COMPE 271 FINAL PROJECT REPORT

Calculating Sine Using the CORDIC Algorithm

Prof. Ken Arnold

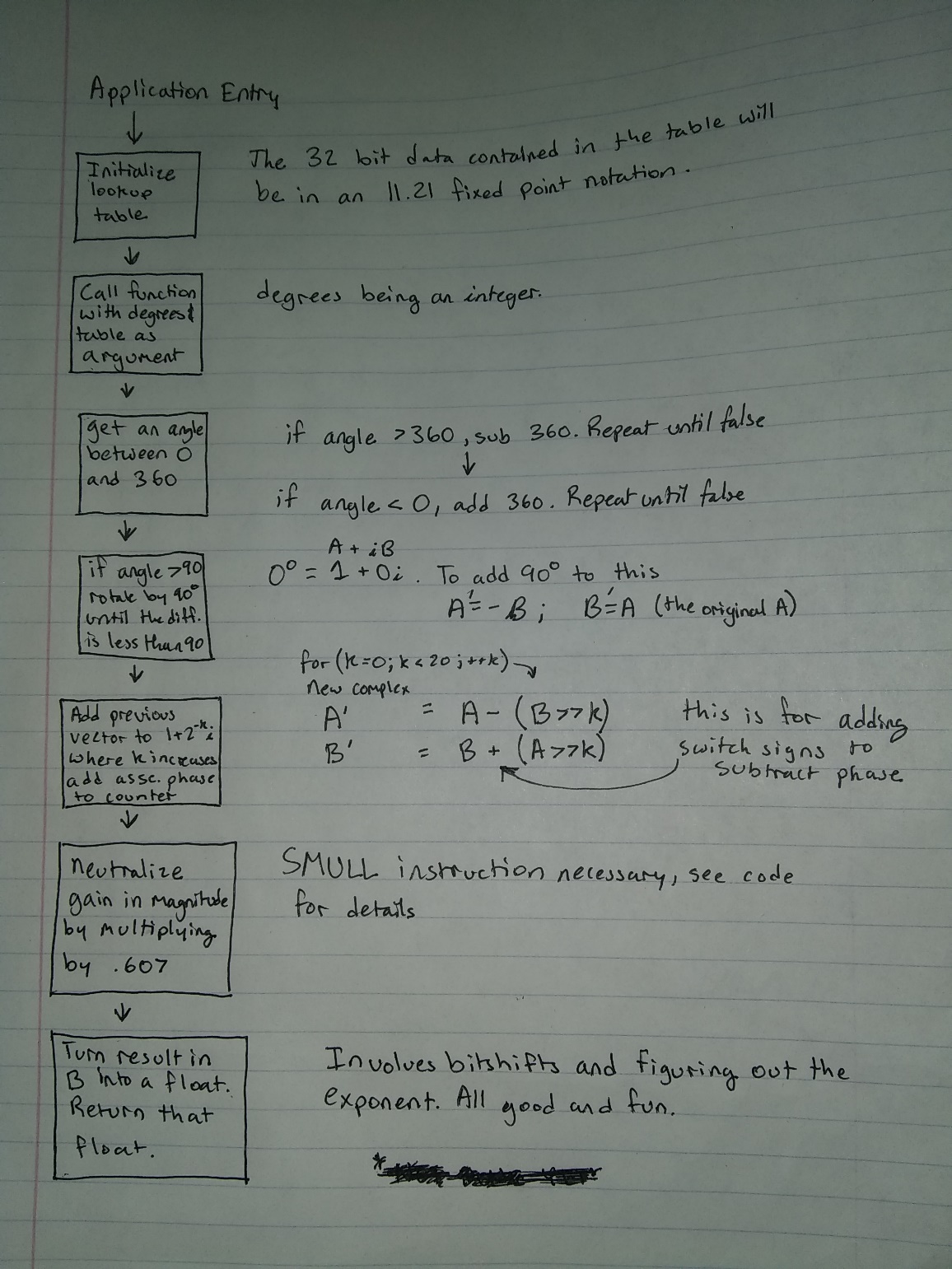
Ethan Cua

822636432

**Introduction**

The intent of this program is to calculate the value of sin for a given angle in degrees using the CORDIC algorithm. This algorithm uses the complex number form A + iB and, starting with a reference vector in the imaginary number form, adds vectors to it until the angle of the vector matches the angle we are trying to calculate the sin of. In this final vector of matching angle, the value of A is the value of sin, without accounting for magnitude gain from the function. Almost everything except printing the angle and creating the lookup table is done in Assembly.

For further explanation on the math behind the CORDIC algorithm, please see the link provided at the end.

**Diagram and Code Description**

CORDIC in a nutshell

This flowchart was ultimately broken into five functions, called by a sixth main function that also contains the CORDIC algorithm itself, which is in the middle of the flowchart. There is also a seventh function written in C that simply creates an array of integer values that make up the “lookup table” for the overall function. I will be discussing each function as it appears from top to bottom in the flowchart.

The first step was to prepare a lookup table of angles used in the CORDIC algorithm to be used. This is because for each iteration, the final complex number we eventually get sin off of is rotated by multiplying it by another complex number. This second complex number takes the form of A + iB, where A is always 1, and B is 2^k, where k is the kth iteration of the function. Each one of these complex numbers we are rotating by has an associated angle, and we need to “add” or “subtract” that angle in order to match the angle that the user has passed through successive iterations. This function literally just prepares an array of 20 integers, each one in the 11.21 fixed point notation. This array will be passed to the rest of the function in Assembly.

The next portion of the program is done in Assembly. The main function is called with the integer angle and the array pointer as parameters, and this branches again into a function which corrects the angle of the integer argument passed. If the angle is greater than 360, it is reduced to an angle between 0 and 360 inclusive. If the angle passed is less than zero, it is increased to an angle between 0 and 360.

Next, a function checks what quadrant of the unit circle the previously corrected angle falls in. If it is greater than 180 degrees, a flag is pushed onto the stack for use later that has 2 values, 0 (for positive) or 1 (for negative).

After this is done, another function is called, which translates the angle (given between 0 and 360 now) to a reference angle between 0 and 90, which has the same sin value as the original angle. This solves the issue of rotating the CORDIC vector used later more than 90 degrees.

Once this is done, we can finally get to the bread and butter of the CORDIC algorithm. It was touched on briefly in the introduction, but it will be explained here to the best of my ability. The CORDIC algorithm basically uses properties of complex numbers of the form A + Bi to calculate the value of sin. These numbers have both angles and magnitudes and will thus be referred to as vectors. Multiplying two complex numbers adds their angles and multiplies their magnitudes. The CORDIC algorithm multiplies a starting vector with a vector of form 1 + 2^- k (or 1 – 2^-k for subtraction), where k is the kth iteration of the algorithm. Each of these vectors have an angle, and this angle is added or subtracted to the angle of the first complex number, which is kept track of separately using the lookup table we had previously established. Essentially, we are waiting for the angle of the CORDIC vector to match the angle passed as an argument, at which point the value of B (which corresponds to sin) is the sin of that angle, albeit with some gain due to the multiplication of magnitudes. This algorithm iterates up to 20 times, after which point a value is returned if the angle does not already match.

After the CORDIC algorithm is complete, we must correct the magnitude gain of the function. Since the vector we are multiplying by has a magnitude slightly greater than 1 (although it does approach 1 with successive iterations), the magnitude of the CORDIC vector which contains the desired sin value is greater than 1, meaning that the value of sin contained inside is also greater than we want. To correct this, we must multiply by the inverse of this cumulative gain (1 / 1.647 = ~.607). The next portion of the function multiplies by this amount, using the SMULL instruction as the result would be larger than 32 bits. It also places the result of this operation into one register, retaining the fixed-point notation used up to this point.

The final portion of this program translates the fixed-point number generated into a floating-point number. This is done by calculating the number of shifts required to move the fixed-point number’s leading 1 into bit 24, which is the bit after the mantissa ends. That bit in bit 24 is then zeroed, and then the number of shifts required to move the number into place is subtracted from bias of 127, and this number is put into the exponent bits. Finally, the flag for positive or negative is popped from the stack and placed into the MSB, where the sign bit goes. This is the value which is ultimately returned from the function.

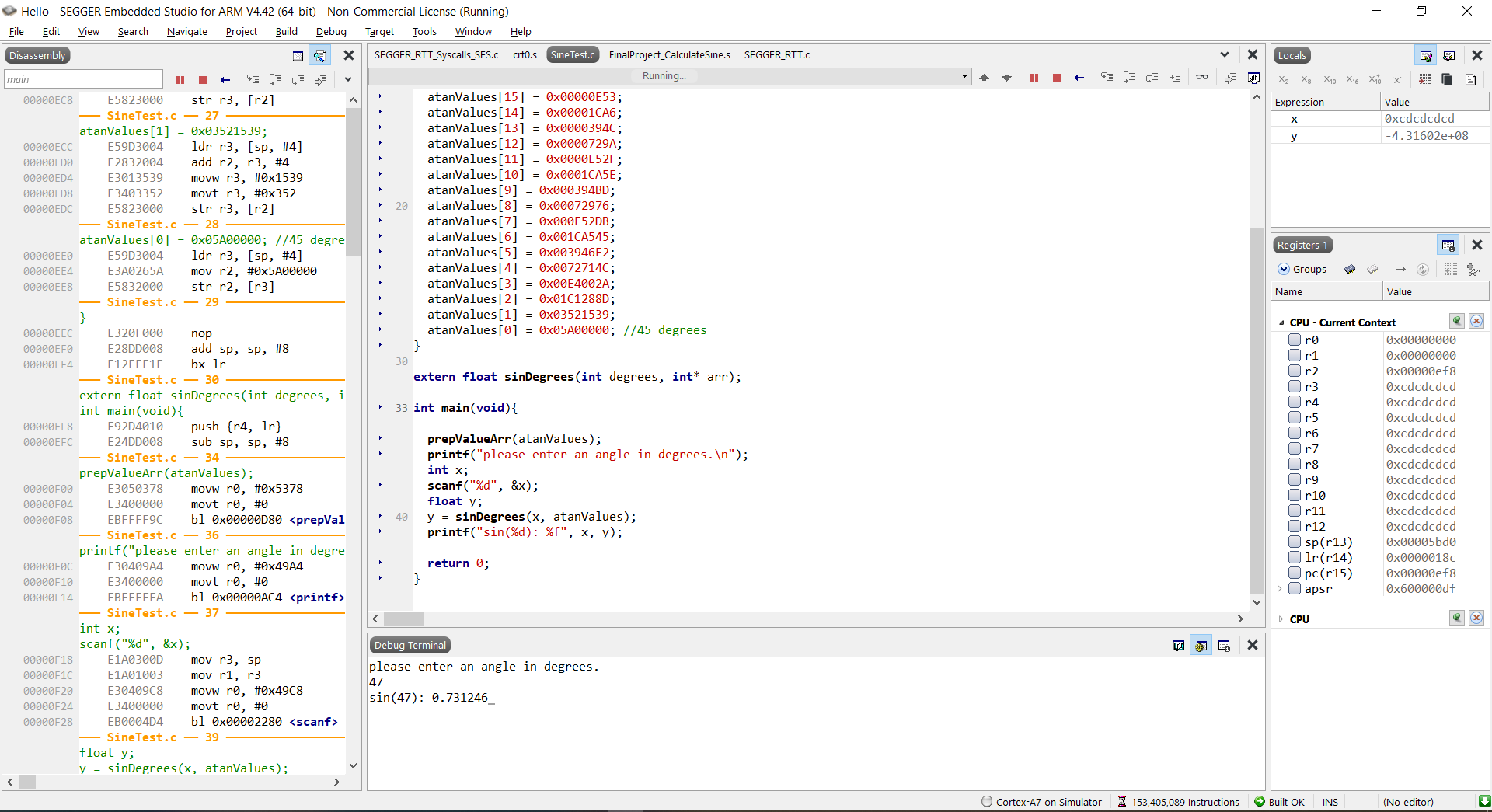
**Toolset Used**

Windows laptop using Segger Embedded Studio, MINGW GCC compiler.

**User Instructions**

On startup, simply type the angle you would like the sin of in degrees as a whole number, then hit enter and wait for output. That’s it.

**Test Screen Capture**



Here is a screen capture for my code operating. For reference, the value of sin(47) as given by my TI-83 is .7313537016.

**Test results**

The results were not as promising as hoped. The output of the program is correct, but only to about 3 digits out. Beyond 3 decimal places, there were errors in the number reported versus the actual value. This does indicate there is some imprecision in the code, somewhere.

A possible source of this could be the angles created by the lookup table, as they were not especially precise when initially calculated, and also lost some places during the conversion between decimal and the fixed-point notation. Another source could be the use of an approximate inverse for the CORDIC gain that was multiplied to the result of the algorithm to normalize it. The constant used to normalize this gain was only represented out to roughly 3 decimal places in precision, and so may contribute to some error in the results.

**If I had more time…**

I would only really like to clean up my code, as I developed it in multiple sessions. Each time, I replaced a “stub” saying where a function had to go with the function itself. This helped keep it organized, but there are still portions of the code that are unnecessary (specifically the use of so many local register variables). There are also parts of the code that seem pointless but are in fact necessary because I was trying to work around existing code rather than choosing to rewrite it.

The fixed-point system I had chosen to use hampered my efforts quite a bit, especially because it was difficult to read. In retrospect, a 12.20 fixed point system would be easier since it can easily be read in hexadecimal for debugging purposes.

Of course, I would also like to fix sources of imprecision in the algorithm.

**Conclusion**

I underestimated the complexity of this project. There was a lot of overhead I didn’t expect when I started this project, like the creation of a fixed-point number system , converting that number into a floating-point number, rotating the passed argument into range of the CORDIC algorithm, et cetera. All said, though, this was a fun and interesting experience, and whacked some concepts very firmly into my noggin.

**Time Spent**

I spent 10 -15 hours on this project, not including any time spent on any status / final report creation.

**Source(s)**

<https://dspguru.com/dsp/faqs/cordic/>

<https://en.wikipedia.org/wiki/Single-precision_floating-point_format>

**C Code**

#include <stdio.h>

#include <stdlib.h>

int atanValues[20];

void prepValueArr(int\* atanValues){

//the following are precalculated values of atan(K) as 32 bit integers. They represent the angle for each iteration of CORDIC algorithm in the

//fixed-point notation specified in the assembly file used here. These are not very precise calculations, leading to significant error.

atanValues[19] = 0x000000E6;

atanValues[18] = 0x000001CA;

atanValues[17] = 0x00000394;

atanValues[16] = 0x0000072B;

atanValues[15] = 0x00000E53;

atanValues[14] = 0x00001CA6;

atanValues[13] = 0x0000394C;

atanValues[12] = 0x0000729A;

atanValues[11] = 0x0000E52F;

atanValues[10] = 0x0001CA5E;

atanValues[9] = 0x000394BD;

atanValues[8] = 0x00072976;

atanValues[7] = 0x000E52DB;

atanValues[6] = 0x001CA545;

atanValues[5] = 0x003946F2;

atanValues[4] = 0x0072714C;

atanValues[3] = 0x00E4002A;

atanValues[2] = 0x01C1288D;

atanValues[1] = 0x03521539;

atanValues[0] = 0x05A00000; //45 degrees

}

extern float sinDegrees(int degrees, int\* arr);

int main(void){

prepValueArr(atanValues);

printf("please enter an angle in degrees.\n");

int x;

scanf("%d", &x);

float y;

y = sinDegrees(x, atanValues);

printf("sin(%d): %f", x, y);

return 0;

}

**Assembly Code**

//NOTE the general structure for this program has been copied from the template given to us in Hw #5. As such, it should be noted that not all

//of this code is completely original work, the overhead of moving the stack pointer and pushing the link register were copied.

//NOTE 2 this program uses a fixed-point notation xxxxxxxxxxx.xxx...xx, with 11 bits before the point, and 21 after. This will be referred to as 11.21 notation

//NOTE 3 this function only accepts INTEGERS as arguments. This may change in the future.

.global sinDegrees

.data

.text

sinDegrees:

mov r12,r13 // save stack pointer into register r12

sub sp,#512 // reserve 512 bytes of space for local variables, specifically for the lookup table

push {r4} // freeing registers for use

push {r5}

push {r6}

push {r7}

push {r8} //just here to be a temp register to pop the lr into for functions this calls

push {lr} // push link register onto stack -- make sure you pop it out before you return

//r0 contains the angle (in degrees) that we want to match.

//r1 contains the pointer to the array of angles

mov r7, r1 //array pointer moved to r7 because r1 was a local variable before this.

mov r1, #0 //r1 is the angle accumulation value. Values from the stack will get added or subtracted to it, as well as +-90 degree rotations.

mov r2, #0x00200000 //this is 1 in the fixed point representation I am using. r2 contains A (cos)

mov r3, #0 //represents iB for the starting number (sin). This register contains the value to be returned (eventually).

bl normalizeAngle //function to get an inital angle between 0 - 360, inclusive.

bl isNegative //function to check whether the sin at this quadrant is positive or negative

bl getReferenceAng //gets a reference angle between 0 - 90, which has a sin value equal to the one for the original value.

//The following loop code executes the cordic algorithm with shifts to calculate the sin value of the angle.

//For angle operations, since the angle will always be positive, we don't have to worry about the whole deal with over/underflow

mov r5, #0 //r5 is now a counting variable

lsl r0, #21 //shift the value. r0 now (finally) contains a value in our 11.21 fixed point notation to be used in comparisons

while: cmp r5, #20 //loop control logic. Iterates 20 times. (0 to 19)

beq done

cmp r0, r1 //compares desired angle with the current rotated angle. If greater, the next iteration must subtract phase, and vice versa.

beq done //if r0 = r1, we're done here.

bgt addAng

//if branch not taken, we need to subtract the next angle

mov r4, r2 //temp move A value to r4, as r2 will be modified

lsr r6, r3, r5 //r5 contains the iteration of the algorithm, which is also the number of places needed for a shift

add r2, r2, r6 //A' = A + iB^2-k

lsr r6, r4, r5 //r6 is temp, r4 is the old value of A

sub r3, r3, r6 //iB' = B - A\*2^-k

ldr r4, [r7], #4 //gets angle for current iteration from r7 (pointer to angle array), then increments r7 by 4 to point to next "int"

sub r1, r1, r4 //subtracts angle to r1 (because we rotated downwards).

add r5, r5, #1 //loop increment

b while

addAng://branch taken, the next angle must be added.

mov r4, r2 //temp move A value to r4, as r2 will be modified

lsr r6, r3, r5 //r5 contains the iteration of the algorithm, which is also the number of places needed for a shift

sub r2, r2, r6 //A' = A - iB^2-k

lsr r6, r4, r5 //r6 is temp, r4 is the old value of A

add r3, r3, r6 //iB' = B - A\*2^-k

ldr r4, [r7], #4 //gets angle for current iteration from r7 (pointer to angle array), then increments r7 by 4 to point to next "int"

add r1, r1, r4 //adds angle to r1

add r5, r5, #1 //loop increment

b while

done:

bl reduceGain //this function works to reduce magnitude gain due to CORDIC function.

bl FixedToFloat //only works for the fixed point notation used in other parts of the program

pop {r1} // pop link register from stack into r1

pop {r8}

pop {r7}

pop {r6} // restoring registers for parent function

pop {r5}

pop {r4}

mov lr, r1 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the

// value it had when you entered the function .

bx lr // return from the function by copying link register into program counter

//Takes an integer and turns it into another integer between 0 and 360.

.global normalizeAngle

.data

.text

normalizeAngle:

push {lr} // push link register onto stack -- make sure you pop it out before you return

//loops to normalize the angle, so that r0 contains an angle between 0 and 360

fang: cmp r0, #360

ble dfang //if the value is less than 360, finish

sub r0, r0, #360

b fang

dfang:

rang: cmp r0, #0 //is value less than 0, add 360 to it

bge drang

add r0, r0, #360

b rang

drang:

pop {r8}

mov lr, r8 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

bx lr // return from the function by copying link register into program counter

//isNegative function starts here. Checks if the sin of this angle should be negative or positive

//and pushes a flag representing this onto the stack. 0 for positive, 1 for negative.

//the push and pop instructions are there solely as a legacy portion of code. It works, I'm leaving it.

.global isNegative

.data

.text

isNegative:

push {lr} // push link register onto stack -- make sure you pop it out before you return

cmp r0, #180 //if greater than 180, the sin value must be negative and this flag is pushed onto the stack to be popped.

ble posAng

mov r4, #1

pop {r5} //temp pop link register to put r4 below it.

push {r4} //the 1 is pushed if this is negative, is popped later.

push {r5} //re-push r5 after r4 is pushed to maintain lr on top of the stack

b flagSet

posAng:

mov r4, #0

pop {r5}

push {r4} //0 is pushed otherwise because a pop is used later.

push {r5}

flagSet:

pop {r8}

mov lr, r8 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

bx lr // return from the function by copying link register into program counter

.global getReferenceAng

.data

.text

getReferenceAng:

push {lr} // push link register onto stack -- make sure you pop it out before you return

//this portion of the code moves the angle to a reference value between 0 and 90

cmp r0, #90

ble doneRef

cmp r0, #180

mov r4, #180

suble r0, r4, r0 //gets reference angle if angle is between 90 and 180

ble doneRef

mov r4, #270

cmp r0, r4

suble r0, r0, #180 //gets reference angle if angle is between 180 and 270

ble doneRef

//if the previous statements didn't trigger, the angle must be between 270 and 360.

mov r4, #360

sub r0, r4, r0

doneRef:

pop {r8}

mov lr, r8 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

bx lr // return from the function by copying link register into program counter

//reduces gain from CORDIC algorithm

.global reduceGain

.data

.text

reduceGain:

push {lr}

//This next part of the code is intended for reversing the magnitude gain from the CORDIC algorithm.

//0x00136DE4 is ~.607 (the inverse of the accumulated gain) in the chosen binary fixed point. Moved in 3 instructions.

mov r1, #0x000000E4

orr r1, r1, #0x00006D00

orr r1, r1, #0x00130000

smull r4, r5, r3, r1 //r5 contains the 32 msb's; r4 the 32 lsb's of the operation. r3 from calling function is sin, r1 is the gain number.

lsl r5, r5, #11 //contents in the top 32 bits are shifted to align with the 11.21 notation

//moving stuff into r1 for the next operation

mov r1, #0xFF000000 //needed for AND operation in next instruction

orr r1, #0x00E00000

and r4, r4, r1 //bitmasks the 11 msb's in r4

lsr r4, r4, #21 //shifts them to be the lsb's

orr r5, r5, r4 //tacks those onto the end of r5

//r5 now contains the value of sin with a proper magnitude in the 11.21 notation being used.

pop {r8}

mov lr, r8 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

bx lr // return from the function by copying link register into program counter

//This function turns the number passed (which is in 11.21 fixed point) into a standard single-precision float.

.global FixedToFloat

.data

.text

FixedToFloat:

push {lr} // push link register onto stack -- make sure you pop it out before you return

//The following lines of code translate the result in r5 into a single-precision float

lsl r5, r5, #2 //The purpose of this shift is to line up the decimal point of the 11.21 fixed notation to where the mantissa would be in a single

//precision float (bits 23 - 0). This allows for further shifts to be done to determine the value of the exponent bits without overcalculation.

//This was a problem in my code, where the exponent would actually be smaller than what it should be after calculation.

mov r2, #0xFF

orr r2, #0xFF00

orr r2, #0xFF0000

orr r2, #0xFF000000 //this register is used for inverting and bitmasking later.

mov r3, #0 //counting var for later, as comparison is done.

shiftLoop:

cmp r5, #0x00800000 //if a 1 is encountered in this position, this is the 1 for 2^0, i.e. the mantissa is in the correct bit place.

bge doneFloat

cmp r5, #0 //special condition for if 0 is the result of the algorithm (multiples of 180 passed as arg)

popeq {r8} //if equal to 0, things have to be popped properly and then we return to caller.

popeq {r1}

pusheq {r8}

beq noSignBit

lsl r5, r5, #1

add r3, r3, #1

b shiftLoop

doneFloat:

and r2, r2, #0xFF7FFFFF //the next few instructions zero the bit in bit 23, where the hanging 1 before the mantissa is. uses R2

and r5, r5, r2 //zeroing instruction

mov r4, #127 //calculating exponent bits and shifting them into position

sub r3, r4, r3

lsl r3, r3, #23 //shifts r3 into position to be ORRed with r5

orr r5, r5, r3 //exponent bits are finally placed in r5

pop {r8} //so the link register was on top of the stack so I popped it and will push it back.

pop {r1} //negative bit flag is finally popped from the stack

push {r8}

cmp r1, #1 //if equal, we did invert the number, and it was originally negative.

bne noSignBit

orr r5, r5, #0x80000000 //places a 1 in sign bit to let us know we're in the negative

noSignBit: //if no sign bit in number, we're done already

mov r0, r5 //finally, move our result into r0 for printing and stuff.

pop {r8}

mov lr, r8 // pop operation did not allow for pops into lr, so value of lr stored in r1 temporarily

bx lr // return from the function by copying link register into program counter

//the following code was written prior to some conceptual code changes simplifying the operation. This rotated the vector 90 degrees

//but was later found to be largely uneccessary. Furthermore, the algorithm loops itself didn't handle negative numbers especially well,

//which was the primary impetus for removing this section of code (which did work properly).

/\*

mov r5, r0 //temp movement to preserve r0 while the vector is rotated. This is for the next function

angle: cmp r0, #90 //if the angle given is greater than 90, we must rotate the cordic vector until the phase difference is less than 90

blt fangle

mov r6, r3 //push r3 (B) to a temp variable. it needs to be inverted before being put into r2

mov r3, r2 //move r2 to r3

//the next few instructions are used to invert the number in r2. using a 32 bit immediate is impossible in this case.

mov r4, #0xFF

orr r4, #0xFF00

orr r4, #0xFF0000

orr r4, #0xFF000000

eor r2, r6, r4

add r2, r2, #1 //2's comp things.

//end inversion.

add r1, #0x0B400000 //adds 90 to angle accumulation value (using our fixed point notation), to make sure that we are matching r0 (which will be restored later in the program)

sub r0, #90 //after rotation, subtract 90 to see if another rotation needs to occur or not. r0 is still an int at this point.

b angle

fangle:

mov r0, r5 //restore r0

mov r5, #0 //r5 will be used as a counting variable for the next function

\*/