Processor support for interval arithmetic

Gerald Shawn Williams
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Williams, Gerald Shawn
Processor Support for Internal Arithmetic

May 13, 1998
PROCESSOR SUPPORT FOR INTERVAL ARITHMETIC

by

Gerald Shawn Williams

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Engineering

Lehigh University

May, 1998
This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

5/6/98
Date

Thesis Advisor

Chairperson of Department
Acknowledgements

I would like to thank my wife Cynthia for all of her love and support. She is my source of encouragement, and has always been willing to do whatever it takes to allow me to complete my studies, even though it has meant doing more than her share of work around the house during the past few years.

I would also like to thank my advisor, Dr. Michael J. Schulte. He has been a continual source of encouragement and guidance to me, and has spent many hours helping to refine my ideas, clearing up misunderstandings, and reviewing papers and conference submissions. He has always been more than willing to help, and has given up many nights and weekends to this end.

Finally, I would like to thank my parents, James and Julia, for instilling in me a will to learn and grow.
Table of Contents

Acknowledgements .................................................. iii
Table of Contents .................................................. iv
List of Tables ........................................................ viii
List of Figures ......................................................... ix
Abstract ................................................................ 1

Chapter 1
Introduction ............................................................. 2
1.1 The Price of Failure ................................................. 2
1.2 Arithmetic Errors in Multiuse Computer Systems .............. 3
1.3 The Role of Simulation and Modeling ............................... 4
1.4 Handling Arithmetic Errors ......................................... 4
1.5 Reliable computing .................................................. 5
1.6 The Need for Speed ............................................... 6
1.7 The Interval Arithmetic Solution ................................... 7
1.8 Performance of Interval Arithmetic ................................. 7
1.9 Architectural Support for Interval Arithmetic ................... 8
1.10 Overview .......................................................... 9

Chapter 2
Number Representations ............................................. 11
2.1 Exact Methods ................................................... 11
2.2 Inexact Methods ................................................ 11
2.3 Floating Point Numbers .......................................... 12
2.4 Limitations of Inexact Methods ................................ 13

Chapter 3
Interval Arithmetic .................................................... 14
3.1 History of Interval Arithmetic ................................... 14
3.2 Interval Representations .......................... ................. 15
3.3 Outward Rounding ............................................. 16
3.4 Interval Operations .............................................. 16
  3.4.1 Binary Interval Operations .............................. 17
  3.4.2 Interval Scalar Decompositions ....... .................... 18
  3.4.3 Unary Interval Operations ............................. 19
  3.4.4 Interval Comparisons .................................. 20
3.5 Software Support for Intervals ................................. 21
List of Tables

Table 1: Binary interval operations ................................................................. 18
Table 2: Scalar interval decompositions .......................................................... 19
Table 3: Unary interval operations ................................................................. 19
Table 4: Interval comparisons ....................................................................... 20
Table 5: Summary of proposed interval extensions ......................................... 29
Table 6: Some traditional DLX instructions ..................................................... 30
Table 7: Rounding mode transitions ............................................................... 31
Table 8: Addition, subtraction, and width in enhanced DLX ......................... 32
Table 9: Interval multiplication and division in enhanced DLX ..................... 33
Table 10: Interval magnitude and magnitude in enhanced DLX ..................... 34
Table 11: Interval midpoint and negate in enhanced DLX .............................. 35
Table 12: Simulator Requirements .................................................................. 42
Table 13: FP module application programming interface (API) ....................... 45
Table 14: UltraSparc functional units .............................................................. 47
Table 15: UltraSparc instruction types ............................................................. 47
Table 16: MIPS R4000 functional units .......................................................... 47
Table 17: MIPS R4000 instruction types ........................................................ 48
Table 18: MIPS R10000 functional units ......................................................... 48
Table 19: MIPS R10000 instruction types ....................................................... 48
Table 20: Interval stress test results ............................................................... 54
Table 21: Interval Newton test results ............................................................ 56
Table 22: Interval replacement test results ..................................................... 56
List of Figures

Figure 1: An equation that causes numerical errors ........................................... 5
Figure 2: An equation that causes a rounding error ........................................... 5
Figure 3: IEEE floating point format ................................................................. 12
Figure 4: C representation of an interval ......................................................... 15
Figure 5: Standard interval notation ................................................................. 15
Figure 6: Outward rounding ........................................................................... 16
Figure 7: Calculation of $c = a \oplus b$ for a typical interval operation $\oplus$ ........... 17
Figure 8: Problems to be solved using the interval Newton method ................... 53
Abstract

Most computer applications require reliable and accurate calculations, although they often fail in this regard when working with real-valued numbers. Interval arithmetic provides reliability and accuracy by computing a lower and upper bound in which each result is guaranteed to reside. By representing numbers as ranges of possible values rather than as point values, it also provides a mathematical basis for implementing many scientific and commercial applications. However, even though interval arithmetic is widely applicable and relatively efficient, current software implementations fail to provide the performance required for it to gain widespread acceptance.

This thesis presents a new approach to improving interval performance. It introduces hardware and instruction set modifications to conventional processors which focus on some of the most costly aspects of interval arithmetic. These modifications are shown to have little or no impact on the processors’ cycle time and do not require much area. General enhancements such as deep pipelines and multiple functional units are relied upon to absorb the cost of extra floating point operations.

Simulation shows that these enhancements, applied to the DLX architecture, can improve the performance of some interval operations by over 400 percent, and the performance of highly optimized interval applications by 33 to 50 percent. More importantly, these improvements bring performance of interval arithmetic to within a factor of three of equivalent floating point implementations. This is low enough for interval arithmetic to be used in many conventional computer applications.
Chapter 1

Introduction

Humanity is becoming increasingly dependent on the successful operation of computers. They are used to design, test, and later control devices, machinery, and even the buildings in which we live. The high performance and apparent reliability of computers have made previously impossible tasks commonplace. Rapid increases in computer capabilities have continued this trend. As computers become faster and less expensive, they are called upon to do more, and the accuracy and reliability of computations becomes even more important. However, computations involving real-world data, and in most cases any computations involving real numbers, are almost guaranteed to contain inaccuracies. Thus, there is a growing need for reliable and validated computing methods.

1.1 The Price of Failure

Embedded computer control systems make many things possible. But in doing so, they become essential components of the systems they control. Many modern warplanes have such complicated aerodynamics that they cannot even fly without the constant guidance of their onboard computers. In this case, a major computer failure almost certainly results in a crash. Yet even slight errors could be disastrous, especially in planes capable of nap-of-the-earth and/or transsonic flying.

A subtle, yet potentially very dangerous, failure is that of incorrect results due to limitations of the arithmetic systems used for calculations. Small errors can accumu-
late, sometimes rapidly, and produce highly inaccurate results [8]. Two disasters caused by such cumulative arithmetic failures are the failure of a Patriot missile during the Persian Gulf War (which resulted in the death of 28 American soldiers) and the explosion of an Ariane rocket forty seconds after takeoff in 1996 [2].

1.2 Arithmetic Errors in Multiuse Computer Systems

Arithmetic failures in embedded computers can be dramatic and immediate. A plane may crash, a rocket may veer off course, or an engine may fail to operate. Designers of these systems usually understand the price of failure, though, and they attempt to design systems that are somewhat failsafe. Redundancy is added, error checking is done, and the effect of rounding errors is minimized, although this can be difficult in larger embedded systems or those that perform complex calculations.

General purpose systems and multiuse applications such as computer aided design and simulation systems are more difficult to design in a failsafe manner, especially since arithmetic errors are highly data dependent. For instance, a computer aided design system might be able to design automobiles reliably in many cases, but might fail if the ratio of wheelbase to total length exceeded a certain value (e.g., due to catastrophic cancellation). Thorough testing could probably uncover this problem, but it is usually impossible to test all possible values that may be input into a system. It can also be very difficult to foresee every way in which a general-purpose or multiuse computer system will be used or which uses may produce unacceptable arithmetic errors.
1.3 The Role of Simulation and Modeling

Simulation and modeling play an important role in engineering today. Not only is it less expensive and faster to design using these techniques, but the results of mistakes are usually much less severe. If the simulation is accurate and complete, a mistake merely results in extended design time rather than later system failure. This allows tolerances to be reduced while improving designs. Arithmetic errors in this case may not result in immediate failure, but rather introduce latent faults that can cause failures months or years later. The more that designs “push the edge,” facilitated by advanced simulation and modeling tools, the more they are susceptible to later catastrophic failure due to arithmetic errors in the design process [6].

1.4 Handling Arithmetic Errors

It is important that arithmetic errors be handled in a general purpose fashion. Although specific causes of arithmetic errors, such as denominators in a particular equation that are known to approach zero under certain conditions, can be handled individually, it can be difficult to identify the conditions to test. Furthermore, it is seldom feasible to predict all possible uses for a given system, much less all combinations of inputs.

Without some indication of the reliability of a particular answer, designers using computer aided design tools may be relying on values that are totally incorrect. The same holds true for suitably advanced embedded systems that rely on real-world models or complex calculations—as soon as an unusual or unforeseen situation occurs, their results may be suspect.
1.5 Reliable computing

The fundamental problem with most real-number computations is that their accuracy is not guaranteed. Increasing precision does not prevent this. Small errors can accumulate rapidly, and limitations in the representation of numbers can quickly cause completely wrong results.

\[ f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + a/(2b) \]

**Figure 1: An equation that causes numerical errors**

Consider the example in Figure 1 [26]. For \( a = 77617.0 \) and \( b = 33096.0 \), this equation yields \( f \approx 1.17260 \) when solved using single precision, double precision, and extended precision arithmetic. Increasing the precision seems to validate the results. However, the correct answer is actually \( f \approx -0.827396 \times 10^{-17} \).

\[ f = 1.0 \times 10^{30} + 100.0 - 5.0 \times 10^{29} - 5.0 \times 10^{29} - 10.0 \]

**Figure 2: An equation that causes a rounding error**

The limitations of point-valued computer arithmetic are shown more clearly in Figure 2. Obviously, the correct solution to Figure 2 is \( f = 90.0 \). However, the second term is lost in either single-precision or double-precision floating point arithmetic, causing \( f = -10.0 \) to be computed instead. Limitations inherent to the representation of real numbers skew the results with no indication of the inaccuracy. In Figure 2, exchanging the terms can rectify the problem. This might not be desirable, however, if
the numbers represent measured values. Perhaps the larger values should dwarf the smaller ones, and the result should not be precise enough to distinguish between $f = -10.0$ and $f = 90.0$ anyway.

Some forms of computation represent real numbers not as simple values, but rather as all of the possible values which they may hold. These provide a guarantee of accuracy. Such a computational model bounds not only arithmetic errors, but also errors due to limitations in measuring and control devices. In fact, such computational models do more than just constrain errors. Properly bounded results of operations on known values (even bounded values) can be shown to be equivalent to a mathematical proof [36]. This can lead to entirely new algorithms to solve problems.

1.6 The Need for Speed

It has been postulated that, in order for general-purpose error bounding arithmetic to gain widespread acceptance, its performance must be no less than one fifth that of equivalent floating point algorithms [16]. This requirement is easy to justify for advanced embedded systems that require complex calculations such as flight control systems, robotic sensing, or embedded systems that rely on simulation and modeling. Speed is also important in general purpose computer systems. Performance gains due to modern processor developments can offset some of the increased cost, but there will always be applications that are "pushing the envelope," especially in high-end simulation and modeling tools. People are also reluctant to give up performance. The performance of personal computers is increasing by an order of magnitude every few years,
at which point the old systems tend to be worth virtually nothing. An order of magnitude performance penalty makes a high end system perform like a system that is so slow that it is destined for disposal. There is also a monetary consideration—extra processing overhead in computer aided design systems translates to longer design times or at least increased design costs.

1.7 The Interval Arithmetic Solution

Interval arithmetic provides a general methodology for bounding errors. The concept is simple: represent all real numbers not as discrete values, but as ranges (intervals) in which the actual (i.e., correct) value is known to reside [21] [1]. An interval’s width indicates the maximum possible error. Since all interval solutions to a problem contain the correct answer, results that are too wide can be narrowed by solving the problem (e.g., reducing its equations) in a different way and intersecting the results.

Interval arithmetic is also an enabling technology for several new algorithms, some of which solve problems previously thought impossible (or at least impractical) to solve with a computer. The interval Newton algorithm is one such algorithm, which efficiently finds or proves non-existence of all roots of any continuously differentiable function [36]. Interval solutions to non-linear global optimization problems also exist.

1.8 Performance of Interval Arithmetic

Interval arithmetic is faster than other error constraining techniques (such as various high-precision or exact methods), and can use arithmetic precisions that allow efficient processing. However, it still has a serious performance penalty. Estimates indicate that
interval computations implemented in software are typically tens to hundreds of times slower than the equivalent floating point computations [5]. A great deal of overhead comes from using function calls to access the interval arithmetic operations [40], but experiments using the DLX architecture [11] have shown that, even without this overhead, interval operations require 10 to 30 times as many instruction cycles as the equivalent floating point operations [39].

1.9 Architectural Support for Interval Arithmetic

There are several ways to reduce the overhead of interval arithmetic in computers. Interval-specific coprocessors that very efficiently handle intervals have been proposed [28] [32] [30] [29]. As improvements in chip design processes allow more features to be added to existing microprocessors, it is conceivable that these could eventually become standard additions to mainstream processors. However, it is unlikely that this will happen until interval arithmetic is more widely used, and this growth is limited by the overhead of performing interval operations on existing machines.

Fortunately, there are a number of less aggressive improvements in processor design that can benefit interval operations greatly. Most are general improvements that benefit other operations as well and already exist in one or more experimental and/or production processors. The addition of a few interval specific operations can provide significant improvements in the performance of several of the more time consuming operations. These additional operations can be compared to the multimedia extensions that are becoming common in modern processors [20] [34] [22].
Experiments with the DLX processor architecture [11] have shown the value of each of these improvements. Overall, they can speed up interval applications by up to 50% and, more importantly, reduce the overhead of interval operations below the cutoff point for common acceptance. By ensuring that at least some of these improvements become common in processor design, the future of interval arithmetic will be made much brighter and reliable computing will take a big step forward.

1.10 Overview

The goal of this thesis is to improve the performance of interval operations in order to help make reliable computing generally available. A set of architectural enhancements, meant to have minimal impact on current processor design, are proposed and evaluated to determine if they might make this possible.

Chapters 2 through 4 contain background material. Chapter 2 is an overview of number systems and how they affect computations. Chapter 3 is an overview of interval arithmetic. Chapter 4 examines current processor architectures and trends, as well as some previously proposed hardware support for intervals.

Chapter 5 proposes a set of architectural enhancements in order to improve the performance of interval operations. Chapter 6 investigates the processor hardware required to implement the enhancements. Chapter 7 details the simulation environment used to evaluate these enhancements. Chapter 8 reports measured performance benefits of these enhancements. Chapter 9 discusses possible consequences of this research and suggests areas for future research.
Appendix A shows implementations of some common interval operations in C. Appendix B shows the equivalent operations using DLX assembly language. Appendix C details the proposed interval extensions as they apply to the DLX architecture. Appendix D describes the floating point pipeline model used in the enhanced DLX architecture. Appendix E lists the interval support types and macros that have been defined for the enhanced DLX architecture.
Chapter 2

Number Representations

Computers can quickly perform integer arithmetic, which usually produces exact results. Real-world problems, however, typically involve non-integer data and require approximations. Treating approximate values as point values is dangerous, since small errors can quickly accumulate into large discrepancies.

2.1 Exact Methods

There are various ways to represent real numbers exactly. For instance, numbers may be represented as sums of fractions or symbolically as equations involving fractions as well as irrational numbers such as $\pi$ or $e$. This is the approach taken by symbolic arithmetic packages such as those used in Mathematica or Maple [4] [42]. These methods are computationally expensive in terms of program speed and size and have limited applicability to real-world problems. They also do not address errors that are implicit to transferring data. A system applying exact methods to real-world problems still needs to input and output data in some form, which typically introduces errors. More errors are typically introduced by converting these values to the finite internal representation as well.

2.2 Inexact Methods

Fortunately, an approximation is usually good enough. By placing limits on precision and range, calculations can be performed much faster. Floating point representations
are typically used, although non-integer fixed point numbers are sometimes employed to achieve performance close to that of integer operations. Fixed point representations often have less precision and dynamic range than floating point representations and are often used when performance is critical but chip area or power are at a premium (e.g., in fixed-point digital signal processors used in cellular phones and other low-power applications that require high performance). In many of these fixed-point applications, the behavior of the system is defined using bit-exact standards so that limitations of the approximation are not treated as errors. In fact, they become requirements [41].

In most cases, however, real values are represented as floating point numbers. By far, the most common floating point representations are those that conform to the IEEE 754 floating point standard [11] [13].

2.3 Floating Point Numbers

The IEEE 754 floating point standard defines both single precision and double precision numbers. Single precision numbers are 32-bit entities consisting of 8 exponent bits, 23 significand bits, and 1 sign bit. Double-precision numbers are 64-bit entities consisting of 11 exponent bits, 52 significand bits, and one sign bit. Extended precision arithmetic with greater ranges and/or precisions is also allowed by the standard. The standard format for floating point numbers is as shown in Figure 3 ("S" represents the sign bit).

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
</table>

Figure 3: IEEE floating point format
For normalized numbers, there is an implied 1 before the significand, which represents a binary fraction. This increases the effective number of significand bits by one. For extremely small (i.e., denormalized) numbers and for zero, the 1 is dropped. This allows gradual underflow. Two other special cases are infinities and non-valued numbers, both of which are represented with the largest-possible value in the exponent field. A non-valued number is called a NaN, which stands for Not-a-Number.

2.4 Limitations of Inexact Methods

Inexact numeric representations are unreliable. Even the initial parameters are typically approximated due to quantization error. Results may suffer from roundoff error, such as was demonstrated in section 1.5. Other phenomena may also affect the accuracy and precision of the results. For instance, when subtracting two numbers that differ only in a few low-order bits, most of the bits cancel each other out, causing a loss of precision. This is referred to as catastrophic cancellation. There is no measure of the accuracy of each result. Inaccuracies can be cumulative, so it is possible for the results of inexact calculations to be very far from the correct results [8] [25].
Chapter 3

Interval Arithmetic

Interval arithmetic provides an efficient method for bounding errors that are inherent to inexact arithmetic [21]. It does this by representing values as ranges rather than discrete numbers, and by controlling the direction of inaccuracies. Inaccuracies cannot be avoided, but control can be exerted over them.

3.1 History of Interval Arithmetic

Interval arithmetic (and interval analysis) is not a new field. The first major publication on interval arithmetic was in the 1960s [21]. As a field, it has been kept alive for many years largely due to the efforts of a small number of researchers [38]. However, it is now starting to gain broader acceptance. Many interval analysis software packages are now freely available [5] [10] [14] [17] [18], and support for intervals is being added to FORTRAN compilers and systems [3] [16] [27].

Researchers and mathematicians have already acknowledged the need for the type of reliable computing that interval arithmetic allows. Many believe it is only a matter of time before reliable computing becomes mainstream technology. Interval arithmetic has also led to the discovery of new algorithms for solving problems, some of which were considered unsolvable just 15 years ago [36].
3.2 Interval Representations

Intervals are ranges of real numbers. They are typically represented as two floating point numbers. A C implementation is shown in Figure 4.

```c
typedef struct Interval {
    double infimum;
    double supremum;
} ;
```

**Figure 4: C representation of an interval**

In interval analysis, each interval corresponds to the set of all values that the real value being represented can possibly have. The two floating point numbers are the lower and upper bounds of that set, which contains them and all real numbers between them (±∞ indicates that the interval is unbounded in that direction). The lower bound is called the infimum and the upper bound the supremum. Intervals are often written using standard closed-interval notation. For example, Figure 5 shows a number that is known to lie between −0.00301 and −0.00299 (inclusive) using this notation.

```
[ -3.01 \times 10^{-3} , -2.99 \times 10^{-3} ]
```

**Figure 5: Standard interval notation**

The width of the interval (i.e., the distance between the interval endpoints) gives an indication of the accuracy of the result. This is useful for monitoring calculation errors such as roundoff error as well as the effects of approximation errors and errors due to non-exact inputs [1]. Intervals thus provide a useful tool for computations in which data are uncertain or take a range of values.
3.3 Outward Rounding

Although arithmetic errors cannot be avoided, it is possible to control the direction of error. This is called the rounding direction. Using this capability, interval computations are performed in a way that guarantees that the results always contain the correct answer. When the endpoints of an interval are not exactly representable, the lower endpoint is rounded towards $-\infty$ and the upper endpoint is rounded towards $+\infty$. This is referred to as outward rounding. For example, if the interval $[2.234, 2.323]$ is rounded to three decimal digits, the resulting interval is $[2.23, 2.33]$. Figure 6 shows how outward rounding can affect division results, assuming interval endpoints are precise to four decimal digits.

\[
[1.000, 1.000] \div [3.000, 3.000] = [3.333 \times 10^{-1}, 3.334 \times 10^{-1}]
\]

Figure 6: Outward rounding

3.4 Interval Operations

Many interval arithmetic operations are similar to "traditional" arithmetic operations, except that they compute the upper and lower bounds rather than just a single number. The resultant infimum is the smallest result that can be generated by applying the operation to all possible input values, rounded towards $-\infty$. Similarly, the resultant supremum is the largest such result, rounded towards $+\infty$.

A given interval arithmetic operation can typically be calculated by applying the corresponding real operator to the endpoints (rounding appropriately) and selecting
the minimum and maximum values. Figure 7 demonstrates this for a binary operator ⊗ (the infimum and supremum of a given interval, a, are denoted as inf(a) and sup(a)).

\[
\begin{align*}
\inf(c) &= \min(\inf(a) \otimes \inf(b), \inf(a) \otimes \sup(b), \sup(a) \otimes \inf(b), \sup(a) \otimes \sup(b)) \\
\sup(c) &= \max(\inf(a) \otimes \inf(b), \inf(a) \otimes \sup(b), \sup(a) \otimes \inf(b), \sup(a) \otimes \sup(b))
\end{align*}
\]

Figure 7: Calculation of \( c = a \otimes b \) for a typical interval operation \( \otimes \)

### 3.4.1 Binary Interval Operations

Typical binary operations do not actually require this many calculations. For addition and subtraction, the same pairs of infimum/supremum terms from the operands are always used to generate the infimum and supremum of the result. For multiplication or division, the signs of the input operands can be used to determine how to compute the result in two multiplies or divides, except when multiplying two intervals that cross zero, in which case four multiplies are needed, plus two compares. Table 1 shows the calculations required for the common binary interval operations.

A very important interval operation is intersection. It produces an interval that encloses all values contained by both operands (if they are disjoint, it produces an empty interval). The definition in Table 1 actually produces an illegal interval (using the standard definition of intervals) instead, so the result should be checked for validity if it matters. Intersection produces an interval no wider than the narrowest operand. This can be exploited to narrow results by simply solving an equation in multiple ways and intersecting the results (narrowing results is also called tightening or sharpening). The converse operation of intersection is convex hull (or interval hull), which produces the narrowest interval that fully contains both of its operands.
Interval division can actually be slightly more complicated than Table 1 shows. If exactly one of the divisor’s endpoints is zero, only one of the endpoints of the result needs to be ±∞. If both are zero, then the endpoints of the result can be −∞, +∞, or NaN, depending on whether the dividend is negative, positive, zero, or crosses zero.

<table>
<thead>
<tr>
<th>Operation ⊗</th>
<th>Result c (= a ⊗ b)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval a</td>
<td>Interval b</td>
<td>Infimum inf(c)</td>
</tr>
<tr>
<td>Addition</td>
<td></td>
<td>inf(a) + inf(b)</td>
</tr>
<tr>
<td>Subtraction</td>
<td></td>
<td>inf(a) − sup(b)</td>
</tr>
<tr>
<td>Multiplication</td>
<td></td>
<td>inf(a) × inf(b)</td>
</tr>
<tr>
<td>positive</td>
<td>positive</td>
<td>inf(a) × inf(b)</td>
</tr>
<tr>
<td>positive</td>
<td>negative</td>
<td>sup(a) × inf(b)</td>
</tr>
<tr>
<td>positive</td>
<td>crosses zero</td>
<td>sup(a) × inf(b)</td>
</tr>
<tr>
<td>negative</td>
<td>positive</td>
<td>inf(a) × sup(b)</td>
</tr>
<tr>
<td>negative</td>
<td>crosses zero</td>
<td>sup(a) × sup(b)</td>
</tr>
<tr>
<td>crosses zero</td>
<td>positive</td>
<td>inf(a) × sup(b)</td>
</tr>
<tr>
<td>crosses zero</td>
<td>negative</td>
<td>sup(a) × inf(b)</td>
</tr>
<tr>
<td>crosses zero</td>
<td>crosses zero</td>
<td>smaller of inf(a) × sup(b), sup(a) × sup(b) larger of inf(a) × inf(b), sup(a) × sup(b)</td>
</tr>
<tr>
<td>Division</td>
<td></td>
<td>inf(a) ÷ sup(b)</td>
</tr>
<tr>
<td>positive</td>
<td>positive</td>
<td>inf(a) ÷ sup(b)</td>
</tr>
<tr>
<td>positive</td>
<td>negative</td>
<td>sup(a) ÷ sup(b)</td>
</tr>
<tr>
<td>negative</td>
<td>positive</td>
<td>inf(a) ÷ inf(b)</td>
</tr>
<tr>
<td>negative</td>
<td>negative</td>
<td>sup(a) ÷ inf(b)</td>
</tr>
<tr>
<td>crosses zero</td>
<td>positive</td>
<td>inf(a) ÷ inf(b)</td>
</tr>
<tr>
<td>crosses zero</td>
<td>negative</td>
<td>sup(a) ÷ sup(b)</td>
</tr>
<tr>
<td>any</td>
<td>crosses zero</td>
<td>−∞</td>
</tr>
<tr>
<td>Intersection</td>
<td></td>
<td>larger of inf(a), inf(b) smaller of sup(a), sup(b)</td>
</tr>
<tr>
<td>Convex Hull</td>
<td></td>
<td>larger of inf(a), inf(b) smaller of sup(a), sup(b)</td>
</tr>
</tbody>
</table>

Table 1: Binary interval operations

### 3.4.2 Interval Scalar Decompositions

Several operations decompose intervals into scalar values. These access the infimum and supremum, compute the midpoint or width of an interval, and compute the furthest
(magnitude) or closest (mignitude) that an interval is to zero. The scalar decomposition operations are shown in Table 2. For midpoint, subtraction and division are rounded towards $+\infty$ and addition towards $-\infty$ to insure that the result lies in the interval. Width is rounded towards $+\infty$ to insure that the result spans the interval.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Calculation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>infimum</td>
<td>inf()</td>
<td>inf([1,3]) = 1</td>
</tr>
<tr>
<td>supremum</td>
<td>sup()</td>
<td>sup([1,3]) = 3</td>
</tr>
<tr>
<td>midpoint</td>
<td>inf() + (sup() - inf()) / 2</td>
<td>midpoint([1,3]) = 2</td>
</tr>
<tr>
<td>width</td>
<td>sup() - inf()</td>
<td>width([-3,0]) = 3</td>
</tr>
<tr>
<td>magnitude</td>
<td>larger of</td>
<td>magnitude([-2,1]) = 2</td>
</tr>
<tr>
<td></td>
<td>[inf(), sup()]</td>
<td>magnitude([-2,-1]) = 1</td>
</tr>
<tr>
<td>magnitude</td>
<td>0 if interval contains zero, otherwise smaller of [inf(), sup()]</td>
<td>magnitude([-2,1]) = 0</td>
</tr>
</tbody>
</table>

Table 2: Scalar interval decompositions

<table>
<thead>
<tr>
<th>Interval</th>
<th>infimum calculation</th>
<th>Supremum calculation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>magnitude()</td>
<td>magnitude()</td>
<td>[-3,1] = [0,3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negation</td>
<td>-sup()</td>
<td>-inf()</td>
<td>[-4,3] = [-5,-4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square</td>
<td>inf() × inf()</td>
<td>sup() × sup()</td>
<td>sqrt([2,3]) = [4,9]</td>
</tr>
<tr>
<td>positive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>negative</td>
<td>sup() × sup()</td>
<td>inf() × inf()</td>
<td>sqrt([-3,-2]) = [4,9]</td>
</tr>
<tr>
<td>crosses zero</td>
<td>smaller of inf() × inf(),</td>
<td>larger of inf() × inf(),</td>
<td>sqrt([-2,1]) = [1,4]</td>
</tr>
<tr>
<td></td>
<td>sup() × sup()</td>
<td>sup() × sup()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Root</td>
<td>sqrt(inf())</td>
<td>sqrt(sup())</td>
<td>sqrt([1,4]) = [1,2]</td>
</tr>
<tr>
<td>positive</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>negative</td>
<td>NaN</td>
<td>NaN</td>
<td>sqrt([-4,-1]) = NaN</td>
</tr>
<tr>
<td>crosses zero</td>
<td>0 or NaN</td>
<td>sqrt(sup())</td>
<td>sqrt([-1,1]) = [0,1]</td>
</tr>
</tbody>
</table>

Table 3: Unary interval operations

3.4.3 Unary Interval Operations

Common unary interval operations include absolute value, negation, square, and square root. These are shown in Table 3. There is some debate in the interval community about how to calculate the infimum of the square root of an interval that crosses zero. Though
not technically correct, intersecting the solution with the set of real numbers tends to be
more useful to applications and is less likely to contaminate future calculations. Either
way, extra error checking is usually required when interval square root is performed.

3.4.4 Interval Comparisons

Comparisons are more difficult than with point-valued operations. There are certainly
ture, possibly true, and set variations of most of them. Interval comparison operators
are shown in Table 4.

<table>
<thead>
<tr>
<th>Comparison operator $\otimes$</th>
<th>Calculation of $a \otimes b$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set comparisons</strong></td>
<td></td>
</tr>
<tr>
<td>set equals $(\equiv)$</td>
<td>$\inf(a) = \inf(b)$ and $\sup(a) = \sup(b)$</td>
</tr>
<tr>
<td>subset of $(\subseteq)$</td>
<td>$\inf(a) \geq \inf(b)$ and $\sup(a) \leq \sup(b)$</td>
</tr>
<tr>
<td>superset of $(\supseteq)$</td>
<td>$\inf(a) \leq \inf(b)$ and $\sup(a) \geq \sup(b)$</td>
</tr>
<tr>
<td>proper subset of $(\subsetneq)$</td>
<td>$(a \subseteq b)$ and not $(a \equiv b)$</td>
</tr>
<tr>
<td>proper superset of $(\supsetneq)$</td>
<td>$(a \supseteq b)$ and not $(a \equiv b)$</td>
</tr>
<tr>
<td>disjoint from $(\not\subset)$</td>
<td>$\inf(a) &gt; \sup(b)$ or $\sup(a) &lt; \inf(b)$</td>
</tr>
<tr>
<td><strong>Possibly true comparisons</strong></td>
<td></td>
</tr>
<tr>
<td>possibly equals</td>
<td>$\inf(a) \leq \sup(b)$ and $\sup(a) \geq \inf(b)$</td>
</tr>
<tr>
<td>possibly not equal to</td>
<td>$\inf(a) \neq \sup(a)$ or $\inf(a) \neq \inf(b)$ or $\sup(a) \neq \sup(b)$</td>
</tr>
<tr>
<td>possibly less than</td>
<td>$\inf(a) &lt; \sup(b)$</td>
</tr>
<tr>
<td>possibly less than or equal</td>
<td>$\inf(a) \leq \sup(b)$</td>
</tr>
<tr>
<td>possibly greater than or equal</td>
<td>$\sup(a) \geq \inf(b)$</td>
</tr>
<tr>
<td><strong>Certainly true comparisons</strong></td>
<td></td>
</tr>
<tr>
<td>certainly equals</td>
<td>$\inf(a) = \sup(b) = \inf(b) = \sup(b)$</td>
</tr>
<tr>
<td>certainly not equal to</td>
<td>same as $(a \not\Rightarrow b)$</td>
</tr>
<tr>
<td>certainly less than</td>
<td>$\sup(a) &lt; \inf(b)$</td>
</tr>
<tr>
<td>certainly less than or equal</td>
<td>$\sup(a) \leq \inf(b)$</td>
</tr>
<tr>
<td>certainly greater than</td>
<td>$\inf(a) &gt; \sup(b)$</td>
</tr>
<tr>
<td>certainly greater than or equal</td>
<td>$\inf(a) \geq \sup(b)$</td>
</tr>
</tbody>
</table>

Table 4: Interval comparisons
3.5 Software Support for Intervals

There are several software packages that support interval arithmetic, including BIAS/PROFIL [18], INTLIB [14], and C-XSC [17]. Efforts are currently underway to add built-in support for interval arithmetic to FORTRAN [3] [16] and other languages. C implementations of common interval operations are shown in Appendix A.

Although naive application of interval arithmetic can lead to wide intervals, many efficient algorithms that produce narrow intervals have been developed [6] [10]. Using these and other tools, interval arithmetic has been successfully applied to a wide range of scientific applications [15] [19].

However, current software implementations of interval operations are not fast enough. It has been postulated that, for interval arithmetic to have wide acceptance, its performance overhead must be no more than two to five times that of regular floating point arithmetic [16]. The overhead for current implementations has been estimated at 20 to 100 times that of floating point [5].

3.6 The Performance Cost of Intervals

The following six factors make interval operations expensive:

- function/subroutine overhead,
- increased size of the data,
- computation of two or more results per interval operation,
- changing of rounding direction,
- data dependent selection of how some operations are performed, and
- handling special cases.

Taken as a whole, these make interval operations very expensive. However, many general improvements to processors and tools are being implemented that
address each of these costs, except those associated with the changing of rounding direction.

3.6.1 Overhead due to Functions/Subroutines

Using software libraries for interval arithmetic incurs additional overhead since they are accessed via function calls. The cost of making a function call for every interval operation is significant [40]. This overhead can be largely eliminated through compilers with native interval support or through inline functions that are customized for particular architectures and compilers. Techniques that eliminate function overhead may also cause a significant increase in code size, however. It is thus desirable for interval operations to be implemented using a minimal amount of code.

3.6.2 Overhead due to Extra Interval Data

The increased data size of intervals (using pairs of values) causes more registers and more memory transfers to be used. Newer processors tend to have more memory, more and larger registers, and wider data buses, which offsets this cost. Eliminating function calls will also reduce this overhead since operand and result passing as well as local variables associated with functions cause increased register and stack usage.

3.6.3 Overhead due to Computing both Endpoints

Computing both endpoints results in extra computations. However, newer processors tend to employ more parallelism (e.g., through deep pipelines, very long instruction words, or superscalar execution), which helps offset the cost of the extra computations
required for intervals. This is especially true for applications that otherwise limit what can be done in parallel. In other words, the extra computations may be able to occupy execution slots that would be otherwise vacant due to stalls or failure to schedule an applicable instruction, as long as the pipeline is not stalled by requesting a change in rounding.

3.6.4 Overhead due to Changing Rounding Direction

Most processors are not designed to have their rounding direction changed very often. It is usually changed through an inefficient "back door" such as loading a floating point status register or writing a control location in memory, causing the floating point pipeline to be flushed (i.e., all pending floating point operations are allowed to complete) before it will accept new operations. This introduces additional latency and prevents the simultaneous computation of the infimum and supremum of an interval, even though the processor may otherwise have sufficient parallelism to do so. Some machines have been designed without this restriction, though. Specifically, to remove execution interlocks, some researchers have broken the dependencies between system state and the processor execution units by duplicating that state, including floating point control, as data that accompanies each instruction in the execution pipeline [35].

3.6.5 Overhead due to Data-dependent Operation Selection

Software implementations of interval operations require many data dependent operations because they must choose between lower and upper bounds. This can introduce significant delays in deeply pipelined processors, which have long branch latencies.
Predicated branches combined with speculative execution, such as will be present in Intel's IA-64 architecture [9], may be able to eliminate some of these stalls. Another approach is to use conditional instructions to reduce the number of branches. This approach is used in the ARM instruction set [7].

3.6.6 Overhead due to Interval Special Cases

Special cases, such as infinities or NaNs, can make interval operations expensive as well. Most processors support floating point special cases in some way (such as by allowing floating point exceptions to be raised). Special cases are handled differently for intervals, however. Floating point handling of special cases is generally disabled while using intervals, since it may not be desirable to have floating point exceptions raised twice and there are some cases when a floating point exception would be raised by an intermediate result that is discarded anyway. Special case handling for intervals is rather complicated, and is an area of active research [23] [24].
Chapter 4

Current Processor Architectures and Research

Although interval arithmetic can have a significant performance cost, current trends in processor design may help offset that cost. Carefully used, hand-optimized interval libraries may soon be fast enough that interval arithmetic may be able to get a foothold in conventional computer applications. If care is taken now to insure that future designs continue to benefit intervals, processors may soon compute intervals efficiently enough to support their widespread use.

4.1 Processor Organization

Modern processors use techniques that allow them to do several things in parallel, so that programs can execute faster. Pipelined architecture do this by splitting instructions into stages and executing one stage each from multiple instructions simultaneously. The current trend is towards deeper pipelines. Each stage executes faster, but more stages are required per instruction. As long as the pipeline stays full, the throughput of the processor is one instruction per cycle, so the full benefit of the shorter clock cycle is realized. Superscalar processors can get even higher throughput by issuing multiple instructions per cycle to multiple functional units.

Superscalar processors are a mixed blessing for intervals. Despite efforts by researchers to keep the functional units busy [12], many of them are often left idle. These can absorb some of the overhead of intervals. However, if changing of rounding modes causes the processor pipeline to flush, the overhead can be severe. As pipelines
become longer, the cost of flushing them goes up. Support for intervals may soon be at a critical juncture. Up to now, processors have been getting increasingly efficient at processing intervals, but this effect could begin to reverse the trend.

Deep pipelines need not cause intervals such a performance penalty, however. Branches used to cause pipeline stalls. Today, speculative execution and predicated branches [9] can often avoid stalls. Data hazards used to cause stalls, although dynamic scheduling techniques such as Tomasulo's algorithm can eliminate write-after-write and write-after-read hazards, and can reduce the number of read-after-write hazards as well. Similarly, steps can be taken to avoid pipeline flushes and stalls when rounding modes and other processor controls are changed [35].

4.2 Separation of Architecture from Implementation

Another trend in modern processors is the separation of the instruction set architecture from the implementation details of the processor. Microcoding, in which instructions are decoded into internal sequences of control words, is a primitive example of this. Processor designers have moved away from microcoding to boost performance, but are finding better ways to isolate architectures from implementations.

This trend may be partly due to competition among processor vendors. Several processor families, such as X86, ARM, and SPARC, are provided by multiple vendors. Each vendor wants to add its own value, and needs to be able to enhance performance to keep up with, or stay ahead of, competitors. For instance, Intel has made drastic changes in the basic architecture of their X86-based processors while maintaining
backward compatibility (and the instruction set has remained essentially the same since
the 80386 was introduced). The IA-64 architecture will continue this trend, and Intel
has hinted that the technology to map the old instruction onto IA-64 long instruction
words will be pretty novel [9].

4.3 More flexibility

Modern processors tend to have greater numbers of registers, and larger register sizes.
This can reduce function overhead by avoiding stack accesses, since registers can be
used to pass parameters and results and more scratch registers are available. It also
allows compilers to optimize across larger blocks of code. This allows programs that
use more data, or data in a more complex fashion, to be optimized.

There is also a trend toward larger word sizes. The associated wider buses offset
the cost of increased data requirements. Wider instructions increase the available
opcode space, allowing more specialized instructions, more orthogonality in existing
instructions, more operations that work on multiple registers, or most likely all three.

4.4 Faster floating point

The performance of processors on floating point calculations has not always been a
focus of processor designers. Significantly more area is now available on processors,
though. In addition, processors are being rated for floating point performance through
measures such as SPECfp. Thus, floating point performance is getting more attention.
Although integer pipelines are being lengthened to boost clock rates, enough extra
hardware is being dedicated to floating point operations that the floating point pipelines
are actually getting shorter. This can be a boon to intervals, both by avoiding stalls due to data hazards and possibly by allowing floating point pipelines to be emptied faster.

4.5 Specialized Hardware

Even with 32-bit bus architectures, there is a growing trend in the processor industry to add specialized hardware and instruction set extensions such as multimedia extensions. This is a way to add value to the device by targeting specific applications and libraries which can cause a demand for the processor. For the most part, these extensions go hand in hand with the set of libraries they were designed to enhance.

4.6 Dedicated Interval Processors

Significantly improving the performance of interval arithmetic requires some form of architectural improvements [31]. Previous research has demonstrated the feasibility of coprocessors for variable-precision [28] [32] and staggered [30] interval arithmetic, as well as custom interval arithmetic units that can be added to existing processors [29]. In the same way that coprocessors introduced native floating point support to systems whose central processing unit did not have it, dedicated interval arithmetic units and coprocessors can add native interval support.

These units increase system cost and add a great deal of specialized hardware that does not speed up most existing applications, although as process improvements allow more features to be packed into each processor, this will become less of an issue. Given sufficient demand for interval arithmetic, native interval units will eventually be found on many processors.
Chapter 5

Interval Support Instructions

Rather than designing specialized hardware which processes intervals as rapidly as possible, this thesis investigates how some of the more costly components of interval processing can be made faster. General enhancements like deep pipelines superscalar execution are relied upon to help absorb the cost of extra floating point operations. This approach requires only a few interval-specific instructions and some features that are more widely applicable and have already been implemented in some processors. A summary of the proposed enhancements is shown in Table 5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set rounding mode</td>
<td>SRND</td>
<td>Set rounding mode as specified</td>
</tr>
<tr>
<td>Initiate interval multiply</td>
<td>ITMD</td>
<td>Set rounding mode, set up operands for multiply</td>
</tr>
<tr>
<td>Initiate interval divide</td>
<td>ITDD</td>
<td>Set rounding mode, set up operands for divide</td>
</tr>
<tr>
<td>In-place absolute value</td>
<td>IABSD</td>
<td>Compare signs of two numbers and make both positive</td>
</tr>
<tr>
<td>Conditional move</td>
<td>MOVDT</td>
<td>Move if floating point status is true/false</td>
</tr>
<tr>
<td></td>
<td>MOVDF</td>
<td></td>
</tr>
<tr>
<td>Divide by two</td>
<td>HALFD</td>
<td>Quickly divide a floating point number by two</td>
</tr>
<tr>
<td>Negate and swap</td>
<td>NEGSD</td>
<td>Negate and exchange two floating point numbers</td>
</tr>
</tbody>
</table>

Table 5: Summary of proposed interval extensions

Only double-precision versions of the new instructions are shown. All but SRND have single-precision versions as well.

5.1 The DLX Architecture

The DLX architecture [11] is used as the basis for designing the new interval support instructions. The DLX architecture with the new instructions is referred to as enhanced DLX. DLX is a RISC architecture created by combining the features of a number of
current processor designs, so modifications to the DLX architecture are likely to apply to other modern processors as well.

For the code examples in this chapter, input operands start at the F4 register (and proceed through F5, F6, etc.). Output operands start at F4 if the code invalidates the current contents of any registers. If not, output operands start at F0. DLX uses register pairs for double-precision numbers, which is why no odd-numbered registers are shown. For example, double-precision references to register F0 actually affect both F0 and F1. DLX defines the R0 integer register to always contain the value zero—a double precision zero can be created by copying R0 and extending the result.

The DLX architecture uses delayed branches—the following instruction is always executed regardless of whether the branch is taken. Descriptions of several common DLX instructions are given in Table 6.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD Fd, Fa, Fb</td>
<td>Add</td>
<td>Fd ← Fa + Fb</td>
</tr>
<tr>
<td>SUBD Fd, Fa, Fb</td>
<td>Subtract</td>
<td>Fd ← Fa - Fb</td>
</tr>
<tr>
<td>MULTD Fd, Fa, Fb</td>
<td>Multiply</td>
<td>Fd ← Fa × Fb</td>
</tr>
<tr>
<td>DIVD Fd, Fa, Fb</td>
<td>Divide</td>
<td>Fd ← Fa ÷ Fb</td>
</tr>
<tr>
<td>GED Fa, Fb</td>
<td>Greater or Equal</td>
<td>FPSR ← (Fa ≥ Fb)</td>
</tr>
<tr>
<td>LED Fa, Fb</td>
<td>Less or Equal</td>
<td>FPSR ← (Fa ≤ Fb)</td>
</tr>
<tr>
<td>BFPF addr</td>
<td>Branch if FP false</td>
<td>if FPSR, PC ← addr</td>
</tr>
<tr>
<td>J addr</td>
<td>Branch always</td>
<td>PC ← addr</td>
</tr>
<tr>
<td>MOVI2FP Fd, Rs</td>
<td>Integer to Float</td>
<td>Fd ← (float)Rs</td>
</tr>
<tr>
<td>CVTF2D Fd, Fs</td>
<td>Float to Double</td>
<td>Fd ← (double)Fs</td>
</tr>
</tbody>
</table>

Table 6: Some traditional DLX instructions

Example implementations of interval operations using the DLX architecture without interval enhancements are given in Appendix B. A write floating point control
word (WFPCW) instruction was also needed for these. This stalls the processor until the floating point pipe is empty and sets rounding mode towards 0, $+\infty$, $-\infty$, or nearest.

### 5.2 SRND Instruction

The set rounding mode (SRND) instruction implies architectural support for rounding mode in the instruction pipeline. Rounding can be set to Near, Down, Up, DownOnce, UpOnce, DownUp, UpUp, or UpDown. The rounding mode automatically changes after a floating point operation, and changing of rounding modes does not introduce any delay. The rounding mode transitions are shown in Table 7. There must also be a way to save and restore the rounding mode during a context switch.

<table>
<thead>
<tr>
<th>Rounding mode</th>
<th>Round Toward</th>
<th>Next Rounding mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>Near</td>
<td>nearest</td>
<td>Near</td>
</tr>
<tr>
<td>Down</td>
<td>$-\infty$</td>
<td>Down</td>
</tr>
<tr>
<td>Up</td>
<td>$+\infty$</td>
<td>Up</td>
</tr>
<tr>
<td>DownOnce</td>
<td>$-\infty$</td>
<td>Near</td>
</tr>
<tr>
<td>UpOnce</td>
<td>$+\infty$</td>
<td>UpOnce</td>
</tr>
<tr>
<td>DownUp</td>
<td>$-\infty$</td>
<td>UpOnce</td>
</tr>
<tr>
<td>UpUp</td>
<td>$+\infty$</td>
<td>UpOnce</td>
</tr>
<tr>
<td>UpDown</td>
<td>$+\infty$</td>
<td>DownOnce</td>
</tr>
</tbody>
</table>

**Table 7: Rounding mode transitions**

Interval addition and subtraction simply require a SRND instruction to set the rounding mode to DownUp followed by two operations to compute the endpoints. Interval width requires only a SRND and a subtract instruction. These are shown in Table 8.
Table 8: Addition, subtraction, and width in enhanced DLX

<table>
<thead>
<tr>
<th>Addition</th>
<th>Subtraction</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRND DownUp</td>
<td>SRND DownUp</td>
<td>SRND</td>
</tr>
<tr>
<td>ADD D F0, F4, F8</td>
<td>SUB D F0, F4, F10</td>
<td>SUB D F0, F6, F4</td>
</tr>
<tr>
<td>ADD D F2, F6, F10</td>
<td>SUB D F2, F6, F8</td>
<td></td>
</tr>
</tbody>
</table>

5.3 IIMD Instruction

The initiate interval multiply (IIMD) instruction sets the rounding mode to DownUp and conditionally exchanges the infimum and supremum of its input operands based on their signs. An IIMD instruction and two floating point multiplies can complete an interval multiply, as shown in Table 9.

To avoid specifying four arguments to IIMD, it is assumed that adjacent pairs of double-precision numbers are used to represent intervals. This should not by itself cause register assignment problems, since to a compiler this is just a register constraint. Adding this constraint is not trivial, though. The current implementation of the IIMD instruction was modified to accept four arguments instead.

Special cases (±∞ or NaN in an endpoint) and cases where both operands cross zero must be handled by the IIMD instruction. One way to do this is to issue a special trap and complete the operation or recompute the operands in the trap handler. The current IIMD implementation fakes a trap if both operands cross zero (the DLX simulator does not support real traps), but does not handle special cases very well.
<table>
<thead>
<tr>
<th>Multiplication</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIMD F4, F8</td>
<td>IDD F4, F8</td>
</tr>
<tr>
<td>MULTD F4, F4, F8</td>
<td>DIVD F4, F4, F8</td>
</tr>
<tr>
<td>MULTD F6, F6, F10</td>
<td>DIVD F6, F6, F10</td>
</tr>
</tbody>
</table>

Table 9: Interval multiplication and division in enhanced DLX

5.4 IIDD Instruction

The initiate interval divide (IIDD) instruction does for interval division what IIMD does for interval multiplication. It sets the rounding mode to DownUp and conditionally exchanges the infimum and supremum of its input operands based on their signs so two floating point divides can complete the operation, as shown in Table 9. Implementation issues are very similar to IIMD. However, no trap is needed when both operands cross zero since the result is always unbounded in at least one direction (for “sharp” division) or in both directions (for “simple” division) [3]. IIDD also must check when an endpoint of the denominator is exactly zero.

5.5 IABSD Instruction

The in-place absolute value (IABSD) instruction computes the absolute value of two registers simultaneously and stores the results back in the same registers. The destinations are not explicitly specified, since that would require two destination fields and it is likely that there is a better use for the limited opcode space. In addition, IABSD sets a flag indicating whether the signs of the registers were different when the instruction began. This check is required for interval magnitude.
5.6 MOVDT and MOVDF Instructions

Conditional move instructions such as MOVDT and MOVDF are becoming common in modern processors (e.g., SPARC v.9). They would likely be used in interval multiplication and division, but only in the special trap handler(s) if IIMD and IIDD are supported. IABSD and MOVDF together allow magnitude and mignitude to be calculated quite efficiently, as shown in Table 10.

<table>
<thead>
<tr>
<th>Magnitude</th>
<th>Mignitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>IABSD F4, F6</td>
<td>IABSD F4, F6</td>
</tr>
<tr>
<td>GED F4, F6</td>
<td>BFPF _notnil</td>
</tr>
<tr>
<td>MOVDF F4, F6</td>
<td>LED F4, F6</td>
</tr>
<tr>
<td></td>
<td>MOVI2FP F4, R0</td>
</tr>
<tr>
<td>J</td>
<td>_end</td>
</tr>
<tr>
<td>CVTF2D F4, F4</td>
<td>_notnil:</td>
</tr>
<tr>
<td></td>
<td>MOVDF F4, F6</td>
</tr>
<tr>
<td></td>
<td>_end:</td>
</tr>
</tbody>
</table>

Table 10: Interval magnitude and mignitude in enhanced DLX

5.7 HALFD Instruction

The divide by two (HALFD) instruction quickly divides a floating point number in half by decrementing its exponent (for normalized numbers) or shifting the significand by one bit (for denormalized numbers). Rounding is not an issue for normalized numbers. It is assumed that rounding towards zero is acceptable for denormalized numbers. If not, a special trap could be added in case a denormalized number is encountered. HALFD is used in interval midpoint calculations, as shown in Table 11.
Table 11: Interval midpoint and negate in enhanced DLX

<table>
<thead>
<tr>
<th>Midpoint</th>
<th>Negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRND</td>
<td>NegSD F4, F6</td>
</tr>
<tr>
<td>SUBD</td>
<td>F0, F6, F4</td>
</tr>
<tr>
<td>HALFD</td>
<td>F0, F0</td>
</tr>
<tr>
<td>ADDD</td>
<td>F0, F0, F4</td>
</tr>
</tbody>
</table>

5.8 NEGSD Instruction

The negate and swap (NEGSD) instruction exchanges two register values and also toggles their sign bits. This implements a one-cycle interval negate, as shown in Table 11. The exchange is not strictly necessary (the compiler could simply be notified that the registers have changed), but could eliminate some copying if an interval multiply or divide follows.

5.9 The Enhanced DLX Instructions

The proposed instructions are designed to be fast, completing in a single execution cycle in most cases. Many of them invalidate the current contents of their source registers. This is unusual for DLX, but is inherent to the nature of IIMD and IIDD, and both IABSD and NEGSD would require two destinations otherwise. This could result in extra copying of the old register values, but these instructions should eliminate more cycles than the extra copies cost. The GCC-DLX compiler is usually intelligent enough to rearrange the code in order to avoid this copying anyway.
Chapter 6

Interval Support Hardware Requirements

The interval extensions are designed to be fast and to complete in a single execution cycle. To avoid having a negative effect on the processor clock speed, they are designed to require only minor modifications to existing hardware. The exact changes required are highly dependent on the processor implementation. For conventional high-performance microprocessors, however, any increase in area or cycle time resulting from these modifications is expected to be negligible.

6.1 General Hardware Requirements

The instructions require decoding, control, and routing circuitry, much of which can usually be shared with other parts of the processor. General floating point support is required, including the ability to disable floating point exceptions. Most processors now support IEEE 754 floating point arithmetic and therefore meet this requirement.

A number of the new instructions read and write two double-precision registers at the same time. Most high-performance processors meet this requirement. Some processors may have only one such write port, though, and adding a second write port might affect basic instruction timing. With only a single write port, some instructions take an extra cycle to complete. However, forwarding hardware may allow some stalls to be avoided. Many processors support dynamic scheduling techniques like register renaming and reservation stations (e.g., Tomasulo's algorithm) which can be used to overcome read and write port limitations.
6.2 SRND Hardware Requirements

The set rounding mode (SRND) instruction requires support for the rounding operations specified in the IEEE 754 floating point standard. One or two 4-bit multiplexers may be required to share the rounding mode settings with the floating point control word and other instructions that access it. It is also necessary for the rounding mode to travel with each instruction as it proceeds through the floating point pipeline. This requires a 4-bit latch at each floating point pipeline stage. Finally, a 4x9 transition table or (more likely) equivalent circuitry is required to specify the rounding mode transitions.

6.3 IIMD and IIDD Hardware Requirements

The initiate interval multiply and divide (IIMD and IIDD) instructions read from four double-precision registers and write up to four double-precision values. Processors that support register renaming can avoid exchanges by simply renaming registers. The write back to the register file can sometimes be avoided since succeeding instructions destroy these values, but may be needed in some cases, such as if the processor is interrupted.

If four read ports are not available, separate floating point state may have to be maintained to indicate which floating point registers are negative, zero, or special cases. The register exchanges can be implemented using endpoint selection logic (which can be implemented using 76 transistors [33]) and four 64-bit multiplexers plus latches and forwarding hardware (two more multiplexers for each multiply and divide unit). More likely, the partially-decoded register state will be combined with simple
logic (or possibly endpoint lookup tables) and used to control existing register renaming hardware. In this case, two 64-bit latches are still required as copy destinations (and perhaps two additional 64-bit latches and 64-bit multiplexers so that there is always an active copy target and one available for renaming). IIMD also must be able to generate a processor trap.

6.4 MOVDT and MOVDF Hardware Requirements

Conditional tests and register transfers are already present in essentially all processors, so the conditional move (MOVDT and MOVDF) instructions probably just require extra decode and control logic. Conditional moves are becoming popular on newer processors and are likely to be available anyway [11].

6.5 HALFD Hardware Requirements

The divide by two (HALFD) instruction requires a test for special cases. This information is likely to be available for the initiate interval multiply and divide instructions anyway. A 52-bit multiplexer is required for shifting the significand and an 11-bit subtractor is needed to update the exponent. If traps are required for denormalized numbers (in order to keep rounding modes consistent), an 11-input OR/NOR is required to detect the exception and a stored vector address is needed.

6.6 IABSD Hardware Requirements

The in-place absolute value (IABSD) instruction requires an exclusive-or for the sign bits and the ability to latch register values and force their sign bits to zero (this
probably just requires two one-bit multiplexers). IABSD writes two double-precision registers, so two register write ports, or possibly some advanced forwarding hardware, are required or else this instruction will take an extra cycle to execute.

Essentially, IABSD just sets the floating point status bit to the exclusive-or of the two sign bits and forces the sign bits to zero.

6.7 NEGSD Hardware Requirements

The negate and swap (NEGSD) instruction requires two inverters to change the sign bits plus the ability to write two double-precision registers as required by IABSD.
Simulation is a crucial element for this thesis, which involves evaluating alternative designs. Determining the performance impact of a design without implementing it in some form is very difficult since the design may have side-effects or be used in unanticipated ways. Although developing a test processor to evaluate new designs such as these is not practical, a simulator may be quickly modified.

To fully understand the impact of a design change, simulations should be used as the processor is eventually intended to be used (i.e., to run entire applications). The real value of a processor modification is usually how much faster applications run. The speedup of the processor can then be measured as the ratio of the time applications take without the change to the time they take with the change [11]. To measure speedup, cycle-accurate models are required. As long as the designs are realizable and do not affect either the fundamental clock rate of the device or the timing of other instructions, more detailed simulation is not needed.

7.1 Simulator Evaluation

Simulator choice had a significant impact on this thesis—it determined which processor would be used as the basis for the instruction set changes. A number of simulators were thus evaluated. In choosing or designing a simulator, it is important to specify the correct capabilities. The five characteristics considered for this thesis are: accuracy, precision, timing, scope, and performance.
7.1.1 Simulator Accuracy

Accuracy is how honestly the device is represented. Measures of accuracy include: Are numbers represented in the same way as the target machine? Are rounding modes modeled correctly? Are the effects of the pipeline modeled correctly?

7.1.2 Simulator Precision

Precision is the level of detail of the simulation. Measures of precision include: Are pin values modeled? Are rounding modes modeled? Are register values kept up to date or just updated at the last cycle of an instruction?

7.1.3 Simulator Timing

Timing is the unit of time modeled. Measures of timing include: Is the model accurate to a millisecond? Is the model updated every instruction cycle? Every clock cycle? Every clock phase?

7.1.4 Simulator Scope

Scope is the size of the system modeled. Measures of scope include: Is the entire processor modeled? Are other system components modeled? Is an entire system modeled (including disk drives, memory, etc.)?

7.1.5 Simulator Performance

Performance is the speed and resource usage of the model itself. Measures of simulator performance include: How fast does the simulator run? How much memory does it use? How much disk space does it require?
7.1.6 Simulator Requirements

All of the characteristics of simulators compete for a set of limited resources. Simulators that are precise and accurate may have a smaller scope. System-wide simulations usually sacrifice timing. Simulations that are complete, precise, accurate, and provide detailed timing tend to have lower performance, require more effort to program and maintain, and may be unwieldy to use since they present too much information. It is thus important not to overspecify simulator requirements. The simulator requirements for this thesis are shown in Table 12.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Required level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Correct timing and I/O</td>
</tr>
<tr>
<td>Precision</td>
<td>Provide output and total run time</td>
</tr>
<tr>
<td>Timing</td>
<td>Machine cycle</td>
</tr>
<tr>
<td>Scope</td>
<td>Run applications</td>
</tr>
<tr>
<td>Performance</td>
<td>Not an issue</td>
</tr>
</tbody>
</table>

Table 12: Simulator Requirements

To measure the performance of applications, an entire application is run all at once through the simulator, which must measure the correct number of cycles to execute from start-to-finish. Correct cycle counts require a certain level of detail (e.g., floating point pipelines must be considered). Correct outputs are needed to verify correct operation. The simulator could take days if needed to run an application—multiple host machines would be used in this case—so performance is not much of an issue.
7.2 Simulator Availability

Many simulators are freely available, but most sacrifice accuracy and precision for scope. This is especially true for newer architectures. Simple processors like the 8051 have good models, but complete, detailed simulators are not available for most modern processors. Detailed models that are available may only consider part of a device. Such is the case with Sun SPARC processors. System-level simulators such as SHADE do not provide accurate timing. More detailed simulators exist but do not model the entire processor and are not publicly available. In fact, better simulators for many processors exist but are kept proprietary by processor design companies that apparent do not want to risk losing their competitive advantage or revealing too much about their processor’s internals.

7.3 Simulator Quality

Unfortunately, the quality of free simulators is often suspect. Fundamental problems have been uncovered in the timing information for all of the simulators considered as candidates for use in this thesis. Ultimately, the selection was not based on the accuracy of a model but rather on which could be easily modified to have accurate timing.

Floating point timing accuracy is one area that was almost universally lacking in the candidate simulators. This may not be a concern of the general public. Floating point applications often have less critical timing, and rough timing estimates made by hand can often be followed by exact cycle counts gathered from actual machines. Designers of modern high-speed processors are also more reluctant than ever to release
detailed descriptions of the internal workings of their devices. This could be to avoid giving away information to competitors, because the details are too complex to explain easily, so that they have freedom to change the implementation (e.g., migrating coprocessor operations into central processing units), or some combination of the three.

7.4 Floating Point Simulation

A fundamental requirement of this thesis is the ability to model cycle-accurate timing and results of floating point operations in modern processors. These needs are pretty modest, but all of the simulators evaluated were lacking in this regard. To get accurate floating point timing, the floating point pipelines must be modeled, taking hazards and stalls into account.

There are three types of data hazards. Read after write (RAW) hazards are caused when the results of a previous operation are not available before an operation that requires them is issued. Write after write (WAW) hazards are caused when the results of a previous operation would overwrite the results of a future operation. Write after read (WAR) hazards are caused when a previous operation attempts to read a value after a future operation writes it.

Structural and control hazards must also be considered. Structural hazards are caused when resources needed to complete an operation are currently being used by another operation. Control hazards, mostly caused by branches, are the result of an instruction invalidating partially-completed future instructions which are not supposed to execute.
Hazards usually stall the floating point pipeline. During a stall, no new floating point instructions are issued until the hazard is resolved. Depending on the architecture and circumstance, floating point stalls may stall the integer pipeline as well [11].

7.5 The FP Module: a General Floating Point Model

Since accurate floating point modeling is lacking in so many simulators, a solution that can address the floating point needs of a large number of simulators is better than one that is tied to a particular simulator. The FP module was thus created to fill this need. The FP module is a general-purpose library which simulates an arbitrary floating point pipeline. It provides a simple yet flexible application programming interface (API) and is designed to be portable so that it can be compiled and linked with a wide variety of simulators. This API is shown in Table 13.

<table>
<thead>
<tr>
<th>API call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FpInitUnit</td>
<td>Initialize state (using the pipeline description file)</td>
</tr>
<tr>
<td>FpShutDownUnit</td>
<td>Clear all internal state</td>
</tr>
<tr>
<td>keyMarkFpRegBeingWritten</td>
<td>Mark register as destination of an executing instruction</td>
</tr>
<tr>
<td>keyMarkFpRegBeingRead</td>
<td>Mark register as source for an executing instruction</td>
</tr>
<tr>
<td>MarkFpRegNotBeingWritten</td>
<td>Unmark destination w/key from keyMarkFpRegBeingWritten</td>
</tr>
<tr>
<td>MarkFpRegNotBeingRead</td>
<td>Unmark source w/key from keyMarkFpRegBeingRead</td>
</tr>
<tr>
<td>isFpRegBeingWritten</td>
<td>Check if register is the destination of an executing instruction</td>
</tr>
<tr>
<td>isFpRegBeingRead</td>
<td>Check if register is the source of an executing instruction</td>
</tr>
<tr>
<td>keyInitiateFpInstruction</td>
<td>Load an instruction into the pipeline</td>
</tr>
<tr>
<td>FpExecuteCycle</td>
<td>Execute one cycle in the model</td>
</tr>
<tr>
<td>isFpOperationPending</td>
<td>Check if any instructions are in the pipeline</td>
</tr>
<tr>
<td>isFpInstructionReady</td>
<td>Check if a given instruction has sufficient resources to execute</td>
</tr>
</tbody>
</table>

Table 13: FP module application programming interface (API)

The FP module focuses on just the floating point pipeline. A description of the pipeline is loaded from a file when the model is initialized, so different pipelines can be modeled without recompiling. It simulates accesses to registers and to functional unit
resources, detecting and resolving data and structural hazards automatically. It can also respond to control hazards or to externally-stimulated data hazards. It uses callbacks (function pointers provided by the main simulator) to perform the actual arithmetic.

7.6 The FP Pipeline Description File

The FP module uses a pipeline description file to initialize its model. This file contains two sections. The first section simply lists the names of all functional units available to service floating point pipeline requests and identifies how many of each are present. The second section lists the floating point instruction types, each of their stages, and how many of each functional unit are required at each stage.

7.6.1 Pipeline Description for the Sun UltraSparc Processor

According to Sun, an UltraSparc performs single-precision divide in 12 cycles, double-precision divide in 22 cycles, and any floating point add/subtract or multiply operation in 3 cycles [34]. Division is not pipelined; the others are fully pipelined and have a throughput of one instruction per cycle. Table 14 identifies the UltraSparc floating point functional units. Table 15 describes its instruction types. In Table 15, commas separate stages and superscripted numbers indicate that a stage is repeated that many times sequentially. In the case of division, the stages are not actually repeated, and the number represents the fact that the division unit takes that many cycles to complete.

The UltraSparc floating point pipeline is fairly representative of leading-edge RISC processors. For this thesis, a non-superscalar version of the UltraSparc floating point pipeline is assumed. A definition of this pipeline is shown in Appendix D.
7.6.2 Pipeline Description for the MIPS R4000 Processor

The MIPS R4000, on the other hand, has eight functional units. Floating point addition, multiplication, and division use up to two different units in each stage [11]. Table 16 and Table 17 describe the R4000 functional units and instruction types.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Mantissa Add stage</td>
</tr>
<tr>
<td>D</td>
<td>Divide pipeline stage</td>
</tr>
<tr>
<td>E</td>
<td>Exception test stage</td>
</tr>
<tr>
<td>M</td>
<td>First stage of multiplier</td>
</tr>
<tr>
<td>N</td>
<td>Second stage of multiplier</td>
</tr>
<tr>
<td>R</td>
<td>Rounding stage</td>
</tr>
<tr>
<td>S</td>
<td>Operand shift stage</td>
</tr>
<tr>
<td>U</td>
<td>Unpack FP numbers</td>
</tr>
</tbody>
</table>

Table 16: MIPS R4000 functional units
7.6.3 Pipeline Description for the MIPS R10000 Processor

The MIPS R10000 floating point pipeline looks much like that of the UltraS-parc. It also has three-stage fully pipelined adder and multiplier pipelines, although divide is done in a unit which shares its first and last stages with the multiplier pipeline [43]. Table 18 and Table 19 describe the R10000 functional units and instruction types.

Table 17: MIPS R4000 instruction types

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ADD</td>
<td>U, S+A, A+R, R+S</td>
</tr>
<tr>
<td>FP MULT</td>
<td>U, E+M, M^3, N, N+A, R</td>
</tr>
</tbody>
</table>

Table 18: MIPS R10000 functional units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Align</td>
</tr>
<tr>
<td>AD</td>
<td>Add/N</td>
</tr>
<tr>
<td>AP</td>
<td>Pack addition results</td>
</tr>
<tr>
<td>MU</td>
<td>Multiply</td>
</tr>
<tr>
<td>MS</td>
<td>Sum/N</td>
</tr>
<tr>
<td>MP</td>
<td>Pack multiply results</td>
</tr>
<tr>
<td>DD</td>
<td>Divide</td>
</tr>
</tbody>
</table>

Table 19: MIPS R10000 instruction types

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVF</td>
<td>MU, DD^{12}, MP</td>
</tr>
<tr>
<td>DIVD</td>
<td>MU, DD^{19}, MP</td>
</tr>
<tr>
<td>FP ADD</td>
<td>AA, AD, AP</td>
</tr>
<tr>
<td>FP MULT</td>
<td>MU, MS, MP</td>
</tr>
</tbody>
</table>

48
7.7 The DLXsim Simulator

The DLXsim simulator [11] provides the base upon which the interval extensions are constructed. This simulator is easily enhanced. A core simulation module, sim.c, is responsible for most of the simulation. An assembler module, asm.c, is responsible for converting DLX source code into a format used by the simulation module. Since it accepts source files as input, no separate assembler is required. DLXsim is also built around a Tcl scripting engine, so programmable macros are also available. Since the DLX architecture is a RISC design intentionally similar to popular modern processors, it is likely that changes made to it would apply to other processors.

7.8 The GCC-DLX C Compiler

The GCC-DLX C compiler provides the compilation technology used with the interval extensions. This is simply a port of the Gnu C compiler to the DLX processor. GCC-DLX was actually designed for a slightly different DLX simulator called FAST (which is optimized for speed and not as easy to modify as DLXsim).

GCC-DLX provides a powerful feature that allows assembly code to be inserted directly into C programs and directly access the contents of variables from registers. GCC-DLX also provides a linker that allows multiple DLX object modules to be linked into a single executable. GCC-DLX is also portable to many platforms.
7.9 Integrating FP, DLXsim, and GCC-DLX

The FP module is designed specifically to support this thesis and is comprised of about a thousand lines of C code. Integrating the FP module requires that only a few hundred lines be changed in DLXsim, mostly in the core simulation module (a few functions and data structures are replaced with simpler FP calls and callback functions). This is actually quite straightforward.

DLXsim itself has many more problems, however. The "half" directive, required to support "short" data types, is not implemented. GCC-DLX requires that "proc" and "endproc" directives be added. DLXsim does not work properly on a little-endian machine—additional support is required for byte and word swapping. DLXsim also does not properly reinitialize itself when a new executable is loaded, although the safest solution is to simply rerun the simulator in this case.

GCC-DLX also has a number of problems. No scratch registers are provided, which forces extraneous stack accesses to needlessly save and restore register contents. Its notion of where signed values (as opposed to unsigned values) are required does not always correspond to those of DLXsim. The GCC-DLX linker does not always link library files correctly, and several of the supplied library functions are not considered by DLXsim to be valid.

A number of small fixes were made to the versions of DLXsim and GCC-DLX used in support of this thesis. Most were corrected before the interval enhancements were added, although some of the flaws had not been discovered at that time. None of the changes were particularly difficult to make, since both applications were designed
to be easy to modify. However, the number of flaws was almost unacceptably high. This is one of the risks of using free software.

7.10 Enhanced DLX

Implementing the proposed extensions requires about a thousand lines of new or changed code in the DLXsim simulator. Since the extensions can be specified explicitly through inline assembly directives, a lower bound can be placed on the speedup without any changes in GCC-DLX. GCC-DLX could be modified to generate the extensions on its own, in which case further speedup would be realized as extensions such as conditional moves are applied to non-interval operations.

Since traditional DLX does not provide any way to set rounding modes, a write floating point control word (WFPCW) instruction is also present in the enhanced DLX simulator. This instruction causes the entire DLX processor to stall until all floating point operations already issued have been allowed to complete. Enhanced DLX currently stalls the entire processor whenever any floating point operation causes a stall.

Enhanced DLX assumes that all instructions execute in a single cycle except when hazards result in a stall or IIMD is issued and both operands cross zero. For IIMD and IIDD, this implies both register renaming support and four read ports or an extra set of floating point register state in order to avoid read and write port limitations on the register bank. Without these technologies, these operations take several extra cycles to complete.
7.11 Implementing Interval Operations in Enhanced DLX

The basic interval data type and a fairly complete set of interval operations are defined in a single interval support module. The module is implemented as a single header file. All of the operations are implemented as macros in order to give the compiler every opportunity to optimize the code that uses them. Most operations have several variants. These allow specific DLX enhancements to be tested. Thus, one module implements interval operations on the base DLX (traditional DLX with a WFPCW instruction) or on various types of enhanced DLX architectures which implement one or more of the proposed extensions. The code relies heavily on the inline assembly capability of GCC-DLX, both to access the enhanced operations and to set the base DLX rounding mode. The operations are hand optimized for different DLX variations.

The interval data types are “REAL”, which is defined as a double-precision number, and “struct Interval”, which is defined as in Figure 4. The rest of the macros are detailed in Appendix E. Square root is not currently specified, nor are trigonometric functions. Square is currently only optimized for base DLX.
Chapter 8

Performance Evaluation

A small number of interval applications have been implemented and tested using the enhanced DLX simulator and the interval support module. The results of five have been analyzed in detail. The proposed extensions made some interval applications at least 50% faster than equivalent hand optimized interval code on a traditional DLX processor. An interval application created by strictly replacing double precision floating point code with interval operations also ran almost 50% faster on enhanced DLX. Three interval Newton applications, which balance interval and non-interval calculations, averaged around 35% faster.

8.1 Interval Reference Applications

Of the five reference applications, three are instances of using the interval Newton method to obtain the roots of equations. The equations used are shown in Figure 8. The first equation has four roots at 0, 3, 4, and 5. The second has two real roots at 1 and approximately 0.888. The third has no real roots. These particular equations are interesting because in 1983 researchers used them as an example, stating that a general equation solving program could not be used to find all of these roots [36].

\[ f_1(x) = x^4 - 12x^3 + 47x^2 - 60x \]
\[ f_2(x) = x^4 - 12x^3 + 47x^2 - 60x + 24 \]
\[ f_3(x) = x^4 - 12x^3 + 47x^2 - 60x + 24.1 \]

Figure 8: Problems to be solved using the interval Newton method
The other two reference applications are an interval stress test that uses all of the interval operations provided and a math-intensive program that was originally written using double-precision numbers and then converted to use intervals through straight replacements. The former insures that all of the extensions are tested. The latter demonstrates the required overhead of using intervals instead of real numbers.

8.2 Interval Stress Test Results

Two base configurations are used to evaluate the interval stress test. Configuration A is the base DLX. Configuration B adds the conditional move instructions. Each of the proposed new instructions is then individually added to each. The results are as shown in Table 20.

<table>
<thead>
<tr>
<th>Enhancement</th>
<th>Data Size</th>
<th>Configuration A</th>
<th>Configuration B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Size</td>
<td>Cycles</td>
<td>Speedup</td>
</tr>
<tr>
<td>None</td>
<td>320</td>
<td>320184</td>
<td>1</td>
</tr>
<tr>
<td>NEGSD</td>
<td>320</td>
<td>31212</td>
<td>312984</td>
</tr>
<tr>
<td>IIDD</td>
<td>304</td>
<td>2592</td>
<td>310778</td>
</tr>
<tr>
<td>HALFD</td>
<td>312</td>
<td>3220</td>
<td>310281</td>
</tr>
<tr>
<td>IIMD</td>
<td>320</td>
<td>2700</td>
<td>301480</td>
</tr>
<tr>
<td>SRND</td>
<td>320</td>
<td>3200</td>
<td>290584</td>
</tr>
<tr>
<td>IABSD</td>
<td>320</td>
<td>3008</td>
<td>287175</td>
</tr>
<tr>
<td>All</td>
<td>296</td>
<td>1768</td>
<td>212862</td>
</tr>
</tbody>
</table>

Table 20: Interval stress test results

The high speedup from the IABSD is partly attributable to the fact that magnitude and magnitude were used an abnormally large number of times (each was used as many times as addition or subtraction). It may also be partly attributable to the fact that the base implementations of these operations, which do not require setting of rounding modes, were implemented in C, not DLX assembly language.
Further analysis of the results reveals that the IIMD operation saves 18704 cycles. Since the stress test performs 1800 multiplies, the average savings per multiply are 10.39 cycles, meaning for certain inputs, 11 or more cycles are saved. The base interval multiply always takes the same number of cycles to complete and the enhanced DLX version can be issued once every three cycles if there are no data dependencies, so the “best-case” speedup from IIMD is at least \((11 + 3)/3 = 4.6667\) cycles, and is probably higher. Previous studies corroborate this claim [39].

**8.3 Interval Newton Test Results**

Only one configuration is required for the interval Newton tests. These tests do not use magnitude or magnitude, and do not benefit from conditional moves or from IABSD. They also do not benefit from NEGSD since they do not perform interval negation. The results are shown in Table 21.
Table 21: Interval Newton test results

8.4 Interval Replacement Test Results

The interval replacement test also required only one base configuration. In addition to magnitude, magnitude, and negation, this test does not use midpoint. It thus does not benefit from conditional moves, IABSD, NEGS, or HALFD. Its results are shown in Table 22.
Table 22 also shows the performance of the double-precision implementation of
the reference application. The ratio of the performance of interval to real arithmetic on
an enhanced DLX is only 2.91. What is surprising is that the ratio is only 4.29 on the
base DLX. This is not caused by the short floating-point pipelines used in enhanced
DLX. Changing the latency of multiplication and addition to 5 cycles raises execution
time of the reference implementation to 2500 cycles and execution time of the interval
implementations to 6708 and 10217 cycles. The performance ratios actually drop to
2.68 and 4.09, so in this case the longer pipeline favors the interval implementations.

8.5 Testing Summary

The utility of many of the proposed extensions is tied very much to the use of particular
interval operations. Even conditional moves only affect the magnitude and mignitude
interval operations if the other interval extensions are used.

Some of the traditional interval applications, such as interval Newton method
from [10], have been converted to work with the enhanced DLX simulator and interval
libraries. However, they took over 20 times as many cycles to complete. The resultant
speedup from all of the enhancements was just over 1.02. This indicates that these
applications have considerable overhead not caused by the interval operations, which is
easily verified by simple inspection of the source code.
A few minor additions to current processor designs can improve the performance of interval arithmetic significantly. These enhancements also bring the performance of interval arithmetic to well within a factor of five of the performance of double precision implementations. This is an important accomplishment, since it has been speculated that this is the threshold for widespread acceptance of intervals.

9.1 Faster Interval Calculations

After extrapolating slightly on current technology trends, it has been demonstrated that improving the handling of rounding modes and adding a few operations that help set up interval operations allow even hand optimized interval applications to run 35% to 50% faster. Previous studies estimate the execution time of current fast interval libraries at 10 to 30 times that of equivalent floating point operations. Other studies estimate the overhead at 20 to 100 times, presumably for the more general purpose interval libraries. Using these extensions, intervals can thus be processed 3 to 40 times faster than with current interval libraries on existing hardware.

For even the hand-optimized libraries to be as fast as they are, they have to be expanded inline. Without interval extensions, this causes a significant increase in code size. The interval extensions allow interval operations to be encoded as very small blocks of instructions that are simpler and use less register resources. This can help interval-enhanced compilers generate better code. It is important that the code can
expand inline, since much of the potential performance benefits from such compilers will be lost if many additional function calls are needed. Even functions that complete as quickly as short, simple blocks of inline code will cause calling functions to sacrifice some scratch registers and the opportunity to schedule operations to be performed in parallel.

9.2 Replacing Floating Point Numbers with Intervals

The proposed extensions reduce the total execution time of interval applications to within a factor of three of their floating point counterparts. Analyzing the resulting code by hand reveals that this is close to the expected relative throughput of individual interval operations, so this ratio should hold for applications of different sizes.

Even without the interval extensions, careful use of hand optimized libraries has been shown to bring the overhead for intervals down to under 4.5 times that of floating point numbers. Although this requires inline expansion and causes code size to grow significantly, it is an important milestone in interval performance. Experiments have shown that the ratio is not drastically higher with longer floating point pipelines, and is even lower in some cases. Therefore, it can be reasonably expected that interval libraries for existing processors can be created now such that the execution time of the resulting interval applications is 4 to 5 times that of floating point. Similar performance can be expected for interval-enhanced compilers, since they can do at least as well by simply copying the hand optimized libraries, and may be able to do more optimizations afterward.
9.3 Dedicated versus General Purpose Enhancements

The original design goal for the interval enhancements was a set of general purpose extensions that make intervals faster but do not have much impact on processor design and are useful to other applications. Over time, the interval enhancements have evolved into their current state. Only the conditional move instructions remained faithful to the original design premise, and these are widely available anyway. They also are not used very much in interval operations if the other extensions are provided.

The rounding mode improvements also meet these criteria, but implementing them is likely to be done in the larger context of putting all of the processor state into the pipeline and removing the rest of the processor state dependencies. Suddenly, the design impact is not so trivial. Another issue with the rounding mode improvements is their impact on multiple-issue processors, especially when speculative and out-of-order execution is considered. By requiring many additional state transitions, this simple solution could quickly become quite complex.

An alternative is to abandon the idea of making the extensions available for general use and simply add a full set of interval operations while still leaving the basic processor architecture unchanged. These could handle special cases without causing system traps, and would not depend as much on system-wide services like register renaming. They could also send operands to multiple execution units in parallel. Since they can execute atomically, intermediate state would not need to be held in registers. Interval multiply and divide could latch their intermediate state and thus need not destroy the contents of the source registers.
9.4 Opportunities for Future Research

Clearly, an area that deserves further study is the where this thesis converges with some of the previous research into dedicated processors and functional units. Adding interval instructions such as add and multiply can avoid many of the limitations of dynamic rounding modes and of instructions that merely set up state for future instructions. The lesson to be learned from this thesis is to keep the impact on processors to a minimum. The existing hardware can typically process intervals with the required performance as long as steps are taken to keep it busy.

Another area that deserves study is the effect of multiple-issue, superscalar, and out-of-order execution on these extensions. Dynamic scheduling should be considered as well. Currently, the enhanced DLXsim only supports simple forwarding.

Square, square root, and other operations such as trigonometric functions ought to be considered in light of this new approach to intervals. Speeding up these interval operations has not received much attention in the past.

Finally, the FP module deserves to be supported and made public. It is a very powerful tool that allows the impact of new pipeline designs to be evaluated in mere seconds.
Bibliography


Appendix A

C Implementations of Interval Operations

A.1 Interval Addition in C

```c
void IAdd(struct Interval *r, struct Interval *a, struct Interval *b) {
    Set_Rounding_Direction(DOWN);
    r->infimum = a->infimum + b->infimum;
    Set_Rounding_Direction(UP);
    r->supremum = a->supremum + b->supremum;
    Set_Rounding_Direction(DEFAULT);
}
```

A.2 Interval Subtraction in C

```c
void ISub(struct Interval *r, struct Interval *a, struct Interval *b) {
    Set_Rounding_Direction(DOWN);
    r->infimum = a->infimum + b->supremum;
    Set_Rounding_Direction(UP);
    r->supremum = a->supremum + b->infimum;
    Set_Rounding_Direction(DEFAULT);
}
```

A.3 Interval Negation in C

```c
void INegate(struct Interval *r, struct Interval *a) {
    Set_Rounding_Direction(DOWN);
    r->infimum = -(a->supremum);
    Set_Rounding_Direction(UP);
    r->supremum = -(a->infimum);
    Set_Rounding_Direction(DEFAULT);
}
```

A.4 Interval Multiplication in C

```c
void IMul(struct Interval *r, struct Interval *a, struct Interval *b) {
    Set_Rounding_Direction(DOWN);
    if (a->infimum > 0) {
        if (b->infimum > 0) {
            r->infimum = a->infimum * b->infimum;
        }
    }
}
```
Set_Rounding_Direction(UP);
   r->supremum = a->supremum * b->supremum;
 } else if (b->supremum < 0) {
   r->infimum = a->supremum * b->infimum;
   Set_Rounding_Direction(UP);
   r->supremum = a->infimum * b->supremum;
 } else {
   r->infimum = a->supremum * b->infimum;
   Set_Rounding_Direction(UP);
   r->supremum = a->supremum * b->supremum;
 }

else if (a->supremum < 0) {
   if (b->infimum > 0) {
     r->infimum = a->infimum * b->supremum;
     Set_Rounding_Direction(UP);
     r->supremum = a->supremum * b->infimum;
   } else if (b->supremum < 0) {
     r->infimum = a->supremum * b->supremum;
     Set_Rounding_Direction(UP);
     r->supremum = a->infimum * b->infimum;
   } else {
     r->infimum = a->supremum * b->supremum;
     Set_Rounding_Direction(UP);
     r->supremum = a->infimum * b->infimum;
   }

else if (b->infimum < 0) {
   if (b->supremum < 0) {
     r->infimum = a->infimum * b->infimum;
     Set_Rounding_Direction(UP);
     r->supremum = a->infimum * b->infimum;
   } else if (b->infimum < 0) {
     r->infimum = a->supremum * b->supremum;
     Set_Rounding_Direction(UP);
     r->supremum = a->infimum * b->infimum;
   } else {
     double temp1
     double temp2;
     temp1 = a->infimum * b->supremum;
     temp2 = a->supremum * b->infimum;
     r->infimum = (temp1 < temp2) ? temp1 : temp2;
     Set_Rounding_Direction(UP);
     temp1 = a->infimum * b->infimum;
     temp2 = a->supremum * b->supremum;
     r->supremum = (temp1 > temp2) ? temp1 : temp2;
   }
   Set_Rounding_Direction(DEFAULT);
A.5 Interval Division in C

```c
void IDiv(struct Interval *r, struct Interval *a, struct Interval *b)
{
    Set_Rounding_Direction(DOWN);
    if (b->infimum > 0) {
        if (a->infimum >= 0) {
            r->infimum = a->infimum / b->supremum;
            Set_Rounding_Direction(UP);
            r->supremum = a->supremum / b->infimum;
        } else if (a->supremum <= 0) {
            r->infimum = a->infimum / b->infimum;
            Set_Rounding_Direction(UP);
            r->supremum = a->supremum / b->supremum;
        } else {
            r->infimum = a->infimum / b->infimum;
            Set_Rounding_Direction(UP);
            r->supremum = a->infimum / b->supremum;
        }
    } else if (b->infimum < 0) {
        if (a->infimum >= 0) {
            r->infimum = a->supremum / b->supremum;
            Set_Rounding_Direction(UP);
            r->supremum = a->infimum / b->infimum;
        } else if (a->supremum <= 0) {
            r->infimum = a->supremum / b->infimum;
            Set_Rounding_Direction(UP);
            r->supremum = a->infimum / b->supremum;
        } else {
            r->infimum = a->supremum / b->supremum;
            Set_Rounding_Direction(UP);
            r->supremum = a->infimum / b->supremum;
        }
    } else {
        r->infimum = MINUS_INFINITY;
        r->supremum = PLUS_INFINITY;
    }
    Set_Rounding_Direction(DEFAULT);
}
```

A.6 Interval Width in C

```c
void IWidth(REAL *r, struct Interval *a)
{
    Set_Rounding_Direction(UP);
    r = a->supremum - a->infimum;
    Set_Rounding_Direction(DEFAULT);
}
```
A.7 Interval Midpoint in C

void IMidpoint(REAL *r, struct Interval *a)
{
    Set_Rounding_Direction(UP);
    r = (a->supremum - a->infimum) / 2.0;
    Set_Rounding_Direction(DOWN);
    r = a->infimum + r
    Set_Rounding_Direction(DEFAULT);
}

A.8 Interval Magnitude in C

void IMagnitude(REAL *r, struct Interval *a)
{
    REAL tempi, temp2;
    tempi = (a->infimum < 0.0) ? -(a->infimum) : a->infimum;
    temp2 = (a->supremum < 0.0) ? -(a->supremum) : a->supremum;
    r = (tempi > temp2) ? tempi : temp2;
}

A.9 Interval Mignitude in C

void IMignitude(REAL *r, struct Interval *a)
{
    if ((a->infimum > 0.0) != (a->supremum > 0.0)) {
        r = 0.0;
    } else {
        REAL temp1, temp2;
        temp1 = (a->infimum < 0.0) ? -(a->infimum) : a->infimum;
        temp2 = (a->supremum < 0.0) ? -(a->supremum) : a->supremum;
        r = (temp2 < temp1) ? temp2 : temp1;
    }
}
Appendix B

DLX Implementations of Interval Operations

B.1 Interval Addition in DLX

\[
\begin{align*}
\text{iadd:} & \quad \text{wfpcw} \quad \text{roundDown} \\
& \quad \text{add} \quad f0,f4,f8 \\
& \quad \text{wfpcw} \quad \text{roundUp} \\
& \quad \text{add} \quad f2,f6,f10 \\
& \quad \text{wfpcw} \quad \text{roundNear} \\
& \quad \text{jr} \quad r31 \\
& \quad \text{nop}
\end{align*}
\]

B.2 Interval Subtraction in DLX

\[
\begin{align*}
\text{isub:} & \quad \text{wfpcw} \quad \text{roundDown} \\
& \quad \text{subd} \quad f0,f4,f10 \\
& \quad \text{wfpcw} \quad \text{roundUp} \\
& \quad \text{subd} \quad f2,f6,f8 \\
& \quad \text{wfpcw} \quad \text{roundNear} \\
& \quad \text{jr} \quad r31 \\
& \quad \text{nop}
\end{align*}
\]

B.3 Interval Negation in DLX

\[
\begin{align*}
\text{ineg:} & \quad \text{movi2fp} \quad f8,r0 \\
& \quad \text{cvtf2d} \quad f8,f8 \\
& \quad \text{wfpcw} \quad \text{roundDown} \\
& \quad \text{subd} \quad f0,f8,f6 \\
& \quad \text{wfpcw} \quad \text{roundUp} \\
& \quad \text{subd} \quad f2,f8,f4 \\
& \quad \text{wfpcw} \quad \text{roundNear} \\
& \quad \text{jr} \quad r31 \\
& \quad \text{nop}
\end{align*}
\]

B.4 Interval Multiplication in DLX

\[
\begin{align*}
\text{imul:} & \quad \text{movi2fp} \quad f12,r0 \\
& \quad \text{cvtf2d} \quad f12,f12 \\
& \quad \text{ged} \quad f4,f12 \\
& \quad \text{bfpt} \quad \text{mul\_ap} \\
& \quad \text{led} \quad f6,f12 \\
& \quad \text{bfpt} \quad \text{mul\_an}
\end{align*}
\]

\[
\text{mul\_az:}
\]
ged    f8,f12
bfpt   mul_az_bp
led    f10,f12
bfpt   mul_az_bn

mul_az_bz:
   wfpw   roundDown
   multd f0,f4,f10
   multd f2,f6,f8
   led    f0,f2
   bfpt   mul_az_bz_i1
   sd     -8(r29),f0
   sd     -8(r29),f2

mul_azb_z_i1:
   wfpw   roundDownUp
   multd f0,f4,f8
   multd f2,f6,f10
   ged    f2,f0
   bfpt   mul_azb_z_i2
   nop
   movd   f2,f0

mul_azb_z_i2:
   j       mul_end
   ld     f0,-8(r29)

mul_az_bn:
   ; ROUNDDOWN() in delay slot
   multd f0,f6,f8
   wfpw   roundDownUp
   j       mul_end
   multd f2,f4,f8

mul_az_bp:
   wfpw   roundDownDown
   multd f0,f4,f10
   wfpw   roundDownUp
   j       mul_end
   multd f2,f6,f10

mul_an:
   ; ged    f8,f12 in delay slot
   bfpt   mul_an_bp
   led    f10,f12
   bfpt   mul_an_bn

mul_an_bz:
   wfpw   roundDownDown
   multd f0,f4,f10
   wfpw   roundDownUp
   j       mul_end
   multd f2,f4,f8

mul_an_bn:
   ; ROUNDDOWN() in delay slot
   multd f0,f6,f10
   wfpw   roundDownUp
   j       mul_end
B.5 Interval Division in DLX

idiv: movi2fp f12,r0
cvtf2d f12,f12
lhi r1,(neginf>>16)&0xffffffff
addui r1,r1,neginf&0xffffffff
ld f14,(r1)
lhi r1,(posinf>>16)&0xffffffff
addui r1,r1,posinf&0xffffffff
ld f16,(r1)
ged f4,f12
bfpt div_ap
led f6,f12
bfpt div_an

div_az:
gtd f8,f12
bfpt  div_az_bp
ld  f10,f12
bfpt  div_az_bn
wfpcw  roundDownDown

div_az_bp:
  movd  f0,f14
  j  div_end
  movd  f2,f16

div_az_bn:
  ; ROUNDDOWN() in delay slot
  divd  f0,f6,f10
  wfpcw  roundDownUp
  j  div_end
  divd  f2,f4,f10

div_az_bp:
  wfpcw  roundDownDown
  divd  f0,f4,f8
  wfpcw  roundDownUp
  j  div_end
  divd  f2,f6,f8

div_an:
  ; gtd  f8,f12 in delay slot
  bfpt  div_an_bp
  ltd  f10,f12
  bfpt  div_an_bn

div_an_bz:
  wfpcw  roundDownDown
  eqd  f8,f10
  bfpt  div_an_bzn
  eqd  f10,f12
  bfpt  div_an_bzn
  eqd  f8,f12
  bfpt  div_an_bzp

div_an_bzn:
  movd  f0,f14
  j  div_end
  movd  f2,f16

div_an_bzp:
  ; movd  f0,f14 in delay slot
  wfpcw  roundDownUp
  j  div_end
  divd  f2,f6,f10

div_an_bzn:
  ; ROUNDDOWN() done already
  divd  f0,f6,f8
  j  div_end
  movd  f2,f16

div_an_bn:
  ; ROUNDDOWN() in delay slot
  divd  f0,f6,f8
  wfpcw  roundDownUp

72
j    div_end
divd f2,f4,f10

div_an_bp:
   wfp cw roundDownDown
   divd f0,f4,f8
   wfp cw roundDownUp
j    div_end
   divd f2,f6,f10

div_ap:
   gtd  f8,f12
   bftp div_ap_bp
   ltd  f10,f12
   bftp div_ap_bn

div_ap_bz:
   wfp cw roundDownDown
   eqd  f8,f10
   bftp div_ap_bznp
   eqd  f8,f12
   bftp div_ap_bzp
   eqd  f10,f12
   bftp div_ap_bzn

div_ap_bznp:
   movd f0,f14
   j    div_end
   movd f2,f16

div_ap_bzn:
   ; movd f0,f14 in delay slot
   wfp cw roundDownUp
   j    div_end
   divd f2,f4,f8

div_ap_bzp:
   divd f0,f4,f10
   j    div_end
   movd f2,f16

div_ap_bn:
   ; ROUNDDOWN() done already
   divd f0,f6,f10
   wfp cw roundDownUp
   j    div_end
   divd f2,f4,f8

div_ap_bp:
   wfp cw roundDownDown
   divd f0,f4,f10
   wfp cw roundDownUp
   divd f2,f6,f8

div_end:
   wfp cw roundDownNear
   jr    r31
   nop

neginf: .word 0xffff0000, 0x00000000
posinf: .word 0x7ff00000, 0x00000000
B.6 Interval Width in DLX

\[
\text{iwid: } \quad \text{wfpcw roundUp} \\
\text{subd } \quad f0, f6, f4 \\
\text{wfpcw roundNear} \\
\text{jr } \quad r31 \\
\text{nop}
\]

B.7 Interval Midpoint in DLX

\[
\text{imidpt: } \quad \text{lhi } \quad r1, (\text{two}>>16) & 0xffff \\
\text{addui } \quad r1, r1, \text{two} & 0xffff \\
\text{ld } \quad f2, (r1) \\
\text{wfpcw roundUp} \\
\text{subd } \quad f0, f6, f4 \\
\text{divd } \quad f0, f0, f2 \\
\text{wfpcw roundDown} \\
\text{add } \quad f0, f0, f4 \\
\text{wfpcw roundNear} \\
\text{jr } \quad r31 \\
\text{nop}
\]

\text{two: } \quad \text{.double 2.0}

B.8 Interval Magnitude in DLX

\[
\text{imag: } \quad \text{movi2fp } f0, r0 \\
\text{cvtf2d } \quad f0, f0 \\
\text{ged } \quad f4, f0 \\
\text{bfpt } \quad \text{infOK} \\
\text{led } \quad f6, f0 \\
\text{subd } \quad f4, f0, f4 \\
\text{infOK: } \quad \text{bfpt } \quad \text{supOK} \\
\text{gtd } \quad f4, f6 \\
\text{subd } \quad f6, f0, f6 \\
\text{supOK: } \quad \text{bfpt } \quad \text{end} \\
\text{movd } \quad f0, f4 \\
\text{movd } \quad f0, f6 \\
\text{end: } \quad \text{jr } \quad r31 \\
\text{nop}
\]

B.9 Interval Mignitude in DLX

\[
\text{imig: } \quad \text{movi2fp } f0, r0 \\
\text{cvtf2d } \quad f0, f2 \\
\text{ged } \quad f4, f0 \\
\text{bfpt } \quad \text{infOK} \\
\text{led } \quad f6, f0 \\
\text{bfpf } \quad \text{end}
\]
subd    f4, f0, f4
subd    f6, f0, f6
infOK:  ltd    f6, f4
         bfp t   end
         movd   f0, f6
         movd   f0, f4
end:    jr     r31
         nop
### Appendix C

**Definitions of the Proposed Extensions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
</table>
| State[round] | if (a floating point operation is being performed) {  
Round as indicated  
State[round] ← NextState[State[round]]  
} |
| SRND mode | State[round] ← mode |
| MOVDT Fa, Fb | if (Flags[FP] = 1) {  
Regs[Fa] ← Regs[Fb]  
Regs[Fa+1] ← Regs[Fb+1]  
} |
| MOVDF Fa, Fb | if (Flags[FP] = 1) {  
Regs[Fa] ← Regs[Fb]  
Regs[Fa+1] ← Regs[Fb+1]  
} |
| IABSD Fa, Fb | Flags[FP] ← (Regs[Fa]₀ ≠ Regs[Fb]₀)  
Regs[Fa]₀ ← 0  
Regs[Fb]₀ ← 0 |
| NEGSD Fa, Fb | temp ← Regs[Fb]  
Regs[F(b)] ← Regs[Fa]  
Regs[F(a)] ← temp  
Regs[F(a)]₀ ← Regs[Fa]₀  
Regs[F(b)]₀ ← Regs[Fb]₀ |
| IIMD Fa, Fb | State[round] ← DownUp  
if (Regs[Fa]₀ = 0) {  
if (Regs[Fb]₀ = 0) {  
/* do nothing */  
} else if (Regs[Fb+2]₀ = 1) {  
temp ← Regs[Fa+2]  
Regs[Fa+2] ← Regs[Fa]  
Regs[Fa] ← temp  
temp ← Regs[Fa+3]  
Regs[Fa+3] ← Regs[Fa+1]  
Regs[Fa+1] ← temp  
} else {  
Regs[Fa] ← Regs[Fa+2]  
Regs[Fa+1] ← Regs[Fa+3]  
}  
} else if (Regs[Fa+2]₀ = 1) {  
if (Regs[Fb]₀ = 0) {  
|  

---

76
IIMDFa, Fb
(continued)

<table>
<thead>
<tr>
<th></th>
<th>temp ← Regs[Fb+2]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Regs[Fb+2] ← Regs[Fb]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fb+3]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+3] ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← temp</td>
</tr>
<tr>
<td></td>
<td>else if (Regs[Fb+2]₀ = 1) {</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fa+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+2] ← Regs[Fa]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fa+3]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+3] ← Regs[Fa+1]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+1] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← Regs[Fb]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fb+3]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+3] ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← temp</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+2] ← Regs[Fa];</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+3] ← Regs[Fa+1];</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← Regs[Fb]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fb+3]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+3] ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← temp</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>if (Regs[Fb]₀ = 0) {</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb] ← Regs[Fb+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+1] ← Regs[Fb+3]</td>
</tr>
<tr>
<td></td>
<td>} else if (Regs[Fb+2]₀ = 1) {</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fa+2]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+2] ← Regs[Fa]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa] ← temp</td>
</tr>
<tr>
<td></td>
<td>temp ← Regs[Fa+3]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+3] ← Regs[Fa+1]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fa+1] ← temp</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+2] ← Regs[Fb]</td>
</tr>
<tr>
<td></td>
<td>Regs[Fb+3] ← Regs[Fb+1]</td>
</tr>
<tr>
<td></td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>R31 ← PC</td>
</tr>
<tr>
<td>state</td>
<td>code</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| DownUp| ```
if (Regs[Fb] == 0) {
  if (Regs[Fa] == 0) {
    temp ← Regs[Fb+2]
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb] ← temp
    temp ← Regs[Fb+3]
    Regs[Fb+3] ← Regs[Fb+2]
    Regs[Fb+2] ← temp
  } else if (Regs[Fa+2] == 1) {
    /* do nothing */
  } else {
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb+3] ← Regs[Fb+1]
  }
} else if (Regs[Fb+2] == 1) {
  if (Regs[Fa] == 0) {
    temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← temp
    temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
    Regs[Fa+1] ← temp
    temp ← Regs[Fb+2]
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb] ← temp
    temp ← Regs[Fb+3]
    Regs[Fb+3] ← Regs[Fb+2]
    Regs[Fb+2] ← temp
  } else if (Regs[Fa+2] == 1) {
    temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← temp
    temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
    Regs[Fa+1] ← temp
  } else {
    temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← temp
    temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
  }
``` | State[round] ← DownUp
If (Regs[Fb] == 0) {
  If (Regs[Fa] == 0) {
    Temp ← Regs[Fb+2]
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb] ← Temp
    Temp ← Regs[Fb+3]
    Regs[Fb+3] ← Regs[Fb+2]
    Regs[Fb+2] ← Temp
  } Else if (Regs[Fa+2] == 1) {
    /* Do nothing */
  } Else {
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb+3] ← Regs[Fb+1]
  }
} Else if (Regs[Fb+2] == 1) {
  If (Regs[Fa] == 0) {
    Temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← Temp
    Temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
    Regs[Fa+1] ← Temp
    Temp ← Regs[Fb+2]
    Regs[Fb+2] ← Regs[Fb]
    Regs[Fb] ← Temp
    Temp ← Regs[Fb+3]
    Regs[Fb+3] ← Regs[Fb+2]
    Regs[Fb+2] ← Temp
  } Else if (Regs[Fa+2] == 1) {
    Temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← Temp
    Temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
    Regs[Fa+1] ← Temp
  } Else {
    Temp ← Regs[Fa+2]
    Regs[Fa+2] ← Regs[Fa]
    Regs[Fa] ← Temp
    Temp ← Regs[Fa+3]
    Regs[Fa+3] ← Regs[Fa+1]
  }
|
### IIDD Fa, Fb

(continued)

| Regs[Fa+1] ← temp  
| Regs[Fb] ← Regs[Fb+2]  
| Regs[Fb+1] ← Regs[Fb+3] |

}  
} else {  
    Regs[Fa] ← a_negative_value  
    Regs[Fa+2] ← a_positive_value  
    Regs[Fb] ← (0)  
    Regs[Fb+1] ← (0)  
    Regs[Fb+2] ← (0)  
    Regs[Fb+3] ← (0)  
}

### HALFD Fa, Fb

if (Regs[Fb]_{exponent} = (E_{max} + 1)) {  
    Regs[Fa] ← Regs[Fb]  
    Regs[Fa+1] ← Regs[Fb+1]  
} else if (Regs[Fb]_{exponent} = E_{min}) {  
    Regs[Fa]_{exponent} ← (E_{min} - 1)  
    Regs[Fa]_{significand}0 ← 1  
    Regs[Fa]_{significand}1...N ← Regs[Fb]_{significand}0...N-1  
    Regs[Fa+1]_{0} ← Regs[Fb]_{significand}N  
    Regs[Fa+1]_{1...32} ← Regs[Fb]_{0...31}  
} else if (Regs[Fb]_{exponent} = (E_{min} - 1)) {  
    Regs[Fa]_{exponent} ← (E_{min} - 1)  
    Regs[Fa]_{significand}0 ← Regs[F(b)]_{significand}0  
    Regs[Fa]_{significand}1...N ← Regs[Fb]_{significand}0...N-1  
    Regs[Fa+1]_{0} ← Regs[Fb]_{significand}N  
    Regs[Fa+1]_{1...32} ← Regs[Fb]_{0...31}  
} else {  
    Regs[Fa]_{exponent} ← (Regs[Fb]_{exponent} - 1)  
    Regs[Fa]_{significand} ← Regs[Fb]_{significand}  
    Regs[Fa+1] ← Regs[Fb+1]  
}
Appendix D

Pipeline Description File

The following pipeline description file describes the floating point pipeline used in the enhanced DLX architecture.

```
UNITS
FPADDX1 1
FPADDX2 1
FPADDX3 1
FDPDIV 1
FPMULX1 1
FPMULX2 1
FPMULX3 1
DIV 1
IMUL 1
ENDUNITS

USAGE
INSTRUCTION DADD
FPADDX1 1
ENDSTAGE
FPADDX2 1
ENDSTAGE
FPADDX3 1
ENDINSTRUCTION

INSTRUCTION FADD
FPADDX1 1
ENDSTAGE
FPADDX2 1
ENDSTAGE
FPADDX3 1
ENDINSTRUCTION

INSTRUCTION DSUB
FPADDX1 1
ENDSTAGE
FPADDX2 1
ENDSTAGE
FPADDX3 1
ENDSTAGE
ENDINSTRUCTION

INSTRUCTION FSUB
FPADDX1 1
ENDSTAGE
FPADDX2 1
ENDSTAGE
FPADDX3 1
ENDSTAGE

INSTRUCTION FADD
FPADDX1 1
ENDSTAGE
FPADDX2 1
ENDSTAGE
FPADDX3 1
ENDINSTRUCTION

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

INSTRUCTION UIMUL
IMUL 1
ENDSTAGE

INSTRUCTION UIMUL
IMUL 1
ENDSTAGE

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

INSTRUCTION IEV
IMUL 1
ENDSTAGE

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

INSTRUCTION IDIV
IMUL 1
ENDSTAGE

ENDUSAGE
```
## Appendix E

### Defined Interval Types and Macros

<table>
<thead>
<tr>
<th>Data types</th>
<th>REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>struct Interval</td>
</tr>
<tr>
<td>Initializers for static interval data</td>
<td>INTERVAL_INITZERO()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_INIT()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_INITREAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_INITINVALID()</td>
</tr>
<tr>
<td>To set intervals directly</td>
<td>INTERVAL_SETZERO()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SET()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SETREAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SETINVALID()</td>
</tr>
<tr>
<td>Scalar decomposition</td>
<td>INTERVAL_INF()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SUP()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_MAGNITUDE()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_MAGNITUDE()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_MIDPOINT()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_WIDTH()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SQUARE()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_NEGATE()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_ABS()</td>
</tr>
<tr>
<td>Binary operations</td>
<td>INTERVAL_ADD()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SUBTRACT()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_MULTIPLY()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DIVIDE()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_INTERSECTION()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_CONVEXHULL()</td>
</tr>
<tr>
<td>Unary tests</td>
<td>INTERVAL_OK()</td>
</tr>
<tr>
<td>Test for scalar inclusion</td>
<td>INTERVAL_POSSIBLYZERO()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYZERO()</td>
</tr>
<tr>
<td>Set comparisons</td>
<td>INTERVAL_CONTAINS()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_PROPERSUBSET()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_PROPERSUPERSET()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SUBSET()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SUPERSET()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DISJOINT()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_SETEQUALS()</td>
</tr>
<tr>
<td>Possibly true comparisons</td>
<td>INTERVAL_POSSIBLYGREATERTHANOREQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_POSSIBLYGREATER_THAN()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_POSSIBLYEQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_POSSIBLYNOTEQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_POSSIBLYLESS_THAN()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_POSSIBLYLESSTHANOREQUAL()</td>
</tr>
<tr>
<td>Definitely true comparisons</td>
<td>INTERVAL_DEFINITELYGREATERTHANOREQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYGREATER_THAN()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYEQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYNOTEQUAL()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYLESS_THAN()</td>
</tr>
<tr>
<td></td>
<td>INTERVAL_DEFINITELYLESSTHANOREQUAL()</td>
</tr>
</tbody>
</table>
Vita

Gerald Shawn Williams, the son of James and Julia Williams, was born in Somerville, New Jersey on July 5, 1966. In 1988, he received the degree of Bachelor of Science in Computer Science at Stevens Institute of Technology in Hoboken, New Jersey. During his undergraduate studies, he interned at Ciba-Geigy Pharmaceuticals in Summit, New Jersey, Stone and Webster Engineering Management Consultants in New York City, and Canadian Imperial Bank of Commerce in New York City. He has since worked as a design engineer for AT&T Bell Laboratories, now part of Lucent Technologies. He has filed for four patents related to software technology. He is currently pursuing a Masters Degree in Computer Engineering at Lehigh University in Bethlehem, Pennsylvania. He resides in Macungie, Pennsylvania with his wife Cynthia.
END
OF
TITLE