Vlad Latushko

Tyler Kuczinski

CMSC 411 Fall 2018

**Cordic Algorithm in ARM**

Using the ARMSim application, we have written code to compute several complex values:

* sin(x)
* cos(x)
* sinh(x)
* cosh(x)
* ex

Our code makes use of the following ARM instructions, which are listed with their corresponding cycle count:

* LDR – 2 cycles
* MOV – 1 cycle
* CMP – 1 cycle
* BEQ – 3 cycles
* ADD – 1 cycle
* BLT – 3 cycles
* LSR – 3 cycles
* SUB – 1 cycle
* B – 3 cycles
* STR – 2 cycles

**Cycles per Instruction Calculations**

Below is a detailed analysis of the functions that exist within our code (for the purposes of this paper, a function will be the section of code that follows a label, ending where the next label begins):

* Main
  + 9 LDR instructions
  + 2 MOV instructions
* Loop
  + 2 CMP
  + 3 branch type instructions
  + 1 LDR
  + 2 LSR
  + 3 ADD
  + 2 MOV
  + 2 SUB
* Neg
  + 2 MOV
  + 2 LSR
  + 3 ADD
  + 1 SUB
  + 1 Branch
* End\_rot
  + 1 LDR
  + 2 STR
  + 1 ADD
* Start\_hyp
  + 5 LDR
  + 3 MOV
* Looph
  + 4 CMP
  + 4 Branches
  + 1 LDR
  + 3 ADD
  + 2 MOV
  + 2 LSR
  + 1 SUB
* Negh
  + 2 MOV
  + 2 LSR
  + 2 SUB
  + 2 ADD
  + 2 CMP
  + 3 Branch
* Rep
  + 1 CMP
  + 2 Branches
  + 1 ADD
* Yes
  + 1 ADD
  + 1 Branch
* End\_hyp
  + 1 LDR
  + 3 STR
  + 3 ADD
* Totals for the program

**Run Time**

Using the estimated cycles calculated above, we can estimate the total processing time the implemented algorithm takes a on specific architecture. We will examine three hypothetical systems clocked at 32KHz, 1MHz, and 1GHz each.

* 32KHz System
* 1MHz System
* 1GHz System

**Description of the Algorithm**

The Cordic algorithm is a convergence algorithm, where the cosine and sine values being calculated are determined by examining the current angle calculation against the target angle: when the current calculation is less than 0, then the value from our lookup table that corresponds to the iteration number is added to our running total. If the current running calculation is greater than or equal to zero, then the corresponding value from our lookup table is subtracted from our running total.

Our lookup table is comprised of 32 values, where the nth value is pre-computed as a *arctan(2n)*. Because our program does not use floating point values, we multiply all of our values by 65536 before they are stored. In addition, all of our output values are calculated and scaled up with the same value.

Below is a high-level, C-style pseudocode interpretation of our algorithm:

#define AG\_CONST 0.6072529350

#define FXD(X) ((long int)((X) \* 65536.0))

**static** **long** **int** Alpha[] = { FXD(45.0), FXD(26.565),

FXD(14.0362), FXD(7.12502), FXD(3.57633), FXD(1.78991),

FXD(0.895174), FXD(0.447614), FXD(0.223811), FXD(0.111906),

FXD(0.055953), FXD(0.027977) };

**long** **int** X, Y, CurrAngle;

**unsigned** i;

**for** (i = 0; i < 12; i++) {

**long** **int** NewX;

**if** (CurrAngle > 0) {

NewX = X - (Y >> i);

Y =+ (X >> i);

X = NewX;

CurrAngle -= Alpha[i];

}

**else** {

NewX = X + (Y >> i);

Y =- (X >> i);

X = NewX;

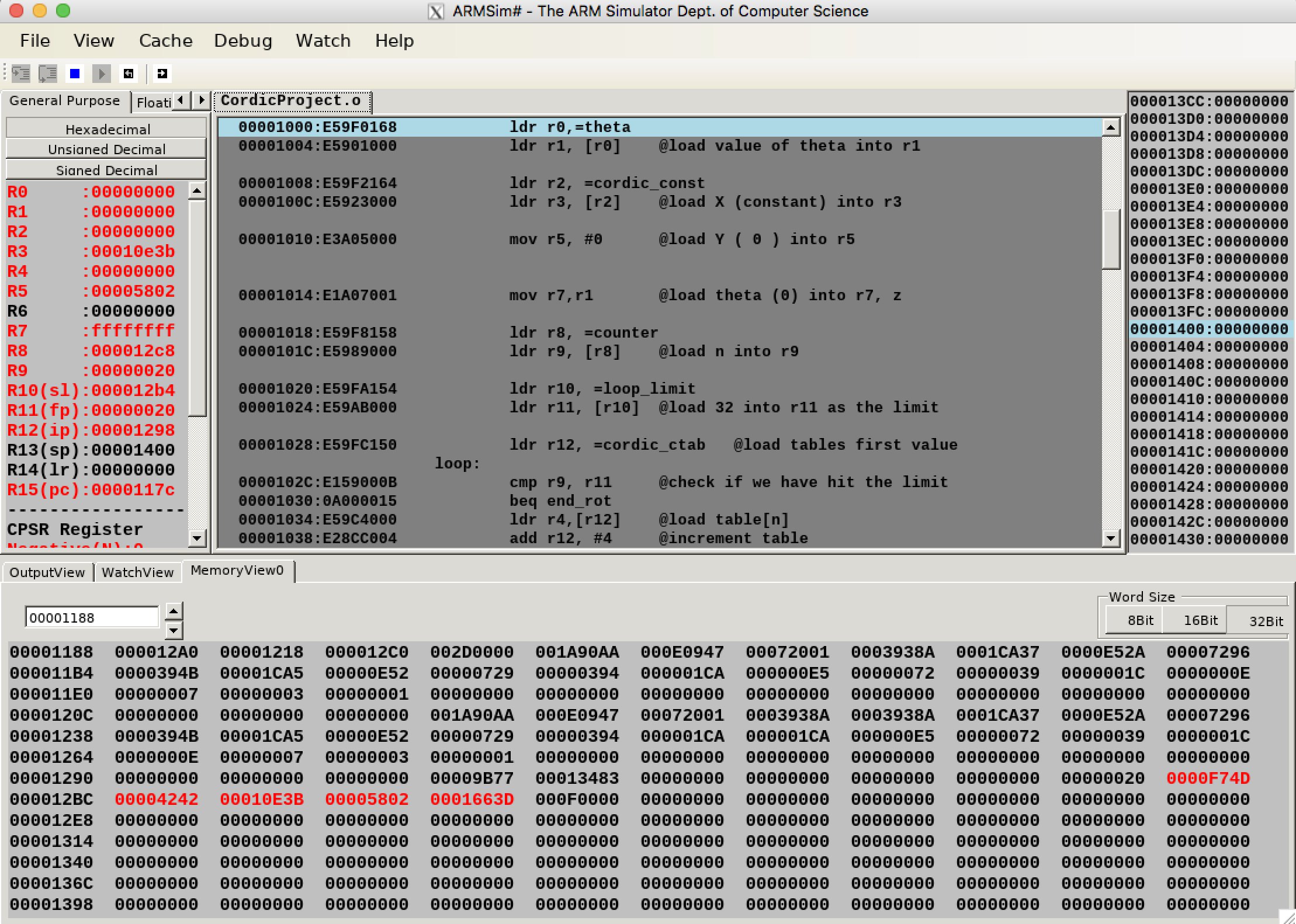
CurrAngle += Alpha[i];

}

}

**Sample Input and Output**

Here is a sample run of our program calculating the complex values:



In the screenshot of ARMSim, you can see the resulting execution of our program when running with a theta value of 15 degrees. At the bottom of the screen in the Memory View section, the red values are the calculated complex values, occurring in order: cos(x) = 0xF74D, sin(x) = 0x4242, cosh(x) = 0x10E3B, sinh(x) = 0x5802, and ex = 0x1663D. Since our project calculates these at integer values, we have provided a table to show the values when they are normalized back to their correct representations.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Program Value | Normalized Value\* | Correct Value |
| cos(x) | 0xF74D | 0.966018676757812 | 0.965925826289068 |
| sin(x) | 0x4242 | 0.258819580078125 | 0.258819045102520 |
| cosh(x) | 0x10E3B | 1.055587768554687 | 1.034465640095510 |
| sinh(x) | 0x5902 | 0.347686767578125 | 0.264800227602270 |
| ex | 0x1663D | 1.399368286132812 | 1.299265867697781 |

\*Normalize value calculation is done by taking the hexadecimal value, converting it to decimal, and then dividing by 65,536.

It is easy to observe the relative accuracy of this algorithm, given that it uses no multiplication instructions at all. With no requirements for multiplication support, this makes the algorithm ideal for limited architecture types. The values are not exact due to rounding errors, as we drop decimal values when converting our constants floats in our lookup table to large integer values. However, the speed at which the values are calculated and the relative accuracy makes up for that deficiency.