Assembly Language Math Co-processor Efficiency Study and Improvements on a Multi-Core Microcontroller

Matthew Lang and Adam Stienecker
Ohio Northern University, m-lang@onu.edu, a-stienecker.1@onu.edu

***Abstract* - Traditionally, a math co-processer is a hardware device that resides next to a microprocessor or microcontroller on a circuit board. The math co-processor is there to complete complex and lengthy mathematical processes such that the main processor is not bogged down. With the rising popularity and falling price of multi-core microcontrollers such as the Parallax Propeller, the interest in on-board math co-processors has increased. This paper aims to document efforts of a work-in-progress to improve upon an assembly language math co-processer by increasing the dyadic form of some functions to triadic and higher. The co-processor resides internal to the processor and uses one of the chip's cogs, or cores. This co-processer can then be used by any user of the Parallax Propeller by adding the code to the microcontroller. The Parallax Propeller chip is a multi-core microcontroller that can run up to 80 MHz. It consists of 8 cores that share a set of common resources such as memory and I/O and is programmable with the supplied software in SPIN (a high level language) or assembly language.**

Index Terms – Math co-processer, microcontroller, IEEE 754, multi-core, assembly language.

BACKGROUND

The Parallax Propeller chip was made to perform high-speed computing for embedded systems sustaining low current utilization. It consists of eight internal processers that can complete synchronized mutual errands that are fairly basic and easy to use. There are three main objectives of this particular multi-core microcontroller. The memory map is flat which enables the chip to save time throughout function creation. The multi-core microcontroller performs asynchronous operations that allow the hardware to operate better than hardware that uses interrupts or synchronous operations. Asynchronous operations are structured to perform tasks independently, whereas hardware that uses interrupts must rely on other processes being completed before a different task can be performed. This structure keeps the other processors free and provides the hardware with maximum efficiency. The last goal is to make the hardware more applicable by giving the user the option of using SPIN or assembly language for programming code

Traditional math co-processors perform various operations such as add, subtract, multiply, divide, tangent, sine, cosine, and log functions, etc… The math co-processor is equipped with data types, registers, instructions, and executes number processing quickly. It has assemblers and compilers built internally to interpret high level languages for the user's convenience. These co-processors have traditionally been located external to the microcontroller connected by some kind of communication link. Examples include the Intel 387^{TM}DX and the Motorola M68000 devices.

A standard math co-processor performs basic dyadic math operations that include add, subtract, multiply, divide, and trigonometric functions among others. An existing assembly language code for an on-board math co-processor has been built around this long standing method and also performs only dyadic operations. If the user wishes to manipulate more than two variables multiple calls to the co-processor are required. For example, if a math co-processor existed on-board as a core and it was used to multiply four variables (A, B, C, and D) the following pseudo-code would be used.

```
A=CoP.FMul (A, B)
A=CoP.FMul (A, C)
A=CoP.FMul (A, D)
```

Where CoP is the co-processor object name and FMul is the multiply command. This operation is performed until the desired number of variables has been manipulated. This causes significantly more inefficiencies than if the co-processor could handle triadic and greater functions. It is our intent in this paper to describe the results of one such co-processor operating on a Parallax Propeller multi-core microcontroller. [1]

IEEE-754 1985

The IEEE Standard for Floating-Point Arithmetic (IEEE 754-1985) is a piece of hardware or software that represents floating-point numbers in binary format [2]. The standard was upgraded in 2008, but the 1985 version will be used in this paper. IEEE-754 uses a thirty-two and a sixty-

four bit system to represent binary numbers. The thirty-two bit system is used in this project. Figure 1 and (1) below describe how the IEEE-754 represents the decimal number in 32 bits.

$$v = (-1)^s \cdot 2^{e-bias} \cdot (1.m) \qquad (1)$$

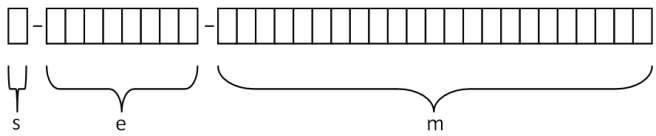

FIGURE 1
IEEE-754 1985 32 BIT REPRESENTATIONS.

The sign portion of the representation determines whether the number is positive or negative. This is accomplished by using zero for positive numbers and one for negative numbers. The exponent is represented by eight bits and is biased by 127 to allow for negative exponents. The mantissa consists of the remaining twenty three bits and represents the fractional portion of the equation. As shown above, there is an understood initial bit that is followed by the fraction bits. For example the number 26.5354 is represented as follows.

0 10000011 10101000100100001111111

S Exponent Mantissa

$$v = (-1)^0 \cdot 2^{131-127} \cdot (1.6584625) = 26.5353984833$$

This representation demonstrates one drawback to the method, which is the non-represent ability of some numbers. 26.5353 are represented as 26.5353984833. This represents a representation error of just greater than 1.5×10^{-6}. Another drawback to this method is round-off error which will be shown below.

Two floating point numbers can be added as follows with an included example.

26.5354 = 0 10000011 10101000100100001111111

1456.2673 = 0 10001001 01101100000100010001101

1. Retain the sign bit of the largest number: $s_x = 0$.
2. Shift the mantissa associated with the smallest exponent by the difference in exponents: Shift the mantissa of 26.5354 by 6 bits to the right and limit to 23 bits: m_1 = 00000010101000100100001
3. Add the mantissas: m_1 = 00000010101000100100001
   m_2 = 01101100000100010001101
   ---
   m_x = 01101110101100110101110
4. Retain the new common exponent: e_x = 10001001

The result is then 0 10000011 01101110101100110101110 or 1482.80249. This represents an error of 2.1×10^{-4}.

Two floating point numbers can be multiplied as follows with an included example. Given the following numbers:

26.5354 = 0 10000011 10101000100100001111111

1456.2673 = 0 10001001 01101100000100010001101.

1. XOR the sign bits: $s_x = s_1$ (+) s_2 = 0 (+) 0 = 0.
2. Add the unbiased exponent of each number together and limit to 8 bits. 4 + 10 = 13.
3. Re-bias the number by adding 127; 14 + 127 = 141; e_x = 10001101.
4. Multiply the mantissas, re-justify and limit to 23 bits:
   m_1=1.10101000100100001111111
   m_2=1.01101100000100010001101
   ---
   m_x=10.01011011111001010100001
   m_x=1.00101101111001010100001

The result is then 0 10001101 00101101111001010100001 or 38642.63531. This represents an error of 6.41×10^{-3}.

MODIFICATIONS

Starting with the program file mentioned earlier, a more efficient set of functions were created. To begin with, the functions that were improved were the multiplication and addition functions as they seemed logical due to their current dyadic nature. Additionally, they are more commonly used. To successfully multiply triadic variables or more, the user would have to multiply two variables then take the product of the dyadic variables, call upon the function again and multiply another variable with the product.

For test purposes, functions were created that will multiply from three through ten variables depending on the user's desire. Traditionally, this would not have been accomplished on a math co-processor because of the higher level of memory consumed. However, in a multi-core microcontroller code can be selected as needed. These functions were implemented in assembly language and are defined as follows.

FMul3(a,b,c)=a·b·c (1)
FMul4(a,b,c,d)=a·b·c·d (2)
FMul5(a,b,c,d,e)=a·b·c·d·e (3)
FMul6(a,b,c,d,e,f)=a·b·c·d·e·f (4)
FMul7(a,b,c,d,e,f,g)=a·b·c·d·e·f·g (5)
FMul8(a,b,c,d,e,f,g,h)=a·b·c·d·e·f·g·h (6)
FMul9(a,b,c,d,e,f,g,h,i)=a·b·c·d·e·f·g·h·i (7)
FMul10(a,b,c,d,e,f,g,h,i,j)=a·b·c·d·e·f·g·h·i·j (8)

The addition function was set up the same way as the multiplication function in the original program. Similar code was added to the assembly language co-processor such that triadic through decadal operations could be handled.

FAdd3(a,b,c)=a+b+c (9)
FAdd4(a,b,c,d)=a+b+c+d (10)
FAdd5(a,b,c,d,e)=a+b+c+d+e (11)
FAdd6(a,b,c,d,e,f)=a+b+c+d+e+f (12)
FAdd7(a,b,c,d,e,f,g)=a+b+c+d+e+f+g (13)
FAdd8(a,b,c,d,e,f,g,h)=a+b+c+d+e+f+g+h (14)
FAdd9(a,b,c,d,e,f,g,h,i)=a+b+c+d+e+f+g+h +i (15)
FAdd10(a,b,c,d,e,f,g,h,i,j)=a+b+c+d+e+f+g+h+i+j (16)

The functionality was implemented into the Propeller multi-core microcontroller and the execution times were recorded for multiplying and adding the following test numbers together.

A = 2.3465
B = 2.3560
C = 8.0000
D = 5.1255
E = 1.06259
F = 2.457
G = 2.113
H = 2.22
I = 9.1234
J = 2.235
K = 6.84654
L = 4.84891
M = 2.3
N = 5.654
O = 8.64
P = 0.0085
Q = 0.88
R = 1.55
S = 0.008995
T = 0.2341

The highest applicable function was used in all cases. For example when FMul10 was being tested the code was as follows.

X = AB.FAdd10(A,B,C,D,E,F,G,H,I,J)
X = AB.FAdd10(X, K, L, M, N, O, P, Q, R, S)
X = AB.FAdd(X, T)

These results shown in figures 2 and 3 indicate that the modified program is more efficient.

By performing math operations to multiple variables at once, the program has shown up to a 56% improvement with multiplication speeds and up to a 64% improvement with addition speeds. Equipping the program with palpable superiority over the original program. [3], [4]

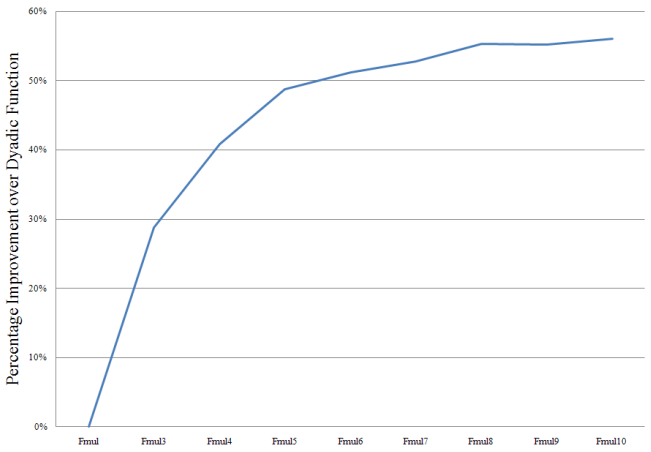

FIGURE 2
MULTIPLICATION TIMES: DYADIC THROUGH DECADAL.

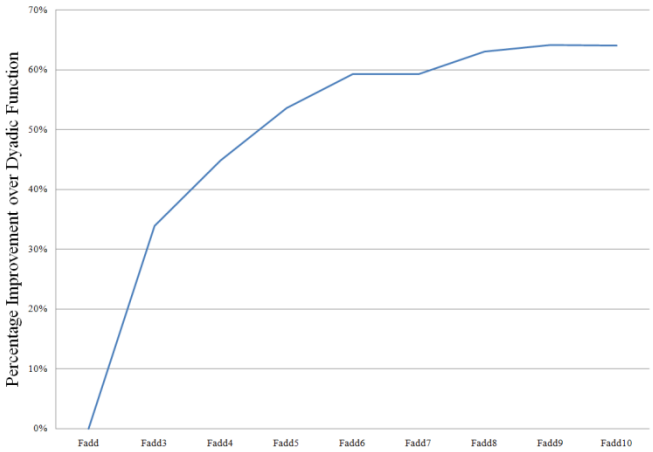

FIGURE 3
ADDITION TIMES: DYADIC THROUGH DECADAL.

APPLICATIONS

There are many areas of calculation solving that are not sufficient enough to compute the results of mathematical operations promptly using only dyadic functionality. An example of this situation is the Newton-Raphson iterative method for systems of equations.

This method is a common iterative solution and can be used to solve for the inverse kinematics of a manipulator [5]. Inverse kinematics is the solution for the angle of each axis in a manipulator when given a Cartesian point in space with respect to a known coordinate system.

The method begins with an initial estimate of the angles of each axis, $q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$ and the goal point in Cartesian space, $T = \begin{bmatrix} d_X \\ d_Y \\ d_Z \end{bmatrix}$. From there a Jacobian Matrix is defined as

$$J = \left[\frac{\partial T_i}{\partial q_j}\right] = \begin{bmatrix} \frac{\partial d_X}{\partial \theta_1} & \frac{\partial d_X}{\partial \theta_2} & \frac{\partial d_X}{\partial \theta_3} \\ \frac{\partial d_Y}{\partial \theta_1} & \frac{\partial d_Y}{\partial \theta_2} & \frac{\partial d_Y}{\partial \theta_3} \\ \frac{\partial d_Z}{\partial \theta_1} & \frac{\partial d_Z}{\partial \theta_2} & \frac{\partial d_Z}{\partial \theta_3} \end{bmatrix} \quad (17)$$

The Newton-Raphson method defines the iterated equation as follows.

$$q^{i+1} = q^i + J^{-1}\left(q^i\right)\delta T.$$

The method begins with an initial guess, q^i, and it is used to compute the inverse Jacobian and the error in Cartesian space given the guessed angles, δT.

The Newton-Raphson method was applied to a particular manipulator and is shown below in Matlab.

```
%Original Guess for joint angles.
theta1=50.0;
theta2=50.0;
theta3=50.0;

for i=1:5000 %Loop up to 5000 times.

%Compute all trigonometric equations
   c1=cosd(theta1);
   c2=cosd(theta2);
   c3=cosd(theta3);
   s1=sind(theta1);
   s2=sind(theta2);
   s3=sind(theta3);

%Compute the partial derivatives, X1 is the partial derivative of dX with
respect to theta 1.
   X1=-a3*s1*c2*c3-a3*c1*s3+d3*s1*s2-a2*s1*c2+d2*c1;
   X2=-a3*c1*s2*c3-d3*c1*c2-a2*c1*s2;
   X3=-a3*c1*c2*s3-a3*s1*c3;
   Y1=a3*c1*c2*c3-a3*s1*s3-d3*c1*s2+a2*c1*c2+d2*s1;
   Y2=-a3*s1*s2*c3-d3*s1*c2-a2*s1*s2;
   Y3=-a3*s1*c2*s3+a3*c1*c3;
   Z1=0;
   Z2=a3*c2*c3-d3*s2+a2*c2;
   Z3=-a3*s2*s3;

   %Compute the error, deltaT.
   deltaT=[dX-Y1;dY+X1;dZ-(a3*s2*c3+d3*c2+a2*s2+d1)];

   %Compute the determinant of the Jacobian Matrix, to be used in the
   %final calculation of the inverse of the Jacobian Matrix.
   DET=X1*(Z3*Y2-Z2*Y3)-Y1*(Z3*X2-Z2*X3);

   %Compute the inverse of the Jacobian Matrix.
   Jinv=(1/DET)*[(Z3*Y2-Z2*Y3) , -(Z3*X2-Z2*X3) , (Y3*X2-Y2*X3) ;
-(Z3*Y1) , (Z3*X1) , -(Y3*X1-Y1*X3) ; (Z2*Y1) , -(Z2*X1) , (Y2*X1-
Y1*X2)];

   %Compute the next best guess in the iteration.
   theta=[theta1 ; theta2 ; theta3]+Jinv*deltaT;

   %Check the tolerance to see if we should be continuing or not.
   if ((abs(deltaT(1))<tolerance) && (abs(deltaT(2))<tolerance) &&
         (abs(deltaT(3))<tolerance))
      break
   end
end
```

The code was ported into the microcontroller and the improved assembly language math co-processor was implemented to assist in solving this iterative algorithm and its performance is compared to the original co-processor below.

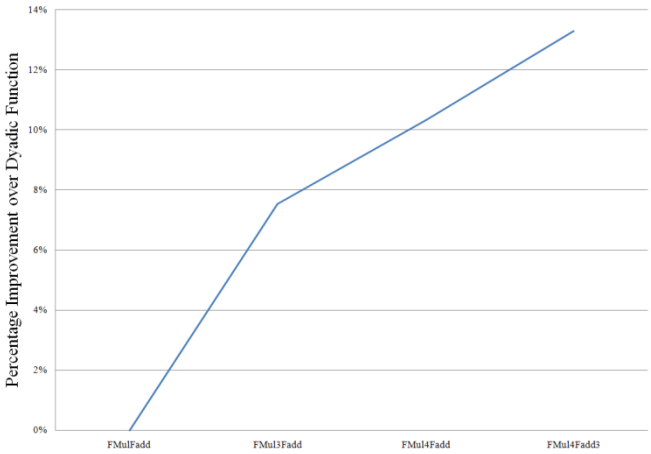

FIGURE 3
NEWTON-RAPHSON PERFORMANCE IMPROVEMENTS: SYSTEM OF 3 EQUATIONS, 412 ITERATIONS.

As compared with the Matlab algorithm the solve times are as follows when Matlab runs on a 3.0GHz PC and the microcontroller is running at 37.5 times slower at 80MHz. The minimum Matlab solve time was found to be 45.8mS when the algorithm was run 1000 times to eliminate the possibility of the operating system from interfering with the recorded solve time. In comparison, the fastest solve time on the microcontroller was 2.351 seconds. If we assume a linear relationship between clock frequency and algorithm solve time then the normalized solve time is 62.7mS.

REFERENCES

1. Martin, J. and Lindsay, S., "*Parallax Propeller Manual*", 2006.
2. Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *Computing Surveys*, Association for Computing Machinery, 1991,

3. Thompson, C., Assembly Language File, "*Float32*," IEEE 754 Compliant 32-Bit Floating Point Math Routines," Micromega Corporation, Copyright (c) 2006-2007. Parallax, Inc.

4. Gracey, C., "Float-Point Math: Single-precision IEEE-754", Parallax, Inc. Copyright (c) 2006.

5. Rankin J. and Hradek R., "The Controls and Manipulator Design of a Robotic Table Tennis Player", *ASEE NCS Conference Proceedings* 2009.

AUTHOR INFORMATION

Matthew Lang Manufacturing Technology Student, Ohio Northern University, Department of Technological Studies, Ada, OH, 45810, m-lang@onu.edu.

Adam W. Stienecker Assistant Professor and Director of Ohio Northern University's Robotics Center of Excellence, Ohio Northern University, Department of Technological Studies, Ada, OH, 45810, a-stienecker.1@onu.edu.