University of Victoria
Faculty of Engineering
Summer 2016 SENG440 Report

CORDIC Optimization and Analysis

Wu Xian
V00760922
Computer Science
xianwu@uvic.ca

Huang Jinmin
V00787639
Software Engineering
jinminh@uvic.ca

Introduction

CORDIC (Coordinate Rotation Digital Computer) is a simple and efficient algorithm that can calculate hyperbolic and trigonometric functions, typically converging with one digit (or bit) per iteration. [1] The detailed description of CORDIC is included in theoretical background. In this project, firstly, our group encode the CORDIC algorithm by using C language. Then, based on the functionally-correct C code, we made optimization by applying loop unrolling, register specifier, software pipelining and several other techniques from the class and the performance and cost evaluation is produced according to original C code and optimized C code. A corresponding optimized ARM code are generated through O3 option. We also implemented automaton design, firmware (which is the file *Firmware Schedule* in submission) and a UML description of the project is also included in this report.

Theoretical background

- CORDIC algorithm

  CORDIC was proposed by Jack E. Volder in 1956. This algorithm is mainly applied in the calculation of trigonometric function, hyperbola, exponential function and logarithm. CORDIC requires addition, subtraction and bitshift instead of multiplication, which means that it is not necessary to use trigonometric function, multiplication, SQRT, inverse trigonometric function and so on to calculate the rotation and orientation of a vector.

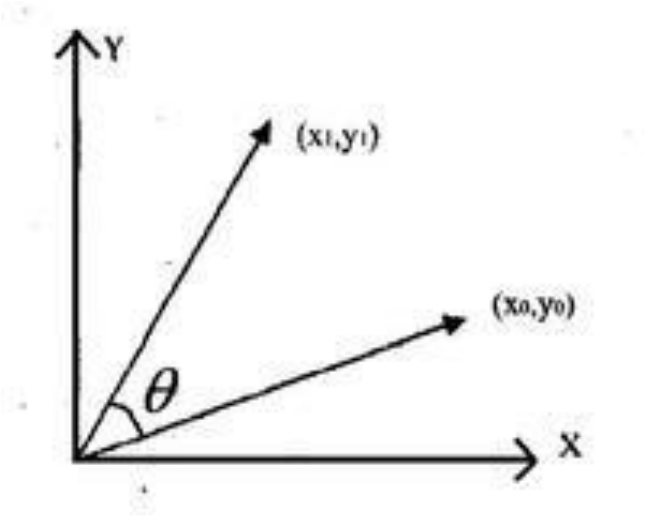


Figure 1 - CORDIC instance

According to the Figure 1, the relationship between the original vector (x0, y0) and the vector after rotation with θ (x1, y1) can be written as:

$X1=X0*\cos(\theta)-Y0*\sin(\theta)=\cos(\theta)(X0-Y0*\tan(\theta))$

$Y1=Y0*\cos(\theta)+X0*\sin(\theta)=\cos(\theta)(Y0+X0*\tan(\theta))$

Θ is the target angle. In order to get the value of θ, we can split θ into a sequence of subrotations of elementary angles θ[i]. The value of each tanθ[i] is $\pm 2^{-i}$. We can get that:

X(i+1)=cos(θ[i])(X(i)+Y(i)2^(-i))

Y(i+1)=cos(θ[i])(Y(i)+X(i)2^(-i))

θ[i+1] = θ[i] - σ[i] arctan(2^{-i})

$$\text{Where } \sigma[i] = \begin{cases} -1 & if\ z[i] < 0 \\ 1 & otherwise \end{cases}$$

After n iterations, the result is:

x[n] = A[n](x[0] cos (z[0]) − y[0] sin (z[0]))

y[n] = A[n](y[0] cos (z[0]) − x[0] sin (z[0]))

z[n] = 0

$$\text{where} \quad A[n] = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}}$$

- Loop unrolling

  Loop unwinding, also known as loop unrolling, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its

binary size (space-time tradeoff). The transformation can be undertaken manually by the programmer or by an optimizing compiler.

The goal of loop unwinding is to increase a program's speed by reducing (or eliminating) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration;reducing branch penalties; as well as "hiding latencies, in particular, the delay in reading data from memory". Loops can be re-written instead as a repeated sequence of similar independent statements eliminating this overhead. [2]

- Register specifier

  The compiler places the object in the processor's register. The objects are already stored in registers so that the system will no longer load and store those objects when using them.

- Software Pipelining

  The technique is to reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop

- Reduce the number of Load from memory

By packing two elementary angles into one element in the array, every load operation provides the value good for two iterations.

C & ARM Design Process

C Code Level

In C level, there are three input values, x, y, z, the optimized program has two modes, cordic_rotation which returns a vector contains final calculated x, y and cordic_vector which returns final calculated z in unit of degree angle. The fixed-point representation of floating point numbers x and y has integer data type, where 1 is represented as 2^20, the fixed-point representation of the elementary angles has integer data type, where 45 degrees is represented as 2^15, so the elementary angles can only be 16 bits long, the fixed-point representations of z and elementary angles have the same scale factor. Because angles and vectors are evaluated separately, the program allows them to have different scale factors. Both functions take inputs parameters x, y, z, create intermediate value holder x_temp and loop counter i, and try to indicate the compiler to store all of them in registers by using register specifiers because of frequent use of these variables. There are sixteen elementary angles prepared in the program, and they are stored in an eight-element array where each 32 bits element contains two 16 bits elementary angles,

by doing this, it reduces the number of load operations when angles in the elementary rotations array are needed. The program extracts elementary angles from 32 bits array value by using logical shift right to extract leftmost 16 bits or using bitwise operation to select rightmost 16 bits. Loop unrolling is also implemented in this program, inside the partially unrolled loop, optimized program repeat original loop-body instructions four times and load from array twice. Because the true dependency between load operations and extracting elementary angles, software pipelining also appears in the code. For loop is arranged so that the load of each element in the array is the value for extracting later.

The cordic function has been used to calculate trigonometric functions. For calculating sine and cosine, cordic receives input in unit of degree angle, and gives output in unit of radian. And for arctan, cordic receives input in unit of radian and gives output in unit of degree angle.

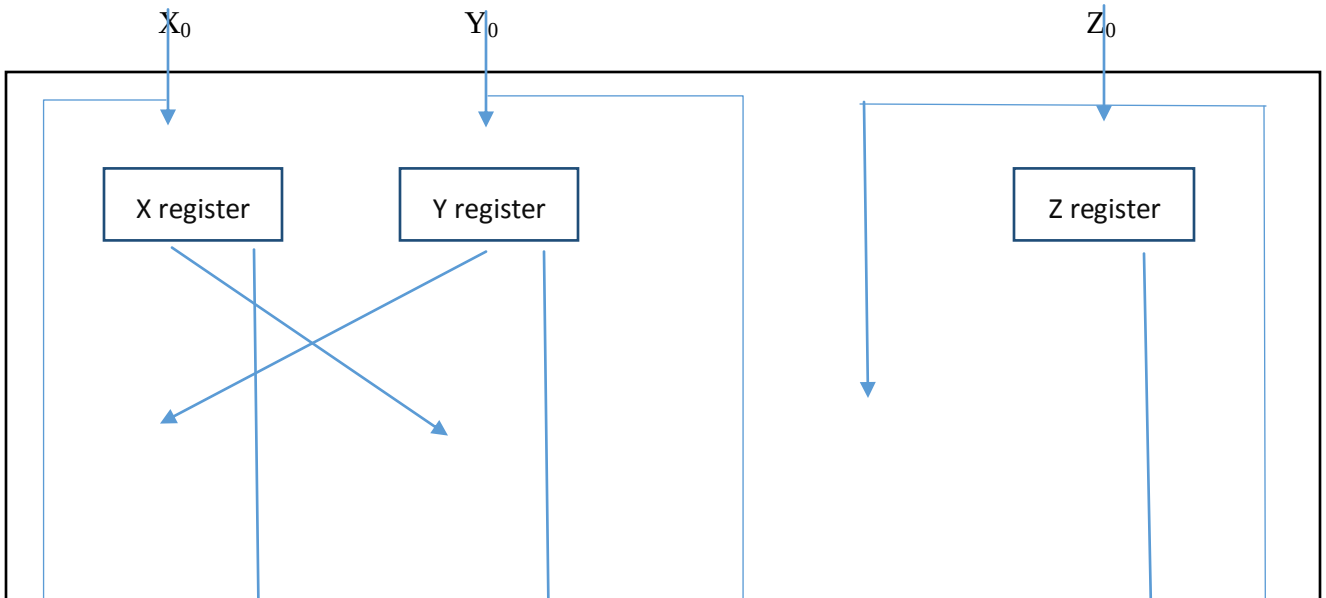
ARM Level

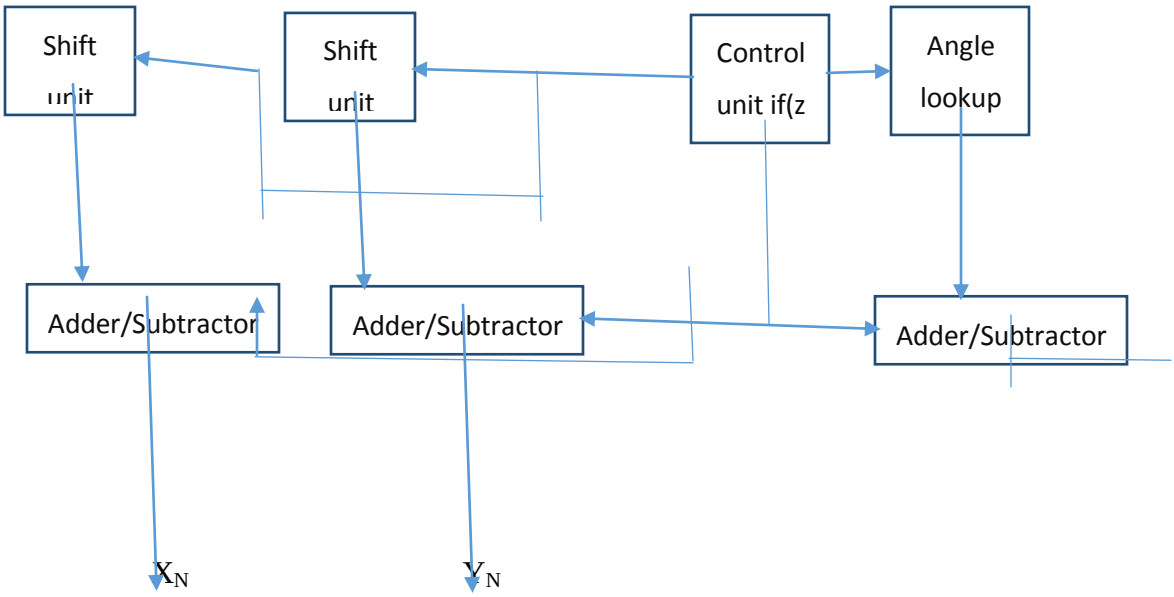
In assembly level, O3 optimized assembly code provides six predicated instructions to simulate one if-then-else situation (three for if and three for else). Although, based on different conditions, only three instructions will assign their values to registers, all six

conditional operations may be executed because it is dependent on the processor architecture, how far a predicated instruction proceeds speculatively in the pipeline before its predication is resolved. Since predicated instructions could be fetched, decoded and placed in the instruction window like non predicated instructions, two successive predicated instructions using the same register as output operand and one of input operand(rsbge r3, r4, r3 and addlt r3, r3, r4) may have true dependency problem. To solve the problem in *cordic_optimized_final.s*, all instructions with same condition will run successively because there's no true dependency between them.

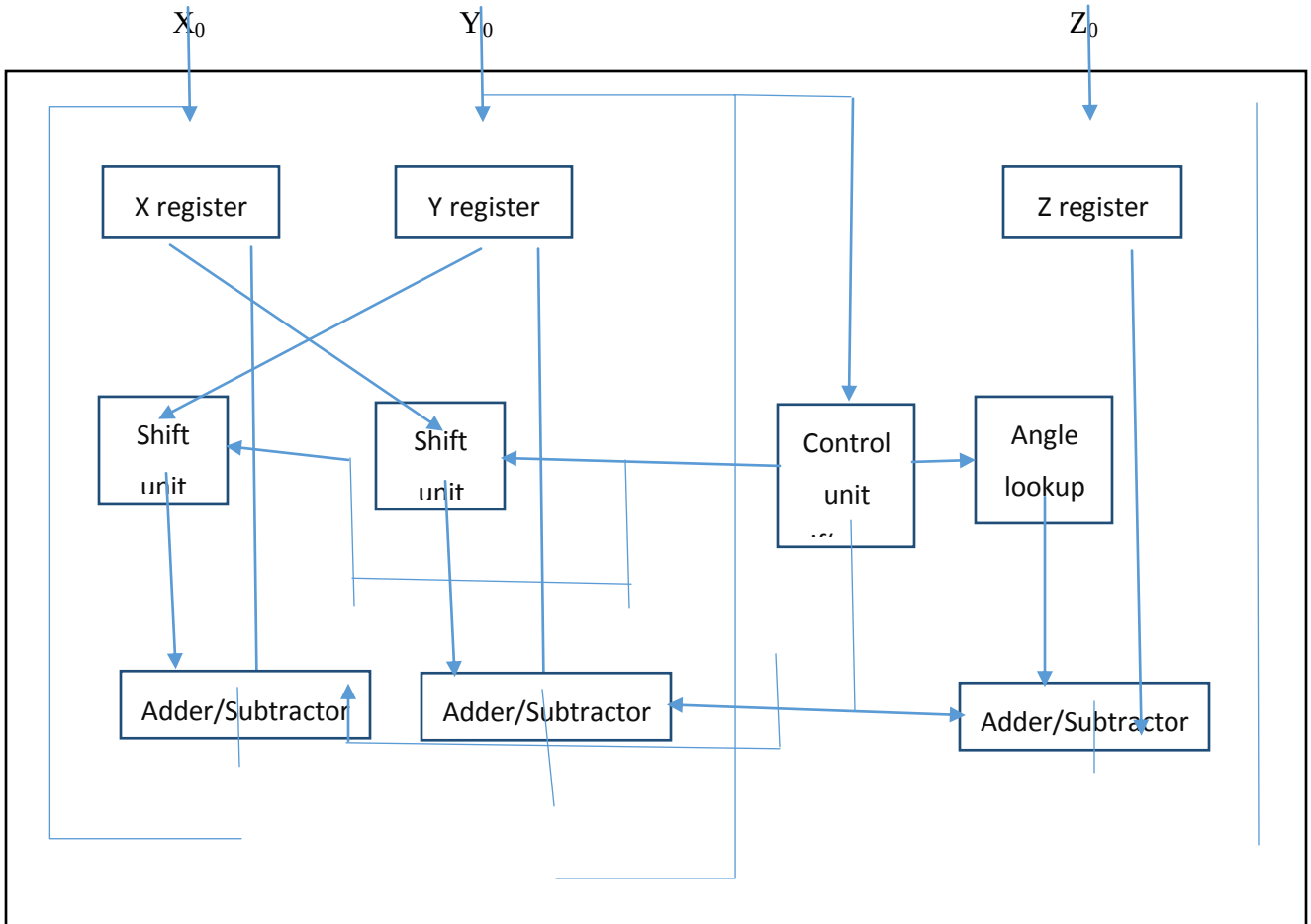
Also, to avoid true dependency between loading from array and extracting angles from elements, more operations are inserted between loading and extracting as long as the code is still functionally correct.

VHDL Behavior of Cordic Rotation

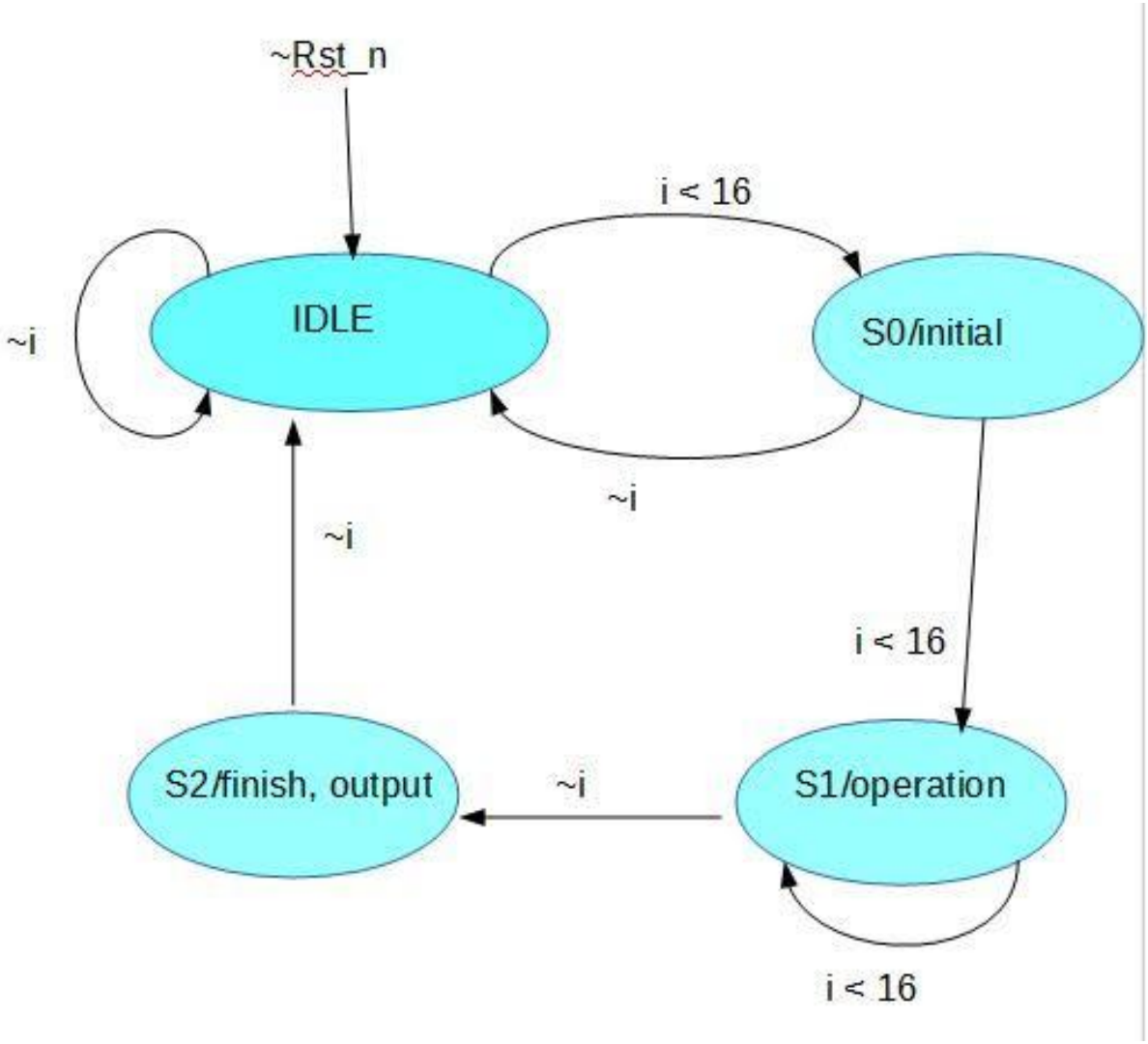




VHDL Behavior of Cordic Vector



Automaton Design


~Rst_n
i < 16
IDLE
S0/initial
~i
~i
~i
i < 16
S2/finish, output
~i
S1/operation
i < 16

UML Description

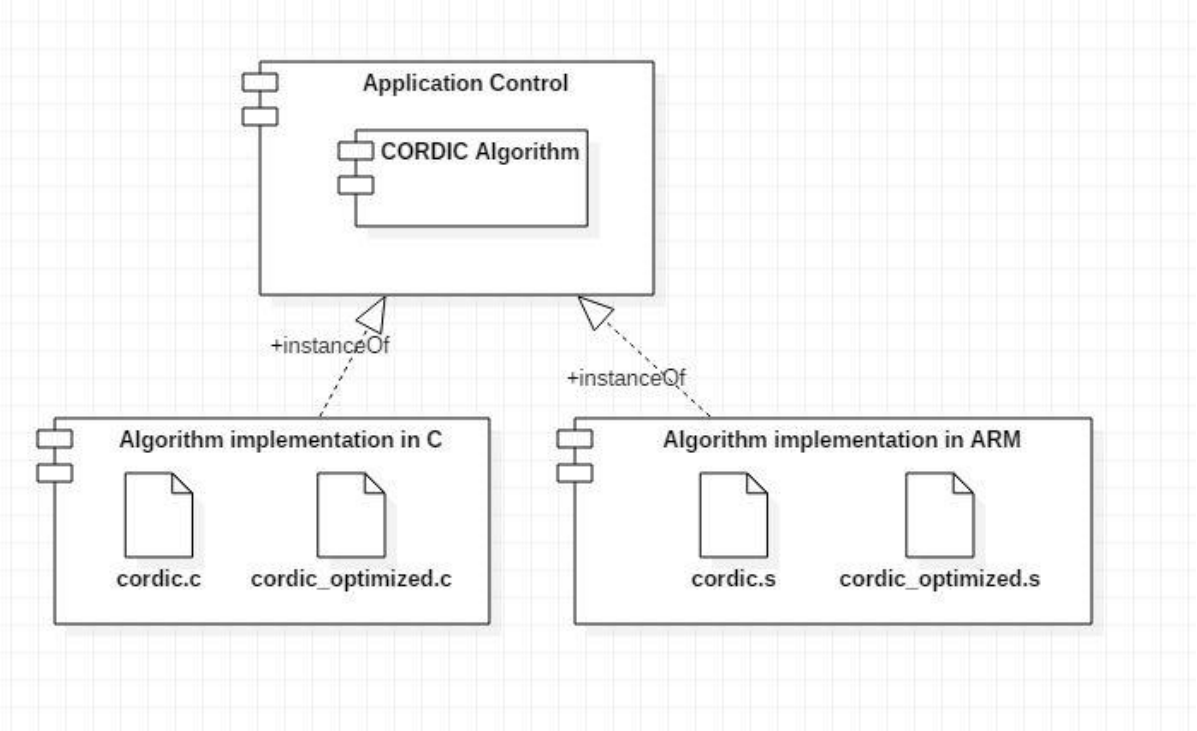


Figure 2 - CORDIC component diagram

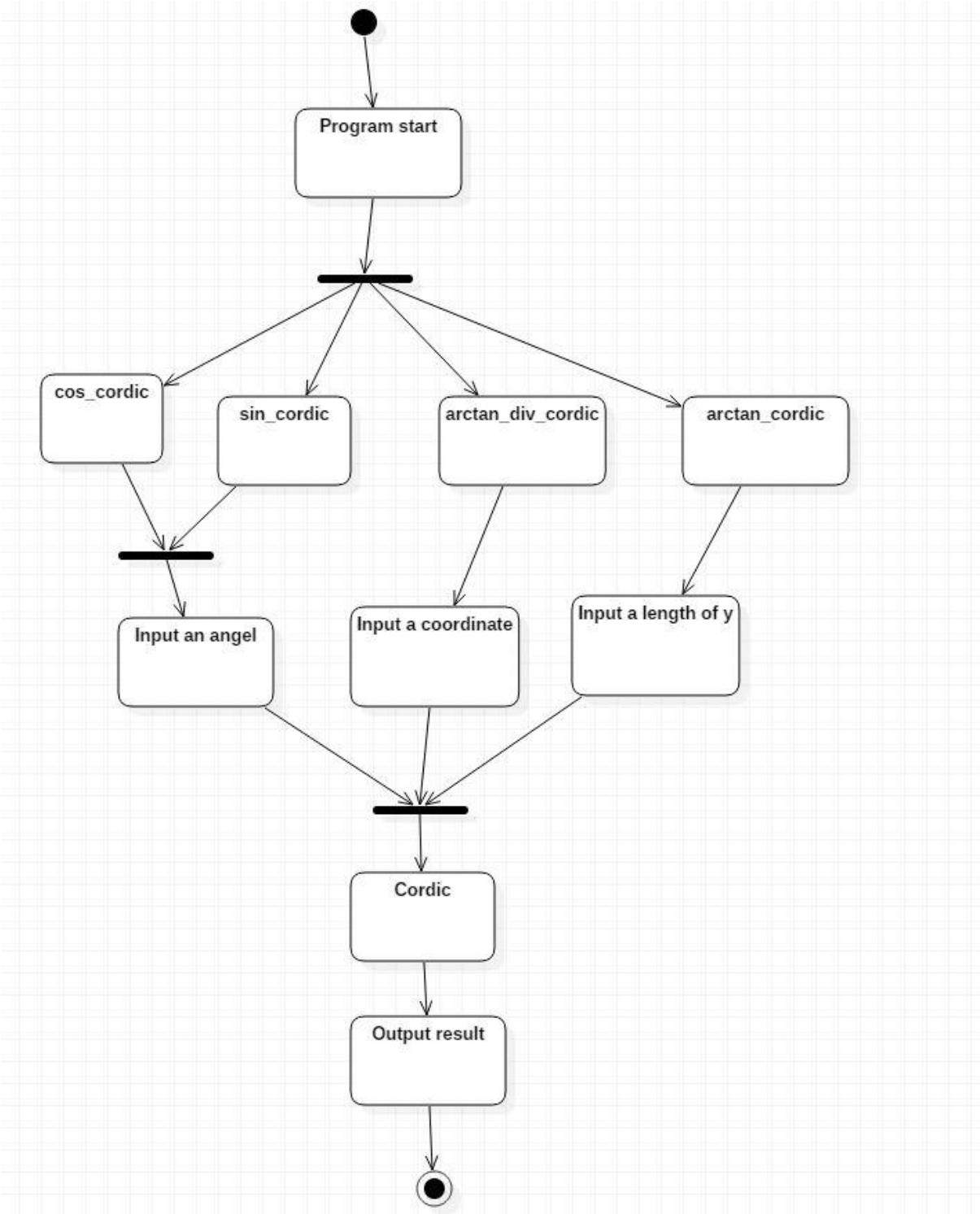


Figure 3 - Activity diagram for cordic.c

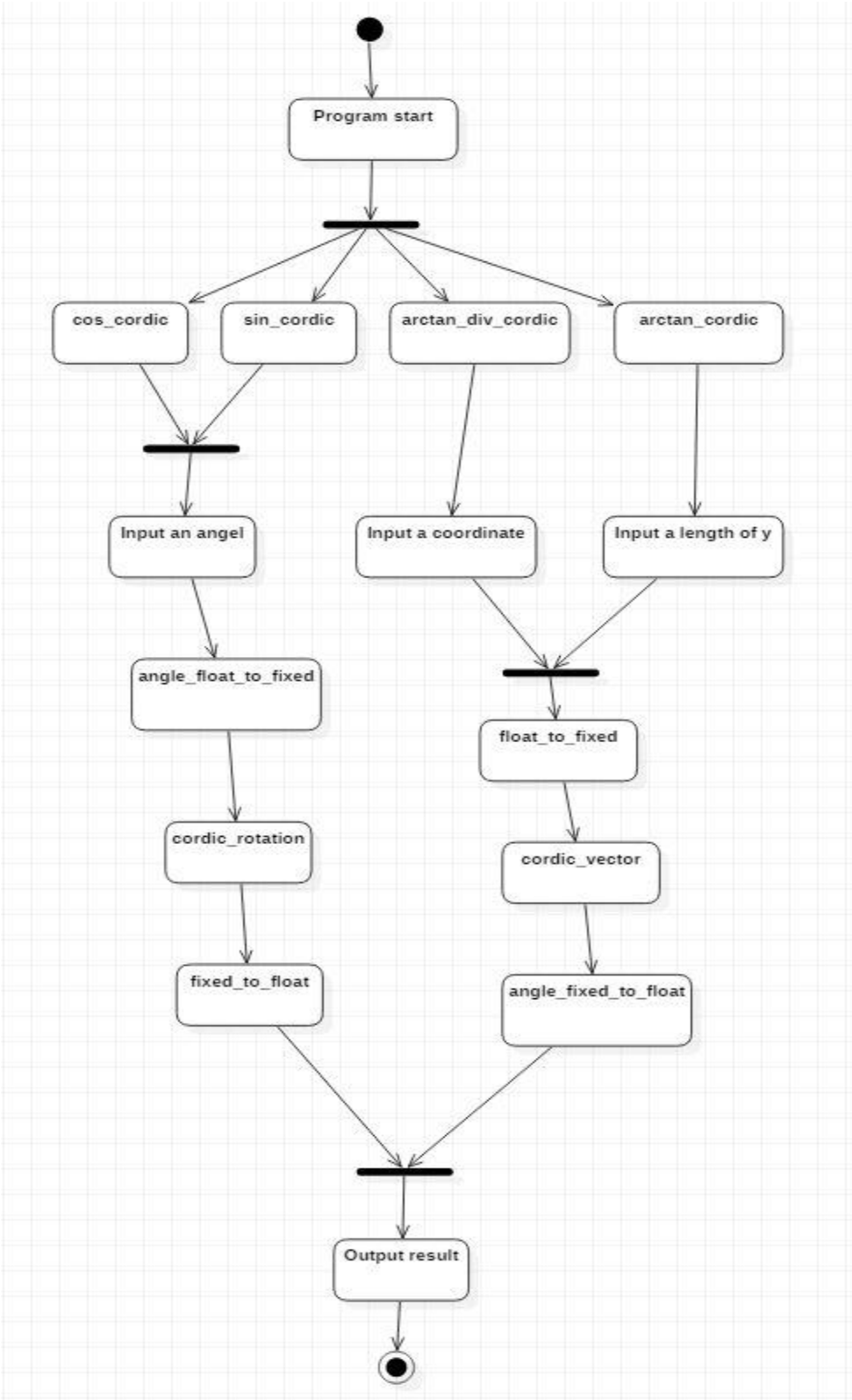


Figure 4 - Activity diagram for cordic_optimized.c

Performance/cost evaluation

In the main function of both *cordic.c* and *cordic_optimized.c*, we add time stamps in core cordic functions and obtain the performance data by calculating 102400 times of some trigonometric functions with cordic. The code and output data are below.

```
139
140  int main()
141  {
142      clock_t start = clock();
143      int i;
144      double test = 0;
145      for(i = 0; i < 102400; i++){
146          test = cos_cordic(1.0);
147          test = cos_cordic(20.0);
148          test = cos_cordic(30.0);
149          test = cos_cordic(45.0);
150          test = sin_cordic(40.0);
151          test = sin_cordic(90.0);
152          test = arctan_div_cordic(1.0, 4.0);
153          test = arctan_div_cordic(1.0, 2.0);
154          test = arctan_div_cordic(1.0, 6.0);
155          test = arctan_cordic(1.0);
156          test = arctan_cordic(2.0);
157          test = arctan_cordic(3.0);
158      }
159      clock_t end = clock();
160      printf("running time of main function: %fms\n", 1000*(double)(end - start) / CLOCKS_PER_SEC);
161      printf("total running time of cordic: %fms\n", time_cordic*1000);
162      printf("avg running time of cordic: %fms\n", 1000*time_cordic/count_cordic);
163      return 0;
164  }
```

Figure 5 - Main function of cordic.c

```c
int main()
{
    clock_t start = clock();
    int i;
    double test = 0;
    for(i = 0; i < 102400; i++){
        test = cos_cordic(1.0);
        test = cos_cordic(20.0);
        test = cos_cordic(30.0);
        test = cos_cordic(45.0);
        test = sin_cordic(40.0);
        test = sin_cordic(90.0);
        test = arctan_div_cordic(1.0, 4.0);
        test = arctan_div_cordic(1.0, 2.0);
        test = arctan_div_cordic(1.0, 6.0);
        test = arctan_cordic(1.0);
        test = arctan_cordic(2.0);
        test = arctan_cordic(3.0);
    }
    clock_t end = clock();
    printf("running time of main funciton: %fms\n", 1000*(double)(end - start) / CLOCKS_PER_SEC);

    printf("total running time of cordic_rotation: %fms\n", time_cordic_rotation*1000);
    printf("avg running time of cordic_rotation: %fms\n", 1000*time_cordic_rotation/count_cordic_rotation);

    printf("total running time of cordic_vector: %fms\n", time_cordic_vector*1000);
    printf("avg running time of cordic_vector: %fms\n", 1000*time_cordic_vector/count_cordic_vector);

    return 0;
}
```

Figure 6 - Main function of cordic_optimized.c

```
jinminh@linux1:~/Desktop/seng440/new$ gcc cordic.c
jinminh@linux1:~/Desktop/seng440/new$ ./a.out
running time of main function: 1206.227000ms
total running time of cordic: 833.426000ms
avg running time of cordic: 0.000678ms_
```

Figure 7 - Performance time of cordic.c

```
jinminh@linux1:~/Desktop/seng440/new$ gcc cordic_optimized.c
jinminh@linux1:~/Desktop/seng440/new$ ./a.out
running time of main funciton: 870.193000ms
total running time of cordic_rotation: 224.535000ms
avg running time of cordic_rotation: 0.000365ms
total running time of cordic_vector: 220.206000ms
avg running time of cordic_vector: 0.000358ms
```

Figure 8 - Performance time of cordic_optimized.c

The difference of performance time is following.

	Before optimization (ms)	After optimization (ms)	Time ratio (After/Before)
Main function	1206.227	870.193	72.14%
Total running time of cordic	833.426	444.741	50.24%
Avg time of cordic(rotation)	0.000678	0.000365	53.83%
Avg time of cordic(vector)	0.000678	0.000358	52.80%

Conclusion

Generally speaking, we learned a lot from SENG440 course and we finished almost all the requirement in project. Prior to this class, our group had almost no experience in software optimization. Through the project design and completion, we are able to apply various methods to optimize software in both C and assembly level. Besides, we can partially achieve required functionality in hardware level such as designing state machine and summarizing VHDL behavior (we tried to implement VHDL in this project but unfortunately we failed) as well. After the optimization, the efficiency of the CORDIC

is increased significantly according to the time cost evaluation. Overall, the Cordic project is successful and it is an excellent learning experience.

Reference

[1] Wikipedia. CORDIC. [Online].

Available: https://en.wikipedia.org/wiki/CORDIC

[2] Wikipedia. Loop unrolling. [Online].

Available: https://en.wikipedia.org/wiki/Loop_unrolling