Journal of Universal Computer Science, vol. 1, no. 7 (1995), 439-453
submitted: 15/12/94, accepted: 26/6/95, appeared: 28/7/95©Springer Pub. Co.

A High Radix On-line Arithmetic for Credible and Accurate Computing

Thomas Lynch
(Advanced Micro Devices, U.S.A
Tom.Lynch@amd.com)

Michael J. Schulte
(University of Texas at Austin, U.S.A
schulte@pine.ece.utexas.edu)

Abstract: The result of a simple floating-point computation can be in great error, even though no error is signaled, no coding mistakes are in the program, and the computer hardware is functioning correctly. This paper proposes a set of instructions appropriate for a general purpose microprocessor that can be used to improve the credibility and accuracy of numerical computations. Such instructions provide direct hardware support for monitoring events which may threaten computational integrity, implementing floating-point data types of arbitrary precision, and repeating calculations with greater precision. These useful features are obtained by the efficient implementation of *high radix on-line arithmetic*. The prevalence of super-scalar and VLIW processors makes this approach especially attractive.
Key Words: High-radix, on-line arithmetic, precision, accurate, reliable, credible, super-scaler, VLIW.

1 Introduction

> *One of the principle problems of numerical analysis is to determine how accurate the results of certain numerical methods will be. A "credibility-gap" problem is involved here: we don't know how much of the computer's answers to believe.*
> - Donald Knuth [Knuth 81]

For a program to be *credible*, the results it produces must not be misleading. Hence, a program that always returns the value 'indeterminate' is credible; although it is not accurate. It is highly desirable to define an arithmetic that can be used to develop credible and accurate programs, and then to support the arithmetic in hardware so that it can be fast and efficient.

The most common approach to the credibility/accuracy problem has been the "use lots of bits" approach. For example, IEEE std. 754 [IEEE 85] implementations often have 64 bit data paths. Although it is unlikely that so much precision is needed at any step in a program, in the rare case that it is needed the precision is available. Still, an IEEE std. 754 conformant program can produce results that are completely inaccurate without warning [Lynch and Swartzlander 92, Bohlender 90].

The IEEE std. 754 rounding specifications facilitate a credible interval arithmetic [Moore 66, Nickel 85, Alefeld 83]. Accordingly, upper and lower bounds of intervals which contain the true results are calculated. This, however, does not guarantee accuracy, since interval boundaries may diverge due to accumulated numerical errors and pessimistic assumptions.

Several software approaches have been developed to produce credible and accurate arithmetic. Some special computer languages such as [Cohen et al. 83, Klatte et al. 92] give the programmer control over precision and cancelation. LeLisp is based on on continued fractions [Vuillemin 90], while the program described in [Boehm et al. 86] is based on a form of on-line arithmetic.

In [Wiedmer 80], a method is described by which abstract symbolic manipulations can be used to exactly manipulate values which have infinite representations in conventional form. In [Schwartz 89], a C++ library is presented which allows results to be evaluated to arbitrary precision. Numbers are represented in two parts: a data value, which corresponds to the already known bits of the number, and an expression. When more bits are required, the expression is manipulated to generated the required bits.

A common component of many credible and accurate programs is variable-precision arithmetic. The reason for this is discussed in [Section 2]. Based on this observation, G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula, argue that certain hardware hooks should be added to microprocessors in order to make variable-precision arithmetic more efficient [Bohlender et al. 91]. We take this a step further, and describe a set of microprocessor instructions which implement high radix on-line arithmetic. These instructions can be implemented by simple extensions to conventional microprocessor architectures, and they are well suited for very long instruction word (VLIW) and super-scalar techniques.

On-line arithmetic performs operations serially most significant digit first [Ercegovac 84, Ercegovac 91, Irwin and Owens 87, Muller 94], [Duprat and Muller 93, Bajard et. al 94]. This is possible because of the redundancy in the underlying signed digit representation. On-line operations conceptually operate on arbitrarily long digit streams, and as a consequence changing or mixing precisions is straight forward. Most significant digit first variable-precision techniques were successfully used on the AMD K5(tm) microprocessor for implementing accurate transcendental functions on a processor with a narrow data path [Lynch et al. 95].

This paper presents an efficient method for performing credible and accurate computation through the use of high radix on-line arithmetic. The relationship between credible, accurate arithmetic and variable-precision arithmetic is discussed in [Section 2]. [Section 3] discusses our method for performing high radix on-line arithmetic using sequences of three operand microprocessor instructions. Hardware designs for a significand adder unit and significand multiplier unit are discussed in [Section 4] and [Section 5], respectively. High radix on-line floating-point algorithms are discussed in [Section 6], followed by conclusions in [Section 7]. This paper is an extended version of the research presented in [Lynch 95].

2 Range Expansion and Error

The goal of this section is to establish the relationship between credible, accurate arithmetic and variable-precision calculations. We start by quantifying the limitations of conventional floating-point representations as a function of precision. We then show how these representation limitations interact with the behavior of accurate computer approximations. We conclude the section with an example of how relative error can be controlled by using variable-precision arithmetic.

The distance between neighboring representable values around the point x in a conventional binary floating-point system is:

$$\Delta(x,p) = 2^{\lfloor \log_2 |x| \rfloor - (p-1)} \tag{1}$$

where p is the precision of floating-point numbers in the system. This function bounds the minimum worst case error that may be introduced by a computer approximation of any continuous function with a range that spans at least two representable values. If the rounding mode is round-to-nearest, then the worst case absolute representation error in a neighborhood around x is:

$$\frac{1}{2}\Delta(x,p) \tag{2}$$

and the worst case relative error is:

$$\frac{1}{2}\Delta(x,p)/x \tag{3}$$

The distance between neighboring result values of a perfect computer approximation of a continuous function, f, around a point x is closely described as:

$$\delta(f,x,p_{in}) = \Delta(x,p_{in})\frac{df(x)}{dx} \tag{4}$$

where p_{in} is the input precision. The distance between representable result values with a precision of p_{out} around $f(x)$ is $\Delta(f(x),p_{out})$. The ratio of $\delta(f,x,p_{in})$ to $\Delta(f(x),p_{out})$ is:

$$\eta(f,x,p_{in},p_{out}) = \frac{\delta(f,x,p_{in})}{\Delta(f(x),p_{out})} \tag{5}$$

This ratio is a measure of how well round-to-nearest of $f(x)$ maps continuous values of x into a floating-point representation of precision p_{out} without considering the effects of approximation error. We call this ratio the *range expansion* because when this ratio is greater than one (or $1/2$ at exponent boundaries), all values belonging to the floating-point representation of precision p_{out} cannot be produced by $f(x)$. Many representable values fall in between neighboring output values and hence there is ambiguity in determining the correct output value.

This point is illustrated further in [Figure 1]. This figure shows that for a perfect approximation of $e^x; e^x \in [1000.000, 1111.111]$, where $p_{in} = 7$ and $p_{out} = 7$, there are multiple representable values between possible output values. In this example, the function was rounded to nearest as though it had been calculated to infinite precision, yet input representation error, which is guaranteed to be present, causes a worst case error of 7 ulps on the output. If equation (5) had been applied to this example to force a small range expansion by raising the input precision, this effect would not occur, as there would not be multiple representable values between output values.

By definition, a credible and accurate computer approximation has a guaranteed output accuracy. The output precision can be set from the output accuracy since it is not helpful for the distance between produced values to be much

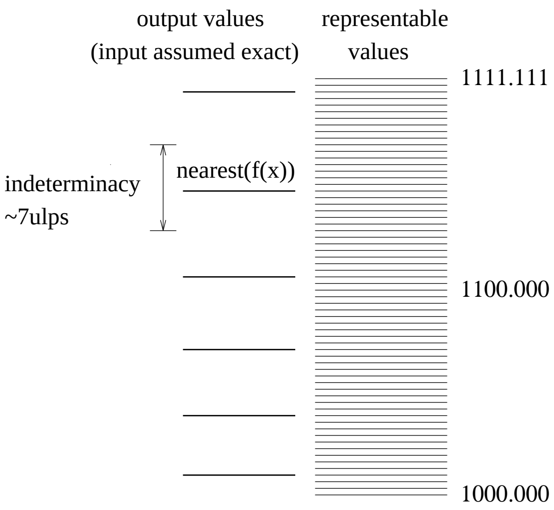


Figure 1: Range Expansion of e^x With a 7 Bit Precise Input

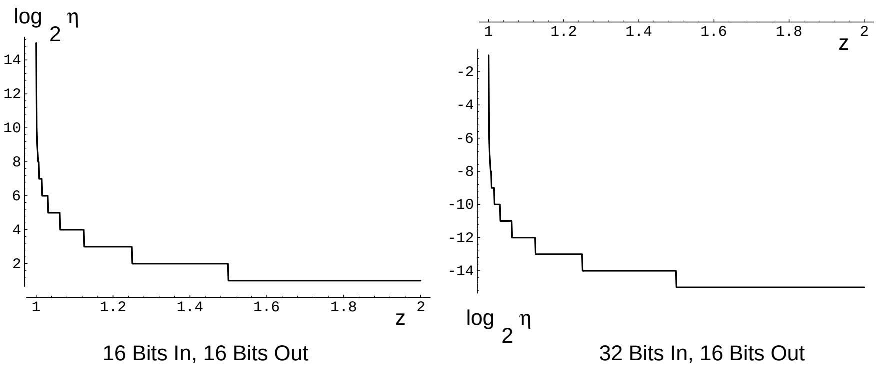
smaller than the dominate output error. By using equation (5), this output precision implies an input precision, which implies an input precision for the next operation back, etc. When precision is set in this way, the accumulation of small rounding errors occur at the end of the word, and therefore are of the order of the representation error, $\Delta(x, p)$. This suggests that error propagates into the values like carries from the bottom of the word, at a rate of $O(log(m))$, where m is the number of floating-point operations.

[Figure 2] shows the range expansion for the function $z-1$ for $z \in [1+2^{-15}, 2]$. The plot on the left shows a worst case factor of 2^{15} more distance between result neighbors than between representable neighbors. The plot on the right shows that the factor can be reduced to a worst case of $\frac{1}{2}$ by adding 16 more bits to the input precision.

The technique of using variable-precision arithmetic to reduce error is illustrated further by coding the function

$$f(z) = \frac{\sqrt{z} + 1}{\sqrt{z} - 1} \tag{6}$$

in both fixed precision and variable-precision arithmetics. A supplemented version of the language "Mathematica" [Wolfram 91] is used for coding this example. The code segment on the left is a 16 bit fixed precision implementation. The code on the right is a variable precision version with a range expansion of at most one half for each operation. The function *round* rounds the first operand to nearest using the precision specified by the second operand. The accuracy goal for this transformation is a relative error of 2^{-12}. Its domain is $z \in [1 + 2^{-15}, 2]$.



16 Bits In, 16 Bits Out

32 Bits In, 16 Bits Out

Figure 2: Range Expansion of $z-1$ for $z \in [1+2^{-15}, 2]$

```
f_fixed[z_] := Module[ {},
      zp = round[z,16];
     zsq = round[Sqrt[zp], 16];
       n = round[zsq + 1, 16];
       d = round[zsq - 1, 16];
     nsd = round[ n/d, 16];
     Return [nsd]
]
```

```
f_variable[z_] := Module[ {},
      zp = round[z,31];
     zsq = round[Sqrt[zp], 32];
       n = round[zsq + 1, 16];
       d = round[zsq - 1, 16];
     nsd = round[ n/d, 14];
     Return [nsd]
]
```

In the variable precision code, an output precision of 14 gives two guard bits for roundoff error accumulation. The numerator and denominator sums are computed to an output precision of 16 bits to minimize the roundoff error accumulated in the subsequent divide, and to make up for the small input alignment shifts in the add. The square root is performed with an output precision of 32 bits so that the input precision to the subsequent $zsq - 1$ will be sufficient to ensure a range expansion of at most one half. The input precision to the square root is set to 31 bits to give a satisfactory range expansion.

Here are the results of evaluating each of these code segments for the input value 3072/3071. Values are given in hexadecimal.

$$z = 1.0015571\text{c}97\text{b}74\text{f}469\text{b}3\ldots \tag{7}$$

$$f(z) = 2\text{ffd.fffaaa}71\text{c}42\ldots \tag{8}$$

$$f_{\text{fixed\_precision}}(z) = 3334.0 \tag{9}$$

$$f_{\text{interval}}(z) = [2\text{aab}.4, 4001.8] \tag{10}$$

$$f_{\text{variable\_precision}}(z) = 2\text{ffe}.0 \tag{11}$$

The fixed precision result has only one significant bit. The interval result (16 bit calculation) has less than one significant bit, but it does contain the exact result. Only the variable-precision code produces a result with a small relative error. [Figure 3] shows scatter plots of the relative error in the fixed precision and variable-precision code. The variable-precision code has an even relative error, bounded by about 2^{-13}. The fixed precision code has unbounded error, which becomes large as z approaches 1.

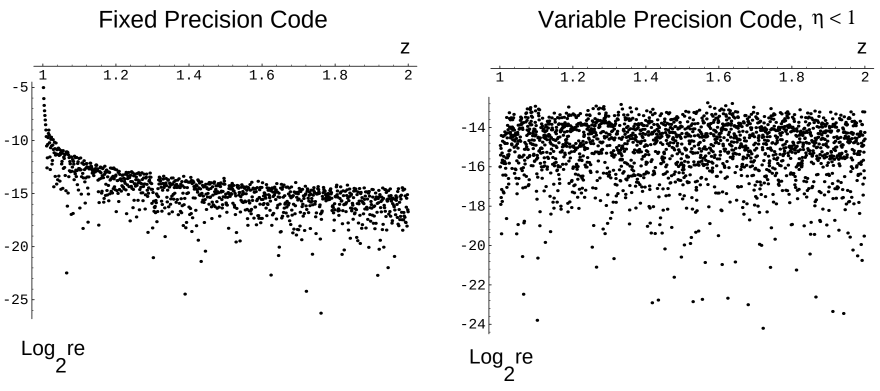


Figure 3: Log Relative Error of $f(z)$ for $z \in [1 + 2^{-15}, 2]$

3 High Radix On-line Arithmetic

In [Bohlender et al. 91], G. Bohlender, W. Walter, P. Kornerup, and D. W. Matula suggest hardware features which make variable-precision arithmetic easier to implement. They suggest that adders should return the rounded sum and the bits shifted off during alignment, that multipliers should return the upper and lower bits of the product, and that dividers should return the quotient and the remainder. The instruction specifications for add, subtract, multiply, and divide using this technique are:

```
add c,d ; a,b
sub c,d ; a,b
mul c,d ; a,b
div c,d ; a,b
```

where a and b are source registers and c and d are destination registers.

There are two main disadvantages of implementing these instructions on a general purpose microprocessor. First, most microprocessors' instruction sets allow at most three operands and would be unable to support these four operand instructions. Second, the internal control of most microprocessors works on the principal of one destination operand per instruction, while these instructions each have two.

To overcome these disadvantages, we propose the use of a high radix on-line arithmetic. The high radix on-line arithmetic is implemented as a sequence of three operand instructions, with two source operands and one destination operand. These instructions require no extraordinary timing, instruction formats, or decoding. Hence they are sufficient primitives for implementing an efficient variable-precision arithmetic congruent with modern architectures. They have the added advantage of being simpler to implement than their conventional fixed-precision, floating-point counter-parts. The sets of instructions for implementing high radix on-line addition and multiplication are:

```
    add_init    spill,  a_0, b_0
  add_extend  y_{i-1},  a_i, b_i
add_complete      y_n, null, null

    mul_init     null,  a_0, b_0
  mul_extend  y_{i-1},  a_i, b_i
mul_complete      y_n, null, null
```

Sequences of these instructions are combined to perform high radix on-line arithmetic. For example, the instruction sequences for implementing four digit high radix on-line addition and multiplication are:

```
add_init     spill,  a_0,  b_0
add_extend     y_0,  a_1,  b_1
add_extend     y_1,  a_2,  b_2
add_extend     y_2,  a_3,  b_3
add_complete   y_3, null, null

mult_init      null,  a_0,  b_0
mult_init      null,  a_1,  b_1
mult_extend     y_0,  a_2,  b_2
mult_extend     y_1,  a_3,  b_3
mult_complete   y_2, null, null
mult_complete   y_3, null, null
```

An n digit plus n digit high radix on-line addition consists of one *add_init* instruction, $n-1$ *add_extend* instructions, and one *add_complete* instruction. An n digit by n digit high radix on-line multiplication consists of two *mult_init* instructions, $n-2$ *mult_extend* instructions, and two *mult_complete* instructions. For these instructions, each digit is a machine word. Machine integers work naturally as signed digits, and they are supported directly in the processor's data paths, caches, memory busses, etc. Each instruction executes in one machine cycle until the execution unit runs out of some resource such as multiplier width or operand register width, as discussed later.

Since high radix on-line arithmetic produces results most significant digit first, separate code segments may be pipelined. For example, it is not necessary to wait for a series of digit adds to complete before starting a subsequent series of digit multiplies. This can be used to speed up operations on a super-scalar processor where multiple execution units are available simultaneously. In a super-scalar design, implementing a number of small units has advantages over using one large unit, because the small units fit in the integer data path, do not limit clock periods, and can also be used for integer instructions. For example, a 64 by 64 bit multiplier may only be partially utilized by programs that perform 32 bit arithmetic, and such programs may even stall for lack of multiplication resources. On the other hand four 32 by 32 bit multipliers require approximately the same total die area, but can be better utilized.

According to this method a floating-point number x is represented as a string of $n_x + 1$ signed integers $(e_x; x_0, x_1, \ldots, x_{n_x - 1})$, where e_x is the exponent of x and x_i is the *ith* significand digit. The value of x is:

$$x = \sum_{i=0}^{n_x - 1} x_i \cdot r^{e_x - i}$$

where r is the radix of the number system. If each digit is a k-bit signed integer, then $r = 2^{k-1}$. Increasing n_x increases the precision of x, which allows variable-precision computations to be performed. Integer arithmetic can be efficiently performed as single-digit operations on the same hardware.

A signed-digit, floating-point number can have multiple representations. For example, if the digits are decimal, the number 1024 can be represented as $x = (3; 1, 0, 2, 4)$ or equivalently $x = (3; 1, 1, -8, 4)$. Conversion from signed-digit notation to conventional notation is accomplished by subtracting the negative digits from the non-negative digits, as shown below.

```
     (1, 1, 0, 4)
  -  (0, 0, 8, 0)
  ---------------
     (1, 0, 2, 4)
```

4 Adder Significand Unit

The functionality of the adder significand unit is described here using C++ classes. These classes can be viewed as hardware behavior models. The declaration for the adder significand unit is:

```
class adder{
public: //three instructions
    int  initial( int a, int b);
    int  extend( int a, int b);
    int  complete();
protected:
    int keepsum;
};
```

Each of the C++ class methods performs the same function that a hardware unit would perform if it received an analogous instruction. For example, calling the method *initial* with two values, a and b is analogous to sending the instruction opcode for *add_init* along with the values a and b on the operand busses to the adder significand unit.

With the proposed method, all digits are streamed through the same functional unit. The signed digit adder carries the sums to the right instead of propagating carries to the left. The sum carried to the right is called *keepsum* in the code. This sum is stored in a register in the add unit.

In the case of a VLIW machine, it is easy to stream instructions through a specific unit, since the instruction sequences may be placed into the correct unit's decode slot. However, for a super-scalar microprocessor there is a problem since the usual dependency checking hardware will not see dependencies between related initial, extend, and complete instructions. A solution to this problem is to code related initial, extend, and complete instructions consecutively in the instruction stream and have the decode unit treat them specially by placing them in an instruction queue in front of the appropriate execution unit.

The *initial* instruction resets the unit's state in preparation for a new sequence of digits. It sets *keepsum* to zero, calculates the most significant sum digit, and returns the value of *spill*. The value of *spill* is zero, unless the most significant sum digit produces a carry. Carry from the most significant digit is a special condition, since the exponent calculation is affected. This event should be fairly rare, since the likelihood of carry when adding full word integers is low.

```
int adder::initial( int a, int b ){ //on-line delay of one
   int spill;
   keepsum = 0;
   return spill = extend(a, b);
}
```

The *extend* instruction outputs a new sum digit by adding a new transfer digit t to the *keepsum* calculated in the previous iteration. It also calculates a new *keepsum* digit to be used in the next iteration.

```
int adder::extend( int a, int b ){
  long long dig_sum;           // this is larger than int
        int adj_sum;
        int t;
        int sum;
        int result_sum;

  dig_sum = (long long)a + (long long)b;//add two digits
  if( dig_sum >= DIG_MAX ){             //check for carry
     t = 1;
     adj_sum = dig_sum - DIG_MAX - 1l;  //may go negative
  }else
  if( dig_sum <= -DIG_MAX ){            //check for borrow
     t = -1;
     adj_sum = dig_sum + DIG_MAX + 1l;  //may go positive
  }else{
     t = 0;
     adj_sum = dig_sum;
  }
  result_sum = keepsum + t;
  keepsum = adj_sum;
  return result_sum;
}
```

The last sum digit is returned directly by issuing an *add_complete* instruction.

```
int adder::complete(){
```

```
    return keepsum;
}
```

5 Multiplier Significand Unit

The interface to the multiplier significand unit is similar to the interface to the adder significand unit. The multiplier state consists of two operand registers, a partial product accumulator, and a digit counter which points into the operand registers. The class definition for the multiplier significand unit is shown below.

```
class mul{
 public:
     void initial( int a, int b);
      int extend( int a, int b);
      int complete();
 protected:
      word Xi,Yi; //operand registers:
      word running_product; // a partial product accumulator
      int i; // digit counter
};
```

The used portion of the two operand registers increases as new digits are introduced. This places a limitation on the number of digits that can be multiplied. When the internal state is saturated, a program can scan out the internal partial remainder value by issuing *mult_complete* instructions. This value can then be used to extend the operation.

In order to support internal state manipulations in the multiplier, we introduced maximum word length operations for addition, shifting, and word by digit multiplication. The C++ definition for the *word* class is given below.

```
class word {
 public:
    word( char * d0, ...);
    word();
    word(int);
   ~word();
    void print();
     int & operator[] (unsigned int index);
    word simplify();
    word operator-();
    word operator+( word b);
    word operator-( word b);
    word operator>>( int count );
    word operator<<( int count );
    word word_by_digit( int digit );
 protected:
    int digit[DIGS];  // DIGS is the register width
};
```

The algorithm we use is basically that presented by Trivedi and Ercegovac [Trivedi and Ercegovac 75]. The introduction of new operand digits at each step results in a new row and a new column in the partial product matrix. Carries created by this addition of new rows and columns are limited in duration, and so it is possible to return the leading partial product digit after two new row/column sums are added. The rows and columns are produced with digit by word multiplies. In high radix on-line arithmetic a digit by digit multiplier, or perhaps a digit by a few digits multiplier will be the largest practical unit, so the digit by word operation will become slower as the number of operand digits becomes larger. This causes the result digit latencies to grow as the number of input operands becomes larger.

The *initial* instruction sets the digit counter, i, to zero and computes the most significant partial product digit. The result busses are not driven, since this digit may need to be adjusted in the next iteration.

```
void mul::initial( int xx, int yy){ //on-line delay of two,
   i = 0;
   extend( xx, yy);
}
```

The *extend* instruction produces a new product digit. Initially, it computes a new row and column of the matrix by multiplying the new digit of x by the previous digits of y and the new digit of y by the previous digits of x. The new row and column are added to the running partial product and the leading digit of the running partial product is returned.

```
int mul::extend( int xx, int yy){
  // overflowed our state?
    if( i >= DIGS ){
       fprintf(stderr,"digit overflow\n");
       abort();
    }
  // shift the new digits into the operand registers
    Yi[i+1] = yy;  // Yi is real state
    word Xim1;
    Xim1 = Xi;  // Xi is real state
    Xi[i+1] = xx;
  // add in the new row and the new column to the matrix
    word        trap_row =   Yi.word_by_digit(xx);
    word        trap_col = Xim1.word_by_digit(yy);
    word partial_product = add_nr(trap_row, trap_col);
    running_product = add_nr(running_product, partial_product);
  // extract leading digit of the running product
    word S = add_nr( abs_word(running_product) , half);
    int dj = sign(running_product) * S[0];
  // add cancels the lead digit
    running_product = add_nr(running_product, -dj);
    running_product = running_product << 1;// walks to the left
    i++;
    return dj;
}
```

Two *mult_complete* instructions are used to return the last two product digits.

```
int mul::complete(){
    return extend(0,0);
}
```

6 Floating-point Algorithms

The high radix on-line floating-point algorithms presented in this paper are similar to those given in [Watanuki and Ercegovac 81]. However, these operations have been partitioned into microprocessor instruction sequences. Overflow or cancelation from the leading digit should be fairly rare, because of the large radix. Hence, the exponent and significand operations are somewhat independent. The programmer specifies the steps in the operation explicitly. The following shows the assembly code for a five digit floating-point multiply. The *jnz* instruction should be coded so that it is predicted to be taken by the branch prediction unit. When the branch is found to be not taken, the processor will delete the speculative state. The *jnz* instruction is placed at the end of the sequence so that the decoder gets the multiply instructions to the multiplier as soon as possible. It could be moved up a little to fill delay slots caused by the multiplier getting slower with the introduction of operand digits.

```
 add           exp_y, exp_a, exp_b // may get ov trap
 mult_init      null,   a_0,   b_0 // perform multiply
 mult_init      null,   a_1,   b_1
 mult_extend     y_0,   a_2,   b_2
 mult_extend     y_1,   a_3,   b_3
 mult_extend     y_2,   a_4,   b_4
 mult_complete   y_3,   null, null
 mult_complete   y_4,   null, null
 mult_test       norm,  y_0,  null  // test for normalized result
 jnz             done,  norm, null
    // add program personality here;
done: exit
```

The *program personality* code is seldom executed, since it is only looking for three out of 2^k cases, where k is the number of bits per integer. This is a benefit of using a high radix. Hence, we can normally ignore the exponent adjustment, and performance does not suffer. No additional hardware structures beyond those needed for the significand unit are required for performing high radix on-line floating-point multiply.

For addition, the operand alignment step does not require a real shifter, since the operand with the smaller exponent is simply delayed. We decided to perform the exponent subtract and the operand delay inside the significand adder unit. Otherwise floating-point addition code sequences require the declaration of an index variable, and the use of an *if* block and *while* loop. This decision causes the need for an operand register in the unit. The operand register width grows with alignment delay as operand digits are introduced. Operand length limits are already caused by the operand registers in the multiplier, so this situation is tolerable in the adder. The proposed assembly code for a four digit on-line addition is shown below.

```
      add_exponent  exp_y, exp_a, exp_b
      add_init      spill,   a_0,   b_0
      add_extend      y_0,   a_1,   b_1
      add_extend      y_1,   a_2,   b_2
      add_extend      y_2,   a_3,   b_3
      add_complete    y_3,  null,  null
      add_test        spc,  y_0,  spill // test for special cases
      jnz             done, spc,  null
         // add program personality here;
done: exit
```

The add_exponent and the add_init instruction can be executed in parallel in single cycle. Each add_extend instruction the requires another cycle. Finally the add_test instruction does not effect performance since it is done after the add is complete – unless the normalization or carry test fails. In which case the fix-up code must be executed. Hence, in the usual case there is an online delay of one cycle followed by one cycle to calculate each digit, for a total of 5 cycles for this add. The adder unit is unavailable for an additional cycle which the test is being performed.

The additional area requirements over that needed for an integer unit is modest. The operand registers and the instruction queue are the largest items. Also these units will fit in an integer data path (unlike most conventional floating-point units). The hardware for the on-line floating-point multiplier unit is:

1. two word width operand registers
2. a word width partial product register
3. a digit counter
4. a digit by word multiplier
5. a word width adder
6. an instruction queue

The hardware needed for the on-line floating-point adder is:

1. a word width operand register
2. a digit width register
3. a digit width adder
4. an instruction queue

The instructions presented can be used to implement various data types by filling in the *personality* sections, and by varying the number of extend instructions. For example, arbitrary width floating-point data types can be easily generated by the compiler by varying the number of extend instructions generated for each type. Language support is as simple as passing a parameter into the type declaration. In C one might imagine a statement such as:

```
rbfloat x(3), y(4);
```

which declares two high radix on line floating-point variables x and y with significand lengths of 3 words and 4 words, respectively.

When performing addition or multiplication, there may be need for normalization, as can be determined by looking at the most significant digit. If normalization is needed, there are a variety of options. The simplest (other than doing nothing) is to adjust the exponent, and then shift the significand digits to the left. The action taken depends on the program's requirements.

We believe that these instructions can also be used with backtracking to produce guaranteed precision results as described by Boehm, et al. [Boehm et al. 86]. When cancelation occurs, the *personality* section would trigger the lazy evaluation of previous operations to fill in the lost significance.

7 Conclusions

The propagation of numerical errors can ameliorated by limiting the disparity between a) the distance between values produced by an operation, and b) the distance between possible representable values. Also, this disparity, *range expansion*, can be controlled through the use of variable precision, multi-precision, arithmetic.

Such an arithmetic is *high radix online* arithmetic. This variation of online arithmetic differs in that the radix is set so that digits are machine integers, and operations are executed on a programmed microprocessor instead of on dedicated hardware.

The proposed high radix on-line execution units have many advantages over conventional floating point execution units: they requires less area than the conventional floating-point hardware; they can perform integer operations; they fit into an integer data path; and they can be used to efficiently implement accurate and credible floating point arithmetic.

References

[Alefeld 83] Alefeld, G.; Herzberger, J.: "An Introduction to Interval Computations"; Academic Press, New York, 1983.

[Bajard et. al 94] J. C. Bajard, J. Duprat, S. Kla, and J. M. Muller.: "Some Operators for On-line Radix 2 Computations"; Journal of Parallel and Distributed Computing, pp 336-345, vol. 22, 2, Aug 1994.

[Boehm et al. 86] Boehm, H., Cartwright, R., Riggle, M., O'Donnell, M.: "Exact Real Arithmetic: A Case Study in Higher Order Programming"; ACM 0-89791-200-4/86/0800-0162.

[Bohlender et al. 91] Bohlender, G., Walter, W., Kornerup, P., Matula, D.: "Semantics for Exact Floating Point Operations"; Proceedings 10th Symposium on Computer Arithmetic, IEEE Computer Society Press, Grenoble, France, 1991, 22-26.

[Bohlender 90] Bohlender, G.: "What Do We Need Beyond IEEE Arithmetic?"; Computer Arithmetic and Self-Validating Numerical Methods Academic Press, New York, 1990, 1-32.

[Cohen et al. 83] Cohen, M., Hull, T., Hamacher, V.: "CADAC: A Controlled Precision Decimal Arithmetic Unit"; IEEE Transactions on Computers, C-32, 4, 1983, 370-377.

[Duprat and Muller 93] J. Duprat and J. M. Muller.: "The Cordic Algorithm: New Results for Fast VLSI Implementation."; IEEE Transactions on Computers, pp 168-178, vol. 42, 2, Feb 1993.

[Ercegovac 84] Ercegovac, M.: "On-line Arithmetic: an Overview"; SPIE, Real Time Signal Processing VII, 1984, 86-93.

[Ercegovac 91] Ercegovac, M.: “On-line Arithmetic for Recurrence Problems”; Advanced Signal Processing Algorithms, Architectures, and Implementations II, SPIE-The International Society for Optical Engineering, 1991.

[IBM 86] IBM: “IBM High-Accuracy Arithmetic Subroutine Library (ACRITH)”; General Information Manual, GC 33-6163-02, IBM Deutschland GmbH (Department 3282, Schönaicher Strasse 220, 7030 Böblingen), 3rd edition, 1986.

[IEEE 85] American National Standards Institute / Institute of Electrical and Electronics Engineers: “A Standard for Binary Floating-Point Arithmetic”; ANSI/IEEE Std. 754-1985, New York, 1985.

[Irwin and Owens 87] Irwin, M. and Owens, R.: “Digit-pipelined Arithmetic as Illustrated by the Paste-Up System: A tutorial”; IEEE Computer, 1987, 61-73.

[Klatte et al. 92] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: “PASCAL–XSC - Language Reference with Examples”; Springer-Verlag, Berlin/Heidelberg/New York, 1992.

[Knuth 81] Knuth, D.: “The Art of Computer Programming: Seminumerical Algorithms”; Vol. 2, 2nd ed., Addison-Wesley, Reading, MA, 1981.

[Lynch et al. 95] Lynch, T., Ahmed, A., Schulte, M., Callaway, T., Tisdale, R.: ”The K5 Transcendental Functions”; Proceedings of the 12th Symposium on Computer Arithmetic, IEEE Computer Society Press, Bath, England, 1995.

[Lynch 95] Lynch, T.: ”High Radix Online Arithmetic for Credible and Accurate General Purpose Computing”; Real Numbers and Computers ... Les Nombre Réels et L’Ordinateur, Ecole des Mines de Saint-Etienne, France, 1995, 78-89.

[Lynch and Swartzlander 92] Lynch, T., Swartzlander E.: “A Formalization for Computer Arithmetic”; Computer Arithmetic and Enclosure Methods, Elsevier Science Publishers, Amsterdam, 1992.

[Moore 66] Moore, R.: “Interval Analysis”, Prentice Hall Inc., Englewood Cliffs, NJ, 1966.

[Muller 94] J. M. Muller.: “Some Characterizations of Functions Computable in On-line Arithmetic”; IEEE Transactions on Computers, pp 752-755, vol. 43, 6, June 1994.

[Nickel 85] Nickel, K.(Ed.): “Interval Mathematics 1985: Proceedings of the International Symposium”; Freiburg 1985, Springer-Verlag, Vienna, 1986.

[Schwartz 89] Schwarz, G.: “Implementing Infinite Precision Arithmetic”; Proceedings of the 9th Symposium on Computer Arithmetic, IEEE Computer Society Press, Santa Monica, CA, 1989, 10-17.

[Trivedi and Ercegovac 75] Trivedi, K., Ercegovac, M. “On-line Algorithms for Division and Multiplication”; Proceedings of the IEEE 3rd Symposium on Computer Arithmetic, IEEE Computer Society Press, Dallas, TX, 1975.

[Vuillemin 90] Vuillemin, J: “Exact Real Computer Arithmetic with Continued Fractions”; IEEE Transactions on Computers, C-39, 8, 1990.

[Watanuki and Ercegovac 81] Watanuki, O., Ercegovac M.: “Floating-point On-line Arithmetic Algorithms”; Proceedings of the 5th Symposium on Computer Arithmetic, IEEE Computer Society Press, Ann Arbor, MI, 1981.

[Wiedmer 80] Wiedmer, E.: “Computing with Infinite Objects,” Theoretical Computer Science, 10, 1980, 133-155.

[Wolfram 91] Wolfram S., ”Mathematica - A System for Doing Mathematics by Computer”; Addison-Wesley, Redwood City, 1991.