Function Approximation 4

4.1 OVERVIEW

Transcendental functions such as sines and logarithms are often approximated by polynomial expansions. The most widely used of these expansions are the Taylor and McLaren series. They can be used to approximate almost any function whose derivative is defined over the specified input range. The ADSP-2100 routines in this chapter produce function approximations from polynomial expansions, except for random number generation, which is accomplished using the linear congruence method.

Because the ADSP-2100 performs single precision (16-bit) fixed-point operations, the accuracy of a polynomial expansion approximation decreases as the order of the polynomial increases. In order to achieve accuracy in a polynomial expansion of a limited order, we have provided optimized coefficients for the polynomials used in the function approximations in this chapter. Most of these coefficients were calculated using the statistical analysis technique of regression. The coefficients are given in the fixed-point hexadecimal format that allows the maximum precision for the necessary magnitude.

In the interests of simplicity and accuracy, some formulas used in this chapter are valid for a limited range of input. These routines employ the properties of the particular function to scale or offset the input value to a value within the valid range and thereby expand the range to accommodate virtually any input.

4.2 SINE APPROXIMATION

The following formula approximates the sine of the input variable x:

$$\sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$$

The approximation is accurate for any value of x from 0° to 90° (the first quadrant). However, because $\sin(-x) = -\sin(x)$ and $\sin(x) = \sin(180^\circ - x)$, you can infer the sine of any angle from the sine of an angle in the first quadrant.

4 Function Approximation

The routine that implements this sine approximation, accurate to within two LSBs, is shown in Listing 4.1. This routine accepts input values in 1.15 format. The coefficients, which are initialized in data memory in 4.12 format, have been adjusted to reflect an input value scaled to the maximum range allowed by this format. On this scale, 180˚ equals the maximum positive value, H#7FFF, and –180˚ equals the maximum negative value, H#8000, as shown in Figure 4.1.

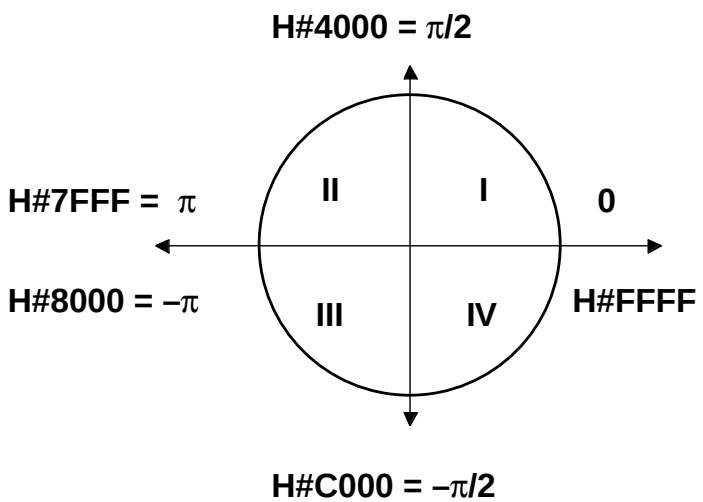

Figure 4.1 Scaled Angle Values

The routine shown in Listing 4.1 reads the scaled input angle from AX0. This angle is first modified to generate the angle in the first quadrant that will yield the same sine (or negative sine). If the input is in the second or fourth quadrant (bit 14 of the input value is a one) the input is negated to produce the twos complement, which represents an angle in the third or first quadrant, respectively. The sign bit of this angle is cleared to produce an angle in the first quadrant, and this result is stored in AR.

If the original angle is in the first quadrant, its value is unchanged. If it is in the second quadrant, negation changes it to the third quadrant, and the sign bit removal changes it to the first quadrant. If the original angle is in the third quadrant, the removal of the sign bit changes it to the first quadrant. An angle that is originally in the fourth quadrant is changed to the first quadrant by negation.

Function Approximation 4

The sine of the modified angle is calculated by multiplying increasing powers of the angle by the appropriate coefficients. The square of the angle is computed and stored in MF while the first coefficient is fetched from data memory. The first term of the sine approximation is stored in the MR registers (in which the result is subsequently accumulated) in parallel with the second coefficient fetch. In the *approx* loop, the next term of the approximation is computed and added to the partial result in MR; then a multifunction instruction fetches the next coefficient and generates the next power of the angle at the same time.

Because the coefficients are in 4.12 format, a shift instruction is needed to scale the result to 1.15 format. The result is then checked for overflow. If the value in SR1 exceeds H#7FFF, the routine saturates the result at the maximum positive value, H#7FFF, which is read from AY0. Then the sign of the result is restored, if necessary. If the input angle (stored in AX0) is negative, the result must be negated.

```
.MODULE Sin_Approximation;

{
   Sine Approximation
        Y = Sin(x)

   Calling Parameters
        AX0 = x in scaled 1.15 format
        M3 = 1
        L3 = 0

   Return Values
        AR = y in 1.15 format

   Altered Registers
        AY0,AF,AR,MY1,MX1,MF,MR,SR,I3

   Computation Time
        25 cycles
}
```

(listing continues on next page)

4 Function Approximation

```
.VAR/DM sin_coeff[5];

.INIT   sin_coeff : H#3240, H#0053, H#AACC, H#08B7, H#1CCE;

.ENTRY  sin;

sin:    I3=^sin_coeff;                    {Pointer to coeff. buffer}
        AY0=H#4000;
        AR=AX0, AF=AX0 AND AY0;           {Check 2nd or 4th quad.}
        IF NE AR=-AX0;                    {If yes, negate input}
        AY0=H#7FFF;
        AR=AR AND AY0;                    {Remove sign bit}
        MY1=AR;
        MF=AR*MY1 (RND), MX1=DM(I3,M3);   {MF = x^2}
        MR=MX1*MY1 (SS), MX1=DM(I3,M3);   {MR = C1x}
        CNTR=3;
        DO approx UNTIL CE;
            MR=MR+MX1*MF (SS);
approx:     MF=AR*MF (RND), MX1=DM(I3,M3);
        MR=MR+MX1*MF (SS);
        SR=ASHIFT MR1 BY 3 (HI);
        SR=SR OR LSHIFT MR0 BY 3 (LO);    {Convert to 1.15 format}
        AR=PASS SR1;
        IF LT AR=PASS AY0;                {Saturate if needed}
        AF=PASS AX0;
        IF LT AR=-AR;                     {Negate output if needed}
        RTS;
.ENDMOD;
```

Listing 4.1 Sine Approximation

4.3 ARCTANGENT APPROXIMATION

The following polynomial expansion computes the arctangent of the variable x, where $x < 1$:

$$\arctan(x) = 0.318253x + 0.003314x^2 - 0.130908x^3 + 0.068542x^4 - 0.009159x^5$$

If $x \geq 1$, the following formula can be used to derive the arctangent:

$$\arctan(x) = 0.5 - \arctan(1/x)$$

The reciprocal of x when $x \geq 1$ is a valid input for the polynomial expansion. The arctangent approximated by these equations is scaled to a range that corresponds to +90° to –90°.

Function Approximation 4

The subroutine shown in Listing 4.2 computes the arctangent of a 32-bit value to within two LSBs. It reads the input in 16.16 format from MR0 (LSW) and MR1 (MSW). The absolute value of the input is calculated and written back to the MR registers. In the section beginning at the *posi* label, the fractional part of the input number (MR0) is shifted one bit to the right to put it in 1.15 format.

If the integer part of the input (in MR1) is zero, the arctangent approximation can be calculated using the input value (in AR) in the polynomial expansion directly, and execution jumps to the *noinv* label. If the MR1 value is not zero, the input is greater than one, and the reciprocal of the input value must be calculated. The value is normalized, and a one in 16.0 format is normalized with the same SE value. Dividing the normalized input value into the shifted one generates the reciprocal of the input value in 1.15 format; this value is written to the AR register.

The input value in AR is used in the polynomial expansion calculation beginning at the *noinv* label. The square of the input value is calculated while the first coefficient is fetched from data memory. The first term of the approximation is calculated while the second coefficient is fetched from data memory. In the *approx* loop, which is executed three times, the next term of the approximation is calculated and added to the partial result in MR in parallel with the fetch of the next coefficient. Then the next power of the input value is calculated and stored in MF. After the loop execution completes, one more instruction is needed to calculate the last term of the approximation and complete the result in MR.

If the input value was less than one, the calculation is complete. If the input was greater than one, (integer part greater than zero), the result must be subtracted from 0.5. The subroutine checks the integer part of the original input, which is in AY1, and if it is not zero, the result is subtracted from 0.5 (H#4000 in 1.15 format, stored in AY0). The last step determines the sign of the result; if the input was negative (determined by the sign of the integer value in AX1), the result is negative; otherwise, the result is positive.

The result in AR is in 1.15 format. It is scaled to a range in which 180˚ is represented by the maximum positive value (H#7FFF) and –180˚ is represented by the maximum negative value (H#8000). This approximation yields angles scaled to the range from 90˚ to –90˚ (represented by 0.5 to –0.5), as shown in Figure 4.1.

4 Function Approximation

```
.MODULE Arctan_Approximation;

{
   Arctangent Approximation
        y = Arctan(x)

   Calling Parameters
        MR1 = Integer part of x
        MR0 = Fractional part of x
        M3 = 1
        L3 = 0

   Return Values
        AR = Arctan(x) in 1.15 scaled format
        (-0.5 for -90°, 0.5 for 90°)

   Altered Registers
        AX0,AX1,AY0,AY1,AR,AF,MY0,MY1,MX1,MF,MR,SR,SI

   Computation Time
        58 cycles (maximum)
}

.VAR/DM/RAM atn_coeff[5];

.INIT   atn_coeff : H#28BD, H#006D, H#EF3E, H#08C6, H#FED4;

.ENTRY  arctan;

arctan: I3 = ^atn_coeff;              {Point to coefficients}
        AY0=0;
        AX1=MR1;
        AR=PASS MR1;
        IF GE JUMP posi;              {Check for positive input}
        AR=-MR0;                      {Make negative number positive}
        MR0=AR;
        AR=AY0-MR1+C-1;
        MR1=AR;
posi:   SR=LSHIFT MR0 BY -1 (LO);     {Produce 1.15 value in SR0}
        AR=SR0;
        AY1=MR1;
        AF=PASS MR1;
        IF EQ JUMP noinv;             {If input < 1, no need to invert}
        SR=EXP MR1 (HI);              {Invert input}
        SR=NORM MR1 (HI);
        SR=SR OR NORM MR0 (LO);
        AX0=SR1;
        SI=H#0001;
        SR=NORM SI (HI);
        AY1=SR1;
        AY0=SR0;
```

```
        DIVS AY1,AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        DIVQ AX0; DIVQ AX0; DIVQ AX0;
        AR=AY0;
noinv:  MY0=AR;
        MF=AR*MY0 (RND), MY1=DM(I3,M3);
        MR=AR*MY1 (SS), MX1=DM(I3,M3);
        CNTR=3;
        DO approx UNTIL CE;
            MR=MR+MX1*MF (SS), MX1=DM(I3,M3);
approx:     MF=AR*MF (RND);
        MR=MR+MX1*MF (SS);
        AR=MR1;
        AY0=H#4000;
        AF=PASS AY1;
        IF NE AR=AY0-MR1;
        AF=PASS AX1;
        IF LT AR=-AR;
        RTS;
.ENDMOD;
```

Listing 4.2 Arctangent Approximation

4.4 SQUARE ROOT APPROXIMATION

The following equation approximates the square root of the input value x, where $0.5 \geq x \geq 1$:

$$\text{sqrt}(x) = 1.454895x - 1.34491x^2 + 1.106812x^3 - 0.536499x^4 + 0.1121216x^5 + 0.2075806$$

To determine the square root of an input value outside the range from 0.5 to 1, you must scale the value to a number within the range. After computing the square root of the scaled number, you multiply the result by the square root of the scaling value to produce the square root of the original number. The program shown in this section performs all necessary scaling.

The exponent detector of the shifter in the ADSP-2100 can be used to calculate the necessary scaling value. It determines the amount of left shifting needed to remove redundant sign bits of the input value, if any exist, and stores a number that represents the shift amount in the SE

register. Because the format of the input number is 16.16, a left shift of 15 bits (register SE = –15) indicates that the input number is already between 0.5 and 1; no scaling is needed, so the scaling value *s* is one. If the number is shifted left more than 15 bits (register SE < –15), the input number must be multiplied by a scaling value that is greater than one. If the number is shifted left fewer than 15 bits (register SE > –15), the scaling value must be less than one.

The value in SE is the negative of the power of two necessary to shift the value; therefore, the scaling value *s* is equal to 2^{SE+15}. The square root is calculated as follows:

$X = \sqrt{Y}$
$Z = \sqrt{(sY)} = \sqrt{s}\ \sqrt{Y}$
$X = Z \div \sqrt{s}$

The square root of *s* is found as follows:

$s = 2^{SE+15}$
$\sqrt{s} = \sqrt{(2^{SE+15})} = (1 \div (1 \div (\sqrt{2^{SE+15}}))) = (1 \div \sqrt{2})^{-(SE+15)}$

Incorporating the value of $\sqrt{s}$ into the equation for X yields:

$X = Z \div ((1 \div \sqrt{2})^{-(SE+15)})$

The value $(1 \div \sqrt{2})^{-(SE+15)}$, is calculated by storing the reciprocal of $\sqrt{2}$ as a 1.15 constant and multiplying this constant by itself SE+15 times, producing a result in 1.15 format. This result is the value of $\sqrt{s}$ if SE+15 is negative. If SE+15 is positive, the value of $\sqrt{s}$ is the reciprocal of the result, which is found by dividing the result into 1 (in 9.23 format) to produce a value in 9.7 format.

Listing 4.3 shows the ADSP-2100 routine to approximate the square root of *x*, where $0 \leq x < 32768$. The first part of the routine scales the input number and computes the square root of the scaled number. The constant term of the polynomial is stored as the constant *base*, which is loaded into MR to be added to the other terms as they are computed in the *approx* loop.

If the scaling value $\sqrt{s}$ is one (SE+15 = 0), the result is shifted into 8.8 *unsigned* format and returned; otherwise, $\sqrt{s}$ must be computed, beginning at the label *scale*.

Function Approximation 4

The constant *sqrt2*, which is equal to 1 ÷ √2, is loaded into MR1 and MY1. The AR register is loaded with the absolute value of SE+15. If this value is one, then MR contains the value of √s or its reciprocal; if not, the *compute* loop computes the correct power of 1 ÷ √2.

The section that begins at the *pwr_ok* label checks the sign of SE+15. If it is negative, MR contains the correct value of √s. This value is multiplied by the square root approximation of the scaled input, which was stored in MY0. The product is shifted right six bits to put it in 8.8 *unsigned* format and returned. If SE+15 is positive, the reciprocal of the value in MR is calculated to yield the correct value of √s. The product of √s and the square root approximation in MY0 is calculated and added to H#2000 in MR0 to round the low order bits. This result is shifted right two bits to form an *unsigned* result in 8.8 format.

```
.MODULE Square_root;

{
   Square Root
        y = √x

   Calling Parameters
        MR1 = MSW of x (16.0 portion)
        MR0 = LSW of x (0.16 portion)
        M3 = 1
        L3 = 0

   Return Values
        SR1 = y in 8.8 UNSIGNED format

   Altered Registers
        AX0,AY0,AY1,AR,AF,MY0,MY1,MX0,MF,MR,SE,SR,I3

   Computation Time
        75 cycles (maximum)
}

.CONST  base=H#0D49, sqrt2=H#5A82;

.VAR/DM sqrt_coeff[5];

.INIT   sqrt_coeff : H#5D1D, H#A9ED, H#46D6, H#DDAA, H#072D;

.ENTRY  sqrt;
```

(listing continues on next page)

```
sqrt:   I3=^sqrt_coeff;                          {Pointer to coeff. buffer}
        SE=EXP MR1 (HI);                         {Check for redundant bits}
        SE=EXP MR0 (LO);
        AX0=SE, SR=NORM MR1 (HI);                {Remove redundant bits}
        SR=SR OR NORM MR0 (LO);
        MY0=SR1, AR=PASS SR1;
        IF EQ RTS;
        MR=0;
        MR1=base;                                {Load constant value}
        MF=AR*MY0 (RND), MX0=DM(I3,M3);          {MF = x^2}
        MR=MR+MX0*MY0 (SS), MX0=DM(I3,M3);       {MR = base + C1x}
        CNTR=4;
        DO approx UNTIL CE;
            MR=MR+MX0*MF (SS), MX0=DM(I3,M3);
approx:     MF=AR*MF (RND);
        AY0=15;
        MY0=MR1, AR=AX0+AY0;                     {SE + 15 = 0?}
        IF NE JUMP scale;                        {No, compute √s}
        SR=ASHIFT MR1 BY -6 (HI);
        RTS;
scale:  MR=0;
        MR1=sqrt2;                               {Load 1÷√2}
        MY1=MR1, AR=ABS AR;
        AY0=AR;
        AR=AY0-1;
        IF EQ JUMP pwr_ok;
        CNTR=AR;                                 {Compute (1÷√2)^(SE+15)}
        DO compute UNTIL CE;
compute:    MR=MR1*MY1 (RND);
pwr_ok: IF NEG JUMP frac;
        AY1=H#0080;                              {Load a 1 in 9.23 format}
        AY0=0;                                   {Compute reciprocal of MR}
        DIVS AY1, MR1;
        DIVQ MR1; DIVQ MR1; DIVQ MR1;
        DIVQ MR1; DIVQ MR1; DIVQ MR1;
        DIVQ MR1; DIVQ MR1; DIVQ MR1;
        DIVQ MR1; DIVQ MR1; DIVQ MR1;
        DIVQ MR1; DIVQ MR1; DIVQ MR1;
        MX0=AY0;
        MR=0;
        MR0=H#2000;
        MR=MR+MX0*MY0 (US);
        SR=ASHIFT MR1 BY 2 (HI);
        SR=SR OR LSHIFT MR0 BY 2 (LO);
        RTS;
frac:   MR=MR1*MY0 (RND);

        SR=ASHIFT MR1 BY -6 (HI);
        RTS;
.ENDMOD;
```

Listing 4.3 Square Root Approximation

Function Approximation 4

4.5 LOGARITHM APPROXIMATION

The common logarithm (base ten) of any number x between one and two can be approximated using the following equation:

$$2\log_{10}(x) = 0.8678284(x-1) - 0.4255677(x-1)^2 + 0.2481384(x-1)^3 - 0.1155701(x-1)^4 + 0.0272522(x-1)^5$$

The natural logarithm (base e) of any number x between one and two can be approximated using the following equation:

$$\ln_e(x) = 0.9991150(x-1) - 0.4899597(x-1)^2 + 0.2856751(x-1)^3 - 0.1330566(x-1)^4 + 0.03137207(x-1)^5$$

To calculate the logarithm of a number greater than two or less than one using these formulas, you must scale the value to a number within the valid input range. The exponent detector of the shifter in the ADSP-2100 can be used to scale the input number. It determines the amount of left shifting needed to remove redundant sign bits of the input value, if any exist, and stores a number that represents the shift amount in the SE register. Because the format of the input number is 16.16, a left shift of 14 bits indicates that the input number is already between one and two; no scaling is needed. If the shift is more than 14 bits, the input must be scaled by a value greater than one. If the shift is fewer than 14 bits, the input must be scaled by a value less than one. The exponent will shift the number into the range $0.5 < x < 1$, so the number must be shifted to the left one bit more to place it in the range $1 < x < 2$.

The logarithm of the scaled input must be adjusted to produce the logarithm of the original unscaled input. The adjustment is determined as follows:

$$Y = \log(X)$$
$$Z = \log(sX)$$
$$\log(sX) = \log(s) + \log(X)$$
$$Y = Z - \log(s)$$

Therefore, the logarithm of the unscaled input is equal to the logarithm of the scaled input less the logarithm of the scaling factor s. Computation of $\log(s)$ is simplified by the fact that s is a power of two.

$$s = 2^{SE+14}$$
$$\log(s) = \log(2^{SE+14}) = (SE+14)\log(2)$$

4 Function Approximation

Computation of the $\log_{10}$ and $\ln_e$ can be accomplished using a single routine initialized with one of two sets of coefficients (for either $\log_{10}$ or $\ln_e$). The routine shown in Listing 4.4 has two entry points: *ln*, to compute the natural log, and *log*, to compute the common log. The entry point sets I3 to the start of the appropriate coefficient buffer and MY1 to either ln(2) or $\log_{10}(2)$, as appropriate. This routine yields results accurate to within two LSBs.

At the *compute* label, the input number is adjusted to the range from 0.5 to 1 by the exponent detector. Because the negative of the value in SE is used by the NORM command, the value in SE is decremented by one, so that the number is normalized to the range from one to two in unsigned 1.15 format.

The value of SE+14 is stored in AR. After the input number is normalized, ln(2) or $\log_{10}(2)$ (stored in MY1) is multiplied by SE+14 in AR. The value in AR is in 16.0 format and the value in MY1 is in either 1.15 format (common log) or 2.14 format (natural log). The product is stored in MR and shifted left 16 bits to match the format of the terms to be accumulated in the MR register during the approximation.

In the natural log computation, ln(2) is in 2.14 format. When it is multiplied by SE+14, a 25.15 formatted number is produced. Shifting this value left 16 bits yields a number in MR in 9.31 format. Each term of the approximation is produced in 1.31 format and added to the MR value. The final result in MR is shifted left 12 bits (right four bits) to place it in 5.11 format in SR1.

```
.MODULE Logarithms;

{
   Logarithm Approximations
        y=log10(x)
        y=lne(x)

   Calling Parameters
        MR1 = Integer Portion of x in 16.0 twos complement
        MR0 = Fractional Portion of x in 0.16 unsigned
        M3 = 1
        L3 = 0

   Return Values
        SR1 = Y (4.12 format for log; 5.11 format for ln)
```

```
   Altered Registers
        AX0,AY0,AR,MY1,MX1,MF,MR,SE,SR,I3

   Computation Time
        33 cycles (maximum)
}

.CONST  log_2=H#2688,ln_2=H#2C5D;

.VAR/DM ln_coeffs[5];
.VAR/DM log_coeffs[5];
.INIT   ln_coeffs : H#7FE3, H#C149, H#2491, H#EEF8, H#0404;
.INIT   log_coeffs : H#6F15, H#C987, H#1FC3, H#F135, H#037D;

.ENTRY  log,ln;

ln:     MY1=ln_2;                               {Natural log start here}
        I3=^ln_coeffs;
        JUMP compute;
log:    I3=^log_coeffs;                         {Common log start here}
        MY1=log_2;
compute:SE=EXP MR1 (HI);                        {Check for redundant bits}
        SE=EXP MR0 (LO);
        AY0=SE;
        AR=AY0-1;
        AX0=14;
        SE=AR, AR=AX0+AY0;
        SR=NORM MR1 (HI);                       {Remove redundant bits}
        SR=SR OR NORM MR0 (LO);
        MR=AR*MY1 (SS);                         {(SE+14) × log(2)}
        MY1=MR1;                                {Shift left 16 bits}
        MR1=MR0;
        MR2=MY1;
        MR0=0;
        AY0=H#8000;
        AR=SR1-AY0;
        MY1=AR;
        MF=AR*MY1 (RND), MX1=DM(I3,M3);         {MF = x^2}
        MR=MR+MX1*MY1 (SS), MX1=DM(I3,M3);      {MR = C1x}
        CNTR=3;
        DO approx UNTIL CE;
            MR=MR+MX1*MF (SS);
approx:     MF=AR*MF (RND), MX1=DM(I3,M3);
        MR=MR+MX1*MF (RND);
        SR=ASHIFT MR2 BY 12 (HI);               {Shift to correct format}
        SR=SR OR LSHIFT MR1 BY 12 (LO);
        RTS;
.ENDMOD;
```

Listing 4.4 Logarithm Approximation

4 Function Approximation

4.6 UNIFORM RANDOM NUMBER GENERATION

Although the generation of a random number is not, strictly speaking, a function, it is a useful operation for many applications. One such application is in high-speed modems, in which it can be used as a training signal for the adaptive equalizer (see Chapter 13). The means for generating random numbers on a digital computer, of course, is by the computation of a function that approximates the random number. Many of such functions have been proposed (Knuth, 1969). The implementation presented here is based on the linear congruence method, which uses the following equation.

$$x(n+1) = (ax(n) + c) \bmod m$$

The initial value of x, x(0), is called the seed value and is generally not important, because with a good choice of a and c all m values are generated before the output sequence repeats. The random number sequence produced by the above equation is thus uniform in the sense that the output is uniformly distributed between 0 and m–1. Of course, different seed values should be used at different times if different sequences are desired. By choosing the modulus $m=2^{32}$, we ensure a long sequence and have a convenient modulus for the ADSP-2100. The values of a and c that are used in the following program (a=1664525 and c=32767) were chosen according to the rules in Knuth, 1969.

Listing 4.5 (on the next page) shows the ADSP-2100 routine used to compute random numbers based on the linear congruence method. The first number produced by this routine is the initial seed value in SR1. Note that, although only the most significant 16 bits of the 32-bit x value are used as random numbers in this routine, any or all of the bits can be used. However, as stated in Knuth, 1969, when using a value of m equal to the word size of the machine, the least significant bits of x(n) are much less random than the most significant bits. Thus, one should always use the b most significant bits when only a b-bit random number is desired.

The routine requires 10N+4 cycles to execute, where N is the number of random numbers desired. For example, computing 2^{16} (65,536) random numbers using an 8 MHz ADSP-2100 takes 81.9 milliseconds. Computing all $m=2^{32}$ numbers in the sequence requires almost one and a half hours.

```
.MODULE urand_sub;

{
   Linear Congruence Uniform Random Number Generator

   Calling Parameters
        I0 —> Output buffer                        L0 = 0
        M0 = 1
        SR1 = MSW of seed value
        SR0 = LSW of seed value
        CNTR = desired number of random numbers

   Return Values
        Desired number of random numbers in output buffer
        SR1 = MSW of updated seed value
        SR0 = LSW of updated seed value

   Altered Registers
        MY0,MY1,MR,SI,SR

   Computation Time
        10 × N + 4 cycles
}

.ENTRY  urand;

urand:  MY1=25;                                    {Upper half of a}
        MY0=26125;                                 {Lower half of a}
        DO randloop UNTIL CE;
            DM(I0,M0)=SR1, MR=SR0*MY1(UU);         {a(hi) × x(lo)}
            MR=MR+SR1*MY0(UU);                     {a(hi) × x(lo) + a(lo) × x(hi)}
            SI=MR1;
            MR1=MR0;
            MR2=SI;
            MR0=H#FFFE;                            {c=32767, left-shifted by 1}
            MR=MR+SR0*MY0(UU);                     {(above) + a(lo) × x(lo) + c}
            SR=ASHIFT MR2 BY 15 (HI);
            SR=SR OR LSHIFT MR1 BY -1 (HI);        {right-shift by 1}
randloop:   SR=SR OR LSHIFT MR0 BY -1 (LO);
        RTS;
.ENDMOD;
```

Listing 4.5 Random Number Generator

4 Function Approximation

4.7 REFERENCES

Burrington, R.S. 1973. *Handbook of Mathematical Tables and Formulas.* Fifth Edition. New York: McGraw-Hill Book Company.

Knuth, D. E. 1969. *The Art of Computer Programming: Volume 2 / Seminumerical Algorithms*. Second Edition. Reading, MA: Addison-Wesley Publishing Company.