Two-Dimensional FFTs ■ 7

7.1 TWO-DIMENSIONAL FFTS

The two-dimensional discrete Fourier transform (2D DFT) is the discrete-time equivalent of the two-dimensional continuous-time Fourier transform. Operating on $x(n_1,n_2)$, a sampled version of a continuous-time two-dimensional signal, the 2D DFT is represented by the following equation:

$$X(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2)\, W_N^{n_1 k_1}\, W_N^{n_2 k_2}$$

where $W_N = e^{-j2\pi/N}$ and the signal is defined for the points $0 \le n_1, n_2 < N_1, N_2$.

Direct calculation of the 2D DFT is simple, but requires a very large number of complex multiplications. Assuming all of the exponential terms are precalculated and stored in a table, the total number of complex multiplications needed to evaluate the 2D DFT is $N_1^2N_2^2$. The number of complex additions required is also $N_1^2N_2^2$. Direct evaluation of the 2D DFT for a square image, 128 pixels by 128 pixels, requires over 268 million complex multiplications.

Two techniques can be employed to reduce the operation count of the two-dimensional transform. First, the row-column decomposition method (Dudgeon and Mersereau, 1984) partitions the two-dimensional DFT into many one-dimensional DFTs. Row-column decomposition reduces the number of complex multiplications from $N_1^2N_2^2$ (direct evaluation) to $N_1N_2(N_1+N_2)$ (row-column decomposition with direct DFT evaluation). For the 128-by-128-pixel example, the number of complex multiplications is reduced from 268 million to less than 4.2 million, a factor of 63.

The second technique to reduce the operation count of the two-dimensional transform is the fast Fourier transform (FFT). The FFT is a shortcut evaluation of the DFT. The FFT is used to evaluate the one-dimensional DFTs produced by the row-column decomposition.

7 Two-Dimensional FFTs

7.1.1 Row-Column Decomposition

Row-column decomposition is straightforward. The 2D DFT double summation

$$X(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2}$$

can be rewritten as

$$X(k_1, k_2) = \sum_{n_1=0}^{N_1-1} \left[\sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_2}^{n_2 k_2} \right] W_{N_1}^{n_1 k_1}$$

in which all occurrences of n_2 are grouped within the square brackets. The result is that the quantity within the brackets is a one-dimensional DFT. The equation can then be separated into the following two equations:

$$A(n_1, k_2) = \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_2}^{n_2 k_2}$$

$$X(k_1, k_2) = \sum_{n_1=0}^{N_1-1} A(n_1, k_2) W_{N_1}^{n_1 k_1}$$

In the equations above, the function $A(n_1, k_2)$ has rows (n_1) and columns (k_2). The columns are one-dimensional DFTs of the corresponding columns of the original signal, $x(n_1, n_2)$. The second equation evaluates row DFTs. The DFT of each row is evaluated for the intermediate function $A(n_1, k_2)$. Therefore, in order to evaluate the DFT of a two-dimensional signal, $x(n_1, n_2)$, you first evaluate the DFT for each row. The results of these 1D DFTs are stored in an intermediate array, $A(n_1, k_2)$. Then 1D DFTs are evaluated for each column of the intermediate array. The result is the 2D DFT of the original signal. For an image of 128 by 128 pixels, 256 1D DFTs need to be evaluated.

When the FFT algorithm is employed to evaluate the 1D DFTs from the row-column decomposition, there are further significant computational savings. The number of complex multiplications is reduced from

$N_1N_2(N_1+N_2)$ for direct evaluation of each DFT to $N_1N_2(\log_2(N_1N_2/2))$ for FFT evaluation using the row-column decomposition method. For the 128-by-128 pixel example mentioned above, the number of complex multiplications necessary is less than 115 thousand. Therefore, using row-column decomposition and FFTs, there is a reduction by a factor of 2300 in the number of complex multiplications for a 128-by-128-point image.

The one-dimensional radix-2, decimation-in-frequency FFT algorithm from Chapter 6 is adapted for two dimensions in this chapter. Row-column decomposition facilitates the transition to two dimensions. To evaluate the FFT of a N-by-N-point image, you must simply evaluate 2N one-dimensional FFTs.

7.1.2 Radix-2 FFT

The FFT algorithm divides an input sequence into smaller sequences, evaluates the DFTs of these smaller sequences, and combines the outputs of the small DFTs to give the DFT of the original input sequence. The radix-2 FFT divides an N-point input sequence in half, into two N/2-point sequences. This division provides computational savings. Further computational savings are realized by dividing each N/2-point sequence into two N/4-point sequences. The strategy of a radix-2 FFT is to divide the input sequence successively by two until the resulting sequences contain only two points. For example, an 8-point DFT is reduced to four 2-point DFTs. The number of multiplications needed to evaluate a DFT is proportional to the square of the number of input points. Dividing the number of input points in half and evaluating two DFTs reduces the multiplication count from N^2 to $2(N/2)^2$. Each of these N/2-point sequences are again divided in half, and the operation count is reduced in the same fashion. With each decimation, the multiplication count is reduced by a factor of two. This scheme dramatically reduces the number of multiplications necessary to evaluate the DFT of the original sequence.

There are two basic varieties of radix-2 FFTs: decimation-in-time (DIT) and decimation-in-frequency (DIF). The DIT FFT divides the input sequence into even samples and odd samples. Each half-sequence is, in turn, divided into even and odd samples. Division into even and odd samples is facilitated by bit-reversing the addresses of the input sequence. The radix-2 DIT FFT takes a bit-reversed input sequence and outputs samples in normal order.

The DIF radix-2 FFT also successively divides the input sequence in half; however, the two N/2-point sequences consist of the first N/2 samples and the last N/2 samples. One sequence contains the points 0 through

N/2–1, and the other sequence contains the points N/2 through N–1. The inputs to the DIF FFT are addressed in normal order, and the results are output in bit-reversed order, which is the converse of the operation of the DIT FFT.

The row-column decomposition described in this chapter evaluates the 2D DFT of an N-by-N-point signal using 2N one-dimensional FFTs. The DIF radix-2 FFT is used to evaluate the rows and columns of an N-by-N-point input signal.

7.1.3 ADSP-2100 Implementation

This implementation of the 2D FFT uses nine software routines. The main routine declares and initializes data buffers and calls subroutines that perform the 2D FFT. Two subroutines perform initialization tasks. Row and column 1D FFTs are performed with block floating-point DIF FFTs. There are four subroutines that adjust the FFT output for bit growth. Two of these routines operate on row FFT output and two operate on column FFT output. There are also two bit reversal subroutines, one each for row and column FFTs.

Input data for this 2D FFT must be in 16 bit twos-complement fractional format. This input format is often called 1.15. Two guard bits is also necessary to prevent overflow in the first stage of the FFT. The implementation presented in this chapter is a block floating-point implementation of the 2D FFT. This implementation is computationally less intensive than full floating-point and provides more dynamic range than a fixed-point implementation. The ADSP-2100 family of DSP processors posess the necessary hardware to efficiently perform these block floating-point operations. There are four block floating-point adjust routines within the 2D FFT program. Two of these operate on row FFT data and two operate on column FFT data.

7.1.3.1 Main Module

Declaring variables, initializing data buffers and program variables, and calling subroutines are the major functions of the main module. The two-dimensional FFT is performed in place (using the same data buffers for inputs and outputs). The data buffers *realdata* and *imaginarydata* contain the real and imaginary parts of the input signal at the start of the program. As the program executes, all partial results are written to these data buffers, and at program completion, the buffers *realdata* and *imaginarydata* contain the resulting frequency samples. The in-place buffers are shown in Figure 7.1.

0	1	2	3		63
64	65	66	67		127
128					191
4032					4095

Schematic Representation of realdata

4096	4097				4159
4160					
8128					8191

Schematic Representation of imaginarydata

Figure 7.1 In-Place Buffers

Twiddle factors, sine values and cosine values used for evaluation of complex exponentials, are stored in two tables, *twid_real* and *twid_imag*. These twiddle factors are multiplied with the FFT data values. To exploit the Harvard architecture of the ADSP-2100, the twiddle factor tables are placed in program memory. A dual data fetch occurs in a single processor cycle, retrieving a twiddle factor fron program memory and a data value from data memory. The indexing of the twiddle factors is maintained with the index registers I4 and I5.

7 Two-Dimensional FFTs

Because row-column decomposition divides the 2D FFT into many 1D FFTs, a mechanism is needed to keep track of the current FFT. This program uses five variables for this purpose: *offset*, *rrowbase*, *irowbase*, *rcolbase*, and *icolbase*. The *rowbase* variables keep track of the real and imaginary row FFT starting points, and the *colbase* variables keep track of the real and imaginary column FFT starting points. *Offset* is used to update these variables as each FFT is completed. Three other variables—*groups*, *bflys_per_group*, and *node_space*—are used in each FFT.

Listing 7.1 contains the source code for the main module. The main module calls eight subroutines. The first, *initialize*, performs the once-only initialization of data pointers, length registers and modify registers. The second, *fft_start*, performs DIF FFTs on the row data. For an N-by-N input signal, this routine is called N times. After all of the row FFTs have been performed, the routine *unscr_start* unscrambles the row output data by bit-reversing the addresses. Then the *row_final_adj* routine adjusts the outputs of the row FFTs for bit growth.

For the column FFTs, *col_init*, an initialization routine, is called. Because sequential points in the column FFTs do not reside in sequential addresses, the *col_init* routine is needed to initialize the parameters that define the FFT bounds. The column FFT subroutine, *col_fft_strt*, is called N times to transform each column of data in the buffers *realdata* and *tempdata*. Once the column FFTs are done, the subroutine *col_unscr_start* bit-reverses the column outputs' addressing. Finally, the subroutine *col_final_adj* adjusts the column output data for bit growth.

The *initialize* and *col_init* routines are part of the main modules, but are described later in Listings 7.5 and 7.6. The other subroutines are in other modules.

```
.MODULE      fft_2d;

{     Performs a two dimensional FFT using the row column
      decomposition method. One dimensional FFTs are performed
      first on each row. The pointers rrowbase and irowbase keep
      track of row bounds. One dimensional FFTs are then
      performed on each column. The pointers rcolbase and
      icolbase keep track of the column data. The row FFTs and
      column FFTs are decimation in frequency: input in normal
      order and output in bit-reversed order.
```

```
      Variables
           realdata          Array (N X N) of real input data
           imaginarydata     Array (N X N) of imaginary input
data
           twid_real         Real part of twiddle factors
           twid_imag         Imaginary part of twiddles
           groups            # of groups in current stage
           node_space        spacing between dual node points
           bflys_per_group   # of butterflys per group
           rrowbase          real data row pointer
           irowbase          imaginary data row pointer
           rcolbase          real data column pointer
           icolbase          imaginary data column pointer
           fft_start         entry point row FFTs
           col_fft_strt      entry point column FFTs
           shift_strt        entry point normalizer
}

.CONST                  N = 64, N_X_2 = 128;
.CONST                  N_div_2 = 32, log2N = 6;

.VAR/DM/RAM             realdata[4096];
.VAR/DM/RAM/ABS=h#1000  imaginarydata[4096];
.VAR/DM/RAM/ABS=h#2000  tempdata[4096];

.VAR/PM/RAM/CIRC  twid_real[N_div_2];
.VAR/PM/RAM/CIRC  twid_imag[N_div_2];
.VAR/DM           row_exponents[N];
.VAR/DM           col_exponents[N];
.VAR/PM/RAM       padding[8];
.VAR/DM/RAM       groups, node_space, bflys_per_group;
.VAR/DM/RAM       offset, rrowbase, irowbase, rcolbase, icolbase;
.VAR/DM/RAM       blk_exponent;
.VAR/DM/RAM       real_br_pointers[N];
.VAR/DM/RAM       imag_br_pointers[N];
.VAR/DM/RAM       c_real_br_pointers[N];
.VAR/DM/RAM       c_imag_br_pointers[N];
.VAR/DM/RAM       current_rrow;
.VAR/DM/RAM       current_irow;
.VAR/DM/RAM       current_rcol;
.VAR/DM/RAM       current_icol;
```

(listing continues on next page)

7 Two-Dimensional FFTs

```
.INIT       realdata: <realdata.dat>;
.INIT       imaginarydata: <imagdata.dat>;
.INIT       twid_real:  <twid_real.dat>;
.INIT       twid_imag:  <twid_imag.dat>;
.INIT       real_br_pointers: <real_ptr.dat>;
.INIT       imag_br_pointers: <imag_ptr.dat>;
.INIT       c_real_br_pointers: <c_real_p.dat>;
.INIT       c_imag_br_pointers: <c_imag_p.dat>;
.INIT       padding: 0,0,0,0,0,0,0,0;
.INIT       groups: 1;
.INIT       blk_exponent: 0;
.INIT       node_space: N_div_2;
.INIT       bflys_per_group: N_div_2;

.GLOBAL     realdata, imaginarydata, twid_real, twid_imag;
.GLOBAL     groups, bflys_per_group, node_space, offset;
.GLOBAL     rrowbase, irowbase, icolbase, rcolbase, blk_exponent;
.GLOBAL     real_br_pointers, imag_br_pointers;
.GLOBAL     c_real_br_pointers, c_imag_br_pointers;
.GLOBAL     current_rrow, current_irow, tempdata;
.GLOBAL     current_rcol, current_icol;
.GLOBAL     row_exponents, col_exponents;

.EXTERNAL   fft_start, col_fft_strt, unscr_start;
.EXTERNAL   col_unscr_start, row_final_adj, col_final_adj;

            NOP;        NOP;
            NOP;        NOP;

            CALL initialize;
            CNTR = N;               {row counter}
            DO rowloop UNTIL CE;    {do row FFTs}
               CALL fft_start;
rowloop:       NOP;

            CNTR = N;               {bit reverse rows}
            I6 = ^real_br_pointers;
            I7 = ^imag_br_pointers;
            DO unscramble UNTIL CE; {real data -> tempdata}
               CALL unscr_start;    {imaginary -> realdata}
unscramble:    NOP;
```

```
            CALL row_final_adj;
            I7 = ^col_exponents;
            CNTR = N;               {column counter}
            DO colloop UNTIL CE;    {do column FFTs}
               CALL col_init;
               CALL col_fft_strt;
colloop:       NOP;

            CNTR = N;
            I6 = ^c_imag_br_pointers;
            I7 = ^c_real_br_pointers;
            DO col_unscramble UNTIL CE;
                CALL col_unscr_start;
col_unscramble: NOP;

            CALL col_final_adj;     {final BFP adjust}

            TRAP;
```

(initialize and col_init routines shown in Listings 7.5 and 7.6)

```
.ENDMOD;
```

Listing 7.1 Main Module

7.1.3.2 Row DIF Module

The *dif_fft* module operates on row data from the data buffers *realdata* and *imaginarydata*. Rows consist of 64 sequential locations and have start addresses that are multiples of 64. Figure 7.2, on the following page, shows row data from the data buffer *realdata*.

The *dif_fft* routine (entry point at *fft_start*) is called N times for an N-by-N-point image. Each time, the routine performs an FFT on a single row of complex data, actually two rows of data, one representing the real part and one representing the imaginary part of the data. Three data memory variables keep track of the current row of data: *rrowbase* contains the start address of the current real data row, *irowbase* contains the start address of the current imaginary data row, and *offset* calculates *rrowbase* and *irowbase* for the next data row.

7 Two-Dimensional FFTs

0	1	2	3		63
64	65	66	67		127
128					191
4032					4095

Figure 7.2 Row Data

The FFT program is a looped program consisting of three nested loops. They are the *stage* loop, the *group* loop and the *butterfly* loop. This looped structure is identical to that of the DIF FFT programs found in Chapter 6. The innermost loop, the *butterfly* loop, performs the FFT calculations. A generalized DIF butterfly flow graph is shown in Figure 7.3.

Evaluation of the DFT requires multiplication by a complex exponential, W_N. This complex exponential can be divided into real and imaginary parts according to the following relationship:

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

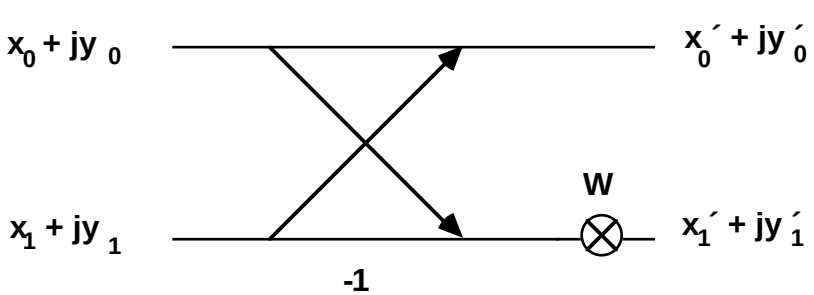

Figure 7.3 Radix-2 DIF Butterfly

The following equations calculate the real and imaginary parts of the DIF butterfly. The variables x_0 and y_0 are the real and imaginary parts of the primary node, and x_1 and y_1 are the real and imaginary parts of the dual node.

$$x'_0 = x_0 + x_1$$

$$y'_0 = y_0 + y_1$$

$$x'_1 = \mathbf{C}(x_0 - x_1) - (-\mathbf{S})\,(y_0 - y_1)$$

$$y'_1 = (-\mathbf{S})\,(x_0 - x_1) + \mathbf{C}(y_0 - y_1)$$

Listing 7.2 details the butterfly loop. The primary and dual nodes are separated in memory by the number of locations indicated by the program variable *node_space*. The primary node and dual node points are accessed throughout the program using index registers; I0 and I2 index the real and imaginary parts (respectively) of the primary node, and I1 and I3 index the real and imaginary parts of the dual node.

```
            DO bfly_loop UNTIL CE;
               AR=AX0+AY0, AX1=DM(I2,M0), MY0=PM(I4,M5);
                                          {AR=x0+x1,AX1=y0,MY0=C}
               DM(I0,M1)=AR, AR=AX1+AY1;  {x0=x0+x1,AR=y0+y1}
               DM(I2,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,M0), MY1=PM(I5,M5);
                        {MR=(x0-x1)C,AX0=next x0,MY1=(-S)}
               MR=MR-AR*MY1 (RND);
                        {MR=(x0-x1)C-(y0-y1)(-S),AY0=next x1}
               DM(I1,M1)=MR1, MR=AR*MY0 (SS);
                        {x1=(x0-x1)C-(y0-y1)(-S),MR=(y0-y1)C}
               MR=MR+MX0*MY1 (RND), AY0=DM(I1,M0);    {AY0=new x1}
                        {MR=(y0-y1)C+(x0-x1)(-S),AY1=next y1}
               DM(I3,M1)=MR1;                         {AY1=new y1}
                        {y1=(y0-y1)C+(x0-x1)(-S),check bit
growth}
bfly_loop:     AY1=DM(I3,M0);
```

Listing 7.2 DIF Butterfly Loop

The DIF algorithm accepts input in a normal order. Output of the DIF algorithm is scrambled and needs to be put through a bit-reverse algorithm to achieve normal ordering of output. For the 2D FFT, both the row and column FFTs use the DIF algorithm and the output of each requires a bit-reverse subroutine. These routines are described later under *Bit Reverse Modules*.

A complete description of the DIF FFT algorithm and its implementation on the ADSP-2100 can be found in Chapter 6. Listing 7.3 contains the complete row FFT module.

```
.MODULE   dif_fft;

{      DIF section for Row FFTs

       Calling Parameters
          realdata = Real input data normal order
          imaginarydata = Imaginary data normal order
          twid_real = Twiddle factor cosine values
          twid_imag = Twiddle factor sine values
          groups = 1
          bflys_per_group = N/2
          node_space = N/2
          rrowbase = 0
          irowbase = 4096

       Return Values
          realdata = row FFT results in bit-reversed order
          imaginarydata = column FFT results bit-reversed

       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
          realdata,imaginarydata, groups, node_space,
          bflys_per_group, irowbase, icolbase, offset
}

.CONST       N=64, N_div_2=32, log2N=6;
```

```
.EXTERNAL    realdata,imaginarydata, twid_real, twid_imag;
.EXTERNAL    offset, irowbase, rrowbase;
.EXTERNAL    groups,bflys_per_group,node_space;
.EXTERNAL    row_bfp, blk_exponent;

.ENTRY       fft_start;

fft_start:   AX0 = N;
             DM(offset) = AX0;
             CNTR=log2N;              {Initialize stage counter}
             DO stage_loop UNTIL CE;
                SB = -2;              {block exponent}
                I0 = DM(rrowbase);
                I2 = DM(irowbase);
                M2 = DM(node_space);
                I1=I0;
                MODIFY(I1,M2);        {I1 -> x1}
                I3=I2;
                MODIFY(I3,M2);        {I3 -> y1}

                CNTR=DM(groups);      {Initialize group counter}
                M5=CNTR;              {Init. twid factor modifier}

                DO group_loop UNTIL CE;
                   CNTR=DM(bflys_per_group);  {Init. bfly counter}
                   AX0=DM(I0,M0);             {AX0=x0}
                   AY0=DM(I1,M0);             {AY0=x1}
                   AY1=DM(I3,M0);             {AY1=y1}
                   DO bfly_loop UNTIL CE;
                      AR=AX0+AY0, AX1=DM(I2,M0), MY0=PM(I4,M5);
                              {AR=x0+x1,AX1=y0,MY0=C}
                      SB = EXPADJ AR;
                      DM(I0,M1)=AR, AR=AX1+AY1;
                              {x0=x0+x1,AR=y0+y1}
                      SB = EXPADJ AR;
                      DM(I2,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,M0),MY1=PM(I5,M5);
                              {MR=(x0-x1)C,AX0=next x0,MY1=(-S)}
                      MR=MR-AR*MY1 (SS);  MR=MR(RND);
                              {MR=(x0-x1)C-(y0-y1)(-S),AY0=next x1}
                      SB = EXPADJ MR1;
                      DM(I1,M1)=MR1, MR=AR*MY0 (SS);
```

(listing continues on next page)

```
                        {x1=(x0-x1)C-(y0-y1)(-S),MR=(y0-y1)C}
                MR=MR+MX0*MY1 (RND), AY0=DM(I1,M0);
                        { AY0 = new x1 }
                        {MR=(y0-y1)C+(x0-x1)(-S),AY1=next y1}
                DM(I3,M1)=MR1, SB = EXPADJ MR1;
                        { AY1 = new y1 }
                        {y1=(y0-y1)C+(x0-x1)(-S),check bit growth}
bfly_loop:      AY1=DM(I3,M0);

                MODIFY(I0,M2); {I0->x0 of 1st bfly next group}
                MODIFY(I1,M2); {I1->x1 of 1st bfly next group}
                MODIFY(I2,M2); {I2->y0 of 1st bfly next group}
group_loop:     MODIFY(I3,M2); {I3->y1 of 1st bfly next group}

            CALL row_bfp;

            SI=DM(groups);
            SR=LSHIFT SI BY 1 (LO);
            DM(groups)=SR0;     {groups=groups X 2}
            SI=DM(node_space);
            SR=LSHIFT SI BY -1 (LO);
            DM(node_space)=SR0; {node_space=node_space / 2}
stage_loop: DM(bflys_per_group)=SR0;
                                {bflys_per_group=bflys_per_group / 2}

          AX0 = DM(offset);      {calculate next fft base}
          AY0 = DM(rrowbase);
          AR = AX0 + AY0;
          I0 = AR;
          DM(rrowbase) = AR;
          AY0 = DM(irowbase);
          AR = AX0 + AY0;
          DM(irowbase) = AR;
          I2 = AR;
          AX1 = n_div_2;
          DM(node_space) = AX1;  {reset node_space}
          DM(bflys_per_group) = AX1;
          AR = 1;
          DM(groups) = AR;

          AX0 = DM(blk_exponent);
          DM(I6,M7) = AX0;
          AX0 = 0;
          DM(blk_exponent)=AX0;

          RTS;
                    .ENDMOD;
```

Listing 7.3 Row DIF Module

7.1.3.3 Column DIF Module

The *col_dif_fft* module operates on column data in the data buffers *realdata* and *tempdata*. Sequential points in a data column are not at sequential addresses. Figure 7.4 illustrates the arrangement of columns in the data

4096	4097				4159
4160					
8128					8191

buffer *tempdata*.

Figure 7.4 Column Data

Accessing data in a column format is easy with the ADSP-2100. In the first stage of a typical DIF FFT, the primary and dual nodes are separated by a node space of N/2. Because this implementation of the DIF algorithm relies on the data memory variable *node_space* to reference the spacing between primary and dual node, however, they need not be sequential. Before the start of each column FFT, *node_space* is initialized to the value 64 x N/2. As the algorithm progresses, the node spacing decrements by a

factor of 2 with each stage.

Listing 7.4 contains the complete column FFT module.

```
.MODULE   col_dif_fft;

{      DIF section for Column FFTs

       Calling Parameters
          realdata = Real input data normal order
          imaginarydata = Imaginary data normal order
          twid_real = Twiddle factor cosine values
          twid_imag = Twiddle factor sine values
          groups = 1
          bflys_per_group = N/2
          node_space = N/2
          rrowbase = 0
          irowbase = 4096

       Return Values
          realdata = row FFT results in bit reversed order
          imaginARydata = column FFT results bit reversed

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

       Altered Memory
          realdta,imaginarydata, groups, node_space
          bflys_per_group, irowbase, icolbase, offset
}

.CONST       N=64, N_div_2=32, log2N=6;

.EXTERNAL    realdata,imaginarydata, twid_real, twid_imag;
.EXTERNAL    offset, icolbase, rcolbase;
.EXTERNAL    groups,bflys_per_group,node_space;
.EXTERNAL    col_bfp, blk_exponent;

.ENTRY       col_fft_strt;

col_fft_strt: CNTR=log2N;             {Initialize stage counter}
              DO stage_loop UNTIL CE;
                 SB = -2;             {block exponent}
```

```
                I0 = DM(rcolbase);
                I2 = DM(icolbase);
                M2 = DM(node_space);
                I1=I0;
                MODIFY(I1,M2);       {I1 -> x1}
                I3=I2;
                MODIFY(I3,M2);       {I3 -> y1}

                CNTR=DM(groups);     {Initialize group counter}
                M5=CNTR;             {Init twid factor modifier}

                DO group_loop UNTIL CE;
                   CNTR=DM(bflys_per_group); {Init bfly counter}
                   AX0=DM(I0,M0);            {AX0=x0}
                   AY0=DM(I1,M0);            {AY0=x1}
                   AY1=DM(I3,M0);            {AY1=y1}
                   DO bfly_loop UNTIL CE;
                      AR=AX0+AY0, AX1=DM(I2,M0), MY0=PM(I4,M5);
                                             {AR=x0+x1,AX1=y0,MY0=C}
                      SB = EXPADJ AR;
                      DM(I0,M1)=AR, AR=AX1+AY1;
                                             {x0=x0+x1,AR=y0+y1}
                      SB = EXPADJ AR;
                      DM(I2,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,M0),MY1=PM(I5,M5);
                                             {MR=(x0-x1)C,AX0=next x0,MY1=(-S)}
                      MR=MR-AR*MY1 (SS);  MR = MR(RND);
                                             {MR=(x0-x1)C-(y0-y1)(-S),AY0=next x1}
                      SB = EXPADJ MR1;
                      DM(I1,M1)=MR1, MR=AR*MY0 (SS);
                                             {x1=(x0-x1)C-(y0-y1)(-S),MR=(y0-y1)C}
                      MR=MR+MX0*MY1 (RND), AY0=DM(I1,M0);
                                             {AY0 = new x1}
                       {MR=(y0-y1)C+(x0-x1)(-S),AY1=next y1}
                      DM(I3,M1)=MR1, SB = EXPADJ MR1;
                                             {AY1 = new y1}
                                             {y1=(y0-y1)C+(x0-x1)(-S),check bit growth}
bfly_loop:            AY1=DM(I3,M0);

                   MODIFY(I0,M2);  {I0->x0 of 1st bfly next grp}
                   MODIFY(I1,M2);  {I1->x1 of 1st bfly next grp}
                   MODIFY(I2,M2);  {I2->y0 of 1st bfly next grp}
```

(lisitng continues on next page)

```
group_loop:         MODIFY(I3,M2);  {I3->y1 of 1st bfly next grp}

                    CALL col_bfp;
                    SI=DM(groups);
                    SR=LSHIFT SI BY 1 (LO);
                    DM(groups)=SR0;       {groups=groups X 2}
                    SI=DM(node_space);
                    SR=LSHIFT SI BY -1 (LO);
                    DM(node_space)=SR0;   {node_space=node_space /
2}
                    SI=DM(bflys_per_group);
                    SR=LSHIFT SI BY -1 (LO);
stage_loop:         DM(bflys_per_group)=SR0;

                  AY0 = DM(rcolbase);
                  AR = AY0 + 1;
                  DM(rcolbase) = AR;
                  AY0 = DM(icolbase);
                  AR = AY0 + 1;
                  DM(icolbase) = AR;

                  AX0 = DM(blk_exponent); {take blk exponent for
this}
                  DM(I7,M7) = AX0;        {fft and store in array}
                  AX0 = 0;
                  DM(blk_exponent) = AX0;

                  RTS;
.ENDMOD;
```

Listing 7.4 Column DIF Module

7.1.3.4 Initialization

There are two initialization subroutines in this implementation of the two-dimensional FFT. Both of the routines are part of the main module. The first, *initialize*, performs once-only initialization of index registers, modify registers and length registers. Pointers are used for data access as well as access to twiddle factors. This routine is listed below in Listing 7.5. The twiddle factors are stored circular buffers of length 32 (for a 64-point FFT) in program memory and use the index registers I4 and I5. Index registers

I0 and I2 set up pointer access to the real and imaginary parts of the data buffers *realdata* and *imaginarydata*. In addition, the *initialize* routine declares the variables *rrowbase, irowbase,* and *offset* to their initial values.

```
{One-time only initialization. Pointers and modifiers}
{and length registers.}

initialize: L0 = 0;
            L1 = 0;
            L2 = 0;
            L3 = 0;
            L4 = N_div_2;
            L5 = N_div_2;
            L6 = 0;
            L7 = 0;

            I0 = ^realdata;
            I2 = ^imaginarydata;
            I3 = ^tempdata;
            I4 = ^twid_real;
            I5 = ^twid_imag;
            I6 = ^row_exponents;

            DM(current_rrow) = I0;
            DM(current_irow) = I2;

            DM(current_icol) = I0;
            DM(current_rcol)=  I3;

            M0 = 0;
            M1 = 1;
            M3 = 0;
            M7 = 1;

            AF = PASS 0;
            AX0 = 0;
            AY0 = 0;
            AY1 = 4096;
            DM(offset) = AY0;       {base address for rows}
            DM(rrowbase) = AY0;
            DM(irowbase) = AY1;
            AY1 = ^tempdata;
            DM(rcolbase) = AY1;
            DM(icolbase) = AY0;
```

```
            RTS;
```

Listing 7.5 Initialize Routine

The second initialization routine is *col_init*, shown in Listing 7.6. This routine initializes variables before the start of each column FFT. This initialization is necessary because the data points in each column FFT are not in sequential order.

```
{      Initialization for column DIF FFTs}

col_init:   AY1 = N_DIV_2;
            DM(bflys_per_group) = AY1; {bflys = 1}
            AY1 = 1;
            DM(groups) = AY1;          {groups = 32}
            AX1 = 2048;
            DM(node_space) = AX1;      {node_space = 64}
            M1 = 64;                   {modifier for col. points}
            L4 = N_DIV_2;
            I4 = ^twid_real;
            I5 = ^twid_imag;

            RTS;
```

Listing 7.6 Column Initialization Routine

7.1.3.5 Bit Reverse Modules

The 1D FFTs used to perform the 2D FFT use the DIF algorithm and produce frequency output points in bit-reversed addressing (scrambled) order. For a complete explanation of bit-reversing in FFTs, see Chapter 6. Bit-reversing the addresses of the output data puts the data back into sequential order (unscrambles it). Because the output for each 1D FFT needs to be bit-reversed, the bit-reverse routine is called 2N times. The bit-reverse subroutines for the row and column FFTs are nearly identical.

The ADSP-2100 has a bit-reversed addressing capability. Unscrambling a row or column requires a seed value (the bit-reversed address of the first location in the row or column) and a modify value. The modify value is the value of 2 raised to the difference between 14 (the number of address bits) and the number of bits to be reversed, i.e., 2^{14-N}. In this example of a 64x64-point FFT, the number of bits to be reversed is six, and so the modify value is 2^8 or 256.

With the bit-reverse capability of the ADSP-2100 enabled, adding the

modify value to a current address provides the correct address for the next bit-reversed sample. Because the 2D FFT needs to perform bit-reversal on 2N 1D FFTs, the program needs the seed values (first location addresses, bit-reversed) for each row and column. The seed values for the rows and columns for a 64x64-point 2D FFT are stored in buffers called *real_br_pointers*, *imag_br_pointers*, *c_real_br_pointers* and *c_imag_br_pointers* (*c_* means "column").

Listings 7.7 and 7.8 contain the row bit-reverse routine and the column bit-reverse routine, respectively.

```
.MODULE dif_unscramble;

{
      Calling Parameters
         Real and imaginary scrambled output data in inplacedata

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

      Altered Registers
         I0,I1,I4,M1,M4, AY1,CNTR

      Altered Memory
         real_results, imaginary_results
}

.VAR/DM    rownum;
.CONST     N=64, mod_value=256;        {Initialize constants}
.CONST     N_x_2 = 128, N_DIV_2 = 32;

.EXTERNAL  current_rrow, current_irow;
.EXTERNAL  real_br_pointers, imag_br_pointers;
.ENTRY     unscr_start;       {Declare entry point into module}

unscr_start:  I4=DM(current_rrow);
                        {I4->real part of 1st data point}
              M4=1;     {Modify by 2 to fetch only real data}
              L0=0;
              L4=0;
              SI = DM(I6,M4);
              I1 = SI;
```

(listing continues on next page)

```
                M1=mod_value;        {Modifier for FFT size}
                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}
                DM(current_rrow) = I4;
                CNTR = N;
                I4 = DM(current_irow);
                SI = DM(I7,M4);
                I1 = SI;

                DO bit_rev_imag UNTIL CE;
                   AY1=DM(I4,M4);    {Read imag data}
bit_rev_imag:      DM(I1,M1)=AY1;    {Place in sequential order}
                DM(current_irow) = I4;
                DIS BIT_REV;         {Disable bit-reverse}
                RTS;
.ENDMOD;
```

Listing 7.7 Row Bit-Reverse Routine

```
.MODULE col_dif_unscramble;

{
     Calling Parameters
        Real and imaginary scrambled output data in inplacedata

     Return Values
        Normal ordered real output data in real_results
        Normal ordered imag output data in imaginary_results

     Altered Registers
        I0, I1, I4, M1, M4, AY1, CNTR

     Altered Memory
        real_results, imaginary_results
}

.VAR/DM      rownum;
.CONST       N=64, mod_value=4;        {Initialize constants}
.CONST       N_x_2 = 128, N_DIV_2 = 32;
```

```
.EXTERNAL    current_rcol, current_icol;
.ENTRY       col_unscr_start;  {Declare entry point into module}

col_unscr_start: I4 = DM(current_icol);
                            {I4->real part of 1st data point}
                 M4 = 64;     {Modify by 64 to fetch next col
val}
                 M5 = 1;
                 L0 = 0;
                 L4 = 0;
                 SI = DM(I6,M5);
                 I1 = SI;
                 M1=mod_value; {Modifier for FFT size}
                 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}
                 AY0 = DM(current_icol);
                              {increment pointer to current}
                 AR = AY0 + 1; {imaginary column data}
                 DM(current_icol) = AR;

                 CNTR = N;
                 I4 = DM(current_rcol);
                 SI  = DM(I7,M5);
                 I1 = SI;

                 DO bit_rev_imag UNTIL CE;
                    AY1=DM(I4,M4);  {Read imag data}
bit_rev_imag:       DM(I1,M1)=AY1;  {Place in sequential order}
                 AY0 = DM(current_rcol);
                              {increment pointer to current}
                 AR = AY0 + 1; {real column data}
                 DM(current_rcol) = AR;
                 DIS BIT_REV;  {Disable bit-reverse}
```

```
                    RTS;
.ENDMOD;
```

Listing 7.8 Column Bit-Reverse Routine

7.1.3.6 Block Floating-Point Adjustment

Block floating-point format provides extended dynamic range of fixed-point arithmetic without the computational burdens of full floating-point arithmetic. In a block floating-point implementation, there is a single exponent for a group of mantissas. In the implementation of the 2D FFT explained in this chapter, there is an associated exponent for each of the 2N 1D FFTs performed.

There are two block floating-point adjustment routines for each 1D FFT. The first block floating-point routine normalizes 1D FFT values. The 64 real and imaginary output values are normalized to a single block exponent. At the completion of all of the rows or all of the columns the second block floating-point routine is called. This routine normalizes the output of the entire output array to a single exponent.

The first adjustment routine is called from within the stage loop of each 1D FFT. If there is any bit growth, then the stage output is adjusted according to a shift value stored in the SB register. For an N-stage FFT, there can be at most N bits of growth. For every bit of growth, this routine shifts the results of the FFT one bit and stores the amount of shift as an exponent in an appropriate buffer. The buffers used are *row_exponents* for the row FFTs and *col_exponents* for the column FFTs.

The second block floating-point adjustment routine is called after all of the row FFTs or column FFTs are complete. These routines *row_final_adj* and *col_final_adj* (for the row FFTs and column FFTs, respectively) search the buffers *col_exponents* and *row_exponents* for the largest exponent. Each row or column value is shifted by the difference between the largest exponent and the exponent associated with the particular row or column. This has the effect of scaling the entire output array to one exponent.

The block floating-point adjustment routines are shown in Listings 7.9

through 7.12. Listings 7.9 and 7.10 show the routines for the stage outputs of the row FFTs and column FFTs, respectively. Listings 7.11 and 7.12 show the routines that adjust the entire output array for the row FFTs and column FFTs, respectively.

```
.MODULE    each_row_bfp_adjust;
{
        Calling Parameters
           FFT stage results in realdata & imaginarydata

        Return Parameters
           realdata & imaginarydata adjusted for bit growth

        Altered Registers
           I0,I1,AX0,AY0,AR,MX0,MY0,MR,CNTR

        Altered Memory
           inplacereal, inplaceimag, blk_exponent
}

.EXTERNAL  realdata, blk_exponent;   {Begin declaration section}
.EXTERNAL  imaginarydata, rrowbase, irowbase;
.CONST     buffer_size = 64;
.ENTRY     row_bfp;

row_bfp:      AY0=CNTR;        {Check for last stage}
              AR=AY0-1;
              IF EQ RTS;       {If last stage, return}
              AY0=-2;
              AX0=SB;
              AR=AX0-AY0;      {Check for SB=-2}
              IF EQ RTS;       {If SB=-2, no bit growth, return}
              I0=DM(rrowbase); {I0=read pointer}
              I1=DM(rrowbase); {I1=write pointer}
              AY0=-1;
              MY0=H#4000;      {Set MY0 to shift 1 bit right}
              AR=AX0-AY0,MX0=DM(I0,M1);
                  {Check if SB=-1; Get first sample}
              IF EQ JUMP strt_shift;
                  {If SB=-1, shift block data 1 bit}
              AY0=-2;          {Set AY0 for block exponent
update}
              MY0=H#2000;      {Set MY0 to shift 2 bits right}
strt_shift:   CNTR=buffer_size - 1; {initialize loop counter}
```

(listing continues on next page)

```
               DO shift_loop UNTIL CE;  {Shift block of data}
                  MR=MX0*MY0(RND),MX0=DM(I0,M1);
                         {MR=shifted data,MX0=next value}
shift_loop:       DM(I1,M1)=MR1;   {Unshifted data=shifted data}
               MR=MX0*MY0(RND);    {Shift last data word}

               I0=DM(irowbase);       {I0=read pointer}
               I1=DM(irowbase);       {I1=write pointer}
               AY0=-1;
               MY0=H#4000;            {Set MY0 to shift 1 bit right}
               AR=AX0-AY0,MX0=DM(I0,M1);
                                      {Check if SB=-1; Get first sample}
               IF EQ JUMP i_strt_shift;
                                      {If SB=-1, shift block data 1 bit}
               AY0=-2;                {Set AY0 for block exponent update}
               MY0=H#2000;            {Set MY0 to shift 2 bits right}
i_strt_shift: CNTR=buffer_size - 1;  {initialize loop counter}
               DO i_shift_loop UNTIL CE;  {Shift block of data}
                  MR=MX0*MY0(RND),MX0=DM(I0,M1);
                                      {MR=shifted data,MX0=next value}
i_shift_loop:    DM(I1,M1)=MR1;      {Unshifted data=shifted data}
               MR=MX0*MY0(RND);       {Shift last data word}

               AY0 = DM(blk_exponent); {Update block exponent and}
```

Listing 7.9 Row BFP Adjustment Routine

```
               DM(I1,M1)=MR1,AR=AY0-AX0;
                                       {store last shifted sample}
               DM(blk_exponent)=AR;
               RTS;

.ENDMOD;
.MODULE     column_bfp_adjust;

.EXTERNAL   realdata, blk_exponent;   {Begin declaration section}
.EXTERNAL   tempdata, rcolbase, icolbase;
.CONST      buffer_size = 64;
.ENTRY      col_bfp;

col_bfp:     AY0=CNTR;        {Check for last stage}
             AR=AY0-1;
```

```
              IF EQ RTS;       {If last stage, return}
              AY0=-2;
              AX0=SB;
              AR=AX0-AY0;      {Check for SB=-2}
              IF EQ RTS;       {If SB=-2, no bit growth, return}

              I0=DM(rcolbase);            {I0=read pointer}
              I1=DM(rcolbase);            {I1=write pointer}
              M1=buffer_size;
              AY0=-1;
              MY0=H#4000;
                                          {Set MY0 to shift 1 bit right}
              AR=AX0-AY0,MX0=DM(I0,M1);
                                          {Check if SB=-1; Get first sample}
              IF EQ JUMP strt_shift;
                                          {If SB=-1, shift block data 1 bit}
              AY0=-2;                     {Set AY0 for block exponent update}
              MY0=H#2000;                 {Set MY0 to shift 2 bits right}
strt_shift:   CNTR=buffer_size - 1;       {initialize loop counter}
              DO shift_loop UNTIL CE;     {Shift block of data}
                 MR=MX0*MY0(RND),MX0=DM(I0,M1);
                                          {MR=shifted data,MX0=next value}
shift_loop:      DM(I1,M1)=MR1;           {Unshifted data=shifted data}
              MR=MX0*MY0(RND);
              DM(I1,M1)=MR1;              {Shift last data word}
                                          {store last shifted sample}

              I0=DM(icolbase);            {I0=read pointer}
              I1=DM(icolbase);            {I1=write pointer}
              AY0=-1;
              MY0=H#4000;                 {Set MY0 to shift 1 bit right}
              AR=AX0-AY0,MX0=DM(I0,M1);
                                          {Check if SB=-1; Get first sample}
              IF EQ JUMP shift_start;
                                          {If SB=-1, shift block data 1 bit}
              AY0=-2;                     {Set AY0 for block exponent update}
              MY0=H#2000;                 {Set MY0 to shift 2 bits right}
shift_start:  CNTR=buffer_size - 1;       {initialize loop counter}
              DO shft_loop UNTIL CE;      {Shift block of data}
                 MR=MX0*MY0(RND),MX0=DM(I0,M1);
                                          {MR=shifted data,MX0=next value}
shft_loop:       DM(I1,M1)=MR1;           {Unshifted data=shifted data}
              MR=MX0*MY0(RND);            {Shift last data word}
              AY0=DM(blk_exponent);       {Update block exponent and}
              DM(I1,M1)=MR1,AR=AY0-AX0;   {store last shifted sample}
              DM(blk_exponent)=AR;
```

```
            RTS;

.ENDMOD;
```

Listing 7.10 Column BFP Adjustment Routine

```
.MODULE        all_row_bfp;

{
       This module does the final adjusting of the row
       block floating point exponents, to normalize the
       entire array. Each row has an associated BFP exponent.
       This module finds the greatest magnitude exponent,
       then shifts each row by the difference of the individual
       BFP exponent and the greatest magnitude exponent.
}

.VAR/DM   largest_re;
.CONST    N=64;
.EXTERNAL row_exponents, rrowbase, irowbase, realdata, tempdata;
.ENTRY    row_final_adj;

{find the greatest of the row exponents}

row_final_adj: I0 = ^row_exponents;
               M0 = 0;
               M1 = 1;
               AX1 = 0;
               AF = PASS AX1;

               AY0 = DM(I0,M1);     {find max of 1st two values}
               AX0 = DM(I0,M1);
               AR = AX0 - AY0;      {see which is greater}

               IF GE AF = PASS AX0; {AF gets greatest of 1st two}
               IF LT AF = PASS AY0;
               AX0 = DM(I0,M1);

               CNTR = N-2;          {buffer size less first two values}
               DO findmax UNTIL CE; {find greatest in the buffer}
                  AR = AX0 - AF;
                  IF GE AF = PASS AX0;
findmax:          AX0 = DM(I0,M1);
               AR = PASS AF;        {put the largest row exponent}
               DM(largest_re) = AR; {in memory}

{shift each row by the difference between the greatest
```

```
and the individual row exponent}

               I1=^tempdata;        {address of post scrambled real data}
               I2=^realdata;        {address of post scrambled imag data}
               I0=^row_exponents;

               AY0 = DM(largest_re);
               CNTR = N;
               AX0=DM(I0,M1);             {get individual row BFP}

               DO row UNTIL CE;
                  AR = AY0 - AX0;
                                          {diff between greatest and row BFP}
                  SE = AR;
                  AF = PASS 0;            {clear AC bit}
                  CNTR = N;
                  DO row_shift UNTIL CE;  {shift the row}
                     SI = DM(I1,M0);
                                          {get next value leave pointer}
                     SR = ASHIFT SI (HI);
                     DM(I1,M1) = SR1;
                                          {put shifted value back, inc pointer}
                     SI = DM(I2,M0);      {do same for imag values}
                     SR = ASHIFT SI (HI);
row_shift:           DM(I2,M1) = SR1;
row:              AX0=DM(I0,M1);          {get indiv. row BFP}
```

```
                RTS;

.ENDMOD;
```

Listing 7.11 Row Final Exponent Adjustment Routine

```
.MODULE         all_col_bfp;

{
     This module does the final adjusting of the column
     block floating point exponents, to normalize the
     entire array. Each column has an associated BFP exponent.
     This module finds the greatest magnitude exponent,
     then shifts each column by the difference of the individual
     BFP exponent and the greatest magnitude exponent.
}
.VAR/DM         largest_ce;
.CONST          N=64;
.EXTERNAL       col_exponents, realdata, imaginarydata;
.EXTERNAL       current_rcol, current_icol;
.ENTRY          col_final_adj;

{find the greatest of the column exponents}

col_final_adj: I0 = ^col_exponents;
               M0 = 0;
               M1 = 1;
               M2 = N;              {increment by N for columns}
               AX1 = 0;
               AY1 = h#1000;
               AF = PASS AX1;
               DM(current_rcol) = AX1;
               DM(current_icol) = AY1;

               AY0 = DM(I0,M1);     {find max of 1st two values}
               AX0 = DM(I0,M1);
               AR = AX0 - AY0;      {see which is greater}

               IF GE AF = PASS AX0; {AF gets greatest of 1st two}
               IF LT AF = PASS AY0;
               AX0 = DM(I0,M1);

               CNTR = N-2;          {buffer size less first two values}
               DO findcolmax UNTIL CE;
                                    {find greatest in the buffer}
```

```
                  AR = AX0 - AF;
                  IF GE AF = PASS AX0;
findcolmax:       AX0 = DM(I0,M1);
               AR = PASS AF;        {put the largest row exponent}
               DM(largest_ce) = AR; {in memory}


{shift each row by the difference between the greatest and the individual row exponent}

               I1=DM(current_rcol);
                                    {address of post scrambled real data}
               I2=DM(current_icol);
                                    {address of post scrambled imag data}
               I0 = ^col_exponents;

               AY0 = DM(largest_ce);
               CNTR = N;
               AX0=DM(I0,M1);       {get indiv. row BFP}

               DO col UNTIL CE;
                  AR = AY0 - AX0;
                                    {diff between greatest and row BFP}
                  SE = AR;
                  AF = PASS 0;      {clear AC bit}
                  CNTR = N;
                  DO col_shift UNTIL CE;  {shift the row}
                     SI = DM(I1,M0);
                                    {get next value leave pointer}
                     SR = ASHIFT SI (HI);
                     DM(I1,M2) = SR1;
                                    {put shifted value back, incr pointer}
                     SI = DM(I2,M0);      {do same for imag values}
                     SR = ASHIFT SI (HI);
col_shift:           DM(I2,M2) = SR1;
                  AY1=DM(current_rcol);   {incr col pointer real}
                  AR = AY1 + 1;
                  I1 = AR;
                  DM(current_rcol) = AR;
                  AY1=DM(current_icol);   {incr col pointer imag}
                  AR = AY1 + 1;
                  I2 = AR;
                  DM(current_icol) = AR;
col:              AX0=DM(I0,M1);          {get next indiv. row BFP}

               RTS;

.ENDMOD;
```

Listing 7.12 Column Final Exponent Adjustment Routine

7 Two-Dimensional FFTs

7.2 BENCHMARKS

Benchmarks for the 2D FFT programs are given below. All are for a 64-by-64-point 2D FFT.

Routine	*Number of Cycles*	*Execution Time* (12.5MHz ADSP-2100A)
fft_2d (main)	597	47.8 μs
initialize	33	2.64 μs
col_init	11 (called 64 times)	0.88 μs
dif_fft	3457 (called 64 times)	277.00 μs
col_dif	3396 (called 64 times)	272.00 μs
row_bfp	287 (multiple calls)*	23.00 μs
col_bfp	288 (multiple calls)*	23.00 μs
all_row_bfp	25105	2.01 ms
all_col_bfp	25118	2.01 ms
unscr	279 (called 64 times)	22.3 μs
col_unscr	280 (called 64 times)	22.4 μs

* This routine is data-dependent; the cycle count shown is worst case.

Experimentally, the 2D FFTs of complex signals ranged from a minimum of 543001 cycles (43.4ms) to a maximum 633701 cycles (50.7ms).