Floating-Point Arithmetic 3

3.1 OVERVIEW

In fixed-point number representation, the radix point is always at the same location. While this convention simplifies numeric operations and conserves memory, it places a limit on the magnitude and the precision of the number representation. In situations that require a large range of numbers or high resolution, a relocatable radix point is desirable. Very large and very small numbers can be represented in floating-point format.

Floating-point format is scientific notation; a floating-point number consists of a mantissa and an exponent. Each part of the floating-point number is stored in a fixed-point format. The mantissa is usually in a full fractional format, and the exponent is in a full integer format (see Chapter 2 for a discussion of fixed-point formats). In some cases, a constant (excess code or bias) is added to the exponent so that it is always positive.

A floating-point number is "normalized" if it contains no redundant sign bits; all bits are significant. Normalization provides the highest precision for the number of bits available. It also simplifies the comparison of magnitudes, because the number with the greater exponent has the greater magnitude; only if the exponents are equal is it necessary to compare the fractions. Most routines (and all the routines presented in this chapter) assume normalized input and produce normalized results.

Floating-point numbers are inherently inexact because each number has multiple representations that differ only in precision. This fact introduces error into floating-point calculations (relative to the exact result). Floating-point multiplication and division do not magnify this error much, but addition or subtraction can cause significant increases in the error. Therefore, the associative law does not always hold for floating-point calculations. For an excellent discussion of floating-point accuracy see Knuth, 1969.

The routines in this chapter demonstrate ways of performing standard mathematical operations on floating-point numbers using the ADSP-2100. Because floating-point numbers can be stored in a variety of formats, each example assumes that the input numbers are converted to a standard

3 Floating-Point Arithmetic

format before the routine is called. The standard format, called "two-word" format, provides one word (16 bits) for the exponent and one word for the fraction. Signed twos-complement notation is assumed for both the fraction and the exponent. The exponent has the option of an excess code; if the excess code is not needed, it should be set to zero for all the routines that use it.

If additional precision is required, the fraction can be expanded. If additional range is needed, the exponent can be expanded. Expanding either the fraction or the exponent can be accomplished by substituting a multiprecision fixed-point arithmetic operation (see Chapter 2) for the basic operation used in the floating-point routine.

The two-word format is tailored for the ADSP-2100. It takes advantage of ADSP-2100 instructions to make calculations or conversions fast and simple. However, certain applications may require the use of IEEE 754 standard floating-point format. Details of the IEEE format can be found in the IEEE-STD-754 document, 1985. The major ways in which IEEE format differs from two-word format are:

- The number consists of a 32-bit doubleword divided into fields for (from left to right) sign bit, exponent, and fraction (mantissa)
- The exponent field is 8-bits wide (unsigned) and biased by +127
- The fraction field is 23-bits wide (unsigned)
- A "hidden bit" with a value 1 is assumed to be to the left of the fraction.

You can choose the numeric format (fixed-point or floating-point) that is better for a particular situation. In this chapter, we present routines that convert numbers from fixed-point format into floating-point format (both IEEE 754 and two-word) and vice versa. These routines are followed by examples of routines for performing basic arithmetic operations on numbers in two-word floating-point format.

3.2 FIXED-POINT TO FLOATING-POINT CONVERSION

Conversion of numbers from 1.15 fixed-point format into IEEE 754 and two-word floating-point format is discussed in this section. The corresponding floating-point to fixed-point conversions are described in the next section.

Two ADSP-2100 instructions used in fixed-point to floating-point conversion are EXP, which derives an exponent, and NORM, which normalizes (eliminates nonsignificant digits from) the mantissa.

3.2.1 Fixed-Point (1.15) to IEEE Floating-Point

Because all numbers that can be represented in 1.15 fixed-point format can also be represented in IEEE 754 floating-point format, the conversion from the fixed-point format to the floating-point format is exact. Conversion from floating-point back to fixed-point format cannot always be exact and therefore generates errors, as discussed in section 3.3.1.

The subroutine in Listing 3.1 converts a number in 1.15 format stored in AX0 into IEEE 754 floating-point format, stored in SR1 (MSW) and SR0 (LSW). The routine first checks for two special-case integer input values, –1 and 0. If the input value is either –1 or 0, the routine outputs the IEEE 754-format values explicitly. All other 1.15 numbers are fractions that can be converted to IEEE 754 format by the section that begins at *cvt*. At this point, the input has been made positive by the absolute value function, and the original sign bit has been preserved in a flag. An exponent is derived, and the number is normalized. The exponent is biased by 126 (the IEEE 754 bias of 127 minus one, because of the hidden bit). The 32-bit result is put together, using the shifter to place the output values into the correct fields.

```
.MODULE cvt_fixed_to_ieee_floating;

{  Converts 1.15 fixed-point to 32-bit IEEE 754 floating-point

   Calling Parameters
        AX0 = 1.15 fixed-point number

   Return Values
        SR1 = MSW of IEEE 754 floating-point number
        SR0 = LSW of IEEE 754 floating-point number

   Altered Registers
        AX1,AY0,AY1,AF,AR,SR,SE,SI

   Computation Time
        20 cycles (maximum)
}
.ENTRY  ieeeflt;

ieeeflt:    AY0=H#8000;               {neg. one = H#8000}
            AY1=126;                  {-127 bias + 1 for hidden bit shift
            AR=AX0-AY0;               {AY0 = -1? (H#8000)}
            IF NE JUMP numok;         {no, do conversion}
            AR=H#BF80;                {if neg one, do float right now}
            SR=LSHIFT AR BY 0 (HI);   {this is IEEE float for -1}
```

(listing continues on next page)

```
            RTS;                        {skip conversion, output right now}
numok:      AR=PASS AX0;                {1.15 number to convert is in AR}
            AF=ABS AR;                  {make positive}
            IF NE JUMP notzero;         {if not zero, do conversion}
            SR=LSHIFT AR BY 0 (HI);     {special case for zero}
            RTS;                        {exit if zero}
notzero:    IF NEG JUMP itisneg;
itispos:    SI=H#0000;                  {CLEAR sign bit flag if positive}
            JUMP cvt;
itisneg:    SI=H#8000;                  {SET sign bit flag if negative}
cvt:        AR=PASS AF;                 {use abs(orig_number) }
            SE=EXP AR (HI);             {derive exponent}
            SR=NORM AR (HI);            {normalize fraction}
            AX1=SE;                     {load AX1 with exponent}
            AR=AX1 + AY1;               {add 126 to exponent}
            SR=LSHIFT SR1 BY +2 (HI);   {remove sign & hidden bit}
            SR=LSHIFT SR1 BY -9 (HI);   {open sign bit and expo field}
            SR=SR OR LSHIFT SI BY 0 (HI);  {paste sign bit}
            SR=SR OR LSHIFT AR BY +7 (HI);{paste exponent}
            RTS;
.ENDMOD;
```

Listing 3.1 Fixed-Point to IEEE Floating-Point

3.2.2 Fixed-Point (1.15) to Two-Word Floating-Point

The routine shown in Listing 3.2 converts a number in 1.15 fixed-point format into two-word floating-point format. An exponent is derived, the number is normalized, and a bias is added to the exponent value. If no bias is needed, the routine should be called with the bias value set to zero.

```
.MODULE single_fixed_to_floating;

{
   Convert 1.15 fixed-point to two-word floating-point

   Calling Parameters
        AR = fixed point number                [1.15 signed twos complement]
        AX0 = exponent bias (0=unbiased)       [16.0 signed twos complement]

   Return Values
        AR = biased exponent                   [16.0 signed twos complement]
        SR1 = mantissa                         [1.15 signed twos complement]
```

```
   Altered Registers
        SE,SR,AY0,AR

   Computation Time
        5 cycles
}

.ENTRY  fltone;

fltone: SE=EXP AR (HI);              {Determine exponent}
        SR=NORM AR (HI);             {Remove redundant sign
bits}
        AY0=SE;
        AR=AX0+AY0;                  {Add bias}
        RTS;
.ENDMOD;
```

Listing 3.2 Fixed-Point to Two-Word Floating-Point

3.3 FLOATING-POINT TO FIXED-POINT CONVERSION

Conversion of numbers from either IEEE 754 or two-word floating-point format into 1.15 fixed-point format is discussed in this section. The corresponding fixed-point to floating-point conversions are described in the previous section.

3.3.1 IEEE Format to Fixed-Point Format (1.15)

Not all numbers that can be represented in IEEE 754 floating-point format can be represented in 1.15 fixed-point format. Therefore, the routine that converts from IEEE 754 floating-point to 1.15 fixed-point format generates an error word. The error word indicates which error condition (positive or negative, overflow or underflow) the IEEE 754 floating-point number conversion creates and also if a loss of precision occurs due to the truncation of the 23-bit mantissa to 15 bits. Truncation forces rounding toward zero, one of four possible rounding modes defined in the IEEE 754

standard. The error word protocol is listed below.

No loss of precision (8 LSBs of IEEE 754 mantissa are all zeros):

Error Condition	*Error Word*	*Result (hexadecimal)*
none	0000	1.15 conversion result
positive overflow	F000	7FFF
positive underflow	0F00	0000
negative underflow	00F0	0000
negative overflow	000F	8001

8-bit loss of precision:

Error Condition	*Error Word*	*Result (hexadecimal)*
precision loss only	FFFF	1.15 conversion result
positive overflow	0FFF	7FFF
positive underflow	F0FF	0000
negative underflow	FF0F	0000
negative overflow	FFF0	8001

The subroutine shown in Listing 3.3 converts an IEEE 754 floating-point number to 1.15 fixed-point format. The IEEE 754 floating-point MSW is read from MR1, and the LSW is read from MR0. The 1.15 fixed-point result is returned in MX1, and the error word is returned in MX0.

The routine first checks the input for the special cases of integers –1 and 0. If the input is either of these integers, the conversion is loaded into MX1 directly and returned. If the input is fractional, the routine checks the exponent field to determine whether the input number is out of the 1.15 fixed-point format range. If it is, the input is examined further to set the appropriate error word for the error condition. Then the error word is toggled if the eight LSBs of the input number are not all zeros. The conversion ignores the eight LSBs, so precision is lost if any of these bits are nonzero.

The conversion of input that can be represented in 1.15 format is done in the section beginning at the label *convert*. First, the exponent field is extracted and unbiased, and the resulting exponent is stored in the SE register. Then, the mantissa is shifted by the amount stored in the SE register. This shift results in a valid 1.15 number in the SR1 register.

Floating-Point Arithmetic 3

Finally, if the input number is negative, the result is negated.

```
.MODULE cvt_ieee_float_to_fixed;

{
   Convert 32-bit IEEE 754 floating-point to 1.15 fixed-point

   Calling Parameters
        MR1 = MSW of IEEE 754 floating point number
        MR0 = LSW of IEEE 754 floating point number

   Return Values
        MX1 = 1.15 fixed point number
        MX0 = ERROR word

   Altered Registers
        AX0,AX1,AY0,AY1,AF,AR,MX0,MX1,SR,SE

   Computation Time
        51 cycles (maximum)
}

.ENTRY  ieeefix;

ieeefix:    AY0=H#00FF;                  {mask to extract exp, lo_bit check}
            AY1=H#0100;                  {mask for overflow sign}
            MX1=H#0000;                  {clear fixed result}
            MX0=H#0000;                  {clear ERROR status}
            AX1=H#0000;                  {clear UNDER/OVERFLOW flag}

zero:       SR=LSHIFT MR1 BY 0 (LO);     {check for flt_num = 0}
            SR=SR OR LSHIFT MR0 BY 0 (LO);
            AR=PASS SR0;
            IF EQ RTS;

negone:     AF=PASS MR0;                 {check for flt_num = -1}
            IF NE JUMP checkexpo;
            AF=PASS MR1;
            AX0=H#BF80;
            AR=AX0-AF;
            IF NE JUMP checkexpo;
            MX1=H#8000;
            MX0=H#0000;
            RTS;

checkexpo:  SR=LSHIFT MR1 BY -7 (LO);    {extract exponent}
            AF=SR0 AND AY0;              {extract exponent}
            AX0=H#007E;                  {upper valid expo – 126 decimal}
            AR=AF-AX0;                   {is expo .gt. max valid ???}
            IF GT JUMP overflow;         {if YES, then overflow}
            AX0=H#0070;                  {lower valid expo – 112 decimal}
            AR=AF-AX0;                   {is expo .lt. min valid ???}
```

(listing continues on next page)

```
            IF LT JUMP underflow;          {if YES, then underflow}
            JUMP eightlsb;                 {go check if 8 LSBs are set}

overflow:   AX1=H#FFFF;                    {set UNDER/OVERFLOW FLAG true}
            AF=SR0 AND AY1;                {extract sign bit with AND mask}
            IF NE JUMP negover;
posover:    MX0=H#F000;                    {ERROR = "positive overflow"}
            MX1=H#7FFF;                    {make fixed result max. pos. value}
            JUMP eightlsb;
negover:    MX0=H#000F;                    {ERROR = "negative overflow"}
            MX1=H#8001;                    {make fixed result max. neg. value}
            JUMP eightlsb;

underflow:  AX1=H#FFFF;                    {set UNDER/OVERFLOW FLAG true}
            AF=SR0 AND AY1;                {extract sign bit with AND mask}
            IF NE JUMP negunder;
posunder:   MX0=H#0F00;                    {ERROR = "positive underflow"}
            JUMP eightlsb;                 {fixed result remains zero}
negunder:   MX0=H#00F0;                    {ERROR = "negative underflow"}
            JUMP eightlsb;                 {fixed result remains zero}

eightlsb:   SR=LSHIFT MR0 BY 0 (LO);       {get 16 LSBs of flt_num}
            AF=SR0 AND AY0;                {extract lower 8 LSBs with AND mask}
            IF EQ JUMP endlsb;
            AR=MX0;                        {if any are set, toggle ERROR}
            AR=NOT AR;
            MX0=AR;                        {ERROR value stored in MX0}
endlsb:     AR=PASS AX1;                   {check for under/overflow situation}
            IF NE RTS;                     {do not convert if under/overflow}

convert:    SR=LSHIFT MR1 BY -7 (LO);      {set up exponent field to mask}
            AF=SR0 AND AY0;                {extract exponent field by AND}
            AX0=H#007F;                    {exponent bias – 127 decimal}
            AR=AF-AX0;                     {subtract bias from exponent}
            SE=AR;                         {unbiased expo into SE register}
            SR=LSHIFT MR1 BY 8 (HI);       {paste hi mantissa word}
            SR=SR OR LSHIFT MR0 BY 8(LO); {paste lo mantissa word}
            AY1=H#8000;                    {hidden "1" bit}
            AR=SR1 OR AY1;                 {paste hidden "1" bit}
            SR=LSHIFT AR (HI);             {denormalize mantissa}
            AF=PASS MR1;                   {set sign bit if orig. was neg.}
            AR=SR1;                        {use only 16 MSBs of result}
            IF LT AR=-SR1;                 {if neg. orig. do twos complement}
            MX1=AR;
                                RTS;
                        .ENDMOD;
```

Listing 3.3 IEEE Floating-Point to Fixed-Point

3.3.2 Two-Word Format to Fixed-Point Format (1.15)

Converting two-word floating-point numbers to 1.15 fixed-point format is very simple using ADSP-2100 instructions. The exponent word is decremented by the exponent bias that has been passed to the conversion routine, and the result is stored in the SE register. The SE register determines the amount of shift performed by a shift instruction for which no immediate shift value is given. After the fractional part is shifted (arithmetically) the 1.15 fixed-point result is in the SR1 register. The conversion routine is shown in Listing 3.4. Note that this routine does not provide error handling for floating-point numbers that cannot be represented in fixed-point format.

```
.MODULE cvt_2word_float_to_fixed;

{
   Convert two-word floating-point to 1.15 fixed-point

   Calling Parameters
        AX0 = exponent              [16.0 signed twos complement]
        AY0 = exponent bias         [16.0 signed twos complement]
        SI = mantissa               [1.15 signed twos complement]

   Return Values
        SR1 = fixed-point number    [1.15 signed twos complement]

   Altered Registers
        AR,SE,SR

   Computation Time
        4 cycles
}

.ENTRY  fixone;

fixone: AR=AX0-AY0;                 {Compute unbiased exponent}
        SE=AR;
        SR=ASHIFT SI (HI);          {Shift fractional part}
        RTS;
.ENDMOD;
```

Listing 3.4 Two-Word Floating-Point to Fixed-Point

3.4 FLOATING-POINT ADDITION

The algorithm for adding two numbers in two-word floating-point format is as follows:

1. Determine which number has the larger exponent. Let's call this number X (= Ex, Fx) and the other number Y (= Ey, Fy).
2. Set the exponent of the result to Ex.
3. Shift Fy right by the difference between Ex and Ey, to align the radix points of Fx and Fy.
4. Add Fx and Fy to produce the fraction of the result.
5. Normalize the result.

Note that if the exponents are equal, the exponent of the result can be set to either, and no shifting of the fraction is necessary before the addition.

The ADSP-2100 version of the above algorithm is shown in Listing 3.5. The routine reads the exponents of the input operands from AX0 and AY0 and the corresponding fractions from AX1 and AY1. Upon return, AR holds the exponent of the result and SR1 holds the fraction. The routine first determines the operand with the largest exponent and shifts the fractional part of the other operand to equate the exponents. The fractions are added to form an unnormalized sum. This sum is fed to the exponent detector (in HIX mode to allow for overflow in the ALU) to determine the direction and magnitude of the shift required to normalize the number. The NORM instruction of the shifter uses the negative of the value in SE for the magnitude of the shift. The value in SE is then added to the exponent of the result to yield the normalized exponent.

```
.MODULE floating_point_add;

{
   Floating-Point Addition
        z = x + y

   Calling Parameters
        AX0 = Exponent of x
        AX1 = Fraction of x
        AY0 = Exponent of y
        AY1 = Fraction of y
```

```
   Return Values
        AR = Exponent of z
        SR1 = Fraction of z

   Altered Registers
        AX0,AY1,AY0,AF,AR,SI,SE,SR

   Computation Time
        11 cycles
}

.ENTRY  fpa;

fpa:    AF=AX0-AY0;                 {Is Ex > Ey?}
        IF GT JUMP shifty;          {Yes, shift y}
        SI=AX1, AR=PASS AF;         {No, shift x}
        SE=AR;
        SR=ASHIFT SI (HI);
        JUMP add;
shifty: SI=AY1, AR=-AF;
        SE=AR;
        SR=ASHIFT SI (HI), AY1=AX1;
        AY0=AX0;
add:    AR=SR1+AY1;                 {Add fractional parts}
        SE=EXP AR (HIX);
        AX0=SE, SR=NORM AR (HI);    {Normalize}
        AR=AX0+AY0;                 {Compute exponent}
        RTS;
.ENDMOD;
```

Listing 3.5 Floating-Point Addition

3.5 FLOATING-POINT SUBTRACTION

The algorithm for subtracting one number from another in two-word floating-point format is as follows:

1. Determine which number has the larger exponent. Let's call this number X (= Ex, Fx) and the other number Y (= Ey, Fy).
2. Set the exponent of the result to Ex.
3. Shift Fy right by the difference between Ex and Ey, to align the radix points of Fx and Fy.
4. Subtract the fraction of the subtrahend from the fraction of the minuend to produce the fraction of the result.
5. Normalize the result.

Note that if the exponents are equal, the exponent of the result can be set

to either, and no shifting of the fraction is necessary before the subtraction.

The ADSP-2100 version of the above algorithm is shown in Listing 3.6. The routine reads the exponents of the input operands from AX0 and AY0 and the corresponding fractions from AX1and AY1. Upon return, AR holds the exponent of the result and SR1 holds the fraction. The routine first determines the operand with the largest exponent and shifts the fractional part of the other operand to equate the exponents. The unnormalized difference of the fractions is then found. This difference is fed to the exponent detector (in HIX mode to allow for overflow in the ALU) to determine the direction and magnitude of the shift required to normalize the number. The NORM instruction of the shifter uses the negative of the value in SE for the magnitude of the shift. The value in SE is then added to the exponent of the result to yield the normalized exponent.

```
.MODULE floating_point_subtract;

{
   Floating-Point Subtraction
        z = x - y

   Calling Parameters
        AX0 = Exponent of x
        AX1 = Fraction of x
        AY0 = Exponent of y
        AY1 = Fraction of y

   Return Values
        AR = Exponent of z
        SR1 = Fraction of z

   Altered Registers
        AX0,AY1,AY0,AF,AR,SI,SE,SR

   Computation Time
        11 cycles
}

.ENTRY  fps;

fps:    AF=AX0-AY0;                 {Is Ex > Ey?}
        IF GT JUMP shifty;          {Yes, shift y}
```

```
        SI=AX1, AR=PASS AF;         {No, shift x}
        SE=AR;
        SR=ASHIFT SI (HI);
        AR=SR1-AY1;                 {Subtract fractions}
        JUMP subt;
shifty: SI=AY1, AR=-AF;
        SE=AR;
        SR=ASHIFT SI (HI);
        AY1=SR1;
        AY0=AX0, AR=AX1-AY1;        {Subtract fractions}
subt:   SE=EXP AR (HIX);
        AX0=SE, SR=NORM AR (HI);    {Normalize}
        AR=AX0+AY0;                 {Compute exponent}
        RTS;
.ENDMOD;
```

Listing 3.6 Floating-Point Subtraction

3.6 FLOATING-POINT MULTIPLICATION

Multiplication of two numbers in two-word floating-point format is simpler than either addition or subtraction, because there is no need to align the radix points. The algorithm to multiply two numbers *x* and *y* (Ex, Fx and Ey, Fy) whose exponents are biased by an excess code of *b* (which may be set to zero) is as follows:

1. Add Ex and Ey; subtract *b* from this sum to produce the exponent of the result.
2. Multiply Fx by Fy to produce the fraction of the result.
3. Normalize the result.

The ADSP-2100 routine shown in Listing 3.7 reads the exponents of the operands from AX0 and AY0 and the corresponding fractions from AX1 and AY1. The excess value, *b*, is read from MX0. This routine returns the exponent of the result in AR, and the fraction in SR1. After the exponent and fraction of the result are calculated, the routine checks the MV bit for overflow of the least significant 32 bits of the MR register. If MV is set, the MR register is saturated to its full scale value. Saturation is necessary because the exponent detector is unable to process overflowed numbers in

the multiplier. If MR were not saturated on overflow, the routine would incorrectly compute the product of –1 and –1 as –1. The routine finishes by normalizing the product.

```
.MODULE floating_point_multiply;

{
   Floating-Point Multiply
        Z = X × Y

   Calling Parameters
        AX0 = Exponent of X
        AX1 = Fraction of X
        AY0 = Exponent of Y
        AY1 = Fraction of Y
        MX0 = Excess Code

   Return Values
        AR = Exponent of Z
        SR1 = Fraction of Z

   Altered Registers
        AF,AR,AX0,MY1,MX1,MR,SE,SR

   Computation Time
        9 cycles
}

.ENTRY  fpm;

fpm:    AF=AX0+AY0, MX1=AX1;        {Add exponents}
        MY1=AY1;
        AX0=MX0, MR=MX1*MY1 (RND);  {Multiply fractions}
        IF MV SAT MR;               {Check for overflow}
        SE=EXP MR1 (HI);
        AF=AF-AX0, AX0=SE;          {Subtract bias}
        AR=AX0+AF;                  {Compute exponent}
        SR=NORM MR1 (HI);           {Normalize}
        RTS;
```

```
.ENDMOD;
```

Listing 3.7 Floating-Point Multiplication

3.7 FLOATING-POINT DIVISION

The algorithm to divide one number X (= Ex, Fx) by another number Y (= Ey, Fy) in two-word floating-point format is as follows:

1. Subtract Ey from Ex; add the excess value (if any) to this number to form the exponent of the result.
2. Divide Fx by Fy to yield the fraction of the result.
3. Normalize the result.

The ADSP-2100 implementation of this algorithm is shown in Listing 3.8. The routine reads the exponent of X (the dividend) from AX0 and the fraction from AX1. It reads the exponent of Y (the divisor) from AY0 and the fraction from AY1. The excess code *b* is read from MX0. The routine returns the exponent of the quotient in AR, and the fraction of the quotient in SR1. Because both Fx and Fy are in 1.15 format, their division produces a 1.15 quotient. To ensure a valid (1.15 format) quotient, Fx must be less than Fy. If Fx is not less than Fy, the routine shifts Fx one bit right, and Ex is increased by one. After the shift, the division can be performed without producing an overflow. The routine finishes by normalizing the result.

```
.MODULE floating_point_divide;

{
   Floating-Point Divide
        z = x ÷ y

   Calling Parameters
        AX0 = Exponent of x
        AX1 = Fraction of x
        AY0 = Exponent of y
        AY1 = Fraction of y
        MX0 = Excess Code

   Return Values
        AR = Exponent of z
        SR1 = Fraction of z

   Altered Registers
```

(listing continues on next page)

```
        AF,AR,MR,SE,SI,SR,AX1,AX0,AY0

   Computation Time
        33 cycles (maximum)
}

.ENTRY  fpd;

fpd:    SR0=AY1, AR=ABS AX1;
        SR1=AR, AF=ABS SR0;
        SI=AX1, AR=SR1-AF;          {Is Fx > Fy?}
        IF LT JUMP divide;          {Yes, go divide}
        SR=ASHIFT SI BY -1 (LO);    {No, shift Fx right}
        AF=PASS AX0;
        AR=AF+1, AX1=SR0;           {Increase exponent}
        AX0=AR;
divide: AX0=MX0, AF=AX0-AY0;
        MR=0;
        AR=AX0+AF, AY0=MR1;
        AF=PASS AX1, AX1=AY1;       {Add bias}
        DIVS AF, AX1;               {Divide fractions}
        DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
        DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
        DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1; DIVQ AX1;
        MR0=AY0, AF=PASS AR;
        SI=AY0, SE=EXP MR0 (HI);
        AX0=SE, SR=NORM SI (HI);    {Normalize}
        AR=AX0+AF;                  {Compute exponent}
        RTS;
.ENDMOD;
```

Listing 3.8 Floating-Point Division

3.8 FLOATING-POINT MULTIPLY/ACCUMULATE

The floating-point multiply/accumulate routine computes the sum of N two-operand products. This value can also be found using repeated calls to the floating-point multiplication and addition routines, but the multiply/accumulate routine functions more efficiently because it removes overhead. The multiply/accumulate algorithm is as follows:

1. Multiply the first two operands and normalize the product.
2. Multiply the next two operands and normalize the product.
3. Compare the product to the accumulated result, and shift one or the other to align the radix points.
3. Add the product to the accumulated result and normalize the sum.
4. Repeat steps 2 to 4 until all input operands are exhausted.

Floating-Point Arithmetic 3

The routine shown in Listing 3.9 uses I0 to point to the *x* buffer, I1 to point to the *y* buffer. Each buffer should be organized with the exponent of each value first, followed by the fraction. The routine calculates the first product before entering the loop, so CNTR should store the value of the buffer length minus one. MX0 stores the excess value (which may be zero). M0 should be initialized to one. The multiply/accumulate result is returned with the exponent in AR and the fraction in SR1.

After each product is calculated, the MV bit is checked to see whether the MR register overflowed. If overflow occurs, MR is saturated to positive full scale. This saturation is necessary because the exponent detector cannot process overflowed MR register values.

```
.MODULE floating_point_multiply_accumulate;

{  Floating-Point Multiply/Accumulate
             n
         z = ∑ (x(i) × y(i))
            i=1

   Calling Parameters
        I0 —> x Buffer              L0 = 0
        I1 —> y Buffer              L1 = 0
        M0 = 1
        CNTR = Length of Buffer - 1
        MX0 = Excess Code

   Return Values
        AR = Exponent of z
        SR1 = Fraction of z

   Altered Registers
        AF,AR,AX0,AX1,AY0,AY1,MX1,MY1,SE,MR,SR

   Computation Time
        13 ×(n-1)+16
}
.ENTRY  fpmacc;

fpmacc: AX0=DM(I0,M0);              {Get 1st Ex}
        AY0=DM(I1,M0);              {Get 1st Ey}
        AF=AX0+AY0, MX1=DM(I0,M0);  {Add exp., get 1st Fx}
        AR=PASS AF, MY1=DM(I1,M0);  {Get 1st Fy}
        AX1=AR, MR=MX1*MY1(RND);    {Multiply fractions}
```

```
        IF MV SAT MR;               {Check for overflow}
        SE=EXP MR1(HI);
        AY1=SE, SR=NORM MR1(HI);    {Normalize}
        AR=AX1+AY1, AX0=DM(I0,M0);
        AX1=AR;
        AY0=DM(I1,M0);
        DO macc UNTIL CE;
        AF=AX0+AY0, MX1=DM(I0,M0);  {Compute product exp.}
        AR=AX1-AF, MY1=DM(I1,M0);   {Sum exp. > product exp.?}
        IF GT JUMP shiftp;          {Yes, shift product}
        SE=AR, MR=MX1*MY1(RND);     {No, shift sum}
        IF MV SAT MR;
        AY1=MR1, AR=PASS AF;
        AX1=AR, SR=ASHIFT SR1(HI);
        JUMP add;
shiftp: AF=PASS AR;
        AR=-AF;
        SE=AR, MR=MX1*MY1(RND);
        IF MV SAT MR;
        AY1=SR1, SR=ASHIFT MR1(HI);
add:    AR=SR1+AY1, AX0=DM(I0,M0);  {Accumulate}
        SE=EXP AR(HIX);
        AY1=SE, SR=NORM AR(HI);     {Normalize}
        AR= AX1+AY1, AY0=DM(I1,M0);
macc:   AX1=AR;
        SR0=MX0;                    {Get bias}
        AF=PASS SR0;
        AR=AX1-AF;                  {Subtract bias}
        RTS;
.ENDMOD;
```

Listing 3.9 Floating-Point Multiply/Accumulate

3.9 REFERENCES

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

IEEE Standard for Binary Floating-Point Arithmetic: ANSI/IEEE Std 754-1985. 1985. New York: The Institute of Electrical and Electronics Engineers, Inc.