SENG 440 Project:
Convergence Computing Method (CCM)

Team ID: 24

Shaun Lyne (V00814753)

Yuying Zhang (V00924070)

Outline

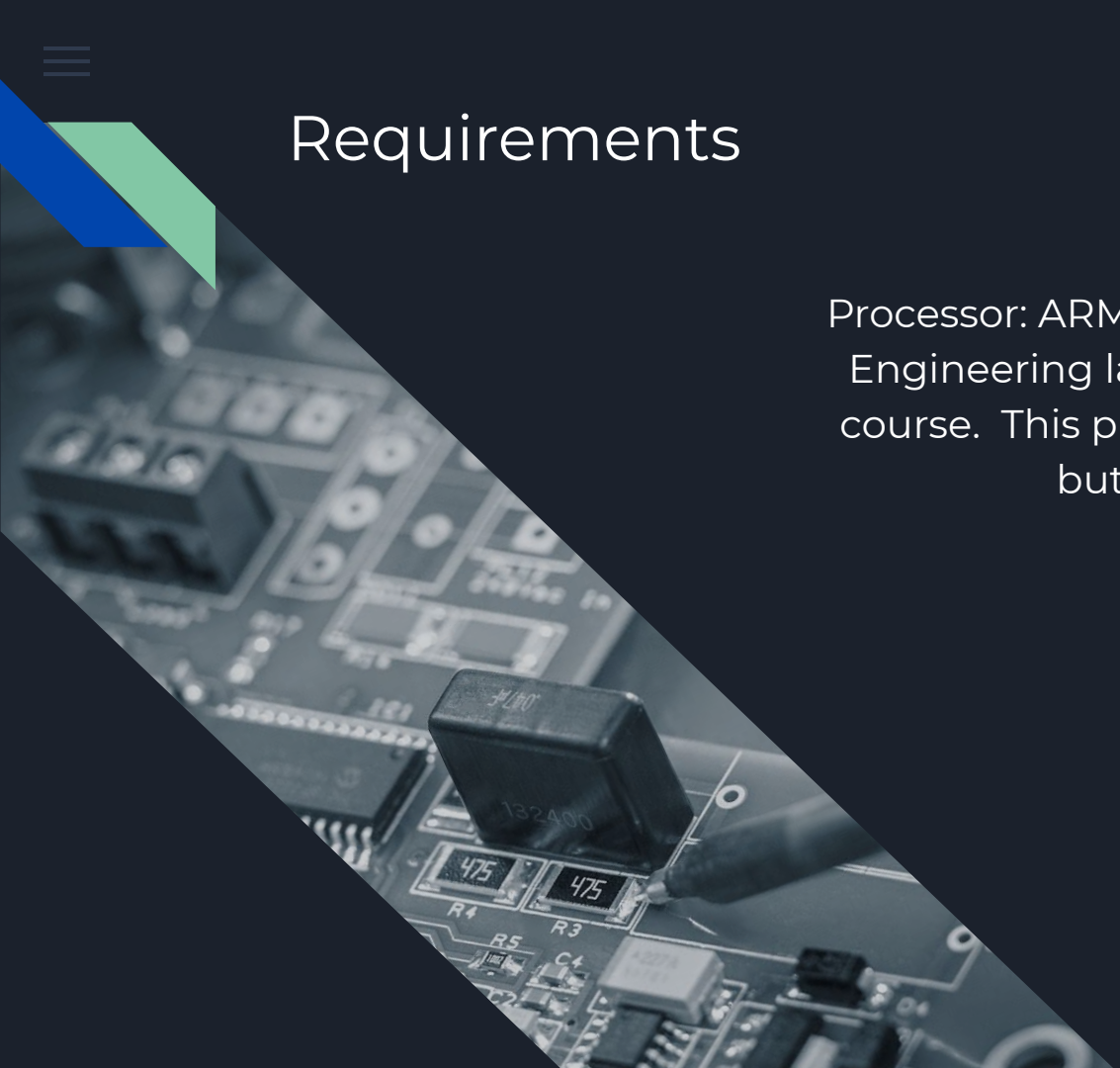
1. Introduction
2. Requirements
3. Understanding the Problem
4. Design
5. Optimization
6. Results

Introduction

Our team's project is focused on implementing an program routine using Convergence Computing Method (CCM) and fixed point arithmetic for calculating the 16th root of a number.

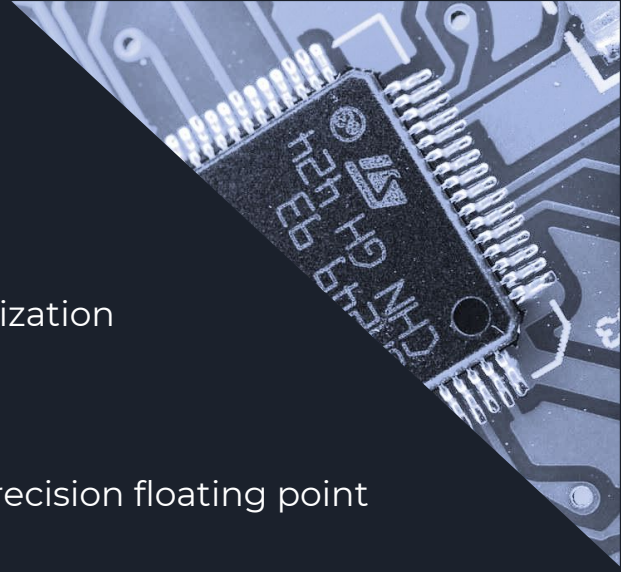
To increase efficiency, we are using CCM algorithmic logic where the program performs a series of additions and shifts rather than multiplication or division operations in order to eventually converge on the correct answer.

Requirements



Processor: 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.

Requirements



Given a range of argument values that require pre-normalization

The range for 16th root is [1.0, 65536.0]

Determine results for these 20 random values in double precision floating point arithmetic through MATLAB

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

Calculate 16Musing fixed point arithmetic, where $1.0 \leq M < 65536.0$, with 64 bits for the integer part and 64 bits for the fractional part.

Understanding the Problem

Employing Convergence Computing Algorithm in Calculating the 16th Root

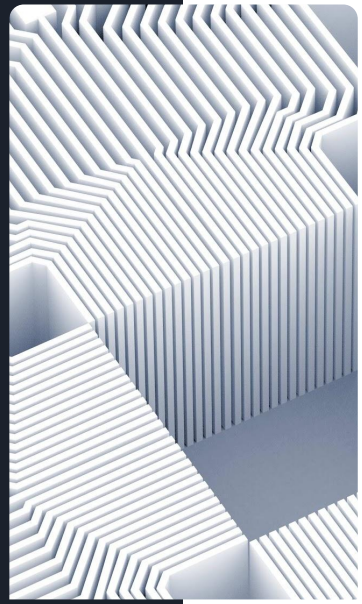
01 The use of logarithm and exponential computations are frequently used in signal processing, such as for procedures like determinant calculation.

02 Direct evaluation of logarithms and exponentials can be computationally demanding since it translates to a sequence of multiplications, additions, and memory look up operations.

03 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.

About CCM

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.



CCM 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 $1.0 \le M < 65536.0$. 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 $M - \Delta < \prod_{i=1}^{K} A_i^{16} \le M$ and $\sqrt{M - \Delta} < \prod_{i=1}^{K} A_i \le \sqrt{M}$. 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.

Pseudocode

Calculation of 16th Root Pseudocode	

f = 1.0
$fsqrt16$ = 1.0
for i = 0 to K - 1 do
$\mu = f \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i})$
$\cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i})$
$\cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i}) \cdot (1+2^{-i})$
$\cdot (1+2^{-i})$
$\mu sqrt16 = fsqrt16 \cdot (1+2^{-i})$
if $\mu \le M$ then
$f = \mu$
$fsqrt16 = \mu sqrt16$
return $fsqrt16$
	√ M with K bits of precision
	
	Potential multiplication by A_i^{16}
	
	Potential multiplication by A_i
	
	If product is less than M accept iteration
	
	Otherwise reject it (do nothing)

Part II: Design

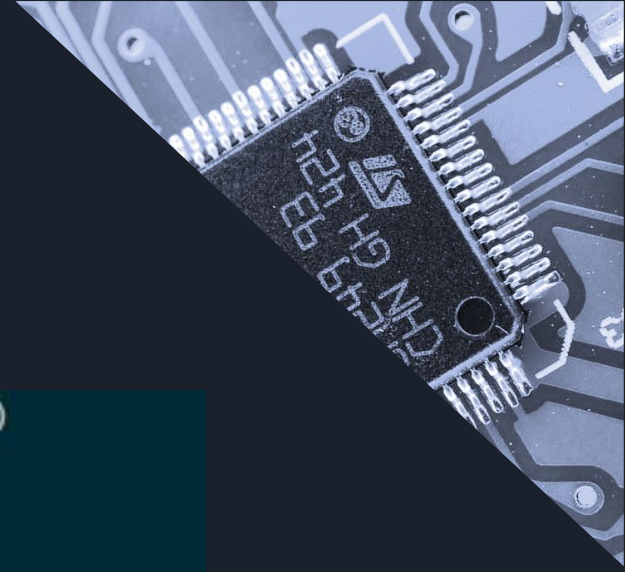
Number Representation

To facilitate the highest level of accuracy in our solution, we implement a struct representation for 128-bit integers, comprised of two unsigned long long ints.

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

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

Addition with 128 bit Numbers



```
my_uint128_t increment(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;
}
```

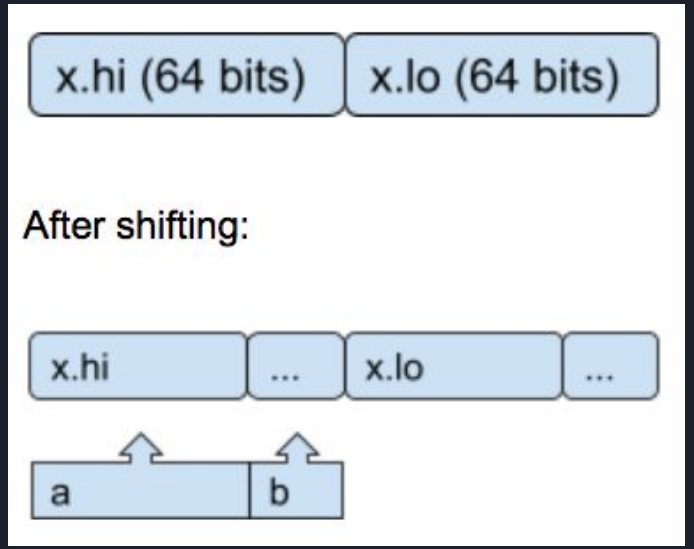
Addition with 128 bit Numbers

```
// 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;
}
```

Shifting with 128 bit Numbers

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.



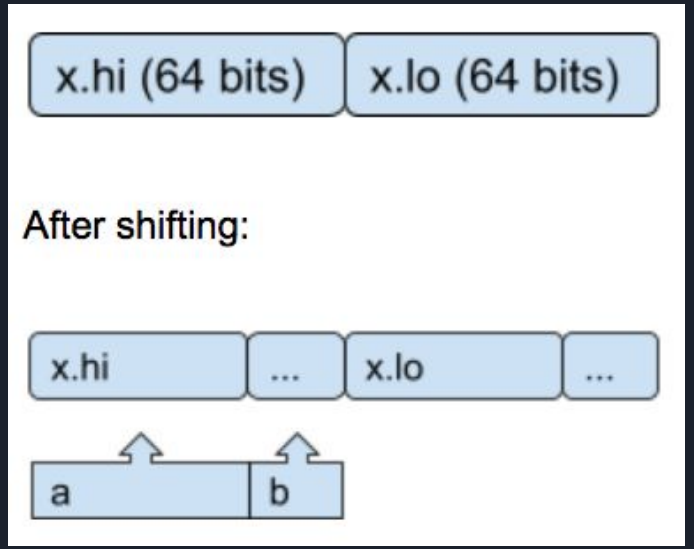
Shifting with 128 bit Numbers

a is x.hi << s and b is the s number of MSB of x.lo shifted over

x.hi and x.lo are conceptually adjacent to each other

When the bits of x.hi is shifted to the left, in order to fill in the space left created on the right end 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.



Left Shifting with 128 bit Numbers

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)
{
    // 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);
    }
}
```

Right Shifting with 128 bit Numbers

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)
{
    // shifting a 64-bit integer by more than 63 bits is "undefined"
    if (s >= 64)
    {
        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);
    }
}
```

Conversions

How were the initial decimal values going to be converted to integers? Multiply the initial value by a factor of 2.

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.

Conversions

Using a lower multiplier: Increases the range of the solution but will impact the accuracy negatively.

Using a higher multiplier: Gives a higher level of accuracy but reduces the range of input values.

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.

Scale Factors

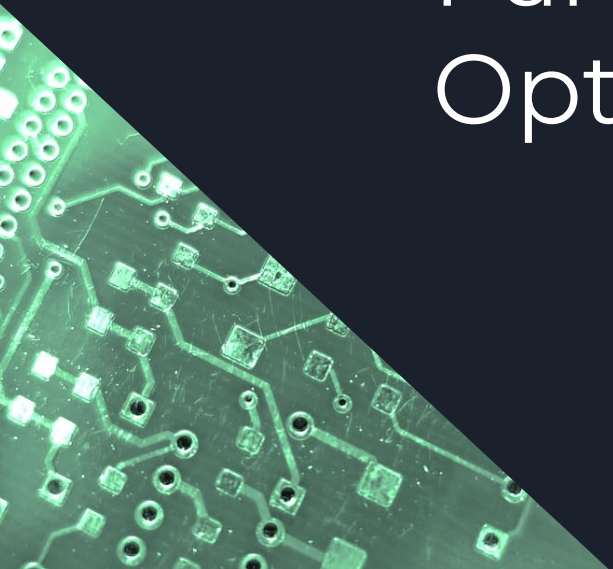
f

- Can get quite large if the first couple iterations are accepted
- Scale factor need to have enough bits to store f without overflow
- Higher scale factors didn't have any noticeable effect on the accuracy of the final solutions
- 2^{64} is used, since this allows f to grow a significant amount without overflowing

f_sqrt_16

- Represents the final answer
- Is in the range [1, 2.8284]
- Thus only 2 bits are needed to represent every integer number of f_sqrt_16.
- Scale factor of 2^{125} is used, leaving 3 bits for the final answer.
- This prevents the final answer from overflowing, with room to increase the range of supported input values.

Part III: Optimization



Evaluation Criteria

We will observe the change in performance that results from each optimization using the average execution time of the program.

Tools: A Bash script that utilizes a for loop and the UNIX “time” command, along with a Python script which averages the output values.

Our solution is run 100,000 times with the same input value, to keep the timings as consistent as possible.

Optimization #1: Algorithm Change

Before Optimization Execution Time (ms)	After Optimization Execution Time (ms)	Execution Reduction (ms)	Execution Reduction (%)
41.00	42.09	-1.09	-2.66%

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.

Optimization #2: For Loop

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

```
// 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
```

Optimization #3: Predicate Operations + Removal of Increment Function

```
// 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)
);
```

```
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;
}
```

Optimization #3: Predicate Operations + Removal of Increment Function

Before Optimization Execution Time (ms)	After Optimization Execution Time (ms)	Execution Reduction (ms)	Execution Reduction (%)
38.10	36.85	1.25	3.28

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.

Optimization #4: Loop Unrolling

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

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;
}
```

Before

Optimization #4: Loop Unrolling

After

```
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;
}
```

Optimization #4: Loop Unrolling

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

Firmware Opts

1. The original assembly code has 556 lines.
2. Using firmware optimization, in this case a horizontal machine with 2 issue slots, the assembly code can be reduced to 446 lines.
3. Instructions that are not related to each other and adjacent in position can be executed simultaneously.
4. Reduced difference of 110 lines of assembly code does not accomplish much performance improvement given the increase in price of using a horizontal machine.

Hardware Opts

1. The first hardware optimization was done on the functions shiftl128 and shiftr128
2. 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
3. The same hardware implementations could be reused for both functions
4. After this hardware optimization the number of generated assembly lines was reduced to 488

Hardware Opts

```
// 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)
   );
}
```

```
// 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)
      );
   }
}
```

Hardware Opts

```
#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
```

```
#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      shift1128
        b       .L4
.L2:
        ldr     r3, [fp, #-16]
```

```c
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;
}
```

Part IV: Results

Accuracy

- High accuracy overall
- 20 test bench values, with many answers having 30+ decimal places of accuracy
- Lower value + higher # of decimal places increases the chance of inaccuracy in the final answer

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	1.406293570152046701338122147710	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+

Performance

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

- 12.34% reduction in execution time
- In one hour, this results in 12,362 more executions

Questions?