One-Dimensional FFTs ■ 6

6.1 OVERVIEW

In many applications, frequency analysis is necessary and desirable. Applications ranging from radar to spread-spectrum communications employ the Fourier transform for spectral analysis and frequency domain processing.

The discrete Fourier transform (DFT) is the discrete-time equivalent of the continuous-time Fourier transform. Whereas the continuous-time Fourier transform operates on a continuous time signal x(t), the DFT operates on samples of x(t). If X(f) is the continuous Fourier transform of x(t), then the DFT of x(t) (sampled) is a sequence of samples of X(f), equally spaced in frequency. Equation (1) computes the DFT (Oppenheim and Schafer, 1975). X(k) is the discrete or sampled Fourier transform, and x(n) is a sequence of samples of the input signal, x(t). The term W_N, defined as $e^{-j2\pi/N}$, corresponds to the term $e^{-j2\pi ft}$ used to compute the continuous Fourier transform. Various powers of W_N are used for each multiplication in the DFT calculation.

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

where $W_N = e^{-j2\pi/N}$

A complex summation of N complex multiplications is required for each of the N output samples. In all, N^2 complex multiplications and N^2 complex additions compute an N-point DFT. The time burden created by this large number of calculations limits the usefulness of the DFT in many applications. For this reason, tremendous effort was devoted to developing more efficient ways of computing the DFT. This effort produced the fast Fourier transform (FFT). The FFT uses mathematical shortcuts to reduce the number of calculations the DFT requires.

This chapter describes three variations of the FFT algorithm: the radix-2 decimation-in-time FFT, the radix-2 decimation-in-frequency FFT and the

6 One-Dimensional FFTs

radix-4 decimation-in-frequency FFT. Optimization, to make the code run faster, and block floating-point scaling, to increase data precision, are also addressed. In addition, bit- and digit-reversal and windowing, operations related to the FFT, are described. An FFT in two dimensions is presented in the next chapter.

6.2 RADIX-2 FAST FOURIER TRANSFORMS

Suppose an N-point DFT is accomplished by performing two N/2-point DFTs and combining the outputs of the smaller DFTs to give the same output as the original DFT. The original DFT requires N^2 complex multiplications and N^2 complex additions. Each DFT of N/2 input samples requires $(N/2)^2 = N^2/4$ multiplications and additions, a total of $N^2/2$ calculations for the complete DFT. Dividing the DFT into two smaller DFTs reduces the number of computations by 50 percent. Each of these smaller DFTs can be divided in half, yielding four N/4-point DFTs. If we continue dividing the N-point DFT calculation into smaller DFTs until we have only two-point DFTs, the total number of complex multiplications and additions is reduced to $N\log_2 N$. For example, a 1024-point DFT requires over a million complex additions and multiplications. A 1024-point DFT divided down into two-point DFTs needs fewer than ten thousand complex additions and multiplications, a reduction of over 99 percent.

Dividing the DFT into smaller DFTs is the basis of the FFT. A radix-2 FFT divides the DFT into two smaller DFTs, each of which is divided into two smaller DFTs, and so on, resulting in a combination of two-point DFTs. In a similar fashion, a radix-4 FFT divides the DFT into four smaller DFTs, each of which is divided into four smaller DFTs, and so on. Two types of radix-2 FFTs are described in this section: the decimation-in-time FFT and the decimation-in-frequency FFT. The radix-4 decimation-in-frequency FFT is described in a later section.

6.2.1 Radix-2 Decimation-In-Time FFT Algorithm

The decimation-in-time (DIT) FFT divides the input (time) sequence into two groups, one of even samples and the other of odd samples. N/2-point DFTs are performed on these sub-sequences, and their outputs are combined to form the N-point DFT.

Decimation-in-time is illustrated by the following equations (Oppenheim and Schafer, 1975). First, x(n), the input sequence in equation (1), is divided into even and odd sub-sequences:

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

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

for $k = 0$ to $N-1$

By the substitutions

$W_N^{2nk} = (e^{-j2\pi/N})^{2nk} = (e^{-j2\pi/(N/2)})^{nk} = W_{N/2}^{nk}$
$x_1(n) = x(2n)$
$x_2(n) = x(2n+1)$

this equation becomes

$$(3) \qquad X(k) = \sum_{n=0}^{N/2-1} x_1(n)\, W_{N/2}^{nk} + W_N^{k} \sum_{n=0}^{N/2-1} x_2(n)\, W_{N/2}^{nk}$$

$$= Y(k) + W_N^{k} Z(k)$$

for $k = 0$ to $N-1$

Equation (3) is the sum of two N/2-point DFTs (Y(k) and Z(k)) performed on the sub-sequences of even and odd samples, respectively, of the input sequence, x(n). Multiples of W_N (called "twiddle factors") appear as coefficients in the FFT calculation. In equation (3), Z(k) is multiplied by the twiddle factor W_N^{k}.

Because $W_N^{k+N/2} = (e^{-j2\pi/N})^{k} \times (e^{-j2\pi/N})^{N/2} = -W_N^{k}$, equation (3) can also be expressed as two equations:

$$(4) \qquad X(k) = Y(k) + W_N^{k} Z(k)$$

$$X(k+N/2) = Y(k) - W_N^{k} Z(k)$$

for $k = 0$ to $N/2-1$

Together these equations form an N-point FFT. Figure 6.1 illustrates this first decimation of the DFT.

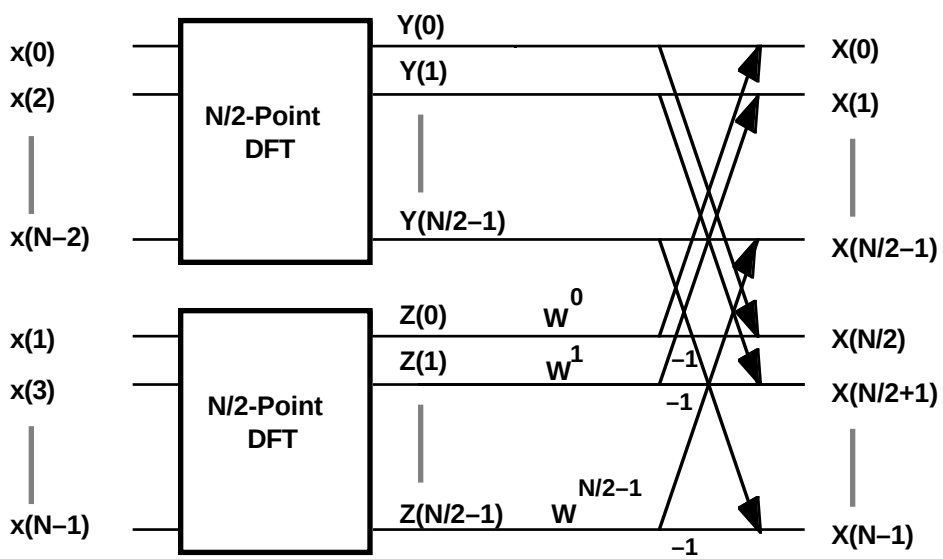

Figure 6.1 First Decimation of DIT FFT

The two N/2-point DFTs (Y(k) and Z(k)) can be divided to form four N/4-point DFTs, yielding equation pairs (5) and (6).

(5) $Y(k) = U(k) + W_N^{2k} V(k)$

$Y(k+N/4) = U(k) - W_N^{2k} V(k)$

for k = 0 to N/4–1

(6) $Z(k) = R(k) + W_N^{2k} S(k)$

$Z(k+N/4) = R(k) - W_N^{2k} S(k)$

for k = 0 to N/4–1

U(k) and V(k) are N/4-point DFTs whose input sequences are created by dividing $x_1(n)$ into even and odd sub-sequences. Similarly, R(k) and S(k) are N/4-point DFTs performed on the even and odd sub-sequences of $x_2(n)$. Each of these four equations can be divided to form two more. The final decimation occurs when each pair of equations together computes a

two-point DFT (one point per equation). The pair of equations that make up the two-point DFT is called a radix-2 "butterfly." The butterfly is the core calculation of the FFT. The entire FFT is performed by combining butterflies in patterns determined by the FFT algorithm.

A complete eight-point DIT FFT is illustrated graphically in Figure 6.2. Each pair of arrows represents a butterfly. Notice that the entire FFT computation is made up of butterflies organized in different patterns, called groups and stages. The first stage consists of four groups of one butterfly each. The second stage has two groups of two butterflies, and the last has one group of four butterflies. Every stage contains N/2 (four) butterflies. Each butterfly has two input points, called the dual node and the primary node. The spacing between the nodes in the sequence is called the dual-node spacing. Associated with each butterfly is a twiddle factor whose exponent depends on the group and stage of the butterfly.

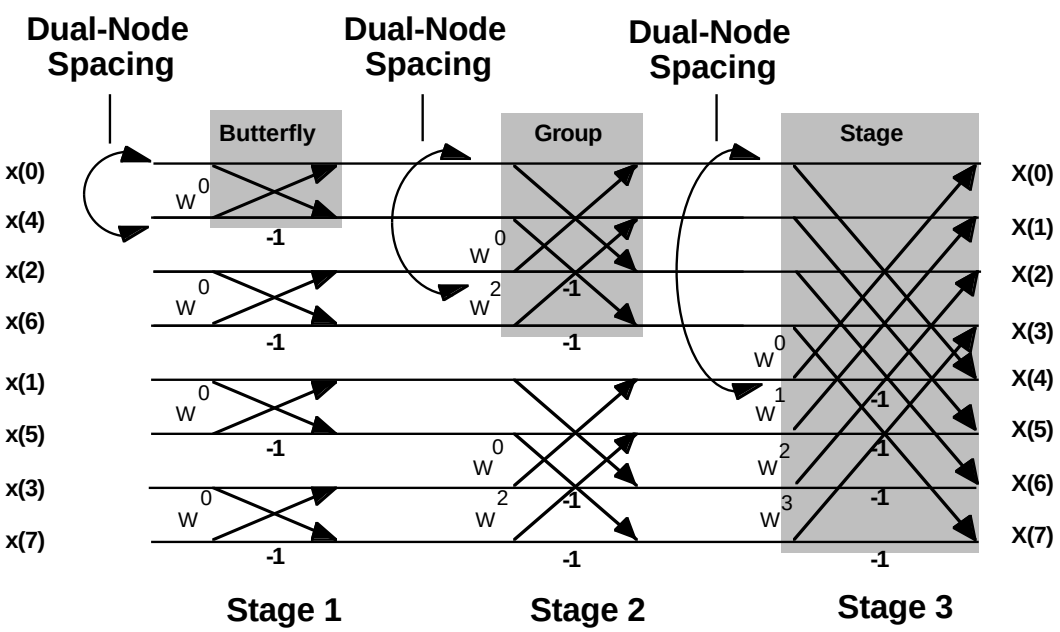

Figure 6.2 Eight-Point DIT FFT

Notice that whereas the output sequence is sequentially ordered, the input sequence is not. This is an effect of repeatedly dividing the input sequence into sub-sequences of even and odd samples. It is possible to perform an FFT using input and output sequences in other orders, but these approaches generally complicate addressing in the FFT program

and can require a different butterfly. In this section, we have opted to scramble the input sequence of the DIT FFT because this approach uses twiddle factors in sequential order, produces the output sequence in sequential order, and requires a relatively simple butterfly. The scrambling of the inputs is achieved by a process called bit reversal, which is described later in this chapter.

The characteristics of an N-point DIT FFT with bit-reversed inputs are summarized below.

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

A generalized butterfly flow graph is shown in Figure 6.3. 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 6.3 as C + j(–S). C represents cosine and –S represents –sine.

The dual node (x_1+jy_1) is multiplied by the twiddle factor C+j(–S). The result of this multiplication is added to the primary node (x_0+jy_0) to produce $x_0´+jy_0´$ and subtracted from the primary node to produce $x_1´+jy_1´$. Equations (7) through (10) calculate the real and imaginary parts of the butterfly outputs.

(7) $$x_0´ = x_0 + [(C)x_1 - (-S)y_1]$$

(8) $$y_0´ = y_0 + [(C)y_1 + (-S)x_1]$$

(9) $x_1' = x_0 - [(C)x_1 - (-S)y_1]$

(10) $y_1' = y_0 - [(C)y_1 + (-S)x_1]$

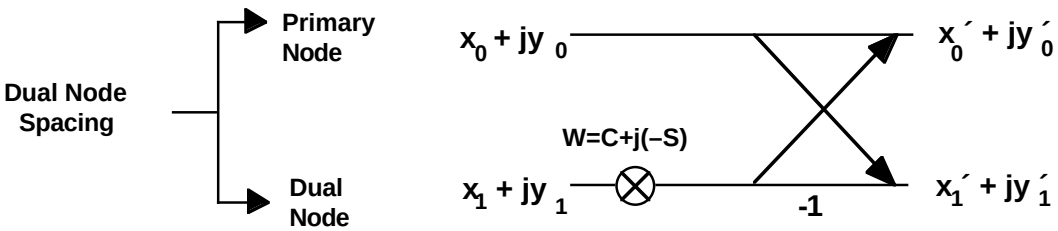

Figure 6.3 Radix-2 DIT FFT Butterfly

The butterfly produces two complex outputs that become butterfly inputs in the next stage of the FFT. Because each stage has the same number of butterflies (N/2), the number of butterfly inputs and outputs remains the same from one stage to the next. An "in-place" implementation writes each butterfly output over the corresponding butterfly input (x_0' overwrites x_0, etc.) for each butterfly in a stage. In an in-place implementation, the FFT results end up in the same memory range as the original inputs.

6.2.2 Radix-2 Decimation-In-Time FFT Program

The flow chart for the DIT FFT program is shown in Figure 6.4. The FFT program is divided into three subroutines. The first subroutine scrambles the input data. The next subroutine computes the FFT, and the third scales the output data.

Four modules are created. The main module declares and initializes data buffers and calls subroutines. The other three modules contain the FFT, bit reversal, and block floating-point scaling subroutines. The main module and FFT module are described in this section. The bit reversal and block floating-point scaling modules are described in later sections.

6.2.2.1 Main Module

The *dit_fft_main* module is shown in Listing 6.1. *N* is the number of points in the FFT (in this example, N=1024) and *N_div_2* is used for specifying the lengths of buffers. To change the number of points in the FFT, you

Figure 6.4 Radix-2 DIT FFT Flow Chart

change the value of these constants and the twiddle factors. The data buffers *twid_real* and *twid_imag* in program memory hold the twiddle factor cosine and sine values. The *inplacereal, inplaceimag, inputreal* and *inputimag* buffers in data memory store real and imaginary data values. Sequentially ordered input data is stored in *inputreal* and *inputimag*. This data is scrambled and written to *inplacereal* and *inplaceimag*, which are the data buffers used by the in-place FFT. A four-location buffer called *padding* is placed at the end of *inplaceimag* to allow data accesses to exceed the buffer length. If no *padding* was placed after *inplaceimag* and the program attempted to read undefined memory locations, the ADSP-2100 Simulator would signal an error. This buffer assists in debugging but is not necessary in a real system. Variables (one-location buffers) *groups, bflys_per_group, blk_exponent* and *node_space* are declared last.

The real part (cosine values) of the twiddle factors in the *twid_real.dat* file are placed in the buffer *twid_real.* Likewise, *twid_imag.dat* is placed in *twid_imag.* The variable *groups* is initialized to *N_div_2*, and *bflys_per_group* and *node_space* are initialized to one because there are N/2 groups of one butterfly in the first stage of the FFT. The *blk_exponent* is initialized to zero. This exponent value is updated when the output data is scaled.

Two subroutines are called. The first subroutine places the input sequence in bit-reversed order. The second performs the FFT and calls the block floating-point scaling routine.

6.2.2.2 DIT FFT Module

The FFT routine uses three nested loops. The inner loop computes butterflies, the middle loop controls the grouping of these butterflies, and the outer loop controls the FFT stage characteristics. These loops are described separately in the following sections. The complete routine is presented at the end.

Butterfly Loop

The butterfly calculation involves a complex multiplication, a complex addition, and a complex subtraction. These operations can potentially cause the butterfly data to grow by two bits from input to output. For example, if x_0 is H#07FF (five sign bits), x_0' could be H#100F (three sign bits). Because of this bit growth, precautions must be taken to ensure that 16-bit data never overflows.

```
.MODULE/ABS=4           dit_fft_main;
.CONST                  N=1024, N_div_2=512;     {Const. for 1024 points}
.VAR/PM/RAM/CIRC        twid_real [N_div_2];
.VAR/PM/RAM/CIRC        twid_imag [N_div_2];
.VAR/DM/RAM/ABS=0       inplacereal [N], inplaceimag [N];
.VAR/DM/RAM/ABS=H#1000  inputreal [N], inputimag [N], padding [4];
.VAR/DM/RAM             groups, bflys_per_group,
                        node_space, blk_exponent;

.INIT       twid_real: <twid_real.dat>;
.INIT       twid_imag: <twid_imag.dat>;
.INIT       inputreal: <inputreal.dat>;
.INIT       inputimag: <inputimag.dat>;
.INIT       inplaceimag: <inputimag.dat>;
.INIT       groups: N_div_2;
.INIT       bflys_per_group: 1;
.INIT       node_space: 1;
.INIT       blk_exponent: 0;
.INIT       padding: 0,0,0,0;                    {Zeros after inplaceimag}

.GLOBAL     inplacereal, inplaceimag;
.GLOBAL     inputreal, inputimag;
.GLOBAL     twid_real, twid_imag;
.GLOBAL     groups, bflys_per_group, node_space, blk_exponent;

.EXTERNAL   scramble, fft_strt;

            CALL scramble;                       {Subroutine calls}
            CALL fft_strt;
            TRAP;                                {Halt program}
.ENDMOD;
```

Listing 6.1 Main Module, Radix-2 DIT FFT

6 One-Dimensional FFTs

An example of bit growth and overflow is shown below.

Bit Growth:
Input to butterfly H#0F00 = 0000 1111 0000 0000
Output from butterfly H#1E00 = 0001 1110 0000 0000

Overflow:
Input to butterfly H#7000 = 0111 0000 0000 0000
Output from butterfly H#E000 = 1110 0000 0000 0000

In overflow, the positive number H#7000 is multiplied by a positive number, resulting in H#E000, which is too large to represent as a positive, signed 16-bit number. H#E000 is erroneously interpreted as a negative number.

To avoid errors caused by overflow, one of three methods of compensating for bit growth can be applied:

- Input data scaling
- Unconditional block floating-point scaling (output data)
- Conditional block floating-point scaling (output data)

Three different code segments for the butterfly calculation are presented in this section; each uses a different method of compensating for bit growth.

One way to ensure that overflow never occurs is to include enough extra sign bits, called guard bits, in the FFT input data to ensure that bit growth never results in overflow (Rabiner and Gold, 1975). Data can grow by a maximum factor of 2.4 from butterfly input to output (two bits of growth). However, a data value cannot grow by this maximum amount in two consecutive stages. The number of guard bits necessary to compensate for the maximum possible bit growth in an N-point FFT is $\log_2 N+1$. For example, each of the input samples of a 32-point FFT (requiring five stages) must contain six guard bits, so ten bits are available for data (one sign bit, nine magnitude bits). This method requires no data shifting and is therefore the fastest of the three methods discussed in this section. However, for large FFTs the resolution of the input data is greatly limited. For small, low-precision FFTs, this is the fastest and most efficient method.

The code segment for a butterfly with no shifting is shown in Listing 6.2. This section of code computes one butterfly equation while setting up values for the next butterfly. The butterfly outputs (x_0', y_0' x_1' and y_1') are written over the inputs to the butterfly (x_0, y_0, x_1 and y_1) in the boldface

instructions. The input and output parameters of the butterfly loop are shown below.

Initial Conditions

MX0 = x_1
MY0 = C
MY1 = (–S)
I0 --> x_0
I1 --> x_1
I2 --> y_0
I3 --> y_1
I4 --> next C
I5 --> next (–S)
I6 --> y_1
CNTR = butterfly count
M0 = 0
M1 = 1
M4 = twiddle factor modify value
M5 = 1

Final Conditions

MX0 = next x_1
MY0 = next C
MY1 = next (–S)
I0 --> next x_0
I1 --> next x_1
I2 --> next y_0
I3 --> next y_1
I4 --> C after next C
I5 --> (–S) after next (–S)
I6 --> next y_1
CNTR = butterfly count –1

```
MR=MX0*MY0(SS),MX1=DM(I6,M5);     {MR=x1(C),MX1=y1}
MR=MR-MX1*MY1(RND),AY0=DM(I0,M0); {MR=x1(C)-y1(-S),AY0=x0}
AR=MR1+AY0,AY1=DM(I2,M0);         {AR=x0+[x1(C)-y1(-S)],AY1=y0}
AR=AY0-MR1,DM(I0,M1)=AR;          {AR=x0-[x1(C)-y1(-S)],x0´=x0+[x1(C)-y1(-S)]}
MR=MX0*MY1(SS),DM(I1,M1)=AR;      {MR=x1(-S),x1´=x0-[x1(C)-y1(-S)]}
MR=MR+MX1*MY0(RND),MX0=DM(I1,M0),MY1=PM(I5,M4);
                                  {MR=x1(-S)+y1(C),MX0=next x1,MY1=next (-S)}
AR=AY1-MR1,MY0=PM(I4,M4);         {AR=y0-[x1(-S)+y1(C)],MY0=next C}
AR=MR1+AY1,DM(I3,M1)=AR;          {AR=y0+[x1(-S)+y1(C)],y1´=y0-[x1(-S)+y1(C)]}
DM(I2,M1)=AR;                     {y0´=y0+[x1(-S)+y1(C)]}
```

Listing 6.2 DIT FFT Butterfly, Input Data Scaled

Another way to compensate for bit growth is to scale the outputs down by a factor of two unconditionally after each stage. This approach is called unconditional block floating-point scaling. Initially, two guard bits are included in the input data to accommodate the maximum bit growth in the first stage. In each butterfly of a stage calculation, the data can grow into the guard bits. To prevent overflow in the next stage, the guard bits are replaced before the next stage is executed by shifting the entire block of data one bit to the right and updating the block exponent. This shift is necessary after every stage except the last, because no overflow can occur after the last stage.

6 One-Dimensional FFTs

The input data to an unconditional block floating-point FFT can have at most 14 bits (one sign bit and 13 magnitude bits). In the FFT calculation, the data loses a total of ($\log_2$N)–1 bits because of shifting. Unconditional block floating-point scaling results in the same number of bits lost as in input data scaling. However, it produces more precise results because the FFT starts with more precise input data. The tradeoff is a slower FFT calculation because of the extra cycles needed to shift the output of each stage.

The code for the unconditional block floating-point butterfly is shown in Listing 6.3. Instructions that write butterfly results to memory are boldface. After the last stage of the FFT, no compensation for bit growth is needed, so a butterfly with no shifting can be used in the last stage.

Initial Conditions

SR0 = last y_0´
MX0 = x_1
MX1 = y_1
MY0 = C
MY1 = (–S)
I0 --> x_0
I1 --> x_1
I2 --> last y_0´
I3 --> y_1
I4 --> next C
I5 --> next (–S)
I6 --> next y_1
CNTR = butterfly count
M0 = 0
M1 = 1
M4 = twiddle factor modify value
M5 = 1
SE = –1

Final Conditions

SR0 = y_0´
MX0 = next x_1
MX1 = next y_1
MY0 = next C
MY1 = next (–S)
I0 --> next x_0
I1 --> next x_1
I2 --> y_0´
I3 --> next y_1
I4 --> C after next C
I5 --> (–S) after next (–S)
I6 --> y_1 after next y_1
CNTR = butterfly count –1

```
MR=MX0*MY0(SS),DM(I2,M1)=SR0;     {MR=x1(C),last y0=last y0´}
MR=MR-MX1*MY1(RND),AY0=DM(I0,M0); {MR=x1(C)-y1(-S),AY0=x0}
AR=MR1+AY0,AY1=DM(I2,M0);         {AR=x0+[x1(C)-y1(-S)],AY1=y0}
SR=ASHIFT AR(LO);                 {Shift result right 1 bit}
DM(I0,M1)=SR0,AR=AY0-MR1;         {x0´=x0-[x1(C)-y1(-S)],AR=x0-[x1(C)-y1(-S)]}
SR=ASHIFT AR(LO);                 {Shift result right 1 bit}
DM(I1,M1)=SR0,MR=MX0*MY1(SS);     {x1´=x0-[x1(C)-y1(-S)],MR=x1(-S)]}
MR=MR+MX1*MY0(RND),MX0=DM(I1,M0),MY1=PM(I5,M4);
                                  {MR=x1(-S)-y1(C),MX0=next x1,MY1=next(-S)}
AR=AY1-MR1,MY0=PM(I4,M4);         {AR=y0-[x1(-S)-y1(C)],MY0=next C}
SR=ASHIFT AR(LO),MX1=DM(I6,M5);   {Shift result right 1 bit,MX1=next y1}
DM(I3,M1)=SR0,AR=MR1+AY1;         {y1´=y0-[x1(-S)-y1(C),AR=y0+[x1(-S)-y1(C)]}
SR=ASHIFT AR(LO);                 {Shift result right 1 bit}
```

Listing 6.3 DIT FFT Butterfly, Unconditional Block Floating-Point Scaling

In conditional block floating-point scaling, data is shifted *only* if bit growth occurs. If one or more outputs grows, the entire block of data is shifted to the right and the block exponent is updated. For example, if the original block exponent is 0 and data is shifted three positions, the resulting block exponent is +3.

The code segment for the conditional block floating-point butterfly is shown in Listing 6.4. As in the other types of butterflies, one butterfly equation is calculated and its outputs (x_0´, y_0´, x_1´ and y_1´) are written over its inputs (x_0, y_0, x_1 and y_1) in the boldface instructions.

The conditional block floating-point butterfly checks each butterfly output for growth with the EXPADJ instruction. This instruction does no shifting; instead, it monitors the output data and updates the SB register if bit growth is detected. (See the *ADSP-2100 User's Manual* for a complete description of this instruction.) If shifting is necessary it is performed after the entire stage is complete (in the block floating-point scaling routine). The butterfly code computes one butterfly equation while setting up values for the next butterfly. The input and output parameters of the butterfly loop are as follows:

Initial Conditions	*Final Conditions*
MX0 = x_1	MX0 = next x_1
MX1 = y_1	MX1 = next y_1
MY0 = C	MY0 = next C
MY1 = (–S)	MY1 = next (–S)
I0 --> x_0	I0 --> next x_0
I1 -> x_1	I1 --> next x_1
I2 --> y_0	I2 --> next y_0
I3 --> y_1	I3 --> next y_1
I4 --> next C	I4 --> C after next C
I5 --> next (–S)	I5 --> (–S) after next (–S)
CNTR = butterfly count	CNTR = butterfly count –1
M1 = 1	
M4 = twiddle factor modify value	
M0 = 0	
SB = monitored block exponent for this stage	

```
MR=MX0*MY1(SS),AX0=DM(I0,M0);     {MR=x1(-S),AX0=x0}
MR=MR+MX1*MY0(RND),AX1=DM(I2,M0); {MR=[y1(C)+x1(-S)];AX1=y0}
AY1=MR1,MR=MX0*MY0(SS);           {AY1=[y1(C)+x1(-S)];MR=x1(C)}
MR=MR-MX1*MY1(RND);               {MR=[x1(C)-y1(-S)]}
AY0=MR1,AR=AX1-AY1;               {AY0=[x1(C)-y1(-S)],AR=y0-[y1(C)+x1(-S)]}
SB=EXPADJ AR,DM(I3,M1)=AR;        {check for bit growth,y1´=y0-[y1(C)+x1(-S)]}
AR=AX0-AY0,MX1=DM(I3,M0),MY1=PM(I5,M4);
                                  {AR=x0-[x1(C)-y1(-S)],MX1=next y1,MY1=next S}
SB=EXPADJ AR,DM(I1,M1)=AR;        {check for bit growth,x1´=x0-[x1(C)-y1(-S)]}
AR=AX0+AY0,MX0=DM(I1,M0),MY0=PM(I4,M4);
                                  {AR=x0+[x1(C)-y1(-S)],MX0=next x1,MY0=next C}
SB=EXPADJ AR,DM(I0,M1)=AR;        {check for bit growth,x0´=x0+[x1(C)-y1(-S)]}
AR=AX1+AY1;                       {AR=y0+[y1(C)+x1(-S)]}
SB=EXPADJ AR,DM(I2,M1)=AR;        {check for bit growth,y0´=y0+[y1(C)+x1(-S)]}
```

Listing 6.4 DIT FFT Butterfly, Conditional Block Floating-Point Scaling

Group Loop

The group loop controls the grouping of butterflies. It sets pointers to the input data and twiddle factors of the first butterfly in the group, initializes the butterfly counter and sets up the butterfly loop for each group.

The code segment for the group loop is shown in Listing 6.5. This code is designed for the conditional block floating-point butterfly and thus requires slight modification for use with the other types (input scaling, unconditional block floating-point) of butterflies. The first butterfly of every group in the first stage of the DIT FFT has a twiddle factor of W^0. Thus, I4 and I5 are initialized to point to the cosine and sine values of W^0 before the butterfly loop is entered. In the group loop, the butterfly counter is initialized and initial butterfly data is fetched. The butterfly loop is executed *bflys_per_group* times to compute all butterflies in the group. After the butterfly loop is complete, pointers I0, I1, I2 and I3 are updated with the MODIFY instruction to point to x_0, x_1, y_0 and y_1 of the first butterfly in the next group. The group loop is executed *groups* times.

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 --> y_0 of first butterfly in group	I2 --> y_0 of first butterfly in next group
I3 --> y_1 of first butterfly in group	I3 --> y_1 of first butterfly in next group
CNTR = group count	CNTR = group count –1
M2 = node_space	

```
            I4=^twid_real;
            I5=^twid_imag;                {Initialize twiddle factor pointers}
            CNTR=DM(bflys_per_group);     {Initialize butterfly counter}
            MY0=PM(I4,M4),MX0=DM(I1,M0);  {MY0=C,MX0=x1}
            MY1=PM(I5,M4),MX1=DM(I3,M0);  {MY1=(-S),MX1=y1}
            DO bfly_loop UNTIL CE;

bfly_loop:     {Calculate All Butterflies in Group}

            MODIFY(I0,M2);                {I0 -->first x0 in next group}
            MODIFY(I1,M2);                {I1 -->first x1 in next group}
            MODIFY(I2,M2);                {I2 -->first y0 in next group}
group_loop: MODIFY(I3,M2);                {I3 -->first y1 in next group}
```

Listing 6.5 Radix-2 DIT FFT Group Loop

Stage Loop

The stage loop controls the grouping characteristics of the FFT. These include the number of groups in a stage, the number of butterflies in each group, and the node spacing. The stage loop also calls a subroutine which performs conditional block floating-point scaling on the outputs of a stage calculation. Note that if unconditional block floating-point scaling or input data scaling were used, this call would be omitted.

The stage loop code for a conditional block floating-point FFT is shown in Listing 6.6. The stage loop sets up the group loop by initializing I0, I1, I2 and I3 to point to x_0, x_1, y_0 and y_1, respectively, for the first butterfly in the first group. It also initializes the group loop counter and node space modifier so that pointers can be updated for new groups. The value of the twiddle factor exponent is increased by *groups* for each butterfly. M4, initialized to *groups*, is the modifier for the twiddle factor pointers.

The group loop calculates all groups in the stage. After the group loop is complete, a block floating-point subroutine is called to check the stage outputs for bit growth and scale the data if necessary. The stage characteristics are then updated for the next stage; *bflys_per_group* and *node_space* are doubled and *groups* is divided by two.

The input and output parameters for the stage loop are as follows. Note that all the parameters except the stage count are passed in memory.

Initial Conditions	*Final Conditions*
groups=# groups current stage	*groups*=# groups 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
CNTR=stage count	CNTR=stage count –1
inplacereal=real stage input data	*inplacereal*=real stage output data
inplaceimag=imag. stage input data	*inplaceimag*=imag. stage output data

```
            I0=^inplacereal;          {I0 -->first x0 in first group of stage}
            I2=^inplaceimag;          {I2 -->first y0 in first group of stage}
            SB=-2                     {SB = -(number of guard bits)}
            SI=DM(groups);            {SI = groups}
            CNTR=SI;                  {Initialize group counter}
            M4=SI;                    {Initialize twiddle factor modifier}
            M2=DM(node_space);        {Initialize node spacing modifier}
            I1=I0;
            MODIFY(I1,M2);            {I1 -->first x1 of first group in stage}
            I3=I2;
            MODIFY(I3,M2);            {I3 -->first y1 of first group in stage}
            DO group_loop UNTIL CE;
group_loop:    {Compute All Groups in Stage}

            CALL bfp_adj;             {Adjust stage output for bit growth}
            SI=DM(bflys_per_group);
            SR=ASHIFT SI BY 1(LO);
            DM(node_space)=SR0;       {node_space=node_space × 2}
            DM(bflys_per_group)=SR0; {bflys_per_group=bflys_per_group × 2}
            SI=DM(groups);
            SR=ASHIFT SI BY -1(LO);
            DM(groups)=SR0;           (groups = groups ÷ 2}
```

Listing 6.6 Radix-2 DIT FFT Stage Loop

DIT FFT Subroutine

The complete conditional block floating-point radix-2 DIT FFT routine is shown in Listing 6.7. The constants N and log_2N are the number of points and the number of stages in the FFT, respectively. To change the number of points in the FFT, you modify these constants. Notice that the length and modify registers (that retain the same values throughout the FFT calculation) and the stage counter are initialized before the stage loop is executed. Instructions that write butterfly results to memory are boldface.

6 One-Dimensional FFTs

```
.MODULE     fft;

{           Performs Radix-2 DIT FFT

            Calling Parameters
               inplacereal = Real input data in scrambled order
               inplaceimag = All zeroes (real input assumed)
               twid_real = Twiddle factor cosine values
               twid_imag = Twiddle factor sine values
               groups = N/2
               bflys_per_group = 1
               node_space = 1

            Return Values
               inplacereal = Real FFT results in sequential order
               inplaceimag = Imaginary FFT results in sequential order

            Altered Registers
               I0,I1,I2,I3,I4,I5,L0,L1,L2,L3,L4,L5
               M0,M1,M2,M3,M4,M5
               AX0,AX1,AY0,AY1,AR,AF
               MX0,MX1,MY0,MY1,MR,SB,SE,SR,SI

            Altered Memory
               inplacereal, inplaceimag, groups, node_space,
               bflys_per_group, blk_exponent
}
.CONST      log2N=10, N=1024;                  {Set constants for N-point FFT}
.EXTERNAL   twid_real, twid_imag;
.EXTERNAL   inplacereal, inplaceimag;
.EXTERNAL   groups, bflys_per_group, node_space;
.EXTERNAL   bfp_adj;
.ENTRY      fft_strt;

fft_strt:   CNTR=log2N;                        {Initialize stage counter}
            M0=0;
            M1=1;
            L1=0;
            L2=0;
            L3=0;
            L4=%twid_real;
            L5=%twid_imag;
            DO stage_loop UNTIL CE;            {Compute all stages in FFT}
               I0=^inplacereal;                {I0 -->x0 in 1st grp of stage}
               I2=^inplaceimag;                {I2 -->y0 in 1st grp of stage}
               SB=-2                           {SB to detect data > 14 bits}
               SI=DM(groups);
               CNTR=SI;                        {CNTR = group counter}
               M4=SI;                          {M4=twiddle factor modifier}
               M2=DM(node_space);              {M2=node space modifier}
               I1=I0;
```

```
                MODIFY(I1,M2);                  {I1 -->x1 of 1st grp in stage}
                I3=I2;
                MODIFY(I3,M2);                  {I3 -->y1 of 1st grp in stage}
                DO group_loop UNTIL CE;
                   I4=^twid_real;               {I4 --> C of W⁰}
                   I5=^twid_imag;               {I5 --> (-S) of W⁰}
                   CNTR=DM(bflys_per_group);    {CNTR = butterfly counter}
                   MY0=PM(I4,M4),MX0=DM(I1,M0);     {MY0=C,MX0=x1 }
                   MY1=PM(I5,M4),MX1=DM(I3,M0);     {MY1=-S,MX1=y1}
                   DO bfly_loop UNTIL CE;
                      MR=MX0*MY1(SS),AX0=DM(I0,M0);{MR=x1(-S),AX0=x0}
                      MR=MR+MX1*MY0(RND),AX1=DM(I2,M0);
                                                {MR=(y1(C)+x1(-S)),AX1=y0}
                      AY1=MR1,MR=MX0*MY0(SS);   {AY1=y1(C)+x1(-S),MR=x1(C)}
                      MR=MR-MX1*MY1(RND);       {MR=x1(C)-y1(-S)}
                      AY0=MR1,AR=AX1-AY1;       {AY0=x1(C)-y1(-S),}
                                                {AR=y0-[y1(C)+x1(-S)]}
                      SB=EXPADJ AR,DM(I3,M1)=AR;    {Check for bit growth,}
                                                {y1´=y0-[y1(C)+x1(-S)]}
                      AR=AX0-AY0,MX1=DM(I3,M0),MY1=PM(I5,M4);
                                                {AR=x0-[x1(C)-y1(-S)],}
                                                {MX1=next y1,MY1=next (-S)}
                      SB=EXPADJ AR,DM(I1,M1)=AR;    {Check for bit growth,}
                                                {x1´=x0-[x1(C)-y1(-S)]}
                      AR=AX0+AY0,MX0=DM(I1,M0),MY0=PM(I4,M4);
                                                {AR=x0+[x1(C)-y1(-S)],}
                                                {MX0=next x1,MY0=next C}
                      SB=EXPADJ AR,DM(I0,M1)=AR;    {Check for bit growth,}
                                                {x0´=x0+[x1(C)-y1(-S)]}
                      AR=AX1+AY1;               {AR=y0+[y1(C)+x1(-S)]}
bfly_loop:            SB=EXPADJ AR,DM(I2,M1)=AR;    {Check for bit growth,}
                                                {y0´=y0+[y1(C)+x1(-S)]}
                   MODIFY(I0,M2);               {I0 -->1st x0 in next group}
                   MODIFY(I1,M2);               {I1 -->1st x1 in next group}
                   MODIFY(I2,M2);               {I2 -->1st y0 in next group}
group_loop:        MODIFY(I3,M2);               {I3 -->1st y1 in next group}
                CALL bfp_adj;                   {Compensate for bit growth}
                SI=DM(bflys_per_group);
                SR=ASHIFT SI BY 1(LO);
                DM(node_space)=SR0;             {node_space=node_space × 2}
                DM(bflys_per_group)=SR0;        {bflys_per_group= }
                                                    {bflys_per_group × 2}
                SI=DM(groups);
                SR=ASHIFT SI BY -1(LO);
stage_loop:     DM(groups)=SR0;                 {groups=groups ÷ 2}
             RTS;
.ENDMOD;
```

Listing 6.7 Radix-2 DIT FFT Routine, Conditional Block Floating-Point