6.2.3 Radix-2 Decimation-In-Frequency FFT Algorithm

In the DIT FFT, each decimation consists of two steps. First, a DFT equation is expressed as the sum of two DFTs, one of even samples and one of odd samples. This equation is then divided into two equations, one that computes the first half of the output (frequency) samples and one that computes the second half. In the decimation-in-frequency (DIF) FFT, a DFT equation is expressed as the sum of two calculations, one on the first half of the samples and one on the second half of the samples. This equation is then expressed as two equations, one that computes even output samples and one that computes odd output samples. Decimation in time refers to grouping the input sequence into even and odd samples, whereas decimation in frequency refers to grouping the output (frequency) sequence into even and odd samples. Decimation-in-frequency can thus be visualized as repeatedly dividing the output sequence into even and odd samples in the same way that decimation in time divides down the input sequence (Oppenheim, 1975). The following equations illustrate radix-2 decimation in frequency.

The DIF FFT divides an N-point DFT into two summations, shown in (11).

$$(11) \qquad X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/2-1} x(n)\, W_N^{nk} + \sum_{n=N/2}^{N-1} x(n) W_N^{nk}$$

$$= \sum_{n=0}^{N/2-1} x(n)\, W_N^{nk} + \sum_{n=0}^{N/2-1} x(n+N/2) W_N^{(n+N/2)k}$$

Because $W_N^{(n+N/2)k} = W_N^{nk} \times W_N^{(N/2)k}$ and $W_N^{(N/2)k} = (-1)^k$, equation (11) can also be expressed as

$$(12) \qquad X(k) = \sum_{n=0}^{N/2-1} x(n)\, W_N^{nk} + (-1)^k \sum_{n=0}^{N/2-1} x(n+N/2)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/2-1} [x(n) + (-1)^k x(n+N/2)] W_N^{nk}$$

for k = 0 to N–1

The decimation of the output (frequency) sequence is accomplished by dividing X(k) into two equations, one that computes even output samples and one that computes odd output samples. For even values of X(k), k=2r.

$$X(2r) = \sum_{n=0}^{N/2-1} [x(n) + (-1)^{2r} x(n+N/2)] W_N^{2nr} \tag{13}$$

$$= \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)] W_{N/2}^{nr}$$

r = 0 to N/2–1

For odd values of X(k), k=2r+1.

$$X(2r+1) = \sum_{n=0}^{N/2-1} [x(n) + (-1)^{2r+1} x(n+N/2)] W_N^{(2r+1)n} \tag{14}$$

$$= \sum_{n=0}^{N/2-1} [[x(n) - x(n+N/2)] W_N^{n}] W_{N/2}^{nr}$$

r = 0 to N/2–1

Note that X(2r) and X(2r+1) are the results of N/2-point DFTs performed on the sum and difference of the first and second halves of the input sequence. In equation (14), the difference of the two halves of the input sequence is multiplied by a twiddle factor, W_N^n. Figure 6.5, on the next page, illustrates the first decimation of the DIF FFT, which eliminates half ($N^2/2$) of the DFT calculations.

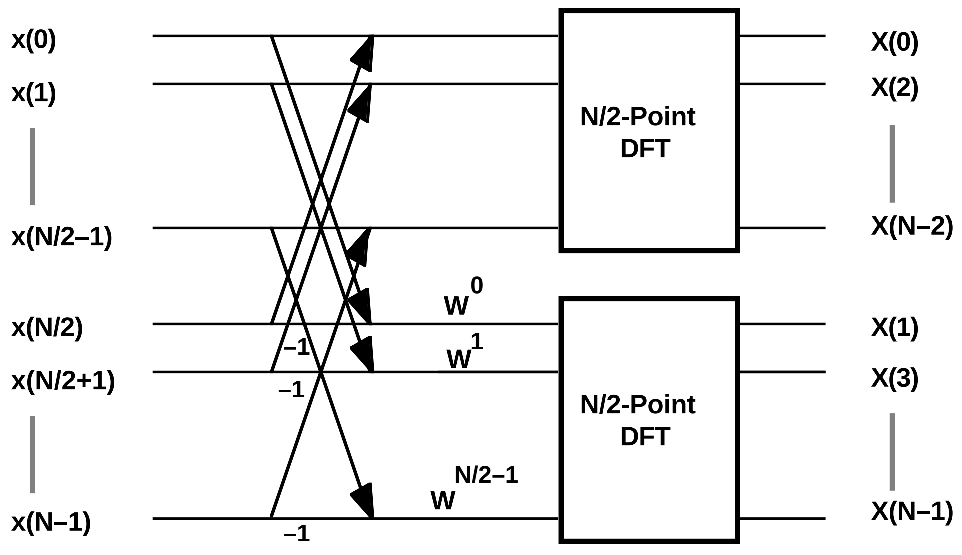


Figure 6.5 First Decimation of DIF FFT

Each of the two N/2-point DFTs (X(2r) and X(2r+1)) are divided into two N/4-point DFTs in the same way as the N-point DFT is divided into two N/2-point DFTs. By the substitutions

$$X_1(r) = X(2r) \qquad r = 0 \text{ to } N/2-1$$

$$x_1(n) = x(n) + x(n+N/2) \qquad n = 0 \text{ to } N/2-1$$

the sequence of even samples in equation (13) becomes

$$(15) \qquad X_1(r) = \sum_{n=0}^{N/2-1} x_1(n)\, W_{N/2}{}^{rk}$$

This N/2-point sequence has the same form as the original N-point sequence in equation (11) and can be divided in half in the same manner to yield

$$(16) \qquad X_1(r) = \sum_{n=0}^{N/4-1} [\, x_1(n) + (-1)^r x_1(n+N/4)]\, W_{N/2}{}^{nr}$$

For even output samples, let r=2s.

$$(17) \qquad X_1(2s) = \sum_{n=0}^{N/4-1} [x_1(n) + x_1(n+N/4)]\, W_{N/4}^{sn}$$

For odd output samples, let r=2s+1.

$$(18) \qquad X_1(2s+1) = \sum_{n=0}^{N/4-1} [(x_1(n) - x_1(n+N/4))\, W_N^{2n}]\, W_{N/4}^{sn}$$

X(2r+1) is also divided into two equations, one that computes even output samples and one that computes odd output samples, in the same way that X(2r) is divided into $X_1(2s)$ and $X_1(2s+1)$. Thus we have four N/4-point sequences.

If we make the substitutions

$$X_2(s) = X_1(2s)$$

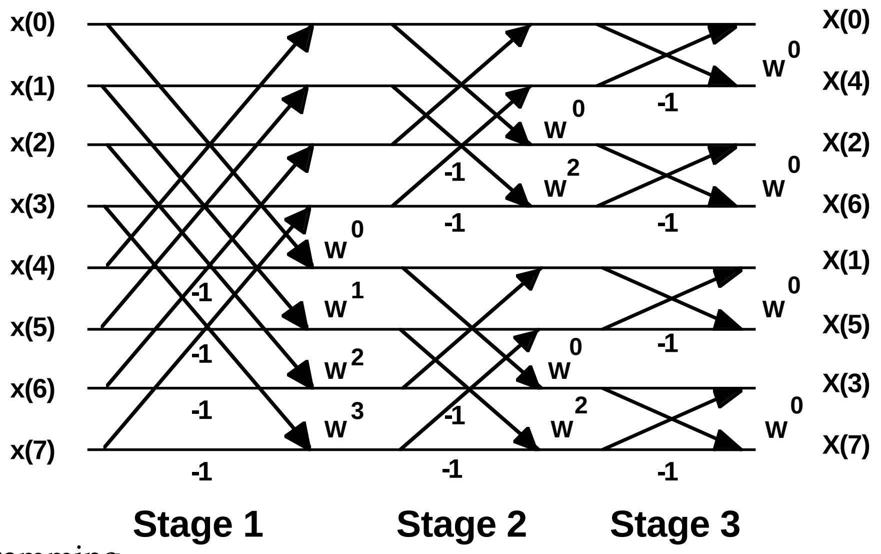
$$x_2(n) = x_1(n) + x_1(n+N/4)$$

equation (17) becomes

$$(19) \qquad X_2(s) = \sum_{n=0}^{N/4-1} x_2(n)\, W_{N/4}^{sn}$$

The four N/4-point sequences that result from the decimation of X(2r) and X(2r+1) are divided to form eight N/8-point sequences in the third decimation. This process is repeated until the division of a sequence results in a pair of equations that together compute a two-point DFT. In this pair, the summation variable *n* (see equations 17 and 18) is equal to zero only, so no summation is performed. The two-point DFT computed by this pair of equations is the core calculation (butterfly) for the radix-2 DIF FFT.

Figure 6.6, on the next page, shows the complete decimation for an eight-point DIF FFT. Notice that the inputs of the DIF FFT are in sequential order and the outputs are in scrambled order. The DIF FFT can also be performed with inputs in bit-reversed order, resulting in outputs in sequential order. In this case, however, the twiddle factors must be in bit-reversed order. In this chapter, both the DIT FFT and the DIF FFT are presented with twiddle factors in sequential order to simplify



programming.

Figure 6.6 Eight-Point DIF FFT

As in the DIT FFT, the DIF FFT butterflies are organized into groups and stages. In the eight-point FFT, the first stage has one group of four (N/2) butterflies. The second stage has two groups of two (N/4) butterflies, and the last has four groups of one butterfly. In general, an N-point DIF FFT has the characteristics summarized below.

	Stage 1	*Stage 2*	*Stage 3*	*Stage Log_2N*
Number of Groups	1	2	4	N/2
Butterflies per Group	N/2	N/4	N/8	1
Dual-Node Spacing	N/2	N/4	N/8	1
Twiddle Factor Exponents	n, n=0 to N/2–1	2n, n=0 to N/4–1	4n, n=0 to N/8–1	(N/2)n, n=0

The DIF FFT butterfly is similar to that of the DIT FFT except that the twiddle factor multiplication occurs after rather than before the primary-node and dual-node subtraction. The DIF butterfly is illustrated graphically in Figure 6.7. The variables *x* and *y* represent the real and imaginary parts, respectively, of a sample. The twiddle factor can be divided into real and imaginary parts because $W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N)$. In the program presented later in this section, the twiddle factors are initialized in memory as cosine and –sine values (not +sine). For this reason, the twiddle factors are shown in Figure 6.7 as C + j(–S). C represents cosine and –S represents –sine.

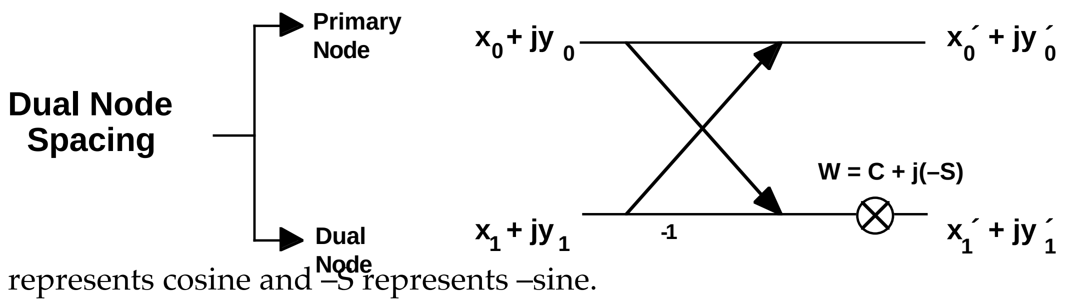


Figure 6.7 Radix-2 DIF FFT Butterfly

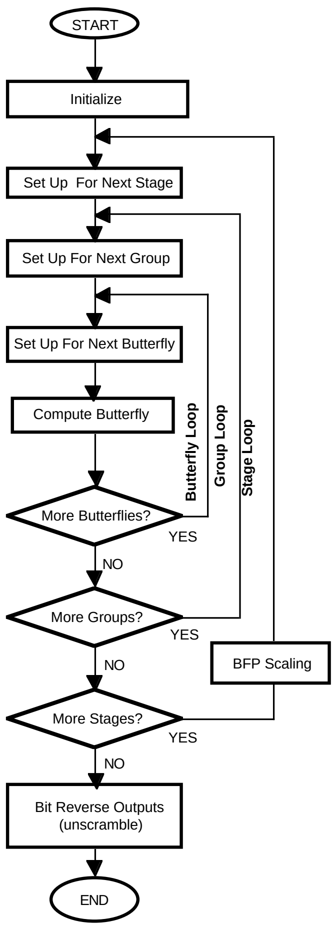


Figure 6.8 Radix-2 DIF FFT Flow Chart

Equations (20) through (23) describe the DIF FFT butterfly outputs.

(20) $x_0' = x_0 + x_1$

(21) $y_0' = y_0 + y_1$

(22) $x_1' = C(x_0 - x_1) - (-S)(y_0 - y_1)$

(23) $y_1' = (-S)(x_0 - x_1) + C(y_0 - y_1)$

As in the DIT FFT, the butterfly is performed in-place; that is, the results of each butterfly are written over the corresponding inputs. For example, x_0' is written over x_0.

6.2.4 Radix-2 Decimation-In-Frequency FFT Program

The DIF flow chart is shown in Figure 6.8. Like the DIT FFT, the DIF FFT uses three subroutines. The first subroutine computes the FFT. The second subroutine performs conditional block floating-point scaling at the end of each stage (except the last). The third subroutine bit-reverses the locations of the FFT output data to “unscramble” the data. The DIF FFT subroutine is described in this section.The block floating-point and bit reversal routines are described in later sections.

6.2.4.1 Main Module

The module *dif_fft_main* is shown in Listing 6.8. The FFT calculation is performed in one buffer (*inplacedata*). In this program, the real and imaginary input data are interleaved in the buffer. The length of *inplacedata* is thus twice the number of points in the FFT and is specified by the constant *N_x_2* (N_x_2 = 2048 for a 1024-point FFT). Unlike the DIT FFT, the DIF FFT is performed on sequentially ordered input data and produces data in bit-reversed order; therefore, no additional buffers for scrambling the input data are needed.

When the output data is unscrambled, it is separated into real and imaginary values and placed in two buffers (*real_results*, *imaginary_results*). Twiddle-factor buffers are defined and initialized as in the DIT FFT.

The DIF FFT uses the variables *groups*, *bflys_per_group* and *blk_exponent.* Because the first stage of the DIF FFT contains one group of N/2 butterflies, *groups* is initialized to one and *bflys_per_group* is initialized to *N_div_2*. The node spacing (*node_space*) is N instead of N/2 because the real and imaginary input data are interleaved.

Two subroutines are called. The first performs the DIF FFT and calls the block floating-point scaling routine. The second bit-reverses the FFT outputs to unscramble them.

```
.MODULE/ABS=4            dif_fft_main;

.CONST                   N=1024,N_x_2=2048;          {Const. for 1024 points}
.CONST                   N_div_2=512,log2N=10;
.VAR/DM/RAM              inplacedata[N_x_2];
.VAR/DM/RAM              real_results[N];
.VAR/DM/RAM              imaginary_results[N];
.VAR/PM/ROM/CIRC         twid_imag[N_div_2];
.VAR/PM/ROM/CIRC         twid_real[N_div_2];

.VAR/DM/RAM              groups,node_space,bflys_per_group,blk_exponent;

.INIT                    inplacedata: <inplacedata.dat>;
.INIT                    twid_imag: <twid_imag.dat>;
.INIT                    twid_real: <twid_real.dat>;
.INIT                    groups: 1;
.INIT                    node_space: N;
.INIT                    bflys_per_group: N_div_2;
.INIT                    blk_exponent: 0;

.GLOBAL                  inplacedata, real_results, imaginary_results;
.GLOBAL                  twid_real, twid_imag;
.GLOBAL                  groups, bflys_per_group, node_space, blk_exponent;

.EXTERNAL                unscramble, fft_start;

                         CALL fft_start;
                         CALL unscramble;
                         TRAP;
.ENDMOD;
```

Listing 6.8 Main Module, Radix-2 DIF FFT

6.2.4.2 DIF FFT Module

The conditional block floating-point DIF FFT program is described in this section. The butterfly loop is described first, then the group and stage loops. The complete FFT program is presented at the end of this section.

Butterfly Loop

The code segment for the DIF butterfly (with conditional block floating-point scaling) is shown in Listing 6.9, on the next page. The primary-node outputs x_0´ and y_0´ are calculated first and written over x_0 and y_0. Complex subtraction for the dual-node calculation is then performed, followed by the twiddle factor multiplication. The outputs x_1´ and y_1´ are written over x_1 and y_1. Instructions that write butterfly results to memory are boldface. Each butterfly output is checked for bit growth using the EXPADJ instruction. This loop is repeated *bflys_per_group* times.

The input and output parameters for the butterfly loop are as follows:

Initial Conditions	*Final Conditions*
AX0 = x_0	AX0 = next x_0
AY0 = x_1	AY0 = next x_1
AY1 = y_1	AY1 = next y_1
I0 --> y_0	I0 --> next y_0
I1 --> next x_1	I1 --> x_1 after next
I2 --> x_1	I2 --> next x_1
I4 --> C	CNTR = butterfly count –1
I5 --> (-S)	
M0 = –1	
M1 = 1	
M5 = twiddle factor modifier	
CNTR = butterfly count	

```
AR=AX0+AY0,AX1=DM(I0,M0),MY0=PM(I4,M5); {AR=x0+x1,AX1=y0,MY0=C,I0 -->x0}
SB=EXPADJ AR;                           {Check for bit growth}
DM(I0,M1)=AR,AR=AX1+AY1;                {x0´=x0+x1,AR=y0+y1,I0 -->y0}
SB=EXPADJ AR;                           {Check for bit growth}
DM(I0,M1)=AR,AR=AX0-AY0;                {y0´=y0+y1,AR=x0-x1,I0 -->next x0}
MX0=AR,AR=AX1-AY1;                      {MX0=x0-x1,AR=y0-y1}
MR=MX0*MY0(SS),AX0=DM(I0,M1),MY1=PM(I5,M5);
                                        {MR=(x0-x1)C,AX0=next x0,MY1=(-S)}
MR=MR-AR*MY1(RND),AY0=DM(I1,M1);        {MR=(x0-x1)C-(y0-y1)(-S),AY0=next x1}
SB=EXPADJ MR1;                          {Check for bit growth}
DM(I2,M1)=MR1,MR=AR*MY0(SS);            {x1´=(x0-x1)C-(y0-y1)(-S),MR=(y0-y1)C}
              MR=MR+MX0*MY1(RND),AY1=DM(I1,M1);      {MR=(y0-y1)C+(x0-x1)(-
              S),AY1=next y1}
              DM(I2,M1)=MR1,SB=EXPADJ MR1;           {y1´=(y0-y1)C+(x0-x1)(-
              S),check}
                                                     {for bit growth}
```

Listing 6.9 Radix-2 DIF FFT Butterfly, Conditional Block Floating-Point

Group Loop

The group loop code is shown in Listing 6.10. The group loop sets up the butterfly loop by fetching initial data and initializing the butterfly loop counter. When all the butterflies in a group have been calculated, data pointers are updated to point to the inputs for the first butterfly of the next group. This loop is repeated until all groups in a stage are complete.

The input and output parameters of the group loop are as follows:

Initial Conditions	*Final Conditions*
I0 --> x_0 of first butterfly in group	I0 --> x_0 of first butterfly in next group
I1 --> x_1 of first butterfly in group	I1 --> x_1 of first butterfly in next group
I2 --> x_1 of first butterfly in group	I2 --> x_1 of first butterfly in next group
CNTR = group count	CNTR = group count –1
M1 = 1	
M2 = *node_space*	
M3 = *node_space*–2	

```
            CNTR=DM(bflys_per_group); {Initialize butterfly counter}
            AX0=DM(I0,M1);            {AX0=x0}
            AY0=DM(I1,M1);            {AY0=x1}
            AY1=DM(I1,M1);            {AY1=y1}
            DO bfly_loop UNTIL CE;
bfly_loop:      {Calculate All Butterflies in Group}

            MODIFY(I2,M2);            {I2 ->x1 of 1st butterfly in next group}
            MODIFY(I1,M3);            {I1 ->x1 of 1st butterfly in
next group}
            MODIFY(I0,M3);
group_loop: MODIFY(I0,M1);            {I0 ->x0 of 1st butterfly in
next group}
```

Listing 6.10 Radix-2 DIF FFT Group Loop

Stage Loop

The stage loop code is shown in Listing 6.11, on the next page. This code segment sets up and computes all groups in a stage and controls stage characteristics, such as the number of groups in a stage. Pointers I0 and I1 are set to point to x_0 and x_1 of the first butterfly in the first group of the stage. Pointer I2 also points to x_1 and is used to write the dual-node butterfly results to data memory. M3 is set to *node_space*–2 and is used to modify pointers for the next group. The group counter is initialized to *groups*, the number of groups in the stage. The twiddle factor modifier stored in M5 is also *groups*. This value is the exponent increment value for the twiddle factors of consecutive butterflies in a group.

SB is set to –2 to detect any bit growth into the guard bits of any butterfly output. When all the groups in a stage are computed, the *bfp_adjustment* routine is called to check for bit growth and adjust the output data if necessary. Then parameters for the next stage are updated; *groups* is doubled and *node_space* and *bflys_per_group* are divided in half. The stage loop is repeated log_2N times.

The input and output parameters of the stage loop are summarized below. Note that the parameters are passed in memory locations.

Initial Conditions	*Final Conditions*
groups=# groups in stage	*groups=#* groups in next stage
bflys_per_group=# butterflies/group	*bflys_per_group=#*butterflies/group (next stage)
node_space=node spacing current stage	*node_space*=node spacing next stage
inplacedata=stage input data	*inplacedata*=stage output data

```
            I0=^inplacedata;            {I0 ->x0 in 1st butterfly of stage}
            I1=^inplacedata;
            AY0=DM(node_space);
            M2=AY0;                     {M2=dual node spacing}
            MODIFY(I1,M2);              {I1 ->x1 in 1st butterfly of stage}
            I2=I1;
            AX0=2;
            AR=AY0-AX0;
            M3=AR;                      {M3=node_space-2}
            CNTR=DM(groups);            {Initialize group counter}
            SB=-2;                      {Set minimum allowable sign bits to two}
            M5=DM(groups);              {M5=twiddle factor modifier}
            DO group_loop UNTIL CE;

group_loop:    {Calculate All Groups in Stage}

            CALL bfp_adjustment;        {Adjust block data for bit growth}
            SI=DM(groups);
            SR=LSHIFT SI BY 1 (LO);
            DM(groups)=SR0;             {groups=groups × 2}
            SI=DM(node_space);
            SR=LSHIFT SI BY -1 (LO);
                          DM(node_space)=SR0;         {node_space=node_space ÷ 2}
                          SR=LSHIFT SR0 BY -1(LO);
                          DM(bflys_per_group)=SR0;
              {bflys_per_group=bflys_per_group ÷ 2}
```

Listing 6.11 Radix-2 DIF FFT Stage Loop

DIF FFT Subroutine

The complete block floating-point DIF FFT subroutine is shown in Listing 6.12. Initializations of index, modifier and length registers that retain the same values throughout the FFT calculation are performed before the stage loop is entered. Instructions that write butterfly results to memory are boldface.

```
.MODULE     dif_fft;

.CONST      N=1024, N_div_2=512, log2N=10;

.EXTERNAL   inplacedata, twid_real, twid_imag;
.EXTERNAL   groups,bflys_per_group,node_space;
.EXTERNAL   bfp_adjust;

.ENTRY      fft_start;

fft_start:  I4=^twid_real;          {I4 -> C OF W0}
            L4=N_div_2;
            I5=^twid_imag;          {I5 -> (-S) OF W0}
            L5=N_div_2;
            M0=-1;
            M1=1;
            CNTR=log2N;             {Initialize stage counter}
            L0=0;
            L1=0;
            L2=0;
            DO stage_loop UNTIL CE;
               I0=^inplacedata;         {I0 -> x0}
               I1=^inplacedata;
               AY0=DM(node_space);
               M2=AY0;
               MODIFY(I1,M2);           {I1 -> x1}
               I2=I1;
               AX0=2;
               AR=AY0-AX0;
               M3=AR;                   {M3=node_space-2}
               CNTR=DM(groups);         {Initialize group counter}
               SB=-2;
               M5=CNTR;                 {Init. twiddle factor modifier}
               DO group_loop UNTIL CE;
                  CNTR=DM(bflys_per_group);     {Init. butterfly counter}
                  AX0=DM(I0,M1);                {AX0=x0}
                  AY0=DM(I1,M1);                {AY0=x1}
                  AY1=DM(I1,M1);                {AY1=y1}
```

(lisitng continues on next page)

```
                  DO bfly_loop UNTIL CE;
                      AR=AX0+AY0, AX1=DM(I0,M0), MY0=PM(I4,M5);
                                                 {AR=x0+x1,AX1=y0,MY0=C}
                      SB=EXPADJ AR;              {Check for bit growth}
                      DM(I0,M1)=AR,AR=AX1+AY1;   {x0´=x0+x1,AR=y0+y1}
                      SB=EXPADJ AR;              {Check for bit growth}
                      DM(I0,M1)=AR,AR=AX0-AY0;   {y0´=y0+y1,AR=x0-x1}
                      MX0=AR, AR=AX1-AY1;        {MX0=x0-x1,AR=y0-y1}
                      MR=MX0*MY0 (SS), AX0=DM(I0,M1), MY1=PM(I5,M5);
                                        {MR=(x0-x1)C,AX0=next x0,MY1=(-S)}
                      MR=MR-AR*MY1 (RND), AY0=DM(I1,M1);
                                        {MR=(x0-x1)C-(y0-y1)(-S),AY0=next x1}
                      SB=EXPADJ MR1;             {Check for bit growth}
                      DM(I2,M1)=MR1, MR=AR*MY0 (SS);
                                   {x1´=(x0-x1)C-(y0-y1)(-S),MR=(y0-y1)C}
                      MR=MR+MX0*MY1 (RND), AY1=DM(I1,M1);
                                   {MR=(y0-y1)C+(x0-x1)(-S),AY1=next y1}
bfly_loop:            DM(I2,M1)=MR1,SB=EXPADJ MR1;
                                   {y1´=(y0-y1)C+(x0-x1)(-S),check bit growth}
                  MODIFY(I2,M2);   {I2->x1 of first butterfly in next group}
                  MODIFY(I1,M3);   {I1->x1 of first butterfly in next group}
                  MODIFY(I0,M3);
group_loop:       MODIFY(I0,M1);   {I0->x0 of first butterfly in next group}
               CALL bfp_adjust;    {Adjust block data for bit growth}
               SI=DM(groups);
               SR=LSHIFT SI BY 1 (LO);
               DM(groups)=SR0;          {groups=groups × 2}
               SI=DM(node_space);
               SR=LSHIFT SI BY -1 (LO);
               DM(node_space)=SR0;      {node_space=node_space ÷ 2}
               SR=LSHIFT SR0 BY -1 (LO);
stage_loop:    DM(bflys_per_group)=SR0;
                                   {bflys_per_group=bflys_per_group ÷ 2}
                             RTS;
                 .ENDMOD;
```

Listing 6.12 Radix-2 DIF FFT Routine, Conditional Block Floating-Point

6.2.5 Bit Reversal

Bit reversal is an addressing technique used in FFT calculations to obtain results in sequential order. Because the FFT repeatedly subdivides data sequences, the data and/or twiddle factors may be scrambled (in bit-reversed order). All radix-2 FFTs can be calculated with either the input sequence or the output sequence scrambled. The twiddle factors may also need to be scrambled, depending on the order of the input and output sequences. In this chapter, however, input and output sequences are set up so that twiddle factors are never scrambled. This simplifies the FFT explanation as well as the program.

As described earlier, the input sequence to the radix-2 DIT FFT is scrambled before the FFT is performed. Similarly, the output sequence of the radix-2 DIF FFT is unscrambled after the FFT is performed. Scrambling and unscrambling are both accomplished through bit reversal.

An example of bit reversal is shown below. Bit reversal operates on the binary number that represents the position of a sample within an array of samples. Each position is shown in decimal and binary. For example, the position of x(4) in sequential order is four (binary 100). Note that this position does not necessarily correspond to the location of the sample in memory. The bit-reversed position is the transpose of the bits of the binary number about its center; the transpose of the binary number 100 is 001. In this example, three bits are needed to represent eight positions, so bits zero and two are interchanged. Four bits are needed to represent 16 positions, so in a 16-point sequence, bits zero and three and bits one and two would be interchanged. A 1024-point sequence requires the reversal of ten bits.

Sample, Sequential Order	*Sequential Location*		*Bit-Reversed Location*		*Sample, Bit-Reversed Order*
	decimal	binary	decimal	binary	
x(0)	0	000	0	000	x(0)
x(1)	1	001	4	100	x(4)
x(2)	2	010	2	010	x(2)
x(3)	3	011	6	110	x(6)
x(4)	4	100	1	001	x(1)
x(5)	5	101	5	101	x(5)
x(6)	6	110	3	011	x(3)
x(7)	7	111	7	111	x(7)

When the samples in sequential order are scrambled, x(0) remains in location zero, x(1) moves to location four, x(2) remains in location two, x(3) moves to location six, etc. Conversely, if samples are already scrambled, bit reversal unscrambles them.

The ADSP-2100 has a bit-reverse capability built into its data address generator #1 (DAG1). When a mode bit is enabled (through software), the 14-bit address generated by DAG1 is automatically bit-reversed for any data memory read or write. The two address generators of the ADSP-2100 greatly simplify bit reversal. One address generator can be used to read sequentially ordered data, and the other can be used to write the same data to its bit-reversed location. Because the address generators are independent, intermediate enabling and disabling of the bit-reverse mode is not needed.

The base (starting) address of an array being accessed with bit-reversed addressing must be an integer multiple of the length (N) of the transform (i.e., base address=0, N, 2N, …).

In many cases, fewer than 14 bits must be reversed (for example, an eight-point FFT needing only three bits reversed). Reversal of fewer than 14 bits is accomplished by adding the correct modify value to the address pointer after each memory access. The following example demonstrates bit reversal of ten bits using I0 to store the address to be reversed and M0 to store the modify value.

First, we determine the first bit-reversed address. This address is the first 14-bit address with the ten least significant bits reversed. For the DIT FFT subroutine, the first address in the *inplacereal* buffer is H#0000. If we reverse the ten least significant bits of H#0000, we still have H#0000. Thus, we want to output H#0000 as the first bit-reversed address. To do so, I0 must be initialized to the number that, when bit-reversed by the ADSP-2100 (all 14 bits), is H#0000. In this case, that number is also H#0000.

The second bit-reversed address must be H#0200 (H#0001 with ten least significant bits reversed). We must modify I0 to the value that, when bit-reversed (all 14 bits) is H#0200. This value is H#0010. Since I0 contains H#0000, we must add H#0010 to it. Thus, H#0010 is loaded into M0. After the first data memory read or write, which outputs H#0000, M0 is added to the (non-bit-reversed) address in I0 so that I0 contains H#0010. On the second data memory read or write, I0 is bit-reversed (14 bits) and the resulting address is H#0200, the correct second bit-reversed address.

In general, the modify value is determined by raising two to the difference between 14 and the number of bits to be reversed. In this ten-bit example, the value is $2^{(14-10)}$ = H#0010. Adding this value to I0 after each memory access and reversing all 14 bits on the next memory access yields the correct bit-reversed addresses for ten bits. The first four bit-reversed addresses are shown below.

Sequence	*I0, Non-Bit-Reversed*		*I0, Bit-Reversed*	
	H#	B#	H#	B#
0	0000	00 0000 0000 0000	0000	00 00**00 0000 0000**
1	0010	00 0000 0001 0000	0200	00 00**10 0000 0000**
2	0020	00 0000 0010 0000	0100	00 00**01 0000 0000**
3	0030	00 0000 0011 0000	0300	00 00**11 0000 0000**

Only the ten least significant bits (boldface) are bit-reversed. Each time a data memory write is performed, I0 is modified by M0. Note that the modified I0 value is not bit-reversed. Bit reversal only occurs when a data memory read or write is executed.

Two bit reversal routines are shown in Listings 6.13 and 6.14 (*scramble* and *unscramble*, respectively). The *scramble* routine places the inputs to the DIT FFT in bit-reversed order. The *unscramble* routine places the output data of the DIF FFT in sequential order. Both modules begin by initializing two constants. The first constant (*N*) is the number of input points in the FFT. The second constant (*mod_value*) is the modify value for the pointer which outputs the bit-reversed addresses. Pointers to the data buffers are initialized, and the bit-reverser is enabled for DAG1. In bit-reverse mode, any addresses output from registers I0, I1, I2, or I3 will be bit-reversed. I0 is used in *scramble*, and I1 is used in *unscramble.*

The *scramble* routine assumes real input data. In this case, the imaginary data is all zeros and can be initialized directly into the *inplaceimag* buffer. The *brev* loop consists of two instructions. First, the sequentially ordered data is read from the *input_real* buffer using I4 (from DAG2). Then, the same data is written to the bit-reversed location in the *inplacereal* buffer using I0 (from DAG1). After all the real input data has been placed in bit-reversed order in the *inplacereal* buffer, the bit-reverser is disabled for the rest of the FFT calculation.

The *unscramble* routine uses two loops: one to unscramble the real FFT output data, the other to unscramble the imaginary output data. I4 points to the first of the scrambled real data values in the *inplacedata* buffer. I4 is modified by two (in M4) after each read. Because the real and imaginary data in *inplacedata* are interleaved, this ensures that only real data is read for the first loop. I1 contains the (bit-reversed) address of the first location in the *real_results* buffer (for unscrambled real data). The appropriate modify value (stored in M1) is added to I1 upon each data memory write. Before entering the second loop, I4 is then updated to point to the first imaginary data in *inplacedata* and I1 is set to the first address (bit-reversed) of the *imag_results* buffer (for sequentially ordered imaginary data).

```
.MODULE     dit_scramble;

{           Calling Parameters
               Sequentially ordered input data in inputreal
               The base (starting) address of an array being accessed with
               bit-reversed addressing must be an integer multiple of the
               length (N) of the transform (i.e., base address=0,N,2N,…).

            Return Values
               Scrambled input data in inplacereal

            Altered Registers
               I0,I4,M0,M4,AY1

            Altered Memory
               inplacereal
}

.CONST       N=1024,mod_value=H#0010;      {Initialize constants}
.CONST      N=1024,mod_value=H#0010;       {Initialize constants}
.EXTERNAL    inputreal, inplacereal;
.ENTRY      scramble;

scramble:   I4=^inputreal;             {I4-->sequentially ordered data}
            I0=^inplacereal;           {I0-->scrambled data}
            M4=1;
            M0=mod_value;              {M0=modifier for reversing N bits}
            L4=0;
            L0=0;
            CNTR = N;
            ENA BIT_REV;               {Enable bit-reversed outputs on DAG1}
            DO brev UNTIL CE;
               AY1=DM(I4,M4);          {Read sequentially ordered data}
brev:          DM(I0,M0)=AY1;          {Write data in bit-reversed location}
            DIS BIT_REV;               {Disable bit-reverse}
            RTS;                       {Return to calling program}
.ENDMOD;
```

Listing 6.13 Bit-Reverse (Scramble) Routine

```
.MODULE         dif_unscramble;

{               Calling Parameters
                   Real and imaginary scrambled output data in inplacedata
                   Output data is stored with bit-reversed addressing, starting
                   at address 0.

                Return Values
                   Sequentially ordered real output data in real_results
                   Sequentially ordered imag. output data in imaginary_results

                Altered Registers
                   I1,I4,M1,M4,L1,AY1,CNTR

                Altered Memory
                   real_results, imaginary_results
}

.CONST          N=1024,mod_value=H#0010;             {Initialize constants}

.EXTERNAL       inplacedata;

.ENTRY          unscramble;         {Declare entry point into module}

unscramble:     I4=^inplacedata;    {I4-->real part of 1st data point}
                M4=2;               {Modify by 2 to fetch only real data}
                L1=0;
                L4=0;
                I1=H#4;             {I1=1st real output addr, bit-reversed}
                M1=mod_value;       {Modifier for 10-bit reversal}
                CNTR=N;             {N=number of real data points}
                ENA BIT_REV;        {Enable bit-reverse}
                DO bit_rev_real UNTIL CE;
                   AY1=DM(I4,M4);   {Read real data}
bit_rev_real:      DM(I1,M1)=AY1;   {Place in sequential order}
                I4=^inplacedata+1;  {I4-->imag. part of 1st data point}
                I1=H#C;             {I1=1st imag. output addr, bit-reversed}
                CNTR=N;             {N=number of imaginary data points}
                DO bit_rev_imag UNTIL CE;
                   AY1=DM(I4,M4);   {Read imag. data}
bit_rev_imag:      DM(I1,M1)=AY1;   {Place in sequential order}
                DIS BIT_REV;        {Disable bit-reverse}
                RTS;
.ENDMOD;
```

Listing 6.14 Bit-Reverse (Unscramble) Routine