnum** specifies the number of entries in the section header table.

**e.shstrndx** specifies the index of string table in section header table.

### 9.3.1.2 Program Header

The system creates the process image by logically copying a file's segment to the virtual memory segment. The executable file can be thought of having segments. The segments are further divided into sections. The program header table describes the segments. The program header table can also be described with the help of a C structure.

```c
typedef struct{
    Elf32_Word p_type;
    Elf32_Off p_offset;
};
```
Elf32.Addr p_vaddr;
Elf32.Addr p_paddr;
Elf32.Word p_filesz;
Elf32.Word p_memsz;
Elf32.Word p_flags;
Elf32.Word p_align;

} Elf32_Phdr;

p_type This specifies the type of the segment this entry describes. For our experiment this must be of type PT_LOAD, which specifies loadable segments.

p_offset Specifies the offset of the first byte of the segment from the beginning of the file.

p_vaddr Specifies the virtual address offset of the first byte of the segment in the memory.

p_paddr This gives the segments physical address and for SPARC systems this is unspecified.

p_filesz Gives the number of bytes of the segment that are in the file.

p_memsz gives the number of bytes of the segment that have to be in memory.

p_flags This specifies whether the segment is executable (PF_X), readable (PF_R), writable (PF_W) or any of the combinations.
### Table 9.3: Typical program header segments

<table>
<thead>
<tr>
<th>Member</th>
<th>Text</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_type</td>
<td>PT_LOAD</td>
<td>PT_LOAD</td>
</tr>
<tr>
<td>p_offset</td>
<td>0x100</td>
<td>0x2bf00</td>
</tr>
<tr>
<td>p_vaddr</td>
<td>0x10100</td>
<td>0x4bf00</td>
</tr>
<tr>
<td>p_paddr</td>
<td>unspecified</td>
<td>unspecified</td>
</tr>
<tr>
<td>p_filesz</td>
<td>0x2be00</td>
<td>0x4e00</td>
</tr>
<tr>
<td>p_memsz</td>
<td>0x2be00</td>
<td>0x5e24</td>
</tr>
<tr>
<td>p_flags</td>
<td>PF.R + PF.X</td>
<td>PF.R + PF.W + PF.X</td>
</tr>
<tr>
<td>p_align</td>
<td>0x10000</td>
<td>0x10000</td>
</tr>
</tbody>
</table>

**p_align** This member gives the value to which the segment must be aligned.

A typical executable file will have two segments: Text and data. The program header table entries for such a file is given in Table.9.3.

#### 9.3.1.3 Section Header

As mentioned earlier, the segments are composed of sections. The section header table describe the sections. There are many types of sections. The section header table describes these sections. The section header table can also be described with the help of a C structure.

```c
typedef struct{
    Elf32_Word   sh_name;
    Elf32_Word   sh_type;
    Elf32_Word   sh_flags;
    Elf32.Addr   sh_addr;
}
```
Elf32_off    sh_offset;
Elf32_Word   sh_size;
Elf32_Word   sh_link;
Elf32_Word   sh_info;
Elf32_Word   sh_addralign;
Elf32_Word   sh_entsize;

} Elf32_Shdr;

**sh_name** Specifies the name of the section. This is actually an index into the string table section which holds all the names.

**sh_type** Holds the section's type that categorizes the section.

**sh_flags** This specifies the section as executable (SHF_EXECINSTR), memory resident (SHF_ALLOC) or writable (SHF_WRITE) or any combination of them.

**sh_addr** Specifies the address of the first byte of the section in the process' memory image.

**sh_offset** gives the offset of the first byte of the section from the beginning of the file.

**sh_size** specifies the size of the section in bytes.

**sh_link** The interpretation of this member depends on the type of the section.
**sh_info** Holds some extra information about the section and its interpretation is dependent on the type.

**sh_addralign** Specifies the section alignment constraint like, doubleword alignment etc.

**sh_entsize** For some sections which are hold tables, this member specifies the size of an entry in the table.

There are many types of sections. They are: `.bss`, `.comment`, `.data`, `.data1`, `.debug`, `.dynamic`, `.dynstr`, `.dynsym`, `.fini`, `.got`, `.hash`, `.init`, `.interp`, `.line`, `.note`, `.plt`, `.relname`, `.relaname`, `.rodata`, `.rodata1`, `.shstrtab`, `.strtab`, `.symtab`, `.text`. Description on each of these sections can be found in [27]. For the purpose of this experiment, only few of the these types are relevant. They are:

**.bss** Holds the uninitialised data. The section occupies no file space but has a definite space in memory.

**.fini** Holds the programs termination code.

**.init** This holds the initialization code. This section is executed before calling the programs entry point (like main in a C program).

**.shstrtab** Holds the names of the sections.
.text Holds the executable instructions.

9.3.2 Input File Format – indata

As it can be seen, the executable file has very complicated structure and is difficult to decipher in a VHDL environment. Also the file is in binary image. It is difficult to read a binary image in VHDL. For these reasons a very primitive file format is suggested and this is read by various load procedures in the models. The file format is as shown in Fig.9.3. In the figure, Segment Separator is \(-1\). The rest of the data are unsigned integers. The Segment Data is unsigned integer specifying the value of each byte as they should be in memory. If there are any discontinuities, the data in memory is assumed to be zero.
CHAPTER 9. SIMULATION ENVIRONMENT

9.4 Mentor Environment

The simulation suite used was from Mentor Graphics, Inc. The actual tools used were:

qvlib To create a work library.

qvcom To compile the VHDL code into the work library.

qvsim To simulate the model. The tool has a graphical user interface. It has facilities to step through each line of code, view the signals, wave forms etc. These facilitated the debugging of the model. But the simulations for data collection were automated using the tools initialization file (refer [24]).

The tools allow any user to have multiple working libraries. So one library of each of the models (SPARC and SPARCAB) were maintained. They were under separate directories: sparc and sparcab. Each time a memory image was created, copies of the file was put in both the directories under the file name indata. This is the file, the models would load from.

9.5 Tools – elf2memimg

To create indata file from the executable file format needs considerable work. This section describes the tool elf2memimg that does the conversion. In general the tools
reads in the executable file using the elf libraries and does some checking. If there are no errors, creates the indata file. The algorithm used can be listed as follows:

1. Open the input and output files.

2. Using elf_version check if the libraries and file are using the same ELF version.

3. Read the ELF header using elf_begin.

4. Using the ELF header obtained retrieve the Program header.

5. Using elf_getscn get the first section. This section is discarded. By calling this function the internal structures are initialized.

6. Retrieve the section header using elf_getshdr.

7. Output the entry program counter and end program counter based on the ELF header retrieved in Step 3.

8. While the number of segments retrieved is less than total number of segments in the file, do the following:

   (a) Output the segment separator.

   (b) Output the segment start address.

   (c) While the end address of this segment is not reached, do the following:

      i. Retrieve the section.
CHAPTER 9. SIMULATION ENVIRONMENT

ii. Load the data (using elf.rawdata).

iii. Output the data.

iv. If the file does not hold data, fill it with zeros.

9. Close the files.

To use the tool:

elf2memimg <filename> <output filename>

The output file is the indata file.

9.6 Process to create the indata

In a UNIX system, the executable file format can be produced with the help of compiler by calling the compiler as cc -dn. A generic makefile\(^3\) would be as follows:

PGM = <program name>

all: $ (PGM).indata

#step 1

$ (PGM).s: $ (PGM).c Makefile

    cc -S $ (PGM).c

#step 2

\(^3\)makefile is used by the program make, a standard program in UNIX. make follows a dependency rule and executes the steps listed to create the target.
The first step is creating an assembly code of the C program. This step is helpful for debugging but not needed. The object file (file with extension .o) can be directly created. This is done in the next step by calling as, the assembler. The file generated is a shared object file described before. Then the a statically linked file is created in
step 3. This file will be the executable file and has all the code necessary (ld and all editors would have been called and hence no further action is required). In the fourth step, the input file \texttt{indata} (refer Section 9.3.2) is created using the tool \texttt{elf2memimg} (refer Section 9.5).

9.7 Other helpful tools

To aid the debugging of the models a whole suite of tools are also provided. They are described in this section.

9.7.1 decipherlib.c

This compiles into a library. This breaks a SPARC instruction into one of the three formats (refer Section 5.1.2). Then based on that, it further breaks into specific instruction fields. It then forms a string describing the instruction.

9.7.2 deciphermemimg.c

Reads in the model input (\texttt{indata}) and uses the decipherlib to decipher the instructions. This gives a printout which helps to follow the model as it executes instruction by instruction. To use the tool:

\begin{verbatim}
deciphermemimg [<filename>]
\end{verbatim}
If a filename is not present, the tool reads from the standard input.

9.7.3 decipher.c

Also based on decipherlib.c. This reads the standard input for a long unsigned integer and treats this as an instruction and deciphers the instruction. To use the tool:

    decipher [<filename>]

If a filename is not present, the tool reads from the standard input.

9.7.4 elfgetscn.c

This program extracts section by section from the executable file based on users' request and writes to indata file. This way only a particular section can be executed by the model. instruction. To use the tool:

    elfgetscn <filename> <output filename>

Once started, it reads the header and then outputs a prompt. The user can then enter a section number, a number greater than zero, and the tool extracts the particular section from the executable, writes it to the output file in indata format. This continues until the user enters any number less than or equal to zero, the program closes all files and exits.
9.7.5 getanddeci.c

Gets a named sections from an object file and deciphers section by section. It also has other formatting facilities. To use the tool:

```
getanddeci [-[s|S][a|A][d|D]] <filename> <output filename>
```

This tool is very similar to elfgetscn in its user interface. It outputs a prompt and based on the section number, gets the section and deciphers the section and writes the output to the output file. The format of the output depends on the command line options. The command line options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Section Headers are printed</td>
</tr>
<tr>
<td>s</td>
<td>Section Headers are NOT printed</td>
</tr>
<tr>
<td>A</td>
<td>Address of each instruction is printed</td>
</tr>
<tr>
<td>a</td>
<td>Address of each instruction is NOT printed</td>
</tr>
<tr>
<td>D</td>
<td>Each instruction is deciphered</td>
</tr>
<tr>
<td>d</td>
<td>Each instruction is NOT deciphered</td>
</tr>
</tbody>
</table>

9.7.6 printhdrs.c

This prints the various headers in the file. This prints the section headers, program headers and the elf header and their values. At the end, it also prints the contents of the .strtab section. To use the tool:
printhdrs <filename>
Chapter 10

Simulation

This chapter describes the actual simulation and the results of the simulation. The simulation process is depicted in Fig.4.8. In the previous chapter the entire simulation environment and various requirements were delineated. It is assumed that there are two directories: sparc and sparcab and models are compiled into separate libraries under each of them(refer Section 9.4). First, a simple program listed below and walk through the various steps to get the results from the models. The same is done for a second sample.

10.1 Sample A

The following C program computes three additions and assigns to the local variables.

```
#include <stdio.h>

int giI, giJ, giK;

main()
```
{
    register int riI, riJ, riK;
    int iI, iJ, iK;

    riI = 99;
    iI = riI;
    giI = iI;

    riJ = riI;
    riJ++;  /* 2 */
    iJ = riJ;
    giJ = iJ;

    riK = riJ;
    riK++;  /* 3 */
    riK++;  /* 4 */
    riK++;  /* 5 */
    riK++;  /* 6 */
    riK++;  /* 7 */
    iK = riK;
giK = iK;
}

The program’s indata file can be created by the same steps delineated in Section 9.6. The simulation is done using the qvsim tool.

10.1.1 Simulation by SPARC

To do the simulation, change to the sparc directory and start the simulation tool. Before this, the work library has to be set to the one that has SPARC compiled into it. Load the system and start the simulation. The model loads in the indata file and executes the instructions. The output, the memory dump at the end of simulation, is put in a file named outdata. The top section of this file is as shown below:

Time to complete: 41900 ns
start of this segment: 131980
0
0
.
.
.
.
.
.
.

From the file, the number of cycles to complete the simulation can be found. It is 41900 ns.
10.1.2 Simulation by SPARCAB

To test the SPARCAB, change to the corresponding directory, set the work library to the one that has SPARCAB complied into and start the simulation. The model would load the indata file and do the simulation. At the end of simulation the outdata file will be produced again. The top section of this file is as shown below:

Time to complete: 45600 ns

start of this segment: 131980

0

0

....

....

Again from the file, the number of cycles to complete the simulation can be found. It is 45600 ns.

10.2 Sample B

The result shown in the previous section is further highlighted in multiplication. The test C program is as shown below:

```
#include <stdio.h>
```
int iI, iJ, iK, iL;

main()
{
    register int riI, riJ, riK, riL;

    iI = riI = 4;
    iJ = riJ = 0;
    iK = riK = 0;
    iL = riL = 0;

    riJ = riI++;
    riK = riJ * riI;
    riK = riK++;
    riL = riK * 30;

    iI = riI;
    iJ = riJ;
    iK = riK;
    iL = riL;
The program's indata file has to be created again by the same steps delineated in Section 9.6.

10.2.1 Simulation by SPARC

To do the simulation, do the same steps as delineated in Section 10.1.1. A new outdata will be created at the end of simulation. The top section of this file is as shown below:

Time to complete: 58000 ns

start of this segment: 132624

0

0

Again from the file, the number of cycles to complete the simulation can be found. It is 58000 ns.
10.2.2 Simulation by SPARCAB

Do the same steps delineated in Section 10.1.2. The top section of the outdata file produced is as shown below:

Time to complete: 63500 ns

start of this segment: 132624

0

0

As before, from the file, the number of cycles to complete the simulation can be found. It is 63500ns.

10.2.3 Comparison

As it can be seen, sparcab takes more cycles in carrying out the simulation. This is so because, the clock time was not changed and extra iops were introduced. Consider the following. For Sample A:

\[
\text{Clock period} = 100\text{ns} \quad (10.1)
\]

\[
\text{Time taken by SPARC} = 41900\text{ns} \quad (10.2)
\]
CHAPTER 10. SIMULATION

Time taken by SPARCAB = 45600ns \hspace{1cm} (10.3)

If, the clock time for SPARCAB is reduced to say, 25ns, then:

Scaled Time taken by SPARCAB = 45600 \times 0.25 \hspace{1cm} (10.4)
= 11400ns \hspace{1cm} (10.5)

A rationale that supports the reduction in cycle time is that the carry over is not done in a regular cycle in sparcab. And there is a dedicated iop cycle to do the carry over or compaction. Applying the same rationale for Sample B:

Time taken by SPARC = 58000ns \hspace{1cm} (10.6)

Time taken by SPARCAB = 63500ns \hspace{1cm} (10.7)

Scaled Time taken by SPARCAB = 63500 \times 0.25 \hspace{1cm} (10.8)
= 15875ns \hspace{1cm} (10.9)

Thus, using "a + b" algorithm, saves time. Even in the case of small programs, a remarkable time improvement is achievable.

It is important to note that though the sample programs are small and focus on testing addition or multiplication, all classes of instructions are executed in various sections like .init (refer Section 9.3).
Also note, in case of Sample B, the models are using \texttt{.mul} of the system library to actually do the simulation (refer Section 9.2.1). This gives the pure comparison of both the models since no special module is used to do multiplication.
Chapter 11

Conclusion

It is appropriate to reiterate that addition and multiplication are the dominant arithmetic operations in digital computers and digital signal processing devices. The faster these operations can be performed, the faster the processor is. By using "a + b" arithmetic, the speed can be increased. Also there the operations are pipelined. All this with the expense in hardware cost.

This thesis started out to explain the "a + b" arithmetic. Algorithms for the arithmetic and a cheaper VLSI layout was also proposed. The algorithms were then used in a generic processor, SPARC and was proved that the algorithms proposed can be used in any processor and just internal to the processor. The external world never noticed the change. And yet gain in speed.

As a by product, this thesis also extended the framework and other tools which can be used for further research.

Thus the accomplishments of this work can be listed as:

1. A faster algorithm for doing arithmetic operations was produced.

2. The concept was tested using a generic architecture.
3. Also, a VHDL code for a SPARC using "a + b" architecture was produced and tested.

11.1 Future Work

The change was done only to the processor and "a + b" was only known to the processor. But as the realm of "a + b" extends over to other units in the architecture, the speed can be further increased. This research topic would be definitely worth pursuing. A "a + b" representation for floating point numbers was already proposed in this thesis. This along with extended arithmetic algorithms can be proposed for floating point numbers. This can be also checked with the present framework. The present tools can be readily used. Also, in "a + b" architecture the overflow may go unnoticed. This is because, in "a + b" architecture, there are more bit positions available to store the actual information; whereas, regular architectures have no such advantage. There are many programs written, which ignore the overflow bits. Some work has to be done to accommodate this requirement.
Bibliography


BIBLIOGRAPHY


Appendix A

As mentioned before, the framework was supplied by Texas Instruments, Inc. Following is the copyright notice:

-- copyright (c) 1991, 1990 texas instruments incorporated

-- permission is granted to any individual or institution

-- to use, copy, modify, and distribute this software,

-- provided that this complete copyright and permission notice

-- is maintained, intact, in all copies and supporting documentation.

-- texas instruments incorporated provides this software "as is"

-- without express or implied warranty.
Appendix B

Following is a list of some important "a + b" algorithms implemented using VHDL.

type abword is record
    a: word32;
    b: word32;
    bzero: bit; -- if 1 indicates b part is zero
end record;

function "+" (x,y: abword) return abword is
    -- add, ignoring overflow etc.
    variable i: integer;
    variable checkingint, checkingint2, checkb: integer;
    variable result, temp : abword;
begin
    temp.b(0) := '0';
    result.bzero:= '1';
    for i in 0 to 31 loop
        -- this is a verbose truth table implementation
        temp.a(i) := (( x.a(i)) and (not x.b(i)) and (not y.a(i))) or
Appendix B

((not x.a(i)) and ( x.b(i)) and (not y.a(i))) or
((not x.a(i)) and (not x.b(i)) and ( y.a(i))) or
(( x.a(i)) and ( x.b(i)) and ( y.a(i)));

if (i /= 31 ) then
  temp.b(i+1) := ((not x.a(i)) and (x.b(i)) and (y.a(i))) or
  ((x.a(i)) and (not x.b(i)) and (y.a(i))) or
  ((x.a(i)) and (x.b(i)) and (not y.a(i))) or
  ((x.a(i)) and (x.b(i)) and (y.a(i)));

end if;

end loop;

result.b(0) := '0';

for i in 0 to 31 loop
  result.a(i) := ((temp.a(i)) and (not temp.b(i))
and (not y.b(i))) or
  ((not temp.a(i)) and (temp.b(i)) and (not y.b(i))) or
  ((not temp.a(i)) and (temp.b(i)) and (not y.b(i))) or
  ((temp.a(i)) and (temp.b(i)) and (y.b(i)));

  if (i /= 31 ) then
    result.b(i+1) := ((not temp.a(i)) and (temp.b(i))
    and (y.b(i))) or
  
  end if;

end loop;

result.b(O) := '0';

for i in 0 to 31 loop
  result.a(i) := ((temp.a(i)) and (not temp.b(i))
and (not y.b(i))) or
  ((not temp.a(i)) and (temp.b(i)) and (not y.b(i))) or
  ((not temp.a(i)) and (temp.b(i)) and (y.b(i)))
and (y.b(i))) or
  
  end if;

end loop;
(((temp.a(i)) and (not temp.b(i)) and (y.b(i))) or
(((temp.a(i)) and (temp.b(i)) and (not y.b(i))) or
(((temp.a(i)) and (temp.b(i)) and (y.b(i)));

if ((result.bzero = '1') and (result.b(i+1) = '1')) then

result.bzero := '0';

end if;

end if;
end loop;

checkingint := intval(x.a) + intval (x.b) + intval(y.a) +
intval (y.b);

checkingint2 := intval ( wabval(result));

assert ( checkingint = checkingint2 )

report "error in abval addition " severity failure;

checkb := intval(result.b);

if ( (result.bzero = '1') and (checkb /= 0) ) or
( (result.bzero = '0') and (checkb = 0) ) then

assert ( false )

report "bzero not set properly" severity failure;

end if;

return(result);
function two_complement(x : abword) return abword is
  variable i: integer;
  variable result, temp: abword;
  begin
    for i in 0 to 31 loop
      temp.a(i) := not x.a(i);
      temp.b(i) := not x.b(i);
    end loop;
    result := temp + abval(2);
    return(result);
  end;

function "-" (x,y: abword) return abword is
  begin
    return (x + two_complement(y));
  end "-";