Elementary Function Evaluation
Using
New Hardware Instructions

Elementary Function Evaluation
Using
New Hardware Instructions

By
Anuroop Sharma M.Tech
IBM Center for Advanced Studies Fellow

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE(2010) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: Elementary function evaluation using New Hardware Instruction

AUTHOR: Anuroop Sharma M.Tech (Indian Institute of Technology, Delhi)

SUPERVISOR: Dr. Christopher Anand

NUMBER OF PAGES: xiv, 89

LEGAL DISCLAIMER: This is an academic research report. I, my supervisor, defence committee, and university, make no claim as to the fitness for any purpose, and accept no direct or indirect liability for the use of algorithms, findings, or recommendations in this thesis.

Abstract

In this thesis, we present novel fast and accurate hardware/ software implementations of the elementary math functions based on range reduction, *e.g.* Bemer's multiplicative reduction and Gal's accurate table methods. The software implementations are branch free, because the new instructions we are proposing internalize the control flow associated with handling exceptional cases.

These methods provide an alternative to common iterative methods of computing reciprocal, square root and reciprocal square root. These methods could be applied to any rational-power operation. These methods require either the precision available through fused multiply-accumulate instructions or extra working precision in registers. We also extend the range reduction methods to include trigonometric and inverse trigonometric functions.

The new hardware instructions enable exception handling at no additional cost in execution time, and scale linearly with increasing superscalar and SIMD widths. Based on reduced instruction, constant counts, and reduced register pressure we would recommend that optimizing compilers always in-line such functions, further improving performance by eliminating function-call overhead.

On the Cell/B.E. SPU, we found an overall 234% increase in throughput for the new table-based methods, with increased accuracy.

The research reported in the thesis has resulted in a patent application [AES10], filed jointly with IBM.

Acknowledgments

I would like to thank my supervisor Dr. Christopher Anand for his wonderful support and guidance throughout my degree.

I would like to thank the IBM Toronto Lab Center for Advanced Studies, and Robert Enenkel, for their support of my research work thus far.

I would like to thank Dr. Wolfram Kahl and Mira Anand for very useful comments.

I would also like to thank my fellow students in the Coconut project.

I would also like to thank my parents, family and friends for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Elementary function libraries, like IBM's Mathematical Acceleration Subsystem (MASS), are often called from performance-critical code sections, and hence contribute greatly to the efficiency of numerical applications. Not surprisingly, such functions are heavily optimized both by the software developer and the compiler, and processor manufacturers provide detailed performance results which potential users can use to estimate the performance of new processors on existing numerical workloads.

Changes in processor design require such libraries to be re-tuned; for example,

- hardware pipelining and superscalar dispatch will favour implementations which use more instructions, and have longer total latency, but which distribute computation across different execution units and present the compiler with more opportunities for parallel execution.
- Single-Instruction-Multiple-Data (SIMD) parallelism, and large penalties for data-dependent unpredictable branches favour implementations which handle all cases in a branchless loop body over implementations with a fast path for common cases and slower paths for uncommon, *e.g.*, exceptional, cases.

1.2 Novelty

In this thesis, we address these issues by defining new algorithms and new hardware instructions to simplify the implementation of such algorithms. In

[AS10, AS09], we introduced new accurate table methods for calculating logarithms and exponentials, including the special versions $\log(x+1)$ and $\exp(x)-1$ which are needed to get accurate values for small inputs (respectively outputs), including subnormal values. In this thesis, we introduce a related approach for calculating fixed powers (roots and reciprocals), and we show that all of these functions can be accelerated by introducing novel hardware instructions. In addition, we also introduce an accurate table range reduction approach for other elementary functions, including trigonometric and inverse trigonometric functions. Hardware-based seeds for iterative root and reciprocal computations have been supported on common architectures for some time. As a result, iterative methods are preferred for these computations, although other table-based methods also exist.

By reducing to seven the number of tables needed for all standard math functions, we have provided an incentive to accelerate such computations widely in hardware.

In this thesis, we show that accelerating such functions by providing hardware-based tables has a second advantage: all exceptions can be handled at minimal computational cost in hardware, thus eliminating all branches (and predicated execution) in these functions. This is especially important for SIMD parallelism.

Many of the ideas in this thesis are covered by patent application "Hardware Instructions to Accelerate Table-Driven Mathematical Function Evaluation", Christopher K. Anand, Robert Enenkel, and Anuroop Sharma, US Patent Application 12/788570.

1.3 Impact

The resulting instruction counts dramatically reduce the barriers to in-lining these math functions, which will further improve performance. We also expect the new instructions to result in reduced power consumption for applications calling these functions. When compared to current software implementations on the Cell/B.E. SPU, these new hardware-assisted versions would result in a **tripling of throughput** for vector libraries (functions which map elementary functions over arrays of inputs). But application codes which are difficult to re-factor to call such efficient implementations would benefit even more, because the hardware-assisted implementations use a third of the number of instructions and have a **memory footprint a hundred times smaller**, completely eliminating the penalties associated with in-lining the instructions.

1.4 Trademarks

IBM is a registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

1.5 Thesis Organisation

The rest of the thesis is organized as follows. Chapter 2 gives an overview of floating point arithmetic and the relevant previous work in the area of elementary function evaluation. Chapter 3 uses the square root function as a case study to demonstrate the relative advantages of the proposed thesis over the conventionally used Newton Raphson methods. Chapter 4 gives an overview of the proposed hardware instructions and discusses the issues related to the hardware implementations. Chapter 5 gives an overview of the Coconut [AK09] project and the Haskell programming language which is used to simulate the proposed hardware instructions and the software implementations of the elementary functions. Chapter 6 through Chapter 13 discuss the implementation of the proposed hardware instructions and the elementary functions, using Coconut. Chapter 14 reports the efficiency and the accuracy results for the elementary functions; with the conclusions of the thesis in the last chapter.

Chapter 2

Background and Previous Work

Accurate evaluation of mathematical functions is very important for the stability of many numerical algorithms. The errors in elementary functions become more significant as they tend to accumulate as algorithms become more complicated. Many efforts have been made to improve the accuracy of elementary functions. To understand the error analysis, given in this thesis, an understanding of the IEEE floating point format and the ulp (unit in the last place) is a requisite. In this chapter, after explaining this floating point representation, we will discuss some previous work in the field of elementary function evaluation.

2.1 IEEE Floating Point Numbers and Ulp Errors

In computer hardware, only discrete values of the real numbers, called floating point numbers, can be represented. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation. This representation uses some fixed number of bits to represent the significant and the exponent, respectively. For single-precision floating point values, 23 and 8 bits are respectively used to represent the significant and the exponent which is biased by 127. In double-precision floating point, these values are 52 and 11 bits, where exponent is biased by 1023. In both representations, the first bit is used to store the sign of the value and 2 is used as the base. The numerical value of a floating point number can be calculated using the following formula

$$(-1)^{\text{sign}} \times (1.\text{significant digits}) \times \text{base}^{\text{exponent-bias}}.$$

The unit ulp (unit in the last place) is frequently used to report the error in numerical calculations, since it represents the smallest representable error. Since an ulp is defined to be the difference between two consecutive IEEE floating point values, it is a relative measure whose absolute value depends on the value whose error is being reported. In the rest of this thesis, most of the error analysis is done using ulp as an unit. For more details, see [Mul05].

2.2 Multiplicative Range Reduction

Multiplicative range reduction goes back to Bemer [Bem63], who used it in square-root subroutines to do range reduction before using a polynomial approximation to generate a starting point for the Newton-Raphson iteration. The effect of multiplicative range reduction is then undone by multiplying the result by a second table value. At the time Bemer invented the method, large tables would not have been practical. As read-only memory became cheaper, however, hardware tables made large, low-latency tables practical. For example, [Tak97] proposed a single-coefficient look-up for linear interpolation of powers, which for double precision would require a table with 2^{24} entries. Correct rounding would require extra precision in the interpolation and an extra multiplication used to generate an additional coefficient. In contrast, our proposed method requires smaller table size (*e.g.*, 2^{13}) and no extra precision registers (other than the extra precision used in a fused multiply-add).

2.3 Accurate Tables

Accurate table methods have been used for long time in the evaluation of the elementary math functions [Gal86, GB91]. Gal's accurate tables are devised to provide accurate values of special functions using a lookup table and interpolation. The idea behind Gal's accurate table method is to use table values that are either exactly representable in IEEE floating point representation or use values which are very close to an IEEE floating point number instead of using the tables of equally spaced argument values which results in rounding errors.

2.4 Multiplicative-Reduction Accurate Tables

The combination of multiplicative reduction and accurate table methods, and the recognition that this combination could be used to reduce the number of tables used in the logarithm and exponential families of functions was introduced in [AS10, AS09]. Obviously, the reduction in the number of tables translates into a reduction in the hardware cost and complexity for hardware-assisted implementations, and is an important enabler for the efficient implementation of the ideas presented in this thesis.

2.5 NR Reduction with Hardware Seed

Most modern architectures which support floating-point computation provide estimate instructions for reciprocal and reciprocal square root which can be used both as rough estimates when low accuracy is sufficient (*e.g.*, some stages of graphics production) and as seeds for iterative methods, most commonly Newton-Raphson refinements, also called Heron's method in the case of square-root computations.

Some architectures like IBM POWER5 [Cor05] provide an instruction which calculates the exact square root of the input register. Even on those architectures which provide these machine instructions, iterative methods offer higher efficiency and therefore higher throughput on pipelined machines because the estimates can be pipelined while single instructions like those mentioned, cannot be reasonably pipelined because of their long latency. For example, 84 cycles for the Cell/B.E. PPE's floating point square root instruction (see table A-1 of [Cor08]), which is significantly longer than other instructions like floating point multiply-add with a latency of 10 cycles.

Iterative methods for $\sqrt{x}$ are most efficiently implemented by estimating and refining $1/\sqrt{x}$, and multiplying the result by x. Vector normalization is one common operation which requires $1/\sqrt{x}$ rather than $\sqrt{x}$, so it is very useful to have efficient computations for both the root and its reciprocal without having to use a second (reciprocal or divide) operation.

Unfortunately, correct final rounding often requires an extra iteration, called Tuckerman rounding [ACG+86], or extra-precision registers [Sch95] [KM97]. Even if the correct rounding of $1/\sqrt{x}$ is available, multiplying by x will often produce incorrectly rounded results — 28% of the time, when we tested 20000 random values in Maple. Showing that a particular scheme results in correctly rounded output is an involved process [Rus98]. Although, to a first approximation, these iterations have quadratic convergence and hence the number of iterations can be adjusted according to the required accuracy, the final error is very sensitive to the seed value, and the best value depends on both the power and the number of iterations [KM06].

Chapter 3

Square Root: A Case Study

We use the square root function to demonstrate the relative advantages of the method proposed in this thesis, compared to the most widely used iterative methods. We also provide the equations governing the proposed algorithm and maximum theoretical errors introduced by different steps of the algorithm.

3.1 Iterative Newton Raphson

The Newton Raphson method to calculate the square root of the input is based on the seed or approximate value of its reciprocal square root provided by the hardware instruction. The hardware instruction returns the approximation of the reciprocal square root of the input using piecewise interpolation, which requires table lookups. Heron's refinements are then performed,

$$x_n = x_{n-1} + \frac{x_{n-1}}{2}\left(1 - {x_{n-1}}^2 v\right), \tag{3.1}$$

in software to get 52-bit accurate reciprocal square root of the input.

The accuracy of the estimate is doubled with every repetition of the refinement, so a certain number of refinements, based on the accuracy of initial estimate, are required to produce the 52-bit accurate reciprocal square root. Square root is then calculated by multiplying the input with the reciprocal square root, but it does not guarantee the correct rounding.

For the inputs 0 and ∞, the reciprocal square root estimate instruction returns ∞ and 0 respectively, which then is multiplied with the input, producing *NaN* as the output of the refinement step. In order to handle the exceptional cases, we need to test the input for these outputs and either branch out or use predication or a floating-point select instruction to substitute the right output. All these computations require extra instructions or a conditional branch out, thus decreasing the throughput.

3.2 Multiplicative Reduction Accurate Tables

In this section, we explain our algorithm with new proposed hardware instructions. Let $2^e \cdot f$ be a floating-point input. Decompose $e = 2 \cdot q + r$ such that $r \in \{0, 1\}$, and use this to rewrite the square root as

$$(2^e \cdot f)^{1/2} = 2^q \cdot 2^{r/2} \cdot f^{1/2}. \tag{3.2}$$

Next, we use the proposed new instruction which will produce multiplicative reduction factor $1/\mu$. Then, we multiplicatively reduce the input by

$$c = \frac{1}{\mu} f - 1, \tag{3.3}$$

where $1/\mu = \frac{1}{2^{-e}\nu}$ is produced using an accurate value $1/\nu$, looked up in a table using the concatenation of the first $n-1$ bits of the mantissa and the low-order exponent bit as an index, and $N = 2^n$ is the number of intervals which map into, but not onto $(-1/N, +1/N)$. This is another way of saying that $|c| < 2^{-N}$, because multiplication by a power of 2 is same as the addition of exponent bits. For small subnormal inputs, this multiplicative factor is larger than the largest representable IEEE floating point number. For this reason, we propose a new extended range double representation which has 12 bits for storing the exponent, and 51-bits for storing the mantissa bit. We have calculated the accurate table values in such a way that the implied 52nd bit is always zero. This effectively doubles the range of normal values, hence subnormal inputs and large inputs which produce subnormals as their reciprocal can be treated in same way as other normal inputs. We also propose a new instruction ***fmaX***, which computes the fused multiply add over the three arguments ***fmaX*** a b c $= a{*}b{+}c$, where the first argument is an extended range double. The reason we include the low-order exponent bit is so that we can look up $\sqrt{\mu} = 2^q \cdot \sqrt{2^r \cdot \nu}$ in parallel with $1/\mu$. This method is called an accurate table method, because for each interval we choose a value ν such that $\sqrt{2^r \cdot \nu}$ is exactly representable, and $1/\nu$ is within $1/2^M$ ulp of a representable number, where M is the parameter which determines the accuracy of the table, and is chosen depending on the properties of the function being evaluated. For the second lookup, we use the same proposed instruction with a different integer argument.

The reduction, (3.3), is very accurate because the output ulp is at most $1/2^M$ times the input ulp and we are using a fused multiply-accumulate. The rounding error is equivalent to a 2^{-53-M} perturbation of the input followed by an exact computation.

We calculate the square root of the reduced fraction using a minimax polynomial, $p(c)$, approximating

$$\frac{\sqrt{c+1}-1}{c} \tag{3.4}$$

with a maximum relative error (before rounding) of less than 2^{-53}. The polynomial is approximately equal to the Taylor series

$$p(c) \approx \frac{1}{2} - \frac{c}{8} + \frac{c^2}{32} + O(c^3), \tag{3.5}$$

so given the small size of c, rounding errors will not accumulate and the final result will have strictly less than one ulp error.

To obtain the final result,

$$(2^e \cdot f)^{1/2} = (\sqrt{\mu} \cdot p(c)) \cdot c + \sqrt{\mu} \tag{3.6}$$

where $\sqrt{\mu} = 2^q \cdot \sqrt{2^r \cdot \nu}$, requires

- a multiplication, $2^q \cdot \sqrt{2^r \cdot \nu}$, the result of which is exact,
- a multiplication by $p(c)$ having at most a 2-ulp error,
- a multiply-add with c with norm $< 2^{-N}$ reduces the contribution of the 2-ulp error to 2^{2-N} giving a total maximum error of $\frac{1}{2} + 2^{2-N} \cdot \left(\frac{1}{8}\right)$,

Under the assumption that the difference in ulps between exact square roots and correctly rounded square roots is uniformly distributed in $\left[\frac{-1}{2}, \frac{1}{2}\right]$, we can expect the number of incorrectly rounded results to be bounded by one over the number of intervals. Since we want to use a small interval size to reduce the required order of the polynomial approximation, we therefore expect very high accuracy, which is what we have found in simulations.

The other advantage of using proposed hardware instruction is that we can detect the special inputs and override the second lookup with the right output, whereas the first lookup can be carefully chosen such that intermediate steps never produce a *NaN*. This eliminates the need to detect the special cases in the software implementation.

Chapter 4

New Instructions

New instructions have been proposed (1) to support new algorithms, and (2) because changes in physical processor design render older implementations ineffective.

On the algorithm side, even basic arithmetic continues to improve notably by eliminating variable execution times for subnormals [DTSS05]. Our work extends this to the most important elementary functions.

Driven by hardware implementation, the advent of software pipelining and shortening of pipelining stages favoured iterative algorithms (see, *e.g.*, [SAG99]); the long-running trend towards parallelism has engendered a search for shared execution units [EL93], and in a more general sense, a focus on throughput rather than low latency, which motivates all the proposals (including this thesis) that combine short-latency seed or table value lookups with standard floating point operations, thereby exposing the whole computation to software pipelining by the scheduler.

In proposing Instruction Set Architecture (ISA) extensions, one must consider four constraints:

- the limit on the number of instructions imposed by the size of the machine word, and the desire for fast (i.e., simple) instruction decoding,
- the limit on arguments and results imposed by the architected number of ports on the register file,
- the limit on total latency required to prevent an increase in maximum pipeline depth,
- the need to balance increased functionality with increased area and power usage.

As new lithography methods cause processor sizes to shrink, the relative cost of increasing core area for new instructions is reduced, especially if the new instructions reduce code and data size, reducing pressure on the memory interface which is more difficult to scale.

To achieve a performance benefit, ISA extensions should do one or more of the following

- reduce the number of machine instructions in compiled code,
- move computation away from bottleneck execution units or dispatch queues,
- reduce register pressure.

We propose to add two instructions (with variations as above) :

d = `fmaX` a b c $= a \cdot b + c$ an extended range floating-point multiply-add, with the first argument having 12 exponent bits and 51 mantissa bits, and non-standard exception handling;

t1 = `lookup` a b fn idx an enhanced table look-up with two vector arguments, and two immediate arguments specifying the function and the lookup index. Some functions, like *log*, *sqrt* and *recip*, only use the first vector argument, whereas functions like *atan2*, trigonometric functions and *exp* use both vector arguments. To keep the number of arguments to the lookup instruction the same, we always accept two vector arguments but ignore the second argument (and do not write it) when only one argument is required. The function index *fn*, an integer $\in \{0, 1, .., 7\}$, specifies the function *log*,*exp*, ... *atan2*. The lookup index specifies which of the lookup values associated with the function is returned. Functions that use multiplicative reduction accurate table methods are defined for $idx \in \{0, 1\}$, and undefined otherwise. For other functions, as many as six lookup values are defined via different values of *idx*.

It is easiest to see them used in an example. For all of the functions using multiplicative reduction accurate tables; for example, *log*, *sqrt* and *recip*; the data flow graphs are the same (see Figure 4.1) with the correct lookup specified as an immediate argument to `lookup`, and the final operation being `fma` for the log functions and `fm` otherwise. The figure only shows the data flow (omitting register constants). All of the floating point instructions also take constant arguments which are not shown. For example, for all the multiplicative reduction methods, the `fmaX` takes third argument which is -1, as an addend.

The dotted box is a varying number of fused multiply-adds used to evaluate a polynomial after the multiplicative range reduction performed by

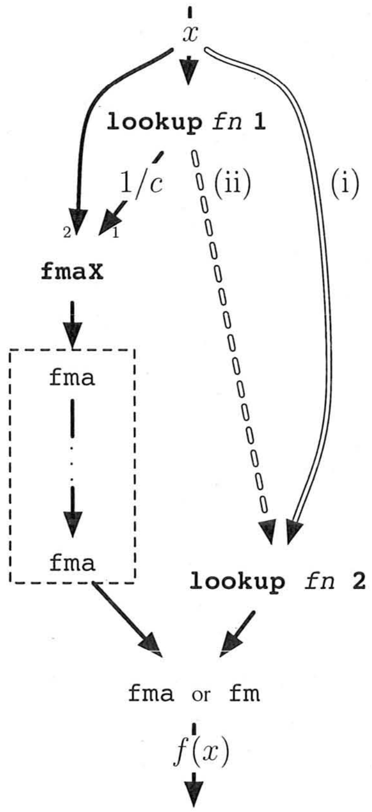

Figure 4.1: Data flow graph with instructions on vertices, for $\log x$, roots and reciprocals. Only the final instruction varies—`fma` for $\log x$ and `fm` for the roots and reciprocals.

the `fmaX`. For the table sizes we have tested, these polynomials are always of order three, so the result of the polynomial (the left branch) is available four floating point operations later (typically about 24-28 cycles) than the result $1/c$. The second `lookup` instruction performs a second lookup, for example, for the *log* function, it looks up $\log_2 c$, and substitutes exceptional results ($\pm\infty$, NaN) when necessary. The final `fma` or `fm` instruction combines the polynomial approximation on the reduced interval with the table value.

The double and double dashed lines indicate two possible data flows for the possible implementations:

(i) the second `lookup` instruction uses the same input;

(ii) the second `lookup` instruction retrieves a value saved by the first `lookup` (in final or intermediate form) from a FIFO queue or rotating scratch registers.

In the first case, the dependency is direct. In the second case the dependency is indirect, via registers internal to the execution unit handling the look-ups. All instruction variations for the different functions have two register inputs and one output, so they will be compatible with existing in-flight instruction and register tracking. Compiler writers will prefer the variants with indirect dependencies, (ii), which reduce register pressure and simplify modulo loop scheduling. In these cases, the input values are only used by the first instruction, after which the registers can be reassigned, while the second `lookup` can be scheduled shortly before its result is required. The case (i), on the other hand, results in a data-dependency graph containing a long edge connecting the input to the last instruction. In simple loops, like a vector library function body, architectures without rotating register files will require as many copy instructions as stages in order to modulo schedule the loop. On many architectures, this cannot be done without a performance degradation.

To facilitate scheduling, it is recommended that the FIFO or tag set be sized to the power of two greater than or equal to the latency of a floating-point operation. In this case, the number of registers required will be less than twice the unrolling factor, which is much lower than what is possible for code generated without access to such instructions.

The combination of small instruction counts and reduced register pressure eliminates the obstacles to in-lining these functions. We recommend that `lookup` be handled by either a load/store unit, or, for vector implementations with a complex integer unit, by that unit. This code is bottlenecked by floating-point instructions, so moving computation out of this unit will increase performance. On the Cell/B.E. SPU, the odd pipeline should be used. On an IBM POWER ISA machine the load/store unit should be used for VMX/Altivec, or scalar instructions. If the variants (ii) are implemented, the hidden registers will require operating system support on operating systems supporting preemptive context switches. Either new instructions to save and restore the state of the hidden registers, or additional functionality for existing context switching support instructions will be required. Alternatively, the processor could delay context switches until the hidden registers are no longer in use, or the process model could avoid the need for context switches altogether, as for example in the systems [AK08, BPBL06].

Chapter 5

Testing Environment

To evaluate the algorithms, we used Coconut (COde CONstructing User Tool), and added the proposed instructions to the Cell/B.E. processor target. Although the software implementations use the Cell/B.E. instruction set, the algorithms do not use any instructions not commonly available in other architectures. The most important existing instruction, which is now commonly available, is the fused multiply and add, which is required to get correctly rounded results.

5.1 Coconut

The functions are implemented in a Domain Specific Language (DSL) embedded in the functional programming language Haskell [PJ$^+$03].

The main advantages of the Haskell embedding are: the ease of adding language features using type classes and higher order functions, and the strong static typing in Haskell which catches many errors at compile time. Some features could be easily implemented using the C preprocessor, or C++ template features, but others, notably table look-ups, would be cumbersome to implement and very difficult to maintain. See [AK09], for a full account of the language, and the results of implementing a single-precision special function library for the Cell/B.E. SPU.

The double precision library, SPU DP MASS, was implemented using this tool, and is distributed starting with the Cell/B.E. SDK 3.1. It was found to be four times faster than the best alternative, SIMD Math, implemented in C using processor intrinsics. Most of the improvements stem from efficient patterns for table look-ups and leveraging higher levels of parallelism through partial unrolling. The performance improvements reported in this thesis are relative to the faster MASS implementations developed using Coconut, so the differences are the result of the improved algorithm and not simply a more efficient implementation.

5.2 Cell/B.E. SPU

The Cell/B.E. (Broadband Engine) architecture contains a POWER architecture microprocessor together with multiple compute engines (SPUs) each with 256K of private local memory, which in our case, put a limit on table size when multiple special functions are used together to process blocks of data.

All SPU computation uses Single Instruction Multiple Data (SIMD) instructions operating on more than one data element packed into a register in parallel. The SPU has a single register file with 128 bit registers. Most instructions operate on components, *i.e.*, four 32-bit integers or 2 64-bit double precision floating point numbers, adding, multiplying or shifting each element in parallel. We call these *pure SIMD* instructions, to distinguish them from operations operating on the register contents as an array of bytes, or a set of 128 bits.

Since SIMD applies a *single* instruction to multiple data, it follows that multiple data elements are treated in the same way. The additional cost of branches in an architecture without deep reordering and branch prediction puts a premium on implementations without any exceptional cases. So it is usually faster to make two versions of a computation and then select the right one based on a third predicate computation than it is to branch and execute one of the two computations. For this reason, all our special functions on the SPU are branch-free. Such branch-free implementations are also useful in real-time applications requiring deterministic execution time.

The SPU has two dispatch pipelines, an even pipeline corresponding roughly to computation, and an odd pipeline including load-store and bit/byte permutations acting on the whole register. By their nature, special functions are computationally bound, so several patterns make special efforts to use odd instructions wherever possible, and we propose the new instructions to be included in the odd pipeline.

5.3 Maple

Maple is a general-purpose mathematical software system, which can perform numerical and symbolic calculations and elementary function evaluations with very high precision. We used Maple to calculate most of the accurate tables and the minimax polynomial approximations of the target function in narrow ranges. High precision Maple functions were also used to test the algorithms proposed in this thesis.

5.4 Keywords and Symbols

Various keywords from Coconut libraries and Haskell are used in the literate code presented in this thesis. This section summarizes some of the keywords and data types.

Coconut defines hardware ISAs in a type class, which is similar to a virtual class in OO languages. This class is implemented in multiple instances, including and interpreter instance and a code-generating instance. This overloading uses parametric type classes [Jon95]. The type class *PowerType* is defined with associated data types [CKJM05] representing different resource types of the processors, for example, *VR* is used for vector registers. We have included the type signatures of the functions in the literate Haskell code. For example, the type signature

$$\mathit{logFamily} :: \mathit{PowerType}\ a \Rightarrow \mathit{MathOptions} \rightarrow \mathit{Bool} \rightarrow \mathit{VR}\ a \rightarrow \mathit{VR}\ a$$

defines the inputs and the outputs of the function *logFamily*. The declaration *PowerType a* adds a constraint that the type *a* must be an instance of the class *PowerType*. Two instances of *PowerType* class are declared in the Coconut library; *INTERP* is used for interpreting the algorithms and *GRAPH* is used for generating codegraphs in the sense of [KAC06], used for assembly code generation. Any instance *a* of the type class *PowerType a* must provide an associated data type *VR a* for vector register values. Data type *MathOptions* is defined to wrap the base in which we want to calculate the function and flags for handled exceptional cases. For example, when *logFamily* is used with the *MathOptions MO2 moAll* it will interpret or generate assembly code for *log2* (base 2), handling all exceptional cases. The input *Bool* is used to identify whether we are calculating *log2p1*, a special function provided to calculate the precise *log* near 1 or standard function *log2*.

Data type *ArbFloat* is provided in the Coconut library to simulate aribitrary-precision floating-point operations. The sign, exponent, mantissa and base are represented as arbitrary precision *Integers*, provided by the standard Haskell library. All the computations are performed using the Haskell *Integer* data type.

$$
\begin{array}{l}
\textbf{data}\ \mathit{ArbFloat} = \mathit{NaN} \mid \mathit{PInf} \mid \mathit{MInf} \\
\quad \mid \mathit{AF}\ \{\ \mathit{afExp} :: \mathit{Integer} \\
\qquad\quad ,\ \mathit{afSig} :: \mathit{Integer} \\
\qquad\quad ,\ \mathit{afSign} :: \mathit{Integer} \\
\qquad\quad ,\ \mathit{afBase} :: \mathit{Integer} \\
\qquad\quad \} \\
\quad \textbf{deriving}\ (\mathit{Show})
\end{array}
$$

Some Haskell operators are frequently used throughout the code presented in this thesis. ($) is equivalent to placing parentheses around the remainder of a clause.

```
($) :: (a -> b) -> a -> b -- Defined in GHC.Base
infixr 0 $
```

The symbol @ is used in Haskell pattern matching to bind the whole value to a name. For example, in

$$\mathit{head1}\ \mathit{list}@(a : as) = a$$

the name *list* could be used in place of whole list, where pattern matching binds the first element to the list to name a.

Chapter 6

Extended Range Doubles and Fused Multiply-Add

The key advantage of the proposed new instructions is that the complications associated with exceptional values (0, ∞, NaN, and values which over- or under-flow at intermediate stages) are internal to the instructions, eliminating branches and predicated execution.

For example, cases similar to 0 and ∞ inputs in square root example treated, in this way. Consider the case when the input to the square root function is ∞, we want to avoid the formation of NaN in the computation of range reduction followed by polynomial evaluation, which has the coefficients of opposite signs. It is achieved by passing special bit patterns as the first `lookup` value, shown in figure 4.1 and modifying the behaviour of `fmaX` for those bit patterns. The behaviour of `fmaX` does not depend on the specific function; depends only the arguments, where the first argument of `fmaX` is always produced by `lookup` instruction; specific to the function. Table 6.1 defines the exceptional behaviour. Only the first input of `fmaX` is in the extended-range format. The second multiplicand, the addend and the result are all IEEE floats.

In Table 6.1, we list the handling of exceptional cases. All exceptional values detected in the first argument are converted to the IEEE equivalent and are returned as the output of the `fmaX`, as indicated by sub-script $_f$ (for final). The NaNs with the sub-scripts are special bit patterns required to produce the special outputs needed for exceptional cases. For example, when `fmaX` is executed with NaN_1 as the first argument (one of the multiplicands) and the other two arguments are finite IEEE values, the result is 2 as an IEEE floating

$+_{ext}$	finite	$-\infty$	∞	NaN
finite	c	c	c	0
$-\infty$	c	c	0	0
∞	c	0	c	0
NaN	c	c	c	0

$*_{ext}$	finite	$-\infty$	∞	NaN
± 0	$\pm 0_f$	$\pm 0_f$	$\pm 0_f$	$\pm 0_f$
finite$\neq 0$	c	2	2	2
$-\infty$	$-\infty_f$	$-\infty_f$	$-\infty_f$	$-\infty_f$
∞	∞_f	∞_f	∞_f	∞_f
NaN_0	NaN_f	NaN_f	NaN_f	NaN_f
NaN_1	2_f	2_f	2_f	2_f
NaN_2	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$	$1/\sqrt{2}_f$
NaN_3	0_f	0_f	0_f	0_f

Table 6.1: Special treatment of exceptional values by `fmaX` follows from special treatment in addition and multiplication. The first argument is given by the row and the second by the column. Conventional treatment is indicated by a "c", and unusual handling by specific constant values.

point number.

$$\begin{aligned}\textbf{\textit{fmaX}}\ \mathrm{NaN}_1\ \mathrm{finite}_1\ \mathrm{finite}_2 &= \mathrm{NaN}_1 \cdot \mathrm{finite}_1 + \mathrm{finite}_2 \\ &= 2\end{aligned}$$

If the result of multiplication is an ∞ and the addend is the ∞ with the opposite sign, then the result is zero, although normally it would be a *NaN*. If the addend is a *NaN*, then the result is zero. For the other values, indicated by "c" in table 6.1, `fmaX` operates as the ***dfma*** instruction provided in the Cell/B.E. SPU hardware [IBM06] except that the first argument is an extended range floating point number. For example, the fused multiplication and addition of finite arguments saturate to $\pm\infty$ in the usual way.

6.1 FMAX Hardware Instruction

The Haskell module presented by this section as a literate program, simulates the proposed ***dfmaX*** instruction, extended floating multiplying add instruction. In order to simulate this instruction, we have used the arbitrary precision functions provided Coconut [AK09] framework. In real hardware, this instruction could be implemented in same fashion as fused multiply-add ***dfma*** instruction. The first argument (one of the multipliers) has extended range for exponents so that subnormals and inverses of subnormals, which saturate to infinity in IEEE representation, can be treated as normal floating point numbers. This representation uses 12 bits for storing exponents, thus doubling the range and 51 bits for mantissa. We assume that the 52nd bit of mantissa is zero, and we have the same precision as before.

module *FMAXHardwareInstr*
  where

dfmaX :: *PowerType a* $\Rightarrow$ *VR a* $\rightarrow$ *VR a* $\rightarrow$ *VR a* $\rightarrow$ *VR a*
dfmaX *a b c* = *result*
  where

We will decompose the extended fused multiply and add ***dfmaX*** into 2 instructions for multiply *multX* and add *addX*, explained below, so that we can define a special argument/result pair relatively easily, and to make the definition analogous to 6.1.

result = *undwrds* \$ *zipWith specialCases* (*dwrds a*) *raddition*

We override the output for special bit patterns in the first argument. These cases are need to output the $NaN^{\,f}$ and the $\pm\infty^f$, described in the table 6.1, for the special cases in many functions. Bit patterns 0x0, 0x7ff8000000000000, 0xfff8000000000000 and 0x7ffc000000000000 are 0, $+\infty$, $-\infty$ and NaN_0 respectively in extended range representation. Bit patterns 0x0, 0x7ff0000000000000, 0xfff0000000000000 and 0x7ff8000000000000 are 0, $+\infty$, $-\infty$ and NaN respectively in IEEE floating point representation.

specialCases x y = **case** *x* **of**
  0x0000000000000000 $\rightarrow$ 0x0000000000000000   -- 0 $\rightarrow$ 0

$$
\begin{array}{lll}
\text{0x7ff8000000000000} \longrightarrow & \text{0x7ff0000000000000} & \text{-- } +\infty \longrightarrow +\infty \\
\text{0xfff8000000000000} \longrightarrow & \text{0xfff0000000000000} & \text{-- } -\infty \longrightarrow -\infty \\
\text{0x7ffc000000000000} \longrightarrow & \text{0x7ff8000000000000} & \text{-- } NaN_0 \longrightarrow NaN
\end{array}
$$

In the computation of *div*, the following special case is needed, where the result is saturating to *Inf*. In those cases, the `lookup` instruction returns the special bit pattern $NaN_1 =$ 0x7ffc000000000001, such that 2 is returned as the result of `fmaX` computation. 2 is represented as 0x4000000000000000 in IEEE floating point representation.

$$
\text{0x7ffc000000000001} \longrightarrow \quad \text{0x4000000000000000} \quad \text{-- } NaN_1 \longrightarrow 2
$$

We use the following special case in the *trig* function, where we want to return $1/\sqrt{2}$ as both sine and cosine values for very large inputs. In these cases, `lookup` instruction returns $NaN_2 =$ 0x7ffc000000000002 as first argument and the result of `fmaX` $1/\sqrt{2} =$ 0x3fe6a09e667f3bcc is returned as the result.

$$
\text{0x7ffc000000000002} \longrightarrow \quad \text{0x3fe6a09e667f3bcc} \quad \text{-- } NaN_2 \longrightarrow \frac{1}{\sqrt{2}}
$$

In the software implementation of *recip* function, we want to return 0 as the result of range reduction, when input is $\pm\infty$. The special case is handled using $NaN_3 =$ 0x7ffc000000000003 bit patterns as first argument of `fmaX` .

$$
\begin{array}{lll}
\text{0x7ffc000000000003} \longrightarrow & \text{0x0000000000000000} & \text{-- } NaN_3 \longrightarrow 0 \\
\_ \longrightarrow & \quad af2DVal\ y &
\end{array}
$$

For the rest of the cases, return the result we obtained from multiplication

$$
\begin{array}{l}
rmult = zipWith\ (\lambda x\ y \longrightarrow multX\ (extDVal2af\ x)\ (dval2af\ y)) \\
\qquad\qquad\qquad (dwrds\ a)\ (dwrds\ b)
\end{array}
$$

and then addition.

$$
\begin{array}{ll}
raddition = zipWith & (\lambda x\ y \longrightarrow addX\ x\ (dval2af\ y)) \\
 & rmult\ (dwrds\ c)
\end{array}
$$

To simulate the fused multiply and add instruction, we used the Arbitrary precision floating point data type, defined in the Coconut libraries. First, we convert the binary representation of an extended range floating point number to *ArbFloat*(arbitrary precision floating point). As explained earlier, we need to extend the range in both directions (near zero and near infinity), hence the bias is increased from standard IEEE 1023 to 2047 for extended range floating point numbers, and 12 bits are used for storing the exponent. The mantissa is stored using the last 51-bits, but we calculated all the extended range values with implied zero, so we do not loose any precision.

$$
\begin{array}{l}
\mathit{extDVal2af} :: \mathit{Integer} \rightarrow \mathit{ArbFloat} \\
\mathit{extDVal2af}\ v = \mathbf{case}\ (\mathit{exp}', \mathit{mantissa}', \mathit{sign})\ \mathbf{of} \\
\qquad\qquad (\mathrm{0xfff}, 0, 0) \rightarrow \mathit{PInf} \\
\qquad\qquad (\mathrm{0xfff}, 0, 1) \rightarrow \mathit{MInf} \\
\qquad\qquad (\mathrm{0xfff}, \_, \_) \rightarrow \mathit{NaN} \\
\qquad\qquad (0, 0, \_) \quad\ \ \rightarrow \mathit{AF}\ (-2099)\ 0\ 1\ 2 \\
\qquad\qquad \_ \qquad\qquad\ \ \rightarrow \mathit{AF}\ (\mathit{exp}' - 2047 - 52)\ \mathit{mantissa} \\
\qquad\qquad\qquad\qquad\qquad (\mathbf{if}\ \mathit{sign} \equiv 0\ \mathbf{then}\ 1\ \mathbf{else} - 1)\ 2 \\
\quad \mathbf{where} \\
\qquad (\mathit{signExp}', \mathit{mantissa}') = \mathit{divMod}\ v\ (2 \uparrow 51) \\
\qquad (\mathit{sign}, \mathit{exp}') \qquad\qquad\ = \mathit{divMod}\ \mathit{signExp}'\ (2 \uparrow 12) \\
\qquad \mathit{mantissa} \qquad\qquad\qquad = 2 \uparrow 52 + \mathit{shiftL}\ \mathit{mantissa}'\ 1
\end{array}
$$

The following function returns the multiplication of an extended range double floating point number with an IEEE floating point number; both converted to AF(arbitrary precision floating point) first; returning an arbitrary precision floating point. We override the output according to the 6.1. number.

$$
\begin{array}{l}
\mathit{multX} :: \mathit{ArbFloat} \rightarrow \mathit{ArbFloat} \rightarrow \mathit{ArbFloat} \\
\mathit{multX}\ \_\ \mathit{NaN}\ = 2 \\
\mathit{multX}\ \_\ \mathit{PInf}\ = 2 \\
\mathit{multX}\ \_\ \mathit{MInf} = 2 \\
\mathit{multX}\ \mathit{x1}@(\mathit{AF}\ \_\mathit{exp1}\ \_\mathit{sig1}\ \_\mathit{sign1}\ 2)\ \mathit{x2}@(\mathit{AF}\ \_\mathit{exp2}\ \_\mathit{sig2}\ \_\mathit{sign2}\ 2) \\
\qquad\qquad = \mathit{x1} * \mathit{x2} \\
\mathit{multX}\ x\ y = \mathit{error}\ \$\ \texttt{"FMAXHardwareInstr.Impossible"} \\
\qquad\qquad +\!\!+\ \ \mathit{show}\ (x, y)
\end{array}
$$

The following function returns the addition of an arbitrary precision floating point number and an IEEE floating number, converted to arbitrary precision floating point number first.

$$
\begin{array}{l}
\mathit{addX} :: \mathit{ArbFloat} \rightarrow \mathit{ArbFloat} \rightarrow \mathit{ArbFloat} \\
\mathit{addX}\ \mathit{PInf}\ \mathit{MInf}\ = 0 \\
\mathit{addX}\ \mathit{MInf}\ \mathit{PInf}\ = 0
\end{array}
$$

$$
\begin{array}{ll}
addX\ \_\ NaN & = 0 \\
addX\ \_\ PInf & = PInf \\
addX\ \_\ MInf & = MInf \\
addX\ x1@(AF\ \_exp1\ \_sig1\ \_sign1\ 2)\ x2@(AF\ \_exp2\ \_sig2\ \_sign2\ 2) & \\
\qquad = x1 + x2 & \\
addX\ x\ y & = error\ \$\ \texttt{"FMAXHardwareInstr.Impossible"} \\
 & \quad +\!\!+\ show\ (x, y)
\end{array}
$$

6.2 Special Bit Patterns

At many instances in the software implementation of the functions and the implementations of lookup instructions, we have used variable names like *nan1X*, *infinityX* or *oneX* to represent the speical bit patterns, NaN_1, ∞, and constants 1, in extended precision, whereas symbols like *nan*, *infinity* or *one* to represent the constant values in IEEE floating point representation.

Chapter 7

Lookup Instructions

In this chapter, the values returned by the hardware lookup instruction for the different elementary functions are reported. The implementation specific to individual functions (corresponding to different immediate arguments) of the lookup instruction is included in their respective chapters. The Haskell module *LookupOpcode* only provides a wrapper for the lookup functions, given in the later chapters. The actual implementation of the *LookupOpcode* module is therefore omitted. The lookup values returned from the ***lookupOpcode*** instruction for the different functions and for the different input ranges are included in the tables given below.

7.1 Lookup Opcode

In the following tables, we use e to represent the unbiased exponent of the input and c to represent the value of the reduction factor, one of the accurate table values. NaN_i are special bit patterns used by ***dfmaX*** to output special values. The usage of these special values are reported in Table 6.1. The subscript ext is used to denote the values in the extended range floating point representation. The subscript sat is used for the IEEE values; saturated to 0 and $\pm\infty$.

$\lvert Input \rvert$	Lookup 1	Lookup 2
finite	$\left(-1^s \frac{2^{-e}}{c}\right)_{ext}$	$\left(-1^s \frac{2^{-e}}{c}\right)_{sat}$
$< 2^{-1024}$	NaN_3	$-1^s\infty$
∞	NaN_3	0
NaN	$\left(-1^s \frac{2^{-e}}{c}\right)_{ext}$	NaN

Table 7.1: The values returned by ***lookupOpcode*** instruction for the function *recip*; for the different ranges of the input.

The first lookup value for the function *recip* is an extended range floating point number and the second lookup is an IEEE floating point number.

$\lvert Output \rvert$	Lookup 1	Lookup 2
finite	$\left(-1^s \frac{2^{-e}}{c}\right)_{ext}$	$\left(-1^s \frac{2^{-e}}{c}\right)_{ext}$
∞	NaN_1	NaN_3
NaN	$\left(-1^s \frac{2^{-e}}{c}\right)_{ext}$	NaN_0

Table 7.2: The values returned by ***lookupOpcode*** instruction for the function *div* for the different ranges of the output.

It is important to note that not all resultant infinities are produced using the special case reported above. For some finite arguments, the algorithm used for normal cases can also saturate to infinity. The first lookup value for *div* function is an extended range floating point number and the second lookup value is an IEEE floating point number.

Input	Lookup 1	Lookup 2
finite > 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	$2^q\sqrt{2^r \cdot c}$
$= 0$	$\left(\frac{2^{-e}}{c}\right)_{ext}$	0
< 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN
∞	$\left(\frac{2^{-e}}{c}\right)_{ext}$	∞
NaN	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN

Table 7.3: The values returned by ***lookupOpcode*** instruction for the function *sqrt* for the different ranges of the input.

The first lookup value for the function *sqrt* is an extended range floating point number and the second lookup value is an IEEE floating point number. In the table 7.3, the symbols q and r are used to represent the quotient and the remainder of the division of e by 2.

Input	Lookup 1	Lookup 2
finite > 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	$2^q\sqrt{\frac{2^r}{c}}$
$= 0$	$\left(\frac{2^{-e}}{c}\right)_{ext}$	∞
< 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN
∞	$\left(\frac{2^{-e}}{c}\right)_{ext}$	0
NaN	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN

Table 7.4: The values returned by ***lookupOpcode*** instruction for the function *rsqrt*; reciprocal square root; for the different ranges of the input.

The first lookup value for the function *rsqrt* is an extended range floating point number and the second lookup value is an IEEE floating point number. The symbols q and r are used in the table 7.4, represents the quotient and the remainder of the division of $-e$ by 2.

Input	Lookup 1	Lookup 2
finite > 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	$e + log_2(c)$
$= 0$	0	$-\infty$
< 0	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN
∞	$\left(\frac{2^{-e}}{c}\right)_{ext}$	∞
NaN	$\left(\frac{2^{-e}}{c}\right)_{ext}$	NaN

Table 7.5: The values returned by ***lookupOpcode*** instruction for the function *log* for the different ranges of the input.

The first lookup value for the function *log* is an extended range floating point number and the second lookup value is an IEEE floating point number. All the versions of the *log* functions use the same lookup values, but with different arguments.

Input	Lookup 1	Lookup 2
$-1074 < finite < 1024$	c	$2^{[Input]} \cdot 2^c$
< -1074	NaN	0
> 1024	NaN	∞
NaN	c	NaN

Table 7.6: The values returned by ***lookupOpcode*** instruction for the function *exp* for the different ranges of the input.

In table 7.6, $[Input]$ is used to represent the integer closest to the input. Both lookup values for the function *exp* are IEEE floating point numbers. All the versions of the *exp* function use the same lookup instruction.

Input	Lookup 1	Lookup 2	Lookup 3	Lookup 4
$< 2^{42}$	$-c_{high,ext}$	$-c_{low,ext}$	$\pm\cos(c)$	$\pm\sin(c)$
$> 2^{42}$	NaN_0	NaN_0	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$
NaN	$-c_{high,ext}$	$-c_{low,ext}$	NaN	NaN

Table 7.7: The values returned by ***lookupOpcode*** instruction for the function *trig* for different ranges of inputs.

The sub-scripts *high* and *low* are used in the table 7.7, represent IEEE value of c and $c - c_{high}$ respectively, where c is calculated using Maple with 500 digits. All the lookup values for *trig* function are extended range floating point numbers. All the trigonometric functions use the same lookup instruction.

Input	Lookup 1	Lookup 2	Lookup 3	Lookup 4	Lookup 5	Lookup 6
(a,b)	$\tilde{\max}(a,b)$	$\tilde{\min}(a,b)$	c	$\arctan(c)$	± 1	$\pm\pi, \pm\frac{\pi}{2}, 0$
$\frac{a}{b} = 0/\infty$	1	1	c	$\arctan(c)$	0	$\pm\pi, \pm\frac{\pi}{2}, 0$
$\frac{a}{b} = \mathrm{NaN}$	$\tilde{\max}(a,b)$	$\tilde{\min}(a,b)$	c	$\arctan(c)$	± 1	NaN

Table 7.8: TheThe values returned by ***lookupOpcode*** instruction for the function *atan2* for different ranges of inputs.

All the lookup values for the function *atan2* are IEEE floating points. The inverse trigonometric functions use *atan2* with appropriate trigonometric identities.

Chapter 8

Logarithmic Family Functions

The algorithm used to evaluate *log* is a simplified version of the accurate table algorithm [AS10] we developed previously. With new proposed instructions, the special treatement needed to handle exceptional and subnormal inputs can be ignored. The algorithm follows three phases, visible in the figure 4.1:

1. The input is reduced to the smaller range $\in [-2^{-N}..2^{-N}]$, using multiplicative reduction.

$$f = \left(2^{-e}/c\right)_{lookup} * v - 1, \tag{8.1}$$

   where 2^N-pairs of $(1/c, log_2(c))$ are used to construct the table and e is the unbiased exponent of the input.

2. The polynomial of considerably smaller order is used to evaluate reduced input.

$$\frac{log_2(1+f)}{f} = poly(f), \tag{8.2}$$

   where *poly* is a minimax approximation of $\frac{log_2(1+f)}{f}$ calculated using high precision Maple.

3. The polynomial evaluation is added to the value of the function, corresponding to the reduction factor returned by the second lookup.

$$log_2(v) = poly(f) * f + (e + log_2(c))_{lookup} \tag{8.3}$$

A fused multiply add **dfma** is used to get correctly rounded results.

In the standard library, $logp1 = log_e(1+v)$ is a function provided to get accurate values of log for very small inputs near 1. A very similar algorithm is used to calculate this special function, governed by the following equations.

$$\begin{aligned} f &= (2^{-e}/c)_{lookup} * v + \left((2^{-e}/c)_{lookup} - 1\right) \\ \frac{log_2(1+f)}{f} &= poly(f) \\ log_2(1+v) &= poly(f) * f + (e + log_2(c))_{lookup} \end{aligned}$$

where, e, c, and the lookup values are determined by $1+v$.

8.1 Log Software Implementation

This Haskell module implements the proposed algorithm for all the variants of log functions. All the log functions are calculated with base 2, then scaled up or down by constant factors accordingly.

```
module LogSoft
  where
```

```
logFamily :: PowerType a ⇒ MathOptions → Bool → VR a
                                → VR a
logFamily (MathOptions base _exceptions) isP1Case v = result
  where
```

In the case of *isP1Case*, the lookup instruction is executed using the argument $v+1$, otherwise the lookup values are calculated using v. Function index 0 is used to specify the *log* function.

```
vPlus1 =  dfa v (undoubles2 1)
vOrVplus1 = if isP1Case
              then vPlus1
              else v
[oneByC, log2ExpC] = map (lookupOpcode vOrVplus1 vOrVplus1
                                                  0) [1, 2]
```

This is the range reduction step in accordance with the given equation.

```
fracMOffset = if isP1Case
                then dfmaX oneByC v
```

$$
\begin{array}{l}
\qquad\qquad\qquad (\textbf{dfmaX}\ \mathit{oneByC}\ (\mathit{undoubles2}\ 1) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{undoubles2}\ (-1))) \\
\qquad \textbf{else dfmaX}\ \mathit{oneByC}\ v\ (\mathit{undoubles2}\ (-1))
\end{array}
$$

The Horner polynomial is used to calculate the log of $1 + \mathit{fracMOffset}$. The following Maple code is used to generate the minimax polynomial.

```
Digits := 100;
numSegments := 2^(12);
polyOrd := 3;
b := 1/numSegments;
plog2:=numapprox[minimax](x->limit(log[2](1+y)/y,y=x)
                          ,-b..b,[polyOrd,0],1,'erLog');
log[2](erLog);
lprint([seq(roundDbl(coeff(plog2(x),x,j)),j=0..polyOrd)]);
```

The last step to combine the result of polynomial evaluation with the second lookup value is merged into the following step, outputting $log_2(v)$ or $log_2(v+1)$, depending on the case.

$$\mathit{evalPoly} = \mathit{hornerVDbl}\ (\mathit{log2ExpC} : \mathit{fixedCoeffs})\ \mathit{fracMOffset}$$

The output is scaled to get the result for different bases. Both high and low parts of constants are used to get correctly rounded results.

$$
\begin{array}{lcl}
\mathit{result} & = & \textbf{case}\ \mathit{base}\ \textbf{of} \\
 & & \quad \mathit{MO2} \ \ \rightarrow \mathit{evalPoly} \\
 & & \quad \mathit{MO10} \rightarrow \textbf{dfma}\ \mathit{evalPoly}\ \mathit{log10ScaleHigh}\ \$ \\
 & & \qquad\qquad\ \ \textbf{dfm}\ \mathit{evalPoly}\ \mathit{log10ScaleLow} \\
 & & \quad \mathit{MOe} \ \ \rightarrow \textbf{dfma}\ \mathit{evalPoly}\ \mathit{logeScaleHigh}\ \$ \\
 & & \qquad\qquad\ \ \textbf{dfm}\ \mathit{evalPoly}\ \mathit{logeScaleLow} \\
 & & \quad \_ \qquad \rightarrow \mathit{error}\ \$\ \texttt{"LogSoft.base "} \\
 & & \qquad\qquad\ \ +\!\!+\ \mathit{show}\ \mathit{base}\ +\!\!+\ \texttt{" not supported"}
\end{array}
$$

8.2 Overview of Log Lookup Logic

A simplified data-flow for the most complicated case, $\log_2 x$, is represented in Figure 8.1. The simplification is the elimination of the many single-bit

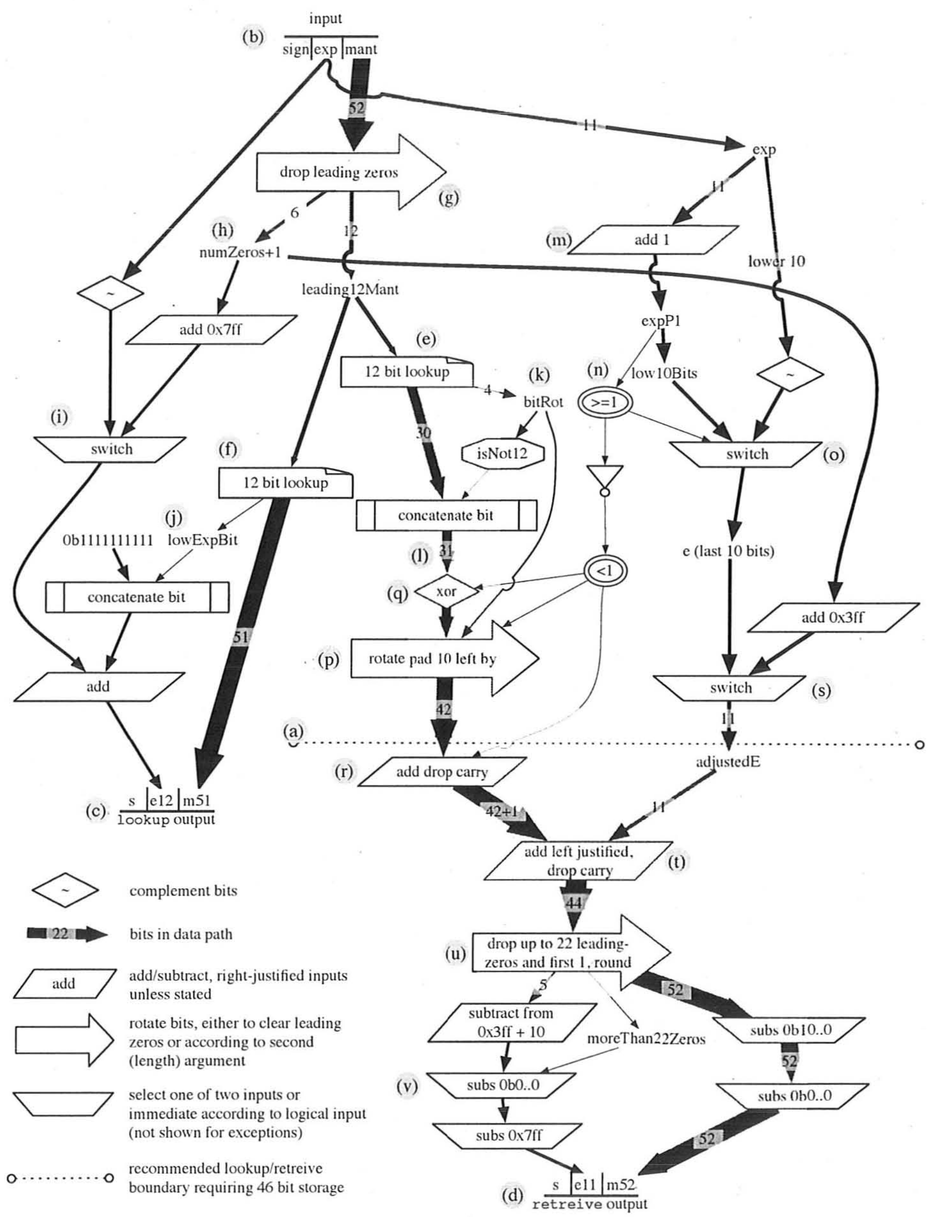

Figure 8.1: Bit flow graph with operations on vertices, for $\log x$ lookup. Shape indicates operation type, and line width indicates data paths width in bits.

operations necessary to keep track of exceptional conditions. The operations to substitute special values are still shown. The purpose of this diagram is to show that the operations around the core look-up operations are of low complexity. This representation is for variant (ii) for the second `lookup` and includes a dotted line (a) (in Figure 8.1) showing a possible set of values to save at the end of the first `lookup` with the part of the data flow below the line computed when the `lookup` instruction is called with the second index value.

The input (b) is used to generate two values (c) and (d), $2^{-e}/\mu$ and $e + \log_2 \mu$ in the case of $\log_2 x$. The heart of the operation contains the two look up operations (e) and (f), with a common index. In implementation (i) the look ups would be implemented separately, while in the shared implementations (ii) and (iii), the lookups would probably be implemented together. Partial decoding of subnormal inputs (g) is required for all of the functions except the exponential functions. Only the leading non-zero bits are needed for subnormal values, and only the leading bits are needed for normal values, but the number of leading zero bits (h) is required to properly form the exponent for the multiplicative reduction. The only switch (i) needed for the first `lookup` output switches between the reciprocal exponents valid in the normal and subnormal cases respectively. Accurate range reduction for subnormals requires both extreme end points, *e.g.* 1/2 and 1, because these values are exactly representable. As a result, two exponent values are required, and we accommodate this by storing an exponent bit (j) in addition to the 51 mantissa bits.

On the right hand side, the look up (e) for the second `lookup` operation also looks up a 4-bit rotation, which also serves as a flag. We need 4 bits because the table size 2^{12} implies that we may have a variation in the exponent of the leading nonzero bit of up to 11 for nonzero table values. This allows us to encode in 30 bits the floating mantissa used to construct the second `lookup` output. This table will always contain a 0, and we encode this as a 12 in the bitRot field. In all other cases, the next operation concatenates the implied 1 for this floating-point format. This gives us an effective 31 bits of significance (l), which are then rotated into the correct position in a 42-bit fixed point number. Only the high-order bit overlaps the integer part of the answer generated from the exponent bits, so this value needs to be padded. Because we output an IEEE float, the contribution of the (padded) value to the mantissa of the output will depend on the sign of the integer exponent part. This sign is computed by adding 1 (m) to the biased exponent, in which case the high-order bit is 1 if and only if the exponent is positive. This bit

(n) is used to control the sign reversal of the integer part (o) and the sign of the sign reversal of the fractional part, which is optimized by padding (p) after xoring (q) but before the +1 (r) required to negate a two's-complement integer.

The integer part has now been computed for normal inputs, but we need to switch (s) in the value for subnormal inputs which we obtain by biasing the number of leading zeros computed as part of the first step. The apparent 75-bit add (t) is really only 11 bits with 10 of the bits coming from padding on one side. This fixed-point number may contain leading zeros, but the maximum number is $\log_2$((maximum integer part) − (smallest nonzero table value)) = 22, for the tested table size. As a result the normalization (u) only needs to check for up to 22 leading zero bits, and if it detects that number set a flag to substitute a zero for the exponent (v) (the mantissa is automatically zero). The final switches substitute special values for $\pm\infty$ and a quiet NaN.

8.3 Log Lookup Instruction

module *LogLookup*
where

The implementation of the log family functions are based on the algorithms presented by [AS09]. In this thesis, we have proposed hardware instructions to improve the performance of the log functions. With the new proposed instructions, all the operations needed, in the paper [AS09], to handle exceptional cases and subnormal inputs are no longer needed. The proposed hardware lookup instruction returns the multiplicative reduction factor and the value of the function corresponding to the reduction. Unlike in [AS09], the lookup values are not fixed table values, but adjusted to the input. The first lookup value is the multiplicative reduction factor $2^{-e}/c$, in extended range double floating point representation, where e is the exponent of the input. In the case of subnormal inputs, the reduction factor is adjusted by the number of leading zeros in the mantissa of the input, such that $2^{-e} \times x \in [1, 2)$ and $2^{-e}/c \times x \in [1 - 2^{-N}, 1 + 2^{-N}]$ holds for all cases (normal and subnormal). Corresponding adjustments are done for the second lookup value, which returns $e + log_2(c)$. Since in extended precision we treat subnormals as normal numbers, we do not need to do anything special for large inputs to calculate the first lookup. Subnormal inputs that do not saturate to Infinity, however, need to be boosted up and then have the lookup calculation corresponding to the boosted value applied to them.

This version of log lookup is implemented in such a way as to make the mathematical semantics of the intermediate variables clear. In some instances this requires some extra computation, but makes it easier to follow the algorithm. A more optimized version closer to the expected hardware implementation is also presented after this function.

$$\begin{array}{l} \mathit{logLookup} :: \mathit{Integer} \rightarrow (\mathit{Integer}, \mathit{Integer}) \\ \mathit{logLookup}\ x = (\mathit{fstLookup}, \mathit{sndLookup}) \\ \quad \mathbf{where} \end{array}$$

For the ease for future development and changes in the table, various constants related to the table are parameterized. A 2^n-interval table with the values $1/c, log_2(c)$, where $c \in [1, 2]$, is constructed. Other constants are explained later as they are used.

$$(n, \mathit{table}, m, k, \mathit{uniqueZero}) = (12, \mathit{log2TableFixed}, 30, 11, \mathrm{0xc})$$

We decompose the input into sign, exponent and mantissa. No logical operations are required at this step.

$$\begin{array}{ll} (\mathit{sign}, \mathit{rest}) = & \mathit{divMod}\ x\ (2 \uparrow 63) \\ (\mathit{exponent}, \mathit{mantissa}) = & \mathit{divMod}\ \mathit{rest}\ (2 \uparrow 52) \end{array}$$

At several points, we need to treat subnormal inputs differently, which is controlled by this bit value.

$$\mathit{expIsZero} = \mathit{exponent} \equiv 0$$

The lookup key is the n leading bits of the significant of the normalized number. For subnormal inputs, we can construct the normalized number by counting the leading zeros and shifting the significant left until the first non-zero bit falls out. At the same time, dropped off leading zeros are required to construct the exponent of the normalized number. These operations result in the boosting of sub-normal inputs into normal range by multiplying it by 2^{52}.

$$\begin{array}{l} (\_, \mathit{leading0s}) = \mathit{countLeadingZeros}\ 52\ 52\ \mathit{mantissa} \\ (\mathit{adjustedExponent}, \mathit{adjustedMantissa}) = \end{array}$$

$$
\begin{aligned}
&\textbf{if } \mathit{expIsZero} \\
&\textbf{then } (52 - \mathit{fromIntegral\ leading0s}, \\
&\quad \mathit{bits}\ 0\ 52\ (\mathit{shiftL\ mantissa}\ (\mathit{leading0s} + 1))) \\
&\textbf{else } (\mathit{exponent}, \mathit{mantissa})
\end{aligned}
$$

Based on the leading n-bits of *adjustedMantissa*, the values from the table are retrieved. Table values are calculated to keep the table size as small as possible and in agreement with the new extended range floating point representation. The inverse of $c \in [1, 2]$ is calculated such that the last bit (52nd) of its mantissa is 0, which is implied in the extended range representation. All the $1/c \in [0.5, 1]$ values have exponents either 0x3ff or 0x3fe, so we only store the last bit of the exponent. At the same time, the mantissas of the $log_2(c)$ values have only m non-zero bits, so the last $52 - m$ zeros bits are not stored and are implied. Since the exponent of $log_2(c)$ has a much smaller range than the standard 11-bit exponents of the IEEE representation, we only need to store $\lceil log_2 k \rceil$ bits, representing the difference of the exponent from the smallest exponent k, other than 0, in the table. This difference also gives us the amount of rotation we need to construct a fixed point integer for $log_2(c)$. $log_2(c) = 0$ is treated as a special case, storing a unique bit pattern in the *rotation* bit field.

$$
\begin{aligned}
&\mathit{lookupKey} \ = \mathit{bits}\ (52 - n)\ 52\ \mathit{adjustedMantissa} \\
&((\mathit{rotation}, \mathit{mantissaLog2C}), (\mathit{lastBit}, \mathit{mantissaOneByC})) \\
&\qquad\qquad = \mathit{table}\ !!\ (\mathit{fromIntegral\ lookupKey})
\end{aligned}
$$

We concatenate the *lastBit* to 0x3fe to get the exponent of $1/c$.

$$\mathit{exponentOneByC} = \text{0x3fe} + \mathit{lastBit}$$

Now, we need to construct the exponent part of $1/x_{approx} = 2^{-e}$ in extended range representation, where the bias is 0x7ff. Complementing the exponent bits with bias 0x3ff returns $-e + 1+$ 0x3ff. Adding *exponentOneByC* which is already biased by 0x3ff to *exponentComplemented* will return the desired exponent biased by 0x7ff.

$$\mathit{exponentComplemented} = \textbf{xor}\ \mathit{adjustedExponent}\ \text{0x7ff}$$

In the case of subnormal inputs, we need to account for the boost factor 2^{52}, by adding 52 to the exponent of extended range $1/x_{approx}$.

$$
\begin{aligned}
&\mathit{approx1ByInputX} = \\
&\quad \textbf{if } \mathit{expIsZero} \\
&\quad \textbf{then } (\mathit{exponentComplemented} + \mathit{exponentOneByC} + 52) * (2 \uparrow 51) \\
&\qquad\quad + \mathit{mantissaOneByC} \\
&\quad \textbf{else } \ (\mathit{exponentComplemented} + \mathit{exponentOneByC}) * (2 \uparrow 51) \\
&\qquad\quad + \mathit{mantissaOneByC}
\end{aligned}
$$

In order to compute the other lookup value, $e_{unbiased} + log_2 c$, we convert the $log_2 c$ and $e_{unbiased}$ to fixed point integers by multiplying them by 2^{52+k}, then adding them together and converting the result back to the floating point representation. The implied 1 of the floating point representation needs to be added in all cases, except in the first interval with $log_2(c) = 0$.

$$
\begin{aligned}
\mathit{impliedOneBit} \quad &= \textbf{if } \mathit{rotation} \equiv \mathit{uniqueZero} \\
&\quad \textbf{then } 0 \textbf{ else } 1 \\
\mathit{mantissa52Bits} \quad &= (2 \uparrow 52) * \mathit{impliedOneBit} \\
&\quad + \mathit{shiftL}\ \mathit{mantissaLog2C}\ (52 - m)
\end{aligned}
$$

mantissa52Bits is shifted up to $\lceil log_2(k) \rceil$ bits, in order to get the fixed point integer $2^{52+k} \times log_2(c_{floating})$.

$$\mathit{fixedLog2C} \quad = \mathit{shiftL}\ \mathit{mantissa52Bits}\ (\mathit{fromIntegral}\ \mathit{rotation})$$

We need to treat the inputs < 1 differently from those ≥ 1, where the unbiased exponent is positive. We calculate either $e_{unbiased} - log_2(c)$ or $e_{unbiased} + log_2(c)$, based on the sign of $e_{unbiased}$, and put the corresponding sign in the sign bit field of the result. We can calculate the value of the unbiased exponent by adding 1, and letting the 11th bit fall off, in the case of the inputs ≥ 1. The falling bit tells us the sign of the unbiased exponent. In case of the inputs < 1, the last 10 bits of the complement of the biased exponent returns the absolute value of the unbiased exponent, which has already been calculated in order to construct $1/x_{approx}$.

$$\mathit{adjustedExponentP1} \quad = \mathit{adjustedExponent} + 1$$

We need to switch between various operations based on the sign of the unbiased exponent.

$$
\begin{aligned}
\mathit{npSwitch}\ n\ p \quad = \quad &\textbf{if } \mathit{bits}\ 10\ 11\ \mathit{adjustedExponentP1} \equiv 1 \\
&\textbf{then } p
\end{aligned}
$$

$$\begin{aligned}
&\qquad\qquad\qquad \textbf{else}\ n \\
\mathit{unbiasedExponent} &= \mathit{bits}\ 0\ 10 \\
&\qquad \$\ \mathit{npSwitch}\ \mathit{exponentComplemented} \\
&\qquad\quad \mathit{adjustedExponentP1}
\end{aligned}$$

In the case of subnormal inputs, the unbiased exponent must be adjusted which extends it to 11-bits.

$$\begin{aligned}
\mathit{adjustE} = \ & \textbf{if}\ \mathit{expIsZero} \\
& \textbf{then}\ \mathit{unbiasedExponent} + 52 \\
& \textbf{else}\ \mathit{unbiasedExponent}
\end{aligned}$$

We convert the adjusted exponent to fixed-point representation, multiplied up by the same factor 2^{52+k}.

$$\mathit{eShifted} = \mathit{shiftL}\ \mathit{adjustE}\ (k + 52)$$

We add or subtract *fixedLog2C* from the absolute value of the unbiased exponent. In hardware implementation, it is a switch between *fixedLog2C* or its 2's-complement, and then the left adjusted $m + k$ bit adder can be used.

$$\begin{aligned}
\mathit{ePlog2CInt}\ = \ & \mathit{npSwitch}\ (\mathit{eShifted} - \mathit{fixedLog2C}) \\
& (\mathit{eShifted} + \mathit{fixedLog2C})
\end{aligned}$$

Now we construct the $e_{unbiased} + log_2(c)$ floating point representation from its fixed point integer by counting the leading zeros and shifting the bits right until first non-zero bit is aligned to the position 53, the position of implied one in floating point representation, and then mask it off. In order to have correct rounding we need to have rounded shift. We do not need to count all the zeros, because we know the smallest number resulting from the above computation is $\min(1 - \max log_2(c), 0 + \min log_2(c)) * 2^{52+k}$ (except 0), which are both $52 + k$ bits long. Thus, we only need to count up to $11 + k$ leading zeros. If all of the first $11 + k$ bits are zeros, we return 0.

$$\begin{aligned}
(\mathit{isZero}, \mathit{leadingZeros}) \quad &= \mathit{countLeadingZeros}\ (52 + 11 + k)\ (11 + k) \\
&\qquad \mathit{ePlog2CInt} \\
\mathit{mantissaEplog2C} &= \textbf{if}\ \mathit{isZero} \\
&\quad \textbf{then}\ 0 \\
&\quad \textbf{else}\ \mathit{bits}\ 0\ 52\ \$ \\
&\quad \mathit{roundShiftR}\ \mathit{ePlog2CInt} \\
&\quad (52 + 11 + k - \mathit{fromIntegral}\ \mathit{leadingZeros} - 53)
\end{aligned}$$

The unbiased exponent of the floating point representation of *ePlog2CInt* is the number of bits which fell off on the right plus k, since we multiplied by 2^{52+k}. Add the bias 0x3ff to the unbiased exponent.

$$\begin{aligned} \mathit{expEplog2C} = \ & \textbf{if } \mathit{isZero} \\ & \textbf{then } 0 \\ & \textbf{else } \mathrm{0x3ff} + 52 + k + 11 - \mathit{leadingZeros} - 53 - k \end{aligned}$$

Concatenate the individual parts of *ePlog2CInt* together.

$$\begin{aligned} \mathit{approxLog2Input} \ = \ & (\mathit{npSwitch}\ 1\ 0) * 2 \uparrow 63 \\ & + (\mathit{fromIntegral}\ \mathit{expEplog2C}) * 2 \uparrow 52 \\ & + \mathit{mantissaEplog2C} \end{aligned}$$

Finally, we check for input values requiring special treatment (0, ∞, NaN, ≤ 0), and produce special results.

$$\begin{array}{ll} (\mathit{fstLookup}, \mathit{sndLookup}) & = \textbf{case } (\mathit{sign}, \mathit{exponent}, \mathit{mantissa}) \textbf{ of} \\ \quad (0, 0, 0) & \rightarrow (0, \mathrm{0xfff0000000000000}) \\ \quad (0, \mathrm{0x7ff}, 0) & \rightarrow (0, \mathrm{0x7ff0000000000000}) \\ \quad (0, \mathrm{0x7ff}, \_) & \rightarrow (0, \mathrm{0x7ff8000000000000}) \\ \quad (1, \_, \_) & \rightarrow (0, \mathrm{0x7ff8000000000000}) \\ \quad \_ & \rightarrow (\mathit{approx1ByInputX}, \mathit{approxLog2Input}) \end{array}$$

8.4 Log Lookup Optimized for Hardware Implementation

The previous implementation uses potentially expensive operations which make the mathematics clearer. In this section, we provide an alternative implementation which makes the high-level transformations required for an efficient hardware implementation. The reader should be able to get a feeling for the logical complexity in terms of bits and gates by inspecting this implementation. The transformations required to support the other `lookup` variants are similar, and not given in this thesis.

$$\begin{aligned} & \mathit{logLookupOptimized} :: \mathit{Integer} \rightarrow (\mathit{Integer}, \mathit{Integer}) \\ & \mathit{logLookupOptimized}\ x = (\mathit{fstLookup}, \mathit{sndLookup}) \\ & \quad \textbf{where} \end{aligned}$$

We will keep using these parameters as much as possible, but some steps will use their specific values.

$$(n, table, m, k, \_uniqueZero) = (12, log2TableFixed, 30, 11, 0\text{xc})$$

Decomposing the input into a different bit field does not require any logical operation.

$$\begin{aligned}(sign, rest) &= divMod\ x\ (2 \uparrow 63)\\ (exponent, mantissa) &= divMod\ rest\ (2 \uparrow 52)\end{aligned}$$

$expIsZero$ kept the same as the previous implementation.

$$expIsZero = exponent \equiv 0$$

Instead of constructing a normalized number which we do not require, we directly extract the lookup key from the mantissa of the input. This can be implemented as a shift left, possibly integrating the count leading zeros operation, but simplified so as to produce only 12 output bits.

$$\begin{aligned}&(\_, leading0s) = countLeadingZeros\ 52\ 52\ mantissa\\ &lookupKey =\\ &\quad \textbf{if}\ expIsZero\\ &\quad \textbf{then}\ bits\ (52 - n)\ 52\ (shiftL\ mantissa\ (leading0s + 1))\\ &\quad \textbf{else}\ bits\ (52 - n)\ 52\ mantissa\end{aligned}$$

$$\begin{aligned}&((rotation, mantissaLog2C), (lastBit, mantissaOneByC))\\ &\quad = table\ !!\ (fromIntegral\ lookupKey)\end{aligned}$$

In the case of normal input, the complement of the exponent is calculated to $-e + 1 +$ $bias$.

$$exponentComplemented = \textbf{xor}\ exponent\ 0\text{x7ff}$$

The operation to calculate the exponent of the extended range multiplicative reduction factor $-e_{redux} + 1 + bias = leading0s + 0\text{x7ff}$ is decomposed into 2 steps. The first step requires a 12-bit adder, which adds 0x3ff, followed by a 2-bit adder, which further adds 0x400, as we require 0x3ff + $leading0s$ at a later stage to calculate the unbiased exponent.

$$
\begin{aligned}
&\mathit{unbiasedExponentSubNorm} &&= \mathrm{0x3ff} + \mathit{fromIntegral\ leading0s}\\
&\mathit{adjustedExponentComplemented} =\\
&\quad \textbf{if } \mathit{expIsZero}\\
&\quad \textbf{then } \mathit{unbiasedExponentSubNorm} + \mathrm{0x400}\\
&\quad \textbf{else } \mathit{exponentComplemented}
\end{aligned}
$$

Calculating *approx1ByInputX* requires another 12-bit adder, then concatenation of the mantissa of $1/c$.

$$
\begin{aligned}
&\mathit{approx1ByInputX} =\\
&\quad (\mathit{adjustedExponentComplemented} + \mathrm{0x3fe} + \mathit{lastBit}) * 2 \uparrow 51\\
&\quad + \mathit{mantissaOneByC}
\end{aligned}
$$

In our table calculation, the unique rotation bit assigned to 0 is 0xc. No other rotation has a value greater or equal to 0xc, hence the implied bit is the result of a ***nand*** operation of leading 2-bits of rotation. We do not need to construct a full $52 + k$ bit fixed point integer for $log_2(c)$. We can use $m + k$ bits to represent it.

$$
\begin{aligned}
&\mathit{impliedOneBit} &&= \textbf{xor}\ 1\ \$\ \mathit{bits}\ 3\ 4\ \mathit{rotation}\ .\&.\ \mathit{bits}\ 2\ 3\ \mathit{rotation}\\
&\mathit{mantissaMp1Bits} &&= \mathit{mantissaLog2C} + (2 \uparrow m) * \mathit{impliedOneBit}
\end{aligned}
$$

Furthermore, we only require $m + k + 1$ bits to represent $log_2(c)$ as a fixed point integer, hence *fixedLog2C* is not padded with extra zeros at the end.

$$
\mathit{fixedLog2C} = \mathit{shiftL\ mantissaMp1Bits}\ (\mathit{fromIntegral\ rotation})
$$

We can use the same operation to get the sign of the unbiased exponent.

$$
\begin{aligned}
&\mathit{exponentP1} &&= \mathit{exponent} + 1\\
&\mathit{npSwitch}\ n\ p &&= \textbf{if } \mathit{bits}\ 10\ 11\ \mathit{exponentP1} \equiv 1\\
& && \quad \textbf{then } p\\
& && \quad \textbf{else } n
\end{aligned}
$$

We have already calculated the unbiased exponent for subnormal inputs. For other inputs, the operations remain the same.

$$
\begin{aligned}
&\mathit{unbiasedExponent} =\\
&\quad \textbf{if } \mathit{expIsZero}\\
&\quad \textbf{then } \mathit{unbiasedExponentSubNorm}
\end{aligned}
$$

$$\begin{aligned}&\textbf{else } \textit{bits } 0\ 10\\ &\quad \$\ \textit{npSwitch exponentComplemented exponentP1}\end{aligned}$$

We convert the adjusted exponent to fixed-point representation by shifting the *unbiased* exponent by $k + m$.

$$\textit{eShifted} = \textit{shiftL unbiasedExponent } (k + m)$$

We add or subtract *fixedLog2C* from a fixed point absolute exponent. In hardware implementation, it is a switch between *fixedLog2C* or its 2's-complement.

$$\begin{aligned}\textit{npfixedLog2C} &= \textit{npSwitch } (1 + \textbf{xor } \textit{fixedLog2C } (2 \uparrow (k + m + 11) - 1))\\ &\quad \textit{fixedLog2C}\\ \textit{ePlog2CInt} &= \textit{bits } 0\ (k + m + 11)\ (\textit{eShifted} + \textit{npfixedLog2C})\end{aligned}$$

We have calculated the log_2c table value with only 30 non-zero leading bits and we need to round shift the fixed point *ePlog2CInt* by only $10+k-$ *leadingZeros*, where $k = 11$, the maximum rotation we need is $10 + k - 0 = 21$, and we have 22-bits to pad to the left. We therefore need to shift *ePlog2CInt* left by *leadingZeros* $+1$. Since *leadingZeros* are bounded above by 22, we only require a 5-bit adder and a 23-bit shift left operation. The choice of the table also guarantees the accuracy of *ePlog2CInt*, which now does not require any rounding.

$$\begin{aligned}(\textit{isZero}, \textit{leadingZeros}) &= \textit{countLeadingZeros } (m + 11 + k)\ (11 + k)\\ &\quad \textit{ePlog2CInt}\\ \textit{mantissaEplog2C} &= \textbf{if } \textit{isZero}\\ &\quad \textbf{then } 0\\ &\quad \textbf{else } \textit{bits } 0\ 52\\ &\quad\ \$\ \textit{shiftL ePlog2CInt}\\ &\qquad (\textit{fromIntegral } \$\ \textit{leadingZeros} + 1)\end{aligned}$$

The unbiased exponent of the floating point representation of *ePlog2CInt* is the number of bits which fell off on right plus k, since we multiplied by 2^{52+k}. We add the bias 0x3ff to the unbiased exponent. This step requires an 11-bit adder.

$$\begin{aligned}\textit{expEplog2C} = &\ \textbf{if } \textit{isZero}\\ &\ \textbf{then } 0\\ &\ \textbf{else } \text{0x3ff} + (m + k + 11 - \textit{leadingZeros}) - (m + 1) - k\end{aligned}$$

We concatenate the individual parts of *ePlog2CInt* together.

$$\begin{aligned}
\mathit{approxLog2Input} &= (\mathit{npSwitch}\ 1\ 0) * 2 \uparrow 63 \\
&+ (\mathit{fromIntegral}\ \mathit{expEplog2C}) * 2 \uparrow 52 \\
&+ \mathit{mantissaEplog2C}
\end{aligned}$$

Finally, we check for input values requiring special treatment (0, ∞, NaN, ≤ 0), and produce special results.

$$\begin{aligned}
&(\mathit{fstLookup}, \mathit{sndLookup}) = \textbf{case}\ (\mathit{sign}, \mathit{exponent}, \mathit{mantissa})\ \textbf{of} \\
&\qquad (0, 0, 0) \qquad\quad\ \longrightarrow (0, \mathrm{0xfff0000000000000}) \\
&\qquad (0, \mathrm{0x7ff}, 0) \qquad \longrightarrow (0, \mathrm{0x7ff0000000000000}) \\
&\qquad (0, \mathrm{0x7ff}, _) \qquad \longrightarrow (0, \mathrm{0x7ff8000000000000}) \\
&\qquad (1, _, _) \qquad\quad\ \longrightarrow (0, \mathrm{0x7ff8000000000000}) \\
&\qquad _ \qquad\qquad\qquad \longrightarrow (\mathit{approx1ByInputX}, \mathit{approxLog2Input})
\end{aligned}$$

Chapter 9

Reciprocal Family Functions

This chapter explains the software implementation and the supporting hardware lookup instructions for the reciprocal and divide functions. The multiplicative reduction accurate table method is used for evaluation of these functions. Since the multiplicative reduction factor and the value of the reciprocal function for multiplicative reduction factor are equal, any of the IEEE values in the interval can be used for the construction of accurate table. In particular, the values used for square root, or reciprocal square root could be used, to reduce the total number of tables required.

9.1 Reciprocal Family Software Implementation

module *RecipSoft*
where

This function generates instructions to evaluate either divide or reciprocal. They both share the same two lookup instructions and extended multiply-add instruction, but some of the instructions around them change.

$recipFamily :: PowerType\ a \Rightarrow Bool \rightarrow VR\ a \rightarrow VR\ a \rightarrow VR\ a$
$recipFamily\ isDiv\ denom\ num = result$
where

The first lookup instruction is for the multiplicative reduction factor, chosen such that

$$multReduc \cdot denom \in [1 - 1/(\text{table size}), 1 + 1/(\text{table size})],$$

except in exceptional cases, and in some cases with outputs close to ∞ which would otherwise saturate prematurely. The second lookup instruction is an approximation of $1/denom$ in the *recip* case, and in the *div* it also includes an exponent adjustment to properly account for subnormal inputs. It is an extended floating point value, and contains special values in exceptional cases.

$$\begin{aligned} [\mathit{multReduc}, \mathit{approxRecip}] \quad &= \mathit{map} \\ &\quad (\textbf{\textit{lookupOpcode}}\ \mathit{denom}\ \mathit{num} \\ &\quad \$\ \textbf{if}\ \mathit{isDiv}\ \textbf{then}\ 3\ \textbf{else}\ 2)\ [1,2] \end{aligned}$$

We use a ***dfmaX*** instruction for the multiplicative reduction to allow extended range values of *multReduc*.

$$f = \frac{1}{C_{lookup}} * denom - 1 \tag{9.1}$$

$$\mathit{fracMoffset} = \textbf{dfmaX}\ \mathit{multReduc}\ \mathit{denom}\ (\mathrm{undoubles2}\ (-1))$$

Both *recip* and *div* use the same higher-order coefficients in the minimax polynomial, and we use Horner evaluation.

$$\mathit{innerPoly} = \mathit{hornerVDbl}\ \mathit{divRecipCoeffs}\ \mathit{fracMoffset}$$

The final combination differs in the two cases. The following two equations govern the

$$\begin{aligned} \frac{1}{denom} &= \left(\frac{1}{C_{lookup}} * poly(f)\right) * f + \frac{1}{C_{lookup}} \\ \frac{num}{denom} &= \left(\left(num * \frac{1}{C_{lookup}}\right) * poly(f)\right) * f + \left(num * \frac{1}{C_{lookup}}\right) \end{aligned}$$

where *innerPoly* is the evaluation of minimax polynomial *poly*, which approximates

$$\frac{\frac{1}{1+f} - 1}{f}$$

in range $f \in [-2^{-n}, 2^{-n}]$.

$$\mathit{result} = \ \textbf{if}\ \mathit{isDiv}$$

In the *div* case, we combine the numerator with the approximate reciprocal of the denominator.

then let *oneByCXnum* = ***dfmaX*** *approxRecip num*
(*undoubles2* 0)
in ***dfma*** (***dfm*** *oneByCXnum innerPoly*)
fracMoffset oneByCXnum

In the *recip* case, We use ***dfmaX*** instead of ***dfm*** because in the cases where $1/denom$ is saturating to infinity, the multiplication by *innerPoly* can change the sign, resulting in formation of *NaN* as output. Those cases are eliminated by returning 0 as first lookup value *multReduc*. As a result, ***dfmaX*** instruction returns 0 and the correct reciprocal is returned by second lookup value *approxRecip*.

else ***dfma*** (***dfmaX*** *multReduc innerPoly* (*undoubles2* 0))
fracMoffset approxRecip

9.2 Recip Lookup Instruction

module *RecipLookup*
where

This lookup instruction returns different lookup values for *div* and *recip*. In both cases, the first lookup value is extended range multiplicative reduction factor $2^{-e}/c$. The second lookup, in the case of *recip*, is the value of the multiplicative reduction factor in IEEE representation. In the case of *div*, we pass the same extended range value, but it is modified to handle the exceptional cases because we know the second lookup value is multiplied by numerator *num*, before being used with the evaluation of polynomial *evalPoly*. We use ***dfmaX*** to get the IEEE representation of the product. Different treatments are needed for the outputs of the *div* function at both ends, i.e. a very big output and a very small output. For very big outputs, we need to use the smallest value $1/c$ in the intervals containing the input. This prevents $2^{-e}/c$ from saturating prematurely to *Infinity*, which would produce *NaN* as the output of the final calculation. In the case of very small outputs, subnormals, we need to choose the largest value of $1/c$, because we do not want to lose accuracy because of dropped of bits, premature saturation to 0, or subnormals. In the case of *recip*, we choose the smallest value of $1/c$ for all intervals, because we know the smallest subnormal output 2^{-1024} will only be dropping 2-bits at most, and in our table calculations these bits are 0s.

$$
\begin{aligned}
&\mathit{recipLookup} :: \mathit{Bool} \to \mathit{Integer} \to \mathit{Integer} \to (\mathit{Integer}, \mathit{Integer})\\
&\mathit{recipLookup}\ \mathit{isDiv}\ \mathit{denom}\ \mathit{num} = \textbf{if}\ \mathit{isDiv}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \mathit{divLookup}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{recipLookup}\\
&\quad \textbf{where}
\end{aligned}
$$

The table we are using for this implementation has 2^n+1 double floating point numbers. An additional 1 double is required because of the aligned lookup for the different inputs of *div*. Since this number at either end is 1 or 0.5, it does not need to be stored, but can be constructed in the hardware.

$$
\begin{aligned}
&n = 12\\
&\mathit{table} = \mathit{recipTable}\ n
\end{aligned}
$$

Boost the denorm into normal range by multiplying by 2^{52}. This part of the calculation is mathematically the same way as for other multiplicative reduction methods, like used in *logLookup*.

$$
\begin{aligned}
&(\mathit{sign}, \mathit{rest}) \;=\; \mathit{divMod}\ \mathit{denom}\ (2 \uparrow 63)\\
&(\mathit{exponent}, \mathit{mantissa}) \qquad = \mathit{divMod}\ \mathit{rest}\ (2 \uparrow 52)\\
&(\_, \mathit{leading0s}) = \mathit{countLeadingZeros}\ 52\ 52\ \mathit{mantissa}\\
&(\mathit{adjustedExponent}, \mathit{adjustedMantissa}) = \textbf{case}\ \mathit{exponent}\ \textbf{of}\\
&\qquad\qquad 0 \to (52 - \mathit{fromIntegral}\ \mathit{leading0s},\\
&\qquad\qquad\qquad \mathit{mod}\ (\mathit{shiftL}\ \mathit{mantissa}\ (\mathit{leading0s} + 1))\ (2 \uparrow 52))\\
&\qquad\qquad \_ \to (\mathit{exponent}, \mathit{mantissa})
\end{aligned}
$$

We check whether the output is going to be large or small in the case of *div*. We can approximate it by subtracting the leading 2-bits of exponent of the denominator from the leading 2-bits of exponent of the numerator. The special treatment is only needed for very large outputs near infinity or very small outputs, sub-normals. In the rest of the range, we do not care which lookup value we get. We also do not care exactly where this approximate calculation starts being true, as long as it is true for very small outputs.

$$\mathit{isSmall} = \qquad (\mathit{bits}\ 61\ 63\ \mathit{num}) + \textbf{xor}\ 0x3\ (\mathit{bits}\ 61\ 63\ \mathit{denom}) \leqslant 0x3$$

Lookup is the leading n-bits of the *adjustedMantissa*, aligned by 1.

$$
\begin{aligned}
&\mathit{lookupKey} = \textbf{if } \mathit{isDiv} \wedge \mathit{isSmall} \\
&\qquad\qquad\quad \textbf{then } \mathit{bits}\ (52 - n)\ 52\ \mathit{adjustedMantissa} \\
&\qquad\qquad\quad \textbf{else } \mathit{bits}\ (52 - n)\ 52\ \mathit{adjustedMantissa} + 1 \\
&(\mathit{lastBitRecipC}, \mathit{mantissaRecipC}) = \mathit{table}\ !!\ (\mathit{fromIntegral}\ \mathit{lookupKey})
\end{aligned}
$$

Construction of the extended range multiplicative reduction factor is the same way like all the *logLookup* instruction.

$$
\begin{aligned}
\mathit{exponentComplemented} &= \textbf{xor } \mathit{adjustedExponent}\ \mathrm{0x7ff} \\
\mathit{exponentFstLookup} &= \mathit{exponentComplemented} + \mathrm{0x3fe} \\
&\quad + \mathit{lastBitRecipC}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fstLookup} = &\ \textbf{case } \mathit{exponent} \textbf{ of} \\
&\ 0 \longrightarrow \mathit{sign} * (2 \uparrow 63) + (\mathit{exponentFstLookup} + 52) * (2 \uparrow 51) \\
&\qquad + \mathit{mantissaRecipC} * (2 \uparrow (51 - n)) \\
&\ \_ \longrightarrow \mathit{sign} * (2 \uparrow 63) + \mathit{exponentFstLookup} * (2 \uparrow 51) \\
&\qquad + \mathit{mantissaRecipC} * (2 \uparrow (51 - n))
\end{aligned}
$$

In case of *div*, the same extended range value is returned as the second lookup value. However, lookup values are overridden in the case of special outputs.

$$
\begin{aligned}
&\mathit{divLookup} \\
&\quad \mid \mathit{isNan}\ \mathit{denom} \vee \mathit{isNan}\ \mathit{num} &&= (0, \mathit{nan0X}) \\
&\quad \mid \mathit{denom} \equiv 0 \ \wedge\ \mathit{num} \equiv 0 &&= (0, \mathit{nan0X}) \\
&\quad \mid \mathit{bits}\ 52\ 63\ \mathit{denom} \equiv \mathrm{0x7ff} && \\
&\quad\ \ \wedge\ \mathit{bits}\ 52\ 63\ \mathit{num} \equiv \mathrm{0x7ff} &&= (0, \mathit{nan0X}) \\
&\quad \mid \mathit{bits}\ 52\ 63\ \mathit{denom} \equiv \mathrm{0x7ff} &&= (\mathit{nan3X}, \mathit{nan3X}) \\
&\quad \mid \mathit{denom} \equiv 0 &&= (\mathit{nan1X} \\
&&&\quad , \mathit{signResult} + \mathit{infinityX}) \\
&\quad \mid \mathit{bits}\ 52\ 63\ \mathit{num} \equiv \mathrm{0x7ff} &&= (\mathit{nan1X} \\
&&&\quad , \mathit{signResult} + \mathit{infinityX}) \\
&\quad \mid \mathit{otherwise} &&= (\mathit{fstLookup}, \mathit{fstLookup}) \\
&\mathit{signResult} = (2 \uparrow 63) * (\textbf{xor}\ (\mathit{bits}\ 63\ 64\ \mathit{num})\ (\mathit{bits}\ 63\ 64\ \mathit{denom}))
\end{aligned}
$$

There are four intervals, two at each end, where we need a special function for the second reciprocal lookup. For the inputs with exponents 0x7fe and 0x7fd, we need to construct the subnormals, which requires at most a 2-bit shift. We also need to construct the outputs which correspond to those input subnormal ranges. We know, for the case of *recip*, we always choose the smallest $1/c \in [0.5, 1)$. Hence, the last bit of exponent of the lookup is always 0, and is not included in the calculations. For all other inputs, the IEEE floating point for reduction factor can be constructed using *exponentComplemented* $= -e + 1$.

$$
\begin{array}{lll}
\mathit{recipLookup} = & \multicolumn{2}{l}{\textbf{case}\ (\mathit{exponent}, \mathit{fstBitMantissa}, \mathit{sndBitMantissa})\ \textbf{of}} \\
 & (0, 1, \_) & \rightarrow (\mathit{fstLookup}, \mathit{sign} * (2 \uparrow 63) + \mathrm{0x7fd} * (2 \uparrow 52) \\
 & & \quad + \mathit{mantissaRecipC} * (2 \uparrow (52 - n))) \\
 & (0, 0, 1) & \rightarrow (\mathit{fstLookup}, \mathit{sign} * (2 \uparrow 63) + \mathrm{0x7fe} * (2 \uparrow 52) \\
 & & \quad + \mathit{mantissaRecipC} * (2 \uparrow (52 - n))) \\
 & (0, 0, 0) & \rightarrow (\mathit{nan3X}, \mathit{sign} * (2 \uparrow 63) + \mathrm{0x7ff} * (2 \uparrow 52)) \\
 & (\mathrm{0x7fd}, \_, \_) & \rightarrow (\mathit{fstLookup}, \mathit{sign} * (2 \uparrow 63) \\
 & & \quad + \mathit{shiftR}\ (((2 \uparrow n) \\
 & & \quad + \mathit{mantissaRecipC}) * (2 \uparrow (52 - n)))\ 1) \\
 & (\mathrm{0x7fe}, \_, \_) & \rightarrow (\mathit{fstLookup}, \mathit{sign} * (2 \uparrow 63) \\
 & & \quad + \mathit{shiftR}\ (((2 \uparrow n) + \mathit{mantissaRecipC}) \\
 & & \quad * (2 \uparrow (52 - n)))\ 2) \\
 & (\mathrm{0x7ff}, \_, \_) & \rightarrow \textbf{if}\ \mathit{mantissa} \equiv 0 \\
 & & \textbf{then}\ (\mathit{nan3X}, 0) \\
 & & \textbf{else}\ (0, \mathit{infinityX}) \\
 & \_ & \rightarrow (\mathit{fstLookup}, \mathit{sign} * (2 \uparrow 63) \\
 & & \quad + (\mathit{exponentComplemented} - 2) * (2 \uparrow 52) \\
 & & \quad + \mathit{mantissaRecipC} * (2 \uparrow (52 - n)))
\end{array}
$$

$$
\begin{array}{ll}
\mathit{fstBitMantissa} & = \mathit{bits}\ 51\ 52\ \mathit{mantissa} \\
\mathit{sndBitMantissa} & = \mathit{bits}\ 50\ 51\ \mathit{mantissa}
\end{array}
$$

Chapter 10

Square-Root Family Functions

Square root and reciprocal square root functions are calculated similarly to reciprocal function above. The lookup instruction is used to get the multiplicative reduction factor and the corresponding value of the function. The following equations are used to decompose the input and combine the individual values together.

$$\sqrt{v} = \left(2^{\lfloor e/2 \rfloor} * 2^{c_0 + e - \lfloor e/2 \rfloor}\right)_{lookup} * \sqrt{1+f}, \tag{10.1}$$

where f is the fractional part left over after the multiplicative range reduction

$$f = \left(\frac{1}{c}\right)_{lookup} * v - 1.$$

10.1 Square Root Software Implementation

> **module** *SqrtSoft*
>   **where**

> *rsqrtFamily* :: *PowerType a* ⇒ *Bool* → *VR a* → *VR a*
> *rsqrtFamily isRecip v* = *result*
>   **where**

We use the appropriate lookup instructions to get the multiplicative reduction factor and corresponding value of the function.

> [*oneByC*, *sqrtC0*] = *map*
>   (***lookupOpcode*** *v v* $

if *isRecip* **then** 4 **else** 5) [1, 2]
debug = **lookupOpcodeDebug** *v v*
$ **if** *isRecip* **then** 4 **else** 5

We perform the multiplicative range reduction using the first lookup value.

fracMoffset = **dfmaX** *oneByC v* (*undoubles2* (−1))

A minimax polynomial approximating

$$poly(f) = \frac{\sqrt{1+f}-1}{f}$$

is used to calculate the value of the function in the reduced range. The following Maple code is used to calculate the minimax polynomial:

```
Digits := 100;
numSegments := 2^(11);
polyOrd := 3;
b := 1/numSegments;
plog2:=numapprox[minimax](x->limit(sqrt(1+y)/y,y=x)
        ,-b..b,[polyOrd,0],1,'erSqrt');
log[2](erSqrt);
lprint([seq(roundDbl(coeff(plog2(x),x,j)),j=0..polyOrd)]);
```

The higher order polynomial is evaluated using Horner's scheme.

evalPoly = *hornerVDbl coeffs fracMoffset*

Finally, the polynomial evaluation is merged with the second lookup, the value of the function corresponding to the multiplicative reduction factor.

result = **dfma** (**dfm** *sqrtC0 evalPoly*) *fracMoffset sqrtC0*

10.2 Sqrt Lookup Instruction

module *SqrtLookup*
  where

$$
\begin{aligned}
&\mathit{sqrtLookup} :: \mathit{Bool} \rightarrow \mathit{Integer} \rightarrow (\mathit{Integer}, \mathit{Integer}) \\
&\mathit{sqrtLookup}\ \mathit{isRecip}\ v = (\mathit{fstLookup}, \textbf{if}\ \mathit{isRecip} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{rsqrtSndLookup} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{sqrtSndLookup}) \\
&\quad \textbf{where}
\end{aligned}
$$

We have calculated the accurate table with 2^n intervals, dividing the range $[0, 1)$. For each interval we have calculated two values of *sqrt*, one with an additional $\sqrt{2}$ factor and the other without it, corresponding to whether the input has an odd or even unbiased exponent.

$$n = 11$$

We boost the subnormal inputs into normal range by multiplying 2^{52}.

$$
\begin{aligned}
&(\mathit{sign}, \mathit{rest}) = \quad \mathit{divMod}\ v\ (2 \uparrow 63) \\
&(\mathit{exponent}, \mathit{mantissa}) \qquad\quad = \mathit{divMod}\ \mathit{rest}\ (2 \uparrow 52) \\
&(\_, \mathit{leading0s}) = \mathit{countLeadingZeros}\ 52\ 52\ \mathit{mantissa} \\
&(\mathit{adjustedExponent}, \mathit{adjustedMantissa}) = \textbf{case}\ \mathit{exponent}\ \textbf{of} \\
&\quad 0 \rightarrow (52 - \mathit{fromIntegral}\ \mathit{leading0s}, \\
&\qquad\quad \mathit{mod}\ (\mathit{shiftL}\ \mathit{mantissa}\ (\mathit{leading0s} + 1))\ (2 \uparrow 52)) \\
&\quad \_ \rightarrow (\mathit{exponent}, \mathit{mantissa})
\end{aligned}
$$

lookupKey is constructed using the leading n-bits of *adjustedMantissa* and the last bit of *adjustedExponent*, which tells us whether the unbiased $_{adjustedExponent}$ is even or odd.

$$
\begin{aligned}
&\mathit{lookupKey} \qquad\qquad = \mathit{bits}\ 0\ 1\ \mathit{adjustedExponent} * (2 \uparrow n) \\
&\qquad\qquad\qquad\qquad\quad + \mathit{bits}\ (52 - n)\ 52\ \mathit{adjustedMantissa} \\
&[\mathit{sqrtC}, \mathit{oneByC}] = (\textbf{if}\ \mathit{isRecip} \\
&\qquad\qquad\qquad\quad \textbf{then}\ \mathit{rsqrtTable} \\
&\qquad\qquad\qquad\quad \textbf{else}\ \mathit{sqrtTable})\ !!\ \mathit{fromIntegral}\ \mathit{lookupKey}
\end{aligned}
$$

We take the complement of the exponent to get $-e + 1 +$ 0x3ff.

$$
\begin{aligned}
&\mathit{exponentComplemented} = \textbf{xor}\ \mathit{adjustedExponent}\ \mathrm{0x7ff} \\
&\mathit{exponentFstLookup} \quad\ = \mathit{exponentComplemented} \\
&\qquad\qquad\qquad\qquad\quad + \mathit{bits}\ 52\ 63\ \mathit{oneByC} \\
&\mathit{mantissaFstLookup} \quad\ = \mathit{bits}\ 1\ 52\ \mathit{oneByC}
\end{aligned}
$$

The multiplicative reduction factor is then constructed using the exponent complement and table value.

$$
\begin{aligned}
\mathit{fstLookup} \quad & = \mathbf{case}\ \mathit{exponent}\ \mathbf{of} \\
& \quad 0 \longrightarrow \mathit{sign} * (2 \uparrow 63) \\
& \qquad\quad + (\mathit{exponentFstLookup} + 52) * (2 \uparrow 51) \\
& \qquad\quad + \mathit{mantissaFstLookup} \\
& \quad \_ \longrightarrow \mathit{sign} * (2 \uparrow 63) \\
& \qquad\quad + \mathit{exponentFstLookup} * (2 \uparrow 51) \\
& \qquad\quad + \mathit{mantissaFstLookup}
\end{aligned}
$$

Dividing the exponent by 2 is same as shifting the exponent right by 1 bit and letting the last bit drop off. We have taken care of the dropped off bit with the lookup value.

$$\mathit{adjustedExponentDiv2} \quad = \mathit{shiftR}\ \mathit{adjustedExponent}\ 1$$

Since we have shifted the biased exponent, the bias also gets divided by 2. To fix the bias again, we add a constant $511 = div 0x3ff2$. In the case of subnormal inputs, an additional factor used for the boost needs to be included.

$$
\begin{aligned}
& \mathit{sqrtExpMultC} \\
& \quad \mid \mathit{exponent} \equiv 0 && = \mathit{adjustedExponentDiv2} * (2 \uparrow 52) \\
& && \quad + \mathit{sqrtC} - (511 * 2 \uparrow (52)) - 26 * 2 \uparrow (52) \\
& \quad \mid \mathit{otherwise} && = \mathit{adjustedExponentDiv2} * (2 \uparrow 52) \\
& && \quad + \mathit{sqrtC} - (511 * 2 \uparrow (52))
\end{aligned}
$$

Finally, we override the second lookup value for special cases.

$$
\begin{aligned}
& \mathit{sqrtSndLookup} \\
& \quad \mid \mathit{isNan}\ v && = \mathrm{0x7ff8000000000000} \\
& \quad \mid \mathit{div}\ v\ (2 \uparrow 63) \equiv 1 && = \mathrm{0x7ff8000000000000} \\
& \quad \mid \mathit{div}\ v\ (2 \uparrow 52) \equiv \mathrm{0x7ff} && = \mathrm{0x7ff0000000000000} \\
& \quad \mid v \equiv 0 && = 0 \\
& \quad \mid \mathit{otherwise} && = \mathit{sqrtExpMultC}
\end{aligned}
$$

Similar adjustments need to be done for reciprocal square root, where the complement of *adjustedExponent* is used to construct the value of the reciprocal square root function.

$$
\begin{aligned}
& \mathit{adjustedExpComplDiv2} = \mathit{shiftR}\ (\mathbf{xor}\ \mathrm{0x7ff}\ \mathit{adjustedExponent})\ 1 \\
& \mathit{rsqrtExpMultC} \\
& \quad \mid \mathit{exponent} \equiv 0 && = \mathit{adjustedExpComplDiv2} * (2 \uparrow 52)
\end{aligned}
$$

$$
\begin{array}{ll}
 & + \mathit{sqrtC} - (512 * 2 \uparrow (52)) + 26 * 2 \uparrow (52) \\
\mid \mathit{otherwise} & = \mathit{adjustedExpComplDiv2} * (2 \uparrow 52) \\
 & + \mathit{sqrtC} - (512 * 2 \uparrow (52))
\end{array}
$$

Finally, we look for special inputs and override the second lookup with the correct function values.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{rsqrtSndLookup}} \\
\mid \mathit{isNan}\ v & = \mathrm{0x7ff8000000000000} \\
\mid \mathit{div}\ v\ (2 \uparrow 63) \equiv 1 & = \mathrm{0x7ff8000000000000} \\
\mid \mathit{div}\ v\ (2 \uparrow 52) \equiv \mathrm{0x7ff} & = 0 \\
\mid v \equiv 0 & = \mathrm{0x7ff0000000000000} \\
\mid \mathit{otherwise} & = \mathit{rsqrtExpMultC}
\end{array}
$$

Chapter 11

Exponential Family Functions

Exponential family functions are calculated using base 2. Inputs are scaled accordingly. After scaling, we decompose the input into the integer part $[x]$ and the fractional part, $x - [x]$. The fractional part is further decomposed using accurate table values c and 2^c, and we multiply the individual powers of two to get the result

$$
\begin{aligned}
2^x &= 2^{[x]} * 2^{x-[x]} \\
&= 2^{[x]} * 2^c * 2^{x-[x]-c}
\end{aligned}
$$

11.1 Exp Software Implementation

module *ExpSoft*
  where

expFamily :: *PowerType a* ⇒ *MathOptions* → *Bool*
  → *VR a* → *VR a*
expFamily (*MathOptions base _exceptions*) *isM1Case v* = *result*
  where

We scale the input according the the base of the exponential to put the scaled integer value into the high-order 11 bits into the mantissa. These bits in the exponent bit field will return $2^{[x]}$.

$$
\begin{array}{rll}
\textit{vScaledOffset} = & \textbf{case}\ \textit{base}\ \textbf{of} & \\
& \textit{MO2} & \rightarrow \textbf{dfa}\ v\ \textit{offsetBump} \\
& \textit{MOe} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2}\ \textit{offsetBump} \\
& \textit{MOeX2} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2X2}\ \textit{offsetBump} \\
& \textit{MO2m1} & \rightarrow \textbf{dfnms}\ v\ (\text{undoubles2}\ 1)\ \textit{offsetM1Bump} \\
& \textit{MO2p1} & \rightarrow \textbf{dfa}\ v\ \textit{offsetM1Bump} \\
& \textit{MOem1} & \rightarrow \textbf{dfnms}\ v\ \textit{invLog2}\ \textit{offsetM1Bump} \\
& \textit{MOep1} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2}\ \textit{offsetM1Bump} \\
& \textit{MO10} & \rightarrow \textbf{dfma}\ v\ \textit{log10InvLog2}\ \textit{offset}
\end{array}
$$

Base *MOeX2* represents the exponential function e^{2x}, *MO2m1* represents 2^{-x-1} and *MO2p1* represents 2^{x-1}. These functions are used in the calculation of hyperbolic functions and other functions not in the standard library.

Now we construct the reduction to $(-0.5, 0.5)$ in two steps:

First, truncate the pseudo fixed-point version of the input to the most significant fractional bits, using the odd instruction ***shufb***.

$$
\begin{array}{l}
\textit{v3Bytes} = \ \underline{\text{shufB1}}\ \textit{vScaledOffset} \\
\quad [0, 1, 2, \textit{shufb0x00}\ \ , \textit{shufb0x00}, \textit{shufb0x00}, \textit{shufb0x00}, \textit{shufb0x00} \\
\quad , 8, 9, 10, \textit{shufb0x00}, \textit{shufb0x00}, \textit{shufb0x00}, \textit{shufb0x00}, \textit{shufb0x00}]
\end{array}
$$

Next subtract a modified fixed offset, to get a truncated version of the input with 0.5 added. Now subtract this from the original input, to get the low-order part of the fraction. This operation is exact, because the resulting exponent is not larger than the exponent of the input. To improve rounding during range reduction, we use high and low-order double values,

$$
\begin{array}{lll}
\textit{frac} = & \textbf{case}\ \textit{base}\ \textbf{of} & \\
& \textit{MO2} & \rightarrow \textbf{dfs}\ v\ (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offset}) \\
& \textit{MOe} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2Low} \\
& & \quad (\textbf{dfms}\ v\ \textit{invLog2}\ (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offset})) \\
& \textit{MOeX2} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2LowX2} \\
& & \quad (\textbf{dfms}\ v\ \textit{invLog2X2}\ (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offset})) \\
& \textit{MO2m1} & \rightarrow \textbf{dfnma}\ v\ (\text{undoubles2}\ 1) \\
& & \quad (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offsetM1}) \\
& \textit{MO2p1} & \rightarrow \textbf{dfs}\ v\ (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offsetM1}) \\
& \textit{MOem1} & \rightarrow \textbf{dfma}\ v\ \textit{invLog2Low} \\
& & \quad (\textbf{dfnma}\ v\ \textit{invLog2}\ (\textbf{dfs}\ \textit{v3Bytes}\ \textit{offsetM1}))
\end{array}
$$

$$\begin{aligned} &\mathit{MOep1} &&\rightarrow \textbf{dfma}\ v\ \mathit{invLog2Low} \\ & &&\quad(\textbf{dfms}\ v\ \mathit{invLog2}\ (\textbf{dfs}\ \mathit{v3Bytes}\ \mathit{offsetM1})) \\ &\mathit{MO10} &&\rightarrow \textbf{dfma}\ v\ \mathit{log10InvLog2}\ (\textbf{dfs}\ \mathit{v3Bytes}\ \mathit{offset}) \end{aligned}$$

Then, table values are looked up using the *vScaledOffset*. The range reduction factor *c0* is in the same interval as *frac*. The corresponding value of function *exp2c0*, also includes the integer part $= 2^{[x]+c}$.

$$\begin{aligned} &[\mathit{c0}, \mathit{exp2c0}] = \mathit{map}\ (\textbf{lookupOpcode}\ \mathit{vScaledOffset}\ \mathit{vScaledOffset}\ 1) \\ &\qquad\qquad\qquad\qquad [1, 2] \\ &\mathit{debug} = \quad \textbf{lookupOpcodeDebug}\ \mathit{vScaledOffset}\ \mathit{vScaledOffset}\ 1 \end{aligned}$$

Now, we reduce the range using first lookup value.

$$\mathit{fracMOffset} = \textbf{dfmaX}\ (\mathit{undwrds2}\ \text{0x3ff8000000000000})\ \mathit{frac}\ \mathit{c0}$$

$2^{x-[x]-c}$ is calculated using a minimax polynomial evaluated by Maple. The following Maple code calculates the minimax polynomial.

```
Digits := 50;
pExp := numapprox[minimax](x -> 2^x, -1/256 .. 1/256
                           ,[4, 0], 1, 'er');
log[2](er);
lprint([seq(roundDbl(coeff(pExp, x, j)), j = 0 .. 4)]);
```

In case of $2^x - 1$ (*i.e.*, *isM1Case* $\equiv$ *True*)

```
pExp := numapprox[minimax](x -> limit((2^y-1)/y,y = x)
                           ,-1/256 .. 1/256, [4, 0], 1, 'er');
log[2](er);
lprint([seq(roundDbl(coeff(pExp, x, j)), j = 0 .. 4)]);
```

$$\begin{aligned} &\mathit{evalPoly}' = \mathit{hornerVDbl}\ \mathit{coeffs}\ \mathit{fracMOffset} \\ &\mathit{evalPoly} = \textbf{if}\ \mathit{isM1Case}\ \textbf{then}\ \textbf{dfm}\ \mathit{evalPoly}'\ \mathit{fracMOffset} \\ &\qquad\qquad\quad \textbf{else}\ \mathit{evalPoly}' \end{aligned}$$

The resultant polynomial evaluation is multiplied with the second lookup value to generate the result.

$$\begin{aligned} &\mathit{result} = \textbf{if}\ \mathit{isM1Case}\ \textbf{then}\ \textbf{dfma}\ \mathit{evalPoly}\ \mathit{exp2c0} \\ &\quad \$\ \textbf{dfs}\ \mathit{exp2c0}\ (\text{undoubles2}\ 1) \\ &\quad \textbf{else}\ \textbf{dfm}\ \mathit{evalPoly}\ \mathit{exp2c0} \end{aligned}$$

11.2 Exp Lookup Instruction

module *ExpLookup*
where

We put the integral part of input $[x]$ into the desired bit position in the mantissa by adding a power to 2, bumped by 0.5 for rounding. In our software implementation of exponential family, we scaled the integer part of the input into the high-order 11 bits of the mantissa. An additional 1023 is already added into integer part to account for biasing. This offset number is fed to the *expLookup* instruction which, based on table lookup, generates *c0* such that $fracMc0 = x - [x] - c_0 \in [-2^{-N}, 2^{-N}]$, and $2^{[x]+c0}$.

$expLookup :: Integer \longrightarrow (Integer, Integer)$
$expLookup\ xOffset = (fstLookup, sndLookup)$
where

We have calculated the table of size 2^n, such that it divides the range $[-0.5, 0.5]$ into 2^n segments.

$(n, table) \qquad = (8, exp2Table)$

The lookup key is the n-bits followed by the integer part of the input.

$lookupKey \qquad = bits\ (40 - n)\ 40\ xOffset$

We know the first table value, representing 2^{c0}, belongs in the range $[\sqrt{1/2}, \sqrt{2}]$. We only need to store the last bit of its exponent and mantissa bits. Similar modifications could be made to $c0$, based on the smallest table value other than 0, to reduce the size of table. This requires decomposing, storing, and composing in the hardware, which is not of particular interest as far as algorithm is concerned.

$(lastBitExpC0, mantissaC0, c0) = table\ !!\ fromIntegral\ lookupKey$

expBits represents the value $[x]+$ 0x3ff.

$expBits = bits\ 40\ 52\ xOffset$

In case the input is > 1021.5, *expBits* represents the the exponent of $2^{[x]}$. Since $\mathit{exp2IpC0} = 2^{[x]+c0} = 2^{[x]} \times 2^{c0}$, multiplication by a power of 2 is just the addition of exponent bits.

$$
\begin{array}{l}
\mathit{exp2IpC0} \\
\quad \mid \mathit{expBits} > \mathrm{0x801} = (\mathit{bits}\ 0\ 11\ \mathit{expBits} - 1 + \mathit{lastBitExpC0}) \\
\qquad\qquad\qquad\qquad\quad * (2 \uparrow 52) + \mathit{mantissaC0}
\end{array}
$$

If the input is less than -1021.5, we need to construct the subnormal numbers for $2^{[x]+c0}$. *mantissaC0* is added to the implied 1 and shifted by the amount of rotation required. The rounded shift is used to minimize the error introduced because of fallen bits.

$$
\begin{array}{ll}
\quad \mid \mathit{otherwise} & = \mathit{roundShiftR}\ (2 \uparrow 52 + \mathit{mantissaC0}) \\
& \quad (\mathit{fromIntegral}\ \$\ \mathrm{0x7ff} - \mathit{expBits} + 2 \\
& \qquad + 1 - \mathit{lastBitExpC0})
\end{array}
$$

Finally, the input is checked for special output values.

$$
\begin{array}{ll}
(\mathit{fstLookup}, \mathit{sndLookup}) & \\
\quad \mid \mathit{isNan}\ \mathit{xOffset} & = (0, \mathrm{0x7ff8000000000000}) \\
\quad \mid \mathit{bits}\ 63\ 64\ \mathit{xOffset} \equiv 1 & = (\mathrm{0x7ff8000000000000}, 0) \\
\quad \mid \mathit{bits}\ 32\ 63\ \mathit{xOffset} < \mathrm{0x40b7cd80} & = (\mathrm{0x7ff8000000000000}, 0) \\
\quad \mid \mathit{bits}\ 32\ 63\ \mathit{xOffset} > \mathrm{0x40bfff80} & = (0, \mathrm{0x7ff0000000000000}) \\
\quad \mid \mathit{otherwise} & = (2 \uparrow 63 + \mathit{c0}, \mathit{exp2IpC0})
\end{array}
$$

Chapter 12

Trigonometric Family Functions

Trigonometric functions are calculated using the following standard identities:

$$sin(\theta + \phi) = sin(\theta)cos(\phi) + cos(\theta)sin(\phi)$$
$$cos(\theta + \phi) = cos(\theta)cos(\phi) - sin(\theta)sin(\phi)$$

The accurate table used for the calculation of trigonometric functions is different from the tables of other functions. In this trigonometric table, we need to have three accurate values, the angle θ, and its sine and cosine values. We have calculated the table such that for each interval in the first half-quadrant, we found two integers such that the sum of their squares is as close to n^2 as possible. These integers are then converted to double floating point representations and stored as $sin(\theta)$ and $cos(\theta)$. Then both high and low parts of θ are calculated using high precision Maple. For the second half-quadrant, we just switch between the sin and cos values; the high and low parts of corresponding angles are stored in the table.

12.1 Trigonometric Functions Software Implementation

module *TrigSoft*
  where

trigFamily :: *PowerType a* $\Rightarrow$ *VR a* $\rightarrow$ (*VR a*, *VR a*, *VR a*)
trigFamily v = (*sin*, *cos*, *tan*)
  where

We mask off the sign bits from the input.

$$absV = \textbf{andc}\ v\ signBitDbl$$

Since we are using 2^7-segment intervals dividing the $[0, pi/2)$ range, we multiply the input by $2^9/(2\pi)$ to convert the angle in radians to an angle in fraction of rotations, where each rotation is divided into 2^9 intervals. We add the *offset* $= 2^{52}$ to put the integral part of the segment number into the last 9-bits of the mantissa bits, where the bits from 9 leftward represent the number of rotations.

$$vScaledOffset = \textbf{dfma}\ scaleHigh\ absV\ offset$$

The fractional part of the segment is dropped off in the calculation of *vScaledOffset*, hence subtracting the *offset* from *vScaledOffset* returns the integer part of the segment.

$$intSegments = \textbf{dfs}\ vScaledOffset\ offset$$

Now we calculate the fractional part of the segment, by subtracting *intSegments* from the segment $= 2^9 v/(2\pi)$ itself. We use both high and low parts of *vOverPiHigh* to get accuracy for the fractional part of the segment.

$$\begin{aligned}
&vOverPiHigh = \textbf{dfm}\ absV\ scaleHigh\\
&vOverPiLow = \textbf{dfms}\ absV\ scaleLow\ vOverPiHigh\\
&vOverPiReducedHigh = \textbf{dfms}\ absV\ scaleHigh\ intSegments\\
&vOverPiReduced \quad = \textbf{dfma}\ absV\ scaleLow\\
&\qquad\qquad\qquad\qquad\quad vOverPiReducedHigh
\end{aligned}$$

Hardware lookup instructions return the high and low parts of the distance of the angle with accurate values (its sine and cosine) from the start of the segment. We use extended precision doubles for sine and cosine values in order to handle the exceptional cases and very large inputs.

$$\begin{aligned}
&[thetaHigh, thetaLow, costheta, sintheta]\\
&\quad = map\ (\textbf{lookupOpcode}\ vScaledOffset\ vScaledOffset\ 6)\ [1, 2, 3, 4]
\end{aligned}$$

We subtract both parts of the *theta* from the fractional part of the segment. We use **dfmaX** in order to get 0 as *fracMOffset* for very large inputs.

$$\begin{aligned}
fracMOffset \quad &= \textbf{dfmaX}\ oneX\\
&\quad (\textbf{dfmaX}\ oneX\ vOverPiReducedHigh\ thetaHigh)\\
&\quad (\textbf{dfma}\ absV\ scaleLow\ thetaLow)
\end{aligned}$$

Since series expansions of both sine and cosine have interleaved powers of the input, we calculate the minimax polynomial in terms of $fracMOffset^2$.

$$fracMOffsetSquared \quad = \mathbf{dfm}\ fracMOffset\ fracMOffset$$

We calculate the minimax polynomials of the following functions to calculate the sine and cosine in the reduced range.

$$polySin(x^2) = minimax\left(\frac{sin(x) - x}{x^3}\right),$$
$$polyCos(x^2) = minimax\left(\frac{cos(x) - 1}{x^2}\right),$$

where x is scaled to represent the fractional part of the segment. The Maple code needed to generate these polynomials is:

```
Digits := 50;
numSegments := 128;
polyOrd := 2;
b := Pi/2 / numSegments;
pSin1:=numapprox[minimax](x->limit((sin(y)-y)/(y^3),y=sqrt(x))
       ,-1.5*b..1.5*b,[polyOrd,0],x->x^2,'erSin');
pSin:=convert(evalf(eval(x + x^3 * pSin1(x^2)
                ,x=2*Pi*x/(2^9))),horner);
log[2](erSin);
lprint([seq(roundDbl(coeff(pSin,x,2*j+1)),j=0..polyOrd+1)]);

polyOrd := 3;
pCos1:=numapprox[minimax](x->limit((cos(y)-1)/(y^2),y=sqrt(x))
       ,-1.5*b..1.5*b,[polyOrd,0],x->x^2,'erCos');
pCos:=convert(evalf(eval(1 + x^2 * pCos1(x^2)
                ,x=x*2*Pi/(2^9))),horner);
log[2](erCos);
lprint([roundDbl(1),seq(roundDbl(coeff(pCos,x,2*j+2))
        ,j=0..polyOrd)]);
```

We calculate the sine and cosine of the leftover value.

$$\begin{aligned} sinInSeg &= \mathbf{dfm}\ fracMOffset \\ &\quad \$\ hornerVDbl\ sinCoeffs\ fracMOffsetSquared \\ cosInSeg &= hornerVDbl\ cosCoeffs\ fracMOffsetSquared \end{aligned}$$

We use the mathematical identities given above to calculate the sine and cosine of the absolute value of the input.

$$\begin{aligned}
absSin &= \textbf{dfmaX}\ sintheta\ cosInSeg \\
&\quad (\textbf{dfmaX}\ costheta\ sinInSeg\ zeroX) \\
cos &= \textbf{dfmaX}\ costheta\ cosInSeg \\
&\quad (\textbf{dfm}\ (\textbf{dfmaX}\ sintheta\ sinInSeg\ zeroX)\ (\mathrm{undoubles2}\ \$\ {-1}))
\end{aligned}$$

In the case of sine, we put back the sign.

$$sin = \textbf{xor}\ absSin\ (PowerType.\textbf{and}\ v\ signBitDbl)$$

We use the ratio of sine and cosine to calculate the tangent of the input.

$$tan = recipFamily\ True\ cos\ sin$$

12.2 Trig Lookup Instruction

module *TrigLookup*
where

The lookup instruction returns the high and the low parts of the difference of the angle with accurate sine and cosine values from the start of the segment. The sine and cosine values are returned in extended precision double representation to handle the exceptional values.

$$\begin{aligned}
&trigLookup :: Integer \longrightarrow (Integer, Integer, Integer, Integer) \\
&trigLookup\ xOffset = (thetaHigh, thetaLow, cos, sin) \\
&\quad \textbf{where}
\end{aligned}$$

The lookup key is constructed using the last 7 bits of the *xOffset*, representing the segment in the quadrant. Bits 7 and 8 represent the quadrant in which the input lies.

$$\begin{aligned}
lookupKey &= bits\ 0\ 7\ xOffset \\
quadrant &= bits\ 7\ 9\ xOffset
\end{aligned}$$

We retrieve the theta values from the table. In the case of large inputs, we return the value *NaN* as it would return zero as the result of range reduction.

$$
\begin{aligned}
&[\mathit{thetaHigh}, \mathit{thetaLow}] \\
&\quad \mid \mathit{bits}\ 52\ 63\ \mathit{xOffset} \\
&\qquad \not\equiv \mathrm{0x433} \quad = [\mathit{nan0X}, \mathit{nan0X}] \\
&\quad \mid \mathit{otherwise} \quad = \mathbf{let}\ [a, b] = \mathit{snd}\ \mathit{sincosTable} \\
&\qquad\qquad\qquad\qquad\qquad\qquad !!\ \mathit{fromIntegral}\ \mathit{lookupKey} \\
&\qquad\qquad\qquad\quad \mathbf{in}\ [2 \uparrow 63 + a, 2 \uparrow 63 + b]
\end{aligned}
$$

We construct the sine and cosine values from the table values, and we switch between the lookup key and its complement-based values based on whether the input lies in the odd or the even half-quadrant.

$$
\begin{aligned}
&(\mathit{costheta}, \mathit{sintheta}) = \mathbf{case}\ (\mathit{quadrant}, \mathit{div}\ \mathit{lookupKey}\ (2 \uparrow 6))\ \mathbf{of} \\
&\quad (0, 0) \rightarrow \mathit{fromKey}\ \mathit{lookupKey} \\
&\quad (0, 1) \rightarrow \mathit{switch}\ \$\ \mathit{fromKey}\ (\mathbf{xor}\ \mathit{lookupKey}\ (2 \uparrow 7 - 1)) \\
&\quad (1, 0) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ \mathit{lookupKey} \\
&\qquad \mathbf{in}\ (2 \uparrow 63 + b, a) \\
&\quad (1, 1) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ (\mathbf{xor}\ \mathit{lookupKey}\ (2 \uparrow 7 - 1)) \\
&\qquad \mathbf{in}\ (2 \uparrow 63 + a, b) \\
&\quad (2, 0) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ \mathit{lookupKey} \\
&\qquad \mathbf{in}\ (2 \uparrow 63 + a, 2 \uparrow 63 + b) \\
&\quad (2, 1) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ (\mathbf{xor}\ \mathit{lookupKey}\ (2 \uparrow 7 - 1)) \\
&\qquad \mathbf{in}\ (2 \uparrow 63 + b, 2 \uparrow 63 + a) \\
&\quad (3, 0) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ \mathit{lookupKey} \\
&\qquad \mathbf{in}\ (b, 2 \uparrow 63 + a) \\
&\quad (3, 1) \rightarrow \mathbf{let}\ (a, b) = \mathit{fromKey}\ (\mathbf{xor}\ \mathit{lookupKey}\ (2 \uparrow 7 - 1)) \\
&\qquad \mathbf{in}\ (a, 2 \uparrow 63 + b) \\
&\quad \_ \rightarrow \mathit{error}\ \texttt{"trigHWLookup.impossible happened"}
\end{aligned}
$$

Finally, we look for exceptional cases and return the corresponding values.

$$
\begin{aligned}
&(\mathit{cos}, \mathit{sin}) \\
&\quad \mid \mathit{isNan}\ \mathit{xOffset} = (\mathit{nan0X}, \mathit{nan0X})
\end{aligned}
$$

We detect very large inputs by comparing the exponent of the offsetted value with the exponent of the offset we added. If the input is very large, we return $1/\sqrt{2}$ for both sine and cosine values.

$$
\begin{aligned}
&\quad \mid \mathit{bits}\ 52\ 63\ \mathit{xOffset} \\
&\qquad \not\equiv \mathrm{0x433} \quad = (\mathit{nan2X}, \mathit{nan2X}) \\
&\quad \mid \mathit{otherwise} \quad = (\mathit{costheta}, \mathit{sintheta})
\end{aligned}
$$

We construct the extended precision double values for the sine and cosine values, which we get from the table.

$$
\begin{aligned}
&\mathit{fromKey}\ \mathit{key} = \mathbf{let}\ ((\mathit{cosE}, \mathit{cosM}), (\mathit{sinE}, \mathit{sinM})) \\
&\qquad\qquad\qquad\qquad = \mathit{fst}\ \mathit{sincosTable}\ !!\ (\mathit{fromIntegral}\ \mathit{key}) \\
&\qquad\qquad\qquad \mathbf{in}\ (\mathit{construct}\ (\mathit{cosE}, \mathit{cosM}) \\
&\qquad\qquad\qquad\quad , \mathit{construct}\ (\mathit{sinE}, \mathit{sinM})) \\
&\mathit{construct}\ (0, 0) \quad = 0 \\
&\mathit{construct}\ (\mathrm{0x3ff}, 0) = \mathrm{0x3ff8000000000000} \\
&\mathit{construct}\ (e, m) = (\mathrm{0x400} + \mathrm{0x3fe} - e) * (2 \uparrow 51) \\
&\qquad\qquad\qquad\quad + m * (2 \uparrow (51 - 32 + e + 1))
\end{aligned}
$$

switch is a small helper function.

$$\mathit{switch}\ (a, b) = (b, a)$$

Chapter 13

Inverse Trigonometric Family Functions

We implement all the inverse trigonometric functions using the atan2 function, with appropriate arguments. The core function, atan2, uses a hardware lookup instruction to provide an angle for angle reduction and its tangent value. These arguments are used to reduce the inputs to smaller range near the x-axis, using the following rotation matrix.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \frac{1}{\sqrt{1 + tan^2(\theta)}} \begin{bmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{13.1}$$

Since we still need to divide the inputs after the rotation by $-\theta$, the common factor $\dfrac{1}{\sqrt{1 + tan^2(\theta)}}$ is omitted in the computation.

13.1 Inverse Trigonometric Functions Software Implementation

module *InverseTrigSoft*
  where

atan2Family :: *PowerType a* ⇒ *VR a* → *VR a* → *VR a*
atan2Family y x = *result*
  where

The six hardware lookup instructions return the ordered inputs, the rotation angle, its tangent, and the adjustment we need to make based on the half-quadrant of the input co-ordinates.

$$\begin{aligned}&[u, v, tanTheta, theta, adjust1, adjust2]\\&\quad = map\ (\mathbf{lookupOpcode}\ y\ x\ 7)\ [1\,..\,6]\\&debug = \mathbf{lookupOpcodeDebug}\ y\ x\ 7\end{aligned}$$

We find the new co-ordinates after rotating by angle θ, to get co-ordinates in the segment near 0.

$$\begin{aligned}uInSegment &= \mathbf{dfma}\ v\ tanTheta\ u\\ vInSegment &= \mathbf{dfnms}\ u\ tanTheta\ v\end{aligned}$$

We find the ratio of the new co-ordinates, representing the tangent of the angle in the segment.

$$tanInSegment \quad = divDbl\ uInSegment\ vInSegment$$

The power series of the *arctan* function has the interleaved power in polynomial.

$$atan(x) = x - 1/3x^3 + 1/5x^5 + O(x^6) \tag{13.2}$$

We therefore find the minimax polynomial using Maple and the interleaved powers.

$$poly(x^2) = minimax\left(\frac{atan(x) - x}{x^3}\right) \tag{13.3}$$

The following maple code is used.

```
Digits := 50;
polyOrd := 2;
numSegment := 2^(12);
b := 2/numSegment;
p := numapprox[minimax](x->limit((arctan(y)-y)/y^3,y=sqrt(x)),
     -b..b,[polyOrd,0],1,'er');log[2](er);
seq(roundDbl(coeff(p(x),x,i)),i=0..polyOrd);
```

$$\begin{aligned}tanInSegmentSquared &= \mathbf{dfm}\ tanInSegment\ tanInSegment\\ evalPoly &= hornerVDbl\ ((undoubles2\ 1) : coeffs)\\ &\qquad tanInSegmentSquared\end{aligned}$$

The rotation angle is added to the evaluation of polynomial.

$atanInSegmentPTheta = \mathbf{dfma}\ tanInSegment\ evalPoly\ theta$

Finally, the adjustments are made based on the half-quadrant in which the inputs lie.

$result = \mathbf{dfma}\ adjust1\ atanInSegmentPTheta\ adjust2$

Inverse sine and cosine functions are calculated using the *atan2* function with appropriate arguments.

$$asin(v) = atan2(v, \sqrt{1 - v^2})$$
$$acos(v) = atan2(\sqrt{1 - v^2}, v)$$

$asincosFamily :: PowerType\ a \Rightarrow Bool \rightarrow VR\ a \rightarrow VR\ a$
$asincosFamily\ isSin\ v = result$
  where

We calculate $\sqrt{1 - v^2}$ for getting the second argument of *atan2*.

$oneMvSquare = \quad \mathbf{dfnms}\ v\ v\ (undoubles2\ 1)$
$sqrtOneMvSquare = rsqrtFamily\ False\ oneMvSquare$

We call the *atan2* function with right arguments.

$result = \mathbf{if}\ isSin$
  $\mathbf{then}\ atan2Family\ v\ sqrtOneMvSquare$
  $\mathbf{else}\ atan2Family\ sqrtOneMvSquare\ v$

13.2 Inverse Trig Lookup Instruction

module *InverseTrigLookup*
  where

Inverse Trig instruction returns the inputs, the rotation angle, its tangent, and the adjustments we need to make based on the half-quadrant in which the inputs lie.

$inverseTrigLookup$
  $:: \quad Integer \rightarrow Integer$

$$\begin{aligned} &\rightarrow \; (Integer, Integer, Integer, Integer, Integer, Integer) \\ inverseTrigLookup\ b\ a \;&= (newU, newV, c0, atanC0, adjust1, adjust2) \\ \textbf{where}& \end{aligned}$$

We use the table of the 2^n-intervals, representing first half quadrant of a rotation $theta \in [0, \pi/4)$.

$$n = 12$$

Since we calculate the inverse tangent in the first half quadrant,

$$\arctan\left(\min(a,b)/\max(a,b)\right),$$

we have to compare the inputs to get the ordering. Because we are comparing the first $n+2$ bits of the mantissas instead of the full mantissas, we need to include the extra interval with values $(1, \pi/4)$. We also need to extend the range of polynomial evaluation from $[0, 2^{-n})$ to $[-2^{-n}, 2^{-n}]$. Moreover, we also have to adjust the inputs such that, when we rotate the input co-ordinates to co-ordinates near zero, we do not loose precision in the y-axis or saturate to infinity in the x-axis. We also need to boost the subnormal inputs to normal range, to get the lookup key. $u = \max(a,b)$ and $v = \min(a,b)$. We will use an extra 2-bits for the approximate calculations, so that we will get a better lookup key.

$$\begin{aligned}
(u, v, switch) \;&= \textbf{case}\ (bits\ 52\ 63\ a, bits\ 52\ 63\ b)\ \textbf{of} \\
(0,0) \rightarrow \;& approxCompare\ (n+2) \\
& (53 * (2 \uparrow 52) + normalizeDenorm\ a) \\
& (53 * (2 \uparrow 52) + normalizeDenorm\ b) \\
(0, _) \;& \rightarrow \textbf{if}\ bits\ 52\ 63\ b \geqslant 52 + 1023 \\
& \quad \textbf{then}\ (b, a, 1) \\
& \quad \textbf{else}\ ((53+52) * (2 \uparrow 52) + b, \\
& \qquad\quad 53 * (2 \uparrow 52) + (normalizeDenorm\ a), 1) \\
(_, 0) \;& \rightarrow \textbf{if}\ bits\ 52\ 63\ a \geqslant 52 + 1023 \\
& \quad \textbf{then}\ (a, b, 0) \\
& \quad \textbf{else}\ ((53+52) * (2 \uparrow 52) + a, \\
& \qquad\quad 53 * (2 \uparrow 52) + (normalizeDenorm\ b), 0) \\
(_, _) \;& \rightarrow \textbf{let}\ (y, x, s) = approxCompare\ (n+2)\ a\ b \\
& \quad \textbf{in if}\ bits\ 52\ 63\ y \equiv \mathrm{0x7fe} \\
& \qquad \wedge\ bits\ 52\ 63\ x \geqslant \mathrm{0x7fe} - 52 \\
& \qquad \textbf{then}\ (y - 2 \uparrow 52, x - 2 \uparrow 52, s)
\end{aligned}$$

$$\begin{aligned}&\textbf{else if } \mathit{bits}\ 52\ 63\ y \leqslant 52\\&\quad\textbf{then } (53 * (2 \uparrow 52) + y, 53 * (2 \uparrow 52) + x, s)\\&\quad\textbf{else } (y, x, s)\end{aligned}$$

The maximum error introduced by these calculations is

$$u(1 + 2^{-(n+2)}) >= v, \tag{13.4}$$

where $u = max_{approx}(a, b)$ and $v = min_{approx}(a, b)$.

Now, we need to calculate the interval in which the input co-ordinates lie. We can do so by approximating the v/u. This can be done by looking up the estimate of the reciprocal of u from the reciprocal lookup table, then approximating their product based on the leading n-bits of their mantissas.

$$\begin{aligned}(\mathit{eRecipEstU}, \mathit{mRecipEstU}) &= \mathit{recipTable}\ (n+2)\ !!\\&\quad \mathit{fromIntegral}\\&\quad (\mathit{bits}\ (52 - n - 2)\ 52\ u)\\(\mathit{eApproxVByU}, \mathit{mApproxVByU}) &= \mathit{approxMult}\ (n+2)\\&\quad \mathit{mRecipEstU}\\&\quad (\mathit{bits}\ (52 - n - 2)\ 52\ v)\end{aligned}$$

The maximum error introduced by these calculations is

$$(u/v)(1 + 2^{n+2}) >= (u/v)_{approx} >= (u/v)(1 - 2^{n+2}). \tag{13.5}$$

The combined error introduced in the lookup key calculation is therefore $< 3 * 2^{-(n+2)}$. Since the accurate table values can be at either end of the interval, we have to double the range of the minimax polynomial.

Now we have to add the implied one to the approximate product and then shift the resultant value to get the lookup key.

$$\begin{aligned}\mathit{shift} &= \mathit{bits}\ 52\ 63\ u - \mathit{bits}\ 52\ 63\ v\\&\quad + 1 - \mathit{eRecipEstU} + 1 - \mathit{eApproxVByU} + 2\\\mathit{lookupKey} &= \mathit{roundShiftR} \qquad (2 \uparrow (n+2) + \mathit{mApproxVByU})\\&\quad (\mathit{fromIntegral\ shift})\end{aligned}$$

The values are retrieved from the table.

$$[\mathit{c0}, \mathit{atanC0}] = \mathit{arcTanTable}\ !!\ \mathit{fromIntegral\ lookupKey}$$

According to the signs and ordering of the inputs, we need to adjust the result.

$$
\begin{array}{l}
signA = bits\ 63\ 64\ a \\
signB = bits\ 63\ 64\ b \\
(sign, rotation) \quad = \textbf{case}\ (signA, signB, switch)\ \textbf{of} \\
\qquad \begin{array}{lll}
(0,0,0) & \rightarrow & (one, 0) \\
(0,0,1) & \rightarrow & (2 \uparrow 63 + one, piBy2) \\
(1,0,1) & \rightarrow & (one, piBy2) \\
(1,0,0) & \rightarrow & (2 \uparrow 63 + one, pi) \\
(1,1,0) & \rightarrow & (one, 2 \uparrow 63 + pi) \\
(1,1,1) & \rightarrow & (2 \uparrow 63 + one, 2 \uparrow 63 + piBy2) \\
(0,1,1) & \rightarrow & (one, 2 \uparrow 63 + piBy2) \\
(0,1,0) & \rightarrow & (2 \uparrow 63 + one, 0)
\end{array}
\end{array}
$$

Finally, we look for special inputs and overwrite the constants for u and v. These constants are returned because we do not want to generate NaN as a result of the calculation.

$$
\begin{array}{l}
(newU, newV, adjust1, adjust2) \\
\qquad \begin{array}{ll}
\mid isZero\ a \wedge isZero\ b & = (1, 1, 0, nan) \\
\mid isInf\ \ a \wedge isInf\ \ b & = (1, 1, 0, nan) \\
\mid isNan\ a \vee isNan\ b & = (1, 1, 0, nan) \\
\mid isZero\ a \vee isZero\ b & = (1, 1, 0, rotation) \\
\mid isInf\ \ a \vee isInf\ \ b & = (1, 1, 0, rotation) \\
\mid otherwise & = (absU, absV, sign, rotation)
\end{array} \\
absU = bits\ 0\ 63\ u \\
absV = bits\ 0\ 63\ v
\end{array}
$$

The following values $recipExtU$ or $approxUByV$ are useful for understanding the semantics of the instruction. These values are constructed for debug purpose.

$$
\begin{array}{ll}
recipEstU & = bits\ 63\ 64\ u * (2 \uparrow 63) + \\
& \quad (2046 - bits\ 52\ 63\ u \\
& \quad + eRecipEstU - 1) * (2 \uparrow 52) \\
& \quad + mRecipEstU * (2 \uparrow (52 - n)) \\
approxVByU & = \textbf{xor}\ (bits\ 63\ 64\ u)\ (bits\ 63\ 64\ v) * (2 \uparrow 63) \\
& \quad + (bits\ 52\ 63\ v - bits\ 52\ 63\ u \\
& \quad + 1023 - 1 + eRecipEstU \\
& \quad + eApproxVByU - 1) * (2 \uparrow 52) \\
& \quad + mApproxVByU * (2 \uparrow (52 - n))
\end{array}
$$

This function boosts subnormal inputs to normal ranges by multiplying by 2^{52}.

$$
\begin{aligned}
&\mathit{normalizeDenorm} :: \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{normalizeDenorm}\ n = (\mathit{bits}\ 63\ 64\ n) * (2 \uparrow 63)\\
&\qquad\qquad + (52 - \mathit{fromIntegral}\ \mathit{leading0s}) * (2 \uparrow 52)\\
&\qquad\qquad + \mathit{mod}\ (\mathit{shiftL}\ \mathit{mantissa}\ (\mathit{leading0s} + 1))\ (2 \uparrow 52)\\
&\quad \textbf{where}\\
&\qquad (\_, \mathit{leading0s}) = \mathit{countLeadingZeros}\ 52\ 52\ (\mathit{bits}\ 0\ 52\ n)\\
&\qquad \mathit{mantissa} = \mathit{mod}\ n\ (2 \uparrow 52)
\end{aligned}
$$

This function compares the absolute values of the inputs based on the leading n-bits of the mantissas.

$$
\begin{aligned}
&\mathit{approxCompare} :: \mathit{Int} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow (\mathit{Integer}, \mathit{Integer}, \mathit{Int})\\
&\mathit{approxCompare}\ n\ a\ b = \textbf{if}\ \mathit{bits}\ (52 - n)\ 63\ a \geqslant \mathit{bits}\ (52 - n)\ 63\ b\\
&\qquad \textbf{then}\ (a, b, 0)\\
&\qquad \textbf{else}\ (b, a, 1)
\end{aligned}
$$

This is the approximate multiplication of the mantissas based on the first n-bits only. Since the result is $\in [1, 4)$, we will return n-bits of the mantissa of the result and the exponent value $\in \{1, 2\}$.

$$
\begin{aligned}
&\mathit{approxMult} :: \mathit{Int} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow (\mathit{Integer}, \mathit{Integer})\\
&\mathit{approxMult}\ n\ a\ b = \textbf{case}\ \mathit{div}\ \mathit{result}\ (2 \uparrow n)\ \textbf{of}\\
&\qquad 1 \rightarrow (1, \mathit{mod}\ \mathit{result}\ (2 \uparrow n))\\
&\qquad \_ \rightarrow (2, \mathit{mod}\ (\mathit{roundShiftR}\ \mathit{result}\ 1)\ (2 \uparrow n))\\
&\quad \textbf{where}\\
&\quad m = (2 \uparrow n + a) * (2 \uparrow n + b)\\
&\quad \mathit{result} = \mathit{div}\ m\ (2 \uparrow n)
\end{aligned}
$$

Chapter 14

Evaluation

We have implemented two types of simulations of these instructions. First, to test our accuracy claims, we implemented the new instructions in the Coconut interpreter, adding them to the existing interpreter for Cell/B.E. SPU instructions. The results were compared to results calculated by Maple with 500 significant digits of precision, and are reported in Table 14.1. As expected, the accuracy results match the results for implementations using current Cell/B.E. SPU instructions. As the next step towards hardware implementation, we then implemented the instructions for logarithm using arrays of bits and logical operations, and verified that the results match the original implementations, which are written using integer and arbitrary-precision floating-point types in Haskell, and therefore both faster and easier to read.

14.1 Accuracy

We tested each of the functions by simulating execution using Coconut for at least 20000 pseudo-random inputs over the full range and compared the results to computations carried out in Maple with 500 significant digits of precision. We found the maximum error of two ulps over all the functions. For details see table 14.1.

14.2 Performance

Since the dependency graphs (Figure 4.1) are nearly linear, the performance of software-pipelined loops will be proportional to the number of floating-point instructions. We report in Table 14.1 the expected throughput for vector math functions which would result from adding these new instructions, assuming no other changes in the SPU execution model.

Table 14.1 reports the accuracy and performance statistics for the different elementary functions. The size of the specific table is only reported once. No entry for the table size implies the table is being shared with some other instruction. The number of zero bits of excess precision in the accurate table values is given, including the case $\log_2 M = \infty$ when all the table values can be chosen to be exact with the given table sizes. Other interpretation of M is that some of the table values are exact, whereas other values are within $\frac{1}{M}$ of an ulp of an IEEE representable floating point number.

function	$\frac{\text{cycles}}{\text{double}}$ new	$\frac{\text{cycles}}{\text{double}}$ SPU	Speedup (%)	max error (ulps)	table size(N)	poly order	$\log_2 M$
recip	3	11.3	**376**	0.500	2048	3	∞
div	3.5	14.9	**425**	1.333	recip	3	
sqrt	3	15.4	**513**	0.500	4096	3	18
rsqrt	3	14.6	**486**	0.503	4096	3	∞
log2	2.5	14.6	**584**	0.500	4096	3	18
log21p	3.5	n/a	n/a	1.106	log2	3	
log	3.5	13.8	**394**	1.184	log2	3	
log1p	4.5	22.5	**500**	1.726	log2	3	
exp2	4.5	13.0	**288**	1.791	256	4	18
exp2m1	5.5	n/a	n/a	1.29	exp2	4	
exp	5.0	14.4	**288**	1.55	exp2	4	
expm1	5.5	19.5	**354**	1.80	exp2	4	
atan2	7.5	23.4	**311**	0.955	4096	2	18
atan	7.5	18.5	**246**	0.955	atan2	2+3	
asin	11	27.2	**247**	1.706	atan2	2+3+3	
acos	11	27.1	**246**	0.790	atan2	2+3+3	
sin	11	16.6	**150**	1.474	128	3+3	52
cos	10	15.3	**153**	1.025	sin	3+3	
tan	24.5	27.6	**113**	2.051	sin	3+3+3	

Table 14.1: Accuracy and throughput (using Cell/B.E. SPU double precision) of standard functions with table sizes.

Overall, the addition of hardware instructions to the SPU results in a 3× improvement. Performance improvements on other architectures will vary, but would be significant.

The results (Table 14.1) show that the multiplicative reduction accurate table method achieves nearly correctly rounded results for some functions like *recip*, *log2* and roots and high throughput for almost all the functions. On the Cell/B.E. SPU, data-dependent branches are expensive, so we chose to implement these functions in branch-free form, which costs several cycles to handle exceptional cases, but saves tens of cycles for mispredicted branches. For some inputs, typical implementations of the Newton-Raphson method produce incorrect results where producing correct results is considered too expensive. For example, in some implementations, *recip* is frequently allowed to saturate to ∞ for some subnormal inputs although the correcly rounded output would be finite. This may be unacceptable for some applications, and it is a remarkable property of the library using the proposed instructions that no such compromises in the handling of rare cases were necessary.

Chapter 15

Conclusion

We have demonstrated that when supported by a hardware lookup instruction and an extended-range fused multiply-add, we can achieve considerable performance and accuracy improvements for the elementary functions. In some of the functions, maximum errors as low as 0.500 ulps are achieved by combining multiplicative reduction using a fused multiply-add and accurate (or exact) tables. Overall, using accurate tables for all the functions bounds the maximum error to 2.051 ulps compared to more than 1000 ulps of error reported for the SPU MASS library. Our analysis has shown that new algorithms based on the proposed hardware instructions would triple throughput on the Cell/B.E. SPU.

We showed by example that the new instructions are simple enough, certainly much simpler than floating point arithmetic, and would fit into the latency and register-use requirements of conventional processors. We were also able to handle all the exceptional cases internally in hardware, thereby eliminating the need for data dependent branches.

The work has been reported in a joint patent application [AES10] with IBM. To further reduce register pressure, we also proposed variants of the new instructions with hidden internal state, and discuss the impact of such a decision on superscalar dispatch and required operating system support.

Although we have not explored it in this thesis, these methods are equally applicable to higher- and lower-precision floating point function evaluation, although the table sizes and polynomial orders should be adjusted if greater performance is required processing 32-bit floats or better accuracy is required processing 128-bit (or higher) floats.

Finally, we acknowledge that although the proposed methods will provide very high performance, some functions are not as accurate as we would like, particularly, *div*, the exponentials and the trigonometric functions, but there is some hope that taking rounding into account [BC07] when searching for both tables and polynomial values could reduce the maximum error.

Bibliography

[ACG$^+$86] Ramesh C. Agarwal, James W. Cooley, Fred G. Gustavson, James B. Shearer, Gordon Slishman, and Bryant Tuckerman. New scalar and vector elementary functions for the IBM System/370. *IBM J. Res. Dev.*, 30(2):123–144, 1986.

[AES10] Christopher K. Anand, Robert Enenkel, and Anuroop Sharma. Hardware instructions to accelerate table-driven mathematical function evaluation. United States Patent Application 12/788570, filed May 27, 2010.

[AK08] Christopher K. Anand and Wolfram Kahl. Synthesising and verifying multi-core parallelism in categories of nested code graphs. In Michael Alexander and William Gardner, editors, *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[AK09] Christopher K. Anand and Wolfram Kahl. An optimized Cell BE special function library generated by Coconut. *IEEE Transactions on Computers*, 2009.

[AS09] Christopher K. Anand and Anuroop Sharma. Unified tables for exponential and logarithm families. AdvOL Report 2009/2, McMaster University, 2009.

[AS10] Christopher K. Anand and Anuroop Sharma. Unified tables for exponential and logarithm families. *ACM Transactions on Mathematical Software*, 37(3), 2010.

[BC07] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞ approximations. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Santa Monica, USA, 2007. IEEE Computer Society Press, Los Alamitos, CA.

[Bem63] R. W. Bemer. A note on range transformations for square root and logarithm. *Commun. ACM*, 6(6):306–307, 1963.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *In POPL 05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13. ACM Press, 2005.

[Cor05] International Business Machines Corporation. *PowerPC User Instruction Set Architecture.* IBM Systems and Technology Group, http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html, 2005.

[Cor08] International Business Machines Corporation. *Cell Broadband Engine Programming Handbook.* IBM Systems and Technology Group, https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC, 2008.

[DTSS05] Son Dao Trong, Eric M. Schwarz, and Martin Schmookler. FPU implementations with denormalized numbers. *IEEE Trans. Comput.*, 54(7):825–836, 2005.

[EL93] Miloš D. Ercegovac and Tomás Lang. Multiplication/division/square root module for massively parallel computers. *Integr. VLSI J.*, 16(3):221–234, 1993.

[Gal86] Shmuel Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Proceedings of the Symposium on Accurate Scientific Computations*, pages 1–16, London, UK, 1986. LNCS 235, Springer-Verlag.

[GB91] Shmuel Gal and Boris Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans. Math. Softw.*, 17(1):26–45, 1991.

[IBM06] IBM Corp. *Synergistic Processor Unit Instruction Set Architecture.* IBM Systems and Technology Group, Hopewell Junction, NY, October 2006.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism, 1995.

[KAC06] Wolfram Kahl, Christopher Kumar Anand, and Jacques Carette. Control-flow semantics for assembly-level data-flow graphs. In

Wendy MacCaull, Michael Winter, and Ivo Düntsch, editors, *RelMiCS 2005*, volume 3929 of *LNCS*, pages 147–160. Springer, 2006.

[KM97] Alan H. Karp and Peter Markstein. High-precision division and square root. *ACM Trans. Math. Softw.*, 23(4):561–589, 1997.

[KM06] Peter Kornerup and Jean-Michel Muller. Choosing starting values for certain Newton-Raphson iterations. *Theor. Comput. Sci.*, 351(1):101–110, 2006.

[Mul05] Jean-Michel Muller. On the definition of ulp (x). Technical report, RR2005-09, LIP, ENS Lyon, France, 2005.

[PJ+03] Simon Peyton Jones et al. *The Revised Haskell 98 Report.* Cambridge University Press, 2003. Also on `http://haskell.org/`.

[Rus98] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD K7 processor. *LMS J. Comput. Math*, 1:148–200, 1998.

[SAG99] Martin S. Schmookler, Ramesh C. Agarwal, and Fred G. Gustavson. Series approximation methods for divide and square root in the Power3(TM) processor. In *ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, page 116, Washington, DC, USA, 1999. IEEE Computer Society.

[Sch95] Eric M. Schwarz. Rounding for quadratically converging algorithms for division and square root. In *ASILOMAR '95: Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers Volume II*, page 600, Washington, DC, USA, 1995. IEEE Computer Society.

[Tak97] Naofumi Takagi. Generating a power of an operand by a table look-up and a multiplication. In *ARITH '97: Proceedings of the 13th Symposium on Computer Arithmetic (ARITH '97)*, page 126, Washington, DC, USA, 1997. IEEE Computer Society.