Software Examples 14

14.1 OVERVIEW

This chapter provides a brief summary of the development process that you use to create executable programs for the ADSP-2100 family processors. The summary is followed by a number of software examples that can give you an idea of how to write your own applications.

The software examples presented in this chapter are used a variety of DSP operations. The FIR filter and cascaded biquad IIR filter are general filter algorithms that can be tailored to many applications. Matrix multiplication is used in image processing and other areas requiring vector operations. The sine function is required for many scientific calculations. The FFT (fast Fourier transform) has wide application in signal analysis. Each of these examples is described in greater detail in *Digital Signal Processing Applications Using The ADSP-2100 Family, Volume 1*, available from Prentice Hall. They are presented here to show some aspects of typical programs.

The FFT example is a complete program, showing a subroutine that performs the FFT and a main calling program that initializes registers and calls the FFT subroutine as well as an auxiliary routine.

Each of the other examples is shown as a subroutine in its own module. The module starts with a .MODULE directive that names the module and ends with the .ENDMOD directive. The subroutine can be called from a program in another module that declares the starting label of the subroutine as an external symbol. This is the same label that is declared with the .ENTRY directive in the subroutine module. The last instruction in each subroutine is the RTS instruction, which returns control to the calling program.

Each module is prefaced by a comment block that provides the following information:

Calling Parameters	Register values that the calling program must set before calling the subroutine
Return Values	Registers that hold the results of the subroutine
Altered Registers	Registers used by the subroutine. The calling program must save them before calling the subroutine and restore them afterward if it needs to preserve their values.
Computation Time	The number of instruction cycles needed to perform the subroutine

14.2 SYSTEM DEVELOPMENT PROCESS

The ADSP-2100 family of processors is supported by a complete set of development tools. Programming aids and processor simulators facilitate software design and debug. In-circuit emulators and demonstration boards help in hardware prototyping.

The software development system includes several programs: System Builder, Assembler, Linker, PROM Splitter, Simulators and C Compiler with Runtime Library. These programs are described in detail in the *ADSP-2100 Family Assembler Tools & Simulator Manual*, *ADSP-2100 Family C Tools Manual*, and *ADSP-2100 Family C Runtime Library Manual*.

Figure 14.1 shows a flow chart of the system development process.

The development process begins with the task of describing the hardware environment for the development software. You create a system specification file using a text editor. This file contains simple directives that describe the locations of memory and I/O ports, the type of processor, and the state of the MMAP pin in the target hardware configuration. The system builder reads this file and generates an architecture description file which passes information to the linker, simulator and emulator.

You begin code generation by creating source code files in C language or

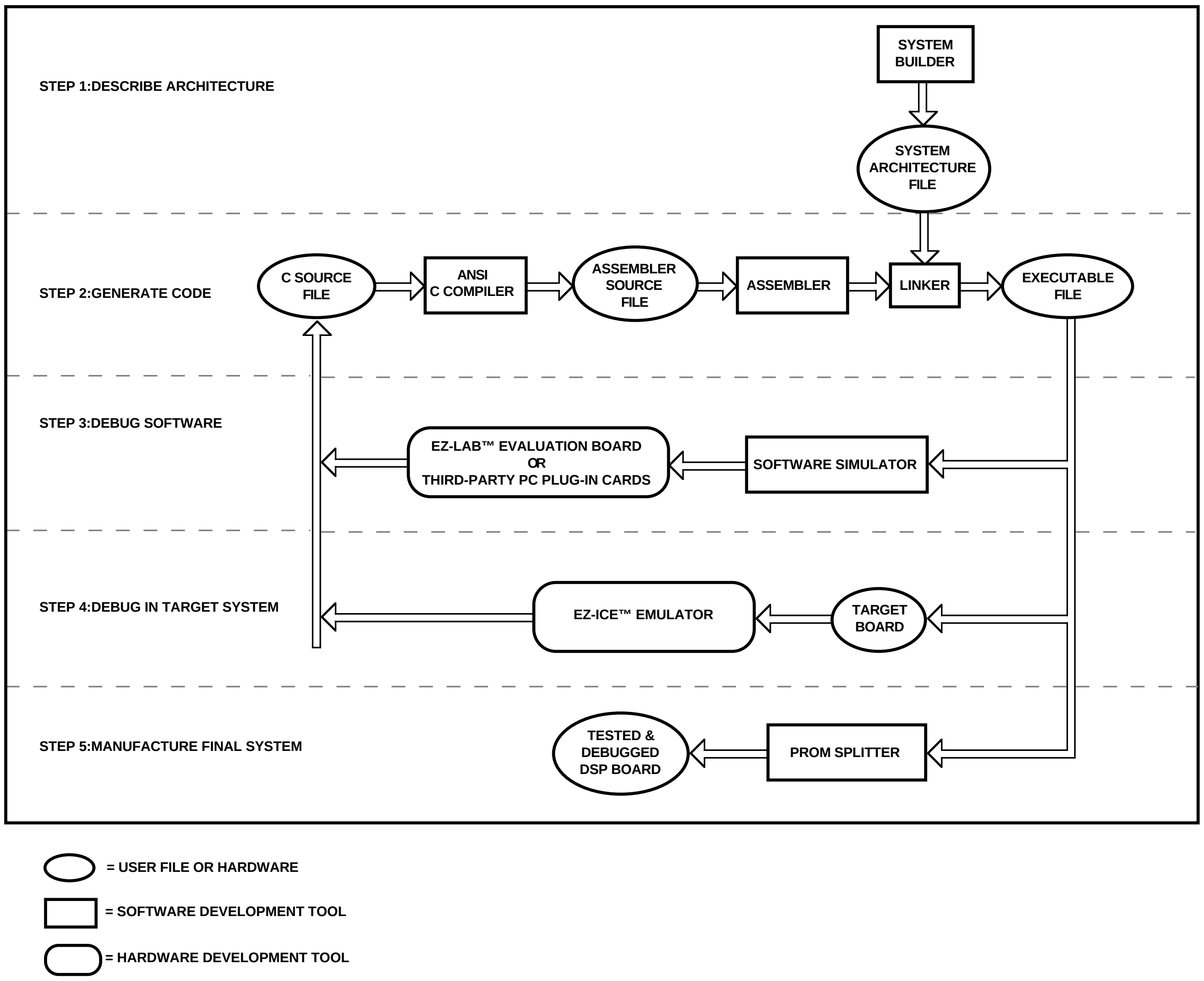

Figure 14.1 ADSP-2100 Family System Development Process

assembly language. A module is a unit of assembly language comprising a main program, subroutine, or data variable declarations. C programmers write C language files and use the C compiler to create assembly code modules from them. Assembly language programmers write assembly code modules directly. Each code module is assembled separately by the assembler.

The linker links several modules together to form an executable program (memory image file). The linker reads the target hardware information from the architecture description file to determine appropriate addresses for code and data. In the assembly modules you may specify each code/data fragment as completely relocatable, relocatable within a defined memory segment, or non-relocatable (placed at an absolute address).

The linker places non-relocatable code or data modules at the specified memory addresses, provided the memory area has the correct attributes. Relocatable objects are placed at addresses selected by the linker. The linker generates a memory image file containing a single executable program which may be loaded into a simulator or emulator for testing.

The simulator provides windows that display different portions of the hardware environment. To replicate the target hardware, the simulator configures its memory according to the architecture description file generated by the system builder, and simulates memory-mapped I/O ports. This simulation allows you to debug the system and analyze performance before committing to a hardware prototype.

After fully simulating your system and software, you can use an EZ-ICE in-circuit emulator in the prototype hardware to test circuitry, timing, and real-time software execution.

The PROM splitter software tool translates the linker-output program (memory image file) into an industry-standard file format for a PROM programmer. Once you program the code in PROM devices and install an ADSP-21xx processor into your prototype, it is ready to run.

14.3 SINGLE-PRECISION FIR TRANSVERSAL FILTER

An FIR transversal filter structure can be obtained directly from the equation for discrete-time convolution.

$$y(n) = \sum_{k=0}^{N-1} h_k(n)\, x(n-k)$$

In this equation, x(n) and y(n) represent the input to and output from the filter at time *n*. The output y(n) is formed as a weighted linear combination of the current and past input values of x, x(n–k). The weights, $h_k(n)$, are the transversal filter coefficients at time *n*. In the equation, x(n–k) represents the past value of the input signal "contained" in the (k+1)th tap of the transversal filter. For example, x(n), the present value of the input signal, would correspond to the first tap, while x(n–42) would correspond to the forty-third filter tap.

The subroutine that realizes the sum-of-products operation used in computing the transversal filter is shown in Listing 14.1.

```
.MODULE fir_sub;

{
   FIR Transversal Filter Subroutine

   Calling Parameters
        I0 —> Oldest input data value in delay line
        L0 = Filter length (N)
        I4 —> Beginning of filter coefficient table
        L4 = Filter length (N)
        M1,M5 = 1
        CNTR = Filter length - 1 (N-1)

   Return Values
        MR1 = Sum of products (rounded and saturated)
        I0 —> Oldest input data value in delay line
        I4 —> Beginning of filter coefficient table

   Altered Registers
        MX0,MY0,MR

   Computation Time
        N - 1 + 5 + 2 cycles

   All coefficients and data values are assumed to be
   in 1.15 format.
}

.ENTRY  fir;

fir:    MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
        DO sop UNTIL CE;
sop:       MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(RND);
        IF MV SAT MR;
        RTS;
.ENDMOD;
```

Listing 14.1 Single-Precision FIR Transversal Filter

14 Software Examples

14.4 CASCADED BIQUAD IIR FILTER

A second-order biquad IIR filter section is represented by the transfer function (in the z-domain):

$$H(z) = Y(z)/X(z) = (B_0 + B_1z^{-1} + B_2z^{-2})/(1 + A_1z^{-1} + A_2z^{-2})$$

where A_1, A_2, B_0, B_1 and B_2 are coefficients that determine the desired impulse response of the system H(z). The corresponding difference equation for a biquad section is:

$$Y(n) = B_0X(n) + B_1X(n-1) + B_2X(n-2) - A_1Y(n-1) - A_2Y(n-2)$$

Higher-order filters can be obtained by cascading several biquad sections with appropriate coefficients. The biquad sections can be scaled separately and then cascaded in order to minimize the coefficient quantization and the recursive accumulation errors.

A subroutine that implements a high-order filter is shown in Listing 14.2. A circular buffer in program memory contains the scaled biquad coefficients. These coefficients are stored in the order: B_2, B_1, B_0, A_2 and A_1 for each biquad. The individual biquad coefficient groups must be stored in the order that the biquads are cascaded.

```
.MODULE     biquad_sub;

{     Nth order cascaded biquad filter subroutine

      Calling Parameters:

         SR1=input X(n)
         I0 —> delay line buffer for X(n-2), X(n-1),
            Y(n-2), Y(n-1)
         L0 = 0
         I1 —> scaling factors for each biquad section
         L1 = 0  (in the case of a single biquad)
         L1 = number of biquad sections
            (for multiple biquads)
         I4 —> scaled biquad coefficients
         L4 = 5 x [number of biquads]
         M0, M4 = 1
         M1 = -3
         M2 = 1 (in the case of multiple biquads)
         M2 = 0 (in the case of a single biquad)
         M3 = (1 - length of delay line buffer)

      Return Value:
         SR1 = output sample Y(n)

      Altered Registers:
         SE, MX0, MX1, MY0, MR, SR

      Computation Time (with N even):
         ADSP-2101/2102: (8 x N/2) + 5 cycles
         ADSP-2100/2100A: (8 x N/2) + 5 + 5 cycles

      All coefficients and data values are assumed to
      be in 1.15 format
}

.ENTRY      biquad;

biquad:     CNTR = number_of_biquads
            DO sections UNTIL CE;   {Loop once for each biquad}
               SE=DM(I1,M2);        {Scale factor for biquad}
               MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MX0*MY0(SS), MX1=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX1*MY0(SS), MY0=PM(I4,M4);
               MR=MR+SR1*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
               DM(I0,M0)=MX1, MR=MR+MX0*MY0(RND);
sections:      DM(I0,M0)=SR1, SR=ASHIFT MR1 (HI);
            DM(I0,M0)=MX0;
            DM(I0,M3)=SR1;
            RTS;
.ENDMOD;
```

Listing 14.2 Cascaded Biquad IIR Filter

14.5 SINE APPROXIMATION

The following formula approximates the sine of the input variable x:

$$\sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$$

The approximation is accurate for any value of x from 0˚ to 90˚ (the first quadrant). However, because sin(–x) = –sin(x) and sin(x) = sin(180˚ – x), you can infer the sine of any angle from the sine of an angle in the first quadrant.

The routine that implements this sine approximation, accurate to within two LSBs, is shown in Listing 14.3. This routine accepts input values in 1.15 format. The coefficients, which are initialized in data memory in 4.12 format, have been adjusted to reflect an input value scaled to the maximum range allowed by this format. On this scale, 180˚ equals the maximum positive value, 0x7FFF, and –180˚ equals the maximum negative value, 0x8000.

The routine shown in Listing 14.3 first adjusts the input angle to its equivalent in the first quadrant. The sine of the modified angle is calculated by multiplying increasing powers of the angle by the appropriate coefficients. The result is adjusted if necessary to compensate for the modifications made to the original input value.

```
.MODULE  Sin_Approximation;

{
   Sine Approximation
         Y = Sin(x)

   Calling Parameters
         AX0 = x in scaled 1.15 format
         M3 = 1
         L3 = 0

   Return Values
         AR = y in 1.15 format

   Altered Registers
         AY0,AF,AR,MY1,MX1,MF,MR,SR,I3

   Computation Time
         25 cycles
}

.VAR/DM  sin_coeff[5];

.INIT    sin_coeff : 0x3240, 0x0053, 0xAACC, 0x08B7, 0x1CCE;

.ENTRY   sin;

sin:     I3=^sin_coeff;                {Pointer to coeff. buffer}
         AY0=0x4000;
         AR=AX0, AF=AX0 AND AY0;       {Check 2nd or 4th quad.}
         IF NE AR=-AX0;                {If yes, negate input}
         AY0=0x7FFF;
         AR=AR AND AY0;                {Remove sign bit}
         MY1=AR;
         MF=AR*MY1 (RND), MX1=DM(I3,M3);  {MF = x²}
         MR=MX1*MY1 (SS), MX1=DM(I3,M3);  {MR = C₁x}
         CNTR=3;
         DO approx UNTIL CE;
            MR=MR+MX1*MF (SS);
approx:     MF=AR*MF (RND), MX1=DM(I3,M3);
         MR=MR+MX1*MF (SS);
         SR=ASHIFT MR1 BY 3 (HI);
         SR=SR OR LSHIFT MR0 BY 3 (LO);   {Convert to 1.15 format}
         AR=PASS SR1;
         IF LT AR=PASS AY0;            {Saturate if needed}
         AF=PASS AX0;
         IF LT AR=-AR;                 {Negate output if needed}
         RTS;
.ENDMOD;
```

Listing 14.3 Sine Approximation

14.6 SINGLE-PRECISION MATRIX MULTIPLY

The routine presented in this section multiplies two input matrices: X, an RxS (R rows, S columns) matrix stored in data memory and Y, an SxT (S rows, T columns) matrix stored in program memory. The output Z, an RxT (R rows, T columns) matrix, is written to data memory.

The routine is shown in Listing 14.4. It requires a number of registers to be initialized, as listed in the “Calling Parameters” section of the initial comment. SE must contain the value necessary to shift the result of each multiplication into the desired format. For example, SE would be set to zero to obtain a matrix of 1.31 values from the multiplication of two matrices of 1.15 values.

```
.MODULE     matmul;

{
            Single-Precision Matrix Multiplication

                        S
               Z(i,j) = ∑ [X(i,k) × Y(k,j)]   i=0 to R; j=0 to T
                        k=0

               X is an RxS matrix
               Y is an SxT matrix
               Z is an RxT matrix

            Calling Parameters
               I1 —> Z buffer in data memory                 L1 = 0
               I2 —> X, stored by rows in data memory        L2 = 0
               I6 —> Y, stored by rows in program memory     L6 = 0
               M0 = 1          M1 = S
               M4 = 1          M5 = T
               L0,L4,L5 = 0
               SE = Appropriate scale value
               CNTR = R

            Return Values
               Z Buffer filled by rows

            Altered Registers
               I0,I1,I2,I4,I5,MR,MX0,MY0,SR

            Computation Time
               ((S + 8) × T + 4) × R + 2 + 2 cycles
}

.ENTRY      spmm;

spmm:    DO row_loop UNTIL CE;
            I5=I6;                      {I5 = start of Y}
            CNTR=M5;
            DO column_loop UNTIL CE;
               I0=I2;                   {Set I0 to current X row}
               I4=I5;                   {Set I4 to current Y col}
               CNTR=M1;
               MR=0, MX0=DM(I0,M0), MY0=PM(I4,M5); {Get 1st data}
               DO element_loop UNTIL CE;
element_loop:     MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0),
MY0=PM(I4,M5);
               SR=ASHIFT MR1 (HI), MY0=DM(I5,M4);  {Update I5}
               SR=SR OR LSHIFT MR0 (LO);           {Finish shift}
column_loop:   DM(I1,M0)=SR1;                      {Save output}
row_loop:   MODIFY(I2,M1);              {Update I2 to next X row}
         RTS;
.ENDMOD;
```

Listing 14.4 Single-Precision Matrix Multiply

14.7 RADIX-2 DECIMATION-IN-TIME FFT

The FFT program includes three subroutines. The first subroutine scrambles the input data (places the data in bit-reversed address order), so that the FFT output will be in the normal, sequential order. The next subroutine computes the FFT and the third scales the output data to maintain the block floating-point data format.

The program is contained in four modules. 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 calls the FFT and bit reversal subroutines. The FFT module calls the data scaling subroutine.

The FFT is performed in place; that is, the outputs are written to the same buffer that the inputs are read from.

14.7.1 Main Module

The dit_fft_main module is shown in Listing 14.5. 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 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. A four-location buffer called padding is placed at the end of inplaceimag to allow data accesses to exceed the buffer length. This buffer assists in debugging but is not necessary in a real system. Variables (one-location buffers) named groups, bflys_per_group, node_space and blk_exponent are declared last.

The real parts (cosine values) of the twiddle factors are stored in the buffer twid_real. This buffer is initialized from the file twid_real.dat. Likewise, twid_imag.dat values initialize the twid_imag buffer that stores the sine values of the twiddle factors. In an actual system, the hardware would be set up to initialize these memory locations.

The variable called groups is initialized to N_div_2, and bflys_per_group and node_space are each initialized to 2 because there are two butterflies per group in the second stage of the FFT. The blk_exponent variable is initialized to zero. This exponent value is updated when the output data is scaled.

After the initializations are complete, 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.

```
.MODULE/ABS=4           dit_fft_main;
.CONST                  N=1024, N_div_2=512; {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], padding
[4];
.VAR/DM/RAM/ABS=H#1000  inputreal [N], inputimag [N];
.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: 2;
.INIT       node_space: 2;
.INIT       blk_exponent: 0;
.INIT       padding: 0,0,0,0;          {Zeros after inplaceimag}

.GLOBAL     twid_real, twid_imag;
.GLOBAL     inplacereal, inplaceimag;
.GLOBAL     inputreal, inputimag;
.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 14.5 Main Module, Radix-2 DIT FFT

14.7.2 DIT FFT Subroutine

The radix-2 DIT FFT routine is shown in Listing 14.6. The constants N and log2N 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.

The first and last stages of the FFT are performed outside of the loop that executes all the other stages. Treating the first and last stages individually allows them to be executed faster. In the first stage, there is only one butterfly per group, so the butterfly loop is unnecessary, and the twiddle factors are all either 1 or 0, so no multiplications are necessary. In the last stage, there is only one group, so the group loop is unnecessary, as are the setup operations for the next stage.

```
{1024 point DIT radix 2 FFT}
{Block Floating Point Scaling}

.MODULE     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, sequential order
         inplaceimag=imag. FFT results, 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, nover2=512, nover4=256;

.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 - 2;   {Initialize stage counter}
            M0=0;
            M1=1;
            L1=0;
            L2=0;
            L3=0;
            L4=%twid_real;
            L5=%twid_imag;
            L6=0;
            SB=-2;

{——————  STAGE 1 ——————}

            I0=^inplacereal;
            I1=^inplacereal + 1;
            I2=^inplaceimag;
            I3=^inplaceimag + 1;
            M2=2;

            CNTR=nover2;
            AX0=DM(I0,M0);
            AY0=DM(I1,M0);
            AY1=DM(I3,M0);

            DO group_lp UNTIL CE;
               AR=AX0+AY0, AX1=DM(I2,M0);
               SB=EXPADJ AR, DM(I0,M2)=AR;
               AR=AX0-AY0;
               SB=EXPADJ AR;
               DM(I1,M2)=AR, AR=AX1+AY1;
               SB=EXPADJ AR, DM(I2,M2)=AR;
               AR=AX1-AY1, AX0=DM(I0,M0);
               SB=EXPADJ AR, DM(I3,M2)=AR;
               AY0=DM(I1,M0);
group_lp:      AY1=DM(I3,M0);
            CALL bfp_adj;

{——————————STAGES 2 TO N-1———————————————————}

         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}
            SI=DM(groups);
            SR=ASHIFT SI BY -1(LO); {groups / 2}
            DM(groups)=SR0;         {groups=groups / 2}
            CNTR=SR0;               {CNTR=group counter}
            M4=SR0;                 {M4=twiddle factor modifier}
            M2=DM(node_space);      {M2=node space modifier}
            I1=I0;
            MODIFY(I1,M2);          {I1 ->y0 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=bfly count}
               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}
stage_loop: DM(bflys_per_group)=SR0;
```

```
                                {bflys_per_group=bflys_per_group / 2}

{———— LAST STAGE —————}

         I0=^inplacereal;
         I1=^inplacereal+nover2;
         I2=^inplaceimag;
         I3=^inplaceimag+nover2;

         CNTR=nover2;
         M2=DM(node_space);
         M4=1;
         I4=^twid_real;
         I5=^twid_imag;

         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_lp 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_lp:    SB=EXPADJ AR,DM(I2,M1)=AR;      {Check for bit growth}

         CALL bfp_adj;

         RTS;
.ENDMOD;
```

Listing 14.6 Radix-2 DIT FFT Routine, Conditional Block Floating-Point

14.7.3 Bit-Reverse Subroutine

The bit-reversal routine, called scramble, puts the input data in bit-reversed order so that the results will be in sequential order. This routine uses the bit-reverse capability of the ADSP-2100 family processors.

```
.MODULE  dit_scramble;

{  Calling Parameters
         Sequentially ordered input data in inputreal

   Return Values
         Scrambled input data in inplacereal

   Altered Registers
         I0,I4,M0,M4,AY1

   Altered Memory
         inplacereal
}

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

.EXTERNAL   inputreal, inplacereal;

.ENTRY      scramble;


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

Listing 14.7 Bit-Reverse Routine (Scramble)

14.7.4 Block Floating-Point Scaling Subroutine

The bfp_adj routine checks the FFT output data for bit growth and scales the entire set of data if necessary. This check prevents data overflow for each stage in the FFT. The routine, shown in Listing 14.8, uses the exponent detection capability of the shifter.

```
.MODULE  dit_radix_2_bfp_adjust;

{  Calling Parameters
      Radix-2 DIT FFT stage results in inplacereal and inplaceimag

   Return Parameters
      inplacereal and inplaceimag adjusted for bit growth

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

   Altered Memory
      inplacereal, inplaceimag, blk_exponent
}

.CONST      Ntimes2 = 2048;
.EXTERNAL   inplacereal, blk_exponent; {Begin declaration section}

.ENTRY      bfp_adj;

bfp_adj:     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=^inplacereal;       {I0=read pointer}
             I1=^inplacereal;       {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 1st sample}
            IF EQ JUMP strt_shift;
                              {If SB=-1, shift block data 1 bit}
            AX0=-2;           {Set AX0 for block exponent update}
            MY0=H#2000;          {Set MY0 to shift 2 bits right}
strt_shift: CNTR=Ntimes2 - 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);           {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 14.8 Radix-2 Block Floating-Point Scaling Routine