CORDIC
Design and Optimization

University of Victoria
Department of Engineering

Lee Zeitz - V00835415
Kirk D’Mello - V00832871
Rickus Senekal - V00826364

leezeitz@uvic.ca
kirkd@uvic.ca
rsenekal@uvic.ca

Submission Date: _

1.0 Introduction

This report dives into the depths of the **CO**ordinate **R**otation **DI**gital **C**omputer algorithm, further referred to as CORDIC. It is an iterative method performing vector rotations by arbitrary angles using only shifts and additions. This implementation of the CORDIC algorithm consists of C language programming and compiled assembly code, which has been crafted and optimized for the ARM architecture. The algorithm defined in this report has very low complexity, using only shifts and adds. In addition to low complexity, it has a very low memory footprint, using only a small look up table with as many integer entries as the number of precision bits required. The ARM CMSIS library occupies 220kB, while CORDIC may be less than 1kB [1].

Optimized implementation of the CORDIC algorithm enabled it to be very efficient, performing, on average, a number of iterations equal to the number of precision bits required. Optimization of the algorithm was done using loop unrolling, software pipelining, register specifiers, as well as other techniques discussed in class.

A large contribution to this project was made by Dr. Sima in supplying us with his CORDIC slide deck. In the slide deck we found lots of useful information, including implementation details, optimization opportunities, and some guidance on performance analysis. We used the functional code in this slide deck as our starting point. A more specific description of how we used this code can be found in section 3.0.

All members of the group were equally responsible for the understanding of the unoptimized C code, and the CORDIC algorithm. Lee was primarily responsible for the software pipelining and single loop unrolling optimizations done, as well as the UML diagrams. Rickus was primarily responsible for the variable assignment modifications made to the original code, as well as the fully unrolled optimization. Kirk took charge of the assembly optimization, implementing predicate operations and register dedication. All group members were equally responsible for the performance analysis of all the steps performed throughout this project, and for the report.

The performance and cost evaluation is based on the original C code, along with the optimized version. Optimized assembly ARM is generated through compilation with the O1 - O3 flags kept in mind and various optimization methods manually applied.

Organization of this project revolves around the shared effort by all team members to successfully deliver UML design specifications, functionally-correct C code, optimized C code, and optimized assembly code based on the implementation of CORDIC as described in this report.

2.0 Theoretical background

CORDIC is a hardware-efficient iterative method which uses rotations to calculate a wide range of elementary functions. Suppose we are given a system that receives a vector and rotates it by an arbitrary angle θ. Choosing the origin as the center of rotation, we can use equation 1 below to get to the point (x1,y1) by rotating the point (x0,y0) by θ. [2]

$$x_R = x_{in}cos(\theta) - y_{in}sin(\theta)$$
$$y_R = x_{in}sin(\theta) + y_{in}cos(\theta)$$

Equation 1: Rotation of coordinate points.

A visual representation of the above equation can be displayed as follows:

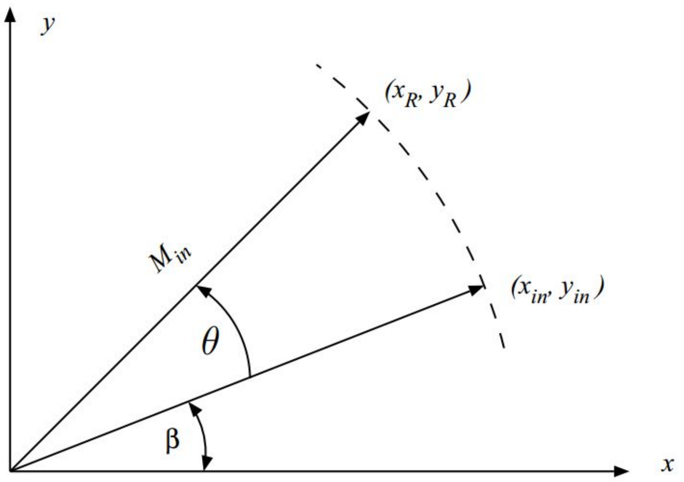


Figure 1: Rotation of an input vector by angle θ. [3]

As becomes clear when analyzing the above equation, we notice that for one rotation, we need to perform 4 multiplications, as well as some additions and subtractions. The main problem we are trying to solve is avoiding these expensive multiplications.

The CORDIC algorithm relies on two fundamental ideas to achieve rotation without multiplication. The first idea is that rotating the input vector by an arbitrary angle

θ_d is equal to rotating the vector by several smaller angles, θ_i, $i = 0, 1, \ldots, n$, provided $\theta_d = \sum_{i=0}^{n} \theta_i$. The second idea is that the chosen elementary angles are very small, such that $tan(\theta_i) = 2^{-i}$ for $i = 0, 1, \ldots, n$. In this way, multiplication by $tan(\theta_i)$ can be achieved by a bitwise shift which is a much faster operation compared to multiplication.

3.0 Design Process

For this project we explored several software optimization techniques such as loop unrolling, software pipelining, and using predicate bits to speed up the performance of the CORDIC algorithm. In this section we go into detail about our implementation of these techniques.

The functional unoptimized C code we used in this project was supplied nearly in full by Dr. Sima. We made slight modifications and augmentations to it, including reading in input coordinates and rotation angles from a formatted input file, writing the results to an output file, and making slight optimizations to variable use and assignments in the cordic_V_fixed_point function. We carefully studied the functionality of the unoptimized code to fully understand its operation and flow. The flow of the program as a while is modeled in Figure 2, and the flow of the cordic_V_fixed_point function in Figure 3.

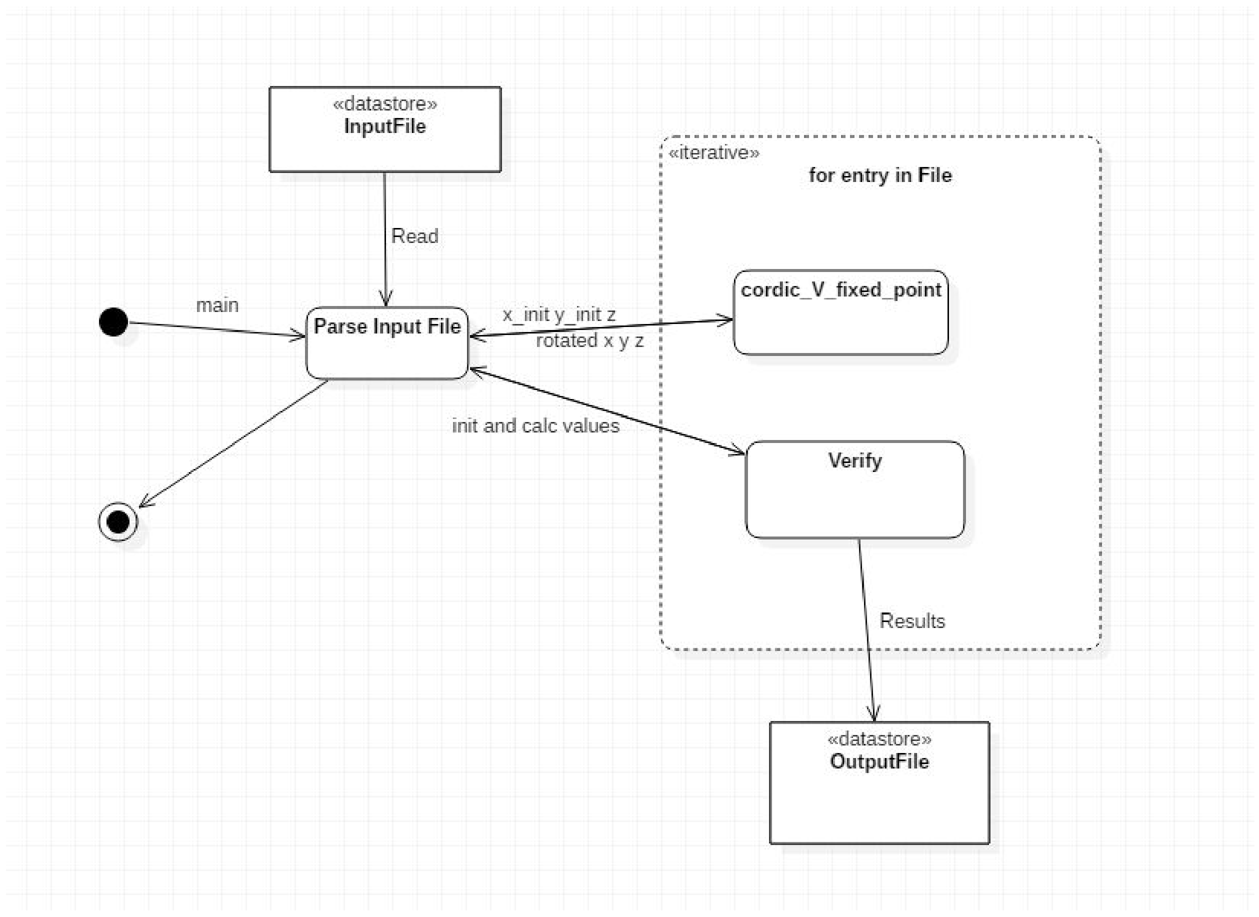


Figure 2: UML Activity Diagram of Unoptimized CORDIC Program

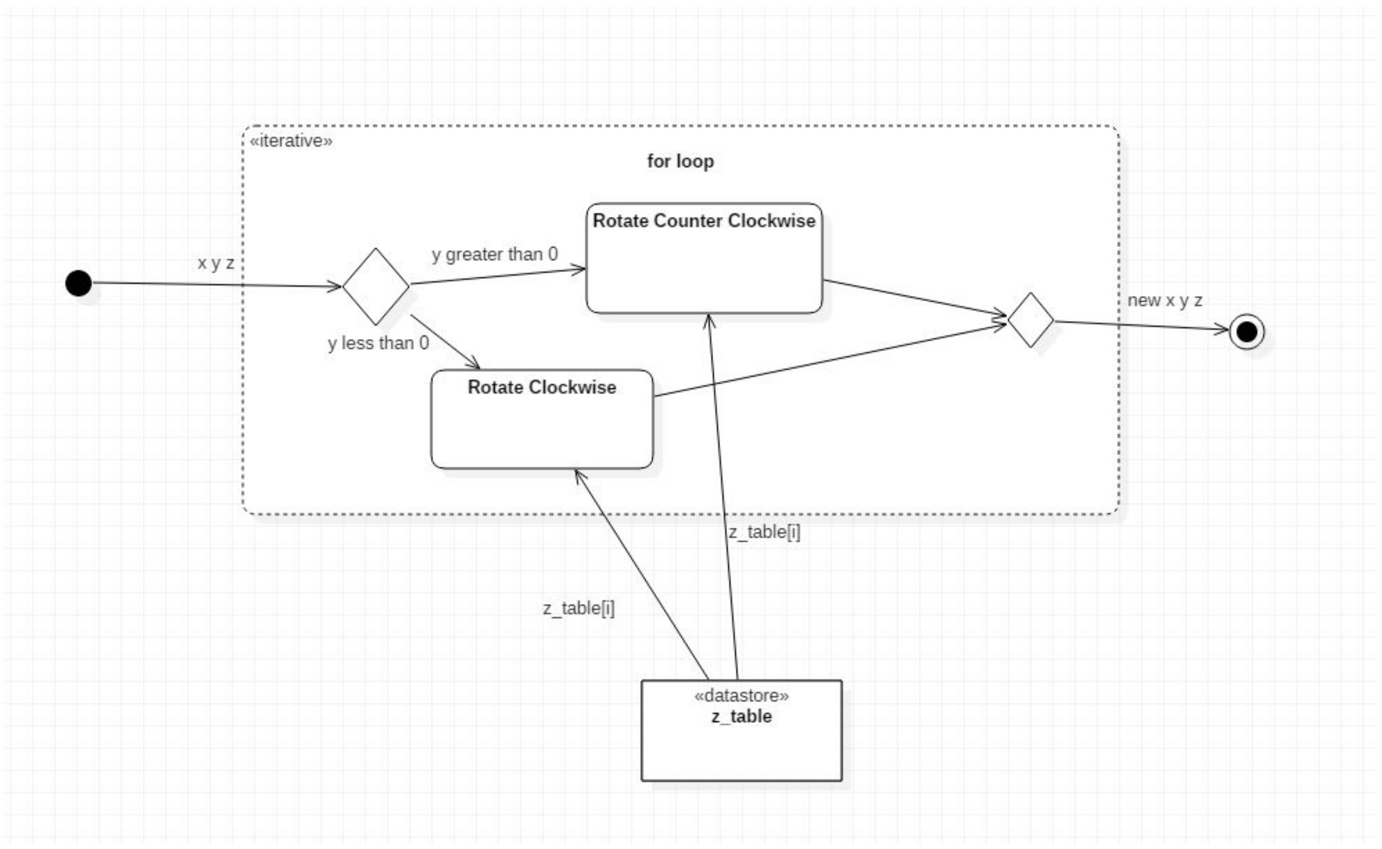
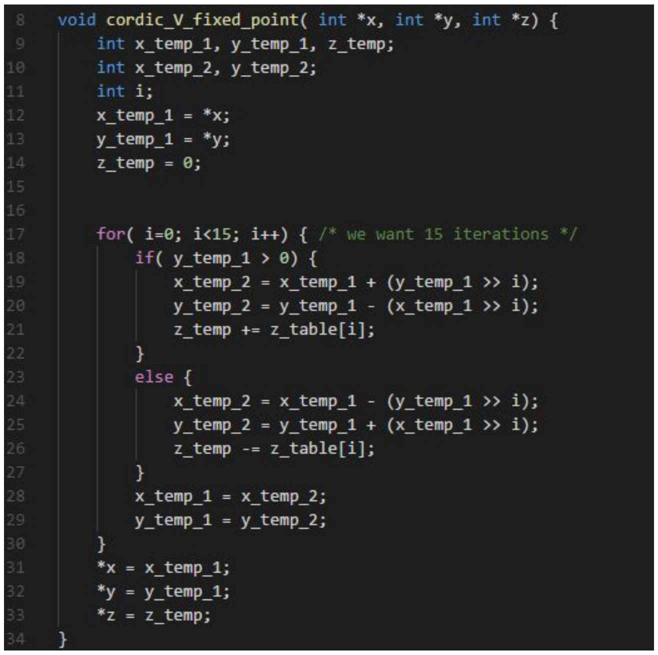


Figure 3: UML Activity Diagram of Unoptimized cordic_V_fixed_point

3.1 Loop Unrolling

Loop unrolling is the process of performing the inner code of a loop multiple times per iteration, instead of the conventional once per iteration. It typically involves repeating the loop body instructions multiple times within the loop. Doing this increases the use of the register files, which can in turn increase the efficiency of the program. It also reduces the number of checks the program would have to make at each iteration of the loop in order to determine if the loop should continue or break, as the number of iterations will be decreases by a factor of how much the loop is unrolled.

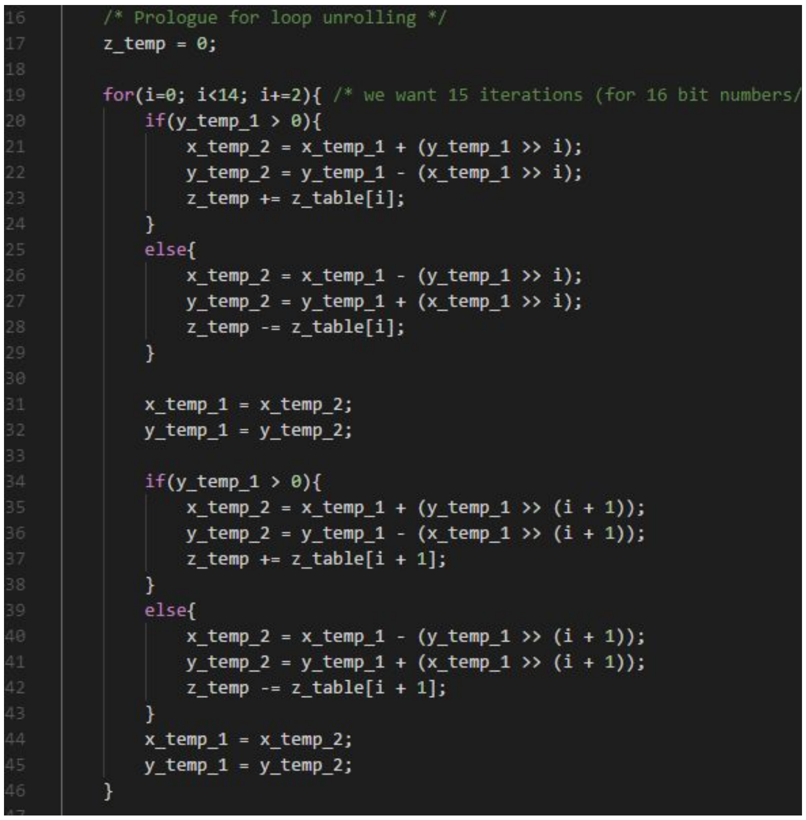
For the CORDIC algorithm in this project, the interesting code, as well as the program bottleneck, is the cordic_V_fixed_point function. This function, original shown in figure 4, is responsible for taking the initial X and Y coordinates of a vector, as well as the angle Z to rotate the vector, and calculating the corresponding rotated vector. Since this is the bottleneck of the program, and it has a relatively short for loop that iterates a significant number of times, it is a candidate for loop unrolling.



```
void cordic_V_fixed_point( int *x, int *y, int *z) {
    int x_temp_1, y_temp_1, z_temp;
    int x_temp_2, y_temp_2;
    int i;
    x_temp_1 = *x;
    y_temp_1 = *y;
    z_temp = 0;


    for( i=0; i<15; i++) { /* we want 15 iterations */
        if( y_temp_1 > 0) {
            x_temp_2 = x_temp_1 + (y_temp_1 >> i);
            y_temp_2 = y_temp_1 - (x_temp_1 >> i);
            z_temp += z_table[i];
        }
        else {
            x_temp_2 = x_temp_1 - (y_temp_1 >> i);
            y_temp_2 = y_temp_1 + (x_temp_1 >> i);
            z_temp -= z_table[i];
        }
        x_temp_1 = x_temp_2;
        y_temp_1 = y_temp_2;
    }
    *x = x_temp_1;
    *y = y_temp_1;
    *z = z_temp;
}
```

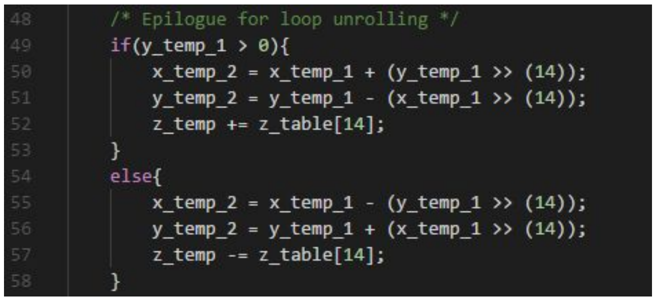
Figure 4: Original CORDIC Calculation Function

The first step taken to unroll the for-loop in this function was to duplicate all the code inside the for-loop so that it would run twice. The next step was to update the pasted code, to ensure proper alignment with the loop counter, i. Since we are now performing two operations in the loop, the second operation, which would have been the operation in the next iteration of the loop, relies on i + 1. The next step was to change the condition of the for loop. Since we are now performing twice the operations per loop, the loop counter must increment by 2 instead of 1. Also, since we are incrementing by 2, and the old condition involved an odd number, we must decrease the upper bound on i to the next even number, in this case 14, and add an epilogue to perform the 15th iteration. The unrolled loop as well as the prologue can be seen in Figure 5, and the epilogue in Figure 6.



```c
/* Prologue for loop unrolling */
z_temp = 0;

for(i=0; i<14; i+=2){ /* we want 15 iterations (for 16 bit numbers/
    if(y_temp_1 > 0){
        x_temp_2 = x_temp_1 + (y_temp_1 >> i);
        y_temp_2 = y_temp_1 - (x_temp_1 >> i);
        z_temp += z_table[i];
    }
    else{
        x_temp_2 = x_temp_1 - (y_temp_1 >> i);
        y_temp_2 = y_temp_1 + (x_temp_1 >> i);
        z_temp -= z_table[i];
    }

    x_temp_1 = x_temp_2;
    y_temp_1 = y_temp_2;

    if(y_temp_1 > 0){
        x_temp_2 = x_temp_1 + (y_temp_1 >> (i + 1));
        y_temp_2 = y_temp_1 - (x_temp_1 >> (i + 1));
        z_temp += z_table[i + 1];
    }
    else{
        x_temp_2 = x_temp_1 - (y_temp_1 >> (i + 1));
        y_temp_2 = y_temp_1 + (x_temp_1 >> (i + 1));
        z_temp -= z_table[i + 1];
    }
    x_temp_1 = x_temp_2;
    y_temp_1 = y_temp_2;
}
```

Figure 5: Prologue and Body of Unrolled Loop



```c
/* Epilogue for loop unrolling */
if(y_temp_1 > 0){
    x_temp_2 = x_temp_1 + (y_temp_1 >> (14));
    y_temp_2 = y_temp_1 - (x_temp_1 >> (14));
    z_temp += z_table[14];
}
else{
    x_temp_2 = x_temp_1 - (y_temp_1 >> (14));
    y_temp_2 = y_temp_1 + (x_temp_1 >> (14));
    z_temp -= z_table[14];
}
```

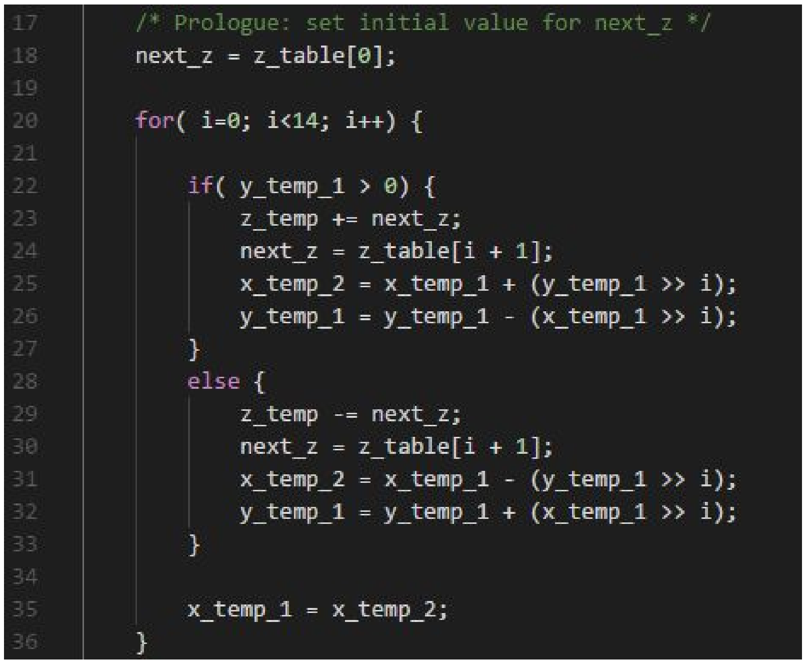
Figure 6: Epilogue for Unrolled Loop

3.2 Software Pipelining

Software pipelining is used to maximize the number of operations in a program that can be run in parallel. Filling the CPU's pipeline as much as possible will greatly increase its efficiency. It is also an important step in ensuring maximum utilization of the machine's resources. If a customer provides a machine with certain capabilities, the program to run on the machine should make use of as many of those capabilities as possible, so not to waste them, and therefore wasting customer's money on overpowered machines.

An opportunity for software pipelining is present in cordic_V_fixed_point. The angle Z for the next iteration can be loaded from the table of angle values, while the arithmetic of the current iteration is executed.

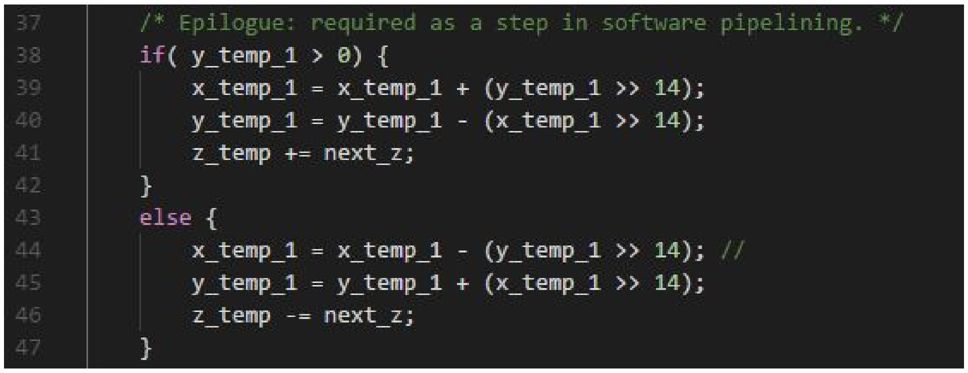
To create this software pipeline, we first create a new variable called next_z. Then, we create a prologue, in which we initialize this new variable to the first value of the tangent lookup table. Then, we replace the two instances of lookups in the tangent table inside the for-loop with next_z, which already has the value of the table for the first iteration, one in the if and the other in the else. We then reorder the operations inside the if and else statements so that the statement using next_z is executed first. This allows us to start the lookup into the tangent table for the next iteration and store the result in next_z variable while the remainder of the arithmetic in the if and/or else statements is executed. The prologue and loop body of the software pipelined code can be seen in figure 7.



```
/* Prologue: set initial value for next_z */
next_z = z_table[0];

for( i=0; i<14; i++) {

    if( y_temp_1 > 0) {
        z_temp += next_z;
        next_z = z_table[i + 1];
        x_temp_2 = x_temp_1 + (y_temp_1 >> i);
        y_temp_1 = y_temp_1 - (x_temp_1 >> i);
    }
    else {
        z_temp -= next_z;
        next_z = z_table[i + 1];
        x_temp_2 = x_temp_1 - (y_temp_1 >> i);
        y_temp_1 = y_temp_1 + (x_temp_1 >> i);
    }

    x_temp_1 = x_temp_2;
}
```

Figure 7: Prologue and Body of Software Pipeline Loop

Lastly, an epilogue was added to account for the last iteration of the loop. Since inside the loop we now calculate a value for the following iteration, an error would be thrown after the last iteration if the bound on the loop counter was left alone. We must decrement this bound, and duplication the arithmetic of the loop code into the software pipelining epilogue to perform the last iteration of the loop, as seen in Figure 8.



```
/* Epilogue: required as a step in software pipelining. */
if( y_temp_1 > 0) {
    x_temp_1 = x_temp_1 + (y_temp_1 >> 14);
    y_temp_1 = y_temp_1 - (x_temp_1 >> 14);
    z_temp += next_z;
}
else {
    x_temp_1 = x_temp_1 - (y_temp_1 >> 14); //
    y_temp_1 = y_temp_1 + (x_temp_1 >> 14);
    z_temp -= next_z;
}
```

Figure 8: *Epilogue of Software Pipelining*

Software pipelining is often one of the most difficult software optimizations to make, as it involves an in depth understanding of the algorithm and the dependencies within it. As such, we found it was an ideal candidate for a UML diagram to help organize the algorithm. A UML activity diagram for our pipelined cordic_V_fixed_point function can be found in Figure 9.

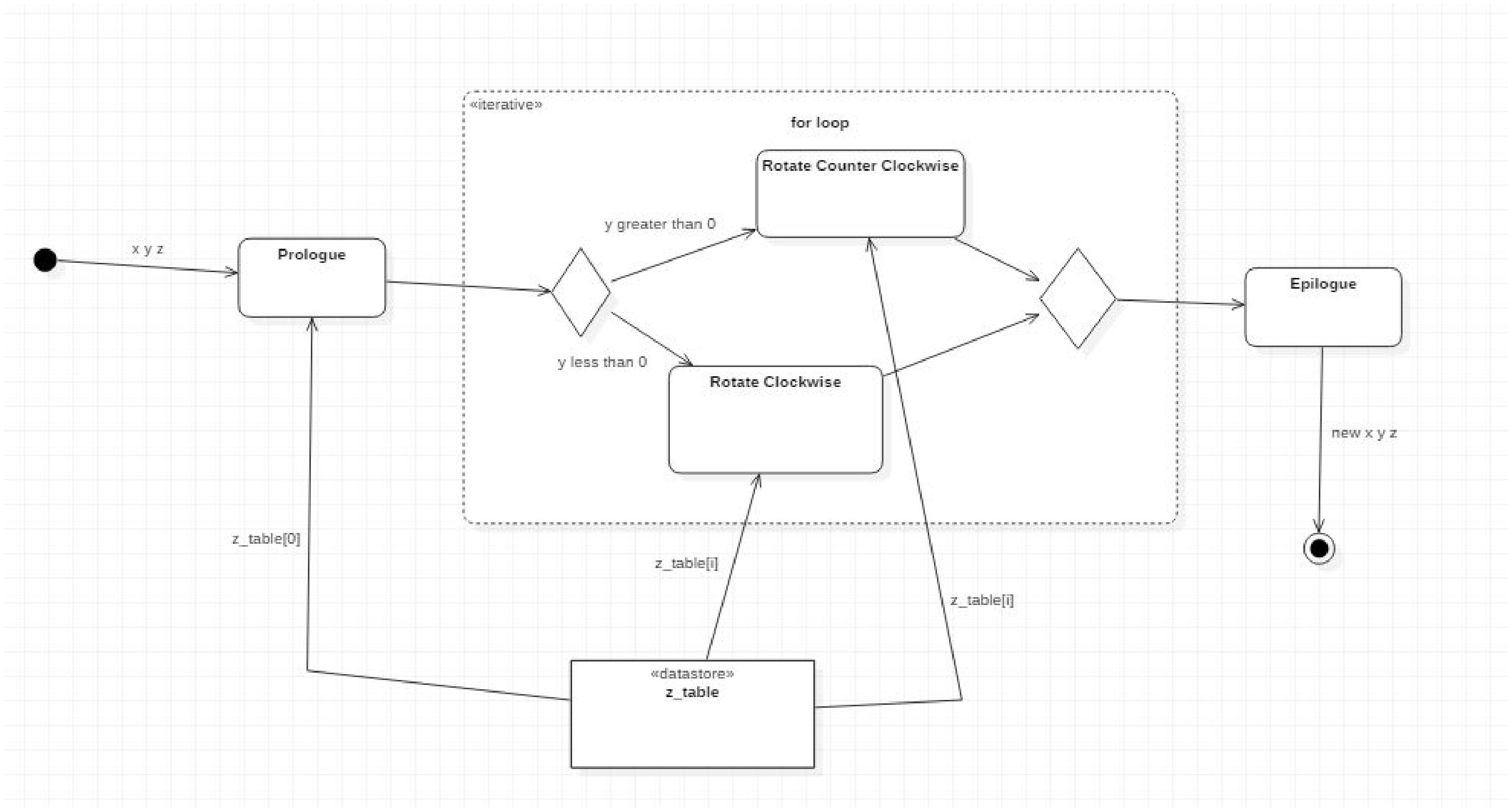


Figure 9: UML Activity Diagram of Software Pipelined cordic_V_fixed_point Function

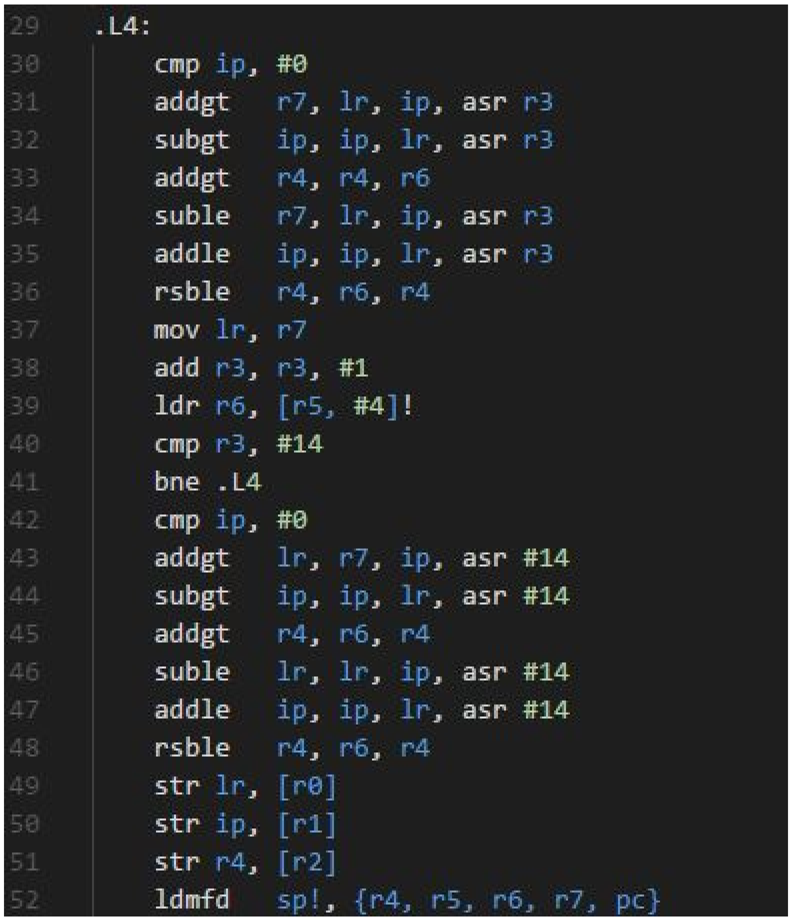
3.3 Register Specifiers
A request is made for the compiler to reserve a slot and place an object in the processor's register. If this request is fulfilled, the objects are easily accessed within a register so that the system no longer needs to load or store the desired object in program memory when using it.

A register int was specified for the for loop inside the cordic_V_fixed_point function. After inspection of the resulting assembly code, it appears that the compiler listened to the register suggestion and did indeed store the loop counter in a register; however, profiling this code led to very similar results as the profiles of the original code. Perhaps if the for loop had more iterations the time saved loading this loop counter from a register instead of memory would be more noticable, but for this small of a loop we saw no noticeable performance increase.

We tried setting the y_temp_1 to a register int as well, but inspection of the assembly code showed that the compiler ignored this suggestion. This is likely due to the program requiring access to that value again after the register it would be stored in needs reallocation. Perhaps if this program was running on a machine with more registers it would be able to afford to allocate a register specifically for y_temp_1.

3.4 Predicated Code
Using predicate operations involved replacing the if and else statement found in the loop body for determining "x_temp_1" and "y_temp_1" and then adding or subtracting "next_z" to "z_temp" based on the value of "y_temp_1".

After compiling the software pipelined C code into assembly using the 5 different optimization flags (none, O1, O2, O3, Os) and comparing the output of each, we noticed that each of the GCC optimized versions of assembly had already performed the predicate optimization. An example of such optimization can be seen in lines 43-48 in Figure 10. This demonstrates the conversion of the if and else statement in the for loop of the cordic_V_fixed_point function (lines 22-33 of Figure 7).



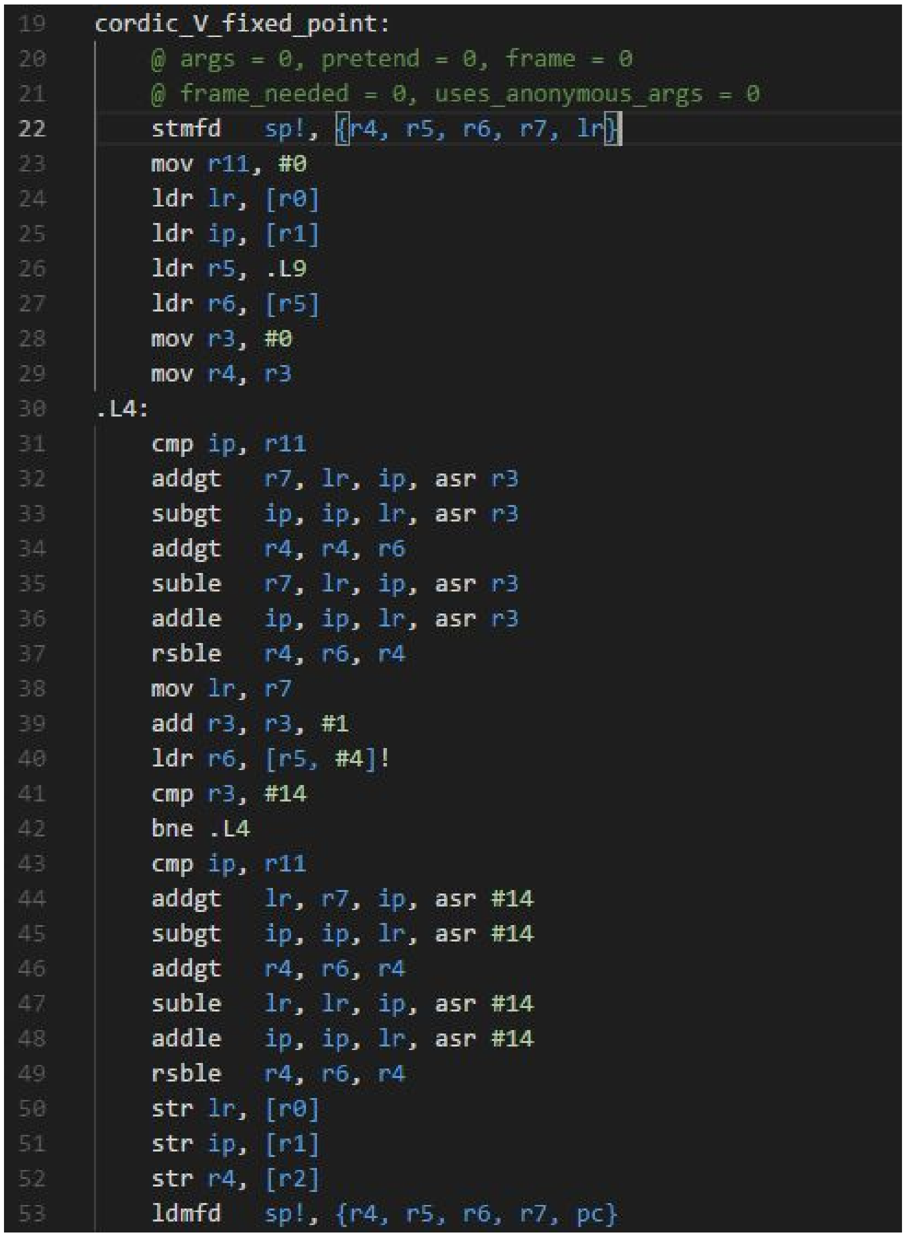
```
.L4:
    cmp ip, #0
    addgt   r7, lr, ip, asr r3
    subgt   ip, ip, lr, asr r3
    addgt   r4, r4, r6
    suble   r7, lr, ip, asr r3
    addle   ip, ip, lr, asr r3
    rsble   r4, r6, r4
    mov lr, r7
    add r3, r3, #1
    ldr r6, [r5, #4]!
    cmp r3, #14
    bne .L4
    cmp ip, #0
    addgt   lr, r7, ip, asr #14
    subgt   ip, ip, lr, asr #14
    addgt   r4, r6, r4
    suble   lr, lr, ip, asr #14
    addle   ip, ip, lr, asr #14
    rsble   r4, r6, r4
    str lr, [r0]
    str ip, [r1]
    str r4, [r2]
    ldmfd   sp!, {r4, r5, r6, r7, pc}
```

Figure 10: Predication in Optimized Assembly Code for CORDIC

3.6 Dedicated Zero Register

Another assembly optimization we considered is dedicating a register to the value 0. This would increase the speed of each loop comparison. Given that there are only 15 iterations of the loop in this program, the expected results of this are not grand, but this is an optimization that has the potential to scale quite well, and is relatively easy to perform, granted there are enough registers to dedicate one to 0. If the machine does not have enough registers to dedicate one specifically to 0, this optimization is not possible.

In the case of the CORDIC algorithm running on the Raspberry Pi 3B, there are enough registers to dedicate one to always be 0. We performed this optimization and found that the inconsistencies in our rounds of profiling were too noisy to notice a significant change in performance. However, it does reduce the number of clock cycles in the overall program. Given the simplicity of this optimization, it is one we recommend if the machine can afford to dedicate a register. An example of this optimization we made can be seen in line 23 of Figure 11, where 0 is assigned to register 11. Register 11 is then used to compare to 0 on lines 31 and 43. (It is worth noting that the Raspberry Pi has an ARM processor, which has 16 registers, as it uses 4 bits to encode them.)



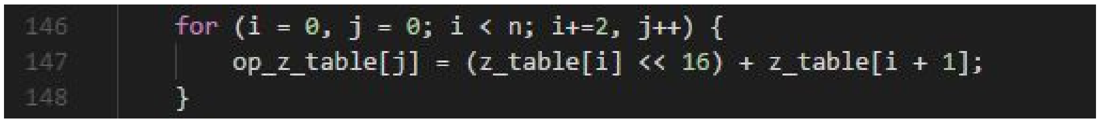
```
19  cordic_V_fixed_point:
20      @ args = 0, pretend = 0, frame = 0
21      @ frame_needed = 0, uses_anonymous_args = 0
22      stmfd   sp!, {r4, r5, r6, r7, lr}
23      mov r11, #0
24      ldr lr, [r0]
25      ldr ip, [r1]
26      ldr r5, .L9
27      ldr r6, [r5]
28      mov r3, #0
29      mov r4, r3
30  .L4:
31      cmp ip, r11
32      addgt   r7, lr, ip, asr r3
33      subgt   ip, ip, lr, asr r3
34      addgt   r4, r4, r6
35      suble   r7, lr, ip, asr r3
36      addle   ip, ip, lr, asr r3
37      rsble   r4, r6, r4
38      mov lr, r7
39      add r3, r3, #1
40      ldr r6, [r5, #4]!
41      cmp r3, #14
42      bne .L4
43      cmp ip, r11
44      addgt   lr, r7, ip, asr #14
45      subgt   ip, ip, lr, asr #14
46      addgt   r4, r6, r4
47      suble   lr, lr, ip, asr #14
48      addle   ip, ip, lr, asr #14
49      rsble   r4, r6, r4
50      str lr, [r0]
51      str ip, [r1]
52      str r4, [r2]
53      ldmfd   sp!, {r4, r5, r6, r7, pc}
```

Figure 11: Dedicated Zero Register

3.5 Load Operation Frequency Reduction

The last bit of C level optimization we did had to do with reducing the required number of loads from the static tangent table. Loads from memory are relatively expensive, and should be avoided if possible. To avoid it in this case, since we know all of the values in the lookup table are small enough to be represented accurately by 16 bits, we can store

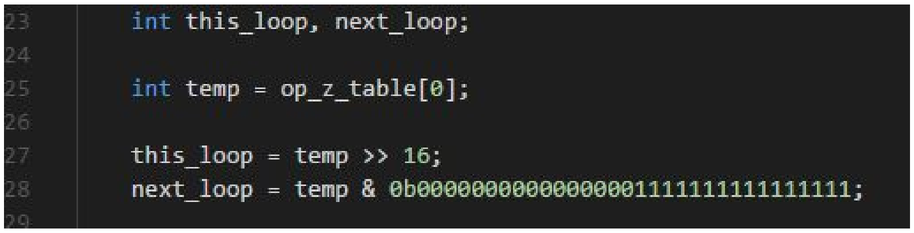
2 table entries into a single integer. To do this, we take one entry and shift it left 16 bits, and then add it to a second entry, giving us a 32 bit integer. This number is stored in a new lookup table with half as many entries as the original. The loop that performs this lossless compression can be seen in Figure 12.



```
for (i = 0, j = 0; i < n; i+=2, j++) {
    op_z_table[j] = (z_table[i] << 16) + z_table[i + 1];
}
```

Figure 12: Loop to Compress z_table to Reduce Lookup Frequency

These values can then be extracted in the cordic_V_fixed_point function, by reversing the shift for the first number, and masking the most significant 16 bits of the number for the second. (Note, the mask will work here since we know all our numbers are positive. If they could be negative, a mask would not work, as we may lose the sign bit. In this case, a shift left of 16 bits followed by an arithmetic shift right 16 bits would be the solution to recover the second number). This extraction can be seen in Figure 13.



```
int this_loop, next_loop;

int temp = op_z_table[0];

this_loop = temp >> 16;
next_loop = temp & 0b00000000000000001111111111111111;
```

Figure 13: Extraction of Compressed Table Values

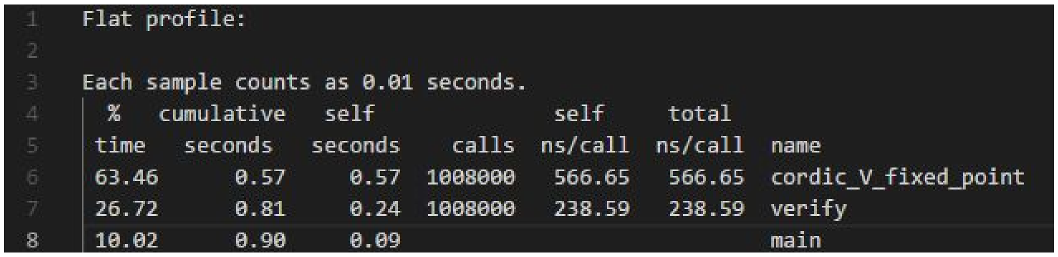
4.0 Performance/cost evaluation

In this section, we will talk about the performance results we saw in the original, unoptimized CORDIC algorithm, as well as the performance of the different optimization techniques we applied. We used the gprod profiler, built into the GCC compiler. The results analyzed in this section are from the flat profile generated by gprod on an input file of 17MB. The profiling was done on a Raspberry Pi 3B.

4.1 Original Un-optimized CORDIC Algorithm in C

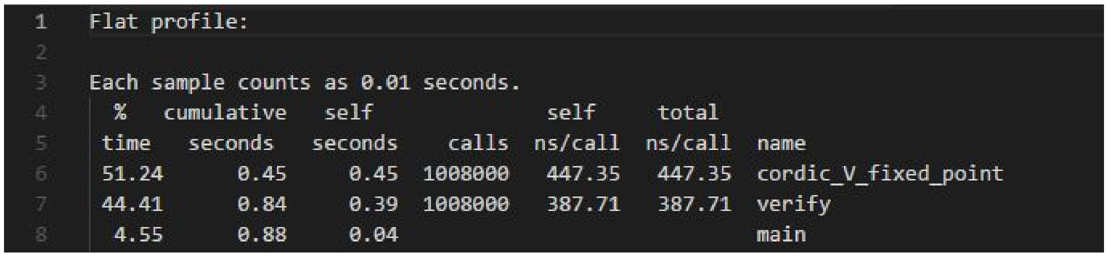
The original, unoptimized cordic_V_fixed_point code was profiled to take an average of 536.82ns per function call. This average was found by profiling the on three separate execution and averaging the results.

We decided that the amount of time per call was the best way for us to evaluate the performance of this algorithm, as it disregards any changes to main or verify (the two other functions in the program). Figures 14, 15, and 16 show the three flat profiles performed on the original code.



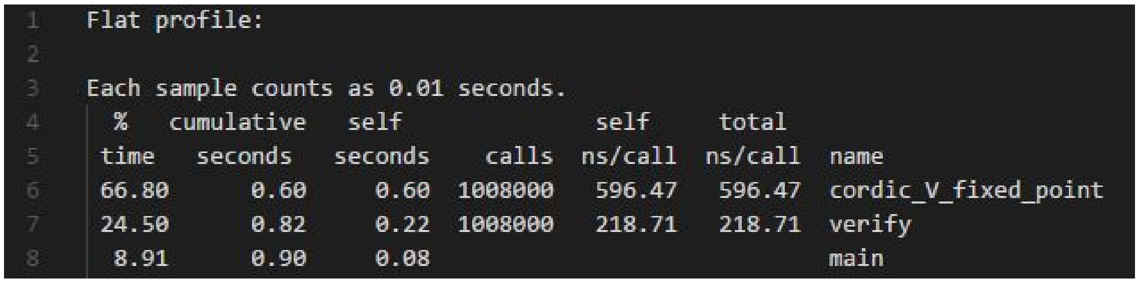
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 63.46      0.57     0.57  1008000   566.65   566.65  cordic_V_fixed_point
 26.72      0.81     0.24  1008000   238.59   238.59  verify
 10.02      0.90     0.09                             main
```

Figure 14: First Profile of Unoptimized CORDIC Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 51.24      0.45     0.45  1008000   447.35   447.35  cordic_V_fixed_point
 44.41      0.84     0.39  1008000   387.71   387.71  verify
  4.55      0.88     0.04                             main
```

Figure 15: Second Profile of Unoptimized CORDIC Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 66.80      0.60     0.60  1008000   596.47   596.47  cordic_V_fixed_point
 24.50      0.82     0.22  1008000   218.71   218.71  verify
  8.91      0.90     0.08                             main
```

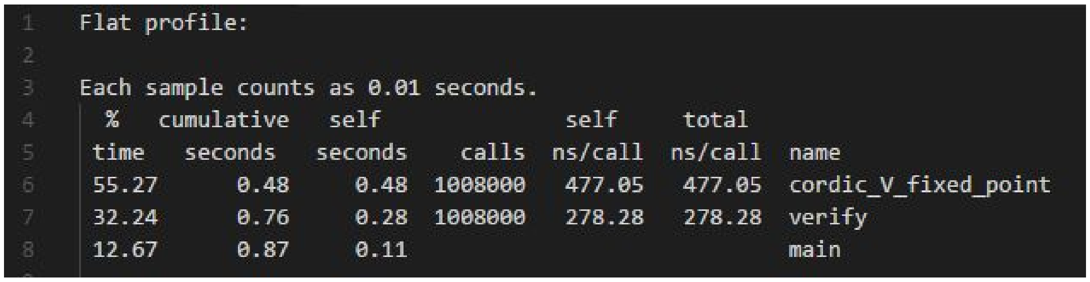
Figure 16: Third Profile of Unoptimized CORDIC Algorithm

This value, 536.82ns, is the baseline for the rest of the profiling for the optimizations done in this study.

4.2 Optimized w/ Loop Unrolling

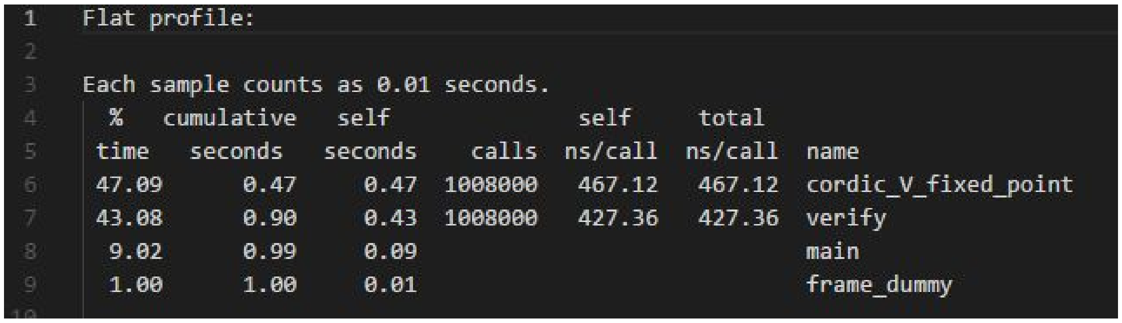
We performed two versions of loop unrolling in this assignment. One version where we only unrolled once, so there are two sets of operations per iteration, and one version where we unrolled completely, leaving 15 sets of operations and no loop at all.

We found the average execution time per function call for the single unrolled loop to be 490.31ns. This is a significant 9% increase in efficiency over the 536.82ns time of the unoptimized code. The profiling results can for the singly unrolled loop can be seen in figures 17, 18, and 19.



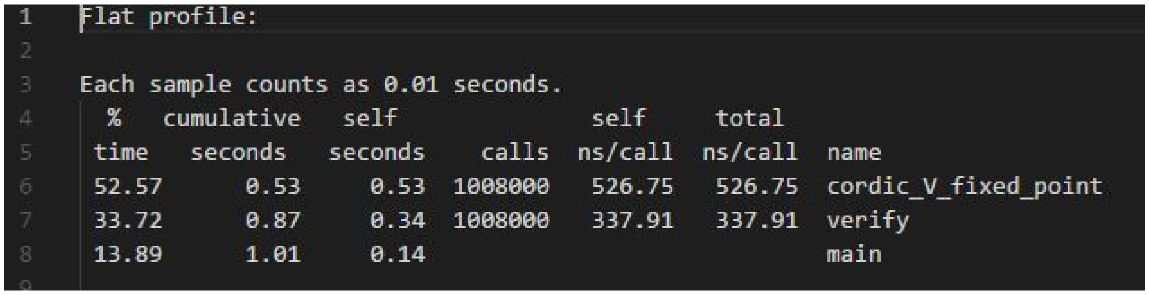
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 55.27      0.48     0.48  1008000   477.05   477.05  cordic_V_fixed_point
 32.24      0.76     0.28  1008000   278.28   278.28  verify
 12.67      0.87     0.11                             main
```

Figure 17: First Profile of Single Loop Unrolled Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 47.09      0.47     0.47  1008000   467.12   467.12  cordic_V_fixed_point
 43.08      0.90     0.43  1008000   427.36   427.36  verify
  9.02      0.99     0.09                             main
  1.00      1.00     0.01                             frame_dummy
```

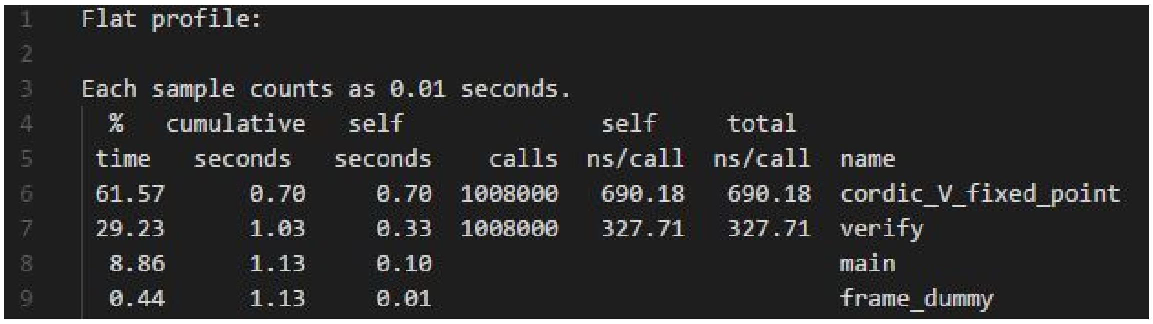
Figure 18: Second Profile of Single Loop Unrolled Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 52.57      0.53     0.53  1008000   526.75   526.75  cordic_V_fixed_point
 33.72      0.87     0.34  1008000   337.91   337.91  verify
 13.89      1.01     0.14                             main
```

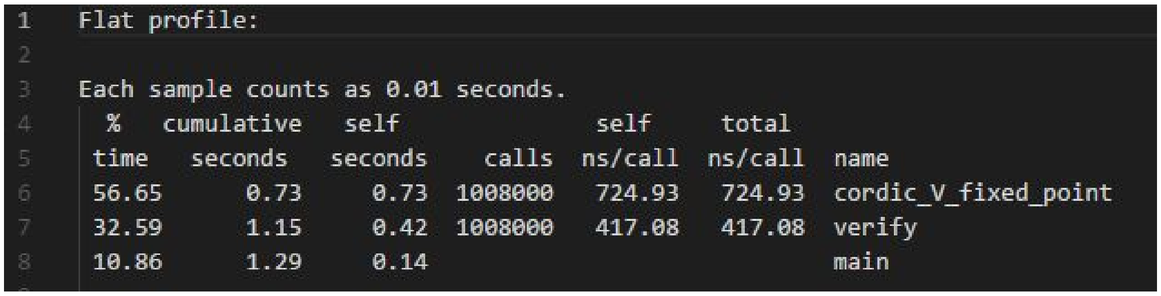
Figure 19: Third Profile of Single Loop Unrolled Algorithm

The average from our profiled unlimited loop unrolling was 713.35ns. This is a large increase in time and therefore decrease in efficiency. Doing an infinite unrolling of the loop caused our performance to decrease drastically. We suspect this is due to overflowing of the register file, causing excess reads and writes to be done to memory instead of registers. We found through our profiling that unrolling once had the largest increase in speed and efficiency. The profiles for the fully unrolled loop can be seen in Figures 20, 21, and 22.



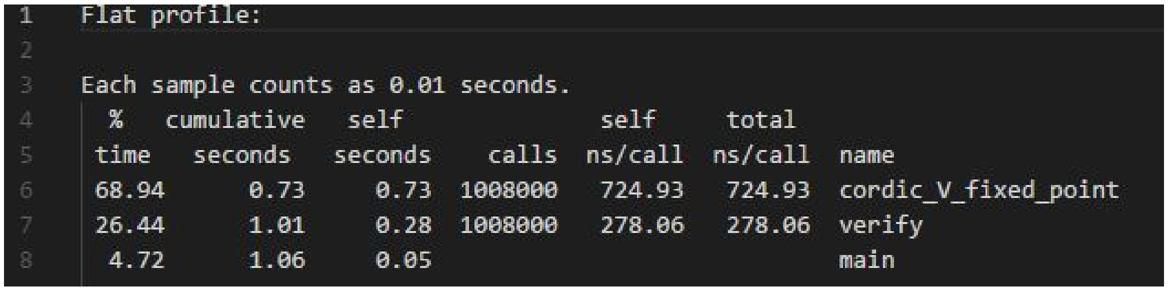
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 61.57      0.70     0.70  1008000   690.18   690.18  cordic_V_fixed_point
 29.23      1.03     0.33  1008000   327.71   327.71  verify
  8.86      1.13     0.10                             main
  0.44      1.13     0.01                             frame_dummy
```

Figure 20: First Profile of Fully Unrolled Loop Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 56.65      0.73     0.73  1008000   724.93   724.93  cordic_V_fixed_point
 32.59      1.15     0.42  1008000   417.08   417.08  verify
 10.86      1.29     0.14                             main
```

Figure 21: Second Profile of Fully Unrolled Loop Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 68.94      0.73     0.73  1008000   724.93   724.93  cordic_V_fixed_point
 26.44      1.01     0.28  1008000   278.06   278.06  verify
  4.72      1.06     0.05                             main
```

Figure 22: Third Profile of Fully Unrolled Loop Algorithm

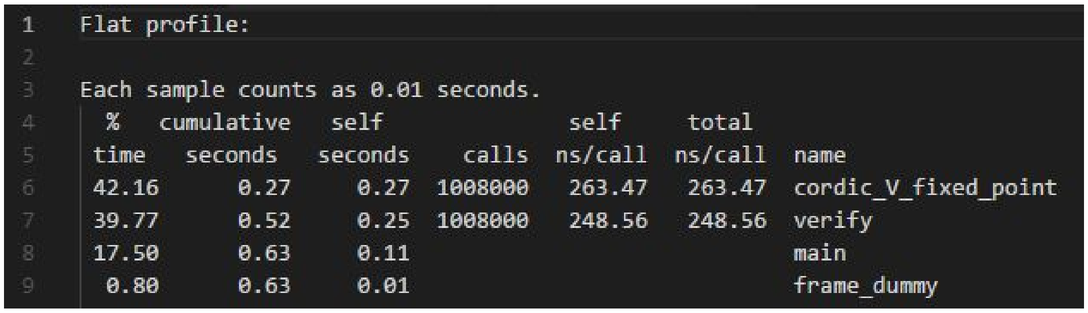
It is conceivable that unrolling another time would further increase the speed, and it would interesting to see at what degree of unrolling the register file starts to overflow, as it is likely that any amount of unrolling up to this point would be highly beneficial to the algorithm's performance.

4.3 Optimized w/ Software Pipelining

The average execution time per function call for the software pipelining implementation was calculated to be 309.87. This is an extremely significant 32% decrease in average execution time and increase in program efficiency. This increase in performance was expected, as the cordic_V_fixed_point function was a prime candidate for software pipelining. Performing the load from the lookup table for the next iteration during the arithmetic for the current iteration minimizes bubbles in the pipeline, taking maximum advantage of the multi-pipeline processor on the Raspberry Pi.

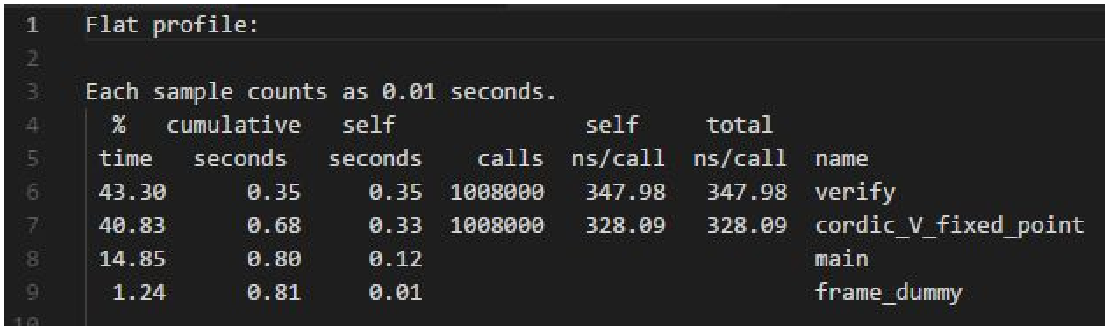
Software pipelining reaps all the benefits of unlimited loop unrolling, without the drawback of excesses stress on the register file and therefore costly memory accesses.

The profiles used to calculate and evaluate the performance of this optimization can be seen in Figures 23, 24, and 25.



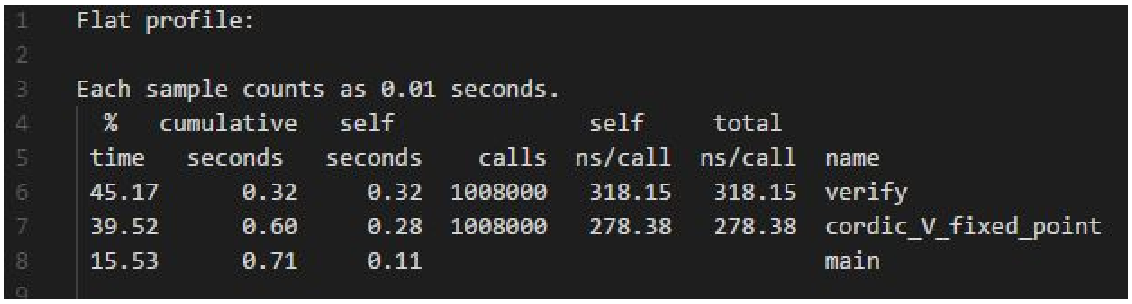
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 42.16      0.27     0.27  1008000   263.47   263.47  cordic_V_fixed_point
 39.77      0.52     0.25  1008000   248.56   248.56  verify
 17.50      0.63     0.11                             main
  0.80      0.63     0.01                             frame_dummy
```

Figure 23: First Profile of Software Pipelined Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ns/call  ns/call  name    
 43.30      0.35     0.35  1008000   347.98   347.98  verify
 40.83      0.68     0.33  1008000   328.09   328.09  cordic_V_fixed_point
 14.85      0.80     0.12                             main
  1.24      0.81     0.01                             frame_dummy
```

Figure 24: Second Profile of Software Pipelined Algorithm



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 45.17      0.32     0.32  1008000   318.15   318.15  verify
 39.52      0.60     0.28  1008000   278.38   278.38  cordic_V_fixed_point
 15.53      0.71     0.11                             main
```

Figure 25: Third Profile of Software Pipelined Algorithm

5.0 Conclusions

Over the course of the term in SENG440, our team has learned an immense amount about software optimization and fixed point implementation. As a result, we gained the ability to complete nearly all desired requirements of this project. Throughout working on the project, various techniques discussed in class, such as loop unrolling, software pipelining, register specifiers, and predication, were used for design and optimization of both C and assembly level code. Observing the performance evaluation, the efficiency of the CORDIC algorithm can be seen to increase significantly.

In closing, our implementation of the CORDIC project was successful and proved to be an excellent learning experience to carry with us into future courses, as well as our engineering careers.

6.0 Bibliography

[1] S.T. Life Augmented. *Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions.* Accessed 2017. Available at: https://www.st.com/content/ccc/resource/technical/document/design_tip/group0/9c/20/c6/67/50/10/4e/9d/DM00441302/files/DM00441302.pdf/jcr:content/translations/en.DM00441302.pdf

[2] All About Circuits. *An introduction to the CORDIC Algorithm.* Accessed 2017. Available at: https://www.allaboutcircuits.com/technical-articles/an-introduction-to-the-cordic-algorithm/

[3] UCLA. *CORDIC Algorithm and Implementations.* Accessed 2017. Available at: http://web.cs.ucla.edu/digital_arithmetic/files/ch11.pdf