## 6.5 RADIX-4 FAST FOURIER TRANSFORMS

Whereas a radix-2 FFT divides an N-point sequence successively in half until only two-point DFTs remain, a radix-4 FFT divides an N-point sequence successively in quarters until only four-point DFTs remain. An N-point sequence is divided into four N/4-point sequences; each N/4-point sequence is broken into four N/16-point sequences, and so on, until only four-point DFTs are left. The four-point DFT is the core calculation (butterfly) of the radix-4 FFT, just as the two-point DFT is the butterfly for a radix-2 FFT.

A radix-4 FFT essentially combines two stages of a radix-2 FFT into one, so that half as many stages are required. The radix-4 butterfly is consequently larger and more complicated than a radix-2 butterfly; however, fewer butterflies are needed. Specifically, N/4 butterflies are used in each of $(\log_2 N)/2$ stages, which is one quarter the number of butterflies in a radix-2 FFT. Although addressing of data and twiddle factors is more complex, a radix-4 FFT requires fewer calculations than a radix-2 FFT. The addressing capability of the ADSP-2100 can accommodate the added complexity, and so the it can compute a radix-4 FFT significantly faster than a radix-2 FFT. Like the radix-2 FFT, the radix-4 FFT requires data scrambling and/or unscrambling. However, radix-4 FFT sequences are scrambled and unscrambled through digit reversal, rather than bit reversal as in the radix-2 FFT. Digit reversal is described later in this section.

### 6.5.1 Radix-4 Decimation-In-Frequency FFT Algorithm

The radix-4 FFT divides an N-point DFT into four N/4-point DFTs, then into 16 N/16-point DFTs, and so on. In the radix-2 DIF FFT, the DFT equation is expressed as the sum of two calculations, one on the first half and one on the second half of the input sequence. Then the equation is divided to form two equations, one that computes even samples and the other that computes odd samples. Similarly, the radix-4 DIF FFT expresses the DFT equation as four summations, then divides it into four equations, each of which computes every fourth output sample. The following equations illustrate radix-4 decimation in frequency.

$$X(k) = \sum_{n=0}^{N-1} x(n)\, W_N^{nk} \tag{24}$$

$$= \sum_{n=0}^{N/4-1} x(n)\, W_N^{nk} + \sum_{n=N/4}^{N/2-1} x(n)\, W_N^{nk} + \sum_{n=N/2}^{3N/4-1} x(n)\, W_N^{nk} + \sum_{n=3N/4}^{N-1} x(n)\, W_N^{nk}$$

$$= \sum_{n=0}^{N/4-1} x(n)\, W_N^{nk} + \sum_{n=0}^{N/4-1} x(n+N/4)\, W_N^{(n+N/4)k}$$

$$+ \sum_{n=0}^{N/4-1} x(n+N/2)\, W_N^{(n+N/2)k} + \sum_{n=0}^{N/4-1} x(n+3N/4) W_N^{(n+3N/4)k}$$

$$= \sum_{n=0}^{N/4-1} [\, x(n) + W_N^{k(N/4)} x(n+N/4) + W_N^{k(N/2)} x(n+N/2) + W_N^{k3N/4} x(n+3N/4)]\, W_N^{nk}$$

The three twiddle factor coefficients can be expressed as follows:

$$(25) \qquad W_N^{k(N/4)} = (e^{-j2\pi/N})^{k(N/4)} = (e^{-j\pi/2})^k = (\cos(\pi/2) - j\sin(\pi/2))^k = (-j)^k$$

$$(26) \qquad W_N^{k(N/2)} = (e^{-j2\pi/N})^{k(N/2)} = (e^{-j\pi})^k = (\cos(\pi) - j\sin(\pi))^k = (-1)^k$$

$$(27) \qquad W_N^{k3N/4} = (e^{-j2\pi/N})^{k3N/4} = (e^{-j3\pi/2})^k = (\cos(3/2\pi) - j\sin(3\pi/2))^k = j^k$$

Equation (23) can thus be expressed as

$$(28) \qquad X(k) = \sum_{n=0}^{N/4-1} [\, x(n) + (-j)^k x(n+N/4) + (-1)^k x(n+N/2) + (j)^k x(n+3N/4)\,]\, W_N^{nk}$$

Four sub-sequences of the output (frequency) sequence are created by setting k=4r, k=4r+1, k=4r+2 and k=4r+3:

$$(29) \quad X(4r) = \sum_{n=0}^{N/4-1} [\, (\, x(n) + x(n+N/4) + x(n+N/2) + x(n+3N/4)\,)\, W_N^{0}\,]\, W_{N/4}^{nr}$$

$$(30)\ X(4r+1) = \sum_{n=0}^{N/4-1} [(x(n) - jx(n+N/4) - x(n+N/2) + jx(n+3N/4))W_N^{\,n}]\ W_{N/4}^{\,nr}$$

$$(31)\ X(4r+2) = \sum_{n=0}^{N/4-1} [(x(n) - x(n+N/4) + x(n+N/2) - x(n+3N/4))W_N^{\,2n}]\ W_{N/4}^{\,nr}$$

$$(32)\ X(4r+3) = \sum_{n=0}^{N/4-1} [(x(n) + jx(n+N/4) - x(n+N/2) - jx(n+3N/4))W_N^{\,3n}]\ W_{N/4}^{\,nr}$$

for $r = 0$ to $N/4-1$

$X(4r)$, $X(4r+1)$, $X(4r+2)$, and $X(4r+3)$ are $N/4$-point DFTs. Each of their $N/4$ points is a sum of four input samples ($x(n)$, $x(n+N/4)$, $x(n+N/2)$ and $x(n+3N/4)$), each multiplied by either $+1$, $-1$, $j$, or $-j$. The sum is multiplied by a twiddle factor ($W_N^{\,0}$, $W_N^{\,n}$, $W_N^{\,2n}$, or $W_N^{\,3n}$).

These four $N/4$-point DFTs together make up an N-point DFT. Each of these $N/4$-point DFTs is divided into four $N/16$-point DFTs. Each $N/16$ DFT is further divided into four $N/64$-point DFTs, and so on, until the final decimation produces four-point DFTs (groups of four one-point DFT equations). The four one-point DFT equations make up the butterfly calculation of the radix-4 FFT. A radix-4 butterfly is shown graphically in Figure 6.9.

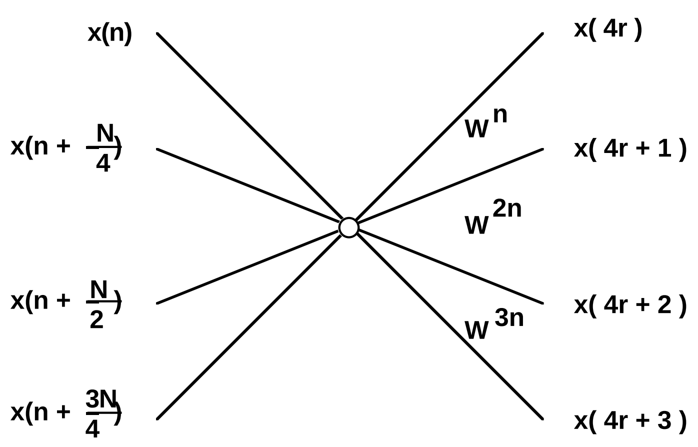


**Figure 6.9 Radix-4 DIF FFT Butterfly**

The output of each leg represents one of the four equations which are combined to make a four-point DFT. These four equations correspond to equations (29) through (32), for one point rather than N/4 points.

Each sample in the butterfly is complex. A butterfly flow graph with complex inputs and outputs is shown in Figure 6.10. The real part of each point is represented by *x*, and *y* represents the imaginary part. 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 6.10 as C + j(–S). C represents cosine and –S represents –sine.

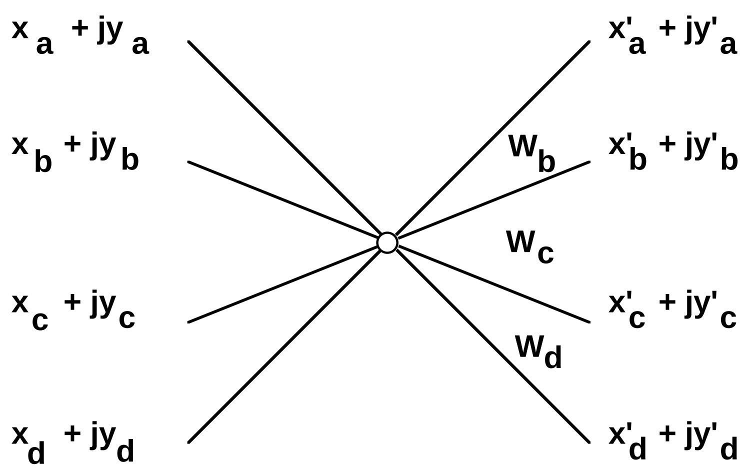


**Figure 6.10 Radix-4 DIF FFT Butterfly, Complex Data**

The real and imaginary output values for the radix-4 butterfly are given by equations (33) through (40).

(33) $x_a' = x_a + x_b + x_c + x_d$

(34) $y_a' = y_a + y_b + y_c + y_d$

(35) $x_b' = (x_a + y_b - x_c - y_d)C_b - (y_a - x_b - y_c + x_d)(-S_b)$

(36) $y_b' = (y_a - x_b - y_c + x_d)C_b + (x_a + y_b - x_c - y_d)(-S_b)$

(37) $x_c' = (x_a - x_b + x_c - x_d)C_c - (y_a - y_b + y_c - y_d)(-S_c)$

(38) $y_c' = (y_a - y_b + y_c - y_d)C_c + (x_a - x_b + x_c - x_d)(-S_c)$

(39) $x_d' = (x_a - y_b - x_c + y_d)C_d - (y_a + x_b - y_c - x_d)(-S_d)$

(40) $y_d' = (y_a + x_b - y_c - x_d)C_d + (x_a - y_b - x_c + y_d)(-S_d)$

A complete 64-point radix-4 FFT is shown in Figure 6.11, on the next page. As in the radix-2 FFT, butterflies are organized into groups and stages. The first stage has one group of 16 (N/4) butterflies, the next stage has four groups of four (N/16) butterflies, and the last stage has 16 groups of one butterfly. Notice that the twiddle factor values depend on the group and stage that are being performed. The table below summarizes the characteristics of an N-point radix-4 FFT.

| | | | | | |
|---|---|---|---|---|---|
| *Stage* | | 1 | 2 | 3 | $(\log_2 N)/2$ |
| *Butterfly Groups* | | 1 | 4 | 16 | N/4 |
| *Butterflies per Group* | | N/4 | N/16 | N/64 | 1 |
| *Dual-Node Spacing* | | N/4 | N/16 | N/64 | 1 |
| *Twiddle* | *leg1* | 0 | 0 | 0 | 0 |
| *Factor* | *leg2* | n | 4n | 16n | (N/4)n |
| *Exponents* | *leg3* | 2n | 8n | 32n | (N/2)n |
| | *leg4* | 3n | 12n | 48n | (3N/4)n |
| | | n=0 to N/4–1 | n=0 to N/16–1 | n=0 to N/32–1 | n=0 |

A 64-point radix-4 FFT has half as many stages (three instead of six) and half as many butterflies in each stage (16 instead of 32) as a 64-point radix-2 FFT.

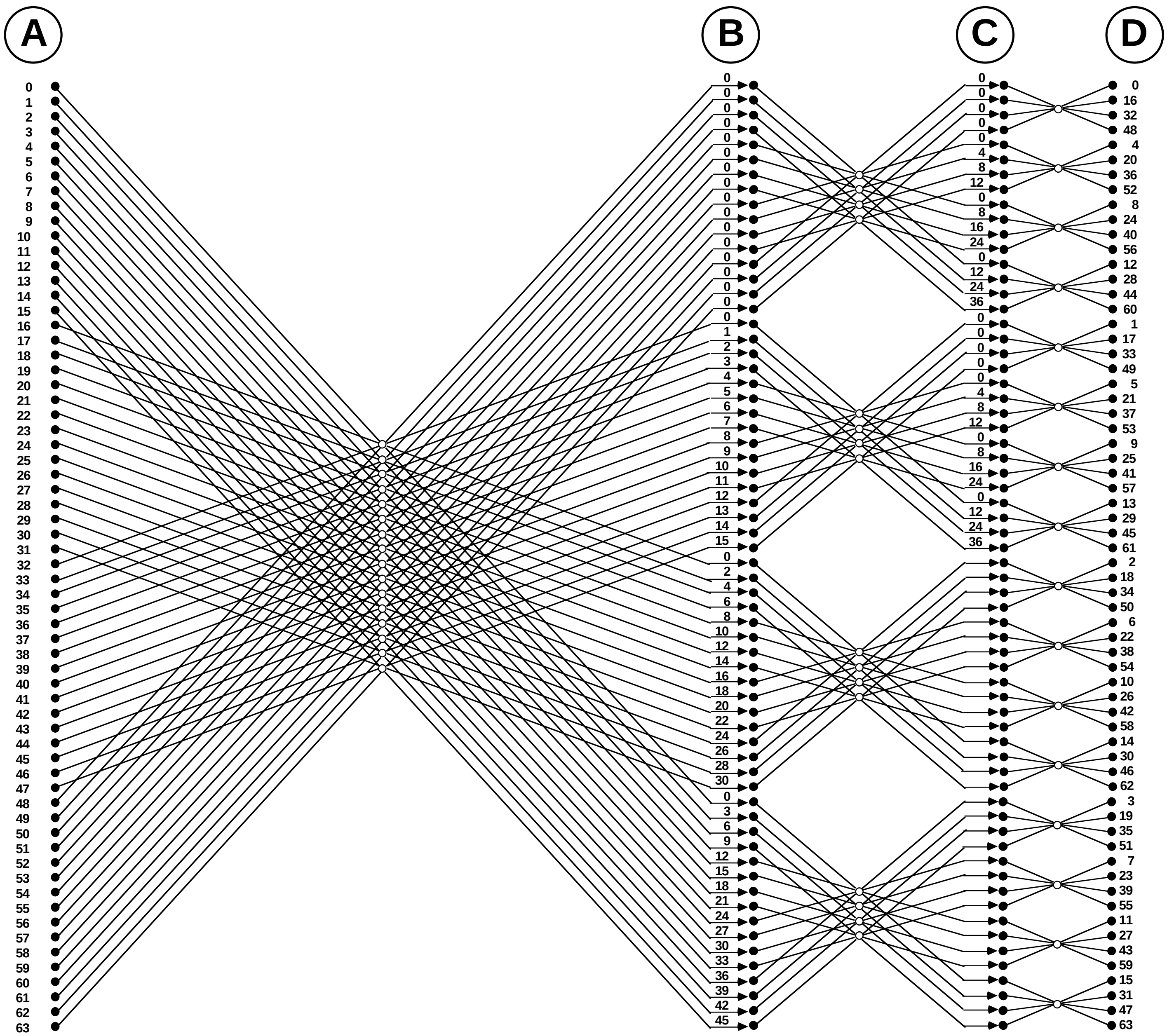


**Figure 6.11 Sixty-Four-Point Radix-4 DIF FFT**

Column a) indicates input sample; 44=x(44).
Column b) indicates twiddle factor exponent, stage one; $5=W_N^5$.
Column c) indicates twiddle factor exponent, stage two.
Column d) indicates output sample; 51=X(51).

### 6.5.2 Radix-4 Decimation-In-Frequency FFT Program

A flow chart for the radix-4 DIF FFT program is shown in Figure 6.12. The program flow is identical to that of the radix-2 DIF FFT except that the outputs are unscrambled by digit reversal instead of bit reversal.

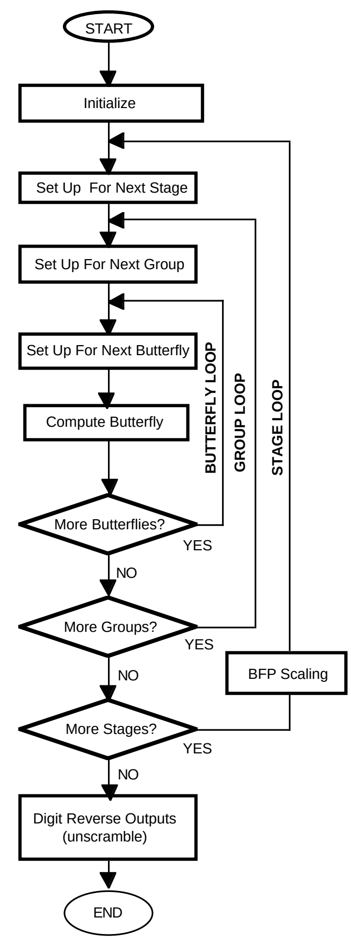


**Figure 6.12 Radix-4 DIF FFT Flow Chart**

The radix-4 DIF FFT routine uses three subroutines; the first computes the FFT, the second performs block floating-point scaling, and the third unscrambles the FFT results. The main routine (*rad4_main*) declares and initializes buffers and variables stored in external memory. It also calls the FFT and digit reversal subroutines. Three other modules contain the FFT, block floating-point scaling and digit reversal subroutines. The *rad4_main* and *rad4_fft* modules are described in this section. The block floating-point scaling and digit reversal routines are described later.

#### *6.5.2.1 Main Module*

The *rad4_main* module is shown in Listing 6.22. Constants *N*, *N_x_2*, *N_div_4*, and *N_div_2* are used throughout this module to specify buffer lengths as well as initial values for some variables. The in-place FFT calculation is performed in the *inplacedata* buffer. A small buffer called *padding* is placed at the end of the *inplacedata* buffer to allow memory accesses to exceed the buffer. The extra memory locations are necessary in a simulation because the ADSP-2100 Simulator does not allow undefined memory locations to be operated on; however, *padding* is not necessary in a real system.

The *input_data* buffer retains the initial FFT input data that is lost during the FFT calculation. This buffer allows you to look at the original input data after executing the program. However, *input_data* is also not needed in a real system.

The *digit_rev* subroutine unscrambles the FFT outputs and writes them in sequential order into *results*. The variables *groups*, *bflys_per_group*, *node_space*, and *blk_exponent* are declared to store stage characteristics and the block floating-point exponent, as in the radix-2 FFT routine.

Buffers *inplacedata*, *twids*, and *input_data* are initialized with data stored in external files. For example, *twids* is initialized with the external file *twids.dat*, which contains the twiddle factor values. Immediate zeros are placed in *padding*.

The variable *groups* is initialized to one and *bflys_per_group* to *N_div_4* because there is one group in the first stage of the FFT and N/4 butterflies in this first group. Node spacing for the radix-4 FFT in the first stage is N/4. However, because the *inplacedata* buffer is organized with real and imaginary data interleaved, the node spacing is doubled to N/2. Thus, the variable *node_space* is initialized to *N_div_2*.

The *rad4_fft* subroutine computes the FFT, and the *digit_rev* routine unscrambles the output using digit reversal. The TRAP instruction halts the ADSP-2100 when the FFT is complete.

### 6.5.2.2 DIF FFT Module

The conditional block floating-point radix-4 DIF FFT subroutine presented in this section consists of three nested loops. To simplify the explanation of this subroutine, each loop is described separately, starting with the innermost loop (the butterfly loop) and followed by the group loop and the stage loop. The entire subroutine is listed at the end of this section.

#### Butterfly Loop

The radix-4 butterfly equations (33-40) are repeated below.

(33) $x_a' = x_a + x_b + x_c + x_d$

(34) $y_a' = y_a + y_b + y_c + y_d$

(35) $x_b' = (x_a + y_b - x_c - y_d)C_b - (y_a - x_b - y_c + x_d)(-S_b)$

(36) $y_b' = (y_a - x_b - y_c + x_d)C_b + (x_a + y_b - x_c - y_d)(-S_b)$

(37) $x_c' = (x_a - x_b + x_c - x_d)C_c - (y_a - y_b + y_c - y_d)(-S_c)$

(38) $y_c' = (y_a - y_b + y_c - y_d)C_c + (x_a - x_b + x_c - x_d)(-S_c)$

(39) $x_d' = (x_a - y_b - x_c + y_d)C_d - (y_a + x_b - y_c - x_d)(-S_d)$

(40) $y_d' = (y_a + x_b - y_c - x_d)C_d + (x_a - y_b - x_c + y_d)(-S_d)$

The code segment to calculate these equations is shown in Listing 6.23. This code segment computes one radix-4 butterfly. The outputs ($x_a'$, $y_a'$, $x_b'$, $y_b'$, etc.) are written over the inputs ($x_a$, $y_a$, $x_b$, $y_b$, etc.) in the highlighted instructions. Each of the eight butterfly results is monitored for bit growth using the EXPADJ instruction and written to data memory in the same multifunction instruction. This code segment also sets up pointers and fetches the initial data for the next butterfly. The butterfly calculation is described in detail in the comments, and the instructions

```
.MODULE/ABS=4         rad4_main;

.CONST                N=1024,N_x_2=2048,    {Define constants for N-point FFT}
                      N_div_4=256,N_div_2=512;
.VAR/DM/RAM/ABS=0     inplacedata[N_x_2], padding[4];
                                            {Pad end of inplacedata so memory}
.VAR/DM/RAM           twids[N_x_2];         {accesses can exceed end of buffer}
.VAR/DM/RAM           outputdata[N_x_2];
.VAR/DM/RAM           input_data[N_x_2];
.VAR/DM/RAM           groups,bflys_per_group,
                      node_space,blk_exponent;

.INIT                 inplacedata: <inplacedata.dat>;
.INIT                 input_data: <inplacedata.dat>;
.INIT                 twids: <twids.dat>;
.INIT                 groups: 1;
.INIT                 bflys_per_group: N_div_4;
.INIT                 node_space: N_div_2;
.INIT                 blk_exponent: 0;
.INIT                 padding: 0,0,0,0;

.GLOBAL               inplacedata,twids, outputdata;
.GLOBAL               groups,bflys_per_group,node_space,blk_exponent;

.EXTERNAL             rad4_fft,digit_rev;

                      CALL rad4_fft;
                      CALL digit_rev;
                      TRAP;                 {Stop program execution}
.ENDMOD;
```

**Listing 6.22 Main Module, Radix-4 DIF FFT**

that check for bit growth and write the butterfly results to data memory are boldface.

The input and output parameters of this code segment are shown below.

| *Initial Conditions* | *Final Conditions* |
|---|---|
| I0 --> $x_a$ | I0 --> next $x_a$ |
| I1 --> $x_b$ | I1 --> next $x_b$ |
| I2 --> $y_c$ | I2 --> next $y_c$ |
| I3 --> $x_d$ | I3 --> next $x_d$ |
| I4 --> $C_b$ | I4 --> next $C_b$ |
| I5 --> $S_c$ | I5 --> next $S_c$ |
| I6 --> $C_d$ | I6 --> next $C_d$ |
| M0 = 0 | AX0 = next $x_a$ |
| M1 = 1 | AY0 = next $x_c$ |
| M3 = –1 | MY0 = next $C_c$ |
| CNTR = butterfly counter | CNTR = butterfly counter – 1 |
| M4 = 1 | |
| M5 = *groups* x 2 – 1 | |
| M6 = *groups* x 4 – 1 | |
| M7 = *groups* x 6 – 1 | |
| AX0 = $x_a$ | |
| AY0 = $x_c$ | |
| MY0 = $C_c$ | |

```
AF=AX0+AY0,AX1=DM(I1,M1);          {AF=xa+xc; AX1=xb; I1 --> yb}
AR=AF-AX1,AY1=DM(I3,M1);           {AR=xa+xc-xb; AY1=xd; I3 --> yd}
AR=AR-AY1,SR1=DM(I1,M3);           {AR=xa-xb+xc-xd; SR1=yb; I1 --> xb}
MR=AR*MY0(SS),SR0=DM(I3,M3);       {MR=(xa-xb+xc-xd)Cc; SR0=yd; I3 --> xd}
MX0=AR,AR=AX1+AF;                  {AR=xa+xb+xc; MX0=(xa-xb+xc-xd)Cc}
AR=AR+AY1;                         {AR=xa+xb+xc+xd}
SB=EXPADJ AR,DM(I0,M1)=AR;         {xa´=xa+xb+xc+xd; I0 --> ya}
AF=AX0-AY0,AX0=DM(I0,M0);          {AF=xa-xc; AX0=ya; I0 --> ya}
AR=SR1+AF,AY0=SR0;                 {AR=xa+yb-xc; AY0=yd}
AF=AF-SR1;                         {AF=xa-yb-xc}
AR=AR-AY0,AY0=DM(I2,M3);           {AR=xa+yb-xc-yd; AY0=yc; I2 --> xc}
MX1=AR,AR=SR0+AF;                  {AR=xa-yb-xc+yd; MX1=xa+yb-xc-yd}
AF=AX0+AY0,DM(I3,M1)=AR;           {AR=ya+yc; location of xd=xa-yb-xc+yd}
                                   {I3 --> yd}
AY0=DM(I3,M3),AR=SR1+AF;           {AR=ya+yb+yc; AY0=yd; I3 --> xd}
AR=AR+AY0,MY1=DM(I5,M6);           {AR=ya+yb+yc+yd; MY1=(-Sc); I5 --> next Cc}
SB=EXPADJ AR,DM(I0,M1)=AR;         {ya´=ya+yb-yc+yd; I0 --> next xa}
AF=AF-SR1;                         {AF=ya-yb+yc}
AR=AF-SR0;                         {AR=ya-yb+yc-yd}
MR=MR-AR*MY1(SS);                  {MR=(xa-xb+xc-xd)Cc - (ya-yb+yc-yd)(-Sc)}
SB=EXPADJ MR1,DM(I2,M1)=MR1;       {xc´=(xa-xb+xc-xd)Cc - (ya-yb+yc-yd)(-Sc)}
                                   {I2 --> yc}
MR=AR*MY0(SS);                     {MR=(ya-yb+yc-yd)Cc}
MR=MR+MX0*MY1(SS),AY0=DM(I2,M0);   {MR=(ya-yb+yc-yd)Cc + (xa-xb+xc-xd)(-Sc)}
                                   {AY0=yc; I2 --> yc}
SB=EXPADJ MR1,DM(I2,M1)=MR1;       {yc´=(ya-yb+yc-yd)Cc + (xa-xb+xc-xd)(-Sc)}
                                   {I2 --> next xc}
AF=AX0-AY0,MY1=DM(I4,M4);          {AF=ya-yc; MY1=Cb; I4 -->(-Sb)}
AR=AF-AX1,AX0=DM(I0,M0);           {AR=ya-xb-yc; AX0=ya; I1 --> ya}
AR=AR+AY1,AY0=DM(I2,M1);           {AR=ya-xb-yc+xd; AY0=yc; I2 --> next xc}
MR=MX1*MY1(SS),MY0=DM(I4,M5);      {MR=(xa+yb-xc-yd)Cb; MY0=Sb; I4 --> next Cb}
MR=MR-AR*MY0(SS);                  {MR=(xa+yb-xc-yd)Cb - (ya-xb-yc+xd)(-Sb)}
SB=EXPADJ MR1,DM(I1,M1)=MR1;       {xb´=(xa+yb-xc-yd)Cb - (ya-xb-yc+xd)(-Sb)}
                                   {I1 --> yb}
MR=AR*MY1(SS);                     {MR=(ya-xb-yc+xd)Cb}
MR=MR+MX1*MY0(SS),MX1=DM(I3,M0);   {MR=(ya-xb-yc+xd)Cb + (xa+yb-xc-yd)(-Sb)}
                                   {MX1=xa-yb-xc+yd; I3 --> xd}
SB=EXPADJ MR1,DM(I1,M1)=MR1;       {yb´=(ya-xb-yc+xd)Cb + (xa+yb-xc-yd)(-Sb)}
                                   {I1 --> next xb}
AR=AX1+AF,MY0=DM(I6,M4);           {AR=ya+xb-yc; MY0=Cd; I6 -->-Sd}
AR=AR-AY1,MY1=DM(I6,M7);           {AR=ya+xb-yc-xd; MY1=-Sd; I6 -->Cd}
MR=MX1*MY0(SS);                    {MR=(xa-yb-xc+yd)Cd}
MR=MR-AR*MY1(SS);                  {MR=(xa-yb-xc+yd)Cd - (ya+xb-yc-xd)(-Sd)}
SB=EXPADJ MR1,DM(I3,M1)=MR1;       {xd´=(xa-yb-xc+yd)Cd - (ya+xb-yc-xd)(-Sd)}
                                   {I3 --> yd}
MR=AR*MY0(SS),MY0=DM(I5,M4);       {MR=(ya+xb-yc-xd)Cd; MY0=next Cc}
                                   {I5 --> next (-Sc)}
MR=MR+MX1*MY1(SS);                 {MR=(ya+xb-yc-xd)Cd + (xa-yb-xc+yd)(-Sd)}
SB=EXPADJ MR1,DM(I3,M1)=MR1;       {yd´=(ya+xb-yc-xd)Cd +(xa-yb-xc+yd)(-Sd)}
                                   {I3 --> next xd}
```

**Listing 6.23 Radix-4 DIF FFT Butterfly, Conditional Block Floating-Point Scaling**

### Group Loop

The group loop is shown in Listing 6.24. This code segment sets up and computes one group of butterflies. Because each leg of the first butterfly in all groups in the FFT has the twiddle factor $W^0$, twiddle-factor pointers are initialized to point to the real part of $W^0$. Next, the butterfly loop is set up by initializing the butterfly loop counter and fetching initial data values ($x_a$, $y_c$ and $C_c$). Notice that these are the initial conditions for the butterfly loop.

After all the butterflies in the group are calculated, pointers used in the butterfly are updated to point to $x_a$, $x_b$, $x_c$, and $x_d$ for the first butterfly in the next group. For example, I0 points to the first $x_a$ in the next group, I1 to the first $x_b$, etc. The group loop is executed *groups* times (the number of groups in a stage).

The input and output parameters of this code segment are as follows:

| *Initial Conditions* | *Final Conditions* |
|---|---|
| I0 --> $x_a$ | I0 --> first $x_a$ of next group |
| I1 --> $x_b$ | I1 --> first $x_b$ of next group |
| I2 --> $x_c$ | I2 --> first $x_c$ of next group |
| I3 --> $x_d$ | I3 --> first $x_d$ of next group |
| M0 = 0 | I4 --> invalid location for twiddle factor |
| M1 = 1 | I5 --> invalid location for twiddle factor |
| M2 = 3 x node_space | I6 --> invalid location for twiddle factor |
| M3 = –1 | CNTR = group count – 1 |
| M4 = 1 | |
| CNTR = group count | |

```
            I4=^twids;                  {I4 --> Cb}
            I5=I4;                      {I5 --> Cc}
            I6=I5;                      {I6 --> Cd}
            CNTR=DM(bflys_per_group);   {Initialize butterfly counter}
            AX0=DM(I0,M0);              {AX0=xa; I0 --> xa}
            AY0=DM(I2,M1);              {AY0=xc; I2 --> yc}
            MY0=DM(I5,M4);              {MY0=Cc; I5 --> Sc}
            DO bfly_loop UNTIL CE;

bfly_loop:  {Calculate All Butterflies}

            MODIFY(I0,M2);              {I0 --> first xa of next group}
            MODIFY(I1,M2);              {I1 --> first xb of next group}
            MODIFY(I2,M3);
            MODIFY(I2,M2);              {I2 --> first xc of next group}
            MODIFY(I3,M2);              {I3 --> first xd of next group}
```

**Listing 6.24 Radix-4 DIF FFT Group Loop**

## Stage Loop

The stage characteristics of the FFT are controlled by the stage loop. For example, the stage loop controls the number of groups and the number of butterflies in each group. The stage loop code segment is shown in Listing 6.25. This code sets up and calculates all groups of butterflies in a stage and updates parameters for next stage.

The radix-4 butterfly data can potentially grow three bits from butterfly input to output (the worst case growth factor is 5.6). Therefore, each input value to the FFT contains three guard bits to prevent overflow. SB is initialized to –3, so any bit growth into the guard bits can be monitored. If bit growth occurs, it is compensated for in the block floating-point subroutine that is called after each stage is computed.

The variable *groups* is loaded into SI and used to calculate various stage parameters. These include *groups*x2–1, the leg b twiddle factor modifier, *groups*x4–1, the leg c twiddle factor modifier, and *groups*x6–1, the leg d modifier. Pointers are set to $x_a$, $x_b$, $x_c$, and $x_d$, the inputs to the first butterfly in the stage. The group loop counter is initialized and M2, which is used to update butterfly data pointers at the start of a new group, is set to three times the node spacing.

In the group loop, all groups in the stage are computed. After the groups are computed, the subroutine *bfp_adjust* is called to perform block floating-point scaling by checking for bit growth in the stage output data and adjusting all of the data in the block accordingly.

After the output data is scaled, parameters are adjusted for the next stage; *groups* is updated to *groups*x4, *node_space* to *node_space*/4, and *bflys_per_group* to *bflys_per_group*/4. The stage loop is repeated ($\log_2$N)/2 times (the number of stages in the FFT).

The input and output parameters for this code segment are as follows:

| *Initial Conditions* | *Final Conditions* |
|---|---|
| *groups* = # groups/stage | *groups* =*groups* x 4 |
| *node_space* = node spacing for stage | *node_space* = *node_space* /4 |
| *bflys_per_group* = # butterflies/group | *bflys_per_group* =*bflys_per_group* /4 |
| *inplacedata*=stage input data | *inplacedata*=stage output data |
| CNTR = stage count | CNTR = stage count – 1 |
| | SB = –(number of guard bits remaining in data word(s) with largest magnitude) |
| | SI = # groups/stage |
| | I0 ->invalid location for data sample |
| | I1 ->invalid location for data sample |
| | I2 ->invalid location for data sample |
| | I3 ->invalid location for data sample |
| | M2 = *node_space* x 3 |
| | M5 = *groups* x 2 – 1 |
| | M6 = *groups* x 4 – 1 |
| | M7 = *groups* x 6 – 1 |

```
            SB=-3;                       {SB detects growth into 3 guard bits}
            SI=DM(groups);               {SI=groups}
            SR=ASHIFT SI BY 1(HI);       {SR1=groups × 2}
            AY1=SR1;                     {AY1=groups × 2}
            AR=AY1-1;                    {AR=groups × 2 - 1}
            M5=AR;                       {M5=groups × 2 - 1}
            SR=ASHIFT SR1 BY 1(HI);      {SR1=groups × 4}
            AY1=SR1;                     {AY1=groups × 4}
            AR=AY1-1;                    {AR=groups × 4 - 1}
            M6=AR;                       {M6=groups × 4 - 1}
            AY0=SI;                      {AY0=groups}
            AR=AR+AY0;                   {AR=groups × 5 - 1}
            AR=AR+AY0;                   {AR=groups × 6 - 1}
            M7=AR;                       {M7=groups × 6 - 1}
            M2=DM(node_space);           {M2=node_space}
            I0=^inplacedata;             {I0 --> xa}
            I1=I0;
            MODIFY(I1,M2);               {I1 --> xb}
            I2=I1;
            MODIFY(I2,M2);               {I2 --> xc}
            I3=I2;
            MODIFY(I3,M2);               {I3 --> xd}
            CNTR=SI;                     {Initialize group counter}
            AY0=DM(node_space);
            M2=I3;                       {M2=node_space × 3}
            DO group_loop UNTIL CE;

group_loop: {Calculate All Groups in a Stage}

            CALL bfp_adjust;             {Check for bit growth}
            SI=DM(groups);               {SI=groups}
            SR=ASHIFT SI BY 2(HI);       {SR1=groups ×4 }
            DM(groups)=SR1;              {group count, next stage}
            SI=DM(bflys_per_group);      {SI=bflys_per_group}
            SR=ASHIFT SI BY -1(HI);      {SR1=bflys_per_group ÷ 2}
            DM(node_space)=SR1;          {node spacing, next stage}
            SR=ASHIFT SI BY -1(HI);      {SR1=node_space ÷ 2}
            DM(bflys_per_group)=SR1;     {butterfly count, next stage}
```

**Listing 6.25 Radix-4 DIF FFT Stage Loop**

## Radix-4 DIF FFT Subroutine

The butterfly, group, and stage loop code segments are combined into the entire radix-4 DIF FFT subroutine, which is shown in Listing 6.26. Note that length and modify registers that retain the same value throughout the routine are initialized outside the stage loop. The stage loop counter is initialized to the number of stages in an N-point FFT (*$log_2N\_div\_2$*). Instructions that write butterfly results to memory are boldface.

```
.MODULE     radix_4_dif_fft;        {Declare and name module}

.CONST      log2N_div_2=5;          {Initial stage count}

.ENTRY      rad4_fft;

.EXTERNAL   groups,node_space,bflys_per_group;
.EXTERNAL   inplacedata,twids,bfp_adjust;

rad4_fft:   CNTR=log2N_div_2;       {Initialize stage counter}
            M0=0;                   {Set constant modifiers, length registers}
            M1=1;
            M3=-1;
            M4=1;
            L0=0;
            L1=0;
            L2=0;
            L3=0;
            L4=0;
            L5=0;
            L6=0;
            L7=0;
            DO stage_loop UNTIL CE;           {Compute all stages}
               SB=-4;                         {Detects bit growth into 4 MSBs}
               SI=DM(groups);                 {SI=groups}
               SR=ASHIFT SI BY 1(HI);         {SR1=groups × 2}
               AY1=SR1;                       {AY1=groups × 2}
               AR=AY1-1;                      {AR=groups × 2 - 1}
               M5=AR;                         {M5=groups × 2 - 1}
               SR=ASHIFT SR1 BY 1(HI);        {SR1=groups × 4}
               AY1=SR1;                       {AY1=groups × 4}
               AR=AY1-1;                      {AR=groups × 4 - 1}
               M6=AR;                         {M6=groups × 4 - 1}
               AY0=SI;                        {AY0=groups}
               AR=AR+AY0;                     {AR=groups × 5 - 1}
               AR=AR+AY0;                     {AR=groups × 6 - 1}
               M7=AR;                         {M7=groups × 6 - 1}
               M2=DM(node_space);             {M2=node_space}
               I0=^inplacedata;               {I0 -->xa}
               I1=I0;
               MODIFY(I1,M2);                 {I1 -->xb}
               I2=I1;
               MODIFY(I2,M2);                 {I2 -->xc}
               I3=I2;
               MODIFY(I3,M2);                 {I3 -->xd}
               CNTR=SI;                       {Initialize group counter}
               AY0=DM(node_space);
               M2=I3;                         {M2=node_space × 3}
               DO group_loop UNTIL CE;        {Compute all groups in stage}
```

```
            I4=^twids;                  {I4 -->Cb}
            I5=I4;                      {I5 -->Cc}
            I6=I5;                      {I6 -->Cd}
            CNTR=DM(bflys_per_group);   {Initialize butterfly counter}
            AX0=DM(I0,M0);              {AX0=xa, I0 -->xa}
            AY0=DM(I2,M1);              {AY0=xc, I2 -->yc}
            MY0=DM(I5,M4);              {MY0=Cc, I5 -->(-Sc)}
            DO bfly_loop UNTIL CE;      {Compute all butterflies in grp}
               AF=AX0+AY0,AX1=DM(I1,M1);
               AR=AF-AX1,AY1=DM(I3,M1);
               AR=AR-AY1,SR1=DM(I1,M3);
               MR=AR*MY0(SS),SR0=DM(I3,M3);
               MX0=AR,AR=AX1+AF;
               AR=AR+AY1;
               SB=EXPADJ AR,DM(I0,M1)=AR;         {xa´=xa+xb+xc+xd}
               AF=AX0+AY0,AX0=DM(I0,M0);
               AR=SR1+AF,AY0=SR0;
               AF=AF-SR1;
               AR=AR-AY0,AY0=DM(I2,M3);
               MX1=AR,AR=SR0+AF;
               AF=AX0+AY0,DM(I3,M1)=AR;
               AY0=DM(I3,M3),AR=SR1+AF;
               AR=AR+AY0,MY1=DM(I5,M6);
               SB=EXPADJ AR,DM(I0,M1)=AR;         {ya´=ya+yb+yc+yd}
               AF=AF-SR1;
               AR=AF-SR0;
               MR=MR-AR*MY1(SS);
               SB=EXPADJ MR1,DM(I2,M1)=MR1;       {xc´=(xa-xb+xc-xd)Cc}
               MR=AR*MY0(SS);                     {-(ya-yb+yc-yd)(-Sc)}
               MR=MR+MX0*MY1(SS),AY0=DM(I2,M0);
               SB=EXPADJ MR1,DM(I2,M1)=MR1;       {yc´=(ya-yb+yc-yd)Cc}
               AF=AX0-AY0,MY1=DM(I4,M4);          {+ (xa-xb+xc-xd)(-Sc)}
               AR=AF-AX1,AX0=DM(I0,M0);
               AR=AR+AY1,AY0=DM(I2,M1);
               MR=MX1*MY1(SS),MY0=DM(I4,M5);
               MR=MR-AR*MY0(SS);
               SB=EXPADJ MR1,DM(I1,M1)=MR1;       {xb´=(xa+yb-xc-yd)Cb}
               MR=AR*MY1(SS);                     {-(ya-xb-yc+yd)(-Sb)}
               MR=MR+MX1*MY0(SS),MX1=DM(I3,M0);
               SB=EXPADJ MR1,DM(I1,M1)=MR1;       {yb´=(ya-xb-yc+xd)Cb}
               AR=AX1+AF,MY0=DM(I6,M4);           {+ (xa+yb-xc-yd)(-Sb)}
               AR=AR-AY1,MY1=DM(I6,M7);
               MR=MX1*MY0(SS);
               MR=MR-AR*MY1(SS);
               SB=EXPADJ MR1,DM(I3,M1)=MR1;       {xd´=(xa-yb-xc+yd)Cd}
               MR=AR*MY0(SS),MY0=DM(I5,M4);       {- (ya+xb-yc-xd)(-Sd)}
               MR=MR+MX1*MY1(SS);
bfly_loop:     SB=EXPADJ MR1,DM(I3,M1)=MR1;       {yd´= (ya+xb-yc-xd)Cd}
                                                  {+ (xa-yb-xc+yd)(-Sd)}
```

*(listing continues on next page)*

```
               DO bfly_loop UNTIL CE;              {Compute all butterflies in grp}
                  AF=AX0+AY0,AX1=DM(I1,M1);
                  AR=AF-AX1,AY1=DM(I3,M1);
                  AR=AR-AY1,SR1=DM(I1,M3);
                  MR=AR*MY0(SS),SR0=DM(I3,M3);
                  MX0=AR,AR=AX1+AF;
                  AR=AR+AY1;
                  SB=EXPADJ AR,DM(I0,M1)=AR;       {xa´=xa+xb+xc+xd}
                  AF=AX0+AY0,AX0=DM(I0,M0);
                  AR=SR1+AF,AY0=SR0;
                  AF=AF-SR1;
                  AR=AR-AY0,AY0=DM(I2,M3);
                  MX1=AR,AR=SR0+AF;
                  AF=AX0+AY0,DM(I3,M1)=AR;
                  AY0=DM(I3,M3),AR=SR1+AF;
                  AR=AR+AY0,MY1=DM(I5,M6);
                  SB=EXPADJ AR,DM(I0,M1)=AR;       {ya´=ya+yb+yc+yd}
                  AF=AF-SR1;
                  AR=AF-SR0;
                  MR=MR-AR*MY1(SS);
                  SB=EXPADJ MR1,DM(I2,M1)=MR1;     {xc´=(xa-xb+xc-xd)Cc}
                  MR=AR*MY0(SS);                   {-(ya-yb+yc-yd)(-Sc)}
                  MR=MR+MX0*MY1(SS),AY0=DM(I2,M0);
                  SB=EXPADJ MR1,DM(I2,M1)=MR1;     {yc´=(ya-yb+yc-yd)Cc}
                  AF=AX0-AY0,MY1=DM(I4,M4);        {+ (xa-xb+xc-xd)(-Sc)}
                  AR=AF-AX1,AX0=DM(I0,M0);
                  AR=AR+AY1,AY0=DM(I2,M1);
                  MR=MX1*MY1(SS),MY0=DM(I4,M5);
                  MR=MR-AR*MY0(SS);
                  SB=EXPADJ MR1,DM(I1,M1)=MR1;     {xb´=(xa+yb-xc-yd)Cb}
                  MR=AR*MY1(SS);                   {-(ya-xb-yc+yd)(-Sb)}
                  MR=MR+MX1*MY0(SS),MX1=DM(I3,M0);
                  SB=EXPADJ MR1,DM(I1,M1)=MR1;     {yb´=(ya-xb-yc+xd)Cb}
                  AR=AX1+AF,MY0=DM(I6,M4);         {+ (xa+yb-xc-yd)(-Sb)}
                  AR=AR-AY1,MY1=DM(I6,M7);
                  MR=MX1*MY0(SS);
                  MR=MR-AR*MY1(SS);
                  SB=EXPADJ MR1,DM(I3,M1)=MR1;     {xd´=(xa-yb-xc+yd)Cd}
                  MR=AR*MY0(SS),MY0=DM(I5,M4);     {- (ya+xb-yc-xd)(-Sd)}
                  MR=MR+MX1*MY1(SS);
bfly_loop:        SB=EXPADJ MR1,DM(I3,M1)=MR1;     {yd´= (ya+xb-yc-xd)Cd}
                                                   {+ (xa-yb-xc+yd)(-Sd)}
               MODIFY(I0,M2);             {I0 -->1st xa of next group}
               MODIFY(I1,M2);             {I1 -->1st xb of next group}
               MODIFY(I2,M3);
               MODIFY(I2,M2);             {I2 -->1st xc of next group}
group_loop:    MODIFY(I3,M2);             {I3 -->1st xd of next group}
            CALL bfp_adjust;              {Check for bit growth}
            SI=DM(groups);                {SI=groups}
            SR=ASHIFT SI BY 2(HI);        {SR1=groups × 4}
```

```
                DM(groups)=SR1;                 {Group count, next stage}
                SI=DM(bflys_per_group);         {SI=bflys_per_group}
                SR=ASHIFT SI BY -1(HI);         {SR1=bflys_per_group ÷ 2}
                DM(node_space)=SR1;             {Node spacing, next stage}
                SR=ASHIFT SI BY -1(HI);         {SR1=node_space ÷ 2}
stage_loop:     DM(bflys_per_group)=SR1;        {Butterfly count, next stage}
            RTS;
.ENDMOD;
```

**Listing 6.26 Radix-4 DIF FFT Routine, Conditional Block Floating-Point Scaling**

A routine similar to the *dit_radix-2_bfp_adjust* routine is used to monitor bit growth in the radix-4 FFT. Because a radix-4 butterfly can cause data to grow by three bits from input to output, the radix-2 block floating-point routine is modified to adjust for three bits instead of two. The *dif_radix-4_bfp_adjust* routine is shown in Listing 6.27. This routine performs block floating-point adjustment on the radix-4 DIF FFT stage output.

The *dif_radix-4_bfp_adjust* routine checks for growth of three bits as well as for zero, one and two bits. This routine shifts data (by one, two or three bits to the right) using the shifter. As described above, shifting right by multiplication allows rounding of the shifted bit(s). However, multiplication is not always possible. This routine illustrates the use of the shifter.

```
.MODULE     dif_radix_4_bfp_adjust;

{           Calling Parameters
               Radix-4 DIF FFT stage results in inplacedata

            Return Values
               inplacedata adjusted for bit growth

            Altered Registers
               I0,I1,AX0,AY0,AR,SE,SI,SR

            Altered Memory
               inplacedata, blk_exponent
}

.CONST      N_x_2=2048;

.EXTERNAL   inplacedata, blk_exponent;

.ENTRY      bfp_adjust;
```

***(listing continues on next page)***

```
bfp_adjust: AY0=CNTR;
            AR=AY0-1;
            IF EQ RTS;                 {If last stage, return}
            AY0=-3;
            AX0=SB;
            AR=AX0-AY0;
            IF EQ RTS;                 {If SB=-3, no bit growth, return}
            AY0=-2;
            SE=-1;
            I0=^inplacedata;           {I0=read pointer}
            I1=^inplacedata;           {I1=write pointer}
            AR=AX0-AY0,SI=DM(I0,M1);   {Check SB, get 1st sample}
            IF EQ JUMP strt_shift;     {If SB=-2, shift block right 1 bit}
            AY0=-1;
            SE=-2;
            AR=AX0-AY0;
            IF EQ JUMP strt_shift;     {If SB=-1, shift block right 2 bits}
            SE=-3;                     {Otherwise, SB=0, shift right 3 bits}
strt_shift: CNTR=N_x_2-1;
            AY0=SE;
            DO shift_loop UNTIL CE;
               SR=ASHIFT SI(LO),SI=DM(I0,M1);  {SR=shifted data, SI=next data}
shift_loop:    DM(I1,M1)=SR0;                  {Unshifted data=shifted data}
            SR=ASHIFT SI(LO);          {Shift last data word}
            AX0=DM(blk_exponent);      {Update block exponent and}
            DM(I1,M1)=SR0,AR=AX0-AY0;  {store last shifted sample}
            DM(blk_exponent)=AR;
            RTS;
.ENDMOD;
```

**Listing 6.27 Radix-4 Block Floating-Point Scaling Routine**

### 6.5.3 Digit Reversal

Whereas bit reversal reverses the order of bits in binary (base 2) numbers, digit reversal reverses the order of digits in quarternary (base 4) numbers. Every two bits in the binary number system correspond to one digit in the quarternary number system. (For example, binary 1110 = quarternary 32.) The quarternary system is illustrated below for decimal numbers 0 through 15.

| *Decimal* | *Binary* | *Quarternary* |
|---|---|---|
| 0 | 0000 | 00 |
| 1 | 0001 | 01 |
| 2 | 0010 | 02 |
| 3 | 0011 | 03 |
| 4 | 0100 | 10 |
| 5 | 0101 | 11 |
| 6 | 0110 | 12 |
| 7 | 0111 | 13 |
| 8 | 1000 | 20 |
| 9 | 1001 | 21 |
| 10 | 1010 | 22 |
| 11 | 1011 | 23 |
| 12 | 1100 | 30 |
| 13 | 1101 | 31 |
| 14 | 1110 | 32 |
| 15 | 1111 | 33 |

The radix-4 DIF FFT successively divides a sequence into four sub-sequences, resulting in an output sequence in digit-reversed order. A digit-reversed sequence is unscrambled by digit-reversing the data positions. For example, position 12 in quarternary (six in decimal) becomes position 21 in quarternary (nine in decimal) after digit reversal. Therefore, data in position six is moved to position nine when the digit-reversed sequence is unscrambled. The digit-reversed positions for a 16-point sequence (samples X(0) through X(15)) are shown on the next page.

| *Sample, Sequential Order* | *Sequential Location* decimal | quarternary | *Digit-Reversed Location* decimal | quarternary | *Sample, Digit-Reversed Order* |
|---|---|---|---|---|---|
| X(0) | 0 | 00 | 0 | 00 | X(0) |
| X(1) | 1 | 01 | 4 | 10 | X(4) |
| X(2) | 2 | 02 | 8 | 20 | X(8) |
| X(3) | 3 | 03 | 12 | 30 | X(12) |
| X(4) | 4 | 10 | 1 | 01 | X(1) |
| X(5) | 5 | 11 | 5 | 11 | X(5) |
| X(6) | 6 | 12 | 9 | 21 | X(9) |
| X(7) | 7 | 13 | 13 | 31 | X(13) |
| X(8) | 8 | 20 | 2 | 02 | X(2) |
| X(9) | 9 | 21 | 6 | 12 | X(6) |
| X(10) | 10 | 22 | 10 | 22 | X(10) |
| X(11) | 11 | 23 | 14 | 32 | X(14) |
| X(12) | 12 | 30 | 3 | 03 | X(3) |
| X(13) | 13 | 31 | 7 | 13 | X(7) |
| X(14) | 14 | 32 | 11 | 23 | X(11) |
| X(15) | 15 | 33 | 15 | 33 | X(15) |

In an N-point radix-4 FFT, only the number of digits needed to represent N locations are reversed. Two digits are needed for a 16-point FFT, three digits for a 64-point FFT, and five digits for a 1024-point FFT.

The digit reversal subroutine that unscrambles the output sequence for the radix-4 DIF FFT is described later in the next section. This routine works with the optimized radix-4 FFT. A similar routine can be used for the unoptimized program.