**CMSC 411 Final Report: CORDIC Algorithm in ARM**

**Team Members: Ben Hazlett, Evan Andre, Stephanie Tam**

**Total Computer Cycles**

branch - 3 cycles compare - 1 cycle

add, sub, mov, shift (normal) - 1 cycle shift (from register) - 2 cycles

load - 2 cycles load (offset) - 3 cycles

store - 2 cycles store (scaled) - 3 cycles

These are the cycles per instruction (CPI) that we pulled from the ARM Information Center on instruction cycles. The information is for an ARMv7 machine and our simulator is for an ARMv5 machine, but this should still produce a reasonable estimate.

From these we went through our code and estimated the expected number of cycles for sine, cosine, and tangent:

* **sin, cos: 346 total cycles**(Note: These values were computed simultaneously as the values of sine and cosine were required during the algorithm. This is why they have the same number of instructions and the same number of total cycles.)
* **tan**: **238** **total cycles**

**Estimated CPI**

* 186 instructions to calculate sin(x) and cos(x)   
  CPI = 346 / 186 = *1.8602 cycles / instruction*
* 134 additional instructions to calculate tan(x)

CPI = (346 + 238) / (186 + 134) = *1.825 cycles / instruction*

**Estimated Total Processing Time**

Assume a system clock of the following:

* 32kHz:
  + sin and cos: 186 \* 1.8602 / 32kHz = *0.0108124125 seconds*
  + tan: (346 + 238) \* 1.825 / 32kHz = *0.03330625 seconds*
* 1MHz:
  + sin and cos: 186 \* 1.8602 / 1MHz = *0.0003459972 seconds*
  + tan: (346 + 238) \* 1.825 / 1MHz = *0.0010658 seconds*
* 1GHz
  + sin and cos: 186 \* 1.8602 / 1GHz = *0.000000345972 seconds*
  + tan: (346 + 238) \* 1.825 / 1GHz = *0.0000010658 seconds*

**Description of Implemented Algorithms**

**The CORDIC Algorithm for Sine and Cosine**

Sine and cosine are estimated through this algorithm using only shifts, adds, and subtracts. The key for this is the convergence on a specific angle by either adding to our target angle when our target angle is negative or subtracting from our target angle when our target angle is positive. The values for sine and cosine are initialized to 0 and 0.6072529350, respectively. To set up our table of angles, we take the *arctan*(2*i*), where *i* started at 0 and counted up to 11. Then, from there, the new cosine and new sine values are calculated as such:

if current angle < 0,

current angle += angle table[*i*]

new cosine = current cosine + (current sine >> *i*)

new sine = current sine – (current cosine >> *i*)

if current angle >= 0,

current angle -= angle table[*i*]

new cosine = current cosine – (current sine >> *i*)

new sine = current sine + (current cosine >> *i*)

current cosine = new cosine

current sine = new sine

The above is repeated until all of the angles in the lookup table have been used.

After the twelve iterations, the current cosine and current sine values will result in our estimate for our cosine and sine values for that angle. One important method we used, when implementing this algorithm in ARM, is shifting all of the numbers by 16 to the left (in other words, we multiply by 216). This is done to avoid having floating point values—and therefore, floating point registers—and instead, only use regular registers. Then, in the interpretation of the final value, we take the hexadecimal value stored in the register and interpret it as a decimal value.

**The Shift-Add/Subtract Algorithm for Division**

In order to calculate tangent, we implemented a simple shift and subtract algorithm since ARMSim does not support the SDIV/UDIV instructions. To begin, we take the calculated sine and cosine values and prepare them for the shift-add algorithm by shifting our sine value by 4 to the right to avoid potential overflow. Then, we have a loop where we compare if cosine is greater than sine, shifting cosine to the left until cosine cannot be shifted anymore without being larger than sin. To keep track of the number of times cosine was shifted, we stored the value 1 in another register and then shifted it equal to the number of times we shifted cosine. Tangent’s initial value is set to zero. We call this value shifter. After all of that the algorithm is set up to start the shifting and subtracting as follows:

while cos > 0

if sine > cosine

sine = sine – cosine

tangent += shifter

shifter = shifter >> 1

cosine = cosine >> 1

The above will continue to loop until sine becomes less than cosine. After the execution of the above loop we store our values into memory, sine, cosine, and tangent.

An important point to note is that shifter had to be shifted to the left 16 bits to handle the fact that sine and cosine are also shifted sixteen bits. Since it started by shifting sine right 4, it ends by shifting tangent left 4 bits.

**Resources Used**

1. CORDIC for Dummies

* used to understand algorithm more easily
* <http://bsvi.ru/uploads/CORDIC--_10EBA/cordic.pdf>

1. ARM Information Center

* used for instruction documentation
* <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/index.html>

1. ARMSim User Guide

* used to understand how to do certain things in ARMSim
* <http://armsim.cs.uvic.ca/AttachedFiles/ARMSim_UserGuide4Plus.pdf>

1. About Assembly Language – Division

* <http://www.tofla.iconbar.com/tofla/arm/arm02/index.htm>

1. Maximell/Cordic: Implementation of the Cordic algorithm

* <https://github.com/Maximell/Cordic>

1. Hex to Decimal converter

* Convert hexadecimal to decimal to verify decimal value of sine, cosine, and tangent
* <https://www.rapidtables.com/convert/number/hex-to-decimal.html>

**PLEASE SEE NEXT PAGE FOR SAMPLE INPUT AND OUTPUT**

**Sample Input & Output**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Angle (°)** | **Cosine Approx.** | | **Sine Approx.** | | **Tangent Approx.** | | **Actual Values** | | |
| **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Cosine** | **Sine** | **Tangent** |
| 5.2954 | 0000.fee3 | 0.9956512 | 0000.17be | 0.0927429 | 0000.17f0 | 0.0935058 | 0.9957321 | 0.0922906 | 0.0926862 |
| 80.1 | 0000.2c0a | 0.1720275 | 0000.fc2e | 0.9850769 | 0005.baf0 | 5.7302246 | 0.1719291 | 0.9851093 | 5.7294164 |
| 24.23 | 0000.e989 | 0.9122467 | 0000.68d4 | 0.4094848 | 0000.7310 | 0.4494628 | 0.9119053 | 0.4104005 | 0.4500473 |
| 48.52 | 0000.a96f | 0.6618499 | 0000.bfe9 | 0.7496490 | 0001.2230 | 1.1335449 | 0.6623585 | 0.7491869 | 1.3108972 |
| 10.001 | 0000.fc17 | 0.9847259 | 0000.2c88 | 0.1739501 | 0000.2d70 | 0.1774902 | 0.9848047 | 0.1736653 | 0.1763449 |
| 70.86159 | 0000.53f1 | 0.3278961 | 0000.f1d5 | 0.9446563 | 0002.e230 | 2.8835549 | 0.3278513 | 0.9447293 | 2.8815787 |
| 30.0 | 0000.ddbd | 0.8661651 | 0000.7ff0 | 0.4997558 | 0000.93f0 | 0.5778808 | 0.8660254 | 0.5 | 0.5773502 |
| 86.5 | .0b043615 | 0.0430330 | .efe84526 | 0.9371379 | 0015.c6f0 | 21.777099 | 0.0610485 | 0.9981347 | 16.349855 |
| 90 | 00000000 | 0 | 00000001 | 1 | 00000000 | 0 | 0 | 1 | UNDEF |
| 0 | 00000001 | 1 | 00000000 | 0 | 00000000 | 0 | 1 | 0 | 0 |
| 4.3667 | .efb4b5da | 0.9363511 | .0ec37009 | 0.0576696 | 0000.0fc0 | 0.0615234 | 0.9970970 | 0.0761407 | 0.0763611 |
| 45 | 0000.b4f6 | 0.7068786 | 0000.b513 | 0.7073211 | 0001.0050 | 1.0012207 | 0.7071067 | 0.7071067 | 1 |
| 60 | 0000.7ff0 | 0.4997558 | 0000.ddbd | 0.8661651 | 0001.bc70 | 1.7360839 | 0.5 | 0.8660254 | 1.7320508 |

**Table 1:** Table of sample input and output, along with the actual values to compare them against.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Angle (°)** | **Cosine Approx.** | | **Sine Approx.** | | **Tangent Approx.** | | **Actual Values** | | |
| **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Cosine** | **Sine** | **Tangent** |
| 88.667 | .021bd47a | 0.0082371 | .f0269e12 | 0.9380832 | 0156.5280 | 342.32226 | 0.0232631 | 0.9997293 | 42.974824 |
| 1.37217 | .f0269e12 | 0.9380832 | .021bd47a | 0.0082371 | 0000.0230 | 0.0085449 | 0.9997132 | 0.0239465 | 0.0239534 |
| 89.75 | .fd690019 | 0.9898834 | .eae56a18 | 0.9175630 | 0000.ed40 | 0.9267578 | 0.0043633 | 0.9999904 | 229.18166 |
| 3.68 | .efdcc932 | 0.9369267 | .0bf41465 | 0.0439647 | 0000.0cc0 | 0.0498046 | 0.9979380 | 0.0641839 | 0.0643165 |
| 4.103 | .efc3016c | 0.9365692 | .0dd3b42f | 0.0540115 | 0000.0ec0 | 0.0561718 | 0.9974370 | 0.0715496 | 0.7173352 |

**Table 2:** Table of sample input and output, but the approximations are noticeably different from the actual values.