MATH CO-PROCESSOR EFFICIENCY AND IMPROVEMENTS ON A MULTI-CORE MICROCONTROLLER

Adam Stienecker, Ohio Northern University; Matthew Lang, Ohio Northern University

Abstract

Traditionally, a math co-processer is a hardware device that resides next to a microprocessor or microcontroller in an electronic device. The math co-processor is there to complete complex and lengthy mathematical processes because either the main processor is not capable of floating point math or because such processes require significant processing resources that are unavailable on the main processor. With the rising popularity and falling price of multi-core microcontrollers such as the Parallax Propeller, the interest in on-chip math co-processors has increased. This paper aims to document improvements made to the traditional methods of math co-processing by increasing the dyadic form of some functions to triadic up through decadic. The co-processor resides inside the processor and uses one of the chip's cogs, or cores. This assembly language co-processer can then be used by any programmer of the Parallax Propeller by adding the code to the microcontroller as an object file that runs completely independent of the user's main code. With the increasing abilities of very small, embedded devices, the computational abilities of the processor are becoming increasingly more important. Product developers are continually creating products that do more at a faster rate with lower power and smaller size. The improvements outlined in this paper that have been made to the standard methods for math co-processing will aid in doing more with less.

Background

The Propeller microcontroller by Parallax was designed to perform multi-core computing in embedded systems with low power consumption. It consists of eight internal processers with a set of shared resources including I/O, memory, and the system clock. Due to this design, the Propeller operates without interrupts, unlike many other microcontrollers. This is made up for by the additional processing power of multiple processing units. In many embedded systems an interrupt may be assigned to a specific input or set of inputs associated with an incoming signal. Instead, in the Propeller multi-core environment, the programmer is able to assign a core to keep up with the higher bandwidth signals.

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 and instructions, and executes number processing quickly. It has internal assemblers and compilers to interpret high-level languages for the user's convenience. These co-processors have traditionally been located external to the microcontroller and connected by some kind of communication link. Examples include the Intel 387^{TM}DX and the Motorola M68000 devices.

Some of the functions that a standard co-processor performs are unary, that is, they require only one argument such as the sine and cosine function. The other major type of function performed by a standard math co-processor is the dyadic math operations that include add, subtract, multiply, and divide. An existing assembly language code for an on-board math co-processor has been built around this long standing method [1]. 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 (Y = A * B * C * D) the following pseudo-code could be used.

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

CoP is the co-processor object name and FMul is the multiply command inside the object. 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 functionality. It was the intent of this study to describe the results of a customizable math co-processor capable of unary and dyadic through decadic functionality operating on a Parallax Propeller multi-core microcontroller [2].

Modifications

Starting with the program file mentioned earlier, a more efficient set of functions was created [3]. To begin with, the functions that were improved were the multiplication and addition functions as they seemed logical due to their current dyadic nature and 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 reference, the currently existing functionality for multiply and add operations was defined as follows.

$$\mathrm{FMul}(a,b)= a\cdot b \tag{1}$$
$$\mathrm{FAdd}(a,b)= a+b \tag{2}$$

For test purposes, functions were created that would multiply three to ten variables, depending on the user's desire. These functions were implemented in assembly language and are defined as follows.

$$\mathrm{FMul3}(a,b,c)=a\cdot b\cdot c \tag{3}$$
$$\mathrm{FMul4}(a,b,c,d)=a\cdot b\cdot c\cdot d \tag{4}$$
$$\mathrm{FMul5}(a,b,c,d,e)=a\cdot b\cdot c\cdot d\cdot e \tag{5}$$
$$\mathrm{FMul6}(a,b,c,d,e,f)=a\cdot b\cdot c\cdot d\cdot e\cdot f \tag{6}$$
$$\mathrm{FMul7}(a,b,c,d,e,f,g)=a\cdot b\cdot c\cdot d\cdot e\cdot f\cdot g \tag{7}$$
$$\mathrm{FMul8}(a,b,c,d,e,f,g,h)=a\cdot b\cdot c\cdot d\cdot e\cdot f\cdot g\cdot h \tag{8}$$
$$\mathrm{FMul9}(a,b,c,d,e,f,g,h,i)=a\cdot b\cdot c\cdot d\cdot e\cdot f\cdot g\cdot h\cdot i \tag{9}$$
$$\mathrm{FMul10}(a,b,c,d,e,f,g,h,i,j)=a\cdot b\cdot c\cdot d\cdot e\cdot f\cdot g\cdot h\cdot i\cdot j \tag{10}$$

Similar code was added to the assembly language co-processor for addition.

$$\mathrm{FAdd3}(a,b,c)=a+b+c \tag{11}$$
$$\mathrm{FAdd4}(a,b,c,d)=a+b+c+d \tag{12}$$
$$\mathrm{FAdd5}(a,b,c,d,e)=a+b+c+d+e \tag{13}$$
$$\mathrm{FAdd6}(a,b,c,d,e,f)=a+b+c+d+e+f \tag{14}$$
$$\mathrm{FAdd7}(a,b,c,d,e,f,g)=a+b+c+d+e+f+g \tag{15}$$
$$\mathrm{FAdd8}(a,b,c,d,e,f,g,h)=a+b+c+d+e+f+g+h \tag{16}$$
$$\mathrm{FAdd9}(a,b,c,d,e,f,g,h,i)=a+b+c+d+e+f+g+h+i \tag{17}$$
$$\mathrm{FAdd10}(a,b,c,d,e,f,g,h,i,j)=a+b+c+d+e+f+g+h+i+j \tag{18}$$

Using the currently existing structure in the assembly language program, the above functions were added. However, instead of duplicating the code for the IEEE-754 multiply [3], [4] four times for the FMul4 command, the FMul command was used three times by the FMul4 command to complete the process all within the assembly language object. While the communication overhead and register setup was reduced when only one function call to the assembly language object was used, a trade-off was made through the use of the other commands in the object in order to save memory. Theoretically, the code could be altered to allow each function a high level of self-reliance at the expense of memory. This would improve the speed even more but, along with increasing the memory size, would decrease the ease of customizability. If further improvements in speed were required the customizability could be compromised, although the improvements would be minimal in comparison to the improvements described herein.

The functionality was implemented into the Propeller multi-core microcontroller and the execution times were recorded for multiplying and adding a series of real numbers ranging from 0.0085 to 9.1234. These twenty numbers were kept constant throughout the test. The highest applicable function was used in all cases. For example, when FAdd10 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 1 and 2, indicate that the modified program is more efficient. Efficiency is defined purely by execution time. For example, in Figure 1, the operation took 808us using the FMul function only, and decreased to 575us when FMul3 was used, yielding a 28.752% improvement.

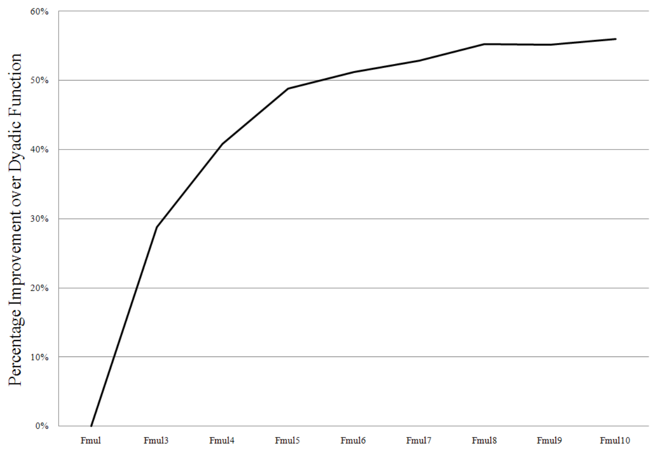

Figure 1. Multiplication Efficiency Improvements [3]

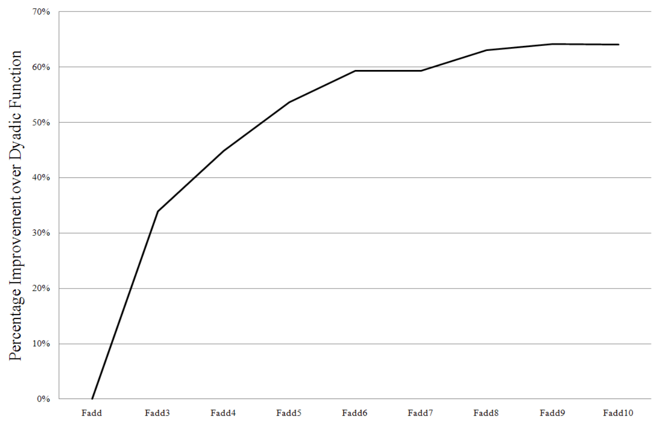

Figure 2. Addition Efficiency Improvements [3]

Applications

There are many areas in embedded systems that may require large amounts of mathematics. Two examples of this include matrix inversion and the Newton-Raphson iterative method for systems of equations.

The Newton-Raphson method is a common iterative algorithm and can be used to solve for the inverse kinematics of a manipulator [3], [5]. Inverse kinematics is the solution for the angle of each axis in a robotic manipulator, when given a Cartesian point in space with respect to a known coordinate system at the base of the manipulator. For this application the authors assumed a manipulator with three joints and D-H parameters as follows:

Table 1. D-H Parameters of an example robot manipulator [5]

Link Length	Joint Angles	Link Twist	Joint Distances
$a_1 = 0$ mm	$\theta_1 = ?$	$\alpha_1 = 90°$	$d_1 = 78$ mm
$a_2 = 292$ mm	$\theta_2 = ?$	$\alpha_2 = -90°$	$d_2 = 43$ mm
$a_3 = 288$ mm	$\theta_3 = ?$	$\alpha_3 = 90°$	$d_3 = -29$ mm

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

$$J = \left[\frac{\partial T_i}{\partial q_j}\right] = \begin{bmatrix} \frac{\partial d_X}{\partial \theta_1} & \frac{\partial d_X}{\partial \theta_2} & \frac{\partial d_X}{\partial \theta_3} \\ \frac{\partial d_Y}{\partial \theta_1} & \frac{\partial d_Y}{\partial \theta_2} & \frac{\partial d_Y}{\partial \theta_3} \\ \frac{\partial d_Z}{\partial \theta_1} & \frac{\partial d_Z}{\partial \theta_2} & \frac{\partial d_Z}{\partial \theta_3} \end{bmatrix}. \quad (19)$$

The Newton-Raphson equation is then defined as follows:

$$q^{i+1} = q^i + J^{-1}\left(q^i\right)\delta T. \quad (20)$$

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

The Newton-Raphson method was applied to the example manipulator and was programmed into the microcontroller. The improved assembly language math co-processor was customized in the same manner as described above to assist in solving this iterative algorithm, and its performance was compared to the original (dyadic only) co-processor code in Figure 3. This algorithm required 412 iterations for the specific initial guesses and given Cartesian point. Because the microcontroller code was quite complicated, the given solution was validated using a known working implementation of the algorithm in Matlab.

As expected, an actual application was not able to make use of all of the available functions that were added to the code in the previous section. The following functions were implemented in the object to incrementally improve the speed of the application.

FMul(a,b) and FAdd(a,b) (baseline implementation)
FMul3(a,b,c) and FAdd(a,b)
FMul4(a,b,c,d) and FAdd(a,b)
FMul4(a,b,c,d) and FAdd3(a,b,c)

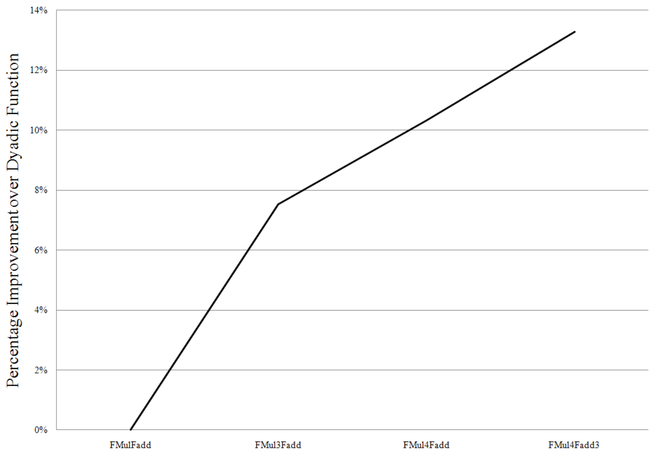

Figure 3. Newton-Raphson Performance Improvements [7]

The Newton-Raphson method for a system of equations was able to make use of the higher-order functions of Add and Multiply. However, not all higher-level mathematics done in an embedded system can make use of this. For example, a standard matrix inverse algorithm for square matrices, the Gauss-Jordan elimination method [6], would not benefit from the above improvements. However, if one other function were created in the math co-processor, the inverse matrix would see an improvement in speed. A standard implementation of the matrix inverse algorithm was implemented such that matrix A (n x n) was concatenated with the identity matrix of the same size to form [AI], then a series of row operations could create [A^{-1}I]. Once implemented, this algorithm was tested for n = 4 through 10 and compared with results of Matlab to verify accuracy. While the results were verified, no attempt to ensure numerical stability of the algorithm by pivoting was completed [7], rather the basic inverse matrix algorithm speeds were of main consideration.

As described above, the algorithm does not benefit from equations (3) through (18) as it makes use of the row operation equations, (21) and (22), among others.

$$A(i,j) = A(i,j) - \frac{A(i,y)}{A(y,y)} A(y,j) \qquad (21)$$

$$A(i,j) = A(i,j) - A(i,y)A(y,j) \qquad (22)$$

Because of the form of these equations, (x = a + (-b*c)), a new function was created in the math co-processer to support this form in order to decrease the solve time. This implementation was the same as described above, a simple function that made use of the FMul and FAdd functions within the object. The algorithm was timed with a standard implementation of a math co-processor and compared to the time required when the new function was added. The results are displayed in Figure 4. Once again, improvement was defined only by the execution speed. For example, the inverse of a 10 x 10 matrix required 436.3ms before implementation of the added function in the math co-processor and 385.8ms after the implementation. This represents an 11.57% improvement.

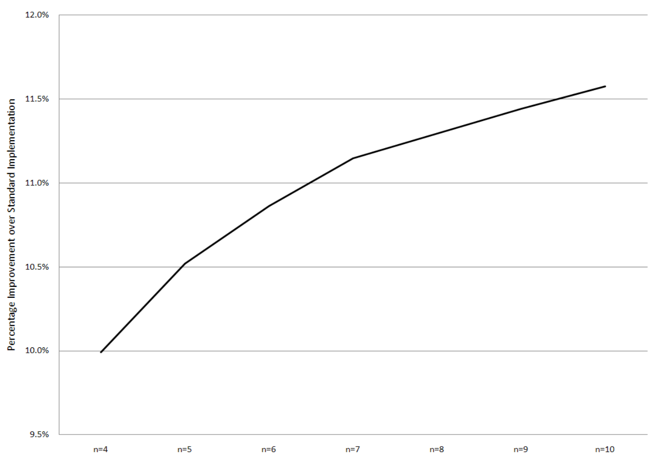

Figure 4. Matrix Inverse Algorithm Performance

Conclusion

In this paper, the authors presented and demonstrated an advancement in math co-processing for embedded systems through the use of two algorithms requiring the use of floating-point math. Because many microcontrollers are not typically well-suited for floating point math, a math co-processor has typically been used to supplement the main processor. What should not be concluded is that a new standard for math co-processors should be created that includes higher functionality. Rather, the ability to adapt the standard implementation of a math co-processor into a customizable co-processor is highly desirable in an electronics world where speed, cost, and size are everything. Increasingly, product developers are required to make them smaller, cheaper, and faster with little to no design time and costs. With a multi-core microcontroller, the ability to customize the functionality of a co-processer with no added design cost is invaluable. Not only are there no added design costs, but code changes in the co-processor don't affect the product cost, the board layout, or the product size. With a potential increase in algorithm execution speeds of over 50% and less cost than a traditional math co-processor, this is an obvious benefit.

References

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

[2] Martin, J. and Lindsay, S., "*Parallax Propeller Manual*", 2006.

[3] M. Lang and A. Stienecker, "Assembly Language Math Co-Processor Efficiency Study and Improvements on a Multi-core Microcontroller", *ASEE NCS Conference Proceedings* 2010.

[4] Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *Computing Surveys*, Association for Computing Machinery, 1991.

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

[6] Leon, S.J., Linear Algebra with Applications. 5th ed. New Jersey: Prentice Hall, 1998.

[7] Trefethen, L.N., Numerical Linear Algebra. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

Biographies

ADAM W. STIENECKER teaches electronics and applied control systems courses at Ohio Northern University in the Department of Technological Studies. He holds undergraduate and doctorate degrees in Electrical Engineering from the University of Toledo in Ohio. His areas of research include embedded systems and advanced control of mobile robots. He can be reached at a-stienecker.1@onu.edu.

MATTHEW LANG recently completed his B.S. in Manufacturing Technology in the Department of Technological Studies at Ohio Northern University. He is currently seeking employment. He can be reached at mlang457@gmail.com.