The K5 Transcendental Functions

Tom Lynch, Ashraf Ahmed, Mike Schulte,
Tom Callaway, and Robert Tisdale
PC Products Division
Advanced Micro Devices
Austin, TX 78741

Abstract

This paper describes the development of the transcendental instructions for the **K5**, *AMD's recently completed x86 compatible superscalar microprocessor. A multi-level development cycle, with testing between levels, facilitated the early detection of errors and limited their effect on the design schedule. The algorithms for the transcendental functions use table-driven reductions followed by polynomial approximations. Multiprecision arithmetic operations are used when necessary to maintain sufficient accuracy and to ensure that the transcendental functions have a maximum error of one unit in the last place.*

1 Introduction

This paper discusses the implementation of an "x86" architecture floating-point unit. The first processor to use this architecture was the 8087 numerical extension which is described in [10]. This processor used CORDIC to implement elementary functions. It was pointed in [7] that series approximations, or rational approximations can be efficiently used with the now common pipelined floating-point units. A successful x86 architectural implementation of this is described in [13]. In this paper, we describe an x86 style floating-point unit implemented on a superscalar micro-processor.

The **K5** [1] microprocessor is a superscalar RISC machine with a special instruction translation unit for x86 architecture compatibility [6] [8]. Accordingly, instructions are dispatched either from the CISC to RISC instruction translation unit, or from a RISC-code ROM. The more complex x86 instructions, such as those for approximating transcendental functions, are implemented as RISC instruction sequences which are stored in the RISC-code ROM.

The K5 processor contains two ALUs, a load store unit, a branch unit, and a floating-point unit, all of which operate independently. Hence, instructions may be issued and completed out of order. Up to four transcendental code RISC operations are dispatched from the RISC-code ROM at a time.

Much of the transcendental code uses the pipelined floating-point unit. This unit contains a 66-bit floating-point adder, a 32 x 32 bit multiplier, appropriate special case detection comparators, and a rounder. The fundamental data types contain a sign bit, a 17-bit exponent, and either a 24, 32, 53 or 64-bit mantissa. The extended exponent range is available for internal calculations. Much of this hardware was inherited from an existing RISC microprocessor.

The nature of the floating-point unit lead to the conclusion that table driven reductions followed by polynomial approximations similar to those described in [12] [5] and [3] would be appropriate.

Our first requirement was that each routine calculate results with a maximum error of less than 1 unit in the last place (*ulp*). The second requirement was that very few changes be made to the existing hardware. This requirement stemmed from the multiplicative effect of widening busses in a superscalar processor, and the short project schedule.

The existing hardware allowed for about one temporary register per pipe stage, little micro-code ROM space, a limited number of constants, and a data path width that did not have extra guard bits beyond the width of the largest architecture-visible data type. These restrictions lead to use of a multiprecision arithmetic, a difficult register scheduling problem, the need to share code sections, and the packing of constants based on their required precision. Add to this the desire to produce bug-free routines on the first try, and the microcode development problem became less than trivial. Hand coding of the RISC operations and register assignments was intractable. A typical routine contains hundreds of independent variables which have to be assigned to 15 41-bit registers. A bug fix done by hand could take nearly a week because of ripple effects into the remaining micro-code. Also, hand coding of

[1] trade mark of Advanced Micro Devices

the micro-code would leave no obvious way to gauge the reliability of the final product.

We adopted a hierarchical development flow which mimics the formal verification ideas described in [15]. The flow started with the development of arbitrary precision versions of the routines, proceeded with versions which contain truncated series evaluations and some finite precision operations, and ended with the coding of fixed precision code which was compiled into a ROM image. Most development was done in the environment of Mathematica, so algorithm specifications were executable. Mathematica was not designed for this application, so a library of conversion routines, and Mathematica bug and feature work-arounds had to be made. A special compiler was written for getting the fixed precision algorithm statements into a ROM image. The ROM image was checked via whole chip RTL level simulation using the *Verilog* simulator.

The next section briefly describes the transcendental algorithms. Section 3 discusses the five-step development and verification flow. Section 4 contains an example of the development process for the function $2^x - 1$. The paper concludes with Section 5.

2 Algorithms

The transcendental functions implemented on the **K5** microprocessor include $sin(x)$, $cos(x)$, $tan(x)$, $arctan(x)$, $y \cdot log(x)$, $y \cdot log(x+1)$, and $2^x - 1$. The algorithms for approximating the transcendental functions consist of three main steps: argument reduction, a Horner's series evaluation, and formation of the final result. The trigonometric functions reduce the domain of θ in $[-2^{63}, 2^{63}]$ to a range of $(-\pi/4, \pi/4)$ by subtracting integer multiples of $\pi/2$ from the input operand. $\pi/2$ is represented with up to about 256 bits depending on how much precision is required. The multiple precision arithmetic made handling such a precise value of π straight forward and efficient. The other functions use table-driven reduction techniques to accurately reduce the domain of the input operand.

2.1 Multiprecision Arithmetic

To obtain a maximum error of less than 1 *ulp* without an extended width data path, it is necessary to incorporate some multiprecision routines. These routines typically operate on a *three-digit* data type, where the first two digits are the top and bottom 32 bits of an internal floating-point number with a 64-bit significand. The third digit is held in a separate floating-point number with a 24-bit significand. This provides a total of at least 88 bits of precision. Each three-digit number occupies three temporary registers. It is the job of the programmer and the compiler to keep this conceptual entity together.

Analysis during design is used to determine which steps in the algorithm require multiprecision calculation. This is later checked with a careful step-by-step derivation of the worst case propagation of roundoff errors. The precision of each operation can be specified in the compiled input language in units of digits. Typically one extra digit is a lot of extra precision so the faster analysis which was performed during the design phase was almost always shown to be correct by the more careful verification analysis.

Arithmetic operations on the multiprecision numbers are performed using algorithms similar to the ones given in [11], for addition, subtraction, multiplication, and division. For example, the algorithm *AddExtExtMp* adds two extended precision numbers a and b to produce a multiprecision number x with a most significant part x_m and a least significant part x_l. Here, $swap(a, b)$ exchanges a and b, and $round(r, p)$ rounds r to p bits of precision using round-to-nearest-even.

```
Algorithm: AddExtExtMp
Input:      Extended precision numbers $a$ and $b$
Output:     A multiprecision number $x$ with most significant part $x_m$ and least significant part
            $x_l$, such that $x_m + x_l = (a + b)(1 + \epsilon)$,
            where $|\epsilon| < 2^{-88}$
Procedure:  if ($|a| < |b|$)
                $swap(a, b)$
            $x_m = round(a + b, 64)$
            $b_h = round(x_m - a, 64)$
            $x_l = round(b - b_h, 24)$
            return $(x_m, x_l)$
```

Initially, a and b are compared to determine the larger of the two operands. The larger operand is assigned to a and the smaller operand is assigned to b. Next, a and b are added together to produce x_m, which is rounded to 64 bits. When adding a and b, if the exponents of a and b differ, b is right shifted with respect to a. After this, a is subtracted from x_m. This returns the value b_h, which corresponds to the most significant bits in b that are not discarded when $a + b$ is rounded to 64 bits. Finally x_l is computed by subtracting b_h from b and rounding the result to 24 bits. Thus, x_l is an approximation to the bits that are discarded when rounding $a + b$ to 64 bits, and $x_h + x_l \approx a + b$.

2.2 The Algorithm for $2^x - 1$

As an example, the identity for approximating $2^x - 1$ is given below. The domain of x is (-1,1).

$$\begin{aligned} 2^x - 1 &= 2^{(u+v)} - 1 \\ &= (2^u - 1)(2^v - 1) + (2^u - 1) + (2^v - 1) \\ &= g \cdot h + g + h \end{aligned}$$

where $x = u + v$, $g = 2^u - 1$ and $h = 2^v - 1$.

The algorithm consists of the following steps:

1. A reduction step that obtains the values for u and v from x.
2. A Taylor's series approximation on the reduced value v, which returns $h = 2^v - 1$.
3. A table lookup to find the value of $g = 2^u - 1$.
4. A step that puts together g and h (based on the above identity) to produce the value of the function.

The reduction step takes the input x and produces two values u and v, such that u is a member of a set with $2r + 1$ values and v has a reduced range. The values of u and v are computed as

$$u = \frac{Rnd(r \cdot x)}{r} \quad (1)$$

$$v = x - u \quad (2)$$

where $Rnd(x)$ returns the integer that is closest to x. Since $x \in (-1, 1)$, $u \in \{i/r : -r \leq i \leq r\}$ and the domain of v is reduced to $[\frac{-1}{2 \cdot r}, \frac{1}{2 \cdot r}]$. For our implementation, r is chosen as 16. A table with $2 \cdot 16 + 1 = 33$ entries is used to stored the values of $g = 2^u - 1$, and v has a reduced range of $[\frac{-1}{32}, \frac{1}{32}]$.

A standard Taylor's series approximation is done to estimate $h = 2^v - 1$. A Taylor's series approximation is used instead of a Chebyshev polynomial, because the coefficients for the Taylor's series approximation can be generated on-the-fly. This is important, since we have limited on-chip memory for storing coefficients.

Before performing the polynomial approximation, the variable v is transformed to the variable w using the identity

$$e^w - 1 = 2^v - 1 \quad (3)$$

where $w = v \cdot \ln(2)$.

The truncated Taylor's series approximation is

$$e^w - 1 \approx T(w) = \sum_{i=1}^{n} \frac{w^i}{i!} \quad (4)$$

where the number of terms n is 9.

Using Horner's rule, we get (from eqn. 4)

$$T(w) = (((w + C_n)w + C_{n-1})w + \cdots + C_3)\frac{w^2}{n!} + w \quad (5)$$

where

$$C_n = n \quad C_{i-1} = C_i \cdot (i - 1), \quad (6)$$

The series coefficients, C_i, are computed, on-the-fly, during each iteration of the series evaluation according to equation (eqn. 6). The only operations done in multiprecision arithmetic in this step are the calculation of w (since ln(2) is multiprecision), and the addition of w at the end of the series (eqn. 5).

Once the series is evaluated, the value for $g = 2^u - 1$ is obtained from a table-lookup. Each constant in the table is stored as a multiprecision value. After this, a multiprecision multiplication and two multiprecision additions are used to perform the computation

$$2^x - 1 = g \cdot h + g + h \quad (7)$$

3 Development

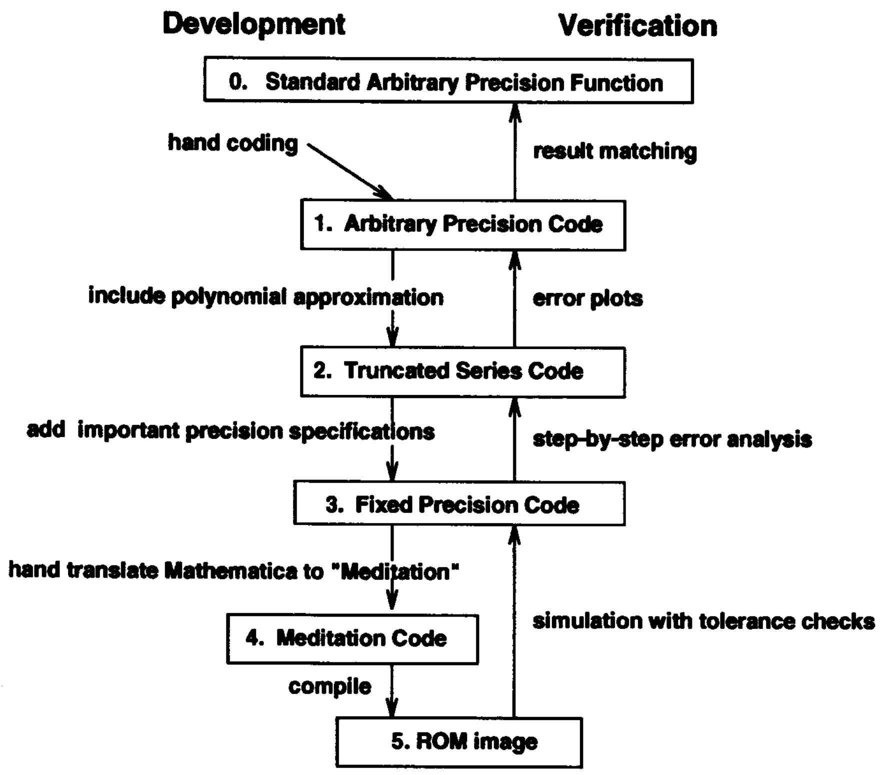
A multi-level approach was used to implement the algorithms discussed in the previous section. Each level consisted of both an implementation stage and a verification stage. There were five levels: the executable reference function, the arbitrary precision code, the truncated series code, the fixed precision code, and the microcode. Figure 1 shows the various development levels, and the implementation and verification associated with each level.

This step-wise refinement procedure isolates different levels of abstraction. Thus, the nature of possible bugs at each level is of a particular type. Hence, a different verification methodology was used for each level. These methodologies were applied in order, starting with the most abstract level and working down. This approach prevents possible bugs from propagating through the design cycle and thus causing risk to the schedule because of the need to repeat a lot of work. For example, the Level 1 code is evaluated with arbitrary precision arithmetic with (nearly) infinitely precise approximations, so no bugs due to roundoff error, propagation or approximation error are likely. When Level 1 is clean, Level 2 can be tested just for the effects of approximation error. When this level is clean, Level 3 is limited to only the effects of roundoff error propagation and so on.

Much of our code was developed using Wolfram Research's Mathematica [1]. Mathematica provided the ability to evaluate functions to an arbitrary precision in binary arithmetic. The complete environment gave us the ability to explore the effects of changing parameters like table-size, domain of the reduced argument, number of terms in a series, etc., often by just making plots of the effect in question.

3.1 Level 0: Arbitrary Precision Reference Function

The highest level of abstraction is a simple reference function which is used to verify the final results



Development
Verification
0. Standard Arbitrary Precision Function
hand coding
result matching
1. Arbitrary Precision Code
include polynomial approximation
error plots
2. Truncated Series Code
add important precision specifications
step-by-step error analysis
3. Fixed Precision Code
hand translate Mathematica to "Meditation"
simulation with tolerance checks
4. Meditation Code
compile
5. ROM image

Figure 1: The Levels of Algorithm Definition and In-between Check Methods

in all other levels. Hopefully, errors in the reference, if any, will be of a different nature than errors in the code and hence they will not mask errors in lower level code. The Mathematica function calls we used, have a wide user base, and no significant bugs were known to exist. This added more credulity to routines as standard references. In addition to relying on the Mathematica results we tried to apply common sense knowledge about the mathematical behavior of the functions.

3.2 Level 1: Arbitrary Precision Code

The Level 1 code is an arbitrary precision implementation of the algorithm in Mathematica. This is the first hand-generated interpretation of the algorithms and is used to verify the correctness of the algorithm. This includes the steps needed for range reduction, function evaluation over the reduced range, and formation of the final result.

Since the Level 1 code contains no rounding or truncation, errors that occur will be due to approximating the function incorrectly. Such errors are located by looking for differences between the results produced by the arbitrary precision code and those produced by the standard. Because of the continuous nature of the functions being approximated, we do not expect a spurious error – many-point checking of sufficient resolution thus seems safe.

Level 1 code is verified by comparing the results produced by the arbitrary precision algorithm to the ones produced by the reference function. These results should be identical since approximation errors and rounding errors have not yet been introduced. This verification ensures that conceptual errors do not appear at later levels.

3.3 Level 2: Truncated Series Code

In Level 2, the evaluation of the function over the reduced range is replaced by an approximation. The decisions made here include the method of polynomial approximation, (e.g., Taylor Series or Chebyshev), and the number of terms in the series. This level is also coded using arbitrary precision operations and hence no rounding errors are introduced. The truncated series code is verified with plots of the approximation error. These plots are generated by comparing the results from the truncated series code to the arbitrary precision code and the reference function. Approximation errors introduced by truncated series are well understood [4], so the error curves are expected to have a specific form - this can be checked for.

3.4 Level 3: Fixed Precision Code

The next development level is a fixed precision implementation of the algorithm in Mathematica. This is a modification to the previous code which introduces fixed precision operations modeled after those supported by the **K5** hardware. This level takes into account quantization of the coefficients, rounding errors caused by fixed precision arithmetic, and the propagation of these errors to the final result.

The precision of the intermediate results is determined to be either one, two, or three digits. Additionally, some operations that are performed in a single step in the arbitrary precision code are split into multiple steps. This is necessary to emulate the operations performed by the hardware.

The verification step for Level 3 is a step-by-step error analysis of the fixed precision code. This analysis computes error bounds due to approximation error and the propagation of roundoff errors through the algorithm.

The error analysis primarily ensures that the fixed precision algorithm has the desired accuracy. It also provides values for the maximum allowable errors for all intermediate results. These numbers are used to verify the microcode, as discussed in the next subsection. In addition to the error analysis, the results of the Level 2 fixed precision code are checked against the truncated series code, the arbitrary precision code and the reference function as a further accuracy check.

3.5 Level 4: Microcode

The final development level is the microcode implementation. The fixed precision code is first translated into an intermediate language called *Meditation*. The *Meditation* code is then run through a compiler which produces assembly language code as output. Finally, the assembly language code is run through a microcode assembler to produce a ROM image.

The Meditation language is an extension of the microcode assembly language which provides function calls for multiprecision operations and support for symbolic variables. The RISC machine does not possess a stack, nor are there sufficient data registers to emulate one. Thus, the compiler either inline expands a called routine, or arranges register concurrence between the subroutine and the calling program.

The Meditation language supports an important feature which allows verification of the algorithm in its final form. The language allows the programmer to embed information about the error tolerance of intermediate variables. Thus, along with the ROM image, the compiler generates a set of *Verilog* "snooper" modules that can be incorporated into the *Verilog* RTL model of the processor. These modules contain information which can be used in full-chip simulation for checking intermediate values and results from the transcendental routines.

Each time a tagged intermediate value is generated while running numerical programs, the linked-in Verilog module generates the Mathematica code necessary for performing a test on that value. After the simulation is finished, the automatically generated Mathematica program is executed. This program produces an error message if the value of any intermediate variable or final result is larger than the error tolerance (as obtained from the error analysis).

4 An Example : The Development of $2^x - 1$

4.1 Level 0: Arbitrary Precision Reference Function

Unlike the other functions, there is not a direct function call for $2^x - 1$ in Mathematica, and the composite calculation has numerical/performance problems when used as an executable specification. Evaluation at a precision equivalent to 10,000 decimal digits is accurate enough for our purposes, but a bit impractical. In practice, the precision of the computation is selected to ensure sufficient accuracy, while maintaining reasonable computational delay. The N function is used to set the precision of the computation to an appropriate value.

```
F2XM1L0[x_] := N[2^x - 1, 10000]
```

4.2 Level 1: Arbitrary Precision Code

The Level 1 implementation contains the steps in the algorithm without series truncation or rounding. Figure 2 is the Level 1 Mathematica formulation for $2^x - 1$. This code implements the algorithm given in Section 2 to arbitrary precision.

This level was tested by comparing plots to ensure identical behavior between the arbitrary precision code and the reference function. In addition, fifty thousand points were randomly chosen to ensure that the results produced by the first two levels matched.

4.3 Level 2: Truncated Series Code

The Level 2 code is similar to the Level 1 code, except that a truncated polynomial series is included. For $2^x - 1$, a Taylor series is used so that coefficients may be calculated on-the-fly. The Level 2 Mathematica code is shown in Figure 3.

Level 2 is verified by plotting the error in the truncated series code as compared to the reference function. The error in the polynomial approximation for **F2XM1L2** is shown in figure 4 over the domain of the

```
r = 16                                         (* number of table entries *)

For[ i = -r, i <= r , i = i + 1,               (* calculate table entries *)
   aTableL1[i] = 2^(i/r) - 1;
];

f2xm1PolyL1[xx_] := Module[ {},                (* Taylor series approximation *)
   Normal[Series[ 2^xx-1, {xx,0,100}]];
];

f2xm1L1[xx_] := Module[ {},                    (* Level 1 algorithm for f2xm1 *)
   x = N[xx,512];
   index = Round[ r x ];                       (* Range reduction *)
   u = index/r;
   v = x - u;
   h = f2xm1PolyL1[v];                         (* Polynomial approximation *)
   g = aTableL1[index];                        (* Formation of result *)
   result = g h + g + h;
   Return result;
]
```

Figure 2: Mathematica Implementation of Arbitrary Precision Code

```
f2xm1PolyL2[xx_] := Module[ {},               (* Taylor series approximation *)
   x = N[xx,80];
   w = N[Log[2],80] x;
   poly1= (((((((w+9)w+9 8)w+9 8 7)w+9 8 7 6)w+9 8 7 6 5)
            w+9 8 7 6 5 4)w+9 8 7 6 5 4 3)w^2;
   poly2 = poly1 1/9!;
   h = w + poly2;
   Return h;
]
```

Figure 3: Mathematica Implementation of Truncated Series Code

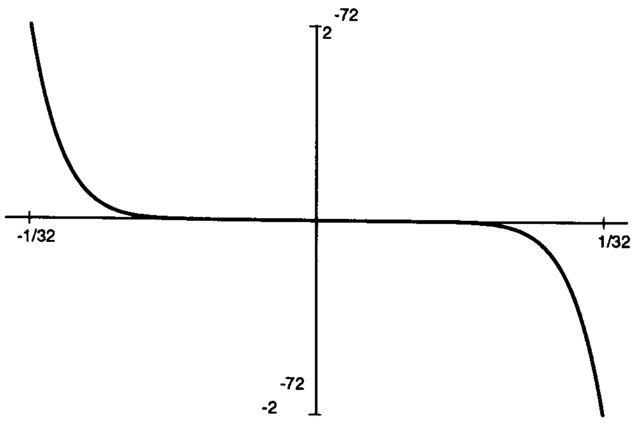


Figure 4: Polynomial Approximation Error for **F2XM1L2**

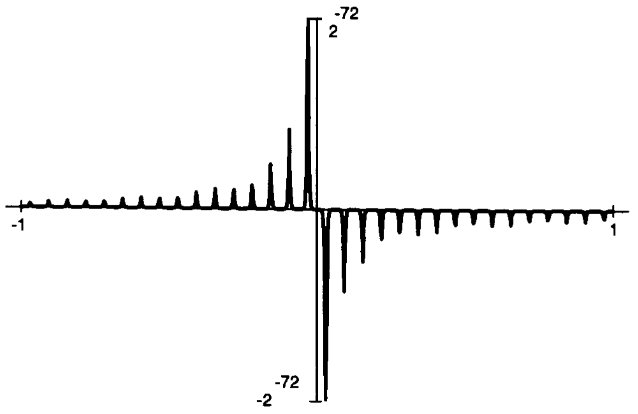


Figure 5: Accumulated Error for **F2XM1L2**

reduced argument v. The error for the entire Level 2 implementation is shown in figure 5 over the domain of the input operand x. Since these polynomials are known to behave in a smooth manner, this method is sufficient to determine the maximum relative approximation error. The maximum relative approximation error for **F2XM1L2** is roughly 2^{-72}.

4.4 Level 3: Fixed Precision Code

The next development level is a modification to the previous code which introduces fixed precision operations modeled after those produced by the **K5** hardware as described in the previous section. The verification procedure for Level 3 is a formal error analysis of the fixed precision code. This analysis also provides maximum allowable errors tolerance that should be expected at steps in the production code. In addition to the error analysis, the final results are also checked against the reference.

The following paragraphs outline the error analysis that was done. The reader may find it useful to refer back to section 2 where the algorithm for $2^x - 1$ was described.

For the error analysis of $2^x - 1$, the operand x is presumed to be exact and there is no error in the operations in which u is obtained from x. Therefore, the difference, $v = x - u$, is also exact (see section 2 equation 1). A multiprecision constant, ln(2), with relative representation error, $|\rho_{\ln(2)}| < \eta$ where $\eta = 2^{-87}$, is used. Thus, the total relative rounding error, $|\rho_w| < 3\eta/2$, in the multiprecision product, $w = v \cdot \ln(2)$, is very small.

A value with a precision of 64 bits, w_0, is obtained by rounding w to 64 bits. This value is used for the first part of the Horner evaluation of the truncated Taylor series, because high accuracy is not required until the final term is added. The total relative error, $|(w - w_0)/w| < \epsilon/2$, where $\epsilon = 2^{-63}$. The total relative rounding error, $|\rho_{\sigma_i}| < \epsilon/2 + 33 \cdot 2^{-11}\epsilon$, in each sum, $\sigma_i = \pi_i + C_i$ where $\pi_i = \sigma_{i+1} w$, is small because the coefficients, C_i, in the Horner evaluation are error free. Error accumulates again when multiplying by $w^2/n!$. This error is reduced when the final multiprecision term, w, is added because w is much larger than the first term. Including the series truncation error, the total relative error, $|\rho_h| < 84.1 \cdot 2^{-11}\epsilon$, in $h = 2^v - 1$ is very small.

The multiprecision values, $g = 2^u - 1$, from the table lookup have a representation error, $\rho_g \leq \eta/2$. The total relative error, $|\rho_{g \cdot h}| < 84.2 \cdot 2^{-11}\epsilon$, in the product is still small, but the possibility of cancelation in the next two sums increases the total relative error, $|\rho_{2^x-1}| < 505.5 \cdot 2^{-11}\epsilon$, Rounding to 64 bits introduces an additional error of $\epsilon/2$ so that the final result has an error less than $3\epsilon/4$, which is less than 1 ulp.

4.5 Levels 3 and 4: Meditation Code and ROM Image

The final step is the Meditation implementation of the Level 2 code followed by the generation of the ROM image and the corresponding *Verilog* tasks. Large numbers of random test cases are run in full-chip simulation and the generated Mathematica error tolerance program is run to verify the accuracy of the implementation.

5 Conclusion

This paper outlined our efforts to develop accurate and reliable transcendental functions for the **K5** microprocessor. The paper described the task at hand for each step in the development, and how the results of performing these tasks were verified. These steps included coding the algorithms in arbitrary precision code, changing the operations to finite precision, and finally producing a ROM image. The verification for each step included graphing relative errors against a standard, formal error analysis, simulation with error tolerance checks, and brute force simulation.

References

[1] *Mathematica : a System for Doing Mathematics by Computer*. Addison-Wesley Pub. Co., 1991.

[2] G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula. Semantics for Exact Floating Point Operations. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 22–27, 1991.

[3] P. Farmwald. High Bandwidth Evaluation of Elementary Functions. In *Proceedings of the 5th Symposium on Computer Arithmetic*, pages 139–142, 1991.

[4] Warren Ferguson and Tom Brightman. Accurate and Monotone Approximations of Some Transcendental Functions. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 237–244, 1991.

[5] C.T. Fike. *Computer Evaluation of Mathematical Functions*. Prentice Hall, 1968.

[6] Shmuel Gal and Boris Bachelus. An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard. *ACM Transactions on Mathematical Software*, 17:26–45, 1991.

[7] Tom R. Halfhill. AMD vs. SuperMan. **Byte**, pages 95–103, November 1994.

[8] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[9] Israel Koren and Ofra Zinaty. Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations. *IEEE Transactions on Computers*, v39 no 8:1030–1037, 1990.

[10] P.W. Markstein. Computation of Elementary Functions on the IBM RISC System/6000 Processor. *IBM Journal of Research and Development*, 34:111–119, 1990.

[11] Rafi Nave. Implementation of Transcendental Functions on a Numerics Processor. In *Microprocessing and Microprogramming 11*, pages 221–225, 1983.

[12] Douglas M. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–143, 1991.

[13] Ping Tak Peter Tang. Table-Lookup Algorithms for Elementary Functions and Their Error Analysis. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 232–236, 1991.

[14] J.H. Wilkinson. *Rounding Errors in Algebraic Processes*. Her Majesty's Stationery Office, 1963.

[15] W.R. Bevier, W.A. Hunt, Jr., J S. Moore, W.D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.