2 Trigonometric, Mathematical & Transcendental Functions

This chapter contains listings and descriptions of several useful trigonometric, mathematical and transcendental functions. The functions are

Trigonometric
- sine/cosine approximation
- tangent approximation
- arctangent approximation

Mathematical
- square root
- square root with single precision
- inverse square root
- inverse square root with single precision
- division

Transcendental
- logarithm
- exponential
- power

2.1 SINE/COSINE APPROXIMATION

The sine and cosine functions are fundamental operations commonly used in digital signal processing algorithms , such as simple tone generation and calculation of sine tables for FFTs. This section describes how to calculate the sine and cosine functions.

This ADSP-210xx implementation of *sin(x)* is based on a min-max polynomial approximation algorithm in the [CODY]. Computation of the function *sin(x)* is reduced to the evaluation of a sine approximation over a small interval that is symmetrical about the axis.

2 Trigonometric, Mathematical & Transcendental Functions

Let

$$|x| = N\pi + f$$

where

$$|f| \leq \pi/2.$$

Then

$$sin(x) = sign(x) * sin(f) * (-1)^N$$

Once the sign of the input, x, is determined, the value of N can be determined. The next step is to calculate f. In order to maintain the maximum precision, f is calculated as follows

$$f = (|x| - xNC_1) - xNC_2$$

The constants C_1 and C_2 are determined such that C_1 is approximately equal to *pi* (π). C_2 is determined such that $C_1 + C_2$ represents pi to three or four decimal places beyond the precision of the ADSP-210xx.

For devices that represent floating-point numbers in 32 bits, Cody and Waite suggest a seven term min-max polynomial of the form $R(g) = g \cdot P(g)$. When expanded, the sine approximation for f is represented as

$$sin(f) = ((((((r_7 \cdot f + r_6) * f + r_5) * f + r_4) * f + r_3) * f + r_2) * f + r_1) \cdot f$$

With *sin(f)* calculated, *sin(x)* can be constructed. The cosine function is calculated similarly, using the trigonometric identity

$$cos(x) = sin(x + \pi/2)$$

2.1.1 Implementation

The two listings illustrate the sine approximation and the calling of the sine approximation. The first listing, `sin.asm`, is an implementation of the algorithm for calculation of sines and cosines. The second listing, `sinetest.asm`, is an example of a program that calls the sine approximation.

Trigonometric, Mathematical & Transcendental Functions 2

Implementation of the sine algorithm on ADSP-21000 family processors is straightforward. In the first listing below, `sin.asm`, two segments are defined. The first segment, defined with the `.SEGMENT` directive, contains the assembly code for the sine/cosine approximation. The second segment is a data segment that contains the constants necessary to perform this approximation.

The code is structured as a called subroutine, where the parameter *x* is passed into this routine using register F0. When the subroutine is finished executing, the value *sin(x)* or *cos(x)* is returned in the same register, F0. The variables, `i_reg` and `l_reg`, are specified as an index register and a length register, in either data address generator on the ADSP-21000 family. These registers are used in the program to point to elements of the data table, `sine_data`. Elements of this table are accessed indirectly within this program. Specifically, index registers I0 - I7 are used if the data table containing all the constants is put in data memory and index registers I8 - I15 are used if the data table is put in program memory. The variable `mem` must be defined as program memory, `PM`, or data memory, `DM`.

The include file, `asm_glob.h`, contains definitions of `mem`, `l_reg`, and `i_reg`. You can alter these definitions to suit your needs.

The second listing, `sinetest.asm`, is an example of a routine that calls the cosine and sine routines.

There are two entry points in the subroutine, `sin.asm`. They are labeled `cosine` and `sine`. Code execution begins at these labels. The calling program uses these labels by executing the instruction

```
call sine (db);
```

or

```
call cosine (db);
```

with the argument *x* in register F0. These calls are delayed branch calls that efficiently use the instruction pipeline on the ADSP-21000 family. In a delayed branch, the two instructions following the branch instruction are executed prior to the branch. This prevents the need to flush an instruction pipeline before taking a branch.

2 Trigonometric, Mathematical & Transcendental Functions

2.1.2 Code Listings

2.1.2.1 Sine/Cosine Approximation Subroutine

```
/************************************************************************

File Name
   SIN.ASM

Version
0.03  7/4/90

Purpose
   Subroutine to compute the Sine or Cosine values of a floating point input.

Equations Implemented
   Y=SIN(X) or
   Y=COS(X)

Calling Parameters
   F0 = Input Value X=[6E-20, 6E20]
   l_reg=0

Return Values
   F0 = Sine (or Cosine) of input Y=[-1,1]

Registers Affected
   F0, F2, F4, F7, F8, F12
   i_reg

Cycle Count
   38 Cycles

# PM Locations
   34 words

# DM Locations
   11 Words

************************************************************************/
```

Trigonometric, Mathematical & Transcendental Functions 2

```
#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;

#define half_PI  1.57079632679489661923

.GLOBAL   cosine, sine;

/**** Cosine/Sine approximation program starts here.  ****/
/**** Based on algorithm found in Cody and Waite.     ****/

cosine:
          i_reg=sine_data;                 /*Load pointer to data*/
          F8=ABS F0;                       /*Use absolute value of input*/
          F12=0.5;                         /*Used later after modulo*/
          F2=1.57079632679489661923;       /* and add PI/2*/

          JUMP compute_modulo (DB);        /*Follow sin code from here!*/
          F4=F8+F2, F2=mem(i_reg,1);
          F7=1.0;                                /*Sign flag is set to 1*/

sine:
          i_reg=sine_data;                 /*Load pointer to data*/
          F7=1.0;                                /*Assume a positive sign*/
          F12=0.0;                         /*Used later after modulo*/
          F8=ABS F0, F2=mem(i_reg,1);
          F0=PASS F0, F4=F8;
          IF LT F7=-F7;                    /*If input was negative, invert
sign*/

compute_modulo:
          F4=F4*F2;                        /*Compute fp modulo value*/
          R2=FIX F4;                       /*Round nearest fractional portion*/
          BTST R2 BY 0;                    /*Test for odd number*/
          IF NOT SZ F7=-F7;                /*Invert sign if odd modulo*/
          F4=FLOAT R2;                     /*Return to fp*/
          F4=F4-F12, F2=mem(i_reg,1);            /*Add cos adjust if necessary,
                                                   F4=XN*/


compute_f:
          F12=F2*F4, F2=mem(i_reg,1);            /*Compute XN*C1*/
          F2=F2*F4, F12=F8-F12;                  /*Compute |X|-XN*C1, and
XN*C2*/
          F8=F12-F2, F4=mem(i_reg,1);            /*Compute f=(|X|-XN*C1)-
XN*C2*/
          F12=ABS F8;                      /*Need magnitude for test*/
          F4=F12-F4, F12=F8;               /*Check for sin(x)=x*/
          IF LT JUMP compute_sign;         /*Return with result in F1*/

compute_R:
          F12=F12*F12, F4=mem(i_reg,1);
```

(listing continues on next page)

```
          LCNTR=6, DO compute_poly UNTIL LCE;
          F4=F12*F4, F2=mem(i_reg,1);             /*Compute sum*g*/

compute_poly:
          F4=F2+F4;                          /*Compute sum=sum+next r*/
          F4=F12*F4;                         /*Final multiply by g*/
          RTS (DB), F4=F4*F8;                /*Compute f*R*/
          F12=F4+F8;                         /*Compute Result=f+f*R*/

compute_sign:
          F0=F12*F7;                         /*Restore sign of result*/
          RTS;                               /*This return only for sin(eps)=eps
path*/

.ENDSEG;



.SEGMENT/SPACE Assembly_Library_Data_Space;
.PRECISION=MEMORY_PRECISION;


.VAR sine_data[11] =
                  0.31830988618379067154,   /*1/PI*/
                  3.14160156250000000000,   /*C1, almost PI*/
                 -8.90891020676153735661 7E-6, /*C2, PI=C1+C2*/
                  9.536743164E-7,           /*eps, sin(eps)=eps*/
                 -0.737066277507114174E-12, /*R7*/
                  0.160478446323816900E-9,  /*R6*/
                 -0.250518708834705760E-7,  /*R5*/
                  0.275573164212926457E-5,  /*R4*/
                 -0.198412698232225068E-3,  /*R3*/
                  0.833333333327592139E-2,  /*R2*/
                 -0.166666666666659653;     /*R1*/

.ENDSEG;
```

Trigonometric, Mathematical & Transcendental Functions 2

Listing 2.1 sin.asm

2.1.2.2 Example Calling Routine

```
/
*************************************************************************

File Name
    SINTEST.ASM

Purpose
    Example calling routine for the sine function.


*************************************************************************/

#include “asm_glob.h”;
#include “def21020.h”;
#define N 4
#define PIE 3.141592654

.SEGMENT/DM     dm_data;                  /* Declare variables in data memory
*/
.VAR input[N]= PIE/2, PIE/4, PIE*3/4, 12.12345678;          /* test data */
.VAR output[N]                                              /* results here */
.VAR correct[N]=1.0, .707106781, .707106781, -.0428573949;   /* correct results
*/
.ENDSEG;

.SEGMENT/PM
          pm_rsti;        /* The reset vector resides in this space */
          DMWAIT=0x21;    /* Set data memory waitstates to zero */
          PMWAIT=0x21;    /* Set program memory waitstates to zero */
          JUMP start;

.ENDSEG;


.EXTERN         sine;
.SEGMENT/PM     pm_code;

start:
          bit set mode2 0x10; nop; read cache 0; bit clr mode2 0x10;
          M1=1;
          B0=input;
          L0=0;
          I1=output;
          L1=0;

          lcntr=N, do calcit until lce;
                CALL sine (db);
                l_reg=0;
                f0=dm(i0,m1);
calcit:
          dm(i1,m1)=f0;

end:
```

2 Trigonometric, Mathematical & Transcendental Functions

```
            IDLE;
.ENDSEG;
```

Listing 2.2 sintest.asm

2.2 TANGENT APPROXIMATION

The tangent function is one of the fundamental trigonometric signal processing operations. This section shows how to approximate the tangent function in software. The algorithm used is taken from [CODY].

Tan(x) is calculated in three steps:

1. The argument x (which may be any real value) is reduced to a related argument *f* with a magnitude less than $\pi/4$ (that is, *t* has a range of $\pm \pi/2$).

2. *Tan(f)* is computed using a min-max polynomial approximation.

3. The desired function is reconstructed.

2.2.1 Implementation

The implementation of the tangent approximation algorithm uses 38 instruction cycles and consists of three logical steps.

First, the argument *x* is reduced to the argument *f*. This argument reduction is done in the sections labeled `compute_modulo` and `compute_f`.

The factor $\pi/2$ is required in the computation to reduce *x* (which may be any floating-point value) to *f* (which is a normalized value with a range of $\pm \pi/2$). To get an accurate result, the constants C_1 and C_2 are chosen so that $C_1 + C_2$ approximates $\pi/2$ to three or four decimal places beyond machine precision. The value C_1 is chosen to be close to $\pi/2$ and C_2 is a factor that is added to C_1 that results in a very accurate representation of $\pi/2$.

Notice that in the argument reduction, the assembly instructions are all multifunction instructions. ADSP-21000 family processors can execute a data move or a register move in parallel with a computation. Because multifunction instructions execute in a single cycle, the overhead for the memory move is eliminated.

A special case is if $tan(x) = x$. This occurs when the absolute value of f is less than epsilon. This value is very close to *0*. In this case, a jump is executed and the tangent function is calculated using the values of *f* and *1* in the final divide.

The second step is to approximate the tangent function using a min-max polynomial expansion. Two calculations are performed, the calculation of *P(g)* and the calculation of *Q(g)*, where *g* is just *f * f*. Four coefficients are used in calculation of both *P(g)* and *Q(g)*. The section labeled `compute_P` makes this calculation:

$$P(g) = ((p3 * g + p2) * g + p1) * g * f$$

Where $g = f*f$ and *f* is the reduced argument of x. The value *f * P(g)* is stored in the register F8.

The section labeled `compute_Q`, makes this calculation:

$$Q(g) = ((q_3*g + q_2)*g + q_1) * g + q_0$$

The third step in the calculation of the tangent function is to divide *P(g)* by *Q(g)*. If the argument, *f*, is even, then the tangent function is

$$tan(f) = f * P(g)/Q(g)$$

If the argument, *f*, is odd then the tangent function is

$$tan(f) = -f * P(g)/Q(g)$$

Finally, the value *N* is multiplied in and the reconstructed function *tan(x)* is returned in the register F0.

Similarly, the cotangent function is easily calculated by inverting the polynomial

$cotan(f) = Q(g)/–f * P(g)$

2.2.2 Code Listing–Tangent Subroutine

```
/
*****************************************************************************

File Name
    TAN.ASM

Version
    Version 0.03  7/5/90

Purpose
    Subroutine to compute the tangent of a floating point input.


Equations Implemented
    Y=TAN(X)

Calling Parameters
    F0 = Input Value X=[6E-20, 6E20]
    l_reg = 0 (usually L3)

Return Values
    F0 = tangent of input X

Registers Affected
    F0, F1, F2, F4, F7, F8, F12
    i_reg (usually I3)

Cycle Count
    38 Cycles
```

```
# PM Locations

# DM Locations

*****************************************************************************/


#include “asm_glob.h”
.SEGMENT/PM      Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   tan;

tan:      i_reg=tangent_data;
          F8=PASS F0, F2=mem(i_reg,1);            /* Use absolute value of input
*/

compute_modulo:
          F4=F8*F2, F1=F0;                 /* Compute fp modulo value */
          R2=FIX F4, F12=mem(i_reg,1);            /* Rnd nearest fractional
                                                     portion */
          F4=FLOAT R2, R0=R2;              /* Return to fp */

compute_f:
          F12=F12*F4, F2=mem(i_reg,1);            /* Compute XN*C1 */
          F2=F2*F4, F12=F8-F12;                   /* Compute X-XN*C1, and XN*C2
*/
          F8=F12-F2, F4=mem(i_reg,1);             /* Compute f=(X-XN*C1)-XN*C2
*/
          F12=ABS F8, F7=F8;
          F4=F12-F4, F12=mem(i_reg,1);            /* Check for TAN(x)=x */
          IF LT JUMP compute_quot;         /* Compute quotient with NUM=f
                                                  DEN=1 */

compute_P:
          F12=F8*F8, F4=mem(i_reg,1);             /* g=f*f */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute p3*g */
          F4=F2+F4;                        /* Compute (p3*g + p2) */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute (p3*g + p2)*g */
          F4=F2+F4;                        /* Compute (p3*g + p2)*g + p1 */
          F4=F12*F4;                       /* Compute
                                                  ((p3*g + p2)*g + p1)*g */
          F4=F4*F8;                        /* Compute
                                                  ((p3*g + p2)*g +p1)*g*f */
          F8=F4+F8, F4=mem(i_reg,1);       /* Compute f*P(g) */
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
compute_Q:
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F4=F2+F4;                          /* Compute sum=sum+next q */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F4=F2+F4;                          /* Compute sum=sum+next q */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F12=F2+F4, F7=F8;                  /* Compute sum=sum+next q */



compute_quot:
          BTST R0 BY 0;                      /* Test LSB */
          IF NOT SZ F12=-F7, F7=F12;         /* SZ true if even value*/
          F0=RECIPS F12;                          /* Get 4 bit seed R0=1/D */
          F12=F0*F12, F11=mem(i_reg,1);      /* D(prime) = D*R0 */
          F7=F0*F7, F0=F11-F12;                   /* F0=R1=2-D(prime), F7=N*R0
*/
          F12=F0*F12;                        /* F12=D(prime)=D(prime)*R1 */
          F7=F0*F7, F0=F11-F12;                   /* F7=N*R0*R1, F0=R2=2-
D(prime) */
          RTS (DB), F12=F0*F12;                   /* F12=D(prime)=D(prime)*R2 */
          F7=F0*F7, F0=F11-F12;                   /* F7=N*R0*R1*R2,
                                                  F0=R3=2-D(prime)*/
          F0=F0*F7;                          /* F7=N*R0*R1*R2*R3 */



.ENDSEG;



.SEGMENT/SPACE Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;
.VAR tangent_data[13] =  0.6366197723675834308,    /* 2/PI */
                  1.57080078125,                   /* C1, almost PI/2 */
                 -4.4544551033807686783O8E-6,            /* C2, PI/2=C1+C2 */
                  9.536743164E-7,                  /* eps, TAN(eps)=eps */
                  1.0,                             /* Used in one path */
                 -0.7483634966612065149E-5,        /* P3 */
                  0.2805918241169988906E-2,        /* P2 */
                 -0.1282834704095743847,           /* P1 */
```

```
                -0.2084480442203870948E-3,      /* Q3 */
                 0.2334485282206872802E-1,      /* Q2 */
                -0.4616168037429048840,         /* Q1 */
                 1.0,                           /* Q0 */
                 2.0;                           /* Used in divide */

.ENDSEG;
```

Listing 2.3 tan.asm

2.3 ARCTANGENT APPROXIMATION

The arctangent function is one of the fundamental trigonometric signal processing operations. Arctangent is often used in the calculation of the phase of a signal and in the conversion between Cartesian and polar data representations.

This section shows how to approximate the arctangent function in software. The algorithm used is taken from [CODY].

Calculation of *atan(x)* is done in three steps:

1. The argument *x* (which may be any real value) is reduced to a related argument f with a magnitude less than or equal to *2– R(3)*.

2. *Atan(f)* is computed using a rational expression.

3. The desired function is reconstructed.

2.3.1 Implementation

This implementation of the tangent approximation algorithm uses 82 instruction cycles, in the worst case. It follows the three logical steps listed above.

The assembly code module can be called to compute either of two functions, `atan` or `atan2`. The `atan` function returns the arctangent of an argument x. The `atan2` function returns the arctangent of y/x. This form

$$H=atan(y/x)$$

is especially useful in phase calculations.

First, the argument x is reduced to the argument f. This argument reduction relies on the symmetry of the arctangent function by using the identity

$$arctan(x) = -arctan(-x)$$

The use of this identity guarantees that approximation uses a non-negative x. For values that are greater than one, a second identity is used in argument reduction

$$arctan(x) = \pi/2 - arctan\ (1/x)$$

The reduction of x to the argument F is complete with the identity

$$arctan(x) = \pi/6 + arctan(f)$$

where

$$f = (x - (R(3) - 1) / (R(3) + x)$$

Just like tangent approximation, the second step in the arctangent calculation is computing a rational expression of the form

$$R = g * P(g) / Q(g)$$

where

$$g * P(g) = (P1 * g + P0) * g$$

and

$$Q(g) = (g + q1) * g + Q0$$

Notice that an eight-cycle macro, `divide`, is implemented for division. This macro is used several times in the program.

The final step is to reconstruct the *atan(x)* from the *atan(f)* calculation.

2.3.2 Listing–Arctangent Subroutine

```
/
*******************************************************************************

File Name
   ATAN.ASM

Version
   Version 0.01   3/20/91

Purpose
   Subroutine to compute the arctangent values of a floating point input.


Equations Implemented
   atan-       H=ATAN(Y)
   atan2-      H=ATAN(Y/X)
  where H is in radians

Calling Parameters
   F0 = Input Value Y=[6E-20, 6E20]
   F1 = Input Value X=[6E-20, 6E20] (atan2 only)
   l_reg=0

Return Values
   F0 = ArcTangent of input
      =[-pi/2,pi/2]   for atan
      =[-pi,pi]       for atan2


Registers Affected
   F0, F1, F2, F4, F7, F8, F11, F12, F15
    i_reg
    ms_reg
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
Cycle Count
   atan    61 Cycles maximum
   atan2   82 cycles maximum


# PM Locations

# DM Locations

*****************************************************************************/

/* The divide Macro used in the arctan routine*/
/* ---------------------------------------------------------------------------
-------
DIVIDE - Divide Macro

Register Usage:
    q       = f0-f15        Quotient
    n       = f4-f7         Numerator
    d       = f12-f15       Denominator
    two     = f8-f11        must have 2.0 pre-stored
    tmp     = f0-f3

Indirectly affected registers:
    ASTAT,STKY

Looping:    none

Special Cases:
    q may be any register, all others must be distinct.

Cycles: 8

Created:    3/19/91
---------------------------------------------------------------------------
-----*/

#define DIVIDE(q,n,d,two,tmp)                                               \
        n=RECIPS d, tmp=n;          /* Get 8 bit seed R0=1/D*/              \
        d=n*d;                      /* D(prime) = D*R0*/                    \
        tmp=tmp*n, n=two-d;         /* N=2-D(prime), TMP=N*R0*/             \
        d=n*d;                      /* D=D(prime)=D(prime)*R1*/             \
        tmp=tmp*n, n=two-d;         /* TMP=N*R0*R1, N=R2=2-D(prime)*/       \
        d=n*d;                      /* D=D(prime)=D(prime)*R2*/             \
        tmp=tmp*n, n=two-d;         /* TMP=N*R0*R1*R2, N=R3=2-D(prime)*/    \
        q=tmp*n
```

```
#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;

.PRECISION=MACHINE_PRECISION;

.GLOBAL     atan, atan2;

atan2:          i_reg=atan_data;
                F11= 2.0;
                F2= 0.0;
                F1=PASS F1;
                IF EQ JUMP denom_zero;      /* if Denom. = 0, goto special
case*/
                IF LT F2=mem(11,i_reg);     /* if Denom. < 0, F2=pi (use at
end)*/

overflow_tst:   R4=LOGB F0, F7=F0;         /* Detect div overflow for atan2*/
                R1=LOGB F1, F15=F1;
                R1=R4-R1;                       /* Roughly exp. of quotient*/
                R4=124;                    /* Max exponent - 3*/
                COMP(R1,R4);
                IF GE JUMP overflow;       /* Over upper range? Goto overflow*/
                R4=-R4;
                COMP(R1,R4);
                IF LE JUMP underflow;           /* Over lower range? Goto
underflow*/

do_division:    DIVIDE(F0,F7,F15,F11,F1);
                JUMP re_entry (DB);

atan:           R10= 0;                    /* Flags multiple of pi to add at
end*/
                F15=ABS F0;

                i_reg=atan_data;                /* This init is redundant for
atan2f*/
                F11= 2.0;                       /* Needed for divide*/
                F2= 0.0;                        /* Result is not in Quad 2 or
3 */

re_entry:       F7 = 1.0;
                COMP(F15,F7), F4=mem(0,i_reg); /* F4=2-sqrt(3)*/
                IF LE JUMP tst_f;          /* If input<=1, do arctan(input)*/
                                           /* else do arctan(1/input)+const*/
                DIVIDE(F15,F7,F15,F11,F1); /* do x=1/x*/

                R10 = 2;                        /* signal to add const at
end*/
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
tst_f:          COMP(F15,F4);              /* Note F4 prev. loaded from memory*/
                IF LT JUMP tst_for_eps;
                R10=R10+1,  F4=mem(1,i_reg);      /* F4=sqrt(3)*/
                F12=F4*F15;
                F7=F12-F7;                 /* F7=F12-1.0*/
                F15=F4+F15;
                DIVIDE(F15,F7,F15,F11,F1); /*  = (sqrt(3)*x-1)/(x+sqrt(3))*/

tst_for_eps:    F7=ABS F15, F4=mem(2,i_reg);      /* F4=eps (i.e. small)*/
                COMP(F7,F4);
                IF LE JUMP tst_N;          /* if x<=eps, then h=x*/
                F1=F15*F15, F4=mem(3,i_reg);      /* else . . .*/
                F7=F1*F4,   F4=mem(4,i_reg);
                F7=F7+F4,   F4=mem(5,i_reg);
                F7=F7*F1;
                F12=F1+F4,  F4=mem(6,i_reg);
                F12=F12*F1;
                F12=F12+F4;
                DIVIDE(F7,F7,F12,F11,F1);  /* e=((p1*x^2 +p0)x^2)/(x^2
+q1)x^2+q0*/
                F7=F7*F15;
                F15=F7+F15;                /* h=e*x+x*/

tst_N:          R1=R10-1, R7=mem(i_reg,7);  /* if R10 > 1, h=-h; dummy read*/
                ms_reg=R10;
                IF GT F15=-F15;
                F4 = mem(ms_reg,i_reg);     /* index correct angle addend to h
*/
                F15=F15+F4;                /* h=h+ a*pi      */

tst_sign_y:     F2=PASS F2;                /* if (atan2f denom <0) h=pi-h else
h=h  */
                IF NE F15=F2-F15;
tst_sign_x:     RTS (DB), F0=PASS F0;      /* if (numer<0) h=-h else h=h*/
                IF LT F15=-F15;
                F0=PASS F15;               /* return with result in F0!*/


underflow:      JUMP tst_sign_y (DB);
                F15=0;
                NOP;


overflow:       JUMP tst_sign_x (DB);
                F15=mem(9,i_reg);          /* Load pi/2*/
                NOP;


denom_zero:     F0=PASS F0;                /* Careful: if Num !=0, then
overflow*/
                IF NE JUMP overflow;

error:          RTS;                       /* Error: Its a 0/0!*/
```

```
.ENDSEG;


.SEGMENT/SPACE  Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;

.VAR atan_data[11] =    0.26794919243112270647,     /* 2-sqrt(3)      */
                        1.73205080756887729353,     /* sqrt(3)  */
                        0.000244140625,             /* eps            */
                        -0.720026848898E+0,         /* p1             */
                        -0.144008344874E+1,         /* p0             */
                        0.475222584599E+1,          /* q1             */
                        0.432025038919E+1,          /* q0             */
                        0.00000000000000000000,     /* 0*pi           */
                        0.52359877559829887308,     /* pi/6           */
                        1.57079632679489661923,     /* pi/2           */
                        1.04719755119659774615,     /* pi/3           */
                        3.14159265358979323846;     /* pi             */
.ENDSEG;
```

Listing 2.4 atan.asm

2.4 SQUARE ROOT & INVERSE SQUARE ROOT APPROXIMATIONS

An ADSP-21000 family DSP can perform the square root function, *sqrt(y)*, and the inverse square root, *isqrt(y)*, quickly and with a high degree of precision. These functions are typically used to calculate magnitude functions of FFT outputs, to implement imaging and graphics algorithms, and to use the DSP as a fast math coprocessor.

A square root exists (and can be calculated) for every non-negative floating-point number. Calculating the square root of a negative number gives an imaginary result. To calculate the square root of a negative number, take the absolute value of the number and use *sqrt(y)* or *isqrt(y)* as defined in this section. Remember that the result is really an imaginary number.

The ADSP-21000 family program that calculates *isqrt(y)* is based on the Newton-Raphson iteration algorithm in [CAVANAGH]. Computation of the function begins with a low-accuracy initial approximation. The Newton-

2 Trigonometric, Mathematical & Transcendental Functions

Raphson iteration is then used for successively more accurate approximations.

Once *isqrt(y)* is calculated it only takes one additional operation to calculate *sqrt(y)*. Given the input, *y*, the square root, *x=R(*y), is determined as follows

$$x = sqrt(y) = y * 1/sqrt(y)$$

Given an initial approximation x_n for the inverse square root of y, isqrt(y), the Newton-Raphson iteration is used to calculate a more accurate approximation, x_{n+1}.

Newton-Raphson iteration: $x_{n+1} = (0.5)\ (x_n)\ (3 - (x_n^2)\ )\ (y))$

The number of iterations determines the accuracy of the approximation. For a 32-bit floating point representation with a 24-bit mantissa, two iterations are sufficient to achieve an accuracy to ±1 LSB of precision. For an extended 40-bit precision representation that has a 32-bit mantissa, three iterations are needed.

For example, suppose that you need to calculate the square root of 30. If you use 5.0 as an initial approximation of the square root, the inverse square root approximation, 1/ 30, is 0.2. Therefore, given $y=30$ and $x_n=0.2$, one iteration yields

$$x_{n+1} = (0.5)\ (0.2)\ (3 - (0.2^2)\ )\ (30))$$
$$x_{n+1} = 0.18$$

A second iteration using 0.18 as an input yields

$$x_{n+2} = (0.5)\ (0.18)\ (3 - (0.18^2)\ )\ (30))$$
$$x_{n+2} = 0.18252$$

And finally a third iteration using 0.18252 as an input yields

$$x_{n+2} = (0.5)\ (0.18252)\ (3 - (0.18252^2)\ )\ (30))$$
$$x_{n+2} = 0.182574161$$

Therefore the approximation of the inverse square root of 30 after three iterations is 0.182574161. To calculate the square root, just multiply the two numbers together

*30 * 0.182574161 = 5.47722483*

The actual square root of 30 is 5.477225575. If the initial approximation is accurate to four bits, the final result is accurate to about 32 bits of precision.

2.4.1 Implementation

To implement the Newton-Raphson iteration method for calculating an inverse square root on an ADSP-21000 processor, the first task is to calculate the initial low-accuracy approximation.

The ADSP-21000 family instruction set includes the instruction `RSQRTS` that, given the floating point input Fx, creates a 4-bit accurate seed for $1/\sqrt{Fx}$. This seed, or low accuracy approximation, is determined by using the six MSBs of the mantissa and the LSB of the unbiased exponent of Fx to access a ROM-based look up table.

Note that the instruction `RSQRTS` only accepts inputs greater than zero. A *±Zero* returns *±Infinity*, *±Infinity* returns *±Zero*, and a *NAN* (not-a-number) or negative input returns an all 1's result. You can use conditional logic to assure that the input value is greater than zero.

To calculate the seed for an input value stored in register F0, use the following instruction:

```
F4=RSQRTS F0;        /*Fetch seed*/
```

Once you have the initial approximation, it is a easy to implement the Newton-Raphon iteration in ADSP-21000 assembly code. With the approximation now in F4 and with F1 and F8 initialized with the constants 0.5 and 3 respectively, one iteration of the Newton-Raphson is

implemented as follows:

```
F12=F4*F4;              /* F12=X0^2 */
F12=F12*F0;             /* F12=C*X0^2 */
F4=F2*F4, F12=F8-F12;   /* F4=.5*X0, F10=3-C*X0^2 */
F4=F4*F12;              /* F4=X1=.5*X0(3-C*X0^2) */
```

The register F4 contains a reasonably accurate approximation for the inverse square root. Successive iterations are made by repeating the above four lines or code. The square root of F0 is calculated by multiplying the approximation F4, by the initial input F0:

```
F0=F4*F0;               /* X=sqrt(Y)=Y/sqrt(Y) */
```

2.4.2 Code Listings

There are four subroutine listings below that illustrate how to calculate *sqrt(y)* and i*sqrt(y)*. Two are for single precision (24-bit mantissa), and two for extended precision (32-bit mantissa).

2.4.2.1 SQRT Approximation Subroutine

```
/
*******************************************************************************
File Name
   SQRT.ASM
```

```
Version
    Version 0.02  7/6/90

Purpose
    Subroutine to compute the square root of x using the 1/sqrt(x) approximation.

Equations Implemented
    X = sqrt(Y) = Y/sqrt(Y)

Calling Parameters
    F0 = Y Input Value
    F8 = 3.0
    F1 = 0.5

Return Values
    F0 = sqrt(Y)

Registers Affected
    F0, F4, F12

Cycle Count
    14 Cycles

# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”
```

2 Trigonometric, Mathematical & Transcendental Functions

```
.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL sqrt;

sqrt:
   F4=RSQRTS F0;                     /*Fetch seed*/

   F12=F4*F4;                        /*F12=X0^2*/
   F12=F12*F0;                       /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                    /*F4=.5*X0, F10=3-C*X0^2*/
   F4=F4*F12;                        /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                        /*F12=X1^2*/
   F12=F12*F0;                       /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                    /*F4=.5*X1, F10=3-C*X1^2*/
   F4=F4*F12;                        /*F4=X2=.5*X1(3-C*X1^2)*/

   F12=F4*F4;                        /*F12=X2^2*/
   F12=F12*F0;                       /*F12=C*X2^2*/
   RTS (DB), F4=F1*F4, F12=F8-F12;   /*F4=.5*X2, F10=3-C*X2^2*/
   F4=F4*F12;                        /*F4=X3=.5*X2(3-C*X2^2)*/

   F0=F4*F0;                         /*X=sqrt(Y)=Y/sqrt(Y)*/
.ENDSEG;
```

Listing 2.5 sqrt.asm

2.4.2.2 ISQRT Approximation Subroutine

```
/
*******************************************************************************
File Name
   ISQRT.ASM
```

```
Version
   Version 0.02  7/6/90

Purpose
   Subroutine to compute the inverse square root of x using the 1/
sqrt(x)                        approximation.

Equations Implemented
   X = isqrt(Y) = 1/sqrt(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = isqrt(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   13 Cycles

#PM Locations

#DM Locations

*******************************************************************************/
```

2 Trigonometric, Mathematical & Transcendental Functions

```
#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL isqrt;

isqrt:
   F4=RSQRTS F0;                    /*Fetch seed*/

   F12=F4*F4;                       /*F12=X0^2*/
   F12=F12*F0;                      /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X0, F10=3-C*X0^2*/
   F4=F4*F12;                       /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                       /*F12=X1^2*/
   F12=F12*F0;                      /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X1, F10=3-C*X1^2*/
   F4=F4*F12;                       /*F4=X2=.5*X1(3-C*X1^2)*/

   F12=F4*F4;                       /*F12=X2^2*/
   F12=F12*F0;                      /*F12=C*X2^2*/
   RTS (DB), F4=F1*F4, F12=F8-F12;  /*F4=.5*X2, F10=3-C*X2^2*/
   F4=F4*F12;                       /*F4=X3=.5*X2(3-C*X2^2)*/
                                    /* =isqrt(Y)=1/sqrt(Y)*/

.ENDSEG;
```

Listing 2.6 isqrt.asm

2.4.2.3 SQRTSGL Approximation Subroutine

```
/
*****************************************************************************
File Name
   SQRTSGL.ASM
```

```
Version
   Version 0.02  7/6/90

Purpose
   Subroutine to compute the square root of x to single-precision (24 bits
mantissa) using the 1/sqrt(x) approximation.

Equations Implemented
   X = sqrtsgl(Y) = Y/sqrtsgl(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = sqrtsgl(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   10 Cycles
```

```
# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL sqrtsgl;

sqrtsgl:
   F4=RSQRTS F0;              /*Fetch seed*/

   F12=F4*F4;                 /*F12=X0^2*/
   F12=F12*F0;                /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;             /*F4=.5*X0, F12=3-C*X0^2*/
   F4=F4*F12;                 /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                 /*F12=X1^2*/
   F12=F12*F0;                /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;             /*F4=.5*X1, F12=3-C*X1^2*/
   F4=F4*F12;                 /*F4=X2=.5*X1(3-C*X1^2)*/
   F0=F4*F0;                  /*X=sqrtsgl(Y)=Y/sqrtsgl(Y)*/
.ENDSEG;
```

Listing 2.7 sqrtsgl.asm

2.4.2.4 ISQRTSGL Approximation Subroutine

```
/
*****************************************************************************
File Name
   ISQRTSGL.ASM
```

```
Version
   Version 0.02        7/6/90

Purpose
   Subroutine to compute the inverse square root of x to single-precision (24
bits                   mantissa) using the 1/sqrt(x) approximation.

Equations Implemented
   X = isqrtsgl(Y) = 1/sqrtsgl(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = isqrtsgl(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   9 Cycles
```

2 Trigonometric, Mathematical & Transcendental Functions

```
# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL isqrtsgl;

isqrtsgl:
   F4=RSQRTS F0;                    /*Fetch seed*/

   F12=F4*F4;                       /*F12=X0^2*/
   F12=F12*F0;                      /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X0, F12=3-C*X0^2*/
   F4=F4*F12;                       /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                       /*F12=X1^2*/
   RTS(DB), F12=F12*F0;                   /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X1, F12=3-C*X1^2*/
   F0=F4*F12;                       /*F4=X2=.5*X1(3-C*X1^2)*/
                                    /* =isqrtsgl(Y)=1/sqrtsgl(Y)*/
.ENDSEG;
```

Listing 2.8 isqrtsgl.asm

2.5 DIVISION

The ADSP-21000 family instruction set includes the `RECIPS` instruction to simplify the implementation of floating-point division.

2.5.1 Implementation

The code performs floating-point division using an in iterative convergence algorithm. The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: F0 = numerator, F12= denominator, F11 = 2.0. The quotients is returned in F0. (In the code listing, the two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

The algorithm is supplied with a startup seed which is a low-precision reciprocal of the denominator. This seed is generated by the `RECIPS` instruction. `RECIPS` creates an 8-bit accurate seed for *1/Fx*, the reciprocal of *Fx*. The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the *Fx* mantissa as an index. The unbiased exponent of the seed is calculated as the twos complement of the

unbiased *Fx* exponent, decremented by one; that is, if *e* is the unbiased exponent of *Fx*, then the unbiased exponent of $Fn = -e - 1$. The sign of the seed is the sign of the input. *±Zero* returns *±Infinity* and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is *±Zero*. A *NAN* input returns an all 1's result.

2.5.2 Code Listing–Division Subroutine

```
/*
File Name
   F.ASM

Version
   Version 0.03

Purpose
   An implementation of division using an Iterative Convergent Divide
Algorithm.

Equations Implemented
    Q = N/D

Calling Parameters
  F0  = N Input Value
    F12 = D Input Value
    F11 = 2.0

Return Values
    F0 = Quotient of input

Registers Affected
    F0, F7, F12

Cycle Count
    8 Cycles  (6 Cycles for single precision)

# PM Locations

# DM Locations
```

```
*/


#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;

.PRECISION=MACHINE_PRECISION;

.GLOBAL divide;

divide:         F0=RECIPS F12, F7=F0;   /*Get 4 bit seed R0=1/D*/
                F12=F0*F12;             /*D(prime) = D*R0*/
                F7=F0*F7, F0=F11-F12;   /*F0=R1=2-D(prime), F7=N*R0*/
                F12=F0*F12;             /*F12=D(prime)=D(prime)*R1*/
                F7=F0*F7, F0=F11-F12;   /*F7=N*R0*R1, F0=R2=2-
D(prime)*/

/* Remove next two instructions for 1 LSB accurate single-precision
result */
        RTS (DB), F12=F0*F12; {F12=D(prime)=D(prime)*R2}
        F7=F0*F7, F0=F11-F12; {F7=N*R0*R1*R2, F0=R3=2-D(prime)}


                F0=F0*F7;       {F0=N*R0*R1*R2*R3}

.ENDSEG;
```

Listing 2.9 Divide..asm

2.6 LOGARITHM APPROXIMATIONS

Logarithms (in base *e, 2,* and *10*) can be approximated for any non-negative floating point number. The *Software Manual for the Elementary Functions* by William Cody and William Waite explains how the computation of a logarithm involves three distinct steps:

1. The given argument (or input) is reduced to a related argument in a small, logarithmically symmetric interval about *1*.

2. The logarithm is computed for this reduced argument.

3. The desired logarithm is reconstructed from its components.

The algorithm can calculate logarithms of any base (base *e*, *2*, or *10*); the

code is identical until step three, so only one assembly-language module is needed.

The first step is to take a given floating point input, Y, and reduce it to $Y=f * 2^N$ where $0.5 \leq f < 1$. Given that $X = log(Y)$, X also equals

$$X=log_e(Y)$$
$$X=log_e(f * 2^N)$$
$$X=log_e(f) + N*log_e(2)$$

$N * log_e(2)$ is the floating point exponent multiplied by $log_e(2)$, which is a constant. The term $log_e(f)$ must be calculated. The definition of the variable s is

$$s = (f - 1) / (f + 1)$$

Then

$$log_e(f) = log_e((1 + s) / (1 - s))$$

This equation is evaluated using a min-max rational approximation detailed in [CODY]. The approximation is expressed in terms of the auxiliary variable $z = 2s$.

Once the value of $log_e(f)$ is approximated, the equation

$$X = log_e(f) + N*log_e(2)$$

yields the solution for the natural (base-e) log of the input Y. To compute the base-*2* or base-*10* logarithm, the result X is multiplied by a constant equal to the reciprocal of $log_e(2)$, or $log_e(10)$, respectively.

2.6.1 Implementation

`LOGS.ASM` is an assembly-language implementation of the logarithm algorithm. This module has three entry points; a different base of logarithm is computed depending on which entry point is used. The label `LOG` is used for calling the algorithm to approximate the natural (base-*e*) logarithm, while the labels `LOG2` and `LOG10` are used for base-*2* and base-*10* approximations, respectively. When assembling the file `LOGS.ASM` you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-D`*`identifier`*

switch at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA logs
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA logs
```

The first step to compute any of the desired logarithms is to reduce the floating point input Y to the form

$$Y=f * 2^N$$

The ADSP-21000 family supports the IEEE Standard 754/854 floating point format. This format has a biased exponent and a significant with a "hidden" bit of 1. The hidden bit, although not explicitly represented, is implicitly presumed to exist and it offsets the exponent by one bit place.

For example, consider the floating point number 12.0. Using the IEEE standard, this number is represented as $0.5 * 2^{131}$. By adding the hidden one and unbiasing the exponent, *12.0* is actually represented as $1.5 * 2^4$. To get to the format $f * 2^N$ where $0.5 \leq f < 1$, you must scale $1.5 * 2^4$ by two to get the format $0.75 * 2^3$. The instruction `LOGB` extracts the exponent from our floating-point input. The exponent is then decremented, and the mantissa is scaled to achieve the desired format.

Use the value of f to approximate the value of the auxiliary variable z. The variable z is approximated using the following formula

$$z=znum/zden$$

where

$$znum = (f - 0.5) - 0.5$$

and

$$zden = (f * 0.5) - 0.5 \qquad for\ f>1/sqrt(2)$$

or

$znum = f - 0.5$ *and*
$zden = znum * 0.5 + 0.5$ *for* $f <= 1/sqrt(2)$

Once z is found, it is used to calculate the min-max rational approximation $R(z)$, which has the form

$$R(z) = z + z * r(z^2)$$

The rational approximation $r(z^2)$ has been derived and for $w=z^2$ is

$$r(z^2) = w * A(w)/B(w)$$

where A(w) and B(w) are polynomials in w, with derived coefficients a0, a1, and b0

$$A(w) = a1 * w + a0$$
$$B(w) = w + b0$$

$R(z)$ is the approximation of $log_e(f)$ and the final step in the approximation of $log_e(Y)$ is to add in $N*log_e(2)$. The coefficients *C0* and *C1* and the exponent N are used to determine this value.

If only the natural logarithm is desired, then the algorithm is complete and the natural *log (ln(Y))* is returned in the register F0. If $log_2(Y)$ was needed, then F0 is multiplied by *1/ln(2)* or *1.442695041*. If $log_{10}(Y)$ is needed, then F0 is multiplied by *1/ln(10)* or *0.43429448190325182765*.

2.6.2 Code Listing

The listing for module `LOGS.ASM` is below. The calling routine uses the appropriate label for the type of logarithm that is desired: `LOG` for natural log (base-*e*), `LOG2` for base-*2* and `LOG10` for base-*10*.

At assembly time, you must specify the memory space where the eight coefficients are stored (Program or Data Memory) by using either the `-DDM_DATA` or `-DPM_DATA` switch. Attempts to assemble `LOGS.ASM` without one of these switches result in an error.

2.6.2.1 Logarithm Approximation Subroutine

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
/*************************************************************
File Name
    LOGS.ASM

Version
    Version 0.03  8/6/90
    revised 26-APR-91

Purpose
    Subroutine to compute the logarithm (bases 2,e, and 10) of its floating
    point input.

Equations Implemented
    Y=LOG(X) or
    Y=LOG2(X) or
    Y=LOG10(X)

Calling Parameters
    F0 = Input Value
    l_reg=0;

Return Values
    F0 = Logarithm of input

Registers Affected
    F0, F1, F6, F7, F8, F10, F11, F12
    i_reg

Computation Time
    49 Cycles

#PM locations

#DM locations

*************************************************************/

#include “asm_glob.h”

.SEGMENT/PM      Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   log, log10, log2;

log2:
    CALL logs_core (DB);                    /*Enter same routine in two cycles*/
    R11=LOGB F0, F1=F0;              /*Extract the exponent*/
    F12=ABS F1;                      /*Get absolute value*/
    RTS (DB);
    F11=1.442695041;                 /*1/Log(2)*/
```

```
   F0=F11*F0;                          /*F0 = log2(X)*/

log10:
   CALL logs_core (DB);                      /*Enter same routine in two cycles*/
   R11=LOGB F0, F1=F0;                 /*Extract the exponent*/
   F12=ABS F1;                         /*Get absolute value*/
   RTS (DB);
   F11=0.43429448190325182765;               /*1/Log(10)*/
   F0=F11*F0;                          /*F12 = log10(X)*/

log:
   R11=LOGB F0, F1=F0;
   F12=ABS F1;

logs_core:      i_reg=logs_data;           /*Point to data array*/
   R11=R11+1;                          /*Increment exponent*/
   R7=-R11, F10=mem(i_reg,1);          /*Negate exponent*/
   F12=SCALB F12 BY R7;                      /*F12= .5<=f<1*/
   COMP(F12,F10), F10=mem(i_reg,1); /*Compare f > C0*/
   IF GT JUMP adjust_z (DB);
   F7=F12-F10;                         /*znum = f-.5*/
   F8=F7*F10;                          /*znum * .5*/
   JUMP compute_r (DB);
   F12=F8+F10;                         /*zden = znum * .5 + .5*/
   R11=R11-1;                          /*N = N - 1*/

adjust_z:
   F7=F7-F10;                          /*znum = f - .5 - .5*/
   F8=F12*F10;                         /*f * .5*/
   F12=F8+F10;                         /*zden = f * .5 + .5*/

compute_r:
   F0=RECIPS F12;                            /*Get 4 bit seed R0=1/D*/
   F12=F0*F12, F10=mem(i_reg,1);       /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                     /*F0=R1=2-D(prime), F7=N*R0*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1, F0=R2=2-D(prime)*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1*R2, F0=R3=2-D(prime)*/
   F6=F0*F7;                           /*F7=N*R0*R1*R2*R3*/
   F0=F6*F6, F8=mem(i_reg,1);          /*w = z^2*/
   F12=F8+F0, F8=mem(i_reg,1);               /*B(W) = w + b0*/
   F7=F8*F0, F8=mem(i_reg,1);          /* w*a1*/
   F7=F7+F8, F8=F0;                    /*A(W) = w * a1 + a0*/
   F0=RECIPS F12;                            /*Get 4 bit seed R0=1/D*/
   F12=F0*F12;                         /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                     /*F0=R1=2-D(prime), F7=N*R0*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1, F0=R2=2-D(prime)*/
```

2 Trigonometric, Mathematical & Transcendental Functions

```
   F12=F0*F12;                          /*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                      /*F7=N*R0*R1*R2, F0=R3=2-
D(prime)*/
   F7=F0*F7;                            /*F7=N*R0*R1*R2*R3*/
   F7=F7*F8;                            /*Compute r(z^2)=w*A(w)/B(w)*/

compute_R:
   F7=F6*F7;                            /* z*r(z^2)*/
   F12=F6+F7;                           /*R(z) = z + z * r(z^2)*/
   F0=FLOAT R11, F7=mem(i_reg,1);       /*F0=XN, F7=C2*/
   F10=F0*F7, F7=mem(i_reg,1);                /*F10=XN*C2, F7=C1*/
   RTS (DB);
   F7=F0*F7, F0=F10+F12;                      /*F0=XN*C2+R(z),
F7=XN*C1*/
   F0=F0+F7;                            /*F0 = ln(X)*/
.ENDSEG;

.SEGMENT/SPACE Assembly_Library_Data_Space;
.VAR logs_data[8] =    0.70710678118654752440,    /*C0 = sqrt(.5)*/
   0.5,                                           /*Constant used*/
   2.0,                                           /*Constant used*/
   -5.578873750242,                               /*b0*/
   0.1360095468621E-1,                            /*a1*/
   -0.4649062303464,                              /*a0*/
   -2.12194440054690582 7679E-4,                         /*C2*/
   0.693359375;                                   /*C1*/
.ENDSEG;
```

Listing 2.10 logs.asm

2.7 EXPONENTIAL APPROXIMATION

The exponential function (e^X or $exp(X)$) of a floating point number is computed by using an approximation technique detailed in the [CODY]. Cody and Waite explain how the computation of an exponent involves three distinct steps.

The first step is to reduce a given argument (or input) to a related argument in a small interval symmetric about the origin. If X is the input value for which you wish to compute the exponent, let

$$X = N*ln(C) + g \quad |g| \leq ln(C)/2.$$

Then

$$exp(X) = exp(g) * C^N$$

where $C = 2$, and N is calculated as $X/ln(C)$. Since the accuracy of the approximation critically depends on the accuracy of g, the effective precision of the ADSP-210xx processor must be extended during the calculation of g. Use the equation

$$g = (|X| - X_N * C_1) - X_N * C_2$$

where $C_1 + C_2$ represent $ln(C)$ to more than working precision, and X_N is the floating point representation of N. The values of C_1 and C_2 are precomputed and stored in program or data memory.

The second step is to compute the exponential for the reduced argument. Since you have now calculated N and g, you must approximate $exp(g)$. Cody and Waite have derived coefficients ($p1$, $q1$, etc.) especially for the approximation of $exp(g)/2$. The divide by two is added to counteract wobbly precision. The approximation of $exp(g)/2$ is

$$R(g) = 0.5 + (g*P(z))/Q(z) - g(P(z))$$

where

$$g*P(z) = ((p1 * z) + p0) * g,$$
$$Q(z) = (q1 * z) + q0,$$
$$z = g^2$$

The third step is to reconstruct the desired function from its components. The components of $exp(X)$ are $exp(g)=R(g)$, $C=2$, and $N=X/ln(2)$. Therefore:

$$\begin{aligned} exp(X) &= exp(g) * C^N \\ &= R(g) * 2^{(N+1)} \end{aligned}$$

Note that N was incremented by one due to the scaling that occurred in the approximation of $exp(g)$.

2.7.1 Implementation

`EXP.ASM` is an implementation of the exponent approximation for the ADSP-21000 family. When assembling the file `EXP.ASM` you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-D`*`identifier`* `switch` at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA exp
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA exp
```

Before the first step in the approximation of the exponential function is performed, the floating-point input value is checked to assure that it is in the allowable range. The floating-point input is limited by the machine precision of the processor calculating the function. For the ADSP-21000 family, the largest number that can be represented is $XMAX = 2^{127} – 1LSB$. Therefore, the largest input to the function *exp(X)* is *ln(XMAX)*, which is approximately 88. The smallest positive floating point number that can be represented is $XMIN=2^{–127}$. Computing *ln(XMIN)* gives approximately –88 as the largest negative input to the function *exp(x)*. Therefore, comparisons to *ln(XMIN)* and *ln(XMAX)* are the first instructions of the `EXP.ASM` module. If these values are exceeded, the subroutine ends and returns either an error *(X > ln(XMAX))* or a very small number *(X < ln(XMIN))*.

The code includes another comparison for the case where the input produces the output *1*. This only occurs when the input is equal to *0.000000001* or *1.0E-9*. If the input equals this value, the subroutine ends and returns a one.

The first step in the approximation is to compute *g* and *N* for the equation

$$exp(X) = exp(g) * 2^N$$

Where

$$N=X/ln(2)$$
$$g = (|X| - X_N * C_1) – X_N * C_2$$
$$C1 = 0.693359375$$
$$C2 = –2.12194440054690582 77E-4$$

Since multiplication requires fewer instructions than division, the constant *1/ln(2)* is stored in memory along with the precomputed values of C_1 and C_2. X_N is a floating point representation of N that can be easily computed using the `FIX` and `FLOAT` conversion features of the ADSP-210xx processors.

Given g and X_N, it is simple to compute the approximation for *exp(g)/2*

using the constants and equations outlined in the [CODY].

$$R(g) = 0.5 + (g * P(z)) / Q(z) - g(P(z))$$

where

$g*P(z) = ((p1 * z) + p0) * g,$
$Q(z) = (q1 * z) + q0$
$z = g^2$
$P1 = 0.59504254977591E-2$
$P0 = 0.24999999999992$
$Q2 = 0.29729363682238E-3$
$Q1 = 0.53567517645222E-1$
$Q0 = 0.5$

Once $R(g)$, the approximation for $exp(g)$, is calculated, the approximation for $exp(x)$ is derived by using the following equation:

$$\begin{aligned} exp(X) &= 2(approx(exp(g)/2)) * 2^N \\ &= approx(exp(g)/2) * 2^{(N+1)} \\ &= R(g) * 2^{(N+1)} \end{aligned}$$

The `SCALB` instruction scales the floating point value of $R(g)$ by the exponent $N + 1$.

2.7.2 Code Listings–Exponential Subroutine

```
/*************************************************************
File Name
    EXP.ASM
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
Version
   Version 0.03    8/6/90
   Modified        9/27/93

Purpose
   Subroutine to compute the exponential of its floating point input

Equations Implemented
   Y=EXP(X)

Calling Parameters
   F0 = Input Value
   l_reg=0;

Return Values
   F0 = Exponential of input

Registers Affected
   F0, F1, F4, F7, F8, F10, F12
   i_reg

Computation Time
   38 Cycles

# PM locations
   46 words

#DM locations
   12 words (could be placed in PM instead)
************************************************************/

#include “asm_glob.h”
.SEGMENT/PM            Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   exponential;
output_too_large: RTS (DB);
   F0=10000;    /*Set some error here*/
   F0=10000;
output_too_small: RTS (DB);
   F0 = .000001;
   F0 = .000001;
output_one:     RTS (DB);
   F0 = 1.0;
   F0 = 1.0;
exponential:    i_reg=exponential_data;   /*Skip one cycle after this*/
   F1=PASS F0;                            /*Copy into F1*/
   F12=ABS F1, F10=mem(i_reg,1);          /*Fetch maximum input*/
   COMP(F1,F10), F10=mem(i_reg,1);        /*Error if greater than max*/
   IF GT JUMP output_too_large;                 /*Return XMAX with error*/
   COMP(F1,F10), F10=mem(i_reg,1);        /*Test for input to small*/
   IF LT JUMP output_too_small;                 /*Return 0 with error*/
   COMP(F12,F10), F10=mem(i_reg,1);       /*Check for output 1*/
```

```
   IF LT JUMP output_one;                  /*Simply return 1*/
   F12=F1*F10, F8=F1;                      /*Compute N = X/ln(C)*/
   R4=FIX F12;                             /*Round to nearest*/
   F4=FLOAT R4, F0=mem(i_reg,1);           /*Back to floating point*/
compute_g:      F12=F0*F4, F0=mem(i_reg,1);            /*Compute
XN*C1*/
   F0=F0*F4, F12=F8-F12;                         /*Compute |X|-XN*C1,
and XN*C2*/
   F8=F12-F0, F0=mem(i_reg,1);                   /*Compute g=(|X|-
XN*C1)-XN*C2*/
compute_R:      F10=F8*F8;                     /*Compute z=g*g*/
   F7=F10*F0, F0=mem(i_reg,1);                 /*Compute p1*z*/
   F7=F7+F0, F0=mem(i_reg,1);              /*Compute p1*z + p0*/
   F7=F8*F7;                               /*Compute g*P(z) =
(p1*z+p0)*g*/
   F12=F0*F10, F0=mem(i_reg,1);                /*Compute q2*z*/
   F12=F0+F12, F8=mem(i_reg,1);                /*Compute q2*z +
q1*/
   F12=F10*F12;                            /*Compute (q2*z+q1)*z)*/
   F12=F8+F12;                             /*Compute
Q(z)=(q2*z+q1)*z+q0*/
   F12=F12-F7, F10=mem(i_reg,1);           /*Compute Q(z) - g*P(z)*/
   F0=RECIPS F12;                              /*Get 4 bit seed
R0=1/D*/
   F12=F0*F12;                             /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                       /*F0=R1=2-D(prime),
F7=N*R0*/
   F12=F0*F12;                             /
*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                       /*F7=N*R0*R1,
F0=R2=2-D(prime)*/
   F12=F0*F12;                             /
*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                       /*F7=N*R0*R1*R2,
F0=R3=2-D(prime)*/
   F7=F0*F7;                               /*F7=N*R0*R1*R2*R3*/
   F7=F7+F8;                               /*R(g) = .5 +
                                               (g*P(z))/(Q(z)-
g*P(z))*/
   R4=FIX F4;                              /*Get N in fixed point
again*/
   RTS (DB);
   R4=R4+1;
   F0=SCALB F7 BY R4;                      /*R(g) * 2^(N+1)*/
.ENDSEG;

.SEGMENT/SPACE  Assembly_Library_Data_Space;
.PRECISION=MEMORY_PRECISION;
.VAR exponential_data[12] =   88.0,        /*BIGX */
   -88.0,                                  /*SMALLX*/
   0.000000001,                            /*eps*/
   1.4426950408889634074,                  /*1/ln(2.0)*/
   0.693359375,                            /*C1*/
   -2.121944400546905827 7E-4,             /*C2*/
   0.59504254977591E-2,                        /*P1*/
   0.24999999999992,                       /*P0*/
   0.29729363682238E-3,                        /*Q2*/
```

```
    0.53567517645222E-1,                    /*Q1*/
    0.5,                                  /*Q0 and others*/
    2.0;                                  /*Used in divide*/
.ENDSEG;
```

Listing 2.11 Exponential Subroutine

2.8 POWER APPROXIMATION

The algorithm to approximate the power function (*pow(x,y)*, *x**y*, or x^y) is more complicated than the algorithms for the other standard floating-point functions. The power function is defined mathematically as

$$x ** y = exp(y * ln(x))$$

Unfortunately, computing pow(x,y) directly with the exponent and logarithm routines discussed in this chapter yields very inaccurate results. This is due to the finite word length of the processor. Instead, the implementation described here uses an approximation technique detailed in the [CODY]. Cody and Waite use pseudo extended-precision arithmetic to extend the effective word length of the processor to decrease the relative error of the function.

The key to pseudo extended-precision arithmetic is to represent the floating point number in the reduced form; for a floating-point number V

$$V = V_1 + V_2$$

where

$$V_1 = FLOAT(INTRND(V * 16))/16$$

and

$$V_2 = V - V_1$$

To compute the power function $Z = X^Y$, let

$$Z = 2^W$$

where

$$W = Y \log_2(X)$$

Let the input X be a positive floating-point number, and let $U = \log_2(X)$. The equation simplified is

$$W = Y * U$$

To implement this approximation accurately, you must compute Y and U to extended-precision using their reduced forms (Y_1+Y_2 and $U_1 + U_2$, respectively). U_1 and U_2 are calculated with the approximations outlined below. Y_1 and Y_2 are formed from the floating-point input Y.

To calculate U_1 and U_2, first represent the floating-point input, X, in the form

$$X = f * 2^m$$

where

$$1/2 \leq f < 1$$

The value of U_2 is determined using a rational approximation generated by Cody and Waite, and the value of U_1 is determined using the equation

$$U_1 = FLOAT(INTRND(U*16))/16$$

where p is an odd integer less than 16 such that

$$f = 2^{(-p/16)} * g/a$$

a = precalculated coefficients
$g = f * 2^r = f$ (in the case of non-decimal processors $r=0$)

The reduced form of W is derived from the values of U_1, U_2, Y_1, and Y_2. Since

$$W1 = m' - p'/16$$

$$Z = 2^{m'} * 2^{(-p'/16)} * 2^{W2}$$

where 2^{W2} is evaluated by means of another rational approximation.

2.8.1 Implementation

`POW.ASM` is an ADSP-21000 implementation of the power algorithm. When assembling the file `POW.ASM`, you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-Didentifier` switch at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA pow
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA pow
```

The power function approximation subroutine expects inputs, *X* and *Y*, to be floating-point values.

The first step of the subroutine is to check that *X* is non-negative. If *X* is zero or less than zero, the subroutine terminates and returns either the value zero or the value of *X*, depending on the conditions.

The second step is to calculate *f* such that

$$X = f * 2^m$$

Use the `LOGB` function to recover the exponent of X. Since the floating point format of the ADSP-210xx has a hidden scaling factor (see Appendix D, "Numeric Formats," in The *ADSP-21020 User's Manual* for details) you must add a one to the exponent. This new exponent, *m*, scales *X* to determine the value of f such that $1/2 \le f < 1$.

$$f = X * 2^{-m}$$

Note for these calculations, the value of *g* is equal to the value of *f*.

The value of *p* is determined using a binary search and an array of floating point numbers A_1 and A_2 such that sums of appropriate array elements represent odd integer powers of $2^{-1/16}$ to beyond working precision.

set $p = 1$
if $(g \le A_1(9))$, then p = 9
if $(g \le A_1(p+4))$, then $p = p + 4$
if $(g \le A_1(p+2))$, then $p = p + 2$

Next, you must determine the values of U_1 and U_2. To determine U_2, you must implement a rational approximation. The equation for U_2 is

$$U_2 = (R + z * K) + z$$

where K is a constant and z and $R(z)$ are determined as described below.

To determine the value of z, the following equation is used

$$z' = 2 * [g - A_1(p+1)] - A_2 ((p+1)/2)$$
$$z = z' + z'$$

At this point, $|z| \leq 0.044$.

To determine the value of $R(z)$, Cody and Waite derived coefficients (p_1, p_2) especially for this approximation. The equation is

$$R(z) = [(p2 * v) + p1] * v * z$$

where

$$v = z * z.$$

To determine the value of U_1

$$U_1 = REDUCE(U)$$
$$U_1 = FLOAT(INTRND(U * 16))/16$$

since

$$U = log_2(X)$$
$$= log_2(f * 2^m)$$
$$= log_2([2^{(-p/16)} * g/a] * 2^m)$$
$$= m - p/16$$

Therefore

$$U_1 = FLOAT(INTRND(16*m-p)) * 0.0625$$

Having calculated U_1 and U_2, reduce Y into Y_1 and Y_2:

$$Y_1 = REDUCE(Y)$$

$Y_2 = Y - Y_1$

and then calculate the value of W using the pseudo extended-precision product of U and Y with the following sequence of operations:

$W = U_2 * Y + U_1 * Y_2$
$W_1 = REDUCE(W)$
$W_2 = W - W1$
$W = W_1 + U_1 * Y_1$
$W_1 = REDUCE(W)$
$W_2 = W_2 + (W - W_1)$
$W = REDUCE(W_2)$
$IW_1 = INT(16 * (W_1 + W))$
$W_2 = W_2 - W$

Now compare IW_1 with the largest and smallest positive finite floating-point numbers to test for overflow. If an overflow occurs, the subroutine ends and an error value should be set.

For the next step IW_2 must be less than or equal to zero. If $W_2 > 0$, add one to IW_1 and subtract 1/16 from W_2.

Determine that values of m' and p' using the equations:

$m' = IW_1/16 + 1$
$p' = 16 * m' - IW_1$

You can now determine the value of Z

$Z = 2^{m'} * 2^{(-p'/16)} * 2^{W2}$

The value of $2^{W2} - 1$ is evaluated for $-0.0625 \le W_2 \le 0$ using a near-minimax polynomial approximation developed by Cody and Waite.

$Z = W_2 * Q(W_2)$

where $Q(W_2)$ is a polynomial in W_2 with coefficients q_1 through q_5. Therefore

$Z = ((((q_5 * W_2 + q_4) * W_2 + q_3) * W_2 + q_2) * W_2 + q_1) * W_2$

Now, add 1 to Z and multiply by $2^{(-p'/16)}$ using the equation

$$Z = (Z * A_1(p'+1)) + A_1(p1+1)$$

Finally, scale Z by the value of m' or $Z = ADX(Z, m')$

2.8.2 Code Listings

2.8.2.1 Power Subroutine

```
/*************************************************************
File Name
   POW.ASM

Version
   Version 0.04    7/6/90

Purpose
Subroutine to compute x raise to the y power of its two floating point
   inputs.
Equations Implemented
   Y=POW(X)

Calling Parameters
   F0 = X Input Value
   F1 = Y Input Value
   l_reg = 0
Return Values
   F0 = Exponential of input

Registers Affected
   F0, F1, F2, F4, F5, F7,
   F8, F9, F10, F11, F12, F13, F14, F15
   i_reg, ms_reg
Computation Time
   37 Cycles

# PM locations
   125 words

#DM locations
   33 words (could be placed in PM instead)
*************************************************************/
 #include “asm_glob.h”
 #include “pow.h”
 #define b_reg   B3
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
#define i_reg   I3
#define l_reg   L3
#define mem(i,m) DM(i,m)
#define m1_reg  M7
#define mm_reg  M6
#define ms_reg  M5
#define SPACE   DM


.SEGMENT/PM   rst_svc;
         jump pow;
.ENDSEG;

.SEGMENT/PM     pm_sram;

.PRECISION=MACHINE_PRECISION;

.GLOBAL     pow;


pow:             F0=PASS F0;                        /*Test for x<=0*/
         IF LT JUMP x_neg_error;                   /*Report error*/
         IF EQ JUMP check_y;                       /*Test y input*/
determine_m:     R2=LOGB F0;                        /*Get exponent of x input*/
         R2=R2+1;                                  /*Reduce to proper format*/
         R3=-R2;                                   /*Used to produce f*/
determine_g:     F15=SCALB F0 BY R3;                /* .5 <= g < 1*/
determine_p:     i_reg=a1_values;
         R14=1;
         R13=9;
         R10=i_reg;
         F12=mem(9,i_reg);                         /*Get A1(9)*/
         COMP(F12,F15), F12=mem(5,i_reg);          /*A1(9) - g*/
         F11=mem(13,i_reg);
         IF GE F12=F11;                            /*Use A(13) next*/
         IF GE R14=R13;                            /*IF (g<=A1(9)) p=9*/
         R9=R10+R14;
         i_reg=R9;
```

```
          R13=4;
          COMP(F12,F15), F12=mem(2,i_reg);      /*A1(p+4) - g*/
          F11=mem(6,i_reg);
          IF GE F12=F11;
          IF GE R14=R14+R13;                    /*IF (g<=A1(p+4)) p=p+4*/
          R13=2;
          COMP(F12,F15);                          /*A1(p+2) - g*/
          IF GE R14=R14+R13;                   /*IF (g<=A1(p+4)) p=p+2*/
 determine_z:    R14=R14+1, R4=R14;
          ms_reg=R14;
          i_reg=a1_values;
          R11=ASHIFT R14 BY -1;                /*Compute (p+1)/2*/
          F12=mem(ms_reg,i_reg);               /*Fetch A1(p+1)*/
          ms_reg=R11;
          i_reg=a2_values;                     /*Correction array*/
          F0=F12+F15;                          /*g + A1(p+1)*/
          F14=F15-F12, F11=mem(ms_reg,i_reg);
          F7=F14-F11, F12=F0;                  /*[g-A1(p+1)]-A2((p+1)/2)*/
          F11=2.0;

 __divide_:      F0=RECIPS F12;                 /*Get 4 bit seed R0=1/D*/
          F12=F0*F12;                          /*D(prime) = D*R0*/
          F7=F0*F7, F0=F11-F12;                /*F0=R1=2-D(prime), F7=N*R0*/
          F12=F0*F12;                          /*F12=D(prime)=D(prime)*R1*/
          F7=F0*F7, F0=F11-F12;                /*F7=N*R0*R1, F0=R2=2-D(prime)*/
          F12=F0*F12;                          /*F12=D(prime)=D(prime)*R2*/
          F7=F0*F7, F0=F11-F12;                   /*F7=N*R0*R1*R2, F0=R3=2-
D(prime)*/
          F7=F0*F7;                            /*F7=N*R0*R1*R2*R3*/

          F7=F7+F7;                          /* z = z + z*/
 determine_R:    i_reg=power_array;
          F8=F7*F7, F9=mem(p2,i_reg);             /* v = z * z */
          F10=F8*F9, F9=mem(p1,i_reg);         /* p2*v */
          F10=F10+F9;                          /* p2*v + p1 */
          F10=F10*F8;                          /* (p2*v + p1) * v */
          F10=F10*F7, F9=mem(K,i_reg);         /* R(z) = (p2*v+p1)*v*z */
 determine_u2:   F11=F10*F9;                    /* K*R */
          F11=F10+F11;                         /* K + K*R */
          F9=F9*F7;                            /* z*K */
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
         F9=F9+F11;                          /* R + z*K */
         F9=F9+F7;                           /* (R + z*K) + z*/
determine_u1:   R3=16;
         R2=R2*R3 (SSI);                     /* m*16 */
         R2=R2-R4;                           /* m*16-p */
         R3=-4;
         F2=FLOAT R2 BY R3;                  /*FLOAT(m*16-p)*.0625*/
determine_w:    R4=4;                        /*Used in reduce*/
         R8=FIX F1 BY R4;
         F8=FLOAT R8 BY R3;                  /* y1=REDUCE(Y) */
         F7=F1-F8;                           /* y2 = y - y1 */
         F15=F9*F1;                          /* U2*Y */
         F14=F2*F7;                          /* U1*Y2 */
         F15=F14+F15;                        /* W = U2*Y + U1*Y2*/
         R14=FIX F15 BY R4;
         F14=FLOAT R14 BY R3;                /* W1=REDUCE(W) */
         F13=F15-F14;                        /* W2 = W - W1 */
         F12=F2*F8;                          /* U1*Y1 */
         F12=F14+F12;                        /* W = W1 + U1*Y1 */
         R14=FIX F12 BY R4;
         F14=FLOAT R14 BY R3;                /* W1 = REDUCE(W) */
         F11=F12-F14;                        /* W-W1 */
         F13=F11+F13;                        /* W2 = W2 + (W-W1) */
         R12=FIX F13 BY R4;
         F12=FLOAT R12 BY R3;                /* W = REDUCE(W2) */
         F10=F12+F14;
         R10=FIX F10 BY R4;                  /* IW1 = INT(16*(W1+W)) */
         F13=F13-F12, R9=mem(bigx,i_reg);    /* W2 = W2 - W */
         COMP(R10,R9), R9=mem(smallx,i_reg); /* Test for overflow */
         IF GE JUMP overflow
         COMP(R10,R9);
         IF LE JUMP underflow;
flow_to_a:      F13=PASS F13;                /* W2 must be <=0 */
         IF LE JUMP determine_mp;
         F8=.0625;
         F13=F13-F8;
         R10=R10+1;
determine_mp:   R8=1;
         R10=PASS R10;
         IF LT R8=R8-R8;                     /* I=0 if IWI < 0 */
         R6=ABS R10;                         /* Take ABS for shift*/
         R7=ASHIFT R6 BY -4;                 /* IW1/16 */
         R6=PASS R10;
         IF LT R7=-R7;
         R7=R7+R8;                          /* m(prime) = IW1/16 + I */
         R6=ASHIFT R7 BY 4;                  /* m(prime)*16 */
         R6=R6-R10, F5=mem(q5,i_reg);        /* p(prime)  = 16*m(prime) - IW1 */
determine_Z:    F4=F5*F13, F5=mem(q4,i_reg);     /* q5*W2 */
         F4=F4+F5, F5=mem(q3,i_reg);             /* q5*W2 + q4 */
```

```
          F4=F4*F13;                          /* (q5*W2+q4)*W2 */
          F4=F4+F5, F5=mem(q2,i_reg);                 /* (q5*W2+q4)*W2+q3 */
          F4=F4*F13;                          /* ((q5*W2+q4)*W2+q3)*W2 */
          F4=F4+F5, F5=mem(q1,i_reg);                 /* ((q5*W2+q4)*W2+q3)*W2+q2 */
          F4=F4*F13;                          /* (((q5*W2+q4)*W2+q3)*W2+q2)*W2 */
          F4=F4+F5;                           /*
Q(W2)=(((q5*W2+q4)*W2+q3)*W2+q2)*W2+q1*/
          F4=F4*F13;                          /* Z = W2*Q(W2) */
          i_reg=a1_values;
          R6=R6+1;                            /* Compute p(prime)+1 */
          ms_reg=R6;
          F5=mem(ms_reg,i_reg);                       /* Fetch A1(p(prime)+1) */
          F4=F4*F5;                           /* A1(p(prime)+1)*Z */
          F4=F4+F5;                           /* Z = A1(p(prime)+1) +
A1(p(prime)+1)*Z */
          F0=SCALB F4 BY R7;                  /* Result = ADX(Z,m(prime)) */
 underflow:       F0=0;                       /*Set error also*/
          RTS;

 x_neg_error:     F0=0;                       /*Set an error also*/
          RTS;

 check_y:         F1=PASS F1, F0=F0;          /*If y>0 return x*/
          IF GT RTS;
 overflow:        RTS;                        /*Set an error also*/
```

```
.ENDSEG;

.SEGMENT/DM  dm_sram;

.VAR    a1_values[18] = 0,
                0x3F800000,
                0x3F75257D,
                0x3F6AC0C6,
                0x3F60CCDE,
                0x3F5744FC,
                0x3F4E248C,
                0x3F45672A ,
                0x3F3D08A3 ,
                0x3F3504F3 ,
                0x3F2D583E ,
                0x3F25FED6 ,
                0x3F1EF532 ,
                0x3F1837F0 ,
                0x3F11C3D3 ,
                0x3F0B95C1 ,
                0x3F05AAC3 ,
                0x3F000000;

.VAR    a2_values[9] =  0,
                0x31A92436,
                0x336C2A95,
                0x31A8FC24,
                0x331F580C,
                0x336A42A1,
                0x32C12342,
                0x32E75624,
                0x32CF9891;

.VAR    power_array[10]= 0.833333286245E-1,      /* p1     */
                         0.125064850052E-1,      /* p2     */
                         0.693147180556341,      /* q1     */
                         0.240226506144710,      /* q2     */
                         0.555040488130765E-1,   /* q3     */
```

```
                        0.961620659583789E-2,   /* q4     */
                        0.130525515942810E-2,   /* q5     */
                        0.44269504088896340736, /* K      */
                     2032.,                     /* bigx   */
                    -2015;                      /* smallx */
    .ENDSEG;
```

Listing 2.12 pow.asm

2.8.2.2 Global Header File

```
#define MACHINE_PRECISION   40
 #define MEMORY_PRECISION    32

 #ifdef PM_DATA

 #define b_reg   B11
 #define i_reg   I11
 #define l_reg   L11
 #define mem(m,i) PM(m,i)
 #define m1_reg  M14
 #define mm_reg  M13
 #define ms_reg  M12
 #define SPACE   PM
 #define Assembly_Library_Data_Space Lib_PMD

 #endif

 #ifdef DM_DATA

 #define b_reg   B3
 #define i_reg   I3
 #define l_reg   L3
 #define mem(i,m) DM(i,m)
 #define m1_reg  M7
 #define mm_reg  M6
 #define ms_reg  M5
 #define SPACE   DM
 #define Assembly_Library_Data_Space Lib_DMD

 #endif
```

Listing 2.13 asm_glob.h

2.8.2.3 Header File

```
/*
    This include file is used by the power routine of the assembly
    library
*/

 #define p1 0
 #define p2 1
 #define q1 2
 #define q2 3
 #define q3 4
 #define q4 5
```