Using CORDIC methods for computation in micro-controllers

Introduction

Many times in designing software for a micro-controller system, it is necessary to make calculations that involve elementary functions such as Sin(x), Cos(x) or $Log_{10}(x)$. For example, many temperature sensors are logarithmic in nature. That is the sensor output voltage may increase by x volts each time the temperature doubles. In this case converting the sensor voltage to a linear temperature scale requires the calculation of 2^x.

Calculation of an elementary function is often times done by using a look-up table. Look-up tables are by far the fastest way to make the computation, however the precision of the result is directly related to size of the look-up table. High precision look-up tables require a large amount of non-volatile memory to store the table. If the table size is reduced to save memory, precision will also be reduced.

Power series may also be used to calculate these same functions with out using look-up tables, however these calculations have the disadvantage of being slow to converge to a desired precision. In effect, the look-up table size is being traded at the expense of computation time.

CORDIC methods of computation represent a compromise between the two methods described above. The CORDIC technique uses a one-bit-at-a-time approach to make computations to an arbitrary precision. In the process, relatively small look-up tables are used for constants necessary for the algorithm. Typically these tables require only one to two entries per bit of precision. CORDIC algorithms also use only right shifts and additions, minimizing the computation time.

Fundamentals of CORDIC algorithms

All CORDIC algorithms are based on the fact that any number may be represented by an appropriate alternating series. For example an approximate value for e may be represented as follows:

$$e = 3 - 0.3 + 0.02 - 0.002 + 0.0003 = 2.7183$$

Notice that in this case each digit gives an additional power of ten resolution to the approximation of the value for e. Also if the series is truncated to a certain number of terms, the resulting value will be the same as the value obtained by rounding the true value of e to that number of digits. In general the series obtained for a value by this method does not always alternate regularly. The series for π is an example:

$$\pi = 3 + 0.1 + 0.04 + 0.002 - 0.0004 - 0.00001 = 3.14159$$

It may also be shown that the series for e is also irregular if the expansion is continued for a few additional terms.

The CORDIC technique uses a similar method of computation. A value to be computed, such as SIN(x) or $Log_{10}(x)$, is considered to be a truncated series in the following format:

$$z = Log_{10}(x) = = \sum_{i=1}^{B} a_i * 2^{-i}$$

In this case the values for a_i are either 0 or 1 and represent bits in the binary representation of z. The value for z is determined one bit at a time by looking at the previously calculated value for z, which is correct to i-1 bits. If this estimate of z is too low, we correct the current estimate by adding a correction factor, obtained from a look-up table, to the current value of z. If the current estimate of z is too high, we subtract a correction factor, also from the look-up table. Depending on whether we add or subtract from the current value of z, the i^{th} bit will be set to the correct value of 0 or 1. The less significant bits from i+1 to B may change during this process because the estimate for z is only accurate to i bits.

Because of the trigonometric relationship between the SIN(x) and COS(x) functions, it is often possible to calculate both of these values simultaneously. If the COS(x) is considered as a projection onto the x axis and SIN(x) as a projection onto the y axis, it is seen that the iteration process amounts to the rotation of an initial vector. It is from this vector rotation that the CORDIC algorithm derives its name: **CO**ordinate **R**otation **DI**gital **C**omputer.

Algorithms for Multiplication and Division

A CORDIC algorithm for Multiplication may be derived by using a series representation for x as follows:

$$z = x * y$$

$$= y * \sum_{i=1}^{B} a_i * 2^{-i}$$

$$= \sum_{i=1}^{B} y * a_i * 2^{-i}$$

$$= \sum_{i=1}^{B} a_i * (y * 2^{-i})$$

From this it is seen that z is composed of shifted versions of y. The unknown coefficients, a_i, may be found by driving x to zero one bit at a time. If the i^{th} bit of x is non-zero, y_i is right shifted by i bits and added to the current value of z. The i^{th} bit is then removed from x by subtracting 2^{-i} from x. If x is negative, the i^{th} bit in the twos complement format would be removed by adding 2^{-i}. In either case, when x has been driven to zero all bits have been examined and z contains the signed product of x and y correct to B bits.

This algorithm is similar to the standard shift and add multiplication algorithm except for two important features. First, arithmetic right shifts are used instead of left shifts, allowing signed numbers to be used. Secondly, computing the product to B bits with the CORDIC algorithm is equivalent to rounding the result of the standard algorithm to the most significant B bits. The final algorithm is as follows:

```
multiply(x,y){
   for (i=1; i=<R; i++){
      if (x > 0)
         x = x - 2(^-i)
```

```
            z = z + y*2^(-i)
         else
            x = x + 2(^-i)
            z = z - y*2^(-i)
      }
      return(z)
   }
```

This calculation assumes that both x and y are fractional ranging from -1 to 1. The algorithm is valid for other ranges as long as the decimal point is allowed to float. With a few extensions, this algorithm would work well with floating point data.

A CORDIC division algorithm is based on re-writing the equation z = x/y into the form x - y*z = 0. If z is expanded into its series representation, The second version of the equation takes the following form:

$$x - y * \sum_{i=1}^{B} a_i * 2^{-i} = 0$$

Which, after some manipulation, yields:

$$x - \sum_{i=1}^{B} a_i * (y * 2^{-i}) = 0$$

This final form of the equation shows that the quotient z may be estimated one bit at a time by driving x to zero using right shifted versions of y. If the current residual is positive, the i^{th} bit in z is set. Likewise if the residual is negative the i^{th} bit in z is cleared.

```
divide(x,y){
   for (i=1; i=<R; i++){
      if (x > 0)
         x = x - y*2(^-i);
         z = z + 2^(-i);
      else
         x = x + y*2(^-i);
```

```
            z = z - 2^(-i);
        }
        return(z)
    }
```

The convergence of this division algorithm is a bit trickier than the multiplication algorithm. While x may be either positive or negative, the value for y is assumed to be positive. As a result, the division algorithm is only valid in two quadrants. Also, if the initial value for y is less than the initial value for x it will be impossible to drive the residual to zero. This means that initial y value must always be greater than x, resulting in domain of $0 < z < 1$. The algorithm may be modified as follows for four quadrant division with $-1 < z < 1$:

```
divide_4q(x,y){
    for (i=1; i=<R; i++){
        if (x > 0)
          if (y > 0)
             x = x - y*2(^-i);
             z = z + 2^(-i);
          else
             x = x + y*2(^-i);
             z = z - 2^(-i);
        else
           if (y > 0)
              x = x + y*2(^-i);
              z = z - 2^(-i);
           else
              x = x - y*2(^-i);
              z = z + 2^(-i);
    }
    return(z)
}
```

As with all division algorithms, the case where y is zero should be trapped as an exception. Once again, a few extensions would allow this algorithm to work well with floating point data.

Algorithms for $Log_{10}(x)$ and 10^x

To calculate the base 10 logarithm of a value x, it is convient to use the following identity:

$$Log_{10}(x*\prod_{i=1}^{B} b_I) = Log_{10}(x) + \sum_{i=1}^{B} Log_{10}(b_i)$$

If the b_i are chosen such that $x*b_1*b_2*b_3...*b_B = 1$, we see that the left hand side reduces to $Log_{10}(1)$ which is 0. With these choices for b_i, we are left with the following equation for $Log_{10}(x)$:

$$Log_{10}(x) = -\sum_{i=1}^{B} Log_{10}(b_i)$$

Since quantities for $Log_{10}(b_i)$ may be stored in a look-up table, the base 10 logarithm of x may be calculated by summing selected entries from the table.

The trick now is to choose the correct b_i such that we drive the product of x and all of the b_i to 1. This may be accomplished by examining the current product. If the current product is less than 1, we choose co-efficient b_i such that b_i is greater than 1. On the other hand, if the current product is greater than 1 the coefficient should be chosen such that its value is less than one. An additional constraint is that the b_i should be chosen such that multiplication by any of the b_i is accomplished by a shift and add operation. Two coefficients which have the desired properties are:

$$b_i = 1+2^{-i} \quad \text{if} \quad x*b_1*b_2...b_{i-1} < 1$$

and

$$b_i = 1-2^{-i} \quad \text{if} \quad x*b_1*b_2...b_{i-1} > 1$$

In choosing these values for the b_i, it is seen that the limit as i approaches infinity of the product of x and the b_i's will be 1 as long as x is in the range:

$$(\prod_{i=1}^{B}(1+2^{-i}))^{-1} < x < (\prod_{i=1}^{B}(1-2^{-i}))^{-1}$$

This represents the range of convergence for this algorithm which may be calculated as approximately:

$$0.4194 < x < 3.4627$$

If it is wished to calculate logarithms outside of this range, the input must be either pre-scaled or the range of the i values must be changed. The final algorithm becomes:

```
log10(x){
   z = 0;
   for ( i=1;i=<B;i++ ){
      if (x > 1)
         x = x - x*2^(-i);
         z = z - log10(1-2^(-i));
       else
         x = x + x*2^(-i);
         z = z - log10(1+2^(-i));
   }
   return(z)
}
```

To calculate the inverse of this algorithm, or 10^x, it is only necessary to modify the existing algorithm such that x is driven to zero while z is multiplied by the successive coefficients, b_i. This follows from the fact that if $z = 10^x$ then:

$$z = b_i * 10^{(x - Log_{10}(b_i))}$$

As the exponent is driven to zero, z is seen to approach the product of all the successive coefficients, $\prod_{i=1}^{B} b_i$. The final algorithm becomes:

```
10_to_power(x){
   z = 1;
   for ( i=1;i=<B; i++ ){
      if (x > 0)
         x = x - log10(1+2^(-i));
         z = z + z*2^(-i);
      else
         x = x - log10(1-2^(-i));
         z = z - z*2^(-i);
   }
   return(z)
}
```

The range of convergence for this algorithm is determined by the range for which x can be driven to zero. By inspection of the algorithm this is determined to be:

$$\sum_{i=1}^{B} Log_{10}(1-2^{-i}) < x < \sum_{i=1}^{B} Log_{10}(1+2^{-i})$$

or x is limited to the range –0.5393< x < 0.3772. As in the previous algorithm, the range my be extended by scaling the initial value of z by $(1+2^{i})$ or $(1-2^{i})$.

The Circular Functions SIN(x) and COS(x)

It is well known that the rotation matrix

$$R(a) = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix}$$

will rotate a vector, $\begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$, counter-clockwise by *a* radians in two dimensional space. If this rotation matrix is applied to the initial vector $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ the result

will be a vector with co-ordinates of $\begin{bmatrix}\cos a \\ \sin a\end{bmatrix}$. It is easily seen that the CORDIC method could be applied to calculate the functions Sin(x) and Cos(x) by applying successive rotations to the initial vector $\begin{bmatrix}1 \\ 0\end{bmatrix}$ and gradually driving the angle a to zero.

A problem arises when an attempt is made to set up the rotation matrix such that all rotations are accomplished by right shifts. Notice that if a_i is chosen such that $\cos(a_i) = 2^{-i}$, the $\sin(a_i)$ is not necessarily a power of 2. It is not possible to choose the successive angle rotations, a_i, such that both the $\cos(a_i)$ and $\sin(a_i)$ amount to right shifts.

In working around this problem, it is possible to modify the rotation matrix by bringing a cos(a) term out of the matrix. Then:

$$R(a_i) = \cos(a_i) * \begin{bmatrix} 1 & -\tan(a_i) \\ \tan(a_i) & 1 \end{bmatrix}$$

Now the rotation angles a_i may be chosen such that $\tan(a_i) = 2^{-i}$ or rather $a_i = \tan^{-1}(2^{-i})$. The result is the final incremental rotation matrix:

$$R(a_i) = \cos(a_i) * \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix}$$

Where:

$$a_i = \tan^{-1}(2^{-i})$$

With these choices for the a_i, rotation is accomplished using only right shifts. If the $\cos(a_i)$ term is neglected in order to avoid the multiplication operations, the length of the initial vector is increased each time it is rotated by using right shifts only. This increase may be compensated for by decreasing the length of the vector prior to rotation. Since the algorithm will use B successive rotations, all rotations may be compensated for initially

using one collective length correction factor, C. The value of C is found by grouping all of the a_i terms together as follows:

$$C = (\prod_{i=0}^{B} \cos(\tan^{-1}(2^{-i}))^{-1}$$

For B = 16 bits, C may be calculated as approximately 0.6072. The final algorithms follow. Notice that x and y represent vector coordinates, while z is now the angle register.

```
sin(z){
   x = 1.6468;
   y = 0;
   for (i=0; i=<R; i++){
      if (z > 0)
         x = x - y*2(^-i)
         y = y + x*2(^-i)
         z = z - arctan(2^(-i))
      else
         x = x + y*2(^-i)
         y = y - x*2^(-i)
         z = z + arctan(2^(-i))
   }
   return(y)
}


cos(z){
   x = 1.6468;
   y = 0;
   for (i=0; i=<R; i++){
      if (z > 0)
         x = x - y*2(^-i)
         y = y + x*2(^-i)
         z = z - arctan(2^(-i))
      else
         x = x + y*2(^-i)
         y = y - x*2^(-i)
```

```
            z = z + arctan(2^(-i))
        }
        return(x)
    }
```

It may be determined that the previous two algorithms will converge as long as:

$$-\sum_{i=0}^{B}\tan^{-1}(2^{-i}) < z < \sum_{i=0}^{B}\tan^{-1}(2^{-i})$$

or

$$-1.7433 < z < 1.7433$$

Since the region of convergence includes both the first and third quadrants, the algorithms will converge for any z such that $-\pi/2 < z < \pi/2$.

Mapping the CORDIC algorithms to Micro-Controllers.

The previously discussed algorithms show that CORDIC based computation methods require minimal hardware features to implement. These are:

1) Three registers of length B bits
2) One, two or three Adders/Subtractors
3) Several small ROM based look-up tables
4) One, two or three shift registers

When implementing CORDIC algorithms on micro-controllers, item four will have the greatest effect on the overall throughput of the system. Multiplication by 2^{-i} requires that the shift register be capable of performing a right shift by i bits. Most microcontrollers are only capable of right shifting by 1 bit at a time. Shifting by i bits requires a software loop to repeat this task i times, greatly increasing the computation time. The 8051, 6805, and 68HC11 are typical examples of micro controllers which will require software loops to implement the shifter.

Other micro-controllers such as the 68HC332, as well as most Digital Signal Processors, will have a feature known as a barrel shifter. This type of shifter will right shift by i bits in one operation. Typically the shift is also accomplished in 1 clock cycle.

Another possibility for implementing a barrel shifter is to use a multiply instruction that has been optimized for speed. An example of this is the 68HC12, which has a 16 by 16 bit signed multiply, EMULS, that produces a 32 bit result in 3 clock cycles. A right shift by i bits could be accomplished by multiplying by 2^{16-i} and discarding the lower 16 bits of the result. One disadvantage of this scheme is that the data is restricted to 16 bits. Other word lengths would require additional cycles.

Once the processor and shift register style is chosen, the next choice to be made involves the data format. Since standard C does not provide a fixed-point data type, the designer has a lot of freedom in choosing the format of the data. It is a good idea, however, to choose a format that fits into 16 or 32 bit words. Even though most CORDIC routines are written in assembly language for speed, 16 or 32 bit words allow data to be passed as either 'int' or 'long int' data types within higher level C subroutines. The format used in the following examples uses a 16 bit format with 4 bits to the left of the decimal point and 12 fractional bits to the right, which is often referred to as 4.12 format. This allows constants such as π, e, and $\sqrt{2}$ to be easily represented without a moving decimal point. The 12 bits of fractional data amount to approximately 3.5 digits of decimal accuracy. The range of this format is calculated as $-8 < x < 7.9997$.

The constants used are found by multiplying by 2^{12} (4096), rounding, and converting to hexadecimal. Take the constant e for example:

$$4096*e = 11134.08 \approx 11134 = 0x267e$$

All of the data tables necessary for CORDIC computing may be built up this way using a calculator.

Finally with the data format and constant tables established, coding of the algorithms proceeds in a straightforward manner. The following examples demonstrate CORDIC algorithms implemented on the 8051, 68HCll and 68332 micro-

controllers. These code fragments were assembled with the INTEL MCS-51 Macro-Assembler and Motorola Freeware Assemblers and tested on hardware development systems.

Conclusion

CORDIC algorithms have been around for some time. Volder's original paper describing the CORDIC technique for calculating trigonometric functions appeared in the 1959 IRE transactions. However, the reasons for using CORDIC algorithms have not changed. The algorithms are efficient in terms of both computation time and hardware resources. In most micro-controller systems, especially those performing control functions, these resources are normally already at a premium. Using CORDIC algorithms may allow a single chip solution where algorithms using the look-up table method may require a large ROM size or where power series calculations require a separate co-processor because of the computation time required.

The algorithms presented have been selected to represent a small core of functions commonly required in micro-controller systems which could be discussed in detail. For each algorithm in this core, three areas have been covered: theory of operation, determining the range of convergence for the algorithm and finally implementation of the algorithm on a typical micro-controller. Using these selected algorithms as a starting point, it is possible to develop libraries containing many similar elementary functions. Among those possible with only minor modifications to the algorithms presented are: $ln x$, e^x, $\tan^{-1}x$, $\sqrt{x^2+y^2}$, and $e^{j\Theta}$. Among the references, Jarvis gives an excellent table of the functions possible using CORDIC routines.

Listing 1 – 10 to the power x algorithm implemented on the 8051

```
;------------------------------------------------------------------------;
;                                                                        ;
;   Power10x.a51                                                         ;
;                                                                        ;
;   Calculation of 10 to the power of x for the 8051 using CORDIC methods.   ;
;   on entry x1 contains the high byte of the 16 bit input and x0 contains   ;
;   the low byte. On exit z1:z0 contains z = 10^x. All data is in 4.12       ;
;   format.                                                              ;
```

```
;                                                                              ;
;   Author: Mike Pashea 3-13-2000                                              ;
;                                                                              ;
;   Comment: This routine requires approximately 1.2mS using a 12Mhz           ;
;            crystal (1161 clock cycles). It will converge for                 ;
;            -0.5393 < x < 0.3772.                                             ;
;                                                                              ;
;------------------------------------------------------------------------------;
x1            data      10h               ; x data register
x0            data      11h

z1            data      12h               ; z data register
z0            data      13h

zs1           data      14h               ; z shift register
zs0           data      15h

power10x:     mov       z1,#10h           ; [2] z = 1.0
              mov       z0,#00            ; [2]
              mov       r0,#1             ; [1]
              mov       dptr,#powr10tab   ; [2]
power10x1:    mov       zs1,z1            ; [2] Put z in the shift register.
              mov       zs0,z0            ; [2]
              mov       a,r0              ; [1] Initialize the loop counter
              mov       r1,a              ; [1]
power10x2:    mov       a,zs1             ; [1] High byte in the accumulator.
              mov       c,acc.7           ; [1] Move sign bit into carry and
              rrc       a                 ; [1] arithmetically shift right.
              mov       zs1,a             ; [1] Update the high byte.
              mov       a,zs0             ; [1] Low byte in the accumulator.
              rrc       a                 ; [1] Now shift the low byte right
              mov       zs0,a             ; [1] and update.
              djnz      r1,power10x2      ; [2] Loop for the correct number
              mov       a,x1              ; [1] is x > 0 ?
              jb        acc.7,power10x3   ; [2]
              movx      a,@dptr           ; [2] yes, x = x - log(1+2^(-i))
              inc       dptr              ; [1]
              add       a,x0              ; [1]
              mov       x0,a              ; [1]
              movx      a,@dptr           ; [2]
              inc       dptr              ; [1]
              addc      a,x1              ; [1]
              mov       x1,a              ; [1]
              mov       a,z0              ; [1] z = z * (1 + 2^(-i))
              add       a,zs0             ; [1]
              mov       z0,a              ; [1]
              mov       a,z1              ; [1]
              addc      a,zs1             ; [1]
              mov       z1,a              ; [1]
              inc       dptr              ; [1]
              inc       dptr              ; [1]
              sjmp      power10x4         ; [2]
power10x3:    inc       dptr              ; [1] no, x = x - log(1 - 2^(-i))
              inc       dptr              ; [1]
              movx      a,@dptr           ; [2]
              inc       dptr              ; [1]
              add       a,x0              ; [1]
```

```
              mov       x0,a              ; [1]
              movx      a,@dptr           ; [2]
              inc       dptr              ; [1]
              addc      a,x1              ; [1]
              mov       x1,a              ; [1]
              clr       c                 ; [1]
              mov       a,z0              ; [1] z = z * (1 - 2^(-i))
              subb      a,zs0             ; [1]
              mov       z0,a              ; [1]
              mov       a,z1              ; [1]
              subb      a,zs1             ; [1]
              mov       z1,a              ; [1]
power10x4:    inc       r0                ; [1] increment loop counter
              cjne      r0,#13,power10x1  ; [2] have we finished 12 bits?
              ret                         ; [2] yes, return z

;-------------------------------------------------------------------------;
;                                                                         ;
; The Rom table for -log10(1+2^(-i)) and -log(1+2^(-i)). The values are   ;
; interlaced and stored with the lower byte first. This format speeds up  ;
; the algorithm for the 8051 processor.                                   ;
;                                                                         ;
;-------------------------------------------------------------------------;
powr10tab:    db        02fh, 0fdh, 0d1h, 004h
              db        073h, 0feh, 000h, 002h
              db        02eh, 0ffh, 0eeh, 000h
              db        094h, 0ffh, 073h, 000h
              db        0c9h, 0ffh, 038h, 000h
              db        0e4h, 0ffh, 01ch, 000h
              db        0f2h, 0ffh, 00eh, 000h
              db        0f9h, 0ffh, 007h, 000h
              db        0fch, 0ffh, 004h, 000h
              db        0feh, 0ffh, 002h, 000h
              db        0ffh, 0ffh, 001h, 000h
              db        000h, 000h, 000h, 000h

              end
```

Listing 2 – Base 10 Logarithm Implemented on the 68HC11

```
*************************************************************************
*                                                                       *
* LOG10.ASM                                                             *
*                                                                       *
* Calculation of log10(x) for the 68HC11 using CORDIC methods. On entry *
* x is on the top of the stack. On exit z = log10(x) is at the top of the *
* stack. All data is 16 bits long using 4.12 format.                    *
*                                                                       *
* The stack frame is used as follows:                                   *
*                                                                       *
*             0,x  ==>  j - shift register counter                      *
*             1,x  ==>  i - outer loop counter                          *
```

```
*             2,x  ==>  z - output                                              *
*             4,x  ==>  xs - x shift register                                   *
*             6,x  ==>  return address                                          *
*             8,x  ==>  x - input                                               *
*                                                                               *
* Comment: This routine is meant to reflect the structure of the algorithm     *
*          without being optimized for speed. As written the algorithm          *
*          requires a maximum of 2836 clock cycles or approximately 1.4mS.      *
*          The execution time could be greatly improved by using internal       *
*          memory to hold variables.                                            *
*                                                                               *
* Author: Mike Pashea 3-11-2000                                                 *
*                                                                               *
*********************************************************************************
log10         ldy       #log10_rom     ; y points to the ROM table
              ldx       #0             ;
              pshx                     ; local space for xs
              pshx                     ; local space for z
              pshx                     ; local space for i and j
              tsx                      ; x points to the top of the stack
              ldaa      #1             ; initialize the loop counter
              staa      1,x            ;
log10_1       ldd       8,x            ; load the shift register with x
              std       4,x            ;
              ldaa      1,x            ;
              staa      0,x            ; shift counter equal to loop counter
log10_2       asr       4,x            ; perform one arithmetic shift right
              ror       5,x            ;
              dec       0,x            ;
              bne       log10_2        ; repeat until all shifts are complete
log10_3       ldd       8,x            ; is x greater than 1 ?
              subd      #4096          ;
              ble       log10_4        ; no, x should be increased
              ldd       8,x            ; yes, x should be decreased
              subd      4,x            ; x = x - x*2^(-i)
              std       8,x            ;
              ldd       2,x            ; z = z + -log10(1-2^(-i))
              addd      0,y            ;
              std       2,x            ;
              bra       log10_5        ;
log10_4       ldd       8,x            ; x = x + x*2^(-i)
              addd      4,x            ;
              std       8,x            ;
              ldd       2,x            ; z = z - log(1+2^(-i))
              subd      2,y            ;
              std       2,x            ;
log10_5       iny                      ; increment the ROM pointer so that it
              iny                      ; points to the next set of entries in
              iny                      ; the ROM table.
              iny                      ;
              inc       1,x            ;
              lda       #12            ; increment loop counter
              cmpa      1,x            ; have we calculated each bit?
              bge       log10_1        ; no, loop until we are done
              ldd       2,x            ; yes, replace x with z
              std       8,x            ;
              pulx                     ; remove local variables from stack
```

```
              pulx                    ;
              pulx                    ;
              rts                     ; return z = log10(x)

*************************************************************************
*                                                                       *
*  The ROM table for log10(1-2^(-i)) and log(1+2^(-i)). The values are  *
*  interlaced and only the magnitude is stored. The software either will*
*  add the positive values and subtract the negative ones.              *
*                                                                       *
*************************************************************************
log10_rom     fdb      $04d1, $02d1
              fdb      $0200, $018d
              fdb      $00ee, $00d2
              fdb      $0073, $006c
              fdb      $0038, $0037
              fdb      $001c, $001c
              fdb      $000e, $000e
              fdb      $0007, $0007
              fdb      $0004, $0004
              fdb      $0002, $0002
              fdb      $0001, $0001
              fdb      $0000, $0000
```

Listing 3 – Sin(z) and Cos(z) algorithms implemented on a 68000 or 68332

```
*************************************************************************
*                                                                       *
*  SINCOS.S - Computation of SIN(z) and COS(z) using CORDIC methods for the  *
*             68000 or 68HC332 processors. On entry the angle z, in radians  *
*             is on the top of the stack. On exit, the COS(x) is at the top  *
*             of the stack followed by SIN(x). All data in this example      *
*             uses 32 bit words in 8.24 format.                              *
*                                                                       *
*  Author: Mike Pashea 3-14-200                                         *
*                                                                       *
*************************************************************************
C             equ       10187768

sincos        move.l    #atantab,a0    ; A0 points to the ROM table
              move.l    4(sp),d4       ; D4 contains the angle, z
              move.l    #C,d0          ; D0 contains x
              move.l    #0,d2          ; D2 contains y
              move.b    #0,d5          ; D5 contains the loop index
sincos1       move.l    d0,d1          ; D1 is the x shift register
              move.l    d2,d3          ; D3 is the y shift register
              asr.l     d5,d1          ; xshift = x >>i
              asr.l     d5,d3          ; yshift = y >>i
              cmpi.l    #0,d4          ; if (z > 0)
              ble       sincos2        ;
              sub.l     d3,d0          ;   x = x - yshift
              add.l     d1,d2          ;   y = y + xshift
              sub.l     (a0),d4        ;   z = z - ai
              bra       sincos3        ; else
```

```
sincos2       add.l     d3,d0          ;   x = x + yshift
              sub.l     d1,d2          ;   y = y - xshift
              add.l     (a0),d4        ;   z = z + ai
sincos3       addq.l    #4,a0          ; increment ROM pointer
              addq.b    #1,d5          ; increment index
              cmpi.b    #24,d5         ; is i <= 24 ?
              ble       sincos1        ; if not loop until finished
              move.l    (sp)+,d1       ; save the return address
              move.l    d2,(sp)        ; put sin(z) on the stack
              move.l    d0,-(sp)       ; put cos(z) on the stack
              move.l    d1,-(sp)       ; restore the return address
              rts                      ; and return

*****************************************************************************
*                                                                           *
* The ROM table for arctan(2^(-i)). All constants are in 8.24 format.       *
*                                                                           *
*****************************************************************************
atantab       dc.l      13176795
              dc.l       7778716
              dc.l       4110059
              dc.l       2086331
              dc.l       1047214
              dc.l        524117
              dc.l        262123
              dc.l        131069
              dc.l         65536
              dc.l         32768
              dc.l         16384
              dc.l          8192
              dc.l          4096
              dc.l          2048
              dc.l          1024
              dc.l           512
              dc.l           256
              dc.l           128
              dc.l            64
              dc.l            32
              dc.l            16
              dc.l             8
              dc.l             4
              dc.l             2
              dc.l             1
```

References

1. Volder, Jack E., “The CORDIC Trignometric Computing Technique”, *IRE Transactions Electronic Computers*, vol. EC-8, pp. 330-334, September 1959.
2. Specker W. H., “A Class of Algorithms for Ln x, Exp x, Sin x, Cos x, $Tan^{-1}x$ and $Cot^{-1}x$”, *IEEE Transactions Electronic Computers*, vol. EC-14, pp. 85-86, 1965.
3. Walther, J. S., “A Unified Algorithm For Elementary Functions”, *1971 Proceedings of the Joint Spring Computer Conference*, pp. 379-385, 1971.
4. Jarvis, Pitts, “Implementing CORDIC Algorithms”, Dr. Dobb’s Journal, #169`, pp. 152-156, October 1990.