SENG 440 Final Report

Employing Convergence Computing Algorithm in Calculating the 16th Root

Team ID: 24
Shaun Lyne (V00814753)
Yuying Zhang (V00924070)

Table of Contents

1.0 Introduction

Our team project is focused on calculating the 16th root of a number using Convergence Computing Method (CCM) and fixed point arithmetic. The CCM algorithm performs a series of additions that eventually converge on the correct answer. This algorithm will be discussed in-depth in section 2.0. Our report will be organized as follows: after explaining the algorithm, we will present our design. There are a few parts to the design, the first being our initial unoptimized solution, and the second part being the final product with optimizations implemented. The accuracy and performance of these two iterations will be compared and the results will be shown. In addition to the optimizations we implemented, there will be an estimation of the potential performance enhancements that would result from a hardware unit encapsulating a portion of the code being introduced.

After the design section, there will be a brief discussion on our results, as well as future enhancements and fixes that could be implemented. Finally, we will present our conclusion. The processor we are using is the ARM machine available via the UVic Engineering lab, and the VM provided for this course. This processor has 32-bit architecture, but still supports 64-bit arithmetic.

1.1 Requirements & Constraints

Requirements	**Constraints**
• Given a range of argument values that require pre-normalization • Determine results for these 20 random values in double precision floating point arithmetic through MATLAB • Determine the bottleneck of the all-software implementation • Define a new instruction that returns the value calculated by the bottleneck • Implement the new instruction(s) in firmware / hardware • Rewrite the high-level code and instantiate the new instruction • Estimate the performance improvement and the penalty of the improved solution	• Use less than 10 variables since there are only 16 registers • Will need 128 bit numbers, which requires the use of arrays to represent individual numbers and a routine to add these numbers together • Range is [1.0, 65536.0] • Calculate $\sqrt[16]{M}$ using fixed point arithmetic, where 1.0 ≤ M < 65536.0, with 64 bits for the integer part and 64 bits for the fractional part.

Table 1.1 - Requirements & Specifications/Constraint

2.0 Background

The use of logarithm and exponential computations are frequently used in signal processing, such as for procedures like determinant calculation, the compensation of non-linear effects and multiplicative noise in wireless communications, as well as Cepstral (spec–tral → ceps–tral) processing in speech processing and recognition applications. However, the direct evaluation of logarithms and exponentials can be computationally demanding since it translates to a sequence of multiplications, additions, and memory look-up operations if the common Taylor series expansion is employed. Therefore, for our project in calculation of the 16th root of a number we applied the convergence computing method (CCM) which is capable of calculating higher-index roots. CCM belongs to the general class of Shift-and-Add algorithms for computing logarithms and exponentials by only using efficient, cheap operations like shifts and additions. It is also sequential in that it is an iterative method.

For the calculation of the 16th root, our problem is defined as follows: Calculate $\sqrt[16]{M}$ where $1.0 \leq M < 65536.0$. In fixed-point representation each integer would be represented as a 128 bit number where M should have 64 bits for the whole part, and 64 bits for the fractional part. If this is not the case, a shift operation will fix the representation. The basic principle of the CCM algorithm is that cyclic multiplication will ensue by a sequence of specially chosen factors A_i^{16} , as needed, until the product falls in a predefined range of (M − Δ,M]. Where the constant Δ specifies the precision of the computation, and after K iterations $M - \Delta < \prod_{i=1}^{K} A_i^{16} \leq M$ and $\sqrt{M - \Delta} < \prod_{i=1}^{K} A_i \leq \sqrt{M}$. Where the factors of A_i can be either equal to 1 or of the form$1 + 2^{-i}$such that a multiplication by A_i will reduce to one addition and one shift, and a multiplication by A_i^{16} reduces to sixteen additions and sixteen shift operations. This logic was applied in deriving a procedure from CCM to calculate the root of any order, specifically the 16th root in this case.

3.0 Design

3.1 Software Design

3.1.1 Overview

Calculation of 16th Root Pseudocode	
$f = 1.0$	√ M with K bits of precision

$f\_sqrt\_16 = 1.0$ $for\ i = 0\ to\ K - 1\ do$: $\mu = f \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 +$ $\cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 +$ $\cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 + 2^{-i}) \cdot (1 +$ $\cdot (1 + 2^{-i})$ $\mu\_sqrt\_16 = f\_sqrt\_16 \cdot (1 + 2^{-i})$ $if\ \mu \leq M\ then$ $f = \mu$ $f\_sqrt\_16 = \mu\_sqrt\_16$ $return\ f\_sqrt\_16$	Potential multiplication by A_i^{16} Potential multiplication by A_i If product is less than M accept iteration. Otherwise reject it (do nothing)

Table 3.1.1.1 - Pseudocode

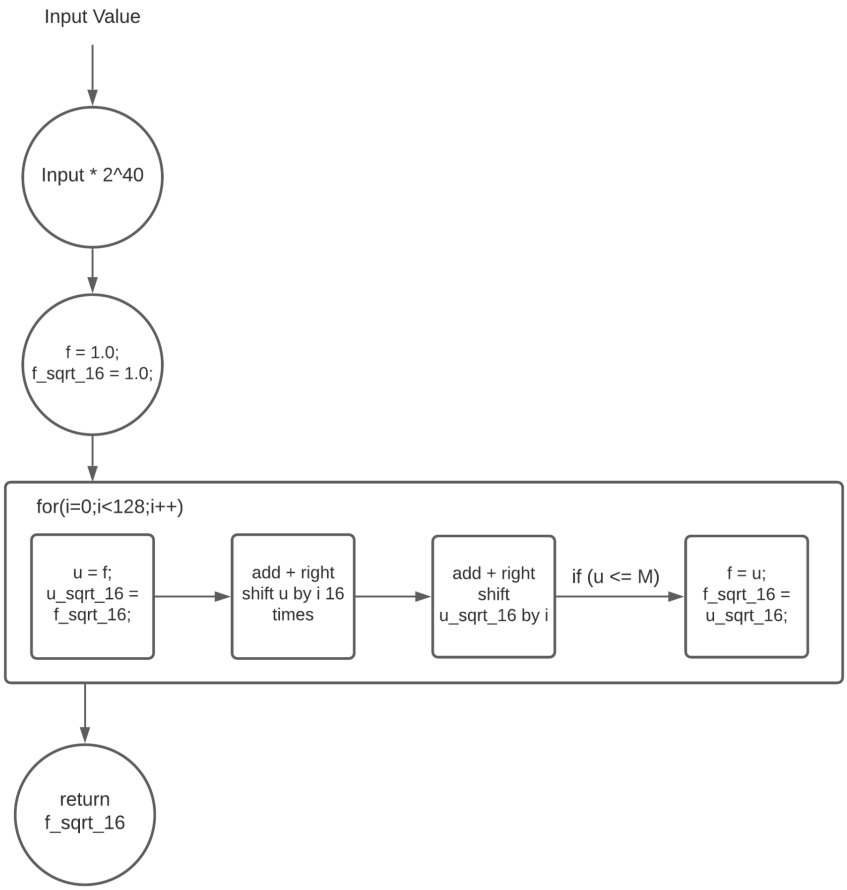


Figure 3.1.1.1 - Flowchart

Our solution runs the CCM algorithm for the 16th root, using 128-bit precision. 128-bit precision is a requirement that was communicated after the initial progress report. In order to facilitate the highest level of accuracy in our solution, we needed to implement a representation for 128-bit integers, since the highest natively supported integer on our processor is 64-bits. The representation that was decided on was the following struct:

```
struct my_uint128_t {
   unsigned long long hi;
   unsigned long long lo;
};
```

This struct utilizes two 64-bit integers (as unsigned long long int) to represent the full 128-bit integer, with the “hi” portion representing the integer part, and the “lo” portion representing the fractional part.

3.1.2 Addition/Shift Routines

In order to conduct the necessary calculations with 128 bit representation, we implemented specialized routines to perform the addition/shift operations that were required in the algorithm. For the 128 bit addition function, it takes in two arguments of struct my_uint128_t, which are passed in by value in order to increase performance since passing by reference would force the GCC compiler to consider aliasing as well. Inside the function another struct my_uint128_t is declared to store the result. First the function adds together the MSB 64 bits which represents the integer portion of the number, this result is stored inside sum.hi. Next the function adds together the LSB 64 bits which represents the fractional portion of the number, this result is stored inside sum.lo. The function then proceeds to check for overflow using the fractional bits of both numbers, should sum.lo be less than x.lo, then sum.hi is incremented to account for the carry. Finally the sum is returned.

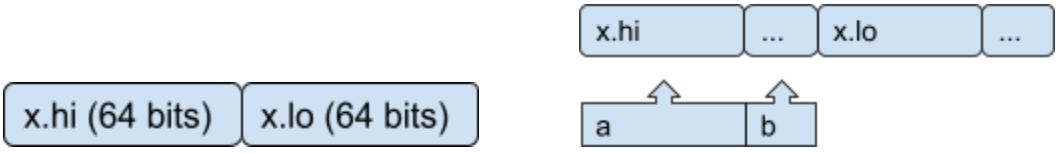
```
my_uint128_t incrementOldVersion(my_uint128_t x, my_uint128_t y)
{
   my_uint128_t sum;
   sum.hi = x.hi + y.hi;
   sum.lo = x.lo + y.lo;
   // check for overflow of low 64 bits, add carry to high
   if (sum.lo < x.lo)
      ++sum.hi;
```

```
    return sum;
}
```

The 128 bit addition function is later revised and optimized in order to remove the conditional branching of the previous function code shown above. In the new 128 bit addition function the result is stored directly in the passed value of struct my_uint128_t x. First the function adds x.lo and y.lo, storing the result in x.lo. Then the function evaluates the comparison of x.lo < y.lo, adds this to y.hi and x.hi, storing the result in x.hi. Finally, x is returned by the function.

```
// 128 bit addition
struct my_uint128_t increment(struct my_uint128_t x, struct my_uint128_t y)
{
   x.lo += y.lo;
   x.hi += y.hi + (x.lo < y.lo);
   return x;
}
```

To accommodate for right/left shifting of 128-bit integer representation numbers, two different functions were created. The function void shiftl128 (struct my_uint128_t *num, size_t s) is responsible for left shifting and takes in two arguments, one the number to be shifted and variable s representing the shift amount. The function void shiftr128 (struct my_uint128_t *num, size_t s) is responsible for right shifting and takes in two arguments, one the number to be shifted and variable s representing the shift amount. The number to be shifted is passed in by reference given that this is a shift operation. To explain the algorithmic operation of this function, we will use the left shift example. The number to be shifted may be visualized in the following (left image is before shifting and right image is after shifting):



Where a is x.hi << s and b is the s number of MSB of x.lo shifted over. Therefore x.lo >> (64-s). Note that when the bits of x.hi is shifted to the left, in order to fill in the space left created on the right end and since x.hi and x.lo are conceptually adjacent to each other, the void left by shifting the bits of x.hi is filled by the bits that are shifted off the end of x.lo. Thus, the expression x.lo >> (64-s) computes the bits that were shifted off of x.lo.

Overall the operation may be represented as x.hi = a | b or x.hi = (x.hi << s) | (x.lo >> (64-s)), and for the fractional portion x.lo = (x.lo << s). The above would be the case if the function was shifting a 64-bit

integer by less than 64 bits. However in the case where the shift happens to be greater or equal to 64 bits (s >= 64), then x.hi will be set to x.lo, and x.lo set to 0, then function is then called recursively upon the number again with the shifted amount decremented by 64 (s-64).

```
// Left shift
void shiftl128 (struct my_uint128_t *num, size_t s)
{
   if (s >= 64) // shifting a 64-bit integer by more than 63 bits is
"undefined"
   {
       num->hi=num->lo;
       num->lo=0;
       shiftl128(num,s-64);
   }
   else
   {
       num->hi = (num->hi << s) | (num->lo >> (64-s));
       num->lo = (num->lo << s);
   }
}
```

The same logic and process is applied by the function shiftr128, which manages right shifts of 128 bit numbers. Overall the operation may be represented as x.lo = (x.hi << (64-s)) | (x.lo >> s), and for the fractional portion x.hi = (x.hi >> s). The above would be the case if the function was shifting a 64-bit integer by less than 64 bits. However in the case where the shift happens to be greater or equal to 64 bits (s >= 64), then x.lo will be set to x.hi, and x.hi set to 0, then function is then called recursively upon the number again with the shifted amount decremented by 64 (s-64).

```
// Right shift
void shiftr128 (struct my_uint128_t *num, size_t s)
{
   if (s >= 64) // shifting a 64-bit integer by more than 63 bits is
"undefined"
   {
       num->lo=num->hi;
       num->hi=0;
       shiftr128(num,s-64);
   }
   else
   {
       num->lo = (num->hi << (64-s)) | (num->lo >> s);
```

```
            num->hi = (num->hi >> s);
        }
    }
```

3.1.3 Conversions

There were some important design decisions that had to be made before the creation of the main loop. The first of these was how were the initial decimal values going to be converted to integers. The solution was simple: multiply the initial value by a factor of 2. However, converting in this way still leaves a margin of error in the fixed point representation. For example, the test bench value of 21.21 multiplied by 2^{40} is 23320641625128.96, which means that the 0.96 is discarded upon conversion to an integer. As the initial value gets lower, the impact of the decimal portion being discarded becomes greater.

Using a lower multiplier will increase the range of our solution (reasoning for this is discussed in Section 3.3.3), but it will also impact the accuracy negatively. Alternatively, using a higher multiplier will give a higher level of accuracy (since the error discussed before is being reduced by a factor of 2^x, where x is the number of bits that the multiplier is being increased by), but reduce the range of input values (reasoning for this is also discussed in Section 3.3.3). In the end, 2^{40} was decided as our initial multiplier, since it makes even the smallest numbers have a very large magnitude upon conversion, lowering the impact on accuracy from the decimal portion being discarded. It also gave us a large range of input values along with high accuracy.

The second design decision that we needed to make were the two scale factors for `f` and `f_sqrt_16`. `f` can get quite large if the first couple iterations are accepted, so its scale factor had to leave enough bits to store `f` without overflowing and causing the final answer to be incorrect. Through testing different scale factors, it was found that higher scale factors didn't have any noticeable effect on the accuracy of the final solutions. Because of this, it was decided that 2^{64} would be used, since this allows `f` to grow a significant amount without overflowing. `f_sqrt_16` represents the final answer, and based on our largest input value, will only ever be in the range [1, 2.8284] (see Section 3.3.3 for more explanation). Therefore only 2 bits are needed to represent every integer number of `f_sqrt_16`. We decided to use a scale factor of 2^{125}, which leaves 3 bits for the final answer. This prevents the final answer from overflowing, and leaves room to increase the range of supported input values in the future.

3.2 Optimizations

3.2.1 Evaluation Criteria

The metric we are using to observe the change in performance that results from each of these optimizations is the average execution time of the program, in milliseconds. We used a Bash script which

utilized a for loop and the UNIX “time” command (which gets the execution time of any shell command), along with a Python script which averaged the output values. The program is run 100,000 times on the ARM virtual machine with the same input value, which keeps the timings as consistent as possible and helps gather a sufficient amount of execution times for the average.

Before any optimization changes, the average execution time was **41.00ms**. Something to take note of is that the changes in execution time are being evaluated with previous optimizations included.

3.2.2 Algorithm Change

The first optimization that was done was modifying the algorithm for increased efficiency. In the pseudocode, f_sqrt_16 is re-calculated for every loop iteration. However, the value of f_sqrt_16 is only used when the iteration is accepted. In order to save on computing time, the calculation for f_sqrt_16 was put after the iteration acceptance logic, which saves unnecessary calculations from being performed.

The change in execution time resulting from this change is shown in the table below:

Before Algorithm Change Execution Time (ms)	**After Algorithm Change Execution Time (ms)**	**Execution Reduction (ms)**	**Execution Reduction (%)**
41.00	42.09	-1.09	-2.66%

Strangely enough, this change seemed to increase the execution time by around 1ms.

3.2.3 Optimizing the For Loops

The next optimization performed was the optimization of the for loop. This optimization changed the main for loop and the for loop for performing the potential multiplications as shown below:

```
// Main for loop
for(i=0; i<128; i++) // before
for(i^=i; !(i&128); i++) // after

// Potential multiplications for loop
for(j=0; i<16; i++) // before
for(j^=j; !(j&16); i++) // after
```

The change in execution time resulting from this change is shown in the table below (including previous optimizations):

Before For Loop Optimization Execution Time (ms)	**After For Loop Optimization Execution Time (ms)**	**Execution Reduction (ms)**	**Execution Reduction (%)**
42.09	41.93	0.16	0.38

This optimization didn't seem to have any significant effect on the average execution time.

3.3.4 Predicate Operations + Removal of Increment Function

The next optimization performed was replacing the conditional if statements within the shiftl128 and shiftr128 functions with predicate operations instead, employing the ternary statement operator. As an example, this is the code before the predicate optimization:

```
// shifting a 64-bit integer by more than 63 bits is "undefined"
if (s >= 64)
{
    num->hi=num->lo;
    num->lo=0;
    shiftl128(num,s-64);
}
else
{
    num->hi = (num->hi << s) | (num->lo >> (64-s));
    num->lo = (num->lo << s);
}
```

This is code after the predicate optimization:

```
// shifting a 64-bit integer by more than 63 bits is "undefined"
(s >= 64) ? (
    num->hi=num->lo,
    num->lo=0,
    shiftl128(num,s-64)
) : (
    num->hi = (num->hi << s) | (num->lo >> (64-s)),
```

```
    num->lo = (num->lo << s)
);
```

After replacing both the conditional statements in the functions with predicates, we then removed the increment function and simply added the addition logic to the addShift function to eliminate unnecessary function calls and reduce the usage of registers. The new addShift function appears as the following:

```
struct my_uint128_t addShift(struct my_uint128_t num, size_t s) {
   struct my_uint128_t num_temp;
   num_temp.hi = num.hi;
   num_temp.lo = num.lo;
   shiftr128(&num, s);
   num_temp.lo += num.lo;
   num_temp.hi += num.hi + (num_temp.lo < num.lo);
   return num_temp;
}
```

Before Predicate Operations + Removal of Increment Function Execution Time (ms)	**After Predicate Operations + Removal of Increment Function Execution Time (ms)**	**Execution Reduction (ms)**	**Execution Reduction (%)**
38.10	36.85	1.25	3.28

These two optimizations had a noticeable effect on the average execution time of the program.

3.3.5 Loop Unrolling

The next optimization performed was loop unrolling. This was done by taking the following portion of code in the for loop:

```
u.hi = f.hi;
u.lo = f.lo;
for(j = 0; j < 16; j++) {
    u = addShift(u, (size_t)i);
}ll
```

```
if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
    u_sqrt_16 = addShift(f_sqrt_16, (size_t)i);

    f.hi = u.hi;
    f.lo = u.lo;
    f_sqrt_16.hi = u_sqrt_16.hi;
    f_sqrt_16.lo = u_sqrt_16.lo;
}
```

And changing it to the following:

```
u.hi = f.hi;
u.lo = f.lo;
for(j = 0; j < 16; j++) {
    u = addShift(u, (size_t)i);
}

if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
    u_sqrt_16 = addShift(f_sqrt_16, (size_t)i);

    f.hi = u.hi;
    f.lo = u.lo;
    f_sqrt_16.hi = u_sqrt_16.hi;
    f_sqrt_16.lo = u_sqrt_16.lo;
}
u.hi = f.hi;
u.lo = f.lo;
for(j = 0; j < 16; j++) {
    u = addShift(u, (size_t)i+1); // loop unrolling
}

if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
    u_sqrt_16 = addShift(f_sqrt_16, (size_t)i+1); // loop unrolling

    f.hi = u.hi;
    f.lo = u.lo;
    f_sqrt_16.hi = u_sqrt_16.hi;
    f_sqrt_16.lo = u_sqrt_16.lo;
}
```

There was also the following change made to the for loop to accommodate the loop unrolling:

```
for(i^=i; !(i&128); i++); // before
for(i^=i; !(i&128); i+=2); // after
```

The resulting change in execution time from this change:

Before Loop Unrolling Execution Time (ms)	After Loop Unrolling Execution Time (ms)	Execution Reduction (ms)	Execution Reduction (%)
36.85	35.94	0.91	2.47

This optimization further reduced the average execution time of the program.

3.3.6 Firmware and Hardware Optimizations

The original assembly code after our C code optimizations generated by arm-linux-gcc has 556 lines. Using firmware optimization, in this case a horizontal machine with 2 issue slots, the assembly code can be reduced to 446 lines since instructions that are not related to each other and adjacent in position can be executed simultaneously. The reduced difference of 110 lines of assembly code does not accomplish much performance improvement given the increase in price of using a horizontal machine. This is especially the case for the functions shiftl128, shiftr128, and addShift since the ARM generated assembly code for these functions contain a majority of load and store operations as well as use of fp register. However, many instructions in the main function benefited from the 2 issue slot design since there were less adjacent references to the same registers for contiguous instructions.

The first hardware optimization was done on the functions shiftl128 and shiftr128, specifically the three line operations on num.hi and num.lo. As an example, for function shiftl128 (responsible for left shifts) the following is the original code:

```
// Left shift
void shiftl128 (struct my_uint128_t *num, size_t s)
{
    // shifting a 64-bit integer by more than 63 bits is "undefined"
    (s >= 64) ? (
        num->hi=num->lo,
        num->lo=0,
        shiftl128(num,s-64)
    ) : (
        num->hi = (num->hi << s) | (num->lo >> (64-s)),
        num->lo = (num->lo << s)
    );
}
```

This was changed into:

```c
// Left shift
void shiftl128 (struct my_uint128_t *num, size_t s)
{
    // shifting a 64-bit integer by more than 63 bits is "undefined"
    if (s >= 64) {
        //num->hi=num->lo,
        //num->lo=0,
        //num = my_func_1(num);
        __asm__ __volatile__ (
            "my_func_1\t%1, %2, %0\n"
            : "=r" (num)
            : "r" (num->hi), "r" (num->lo)
        );
        shiftl128(num,s-64);
    } else {
        //num->hi = (num->hi << s) | (num->lo >> (64-s)),
        //num->lo = (num->lo << s)
        //num = my_func_2(num);
        __asm__ __volatile__ (
            "my_func_2\t%1, %2, %0\n"
            : "=r" (num->hi), "=r" (num->lo)
            : "r" (num)
        );
    }
}
```

Where my_func_1 and my_func_2 are hardware implemented, perhaps through the use of combinational and sequential logic gates given that the operations are purely composed of shifting and assignments. This hardware optimization was similarly done for function shiftr128 (responsible for right shifts), the code replaced was similar to shiftl128 and the same hardware implementations could be reused for both functions. After this hardware optimization the number of generated assembly lines was reduced to 488.

```
#APP
@ 19 "cr.c" 1
        my_func_1       r1, r3, r3

@ 0 "" 2
        str     r3, [fp, #-16]
        ldr     r3, [fp, #-20]
        sub     r3, r3, #64
        ldr     r0, [fp, #-16]
        mov     r1, r3
        bl      shiftl128
        b       .L4
.L2:
        ldr     r3, [fp, #-16]
```

```
#APP
@ 30 "cr.c" 1
        my_func_2       r0, r3, r3

@ 0 "" 2
        ldr     r2, [fp, #-16]
        stmia   r2, {r3-r4}
        ldr     r3, [fp, #-16]
        str     r0, [r3, #8]
        str     r1, [r3, #12]
.L4:
        sub     sp, fp, #8
        ldmfd   sp!, {r4, fp, lr}
        bx      lr
```

The next hardware optimization to be implemented could be to replace the code for 128 bit addition with a hardware function. The number of generated assembly lines was reduced to 436 after this and the previous optimizations. The inline assembly code is as follows:

```
struct my_uint128_t addShift(struct my_uint128_t num, size_t s) {
    struct my_uint128_t num_temp;
    num_temp.hi = num.hi;
    num_temp.lo = num.lo;
    shiftr128(&num, s);
    //num_temp.lo += num.lo;
    //num_temp.hi += num.hi + (num_temp.lo < num.lo);
    //num_temp = my_func_5(num_temp, num);
    __asm__ __volatile__ (
        "my_func_5\t%1, %2, %0\n"
        : "=g" (num_temp.hi), "=g" (num_temp.lo)
        : "g" (num), "g" (num_temp)
    );
    return num_temp;
}
```

Each of these hardware optimizations can be implemented using combinational or sequential circuits since all the executed instructions are focused on addition and shifting.

3.3 Results

3.3.1 Accuracy

Below you can see a table of our solution's answers vs. the floating point solutions. 30 decimal places were used, and almost all of the answers had over 30 decimal places of accuracy.

The average % error for our solution using these values is **0.0000000000000170815%**, which puts our solution accuracy (for the test bench) at **99.99999999999985789%.**

Test Value	Actual Answer	Fixed Point Answer	Accuracy (# Decimal Points)
1	1.000000000000000000000000000000	1.000000000000000000000000000000	30+
21.21	1.210343916863420243856808156124	1.210343916863417135232339205686	13
44	1.266822570064138675505205355876	1.266822570064138675505205355876	30+
69.420	1.303445232982610502148190789740	1.303445232982609613969771089614	13
99.99	1.333513097263667734893033411936	1.333513097263667512848428486905	15
234	1.406293570152046701338122147718	1.406293570152046701338122147718	30+

234.23	1.406379921271535771154503891012	1.406379921271535771154503891012	30+
420.69	1.458806322194733828823132171237	1.458806322194733828823132171237	30+
453.234	1.465615856906956837590882969380	1.465615856906956837590882969380	30+
465.098	1.467984701041839201707261963747	1.467984701041839201707261963747	30+
1143.01143	1.552845188863281933322468830738	1.552845188863281933322468830738	30+
1337.420	1.568164934152385781018779198348	1.568164934152385781018779198348	30+
4573.4	1.693421031539278587274566234555	1.693421031539278587274566234555	30+
6969.6969	1.738604775705355098835980243166	1.738604775705355098835980243166	30+
8000.4983	1.753657652945449729742222189088	1.753657652945449729742222189088	30+
8927.4555	1.765714489573772860708800180873	1.765714489573772860708800180873	30+
30000	1.904671478146056529112684074789	1.904671478146056529112684074789	30+
42069.1337	1.945349882430136734257075659116	1.945349882430136734257075659116	30+
50345.2434	1.967308202942210870745043393981	1.967308202942210870745043393981	30+
655360	2.309563969378916503671916871099	2.309563969378916503671916871099	30+

Table 3.3.1.1 - Test Bench Results

It has been observed that if the number is low (<100) and the amount of decimal places is high (>2) there's a much larger chance that the accuracy will be lower, though there will still be 10+ decimal places of accuracy. The specifics of this phenomena are discussed in Section 3.3 (Limitations), however, it can be assured that the overall accuracy of our solution is extremely high.

3.3.2 Performance

The total change in execution time resulting from our optimizations is shown below:

Before Optimizations (ms)	**After Optimizations (ms)**	**Execution Reduction (ms)**	**Execution Reduction (%)**
41.00	35.94	5.06	12.34

A 12.34% reduction in the execution time has been observed. If the program is going to be run for an hour, this would result in 100,167 executions vs. 87,805 executions for the unoptimized solution, with a 12,362 execution difference.

3.3.3 Limitations

The solution has a few limitations, mainly the range of numbers which it can calculate the 16th root of. Any number below 1 will return 1, which is a restriction of the algorithm itself (Section 1.1). However, our solution will work on values above 65536, which is outside of the given range for the algorithm. The highest number that our solution works on is limited by the bottom portion of the my_uint128_t struct, since in order to convert from decimal to integer, the initial value is multiplied by 2^{40}, which leaves 24 bits for the number.

This puts the upper limit of our solution at **16777216**, which could be increased if a method that accounted for overflow and distributed the bits correctly into both the lo & hi portions of the struct were implemented.

```
16777216
2.828427124746190290949243717478
[root@localhost project]# ./run128
20000000
2.828427124746190290949243717478
```

Figure 3.3.3.1 - Upper Limit Demonstration

The second limitation is the number of decimal places in the input value for the 16th root calculation. As mentioned before, the lower the input value and the higher the amount of decimal places, the less accurate the final solution will be.

```
shaunlyne@shaunlyne-Z370-AORUS-Gaming-7:~/Desktop/rand$ python3 16root.py
1.000000000093485793485793485793485579
1.000000000005842881733997273840941488742828369140625000000000000000000000
shaunlyne@shaunlyne-Z370-AORUS-Gaming-7:~/Desktop/rand$ ./run_ccm
1.000000000093485793485793485793485579
1.000000000005798028723802417516708374023437500000000000000000000000000000
```

Figure 3.3.3.2 - Decimal Place Demonstration

In this example, our fixed point solution (run_ccm) has 12 decimal places of accuracy compared to the floating point solution (16root.py).

4.0 Discussion

4.1 Future Work

4.4.1 Accuracy

As shown before, the accuracy of this solution is very high. However, there's always room for improvement, and the accuracy of our solution is no exception. Using a different representation for a higher bit width integer, the precision of the CCM algorithm could be increased from 128-bits to an even higher number. This would heavily mitigate the phenomena described earlier, where low values with high decimal places had lower accuracy.

4.4.2 Performance

Though we did manage to reduce the average execution time of the program, there could be more optimization work done. For example, the algorithm itself could be changed to pre-emptively skip iterations in the for loop, by finding the first iteration which actually gets accepted and starting there. More C-level or even assembly level optimizations could be performed as well. However, the effect (if any) these would have on the execution time is unknown.

4.4.3 Hardware Addition

As explained in Section 3.3.6, adding a hardware unit to handle some of our code could have a noticeable impact on the performance of our solution.

5.0 Conclusions

Overall, our solution is highly accurate and performant, as demonstrated by the previous sections. Since there were no concrete metrics given in the requirements for accuracy/performance, we strived for what we thought would best meet the non-existent requirements for these two metrics.

To reiterate, our accuracy was at minimum, 12 decimal places, and many of our test bench values had over 30 places of decimal accuracy. Our performance in terms of average execution time for 100,000 runs was 35.94ms, which was improved from 41ms.

Our solution also leaves room for future improvement, whether that be increasing the range of accepted input values, or increasing the level of accuracy in the calculated answers.

Appendix A

The following code is our solution so far (minus hardware optimizations):

```
#include <stdio.h>
#define SCALE_FACTOR 64
#define SCALE 18446744073709551616.0
#define FINAL_SCALE 4611686018427387904.0

struct my_uint128_t {
  unsigned long long hi;
  unsigned long long lo;
};

// Left shift
void shiftl128 (struct my_uint128_t *num, size_t s)
{
   // shifting a 64-bit integer by more than 63 bits is "undefined"
   (s >= 64) ? (
       num->hi=num->lo,
       num->lo=0,
       shiftl128(num,s-64)
   ) : (
       num->hi = (num->hi << s) | (num->lo >> (64-s)),
       num->lo = (num->lo << s)
   );
}

// Right shift
void shiftr128 (struct my_uint128_t *num, size_t s)
{
   // shifting a 64-bit integer by more than 63 bits is "undefined"
   (s >= 64) ? (
       num->lo=num->hi,
       num->hi=0,
       shiftr128(num,s-64)
   ) : (
       num->lo = (num->hi << (64-s)) | (num->lo >> s),
       num->hi = (num->hi >> s)
   );
}

struct my_uint128_t addShift(struct my_uint128_t num, size_t s) {
   struct my_uint128_t num_temp;
   num_temp.hi = num.hi;
```

```
    num_temp.lo = num.lo;
    shiftr128(&num, s);
    num_temp.lo += num.lo;
    num_temp.hi += num.hi + (num_temp.lo < num.lo);
    return num_temp;
}


int main() {

    double M_init;
    scanf("%lf", &M_init);
    M_init *= 1099511627776.0; // multiply by 2^40

    struct my_uint128_t f;
    f.hi = 1ULL;
    f.lo = 0ULL;
    struct my_uint128_t f_sqrt_16;
    f_sqrt_16.hi = 1ULL;
    f_sqrt_16.lo = 0ULL;
    struct my_uint128_t M;
    M.hi = 0ULL;
    M.lo = (unsigned long long) M_init;

    shiftl128(&f, SCALE_FACTOR - 64);
    shiftl128(&f_sqrt_16, 62);
    shiftl128(&M, SCALE_FACTOR - 40);

    int i,j;
    struct my_uint128_t u;
    struct my_uint128_t u_sqrt_16;

    for(i^=i; !(i&128); i+=4) {
        u.hi = f.hi;
        u.lo = f.lo;
        for(j^=j; !(j&16); j++) {
            u = addShift(u, (size_t)i);
        }

        if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
            u_sqrt_16 = addShift(f_sqrt_16, (size_t)i);

            f.hi = u.hi;
            f.lo = u.lo;
            f_sqrt_16.hi = u_sqrt_16.hi;
            f_sqrt_16.lo = u_sqrt_16.lo;
        }
```

```
        u.hi = f.hi;
        u.lo = f.lo;
        for(j^=j; !(j&16); j++) {
            u = addShift(u, (size_t)i+1);
        }

        if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
            u_sqrt_16 = addShift(f_sqrt_16, (size_t)i+1);

            f.hi = u.hi;
            f.lo = u.lo;
            f_sqrt_16.hi = u_sqrt_16.hi;
            f_sqrt_16.lo = u_sqrt_16.lo;
        }
        u.hi = f.hi;
        u.lo = f.lo;
        for(j^=j; !(j&16); j++) {
            u = addShift(u, (size_t)i+2);
        }

        if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
            u_sqrt_16 = addShift(f_sqrt_16, (size_t)i+2);

            f.hi = u.hi;
            f.lo = u.lo;
            f_sqrt_16.hi = u_sqrt_16.hi;
            f_sqrt_16.lo = u_sqrt_16.lo;
        }
        u.hi = f.hi;
        u.lo = f.lo;
        for(j^=j; !(j&16); j++) {
            u = addShift(u, (size_t)i+3);
        }

        if(u.hi < M.hi || (u.hi == M.hi && u.lo <= M.lo) ) {
            u_sqrt_16 = addShift(f_sqrt_16, (size_t)i+3);

            f.hi = u.hi;
            f.lo = u.lo;
            f_sqrt_16.hi = u_sqrt_16.hi;
            f_sqrt_16.lo = u_sqrt_16.lo;
        }

    }

    double final_hi = (double)(f_sqrt_16.hi);
    double final_lo = (double)(f_sqrt_16.lo) / SCALE;
```

```
    double final_answer = (final_hi + final_lo) / FINAL_SCALE;
    printf("%.30f\n", final_answer);

}
```