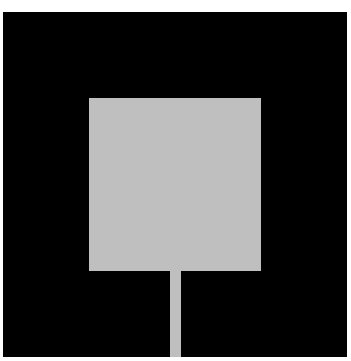
ADSP-21000 Family

Application Handbook Volume 1



ANALOG DEVICES

ADSP-21000 Family
Application Handbook Volume 1

PRINTED IN U.S.A.

Printing History
FIRST EDITION 5/94

For marketing information or Applications Engineering assistance, contact your local Analog Devices sales office or authorized distributor.

If you have suggestions for how the ADSP-2100 Family development tools or documentation can better serve your needs, or you need Applications Engineering assistance from Analog Devices, please contact:

Analog Devices, Inc.
DSP Applications Engineering
One Technology Way
Norwood, MA 02062-9106
Tel: (617) 461-3672
Fax: (617) 461-3010
e-mail: dsp_applications@analog.com

The System IC Products Division runs a Bulletin Board Service that can be reached at speeds up to 14,400 baud, no parity, 8 bits data, 1 stop bit, dialing (617) 461-4258. This BBS supports: V.32bis, error correction (V.42 and MNP classes 2, 3, and 4), and data compression (V.42bis and MNP class 5)

The System IC Products Division Applications Group maintains an Internet FTP site. Login as anonymous using your email address for your password. Type (from your UNIX prompt):

ftp ftp.analog.com (or type: ftp 137.71.23.11)

For additional marketing information, call (617) 461-3881 in Norwood MA, USA.

Literature

ADSP-21000 FAMILY MANUALS

ADSP-21020 User's Manual
ADSP-21000 SHARC Preliminary Users Manual
Complete description of processor architectures and system interfaces.

ADSP-21000 Family Assembler Tools & Simulator Manual
ADSP-21000 Family C Tools Manual
ADSP-21000 Family C Runtime Library Manual
Programmer's references.

ADSP-21020 EZ-ICE Manual
ADSP-21020 EZ-LAB Manual
User's manuals for in-circuit emulators and demonstration boards.

SPECIFICATION INFORMATION

ADSP-21020 Data Sheet
ADSP-2106 SHARC Preliminary Data Sheet
ADSP-21000 Family Development Tools Data Sheet

Contents

Contents

CHAPTER 2 TRIGONOMETRIC, MATHEMATICAL & TRANSCENDENTAL FUNCTIONS

Contents

Contents

Contents

Contents

CHAPTER 8 GRAPHICS

CHAPTER 9 IMAGE PROCESSING

Contents

FIGURES

Contents

TABLES

Contents

LISTINGS

Contents

Introduction 1

This applications handbook is intended to help you get a quick start in developing DSP applications with ADSP-21000 Family digital signal processors.

This chapter includes a summary of available resources and an introduction to the ADSP-21000 Family architecture. (Complete architecture and programming details are found in each processor's data sheet, the *ADSP-21060 SHARC User's Manual*, and the *ADSP-21020 User's Manual*.) The next eight chapters describe commonly used DSP algorithms and their implementations on ADSP-21000 family DSPs. The last chapter shows you how to build a bootstrap program downloader using the ADSP-21020 built-in JTAG port.

1.1 USAGE CONVENTIONS

- Code listings, assembly language instructions and labels, commands typed on an operating system shell command line, and file names are printed in the `Courier` font.

- Underlined variables are vectors: $\underline{V}$

1.2 DEVELOPMENT RESOURCES

This section discusses resources available from Analog Devices to help you develop applications using ADSP-21000 Family digital signal processors.

1.2.1 Software Development Tools

A full set of software tools support ADSP-21000 family program development, including an assembler, linker, simulator, PROM splitter, and C Compiler. The development tools also include libraries of assembly language modules and C functions. See the *ADSP-21000 Family Assembler Tools & Simulator Manual*, the *ADSP-21000 Family C Tools Manual*, and the *ADSP-21000 Family C Runtime Library Manual* for complete details on the development tools.

1 Introduction

1.2.2 Hardware Development Tools

Analog Devices offers several systems that let you test your programs on real hardware without spending time hardware prototyping, as well as help you debug your target system hardware.

1.2.2.1 EZ-LAB

EZ-LAB® evaluation boards are complete ADSP-210xx systems that include memory, an audio codec, an analog interface, and expansion connectors on a single, small printed-circuit board. Several programs are included that demonstrate signal processing algorithms. You can download your own programs to the EZ-LAB from your IBM-PC compatible computer.

EZ-LAB connects with EZ-ICE (described in the next section) and an IBM-PC compatible to form a high-speed, interactive DSP workstation that lets you debug and execute your software without prototype hardware.

EZ-LAB is also available bundled with the software development tools in the EZ-KIT packages. Each member of the ADSP-21000 family is supported by its own EZ-LAB.

1.2.2.2 EZ-ICE

EZ-ICE® in-circuit emulators give you an affordable alternative to large dedicated emulators without sacrificing features. The EZ-ICE software runs on an IBM-PC and gives you a debugging environment very similar to the ADSP-210xx simulator. The EZ-ICE probe connects to the PC with an ISA plug-in card and to the target system through a test connector on the target. EZ-ICE communicates to the target processor through the processor's JTAG test access port. Your software runs on your hardware at full speed in real time, which simplifies hardware and software debugging.

1.2.3 Third Party Support

Several third party companies also provide products that support ADSP-21000 family development; contact Analog Devices for a complete list. Here are a few of the products available as of this writing:

- Spectron SPOX Real-time Operating System
- Comdisco Signal Processing Worksystem
- Loughborough Sound Images/Spectrum Processing PC Plug-in Board
- Momentum Data Systems Filter Design Software (FDAS)
- Hyperceptions Hypersignal Workstation

1.2.4 DSPatch

DSPatch is Analog Devices award-winning DSP product support newsletter. Each quarterly issue includes

- applications feature articles
- stories about customers using ADI DSPs in consumer, industrial and military products
- new product announcements
- product upgrade announcements

and features as regular columns

- *Q & A*—tricks and tips from the Application Engineering staff
- *C Programming*—a popular series of articles about programming DSPs with the C language.

1.2.5 Applications Engineering Support

Analog Devices' expert staff of Applications Engineers are available to answer your technical questions.

- To speak to an Applications Engineer, Monday to Friday 9am to 5pm EST, call (617) 461-3672.
- You can send email to `dsp_applications@analog.com` .
- Facsimiles may be sent to (617) 461-3010.
- You may log in to the DSP Bulletin Board System [8:1:N:1200/2400/4800/9600/14,400] at (617) 461-4258, 24 hours a day.
- The files on the DSP BBS are also available by anonymous ftp, at `ftp.analog.com (132.71.32.11)`, in the directory `/pub/dsp` .
- Postal mail may be sent to "DSP Applications Engineering, Three Technology Way, PO Box 9106, Norwood, MA, 02062-2106."

Technical support is also available for Analog Devices Authorized Distributers and Field Applications Offices.

1 Introduction

1.2.6 ADSP-21000 Family Classes

Applications Engineering regularly offers a course in ADSP-21000 family architecture and programming. Please contact Applications Engineering for a schedule of upcoming courses.

1.3 ADSP-21000 FAMILY: THE SIGNAL PROCESSING SOLUTION

1.3.1 Why DSP?

Digital signal processors are a special class of microprocessors that are optimized for computing the real-time calculations used in signal processing. Although it is possible to use some fast general-purpose microprocessors for signal processing, they are not optimized for that task. The resulting design can be hard to implement and costly to manufacture. In contrast, DSPs have an architecture that simplifies application designs and makes low-cost signal processing a reality.

The kinds of algorithms used in signal processing can be optimized if they are supported by a computer architecture specifically designed for them. In order to handle digital signal processing tasks efficiently, a microprocessor must have the following characteristics:

- fast, flexible computation units
- unconstrained data flow to and from the computation units
- extended precision and dynamic range in the computation units
- dual address generators
- efficient program sequencing and looping mechanisms

1.3.2 Why Floating-Point?

A processor's data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. However, ease-of-use and time-to-market considerations are often equally important.

1.3.2.1 Precision

The precision of converters has been improving and will continue to increase. In the past few years, average precision requirements have risen by several bits and the trend is for both precision and sampling rates to increase.

1.3.2.2 Dynamic Range

Traditionally, compression and decompression algorithms have operated on signals of known bandwidth. These algorithms were developed to behave regularly, to keep costs down and implementations easy. Increasingly, the trend in algorithm development is to remove constraints on the regularity and dynamic range of intermediate results. Adaptive filtering and imaging are two applications requiring wide dynamic range.

1.3.2.3 Signal-To-Noise Ratio

Radar, sonar, and even commercial applications (like speech recognition) require a wide dynamic range to discern selected signals from noisy environments.

1.3.2.4 Ease-Of-Use

Ideally, floating-point digital signal processors should be easier to use and allow a quicker time-to-market than DSPs that do not support floating-point formats. If the floating-point processor's architecture is designed properly, designers can spend time on algorithm development instead of assembly coding, code paging, and error handling. The following features are hallmarks of a good floating-point DSP architecture:

- consistency with IEEE workstation simulations
- elimination of scaling
- high-level language (C, ADA) programmability
- large address spaces
- wide dynamic range

1.3.3 Why ADSP-21000 Family?

The ADSP-21020 and ADSP-21060 are the first members of Analog Devices' ADSP-21000 family of floating-point digital signal processors (DSPs). The ADSP-21000 family architecture meets the five central requirements for DSPs:

- Fast, flexible arithmetic computation units
- Unconstrained data flow to and from the computation units
- Extended precision and dynamic range in the computation units
- Dual address generators
- Efficient program sequencing

1 Introduction

1.3.3.1 Fast & Flexible Arithmetic

The ADSP-210xx can execute all instructions in a single cycle. It provides one of the fastest cycle times available and the most complete set of arithmetic operations, including Seed 1/X, Seed 1/R(x), Min, Max, Clip, Shift and Rotate, in addition to the traditional multiplication, addition, subtraction and combined addition/subtraction. It is IEEE floating-point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

1.3.3.2 Unconstrained Data Flow

The ADSP-210xx has a Harvard architecture combined with a 10-port, 16 word data register file. In every cycle, **all** of these operations can be executed:

- the register file can read or write two operands off-chip
- the ALU can receive two operands
- the multiplier can receive two operands
- the ALU and multiplier can produce two results (three, if the ALU operation is a combined addition/subtraction)

The processors' 48-bit orthogonal instruction word supports parallel data transfer and arithmetic operations in the same instruction.

1.3.3.3 Extended IEEE-Floating-Point Support

All members of the ADSP-21000 family handle 32-bit IEEE floating-point format, 32-bit integer and fractional formats (twos-complement and unsigned), and an extended-precision 40-bit IEEE floating-point format. These processors carry extended precision throughout their computation units, limiting intermediate data truncation errors. The fixed-point formats have an 80-bit accumulator for true 32-bit fixed-point computations.

1.3.3.4 Dual Address Generators

The ADSP-210xx has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus and bit-reverse operations are supported, without constraints on buffer placement.

1.3.3.5 Efficient Program Sequencing

In addition to zero-overhead loops, the ADSP-210xx supports single-cycle setup and exit for loops. Loops are nestable (six levels in hardware) and interruptable. The processor also supports delayed and non-delayed branches.

1.4 ADSP-21000 FAMILY ARCHITECTURE OVERVIEW

The following sections summarize the basic features of the ADSP-21020 architecture. These features are also common to the ADSP-21060 SHARC processor; SHARC-specific enhancements to the base architecture are discussed in the next section.

1.4.1 ADSP-21000 Family Base Architecture

All members of the ADSP-21000 Family have the same base architecture. The ADSP-21060 has advanced features built on to this base, but retains code compatibility with the ADSP-21020 processor. The key features of the base architecture are:

- *Independent, Parallel Computation Units*
  The arithmetic/logic unit (ALU), multiplier, and shifter perform single-cycle instructions. The three units are arranged in parallel, maximizing computational throughput. Single multifunction instructions execute parallel ALU and multiplier operations. These computation units support IEEE 32-bit single-precision floating-point, extended precision 40-bit floating-point, and 32-bit fixed-point data formats.

- *Data Register File*
  A general-purpose data register file transfers data between the computation units and the data buses, and for storing intermediate results. This 10-port, 32-register (16 primary, 16 secondary) register file, combined with the ADSP-21000 Harvard architecture, allows unconstrained data flow between computation units and memory.

- *Single-Cycle Fetch of Instruction & Two Operands*
  The ADSP-210xx features an enhanced Harvard architecture in which the data memory (DM) bus transfers data and the program memory (PM) bus transfers both instructions and data (see Figure 1.1). With its separate program and data memory buses and on-chip instruction cache, the processor can simultaneously fetch two operands and an instruction (from the cache) in a single cycle.

- *Instruction Cache*
  The ADSP-210xx includes a high performance instruction cache that enables three-bus operation for fetching an instruction and two data values. The cache is selective—only the instructions whose fetches conflict with PM bus data accesses are cached. This allows full-speed execution of looped operations such as digital filter multiply-accumulates and FFT butterfly processing.

1 Introduction

- *Data Address Generators with Hardware Circular Buffers*
  The ADSP-210xx's two data address generators (DAGs) implement circular data buffers in hardware. Circular buffers let delay lines (and other data structures required in digital signal processing) be implemented efficiently; circular buffers are commonly used in digital filters and Fourier transforms. The ADSP-210xx's two DAGs contain sufficient registers for up to 32 circular buffers (16 primary register sets, 16 secondary). The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation. Circular buffers can start and end at any memory location.

- *Flexible Instruction Set*
  The ADSP-210xx's 48-bit instruction word accommodates a variety of parallel operations, for concise programming. For example, in a single instruction, the ADSP-210xx can conditionally execute a multiply, an add, a subtract and a branch.

- *Serial Scan & Emulation Features*
  The ADSP-210xx supports the IEEE-standard P1149 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. This serial port also gives access to the ADSP-210xx on-chip emulation features.

1.4.2 ADSP-21020 DSP

The ADSP-21020 is the first member of the ADSP-21000 family. It is a complete implementation of the family base architecture. Figure 1.1 shows the block diagram of the ADSP-21020 and Figure 1.2 shows a system diagram.

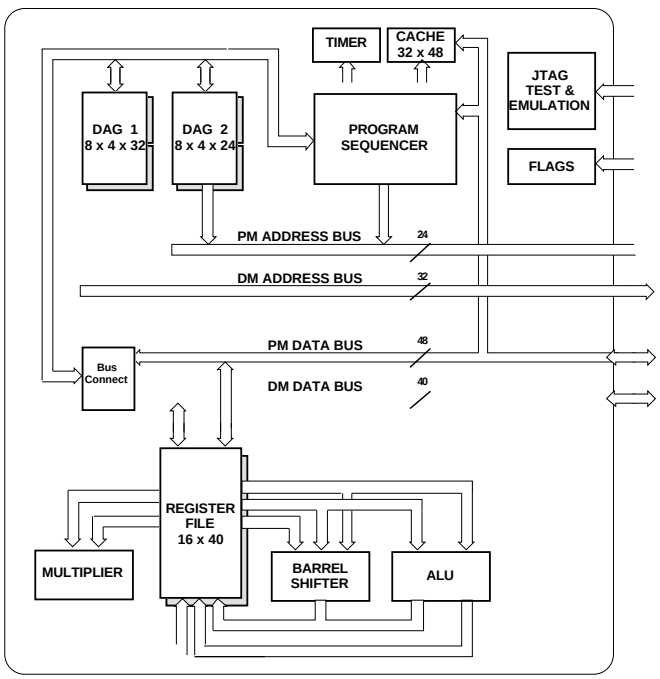


Figure 1.1 ADSP-21020 Block Diagram

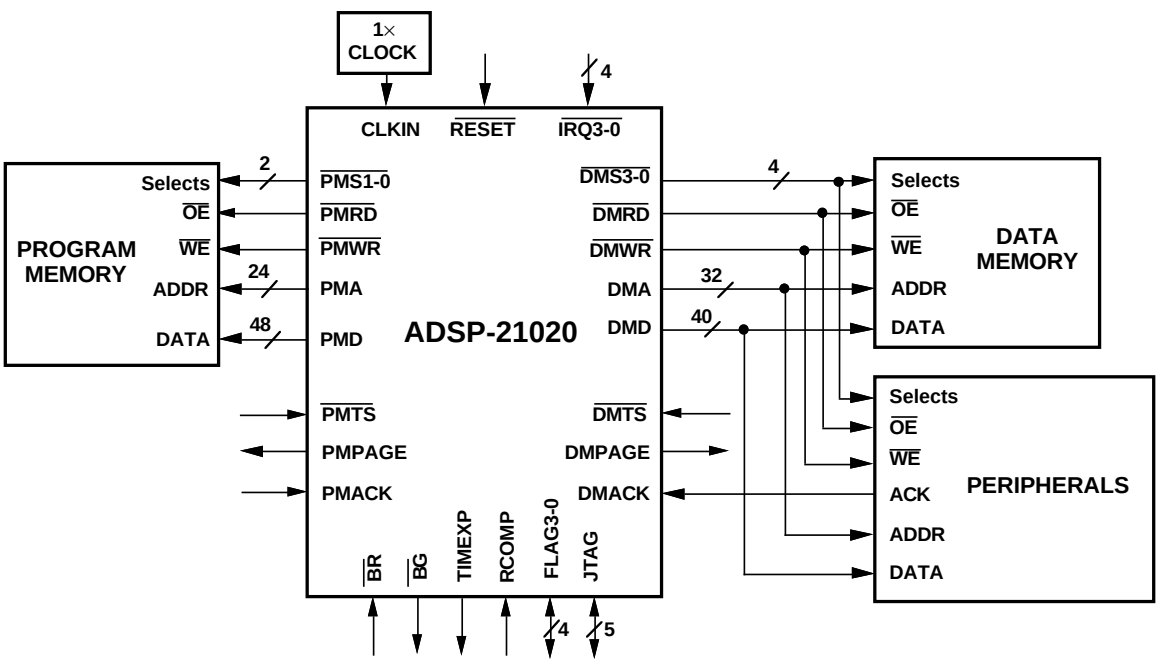


Figure 1.2 ADSP-21020 System Diagram

1.4.3 ADSP-21060 SHARC

The ADSP-21060 SHARC (Super Harvard Architecture Computer) is a single-chip 32-bit computer optimized for signal computing applications. The ADSP-21060 SHARC has the following key features:

Four Megabit Configurable On-Chip SRAM

- Dual-Ported for Independent Access by Base Processor and DMA
- Configurable as Maximum 128K Words Data Memory (32-Bit), 80K Words Program Memory (48-Bit), or Combinations of Both Up To 4 Mbits

Off-Chip Memory Interfacing

- 4 Gigawords Addressable (32-bit Address)
- Programmable Wait State Generation, Page-Mode DRAM Support

DMA Controller

2 Trigonometric, Mathematical & Transcendental Functions

This chapter contains listings and descriptions of several useful trigonometric, mathematical and transcendental functions. The functions are

Trigonometric
- sine/cosine approximation
- tangent approximation
- arctangent approximation

Mathematical
- square root
- square root with single precision
- inverse square root
- inverse square root with single precision
- division

Transcendental
- logarithm
- exponential
- power

2.1 SINE/COSINE APPROXIMATION

The sine and cosine functions are fundamental operations commonly used in digital signal processing algorithms , such as simple tone generation and calculation of sine tables for FFTs. This section describes how to calculate the sine and cosine functions.

This ADSP-210xx implementation of *sin(x)* is based on a min-max polynomial approximation algorithm in the [CODY]. Computation of the function *sin(x)* is reduced to the evaluation of a sine approximation over a small interval that is symmetrical about the axis.

2 Trigonometric, Mathematical & Transcendental Functions

Let

$$|x| = N\pi + f$$

where

$$|f| \leq \pi/2.$$

Then

$$sin(x) = sign(x) * sin(f) * (-1)^N$$

Once the sign of the input, x, is determined, the value of N can be determined. The next step is to calculate f. In order to maintain the maximum precision, f is calculated as follows

$$f = (|x| - xNC_1) - xNC_2$$

The constants C_1 and C_2 are determined such that C_1 is approximately equal to *pi* (π). C_2 is determined such that $C_1 + C_2$ represents pi to three or four decimal places beyond the precision of the ADSP-210xx.

For devices that represent floating-point numbers in 32 bits, Cody and Waite suggest a seven term min-max polynomial of the form $R(g) = g \cdot P(g)$. When expanded, the sine approximation for f is represented as

$$sin(f) = ((((((r_7 \cdot f + r_6) * f + r_5) * f + r_4) * f + r_3) * f + r_2) * f + r_1) \cdot f$$

With *sin(f)* calculated, *sin(x)* can be constructed. The cosine function is calculated similarly, using the trigonometric identity

$$cos(x) = sin(x + \pi/2)$$

2.1.1 Implementation

The two listings illustrate the sine approximation and the calling of the sine approximation. The first listing, `sin.asm`, is an implementation of the algorithm for calculation of sines and cosines. The second listing, `sinetest.asm`, is an example of a program that calls the sine approximation.

Trigonometric, Mathematical & Transcendental Functions 2

Implementation of the sine algorithm on ADSP-21000 family processors is straightforward. In the first listing below, `sin.asm`, two segments are defined. The first segment, defined with the `.SEGMENT` directive, contains the assembly code for the sine/cosine approximation. The second segment is a data segment that contains the constants necessary to perform this approximation.

The code is structured as a called subroutine, where the parameter *x* is passed into this routine using register F0. When the subroutine is finished executing, the value *sin(x)* or *cos(x)* is returned in the same register, F0. The variables, `i_reg` and `l_reg`, are specified as an index register and a length register, in either data address generator on the ADSP-21000 family. These registers are used in the program to point to elements of the data table, `sine_data`. Elements of this table are accessed indirectly within this program. Specifically, index registers I0 - I7 are used if the data table containing all the constants is put in data memory and index registers I8 - I15 are used if the data table is put in program memory. The variable `mem` must be defined as program memory, `PM`, or data memory, `DM`.

The include file, `asm_glob.h`, contains definitions of `mem`, `l_reg`, and `i_reg`. You can alter these definitions to suit your needs.

The second listing, `sinetest.asm`, is an example of a routine that calls the cosine and sine routines.

There are two entry points in the subroutine, `sin.asm`. They are labeled `cosine` and `sine`. Code execution begins at these labels. The calling program uses these labels by executing the instruction

```
call sine (db);
```

or

```
call cosine (db);
```

with the argument *x* in register F0. These calls are delayed branch calls that efficiently use the instruction pipeline on the ADSP-21000 family. In a delayed branch, the two instructions following the branch instruction are executed prior to the branch. This prevents the need to flush an instruction pipeline before taking a branch.

2 Trigonometric, Mathematical & Transcendental Functions

2.1.2 Code Listings

2.1.2.1 Sine/Cosine Approximation Subroutine

```
/************************************************************************

File Name
   SIN.ASM

Version
0.03  7/4/90

Purpose
   Subroutine to compute the Sine or Cosine values of a floating point input.

Equations Implemented
   Y=SIN(X) or
   Y=COS(X)

Calling Parameters
   F0 = Input Value X=[6E-20, 6E20]
   l_reg=0

Return Values
   F0 = Sine (or Cosine) of input Y=[-1,1]

Registers Affected
   F0, F2, F4, F7, F8, F12
   i_reg

Cycle Count
   38 Cycles

# PM Locations
   34 words

# DM Locations
   11 Words

************************************************************************/
```

Trigonometric, Mathematical & Transcendental Functions 2

```
#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;

#define half_PI  1.57079632679489661923

.GLOBAL   cosine, sine;

/**** Cosine/Sine approximation program starts here.  ****/
/**** Based on algorithm found in Cody and Waite.     ****/

cosine:
          i_reg=sine_data;                 /*Load pointer to data*/
          F8=ABS F0;                       /*Use absolute value of input*/
          F12=0.5;                         /*Used later after modulo*/
          F2=1.57079632679489661923;       /* and add PI/2*/

          JUMP compute_modulo (DB);        /*Follow sin code from here!*/
          F4=F8+F2, F2=mem(i_reg,1);
          F7=1.0;                                /*Sign flag is set to 1*/

sine:
          i_reg=sine_data;                 /*Load pointer to data*/
          F7=1.0;                                /*Assume a positive sign*/
          F12=0.0;                         /*Used later after modulo*/
          F8=ABS F0, F2=mem(i_reg,1);
          F0=PASS F0, F4=F8;
          IF LT F7=-F7;                    /*If input was negative, invert
sign*/

compute_modulo:
          F4=F4*F2;                        /*Compute fp modulo value*/
          R2=FIX F4;                       /*Round nearest fractional portion*/
          BTST R2 BY 0;                    /*Test for odd number*/
          IF NOT SZ F7=-F7;                /*Invert sign if odd modulo*/
          F4=FLOAT R2;                     /*Return to fp*/
          F4=F4-F12, F2=mem(i_reg,1);            /*Add cos adjust if necessary,
                                                   F4=XN*/


compute_f:
          F12=F2*F4, F2=mem(i_reg,1);            /*Compute XN*C1*/
          F2=F2*F4, F12=F8-F12;                  /*Compute |X|-XN*C1, and
XN*C2*/
          F8=F12-F2, F4=mem(i_reg,1);            /*Compute f=(|X|-XN*C1)-
XN*C2*/
          F12=ABS F8;                      /*Need magnitude for test*/
          F4=F12-F4, F12=F8;               /*Check for sin(x)=x*/
          IF LT JUMP compute_sign;         /*Return with result in F1*/

compute_R:
          F12=F12*F12, F4=mem(i_reg,1);
```

(listing continues on next page)

```
          LCNTR=6, DO compute_poly UNTIL LCE;
          F4=F12*F4, F2=mem(i_reg,1);             /*Compute sum*g*/

compute_poly:
          F4=F2+F4;                          /*Compute sum=sum+next r*/
          F4=F12*F4;                         /*Final multiply by g*/
          RTS (DB), F4=F4*F8;                /*Compute f*R*/
          F12=F4+F8;                         /*Compute Result=f+f*R*/

compute_sign:
          F0=F12*F7;                         /*Restore sign of result*/
          RTS;                               /*This return only for sin(eps)=eps
path*/

.ENDSEG;



.SEGMENT/SPACE Assembly_Library_Data_Space;
.PRECISION=MEMORY_PRECISION;


.VAR sine_data[11] =
                  0.31830988618379067154,   /*1/PI*/
                  3.14160156250000000000,   /*C1, almost PI*/
                 -8.90891020676153735661 7E-6, /*C2, PI=C1+C2*/
                  9.536743164E-7,           /*eps, sin(eps)=eps*/
                 -0.737066277507114174E-12, /*R7*/
                  0.160478446323816900E-9,  /*R6*/
                 -0.250518708834705760E-7,  /*R5*/
                  0.275573164212926457E-5,  /*R4*/
                 -0.198412698232225068E-3,  /*R3*/
                  0.833333333327592139E-2,  /*R2*/
                 -0.166666666666659653;     /*R1*/

.ENDSEG;
```

Trigonometric, Mathematical & Transcendental Functions 2

Listing 2.1 sin.asm

2.1.2.2 Example Calling Routine

```
/
*************************************************************************

File Name
    SINTEST.ASM

Purpose
    Example calling routine for the sine function.


*************************************************************************/

#include “asm_glob.h”;
#include “def21020.h”;
#define N 4
#define PIE 3.141592654

.SEGMENT/DM     dm_data;                  /* Declare variables in data memory
*/
.VAR input[N]= PIE/2, PIE/4, PIE*3/4, 12.12345678;          /* test data */
.VAR output[N]                                              /* results here */
.VAR correct[N]=1.0, .707106781, .707106781, -.0428573949;   /* correct results
*/
.ENDSEG;

.SEGMENT/PM
          pm_rsti;        /* The reset vector resides in this space */
          DMWAIT=0x21;    /* Set data memory waitstates to zero */
          PMWAIT=0x21;    /* Set program memory waitstates to zero */
          JUMP start;

.ENDSEG;


.EXTERN         sine;
.SEGMENT/PM     pm_code;

start:
          bit set mode2 0x10; nop; read cache 0; bit clr mode2 0x10;
          M1=1;
          B0=input;
          L0=0;
          I1=output;
          L1=0;

          lcntr=N, do calcit until lce;
                 CALL sine (db);
                 l_reg=0;
                 f0=dm(i0,m1);
calcit:
          dm(i1,m1)=f0;

end:
```

2 Trigonometric, Mathematical & Transcendental Functions

```
            IDLE;
.ENDSEG;
```

Listing 2.2 sintest.asm

2.2 TANGENT APPROXIMATION

The tangent function is one of the fundamental trigonometric signal processing operations. This section shows how to approximate the tangent function in software. The algorithm used is taken from [CODY].

Tan(x) is calculated in three steps:

1. The argument x (which may be any real value) is reduced to a related argument *f* with a magnitude less than $\pi/4$ (that is, *t* has a range of $\pm \pi/2$).

2. *Tan(f)* is computed using a min-max polynomial approximation.

3. The desired function is reconstructed.

2.2.1 Implementation

The implementation of the tangent approximation algorithm uses 38 instruction cycles and consists of three logical steps.

First, the argument *x* is reduced to the argument *f*. This argument reduction is done in the sections labeled `compute_modulo` and `compute_f`.

The factor $\pi/2$ is required in the computation to reduce *x* (which may be any floating-point value) to *f* (which is a normalized value with a range of $\pm \pi/2$). To get an accurate result, the constants C_1 and C_2 are chosen so that $C_1 + C_2$ approximates $\pi/2$ to three or four decimal places beyond machine precision. The value C_1 is chosen to be close to $\pi/2$ and C_2 is a factor that is added to C_1 that results in a very accurate representation of $\pi/2$.

Notice that in the argument reduction, the assembly instructions are all multifunction instructions. ADSP-21000 family processors can execute a data move or a register move in parallel with a computation. Because multifunction instructions execute in a single cycle, the overhead for the memory move is eliminated.

A special case is if $tan(x) = x$. This occurs when the absolute value of f is less than epsilon. This value is very close to *0*. In this case, a jump is executed and the tangent function is calculated using the values of *f* and *1* in the final divide.

The second step is to approximate the tangent function using a min-max polynomial expansion. Two calculations are performed, the calculation of *P(g)* and the calculation of *Q(g)*, where *g* is just *f * f*. Four coefficients are used in calculation of both *P(g)* and *Q(g)*. The section labeled `compute_P` makes this calculation:

$$P(g) = ((p3 * g + p2) * g + p1) * g * f$$

Where $g = f*f$ and *f* is the reduced argument of x. The value *f * P(g)* is stored in the register F8.

The section labeled `compute_Q`, makes this calculation:

$$Q(g) = ((q_3*g + q_2)*g + q_1) * g + q_0$$

The third step in the calculation of the tangent function is to divide *P(g)* by *Q(g)*. If the argument, *f*, is even, then the tangent function is

$$tan(f) = f * P(g)/Q(g)$$

If the argument, *f*, is odd then the tangent function is

$$tan(f) = -f * P(g)/Q(g)$$

Finally, the value *N* is multiplied in and the reconstructed function *tan(x)* is returned in the register F0.

Similarly, the cotangent function is easily calculated by inverting the polynomial

$cotan(f) = Q(g)/–f * P(g)$

2.2.2 Code Listing–Tangent Subroutine

```
/
*****************************************************************************

File Name
    TAN.ASM

Version
    Version 0.03  7/5/90

Purpose
    Subroutine to compute the tangent of a floating point input.


Equations Implemented
    Y=TAN(X)

Calling Parameters
    F0 = Input Value X=[6E-20, 6E20]
    l_reg = 0 (usually L3)

Return Values
    F0 = tangent of input X

Registers Affected
    F0, F1, F2, F4, F7, F8, F12
    i_reg (usually I3)

Cycle Count
    38 Cycles
```

```
# PM Locations

# DM Locations

*****************************************************************************/


#include “asm_glob.h”
.SEGMENT/PM      Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   tan;

tan:      i_reg=tangent_data;
          F8=PASS F0, F2=mem(i_reg,1);            /* Use absolute value of input
*/

compute_modulo:
          F4=F8*F2, F1=F0;                 /* Compute fp modulo value */
          R2=FIX F4, F12=mem(i_reg,1);            /* Rnd nearest fractional
                                                     portion */
          F4=FLOAT R2, R0=R2;              /* Return to fp */

compute_f:
          F12=F12*F4, F2=mem(i_reg,1);            /* Compute XN*C1 */
          F2=F2*F4, F12=F8-F12;                   /* Compute X-XN*C1, and XN*C2
*/
          F8=F12-F2, F4=mem(i_reg,1);             /* Compute f=(X-XN*C1)-XN*C2
*/
          F12=ABS F8, F7=F8;
          F4=F12-F4, F12=mem(i_reg,1);            /* Check for TAN(x)=x */
          IF LT JUMP compute_quot;         /* Compute quotient with NUM=f
                                                  DEN=1 */

compute_P:
          F12=F8*F8, F4=mem(i_reg,1);             /* g=f*f */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute p3*g */
          F4=F2+F4;                        /* Compute (p3*g + p2) */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute (p3*g + p2)*g */
          F4=F2+F4;                        /* Compute (p3*g + p2)*g + p1 */
          F4=F12*F4;                       /* Compute
                                                  ((p3*g + p2)*g + p1)*g */
          F4=F4*F8;                        /* Compute
                                                  ((p3*g + p2)*g +p1)*g*f */
          F8=F4+F8, F4=mem(i_reg,1);       /* Compute f*P(g) */
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
compute_Q:
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F4=F2+F4;                          /* Compute sum=sum+next q */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F4=F2+F4;                          /* Compute sum=sum+next q */
          F4=F12*F4, F2=mem(i_reg,1);             /* Compute sum*g */
          F12=F2+F4, F7=F8;                  /* Compute sum=sum+next q */



compute_quot:
          BTST R0 BY 0;                      /* Test LSB */
          IF NOT SZ F12=-F7, F7=F12;         /* SZ true if even value*/
          F0=RECIPS F12;                          /* Get 4 bit seed R0=1/D */
          F12=F0*F12, F11=mem(i_reg,1);      /* D(prime) = D*R0 */
          F7=F0*F7, F0=F11-F12;                   /* F0=R1=2-D(prime), F7=N*R0
*/
          F12=F0*F12;                        /* F12=D(prime)=D(prime)*R1 */
          F7=F0*F7, F0=F11-F12;                   /* F7=N*R0*R1, F0=R2=2-
D(prime) */
          RTS (DB), F12=F0*F12;                   /* F12=D(prime)=D(prime)*R2 */
          F7=F0*F7, F0=F11-F12;                   /* F7=N*R0*R1*R2,
                                                  F0=R3=2-D(prime)*/
          F0=F0*F7;                          /* F7=N*R0*R1*R2*R3 */



.ENDSEG;



.SEGMENT/SPACE Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;
.VAR tangent_data[13] =  0.6366197723675834308,    /* 2/PI */
                  1.57080078125,                   /* C1, almost PI/2 */
                 -4.4544551033807686783O8E-6,            /* C2, PI/2=C1+C2 */
                  9.536743164E-7,                  /* eps, TAN(eps)=eps */
                  1.0,                             /* Used in one path */
                 -0.7483634966612065149E-5,        /* P3 */
                  0.2805918241169988906E-2,        /* P2 */
                 -0.1282834704095743847,           /* P1 */
```

```
                -0.2084480442203870948E-3,      /* Q3 */
                 0.2334485282206872802E-1,      /* Q2 */
                -0.4616168037429048840,         /* Q1 */
                 1.0,                           /* Q0 */
                 2.0;                           /* Used in divide */

.ENDSEG;
```

Listing 2.3 tan.asm

2.3 ARCTANGENT APPROXIMATION

The arctangent function is one of the fundamental trigonometric signal processing operations. Arctangent is often used in the calculation of the phase of a signal and in the conversion between Cartesian and polar data representations.

This section shows how to approximate the arctangent function in software. The algorithm used is taken from [CODY].

Calculation of *atan(x)* is done in three steps:

1. The argument *x* (which may be any real value) is reduced to a related argument f with a magnitude less than or equal to *2– R(3)*.

2. *Atan(f)* is computed using a rational expression.

3. The desired function is reconstructed.

2.3.1 Implementation

This implementation of the tangent approximation algorithm uses 82 instruction cycles, in the worst case. It follows the three logical steps listed above.

The assembly code module can be called to compute either of two functions, `atan` or `atan2`. The `atan` function returns the arctangent of an argument x. The `atan2` function returns the arctangent of y/x. This form

$$H=atan(y/x)$$

is especially useful in phase calculations.

First, the argument x is reduced to the argument f. This argument reduction relies on the symmetry of the arctangent function by using the identity

$$arctan(x) = -arctan(-x)$$

The use of this identity guarantees that approximation uses a non-negative x. For values that are greater than one, a second identity is used in argument reduction

$$arctan(x) = \pi/2 - arctan\ (1/x)$$

The reduction of x to the argument F is complete with the identity

$$arctan(x) = \pi/6 + arctan(f)$$

where

$$f = (x - (R(3) - 1) / (R(3) + x)$$

Just like tangent approximation, the second step in the arctangent calculation is computing a rational expression of the form

$$R = g * P(g) / Q(g)$$

where

$$g * P(g) = (P1 * g + P0) * g$$

and

$$Q(g) = (g + q1) * g + Q0$$

Notice that an eight-cycle macro, `divide`, is implemented for division. This macro is used several times in the program.

The final step is to reconstruct the *atan(x)* from the *atan(f)* calculation.

2.3.2 Listing–Arctangent Subroutine

```
/
*******************************************************************************

File Name
   ATAN.ASM

Version
   Version 0.01   3/20/91

Purpose
   Subroutine to compute the arctangent values of a floating point input.


Equations Implemented
   atan-       H=ATAN(Y)
   atan2-      H=ATAN(Y/X)
  where H is in radians

Calling Parameters
   F0 = Input Value Y=[6E-20, 6E20]
   F1 = Input Value X=[6E-20, 6E20] (atan2 only)
   l_reg=0

Return Values
   F0 = ArcTangent of input
      =[-pi/2,pi/2]   for atan
      =[-pi,pi]       for atan2


Registers Affected
   F0, F1, F2, F4, F7, F8, F11, F12, F15
    i_reg
    ms_reg
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
Cycle Count
   atan    61 Cycles maximum
   atan2   82 cycles maximum


# PM Locations

# DM Locations

*****************************************************************************/

/* The divide Macro used in the arctan routine*/
/* ---------------------------------------------------------------------------
-------
DIVIDE - Divide Macro

Register Usage:
    q       = f0-f15        Quotient
    n       = f4-f7         Numerator
    d       = f12-f15       Denominator
    two     = f8-f11        must have 2.0 pre-stored
    tmp     = f0-f3

Indirectly affected registers:
    ASTAT,STKY

Looping:    none

Special Cases:
    q may be any register, all others must be distinct.

Cycles: 8

Created:    3/19/91
---------------------------------------------------------------------------
-----*/

#define DIVIDE(q,n,d,two,tmp)                                               \
        n=RECIPS d, tmp=n;          /* Get 8 bit seed R0=1/D*/              \
        d=n*d;                      /* D(prime) = D*R0*/                    \
        tmp=tmp*n, n=two-d;         /* N=2-D(prime), TMP=N*R0*/             \
        d=n*d;                      /* D=D(prime)=D(prime)*R1*/             \
        tmp=tmp*n, n=two-d;         /* TMP=N*R0*R1, N=R2=2-D(prime)*/       \
        d=n*d;                      /* D=D(prime)=D(prime)*R2*/             \
        tmp=tmp*n, n=two-d;         /* TMP=N*R0*R1*R2, N=R3=2-D(prime)*/    \
        q=tmp*n
```

Trigonometric, Mathematical & Transcendental Functions 2

```
#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;

.PRECISION=MACHINE_PRECISION;

.GLOBAL     atan, atan2;

atan2:          i_reg=atan_data;
                F11= 2.0;
                F2= 0.0;
                F1=PASS F1;
                IF EQ JUMP denom_zero;      /* if Denom. = 0, goto special
case*/
                IF LT F2=mem(11,i_reg);     /* if Denom. < 0, F2=pi (use at
end)*/

overflow_tst:   R4=LOGB F0, F7=F0;         /* Detect div overflow for atan2*/
                R1=LOGB F1, F15=F1;
                R1=R4-R1;                       /* Roughly exp. of quotient*/
                R4=124;                    /* Max exponent - 3*/
                COMP(R1,R4);
                IF GE JUMP overflow;       /* Over upper range? Goto overflow*/
                R4=-R4;
                COMP(R1,R4);
                IF LE JUMP underflow;           /* Over lower range? Goto
underflow*/

do_division:    DIVIDE(F0,F7,F15,F11,F1);
                JUMP re_entry (DB);

atan:           R10= 0;                    /* Flags multiple of pi to add at
end*/
                F15=ABS F0;

                i_reg=atan_data;                /* This init is redundant for
atan2f*/
                F11= 2.0;                       /* Needed for divide*/
                F2= 0.0;                        /* Result is not in Quad 2 or
3 */

re_entry:       F7 = 1.0;
                COMP(F15,F7), F4=mem(0,i_reg); /* F4=2-sqrt(3)*/
                IF LE JUMP tst_f;          /* If input<=1, do arctan(input)*/
                                           /* else do arctan(1/input)+const*/
                DIVIDE(F15,F7,F15,F11,F1); /* do x=1/x*/

                R10 = 2;                        /* signal to add const at
end*/
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
tst_f:          COMP(F15,F4);              /* Note F4 prev. loaded from memory*/
                IF LT JUMP tst_for_eps;
                R10=R10+1,  F4=mem(1,i_reg);      /* F4=sqrt(3)*/
                F12=F4*F15;
                F7=F12-F7;                 /* F7=F12-1.0*/
                F15=F4+F15;
                DIVIDE(F15,F7,F15,F11,F1); /*  = (sqrt(3)*x-1)/(x+sqrt(3))*/

tst_for_eps:    F7=ABS F15, F4=mem(2,i_reg);      /* F4=eps (i.e. small)*/
                COMP(F7,F4);
                IF LE JUMP tst_N;          /* if x<=eps, then h=x*/
                F1=F15*F15, F4=mem(3,i_reg);      /* else . . .*/
                F7=F1*F4,   F4=mem(4,i_reg);
                F7=F7+F4,   F4=mem(5,i_reg);
                F7=F7*F1;
                F12=F1+F4,  F4=mem(6,i_reg);
                F12=F12*F1;
                F12=F12+F4;
                DIVIDE(F7,F7,F12,F11,F1);  /* e=((p1*x^2 +p0)x^2)/(x^2
+q1)x^2+q0*/
                F7=F7*F15;
                F15=F7+F15;                /* h=e*x+x*/

tst_N:          R1=R10-1, R7=mem(i_reg,7);  /* if R10 > 1, h=-h; dummy read*/
                ms_reg=R10;
                IF GT F15=-F15;
                F4 = mem(ms_reg,i_reg);     /* index correct angle addend to h
*/
                F15=F15+F4;                /* h=h+ a*pi      */

tst_sign_y:     F2=PASS F2;                /* if (atan2f denom <0) h=pi-h else
h=h  */
                IF NE F15=F2-F15;
tst_sign_x:     RTS (DB), F0=PASS F0;      /* if (numer<0) h=-h else h=h*/
                IF LT F15=-F15;
                F0=PASS F15;               /* return with result in F0!*/


underflow:      JUMP tst_sign_y (DB);
                F15=0;
                NOP;


overflow:       JUMP tst_sign_x (DB);
                F15=mem(9,i_reg);          /* Load pi/2*/
                NOP;


denom_zero:     F0=PASS F0;                /* Careful: if Num !=0, then
overflow*/
                IF NE JUMP overflow;

error:          RTS;                       /* Error: Its a 0/0!*/
```

```
.ENDSEG;


.SEGMENT/SPACE  Assembly_Library_Data_Space;

.PRECISION=MEMORY_PRECISION;

.VAR atan_data[11] =    0.26794919243112270647,     /* 2-sqrt(3)      */
                        1.73205080756887729353,     /* sqrt(3)  */
                        0.000244140625,             /* eps            */
                        -0.720026848898E+0,         /* p1             */
                        -0.144008344874E+1,         /* p0             */
                        0.475222584599E+1,          /* q1             */
                        0.432025038919E+1,          /* q0             */
                        0.00000000000000000000,     /* 0*pi           */
                        0.52359877559829887308,     /* pi/6           */
                        1.57079632679489661923,     /* pi/2           */
                        1.04719755119659774615,     /* pi/3           */
                        3.14159265358979323846;     /* pi             */
.ENDSEG;
```

Listing 2.4 atan.asm

2.4 SQUARE ROOT & INVERSE SQUARE ROOT APPROXIMATIONS

An ADSP-21000 family DSP can perform the square root function, *sqrt(y)*, and the inverse square root, *isqrt(y)*, quickly and with a high degree of precision. These functions are typically used to calculate magnitude functions of FFT outputs, to implement imaging and graphics algorithms, and to use the DSP as a fast math coprocessor.

A square root exists (and can be calculated) for every non-negative floating-point number. Calculating the square root of a negative number gives an imaginary result. To calculate the square root of a negative number, take the absolute value of the number and use *sqrt(y)* or *isqrt(y)* as defined in this section. Remember that the result is really an imaginary number.

The ADSP-21000 family program that calculates *isqrt(y)* is based on the Newton-Raphson iteration algorithm in [CAVANAGH]. Computation of the function begins with a low-accuracy initial approximation. The Newton-

Raphson iteration is then used for successively more accurate approximations.

Once *isqrt(y)* is calculated it only takes one additional operation to calculate *sqrt(y)*. Given the input, *y*, the square root, *x=R(*y), is determined as follows

$$x = sqrt(y) = y * 1/sqrt(y)$$

Given an initial approximation x_n for the inverse square root of y, isqrt(y), the Newton-Raphson iteration is used to calculate a more accurate approximation, x_{n+1}.

Newton-Raphson iteration: $x_{n+1} = (0.5)\ (x_n)\ (3 - (x_n^2)\ )\ (y))$

The number of iterations determines the accuracy of the approximation. For a 32-bit floating point representation with a 24-bit mantissa, two iterations are sufficient to achieve an accuracy to ±1 LSB of precision. For an extended 40-bit precision representation that has a 32-bit mantissa, three iterations are needed.

For example, suppose that you need to calculate the square root of 30. If you use 5.0 as an initial approximation of the square root, the inverse square root approximation, 1/ 30, is 0.2. Therefore, given $y=30$ and $x_n=0.2$, one iteration yields

$$x_{n+1} = (0.5)\ (0.2)\ (3 - (0.2^2)\ )\ (30))$$
$$x_{n+1} = 0.18$$

A second iteration using 0.18 as an input yields

$$x_{n+2} = (0.5)\ (0.18)\ (3 - (0.18^2)\ )\ (30))$$
$$x_{n+2} = 0.18252$$

And finally a third iteration using 0.18252 as an input yields

$$x_{n+2} = (0.5)\ (0.18252)\ (3 - (0.18252^2)\ )\ (30))$$
$$x_{n+2} = 0.182574161$$

Therefore the approximation of the inverse square root of 30 after three iterations is 0.182574161. To calculate the square root, just multiply the two numbers together

*30 * 0.182574161 = 5.47722483*

The actual square root of 30 is 5.477225575. If the initial approximation is accurate to four bits, the final result is accurate to about 32 bits of precision.

2.4.1 Implementation

To implement the Newton-Raphson iteration method for calculating an inverse square root on an ADSP-21000 processor, the first task is to calculate the initial low-accuracy approximation.

The ADSP-21000 family instruction set includes the instruction `RSQRTS` that, given the floating point input Fx, creates a 4-bit accurate seed for $1/\sqrt{Fx}$. This seed, or low accuracy approximation, is determined by using the six MSBs of the mantissa and the LSB of the unbiased exponent of Fx to access a ROM-based look up table.

Note that the instruction `RSQRTS` only accepts inputs greater than zero. A *±Zero* returns *±Infinity*, *±Infinity* returns *±Zero*, and a *NAN* (not-a-number) or negative input returns an all 1's result. You can use conditional logic to assure that the input value is greater than zero.

To calculate the seed for an input value stored in register F0, use the following instruction:

```
F4=RSQRTS F0;        /*Fetch seed*/
```

Once you have the initial approximation, it is a easy to implement the Newton-Raphon iteration in ADSP-21000 assembly code. With the approximation now in F4 and with F1 and F8 initialized with the constants 0.5 and 3 respectively, one iteration of the Newton-Raphson is

implemented as follows:

```
F12=F4*F4;              /* F12=X0^2 */
F12=F12*F0;             /* F12=C*X0^2 */
F4=F2*F4, F12=F8-F12;   /* F4=.5*X0, F10=3-C*X0^2 */
F4=F4*F12;              /* F4=X1=.5*X0(3-C*X0^2) */
```

The register F4 contains a reasonably accurate approximation for the inverse square root. Successive iterations are made by repeating the above four lines or code. The square root of F0 is calculated by multiplying the approximation F4, by the initial input F0:

```
F0=F4*F0;               /* X=sqrt(Y)=Y/sqrt(Y) */
```

2.4.2 Code Listings

There are four subroutine listings below that illustrate how to calculate *sqrt(y)* and i*sqrt(y)*. Two are for single precision (24-bit mantissa), and two for extended precision (32-bit mantissa).

2.4.2.1 SQRT Approximation Subroutine

```
/
*******************************************************************************
File Name
    SQRT.ASM
```

```
Version
    Version 0.02  7/6/90

Purpose
    Subroutine to compute the square root of x using the 1/sqrt(x) approximation.

Equations Implemented
    X = sqrt(Y) = Y/sqrt(Y)

Calling Parameters
    F0 = Y Input Value
    F8 = 3.0
    F1 = 0.5

Return Values
    F0 = sqrt(Y)

Registers Affected
    F0, F4, F12

Cycle Count
    14 Cycles

# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”
```

2 Trigonometric, Mathematical & Transcendental Functions

```
.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL sqrt;

sqrt:
   F4=RSQRTS F0;                     /*Fetch seed*/

   F12=F4*F4;                        /*F12=X0^2*/
   F12=F12*F0;                       /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                    /*F4=.5*X0, F10=3-C*X0^2*/
   F4=F4*F12;                        /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                        /*F12=X1^2*/
   F12=F12*F0;                       /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                    /*F4=.5*X1, F10=3-C*X1^2*/
   F4=F4*F12;                        /*F4=X2=.5*X1(3-C*X1^2)*/

   F12=F4*F4;                        /*F12=X2^2*/
   F12=F12*F0;                       /*F12=C*X2^2*/
   RTS (DB), F4=F1*F4, F12=F8-F12;   /*F4=.5*X2, F10=3-C*X2^2*/
   F4=F4*F12;                        /*F4=X3=.5*X2(3-C*X2^2)*/

   F0=F4*F0;                         /*X=sqrt(Y)=Y/sqrt(Y)*/
.ENDSEG;
```

Listing 2.5 sqrt.asm

2.4.2.2 ISQRT Approximation Subroutine

```
/
*****************************************************************************
File Name
   ISQRT.ASM
```

```
Version
   Version 0.02  7/6/90

Purpose
   Subroutine to compute the inverse square root of x using the 1/
sqrt(x)                        approximation.

Equations Implemented
   X = isqrt(Y) = 1/sqrt(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = isqrt(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   13 Cycles

#PM Locations

#DM Locations

*******************************************************************************/
```

```
#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL isqrt;

isqrt:
   F4=RSQRTS F0;                    /*Fetch seed*/

   F12=F4*F4;                       /*F12=X0^2*/
   F12=F12*F0;                      /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X0, F10=3-C*X0^2*/
   F4=F4*F12;                       /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                       /*F12=X1^2*/
   F12=F12*F0;                      /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X1, F10=3-C*X1^2*/
   F4=F4*F12;                       /*F4=X2=.5*X1(3-C*X1^2)*/

   F12=F4*F4;                       /*F12=X2^2*/
   F12=F12*F0;                      /*F12=C*X2^2*/
   RTS (DB), F4=F1*F4, F12=F8-F12;  /*F4=.5*X2, F10=3-C*X2^2*/
   F4=F4*F12;                       /*F4=X3=.5*X2(3-C*X2^2)*/
                                    /* =isqrt(Y)=1/sqrt(Y)*/

.ENDSEG;
```

Listing 2.6 isqrt.asm

2.4.2.3 SQRTSGL Approximation Subroutine

```
/
*****************************************************************************
File Name
   SQRTSGL.ASM
```

```
Version
   Version 0.02  7/6/90

Purpose
   Subroutine to compute the square root of x to single-precision (24 bits
mantissa) using the 1/sqrt(x) approximation.

Equations Implemented
   X = sqrtsgl(Y) = Y/sqrtsgl(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = sqrtsgl(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   10 Cycles
```

```
# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL sqrtsgl;

sqrtsgl:
   F4=RSQRTS F0;              /*Fetch seed*/

   F12=F4*F4;                 /*F12=X0^2*/
   F12=F12*F0;                /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;             /*F4=.5*X0, F12=3-C*X0^2*/
   F4=F4*F12;                 /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                 /*F12=X1^2*/
   F12=F12*F0;                /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;             /*F4=.5*X1, F12=3-C*X1^2*/
   F4=F4*F12;                 /*F4=X2=.5*X1(3-C*X1^2)*/
   F0=F4*F0;                  /*X=sqrtsgl(Y)=Y/sqrtsgl(Y)*/
.ENDSEG;
```

Listing 2.7 sqrtsgl.asm

2.4.2.4 ISQRTSGL Approximation Subroutine

```
/
*****************************************************************************
File Name
   ISQRTSGL.ASM
```

```
Version
   Version 0.02        7/6/90

Purpose
   Subroutine to compute the inverse square root of x to single-precision (24
bits                   mantissa) using the 1/sqrt(x) approximation.

Equations Implemented
   X = isqrtsgl(Y) = 1/sqrtsgl(Y)

Calling Parameters
   F0 = Y Input Value
   F8 = 3.0
   F1 = 0.5

Return Values
   F0 = isqrtsgl(Y)

Registers Affected
   F0, F4, F12

Cycle Count
   9 Cycles
```

2 Trigonometric, Mathematical & Transcendental Functions

```
# PM Locations

# DM Locations

*****************************************************************************/

#include “asm_glob.h”

.SEGMENT/PM            pm_code;
.PRECISION=MACHINE_PRECISION;
.GLOBAL isqrtsgl;

isqrtsgl:
   F4=RSQRTS F0;                    /*Fetch seed*/

   F12=F4*F4;                       /*F12=X0^2*/
   F12=F12*F0;                      /*F12=C*X0^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X0, F12=3-C*X0^2*/
   F4=F4*F12;                       /*F4=X1=.5*X0(3-C*X0^2)*/

   F12=F4*F4;                       /*F12=X1^2*/
   RTS(DB), F12=F12*F0;                   /*F12=C*X1^2*/
   F4=F1*F4, F12=F8-F12;                  /*F4=.5*X1, F12=3-C*X1^2*/
   F0=F4*F12;                       /*F4=X2=.5*X1(3-C*X1^2)*/
                                    /* =isqrtsgl(Y)=1/sqrtsgl(Y)*/
.ENDSEG;
```

Listing 2.8 isqrtsgl.asm

2.5 DIVISION

The ADSP-21000 family instruction set includes the `RECIPS` instruction to simplify the implementation of floating-point division.

2.5.1 Implementation

The code performs floating-point division using an in iterative convergence algorithm. The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: F0 = numerator, F12= denominator, F11 = 2.0. The quotients is returned in F0. (In the code listing, the two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

The algorithm is supplied with a startup seed which is a low-precision reciprocal of the denominator. This seed is generated by the `RECIPS` instruction. `RECIPS` creates an 8-bit accurate seed for *1/Fx*, the reciprocal of *Fx*. The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the *Fx* mantissa as an index. The unbiased exponent of the seed is calculated as the twos complement of the

unbiased *Fx* exponent, decremented by one; that is, if *e* is the unbiased exponent of *Fx*, then the unbiased exponent of $Fn = -e - 1$. The sign of the seed is the sign of the input. *±Zero* returns *±Infinity* and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is *±Zero*. A *NAN* input returns an all 1's result.

2.5.2 Code Listing–Division Subroutine

```
/*
File Name
   F.ASM

Version
   Version 0.03

Purpose
   An implementation of division using an Iterative Convergent Divide
Algorithm.

Equations Implemented
    Q = N/D

Calling Parameters
  F0  = N Input Value
    F12 = D Input Value
    F11 = 2.0

Return Values
    F0 = Quotient of input

Registers Affected
    F0, F7, F12

Cycle Count
    8 Cycles  (6 Cycles for single precision)

# PM Locations

# DM Locations
```

```
*/


#include “asm_glob.h”

.SEGMENT/PM     Assembly_Library_Code_Space;

.PRECISION=MACHINE_PRECISION;

.GLOBAL divide;

divide:         F0=RECIPS F12, F7=F0;   /*Get 4 bit seed R0=1/D*/
                F12=F0*F12;             /*D(prime) = D*R0*/
                F7=F0*F7, F0=F11-F12;   /*F0=R1=2-D(prime), F7=N*R0*/
                F12=F0*F12;             /*F12=D(prime)=D(prime)*R1*/
                F7=F0*F7, F0=F11-F12;   /*F7=N*R0*R1, F0=R2=2-
D(prime)*/

/* Remove next two instructions for 1 LSB accurate single-precision
result */
        RTS (DB), F12=F0*F12; {F12=D(prime)=D(prime)*R2}
        F7=F0*F7, F0=F11-F12; {F7=N*R0*R1*R2, F0=R3=2-D(prime)}


                F0=F0*F7;       {F0=N*R0*R1*R2*R3}

.ENDSEG;
```

Listing 2.9 Divide..asm

2.6 LOGARITHM APPROXIMATIONS

Logarithms (in base *e, 2,* and *10*) can be approximated for any non-negative floating point number. The *Software Manual for the Elementary Functions* by William Cody and William Waite explains how the computation of a logarithm involves three distinct steps:

1. The given argument (or input) is reduced to a related argument in a small, logarithmically symmetric interval about *1*.

2. The logarithm is computed for this reduced argument.

3. The desired logarithm is reconstructed from its components.

The algorithm can calculate logarithms of any base (base *e*, *2*, or *10*); the

code is identical until step three, so only one assembly-language module is needed.

The first step is to take a given floating point input, Y, and reduce it to $Y=f * 2^N$ where $0.5 \leq f < 1$. Given that $X = log(Y)$, X also equals

$$X=log_e(Y)$$
$$X=log_e(f * 2^N)$$
$$X=log_e(f) + N*log_e(2)$$

$N * log_e(2)$ is the floating point exponent multiplied by $log_e(2)$, which is a constant. The term $log_e(f)$ must be calculated. The definition of the variable s is

$$s = (f - 1) / (f + 1)$$

Then

$$log_e(f) = log_e((1 + s) / (1 - s))$$

This equation is evaluated using a min-max rational approximation detailed in [CODY]. The approximation is expressed in terms of the auxiliary variable $z = 2s$.

Once the value of $log_e(f)$ is approximated, the equation

$$X = log_e(f) + N*log_e(2)$$

yields the solution for the natural (base-e) log of the input Y. To compute the base-*2* or base-*10* logarithm, the result X is multiplied by a constant equal to the reciprocal of $log_e(2)$, or $log_e(10)$, respectively.

2.6.1 Implementation

`LOGS.ASM` is an assembly-language implementation of the logarithm algorithm. This module has three entry points; a different base of logarithm is computed depending on which entry point is used. The label `LOG` is used for calling the algorithm to approximate the natural (base-*e*) logarithm, while the labels `LOG2` and `LOG10` are used for base-*2* and base-*10* approximations, respectively. When assembling the file `LOGS.ASM` you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-D`*`identifier`*

switch at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA logs
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA logs
```

The first step to compute any of the desired logarithms is to reduce the floating point input Y to the form

$$Y=f * 2^N$$

The ADSP-21000 family supports the IEEE Standard 754/854 floating point format. This format has a biased exponent and a significant with a "hidden" bit of 1. The hidden bit, although not explicitly represented, is implicitly presumed to exist and it offsets the exponent by one bit place.

For example, consider the floating point number 12.0. Using the IEEE standard, this number is represented as $0.5 * 2^{131}$. By adding the hidden one and unbiasing the exponent, *12.0* is actually represented as $1.5 * 2^4$. To get to the format $f * 2^N$ where $0.5 \leq f < 1$, you must scale $1.5 * 2^4$ by two to get the format $0.75 * 2^3$. The instruction `LOGB` extracts the exponent from our floating-point input. The exponent is then decremented, and the mantissa is scaled to achieve the desired format.

Use the value of f to approximate the value of the auxiliary variable z. The variable z is approximated using the following formula

$$z=znum/zden$$

where

$$znum = (f - 0.5) - 0.5$$

and

$$zden = (f * 0.5) - 0.5 \qquad for\ f>1/sqrt(2)$$

or

$znum = f - 0.5$ *and*
$zden = znum * 0.5 + 0.5$ *for* $f <= 1/sqrt(2)$

Once z is found, it is used to calculate the min-max rational approximation $R(z)$, which has the form

$$R(z) = z + z * r(z^2)$$

The rational approximation $r(z^2)$ has been derived and for $w=z^2$ is

$$r(z^2) = w * A(w)/B(w)$$

where A(w) and B(w) are polynomials in w, with derived coefficients a0, a1, and b0

$$A(w) = a1 * w + a0$$
$$B(w) = w + b0$$

$R(z)$ is the approximation of $log_e(f)$ and the final step in the approximation of $log_e(Y)$ is to add in $N*log_e(2)$. The coefficients *C0* and *C1* and the exponent N are used to determine this value.

If only the natural logarithm is desired, then the algorithm is complete and the natural *log (ln(Y))* is returned in the register F0. If $log_2(Y)$ was needed, then F0 is multiplied by *1/ln(2)* or *1.442695041*. If $log_{10}(Y)$ is needed, then F0 is multiplied by *1/ln(10)* or *0.43429448190325182765*.

2.6.2 Code Listing

The listing for module `LOGS.ASM` is below. The calling routine uses the appropriate label for the type of logarithm that is desired: `LOG` for natural log (base-*e*), `LOG2` for base-*2* and `LOG10` for base-*10*.

At assembly time, you must specify the memory space where the eight coefficients are stored (Program or Data Memory) by using either the `-DDM_DATA` or `-DPM_DATA` switch. Attempts to assemble `LOGS.ASM` without one of these switches result in an error.

2.6.2.1 Logarithm Approximation Subroutine

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
/*****************************************************************
File Name
    LOGS.ASM

Version
    Version 0.03  8/6/90
    revised 26-APR-91

Purpose
    Subroutine to compute the logarithm (bases 2,e, and 10) of its floating
    point input.

Equations Implemented
    Y=LOG(X) or
    Y=LOG2(X) or
    Y=LOG10(X)

Calling Parameters
    F0 = Input Value
    l_reg=0;

Return Values
    F0 = Logarithm of input

Registers Affected
    F0, F1, F6, F7, F8, F10, F11, F12
    i_reg

Computation Time
    49 Cycles

#PM locations

#DM locations

*****************************************************************/

#include “asm_glob.h”

.SEGMENT/PM      Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   log, log10, log2;

log2:
    CALL logs_core (DB);                    /*Enter same routine in two cycles*/
    R11=LOGB F0, F1=F0;              /*Extract the exponent*/
    F12=ABS F1;                      /*Get absolute value*/
    RTS (DB);
    F11=1.442695041;                 /*1/Log(2)*/
```

```
   F0=F11*F0;                          /*F0 = log2(X)*/

log10:
   CALL logs_core (DB);                      /*Enter same routine in two cycles*/
   R11=LOGB F0, F1=F0;                 /*Extract the exponent*/
   F12=ABS F1;                         /*Get absolute value*/
   RTS (DB);
   F11=0.43429448190325182765;               /*1/Log(10)*/
   F0=F11*F0;                          /*F12 = log10(X)*/

log:
   R11=LOGB F0, F1=F0;
   F12=ABS F1;

logs_core:      i_reg=logs_data;           /*Point to data array*/
   R11=R11+1;                          /*Increment exponent*/
   R7=-R11, F10=mem(i_reg,1);          /*Negate exponent*/
   F12=SCALB F12 BY R7;                      /*F12= .5<=f<1*/
   COMP(F12,F10), F10=mem(i_reg,1); /*Compare f > C0*/
   IF GT JUMP adjust_z (DB);
   F7=F12-F10;                         /*znum = f-.5*/
   F8=F7*F10;                          /*znum * .5*/
   JUMP compute_r (DB);
   F12=F8+F10;                         /*zden = znum * .5 + .5*/
   R11=R11-1;                          /*N = N - 1*/

adjust_z:
   F7=F7-F10;                          /*znum = f - .5 - .5*/
   F8=F12*F10;                         /*f * .5*/
   F12=F8+F10;                         /*zden = f * .5 + .5*/

compute_r:
   F0=RECIPS F12;                            /*Get 4 bit seed R0=1/D*/
   F12=F0*F12, F10=mem(i_reg,1);       /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                     /*F0=R1=2-D(prime), F7=N*R0*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1, F0=R2=2-D(prime)*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1*R2, F0=R3=2-D(prime)*/
   F6=F0*F7;                           /*F7=N*R0*R1*R2*R3*/
   F0=F6*F6, F8=mem(i_reg,1);          /*w = z^2*/
   F12=F8+F0, F8=mem(i_reg,1);               /*B(W) = w + b0*/
   F7=F8*F0, F8=mem(i_reg,1);          /* w*a1*/
   F7=F7+F8, F8=F0;                    /*A(W) = w * a1 + a0*/
   F0=RECIPS F12;                            /*Get 4 bit seed R0=1/D*/
   F12=F0*F12;                         /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                     /*F0=R1=2-D(prime), F7=N*R0*/
   F12=F0*F12;                         /*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                     /*F7=N*R0*R1, F0=R2=2-D(prime)*/
```

```
   F12=F0*F12;                          /*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                      /*F7=N*R0*R1*R2, F0=R3=2-
D(prime)*/
   F7=F0*F7;                            /*F7=N*R0*R1*R2*R3*/
   F7=F7*F8;                            /*Compute r(z^2)=w*A(w)/B(w)*/

compute_R:
   F7=F6*F7;                            /* z*r(z^2)*/
   F12=F6+F7;                           /*R(z) = z + z * r(z^2)*/
   F0=FLOAT R11, F7=mem(i_reg,1);       /*F0=XN, F7=C2*/
   F10=F0*F7, F7=mem(i_reg,1);                /*F10=XN*C2, F7=C1*/
   RTS (DB);
   F7=F0*F7, F0=F10+F12;                      /*F0=XN*C2+R(z),
F7=XN*C1*/
   F0=F0+F7;                            /*F0 = ln(X)*/
.ENDSEG;

.SEGMENT/SPACE Assembly_Library_Data_Space;
.VAR logs_data[8] =    0.70710678118654752440,    /*C0 = sqrt(.5)*/
   0.5,                                           /*Constant used*/
   2.0,                                           /*Constant used*/
   -5.578873750242,                               /*b0*/
   0.1360095468621E-1,                            /*a1*/
   -0.4649062303464,                              /*a0*/
   -2.12194440054690582 7679E-4,                         /*C2*/
   0.693359375;                                   /*C1*/
.ENDSEG;
```

Listing 2.10 logs.asm

2.7 EXPONENTIAL APPROXIMATION

The exponential function (e^X or $exp(X)$) of a floating point number is computed by using an approximation technique detailed in the [CODY]. Cody and Waite explain how the computation of an exponent involves three distinct steps.

The first step is to reduce a given argument (or input) to a related argument in a small interval symmetric about the origin. If X is the input value for which you wish to compute the exponent, let

$$X = N*ln(C) + g \quad |g| \leq ln(C)/2.$$

Then

$$exp(X) = exp(g) * C^N$$

where $C = 2$, and N is calculated as $X/ln(C)$. Since the accuracy of the approximation critically depends on the accuracy of g, the effective precision of the ADSP-210xx processor must be extended during the calculation of g. Use the equation

$$g = (|X| - X_N * C_1) - X_N * C_2$$

where $C_1 + C_2$ represent $ln(C)$ to more than working precision, and X_N is the floating point representation of N. The values of C_1 and C_2 are precomputed and stored in program or data memory.

The second step is to compute the exponential for the reduced argument. Since you have now calculated N and g, you must approximate $exp(g)$. Cody and Waite have derived coefficients ($p1$, $q1$, etc.) especially for the approximation of $exp(g)/2$. The divide by two is added to counteract wobbly precision. The approximation of $exp(g)/2$ is

$$R(g) = 0.5 + (g*P(z))/Q(z) - g(P(z))$$

where

$$g*P(z) = ((p1 * z) + p0) * g,$$
$$Q(z) = (q1 * z) + q0,$$
$$z = g^2$$

The third step is to reconstruct the desired function from its components. The components of $exp(X)$ are $exp(g)=R(g)$, $C=2$, and $N=X/ln(2)$. Therefore:

$$exp(X) = exp(g) * C^N$$
$$= R(g) * 2^{(N+1)}$$

Note that N was incremented by one due to the scaling that occurred in the approximation of $exp(g)$.

2.7.1 Implementation

`EXP.ASM` is an implementation of the exponent approximation for the ADSP-21000 family. When assembling the file `EXP.ASM` you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-D`*`identifier`* `switch` at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA exp
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA exp
```

Before the first step in the approximation of the exponential function is performed, the floating-point input value is checked to assure that it is in the allowable range. The floating-point input is limited by the machine precision of the processor calculating the function. For the ADSP-21000 family, the largest number that can be represented is $XMAX = 2^{127} – 1LSB$. Therefore, the largest input to the function *exp(X)* is *ln(XMAX)*, which is approximately 88. The smallest positive floating point number that can be represented is $XMIN=2^{–127}$. Computing *ln(XMIN)* gives approximately –88 as the largest negative input to the function *exp(x)*. Therefore, comparisons to *ln(XMIN)* and *ln(XMAX)* are the first instructions of the `EXP.ASM` module. If these values are exceeded, the subroutine ends and returns either an error *(X > ln(XMAX))* or a very small number *(X < ln(XMIN))*.

The code includes another comparison for the case where the input produces the output *1*. This only occurs when the input is equal to *0.000000001* or 1.*0E-9.* If the input equals this value, the subroutine ends and returns a one.

The first step in the approximation is to compute *g* and *N* for the equation

$$exp(X) = exp(g) * 2^N$$

Where

$$N=X/ln(2)$$
$$g = (\lfloor X \rfloor - X_N * C_1) – X_N * C_2$$
$$C1 = 0.693359375$$
$$C2 = –2.12194440054690582 77E-4$$

Since multiplication requires fewer instructions than division, the constant *1/ln(2)* is stored in memory along with the precomputed values of C_1 and C_2. X_N is a floating point representation of N that can be easily computed using the `FIX` and `FLOAT` conversion features of the ADSP-210xx processors.

Given g and X_N, it is simple to compute the approximation for *exp(g)/2*

using the constants and equations outlined in the [CODY].

$$R(g) = 0.5 + (g * P(z)) / Q(z) - g(P(z))$$

where

$g*P(z) = ((p1 * z) + p0) * g,$
$Q(z) = (q1 * z) + q0$
$z = g^2$
$P1 = 0.59504254977591E-2$
$P0 = 0.24999999999992$
$Q2 = 0.29729363682238E-3$
$Q1 = 0.53567517645222E-1$
$Q0 = 0.5$

Once $R(g)$, the approximation for $exp(g)$, is calculated, the approximation for $exp(x)$ is derived by using the following equation:

$$\begin{aligned} exp(X) &= 2(approx(exp(g)/2)) * 2^N \\ &= approx(exp(g)/2) * 2^{(N+1)} \\ &= R(g) * 2^{(N+1)} \end{aligned}$$

The `SCALB` instruction scales the floating point value of $R(g)$ by the exponent $N + 1$.

2.7.2 Code Listings–Exponential Subroutine

```
/*************************************************************
File Name
    EXP.ASM
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
Version
   Version 0.03    8/6/90
   Modified        9/27/93

Purpose
   Subroutine to compute the exponential of its floating point input

Equations Implemented
   Y=EXP(X)

Calling Parameters
   F0 = Input Value
   l_reg=0;

Return Values
   F0 = Exponential of input

Registers Affected
   F0, F1, F4, F7, F8, F10, F12
   i_reg

Computation Time
   38 Cycles

# PM locations
   46 words

#DM locations
   12 words (could be placed in PM instead)
************************************************************/

#include “asm_glob.h”
.SEGMENT/PM            Assembly_Library_Code_Space;
.PRECISION=MACHINE_PRECISION;
.GLOBAL   exponential;
output_too_large: RTS (DB);
   F0=10000;    /*Set some error here*/
   F0=10000;
output_too_small: RTS (DB);
   F0 = .000001;
   F0 = .000001;
output_one:     RTS (DB);
   F0 = 1.0;
   F0 = 1.0;
exponential:    i_reg=exponential_data;   /*Skip one cycle after this*/
   F1=PASS F0;                            /*Copy into F1*/
   F12=ABS F1, F10=mem(i_reg,1);          /*Fetch maximum input*/
   COMP(F1,F10), F10=mem(i_reg,1);        /*Error if greater than max*/
   IF GT JUMP output_too_large;                 /*Return XMAX with error*/
   COMP(F1,F10), F10=mem(i_reg,1);        /*Test for input to small*/
   IF LT JUMP output_too_small;                 /*Return 0 with error*/
   COMP(F12,F10), F10=mem(i_reg,1);       /*Check for output 1*/
```

```
   IF LT JUMP output_one;                  /*Simply return 1*/
   F12=F1*F10, F8=F1;                      /*Compute N = X/ln(C)*/
   R4=FIX F12;                             /*Round to nearest*/
   F4=FLOAT R4, F0=mem(i_reg,1);           /*Back to floating point*/
compute_g:      F12=F0*F4, F0=mem(i_reg,1);          /*Compute
XN*C1*/
   F0=F0*F4, F12=F8-F12;                          /*Compute |X|-XN*C1,
and XN*C2*/
   F8=F12-F0, F0=mem(i_reg,1);                    /*Compute g=(|X|-
XN*C1)-XN*C2*/
compute_R:      F10=F8*F8;                      /*Compute z=g*g*/
   F7=F10*F0, F0=mem(i_reg,1);                 /*Compute p1*z*/
   F7=F7+F0, F0=mem(i_reg,1);              /*Compute p1*z + p0*/
   F7=F8*F7;                               /*Compute g*P(z) =
(p1*z+p0)*g*/
   F12=F0*F10, F0=mem(i_reg,1);                /*Compute q2*z*/
   F12=F0+F12, F8=mem(i_reg,1);                /*Compute q2*z +
q1*/
   F12=F10*F12;                            /*Compute (q2*z+q1)*z)*/
   F12=F8+F12;                             /*Compute
Q(z)=(q2*z+q1)*z+q0*/
   F12=F12-F7, F10=mem(i_reg,1);           /*Compute Q(z) - g*P(z)*/
   F0=RECIPS F12;                              /*Get 4 bit seed
R0=1/D*/
   F12=F0*F12;                             /*D(prime) = D*R0*/
   F7=F0*F7, F0=F10-F12;                       /*F0=R1=2-D(prime),
F7=N*R0*/
   F12=F0*F12;                             /
*F12=D(prime)=D(prime)*R1*/
   F7=F0*F7, F0=F10-F12;                       /*F7=N*R0*R1,
F0=R2=2-D(prime)*/
   F12=F0*F12;                             /
*F12=D(prime)=D(prime)*R2*/
   F7=F0*F7, F0=F10-F12;                       /*F7=N*R0*R1*R2,
F0=R3=2-D(prime)*/
   F7=F0*F7;                               /*F7=N*R0*R1*R2*R3*/
   F7=F7+F8;                               /*R(g) = .5 +
                                               (g*P(z))/(Q(z)-
g*P(z))*/
   R4=FIX F4;                              /*Get N in fixed point
again*/
   RTS (DB);
   R4=R4+1;
   F0=SCALB F7 BY R4;                      /*R(g) * 2^(N+1)*/
.ENDSEG;

.SEGMENT/SPACE  Assembly_Library_Data_Space;
.PRECISION=MEMORY_PRECISION;
.VAR exponential_data[12] =   88.0,        /*BIGX */
   -88.0,                                  /*SMALLX*/
   0.000000001,                            /*eps*/
   1.4426950408889634074,                  /*1/ln(2.0)*/
   0.693359375,                            /*C1*/
   -2.121944400546905827 7E-4,             /*C2*/
   0.59504254977591E-2,                        /*P1*/
   0.24999999999992,                       /*P0*/
   0.29729363682238E-3,                        /*Q2*/
```

```
    0.53567517645222E-1,                    /*Q1*/
    0.5,                                  /*Q0 and others*/
    2.0;                                  /*Used in divide*/
.ENDSEG;
```

Listing 2.11 Exponential Subroutine

2.8 POWER APPROXIMATION

The algorithm to approximate the power function (*pow(x,y)*, *x**y*, or x^y) is more complicated than the algorithms for the other standard floating-point functions. The power function is defined mathematically as

$$x ** y = exp(y * ln(x))$$

Unfortunately, computing pow(x,y) directly with the exponent and logarithm routines discussed in this chapter yields very inaccurate results. This is due to the finite word length of the processor. Instead, the implementation described here uses an approximation technique detailed in the [CODY]. Cody and Waite use pseudo extended-precision arithmetic to extend the effective word length of the processor to decrease the relative error of the function.

The key to pseudo extended-precision arithmetic is to represent the floating point number in the reduced form; for a floating-point number V

$$V = V_1 + V_2$$

where

$$V_1 = FLOAT(INTRND(V * 16))/16$$

and

$$V_2 = V - V_1$$

To compute the power function $Z = X^Y$, let

$$Z = 2^W$$

where

$$W = Y \log_2(X)$$

Let the input X be a positive floating-point number, and let $U = \log_2(X)$. The equation simplified is

$$W = Y * U$$

To implement this approximation accurately, you must compute Y and U to extended-precision using their reduced forms (Y_1+Y_2 and $U_1 + U_2$, respectively). U_1 and U_2 are calculated with the approximations outlined below. Y_1 and Y_2 are formed from the floating-point input Y.

To calculate U_1 and U_2, first represent the floating-point input, X, in the form

$$X = f * 2^m$$

where

$$1/2 \leq f < 1$$

The value of U_2 is determined using a rational approximation generated by Cody and Waite, and the value of U_1 is determined using the equation

$$U_1 = FLOAT(INTRND(U*16))/16$$

where p is an odd integer less than 16 such that

$$f = 2^{(-p/16)} * g/a$$

a = precalculated coefficients
$g = f * 2^r = f$ (in the case of non-decimal processors $r=0$)

The reduced form of W is derived from the values of U_1, U_2, Y_1, and Y_2. Since

$$W1 = m' - p'/16$$

$$Z = 2^{m'} * 2^{(-p'/16)} * 2^{W2}$$

where 2^{W2} is evaluated by means of another rational approximation.

2 Trigonometric, Mathematical & Transcendental Functions

2.8.1 Implementation

`POW.ASM` is an ADSP-21000 implementation of the power algorithm. When assembling the file `POW.ASM`, you can specify where coefficients are placed—either in data memory (DM) or program memory (PM)— by using the `-Didentifier` switch at assembly time.

For example, to place the coefficients in data memory use the syntax

```
asm21k -DDM_DATA pow
```

To place the coefficients in program memory use the syntax

```
asm21k -DPM_DATA pow
```

The power function approximation subroutine expects inputs, *X* and *Y*, to be floating-point values.

The first step of the subroutine is to check that *X* is non-negative. If *X* is zero or less than zero, the subroutine terminates and returns either the value zero or the value of *X*, depending on the conditions.

The second step is to calculate *f* such that

$$X = f * 2^m$$

Use the `LOGB` function to recover the exponent of X. Since the floating point format of the ADSP-210xx has a hidden scaling factor (see Appendix D, "Numeric Formats," in The *ADSP-21020 User's Manual* for details) you must add a one to the exponent. This new exponent, *m*, scales *X* to determine the value of f such that $1/2 \le f < 1$.

$$f = X * 2^{-m}$$

Note for these calculations, the value of *g* is equal to the value of *f*.

The value of *p* is determined using a binary search and an array of floating point numbers A_1 and A_2 such that sums of appropriate array elements represent odd integer powers of $2^{-1/16}$ to beyond working precision.

set $p = 1$
if $(g \le A_1(9))$, then p = 9
if $(g \le A_1(p+4))$, then $p = p + 4$
if $(g \le A_1(p+2))$, then $p = p + 2$

Next, you must determine the values of U_1 and U_2. To determine U_2, you must implement a rational approximation. The equation for U_2 is

$$U_2 = (R + z * K) + z$$

where K is a constant and z and $R(z)$ are determined as described below.

To determine the value of z, the following equation is used

$$z' = 2 * [g - A_1(p+1)] - A_2 ((p+1)/2)$$
$$z = z' + z'$$

At this point, $|z| \leq 0.044$.

To determine the value of $R(z)$, Cody and Waite derived coefficients (p_1, p_2) especially for this approximation. The equation is

$$R(z) = [(p2 * v) + p1] * v * z$$

where

$$v = z * z.$$

To determine the value of U_1

$$U_1 = REDUCE(U)$$
$$U_1 = FLOAT(INTRND(U * 16))/16$$

since

$$U = log_2(X)$$
$$= log_2(f * 2^m)$$
$$= log_2([2^{(-p/16)} * g/a] * 2^m)$$
$$= m - p/16$$

Therefore

$$U_1 = FLOAT(INTRND(16*m-p)) * 0.0625$$

Having calculated U_1 and U_2, reduce Y into Y_1 and Y_2:

$$Y_1 = REDUCE(Y)$$

$$Y_2 = Y - Y_1$$

and then calculate the value of W using the pseudo extended-precision product of U and Y with the following sequence of operations:

$$W = U_2 * Y + U_1 * Y_2$$
$$W_1 = REDUCE(W)$$
$$W_2 = W - W1$$
$$W = W_1 + U_1 * Y_1$$
$$W_1 = REDUCE(W)$$
$$W_2 = W_2 + (W - W_1)$$
$$W = REDUCE(W_2)$$
$$IW_1 = INT(16 * (W_1 + W))$$
$$W_2 = W_2 - W$$

Now compare IW_1 with the largest and smallest positive finite floating-point numbers to test for overflow. If an overflow occurs, the subroutine ends and an error value should be set.

For the next step IW_2 must be less than or equal to zero. If $W_2 > 0$, add one to IW_1 and subtract 1/16 from W_2.

Determine that values of m' and p' using the equations:

$$m' = IW_1/16 + 1$$
$$p' = 16 * m' - IW_1$$

You can now determine the value of Z

$$Z = 2^{m'} * 2^{(-p'/16)} * 2^{W2}$$

The value of $2^{W2} - 1$ is evaluated for $-0.0625 \le W_2 \le 0$ using a near-minimax polynomial approximation developed by Cody and Waite.

$$Z = W_2 * Q(W_2)$$

where $Q(W_2)$ is a polynomial in W_2 with coefficients q_1 through q_5. Therefore

$$Z = ((((q_5 * W_2 + q_4) * W_2 + q_3) * W_2 + q_2) * W_2 + q_1) * W_2$$

Now, add 1 to Z and multiply by $2^{(-p'/16)}$ using the equation

$$Z = (Z * A_1(p'+1)) + A_1(p1+1)$$

Finally, scale Z by the value of m' or $Z = ADX(Z, m')$

2.8.2 Code Listings

2.8.2.1 Power Subroutine

```
/*************************************************************
File Name
   POW.ASM

Version
   Version 0.04    7/6/90

Purpose
Subroutine to compute x raise to the y power of its two floating point
   inputs.
Equations Implemented
   Y=POW(X)

Calling Parameters
   F0 = X Input Value
   F1 = Y Input Value
   l_reg = 0
Return Values
   F0 = Exponential of input

Registers Affected
   F0, F1, F2, F4, F5, F7,
   F8, F9, F10, F11, F12, F13, F14, F15
   i_reg, ms_reg
Computation Time
   37 Cycles

# PM locations
   125 words

#DM locations
   33 words (could be placed in PM instead)
*************************************************************/
 #include “asm_glob.h”
 #include “pow.h”
 #define b_reg   B3
```

(listing continues on next page)

2 Trigonometric, Mathematical & Transcendental Functions

```
#define i_reg   I3
#define l_reg   L3
#define mem(i,m) DM(i,m)
#define m1_reg  M7
#define mm_reg  M6
#define ms_reg  M5
#define SPACE   DM


.SEGMENT/PM   rst_svc;
         jump pow;
.ENDSEG;

.SEGMENT/PM     pm_sram;

.PRECISION=MACHINE_PRECISION;

.GLOBAL     pow;


pow:             F0=PASS F0;                        /*Test for x<=0*/
         IF LT JUMP x_neg_error;                   /*Report error*/
         IF EQ JUMP check_y;                       /*Test y input*/
determine_m:     R2=LOGB F0;                        /*Get exponent of x input*/
         R2=R2+1;                                  /*Reduce to proper format*/
         R3=-R2;                                   /*Used to produce f*/
determine_g:     F15=SCALB F0 BY R3;                /* .5 <= g < 1*/
determine_p:     i_reg=a1_values;
         R14=1;
         R13=9;
         R10=i_reg;
         F12=mem(9,i_reg);                         /*Get A1(9)*/
         COMP(F12,F15), F12=mem(5,i_reg);          /*A1(9) - g*/
         F11=mem(13,i_reg);
         IF GE F12=F11;                            /*Use A(13) next*/
         IF GE R14=R13;                            /*IF (g<=A1(9)) p=9*/
         R9=R10+R14;
         i_reg=R9;
```

```
          R13=4;
          COMP(F12,F15), F12=mem(2,i_reg);      /*A1(p+4) - g*/
          F11=mem(6,i_reg);
          IF GE F12=F11;
          IF GE R14=R14+R13;                    /*IF (g<=A1(p+4)) p=p+4*/
          R13=2;
          COMP(F12,F15);                          /*A1(p+2) - g*/
          IF GE R14=R14+R13;                   /*IF (g<=A1(p+4)) p=p+2*/
 determine_z:    R14=R14+1, R4=R14;
          ms_reg=R14;
          i_reg=a1_values;
          R11=ASHIFT R14 BY -1;                /*Compute (p+1)/2*/
          F12=mem(ms_reg,i_reg);               /*Fetch A1(p+1)*/
          ms_reg=R11;
          i_reg=a2_values;                     /*Correction array*/
          F0=F12+F15;                          /*g + A1(p+1)*/
          F14=F15-F12, F11=mem(ms_reg,i_reg);
          F7=F14-F11, F12=F0;                  /*[g-A1(p+1)]-A2((p+1)/2)*/
          F11=2.0;

 __divide_:      F0=RECIPS F12;                 /*Get 4 bit seed R0=1/D*/
          F12=F0*F12;                          /*D(prime) = D*R0*/
          F7=F0*F7, F0=F11-F12;                /*F0=R1=2-D(prime), F7=N*R0*/
          F12=F0*F12;                          /*F12=D(prime)=D(prime)*R1*/
          F7=F0*F7, F0=F11-F12;                /*F7=N*R0*R1, F0=R2=2-D(prime)*/
          F12=F0*F12;                          /*F12=D(prime)=D(prime)*R2*/
          F7=F0*F7, F0=F11-F12;                   /*F7=N*R0*R1*R2, F0=R3=2-
D(prime)*/
          F7=F0*F7;                            /*F7=N*R0*R1*R2*R3*/

          F7=F7+F7;                          /* z = z + z*/
 determine_R:    i_reg=power_array;
          F8=F7*F7, F9=mem(p2,i_reg);             /* v = z * z */
          F10=F8*F9, F9=mem(p1,i_reg);         /* p2*v */
          F10=F10+F9;                          /* p2*v + p1 */
          F10=F10*F8;                          /* (p2*v + p1) * v */
          F10=F10*F7, F9=mem(K,i_reg);         /* R(z) = (p2*v+p1)*v*z */
 determine_u2:   F11=F10*F9;                    /* K*R */
          F11=F10+F11;                         /* K + K*R */
          F9=F9*F7;                            /* z*K */
```

(listing continues on next page)

```
         F9=F9+F11;                          /* R + z*K */
         F9=F9+F7;                           /* (R + z*K) + z*/
determine_u1:   R3=16;
         R2=R2*R3 (SSI);                     /* m*16 */
         R2=R2-R4;                           /* m*16-p */
         R3=-4;
         F2=FLOAT R2 BY R3;                  /*FLOAT(m*16-p)*.0625*/
determine_w:    R4=4;                        /*Used in reduce*/
         R8=FIX F1 BY R4;
         F8=FLOAT R8 BY R3;                  /* y1=REDUCE(Y) */
         F7=F1-F8;                           /* y2 = y - y1 */
         F15=F9*F1;                          /* U2*Y */
         F14=F2*F7;                          /* U1*Y2 */
         F15=F14+F15;                        /* W = U2*Y + U1*Y2*/
         R14=FIX F15 BY R4;
         F14=FLOAT R14 BY R3;                /* W1=REDUCE(W) */
         F13=F15-F14;                        /* W2 = W - W1 */
         F12=F2*F8;                          /* U1*Y1 */
         F12=F14+F12;                        /* W = W1 + U1*Y1 */
         R14=FIX F12 BY R4;
         F14=FLOAT R14 BY R3;                /* W1 = REDUCE(W) */
         F11=F12-F14;                        /* W-W1 */
         F13=F11+F13;                        /* W2 = W2 + (W-W1) */
         R12=FIX F13 BY R4;
         F12=FLOAT R12 BY R3;                /* W = REDUCE(W2) */
         F10=F12+F14;
         R10=FIX F10 BY R4;                  /* IW1 = INT(16*(W1+W)) */
         F13=F13-F12, R9=mem(bigx,i_reg);    /* W2 = W2 - W */
         COMP(R10,R9), R9=mem(smallx,i_reg); /* Test for overflow */
         IF GE JUMP overflow
         COMP(R10,R9);
         IF LE JUMP underflow;
flow_to_a:      F13=PASS F13;                /* W2 must be <=0 */
         IF LE JUMP determine_mp;
         F8=.0625;
         F13=F13-F8;
         R10=R10+1;
determine_mp:   R8=1;
         R10=PASS R10;
         IF LT R8=R8-R8;                     /* I=0 if IWI < 0 */
         R6=ABS R10;                         /* Take ABS for shift*/
         R7=ASHIFT R6 BY -4;                 /* IW1/16 */
         R6=PASS R10;
         IF LT R7=-R7;
         R7=R7+R8;                          /* m(prime) = IW1/16 + I */
         R6=ASHIFT R7 BY 4;                 /* m(prime)*16 */
         R6=R6-R10, F5=mem(q5,i_reg);       /* p(prime)  = 16*m(prime) - IW1 */
determine_Z:    F4=F5*F13, F5=mem(q4,i_reg);     /* q5*W2 */
         F4=F4+F5, F5=mem(q3,i_reg);             /* q5*W2 + q4 */
```

```
          F4=F4*F13;                          /* (q5*W2+q4)*W2 */
          F4=F4+F5, F5=mem(q2,i_reg);                 /* (q5*W2+q4)*W2+q3 */
          F4=F4*F13;                          /* ((q5*W2+q4)*W2+q3)*W2 */
          F4=F4+F5, F5=mem(q1,i_reg);                 /* ((q5*W2+q4)*W2+q3)*W2+q2 */
          F4=F4*F13;                          /* (((q5*W2+q4)*W2+q3)*W2+q2)*W2 */
          F4=F4+F5;                           /*
Q(W2)=(((q5*W2+q4)*W2+q3)*W2+q2)*W2+q1*/
          F4=F4*F13;                          /* Z = W2*Q(W2) */
          i_reg=a1_values;
          R6=R6+1;                            /* Compute p(prime)+1 */
          ms_reg=R6;
          F5=mem(ms_reg,i_reg);                       /* Fetch A1(p(prime)+1) */
          F4=F4*F5;                           /* A1(p(prime)+1)*Z */
          F4=F4+F5;                           /* Z = A1(p(prime)+1) +
A1(p(prime)+1)*Z */
          F0=SCALB F4 BY R7;                  /* Result = ADX(Z,m(prime)) */
 underflow:       F0=0;                       /*Set error also*/
          RTS;

 x_neg_error:     F0=0;                       /*Set an error also*/
          RTS;

 check_y:         F1=PASS F1, F0=F0;          /*If y>0 return x*/
          IF GT RTS;
 overflow:        RTS;                        /*Set an error also*/
```

```
.ENDSEG;

.SEGMENT/DM  dm_sram;

.VAR    a1_values[18] = 0,
                0x3F800000,
                0x3F75257D,
                0x3F6AC0C6,
                0x3F60CCDE,
                0x3F5744FC,
                0x3F4E248C,
                0x3F45672A ,
                0x3F3D08A3 ,
                0x3F3504F3 ,
                0x3F2D583E ,
                0x3F25FED6 ,
                0x3F1EF532 ,
                0x3F1837F0 ,
                0x3F11C3D3 ,
                0x3F0B95C1 ,
                0x3F05AAC3 ,
                0x3F000000;

.VAR    a2_values[9] =  0,
                0x31A92436,
                0x336C2A95,
                0x31A8FC24,
                0x331F580C,
                0x336A42A1,
                0x32C12342,
                0x32E75624,
                0x32CF9891;

.VAR    power_array[10]= 0.833333286245E-1,      /* p1     */
                         0.125064850052E-1,      /* p2     */
                         0.693147180556341,      /* q1     */
                         0.240226506144710,      /* q2     */
                         0.555040488130765E-1,   /* q3     */
```

```
                        0.961620659583789E-2,   /* q4     */
                        0.130525515942810E-2,   /* q5     */
                        0.44269504088896340736, /* K      */
                     2032.,                     /* bigx   */
                    -2015;                      /* smallx */
    .ENDSEG;
```

Listing 2.12 pow.asm

2.8.2.2 Global Header File

```
#define MACHINE_PRECISION   40
 #define MEMORY_PRECISION    32

 #ifdef PM_DATA

 #define b_reg   B11
 #define i_reg   I11
 #define l_reg   L11
 #define mem(m,i) PM(m,i)
 #define m1_reg  M14
 #define mm_reg  M13
 #define ms_reg  M12
 #define SPACE   PM
 #define Assembly_Library_Data_Space Lib_PMD

 #endif

 #ifdef DM_DATA

 #define b_reg   B3
 #define i_reg   I3
 #define l_reg   L3
 #define mem(i,m) DM(i,m)
 #define m1_reg  M7
 #define mm_reg  M6
 #define ms_reg  M5
 #define SPACE   DM
 #define Assembly_Library_Data_Space Lib_DMD

 #endif
```

Listing 2.13 asm_glob.h

2.8.2.3 Header File

```
/*
    This include file is used by the power routine of the assembly
    library
*/

 #define p1 0
 #define p2 1
 #define q1 2
 #define q2 3
 #define q3 4
 #define q4 5
```

Matrix Functions 3

Matrices are useful in image processing and graphics algorithms because they provide a natural two-dimensional format to store x and y coordinates. Many signal processing algorithms perform mathematical operations on matrices. These operations range from scaling the elements in the matrix to performing an autocorrelation between two signals described as matrices.

The three basic matrix operations discussed in this chapter are

- multiplying one matrix by a vector
- multiplying one matrix by another matrix
- finding the inverse of a matrix

This chapter describes three ADSP-21000 family assembly language subroutines that implement these operations. The matrix buffers, interrupt vector tables, and DAG registers must be set up by the calling routine before the routines can access the matrices. Optionally, each subroutine can include setup code so it can run independently of a calling routine. This setup code is conditionally assembled by using the assembler's `-D`*`identifier`* command line switch.

The implementations are based on the matrix algorithms described in [EMBREE91].

3 Matrix Functions

3.1 STORING A MATRIX

To minimize index register usage, the two-dimensional array matrix is stored in a single-dimensional array buffer and element positions are kept track of by the processor's DAG registers. The program can access any matrix element by using the index, modify, and length registers. The elements are stored in *row major order;* all the elements of the first row are first in the buffer, then all of the elements in the second row, and so forth.

For example, a 3×3 matrix with these elements

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

is placed in a nine element buffer in this order

$$\{A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{33}\}$$

The elements can be read continuously and consecutively if you

- use circular buffering
- use the correct modify values for index pointers
- keep track of the pointer in the matrix

To read the elements in a row consecutively, set the index pointer to the location of the first element in the row and set the modify value to 1.

To read the elements in a column consecutively, set the index register to the location of the first element in the column and set the modify value equal to the length of the row.

For example, to read the first column in the 3 ×3 matrix example, set the index pointer to point to A_{11} and the set modify value to three (3). After an indirect memory read of the first element, the index pointer points to A_{21}, the second column element.

This method of storing and accessing a matrix keeps available more DAG registers than using a different index pointer for each row of the matrix.

3.2 MULTIPLICATION OF A M×N MATRIX BY AN N×1 VECTOR

This section discusses how to multiply a two-dimensional matrix of arbitrary size, *M*×*N*, by a vector (one-dimensional matrix), *N*×*1*.

3.2.1 Implementation

The matrix elements are stored in the buffer `mat_a`, with length of *M*×*N* (where *M* is the number of rows in a matrix and *N* is the number of columns).

The vector is stored in a separate buffer, `mat_b`, with length of *N*. (The number of rows in the vector must equal the number of columns of the matrix.)

The result of the multiplication is another vector stored in a buffer, `mat_c`, with length of *M* (the number of rows of the first matrix).

The *M*×*N* matrix and *M*×1 result vector are stored in a data memory segment, while the *N*×1 multiply vector is stored in a program memory segment. This lets the algorithm take advantage of the dual fetch of the multifunction instructions to access the next two elements of the matrix and vector while multiplying the current two elements.

The multiplication of the matrix and vector is performed in a two-level deep loop. The inner loop, `row`, is a single instruction that performs the multiplication of the current two elements and accumulates the previous multiplication while fetching the next two elements to be multiplied. This `row` loop is performed *N* times (the number of columns) to account for the number of multiply/accumulate steps done for each row.

The outer loop, `column`, takes the final accumulated result and stores it in the `mat_c` result buffer. The loop also clears the accumulator result, R8, so that it can be used again for the next `row` loop. The `column` loop is performed *M* times (the number of rows) to account for the number of times it has to perform the `row` loop operations.

For efficient loops, the operation of multiplication, accumulation and the next two data fetches are all performed in the same multifunction instruction. Each multiply is done on operands which were fetched on the pervious iteration of the inner loop. Similarly, each addition accumulates the product from the multiply in the previous loop iterations. Before the loops are entered the first time, the operands must be preloaded from memory.

3 Matrix Functions

This technique of preloading registers so multifunction instructions can be used in the loop body is called "rolling the loop." To roll the loop, the first instructions outside of the `row` and `column` loops clear the accumulator, fetch the first two elements to be multiplied and multiplies them while fetching the next two elements.

One cycle can be saved when clearing the accumulator by performing an exclusive or operation (`XOR`) between the accumulator register (R8) and itself. This lets the processor fetch the next two elements while performing a computation, which is faster than using one cycle to clear the accumulator by loading R8 with a zero and then a second cycle to perform the fetches. The processor cannot perform a register load and two data fetches in the same cycle.

This trick of combining the computation with data fetches in a single instruction is also used for the last instruction of the `column` loop when clearing the accumulator for the next loop and storing the final accumulation result to the `mat_c` buffer. Since this loop may be performed many times (depending on the number of rows in the matrix), it can greatly reduce the time spent executing the algorithm.

The accumulation operation of the multifunction instruction used in the `row` loop is performed last. When the accumulation of the last element is performed for the current row, the multiplier has already multiplied the first two elements of the next row and the second two elements have been fetched. Provided the current row is not the last one, the extra multiplication and data fetches roll over into the next iteration of the loop.

When performing the accumulation on the last elements of the last row, the index pointers of the input buffers wrap around to the start of the buffer; the multiplication and data fetches for the first row are repeated. Since those operations are redundant, their destination registers can be written over after the routine completes. Note that the index pointers are also modified and point to the third elements in the matrices when the routine is finished. Therefore, the pointers must be restored if the same matrices must be used in a subsequent routine.

3.2.2 Code Listing–M×N By N×1 Multiplication

```
/*****************************************************************************

File Name
   MxNxNx1.ASM

Version
   25-APR-91

Purpose
   Matrix times a Vector.

   Matrix dimensions are arbitrary. Matrix A accessed as a circular buffer so
that the         last iteration of the inner loop will do a dummy read from a
known location.

   Use the -Dexample Assembler Preprocessor Switch to include assembly of an
   example calling routine

Equations Implemented
   [Mx1]=A[MxN]*B[Nx1]

Calling Parameters
   Constants: m, n
   pm(mat_b[n]) row major, dm(mat_a[m*n]) row major,
   M1=1;
   M9=1;
   B0=mat_a;     L0=@mat_a;
   B1=mat_c;     L1=0;
   B8=mat_b;     L8=@mat_b;

Return Values
   dm(mat_c[m]) row major

Registers Affected
   F0,F4,F8,F12, I0,I1,I8

Cycle Count
   cycles=6+M(3+N)+5   (entrance + core + 5 cache)

# PM Locations
   pm code=8 words, pm data=n words

# DM Locations
   dm data=m*n+m words
*****************************************************************************/
```

(listing continues on next page)

3 Matrix Functions

```
/* dimension constants */
#define          M 4
#define          N 4

#ifndef example
.GLOBAL mxnxnx1;
.EXTERN mat_a, mat_b,mat_c;
#endif

#ifdef example
.SEGMENT/DM dm_data;
.VAR             mat_a[M*N]=”mat_a.dat”;
.VAR             mat_c[M];
.ENDSEG;

.SEGMENT/PM pm_data;
.VAR             mat_b[N]=”mat_bb.dat”;
.ENDSEG;

.SEGMENT/PM      rst_svc;
          dmwait=0x21; /* set dm waitstates to zero */
          pmwait=0x21; /* set pm waitstates to zero */
          jump setup;
.ENDSEG;

          /* example calling code */
.SEGMENT/PM pm_code;
setup:           m1=1;
          m9=1;
          b0=mat_a;     l0=@mat_a;
          b1=mat_c;     l1=0;
          b8=mat_b;     l8=@mat_b;
          call mxnxnx1;
          idle;
.ENDSEG;
#endif

          /* matrix multiply starts here */
.SEGMENT/PM pm_code;
mxnxnx1:  r8=r8 xor r8, f0=dm(i0,m1), f4=pm(i8,m9); /* clear f8 */
          f12=f0*f4,  f0=dm(i0,m1), f4=pm(i8,m9);
          lcntr=M, do column until lce;
             lcntr=N, do row until lce;
row:            f12=f0*f4, f8=f8+f12, f0=dm(i0,m1), f4=pm(i8,m9);
column:       r8=r8 xor r8, dm(i1,m1)=f8;
          rts;
.ENDSEG;
```

Listing 3.1 MxNxNx1.asm

3.3 MULTIPLICATION OF A M×N MATRIX BY A N×O MATRIX

This section discusses how to multiply two matrices of arbitrary size.

3.3.1 Implementation

The two input matrices are stored in separate data memory and program memory segments so multifunction instructions can be used to produce efficient code.

- The first input matrix ($M \times N$), `mat_a`, is stored in data memory.
- The second input matrix ($N \times O$), `mat_b`, is stored in program memory.
- The result matrix ($M \times O$), `mat_c`, is stored in data memory.

You can think of matrix-matrix multiplication as several matrix-vector multiplications. The matrix is always the first input matrix and each "vector" is a column in the second matrix. As discussed in the previous section, the matrix-vector multiplication code consisted of two loops: `row` and `col`. To obtain the matrix-matrix multiplication code a third loop is added, `colrow`, that simply repeats the entire block of matrix-vector code for the number of columns in the second matrix.

Repeating this code, however, requires that the modifications of the index pointers for each matrix be handled differently. The modify value for `mat_a (m1)` is still set to 1 to increment through the matrix row by row. However, to consecutively increment through the columns of `mat_b`, the modify value (m10) needs to be set to the number of columns in that matrix. For this code example, the second matrix is a 4×4 matrix, so the modify value is 4. The result matrix is written to column by column and has the same modify value.

After a single loop through the matrix-vector code, a column in the result matrix is complete. Because the code is looped to reduce the number of cycles inside the loop, the position of the index pointers are incorrect to perform the next matrix-vector multiplication. Modify instructions are included at the end of the `rowcol` loop that modify the index pointers so they begin at the correct position. The index register for `mat_a` is modified to point to the beginning of the first row and first column. The index registers (pointers) for both `mat_b` and the result matrix `mat_c` are modified to point to the beginning of the next column.

3 Matrix Functions

The pointer to the matrix `mat_a` is modified to point to the first element of the first row by performing a dummy fetch with a modify value of –2. (Because the loop is rolled, the DAG fetches the first two row elements again.)

A dummy fetch is also used to modify the matrix pointer for `mat_b` to point to the first element of the next column. The modify value for this dummy fetch is *–(O * 2 – 1)*, where *O* is the number of columns in the matrix. Because of loop rolling, the index pointer points to the third column element. Therefore, the index pointer needs to be adjusted backwards by two rows minus one element. For this code example, the index pointer points to the third column position. To make the pointer point to the first element of the next column, modify the pointer by *–(4 * 2 – 1) = –7*.

Finally, the pointer to the result matrix `mat_c` is modified to point to the first element of the next column by modifying the index pointer by one. Since the instruction that writes the result is the last one in the loop, loop rolling does not affect this index pointer. Dummy reads modify the index registers, instead of modify instructions, so that a multifunction instruction can be performed. This reduces the number of cycles in the loop.

The loop `colrow` is then executed again, which calculates the result for the next column in the result matrix. The loop `colrow` repeats until all the columns in the result matrix are filled.

The final pointer positions are as follows

- position (1,1) for `mat_a` (beginning of the buffer and matrix),
- position (2,1) for `mat_b` (row 2, column 1)
- position (2,1) for the result `mat_c` (row 2, column 1).

To use the same matrices in a subsequent routine, first reset the index pointers to the beginning of each buffer.

3.3.2 Code Listing–M×N By N×O Multiplication

```
/
*******************************************************************************

File Name
   MxNxNxO.ASM

Version
   25-APR-91

Purpose
   Matrix times a Matrix.

   The three matrices have arbitrary dimensions. Matrix A accessed as a
   circular buffer so that the last iteration of the inner loop will do a
   dummy read from a known location.

   Use the -Dexample Assembler Preprocessor Switch to include assembly of an
   example calling routine

Equations Implemented
   C[MxO]=A[MxN]*B[NxO]

Calling Parameters
   Constants: m, n, o
   pm(mat_b[N*O]) row major, dm(mat_a[M*N]) row major
   dm(mat_c[M*O]) row major
   M1=1;
   M2=-2;
   M3=o;
   M9=-(o*2-1);
   M10=o;
   B0=mat_a;     L0=@mat_a;
   B1=mat_c;     L1=@mat_c;
   B8=mat_b;     L8=@mat_b;


Return Values
   F0,F4,F8,F12, I0,I9, B8

Registers Affected
   F0,F4,F8,F12, I0,I1,I8

Cycle Count
   cycles=4+o(m(n+2)+5)+7     (entrance + core + 7 cache)

# PM Locations
   pm code=11 words, pm data=NxO words,

# DM Locations
   dm data=MxN+MxO words
*********************************************** (listing continues on next page) ***/
```

3 Matrix Functions

```
/* dimension constants */
#define         M 4
#define         N 4
#define         O 4

#ifndef example
.GLOBAL mxnxnxo;
.EXTERN mat_a, mat_b, mat_c;
#endif

#ifdef example
.SEGMENT/DM dm_data;
.VAR            mat_a[M*N]=”mat_a.dat”;
.VAR            mat_c[M*O];
.ENDSEG;

.SEGMENT/PM pm_data;
.VAR            mat_b[N*O]=”mat_b.dat”;
.ENDSEG;

.SEGMENT/PM     rst_svc;     /* reset vector */
          dmwait=0X21; /* set dm waitstates to zero */
          pmwait=0X21; /* set pm waitstates to zero */
          jump setup;
.ENDSEG;

          /* example calling code */
.SEGMENT/PM pm_code;
setup:          m1=1;
          m2=-2;
          m3=O;
          m9=-(O*2-1);
          m10=O;
          b0=mat_a;    l0=@mat_a;
          b1=mat_c;    l1=@mat_c;
          b8=mat_b;    l8=@mat_b;
          call mxnxnxo;
          idle;
.ENDSEG;
#endif

          /* matrix multiply starts here */
.SEGMENT/PM pm_code;
mxnxnxo:  lcntr=O, do colrow until lce;
             r8=r8 xor r8, f0=dm(i0,m1), f4=pm(i8,m10); /* clear f8 */
             f12=f0*f4,  f0=dm(i0,m1), f4=pm(i8,m10);
             lcntr=M, do column until lce;
                 lcntr=N, do row until lce;
row:                f12=f0*f4, f8=f8+f12, f0=dm(i0,m1), f4=pm(i8,m10);
column:           r8=xor r8, dm(i1,m3)=f8;
                 f0=dm(i0,m2), f4=pm(i8,m9); /* modify with dummy fetches */
colrow:       modify(i1,1);
          rts;
.ENDSEG;
```

Listing 3.2 MxNxNxO.asm

3.4 MATRIX INVERSION

The inversion of a matrix is used to solve a set of linear equations. The format for the linear equations is

$$Ax = b$$

The vector x contains the unknowns, the matrix A contains the set of coefficients, and the vector b contains the solutions of the linear equations. The matrix A must be a non-singular square matrix. To get the solution for x, multiply the inverse of matrix A by the constant vector, b. The inverse matrix is useful if a different constant vector b is used with the same equations. The same inverse can be used to solve for the new solutions.

Because of round-off error that occurs during the elimination process, it is hard to get accurate results when inverting large matrices. The Gauss-Jordan method elimination with full pivoting, however, provides a highly accurate matrix inverse.

Gauss-Jordan elimination can become numerically unstable unless pivoting is used. Full pivoting is the interchanging of rows and columns in a matrix to have the largest magnitude element on the diagonal of the matrix. This diagonal element, called the pivot, is then used to divide the other elements of the row. The row is used to eliminate other column elements to obtain the identity matrix. The same elimination procedures are performed on an original identity matrix. Once the matrix is reduced to the identity matrix, the original identity matrix will contain the inverse matrix. This resulting matrix must be adjusted for any interchanging of rows and columns.

The Gauss-Jordan algorithm is an in-place algorithm: the input matrix and output result are stored in the same buffer of data.

The algorithm is subdivided into five sections:

- The first section searches for the largest element of the matrix.
- The second section places that element on the diagonal of the matrix making it the pivot element. The row that contained the pivot element is marked so that it won't be used again.
- The third section of code divides the rest of the row by that pivot element.

- The fourth section performs the in-place elimination.
- The first four sections are repeated until all of the rows of the matrix have been checked for a pivot point.
- The fifth section performs the corrections for swapping rows and columns.

3.4.1 Implementation

The matrix is in data memory and uses $N \times N$ locations, where N is the size of the matrix (N=3 for a 3×3 matrix). The pivot flag (`pf`) and swap column (`swc`) arrays are also stored in data memory. The swap row array (`swr`) is in program memory.

This routine uses all of the universal registers (F0-F15) and eight of the DAG registers (I0-I8) as either data storage or temporary registers. Therefore, if this routine is called from a routine that uses these registers, switch to the secondary registers before starting the routine. Make sure to switch back to the primary registers when done.

The first four sections of the algorithm are enclosed in the `full_pivot` loop. Each pass through the loop searches for the largest element in the matrix, places that element on the diagonal, and performs the in-place elimination. The loop repeats for every row of the matrix.

The first section of the algorithm searches the entire matrix for the largest magnitude pivot value in the nested loops `row_big` and `column_big`. At the beginning of each loop, it checks if the pivot flag is set for that row or column. If the pivot flag is set, that row or column contains a pivot point that has already been used. Since any element in a row or column that previously contained a pivot point cannot be reused, the loop is skipped and the index pointer is modified to point to the next row or column.

The loop performs a comparison of all the elements in the matrix and the largest value is stored in register F12—the pivot element. The row that contained the pivot point is stored in the `pf` buffer so that any elements in that row will not be used again. A test is performed to see if the pivot element is a zero. If the pivot point is zero, the matrix is singular and does not have a realizable inverse. The routine returns from the subroutine and the error flag is register F12 containing a zero.

The second section of the algorithm checks if the pivot element is on the diagonal of the matrix. If the pivot element is not on the diagonal, corresponding rows and columns are swapped to place the element on the diagonal. The position of the pivot element is stored in the counters R2 and R10. If these two numbers are equal, then the element is already on a diagonal and the algorithm skips to the next section of the algorithm. If the numbers are not equal, the loop `swap_row` is performed. This loop swaps the corresponding row and column to place the pivot element on the diagonal of the matrix. The row and column numbers that were swapped are stored in separate arrays called `swr(row)` and `swc(column)`. These values will be used in the fifth section to correct for any swapping that has occurred.

The third section of the algorithm divides all the elements of the row containing the pivot point by the pivot point. The inverse of the pivot point is found with the macro `DIVIDE`. The result of the macro is stored in the f1 register. The other elements in the row are then multiplied by the result in the loop `divide_row`.

The fourth section of the algorithm performs the in place elimination. The elimination process occurs within the two loops `fix_row` and `fix_column`. The results of the elimination replace the original elements of the matrix.

These four sections described are repeated *N* times, where *N* is the number of rows in the matrix.

The fifth section of the algorithm is executed after the entire matrix is reduced. This section fixes the matrix if any row and column swapping was done. The algorithm reads the values stored in the arrays `swr` and `swc` and swaps the appropriate columns if the values are not zero.

3 Matrix Functions

3.4.2 Code Listing—Matrix Inversion

```
/
*****************************************************************************

File Name
   MATINV.ASM

Version
   May 6 1991

Purpose
   Inverts a square matrix using the Gauss-Jordan elimination.
   algorithm with full pivoting.

   See  P.M. Embree and B. Kimble. C Language Algorithms For Digital
   Signal Processing. Chap. 6, Sect. 6.2.3, pp. 326-329.Prentice-Hall, 1991

Equations Implemented
   C[MxO]=A[MxN]*B[NxO]

Calling Parameters
           dm(mat_a[n*n]) row major, dm(pf[n+1]), dm(swc[n]);
           pm(swr[n]);
           r14=n;      (n= number of rows (columns))
           m0=1; m1=-1;
           m8=1; m9=-1;
           b0=mat_a;
           b1=pf;
           b7=swc; l7=0;
           b8=swr; l8=0;

Return Values
   dm(mat_a[n*n]) row major;
   f12=0.0 -> matrix is singular


Registers Affected
   f0 - f15,
   i0 - i7, i8, m2

Cycle Count
    maximum number= worst case= 7.5n**3+25n**2+25.5n+23 (approximated)

# PM Locations
   pm code= 93 words, pm data= n words

# DM Locations
   dm data= n*n+2n+1 words
```

```
*****************************************************************************/

/* To assemble the example below type the following command

                  asm21k -Dexample matinv  */

#include  “macros.h”

#define   n      3

#ifndef          example
.GLOBAL          mat_inv;
.EXTERN          mat_a;
#endif

#ifdef           example
.SEGMENT/DM      dm_data;
.VAR      mat_a[n*n]= “mat_a1.dat”;
.VAR      pf[n+1];
.VAR      swc[n];
.ENDSEG;

.SEGMENT/PM rst_svc;
     dmwait=0x21;      /*set dm waitstates to zero*/
          pmwait=0x21;/*set pm waitstates to zero*/
     jump setup;
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      swr[n];
.ENDSEG;
          /* example calling code */

.SEGMENT/PM      pm_code;
setup:           b0=mat_a; /*i0 -> a(row,col)*/
          b1=pf;   /*i1 -> pf= pivot_flag*/
          b7=swc;  /*i7 -> swc= swap_col*/
          b8=swr;  /*i8 -> swr= swap_row*/
          l7=0;
          l8=0;
          m0=1;
          m1=-1;
          m8=1;
          m9=-1;
          r14=n;
          call mat_inv;
```

(listing continues on next page)

```
          idle;
.ENDSEG;
#endif

                /* Matrix inversion starts here */
.SEGMENT/PM     pm_code;
mat_inv:  r13=r14*r14(ssi), b3=b0;
          l0=r13;  /*matrix in a circular data buffer*/
          b4=b0;
          b5=b0;
          b6=b0;
          l3=l0;
          l4=l0;
          l5=l0;
          l6=l0;
          r13=r14+1, b2=b1;
          l1=r13;  /*pf in a circular data buffer*/
          l2=l1;
          f9=0.0;
          f8=2.0;  /*2.0 is required for DIVIDE_macro*/
          f7=1.0;  /*1.0 is a numerator for DIVIDE_macro*/
          r13=fix f9, m2=r14;
          lcntr=r14, do zero_index until lce;
           dm(i7,m0)=r13, pm(i8,m8)=r13;
zero_index: dm(i1,m0)=r13;
          f0=pass f9, dm(i1,m0)=r13; /*f0= big*/

          lcntr=r14, do full_pivot until lce;
                /*find the biggest pivot element*/
           r1=pass r13, r11=dm(i1,1); /*r1= row no., r11= pf(row)*/
           lcntr=r14, do row_big until lce;
              r11=pass r11, i4=i3; /*check if pf(row) is zero*/
              if ne jump (PC,12), f4=dm(i0,m2); /*i0 -> next row*/
              r5=pass r13, r15=dm(i2,1); /*r5= col no., r15= pf(col)*/
              lcntr=r14, do column_big until lce;
                 r15=pass r15; /*check if pf(col) is zero*/
                 if ne jump column_big (db);
                 f4=dm(i0,1); /*f4= a(row,col)*/
                 f6=abs f4;
                 comp(f6,f0); /*compare abs_element to big*/
                 if lt jump column_big;
                 f0=pass f6, f12=f4; /*f0= abs_element, f12= pivot_element*/
                 r2=pass r1, r10=r5; /*r2= irow, r10= icol*/
column_big:       r5=r5+1, r15=dm(i2,1);
row_big:    r1=r1+1, r11=dm(i1,1);

          /*swap rows to make this diagonal the biggest absolute pivot*/
           f12=pass f12, m5=r10; /*check if pivot is zero, m5= icol*/
           if eq rts; /*if pivot is zero, matrix is singular*/
           r1=r2*r14 (ssi), dm(m5,i1)=r5; /*pf(col) not zero*/
           r5=r10*r14 (ssi), m6=r1;
           comp(r2,r10), r1=dm(i3,m6); /*i3 -> a(irow,col)*/
           dm(i7,m0)=r10, pm(i8,m8)=r2; /*store icol in swc and irow in swr*/
                   if eq jump row_divide (db);
           r2=pass r13, m7=r5;
```

```
            modify(i4,m7); /*i4 -> a(icol,col)*/
            i5=i4;
            lcntr=r14, do swap_row until lce;
             f4=dm(i3,0);             /*f4= temp= a(irow,col)*/
             f0=dm(i5,0);             /*f0= a(icol,col)*/
             dm(i3,1)=f0;             /*a(irow,col)= a(icol,col)*/
swap_row:          dm(i5,1)=f4;       /*a(icol,col)= temp*/

                  /*divide the row by the pivot*/
row_divide:        f6=pass f7, i5=i4;
            DIVIDE(f1,f6,f12,f8,f3); /*f1= pivot_inverse*/
            i6=i5;
            f4=dm(i4,1);
            lcntr=r14, do divide_row until lce;
               f5=f1*f4, f4=dm(i4,1);
divide_row:    dm(i6,1)=f5;
            dm(m5,i5)=f1;

                  /*fix the other rows by subtracting*/
            lcntr=r14, do fix_row until lce;
                comp(r2,r10), i6=i5; /*check if row= icol*/
                if eq jump (PC,8), f4=dm(i0,m2); /*i0 -> next row*/
                f4=dm(m5,i0); /*temp= a(row,icol)*/
                dm(m5,i0)=f9;
                f3=dm(i6,1);
                lcntr=r14, do fix_column until lce;
                   f3=f3*f4, f0=dm(i0,0);
                   f0=f0-f3, f3=dm(i6,1);
fix_column:         dm(i0,1)=f0;
fix_row:       r2=r2+1;
full_pivot:  f0=pass f9, i3=i0;

                  /*fix the affect of all the swaps for final answer*/
          r0=dm(i7,m1), r1=pm(i8,m9); /*i7 -> swc(N-1), i8 -> swr(N-1)*/
          r0=dm(i7,m1), r1=pm(i8,m9); /*r0= swc(N-1), r1= swr(N-1)*/
          lcntr=r14, do fix_swap until lce;
             comp(r0,r1), m5=r0; /*m5= swc(swap)*/
             if eq jump fix_swap;
             m4=r1; /*m4= swr(swap)*/
             lcntr=r14, do swap until lce;
             f4=dm(m4,i0); /*f4= temp= a(row,swr(swap))*/
             f0=dm(m5,i0); /*f0= a(row,swc(swap))*/
             dm(m4,i0)=f0; /*a(row,swr(swap))= a(row,swc(swap))*/
             dm(m5,i0)=f4; /*a(row,swc(swap))= temp*/
swap:        modify(i0,m2);
fix_swap: r0=dm(i7,m1), r1=pm(i8,m9);
          rts;
.ENDSEG;
```

Listing 3.3 matinv.asm

3.5 REFERENCES

[EMBREE91] Embree, P. and B. Kimble. 1991. *C Language Algorithms For Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall.

FIR & IIR Filters 4

Digital filtering algorithms are widely used in DSP-based applications, including (but not limited to)

- Audio processing
- Speech compression
- Modems
- Motor control
- Video and image processing

Historically, electronics designers implemented filters with analog components. With the advent of digital signal processing, designers have a superior alternative: filters implemented in software running on DSPs. Digital filters have many advantages that make them more attractive than their analog predecessors:

- Digital filters have transfer functions that can comply to rigorous specifications. The stop band can be highly attenuated without sacrificing a steep transition band.
- DSP filters are programmable. The transfer function of the filter can be changed by changing coefficients in memory. A single hardware design can implement many different filters, by merely changing the software.
- The characteristics of DSP filters are predictable. Available filter design software packages can accurately profile the performance of a filter before it is implemented in hardware. Digital filter designers have the flexibility to try alternative designs to get an expectation of filter performance.

4 FIR & IIR Filters

- Unlike analog filters, the performance of digital filters is not subject to environmental changes, such as voltage and temperature.
- Often, digital filters can be implemented at a lower cost than complex analog filters.

This chapter presents two classes of digital filters and their software implementations on ADSP-21000 family processors. The first section of this chapter discusses the Finite Impulse Response (FIR) filter and the second section discusses the Infinite Impulse Response (IIR) filter.

4.1 FIR FILTERS

An FIR filter is a weighted sum of a finite set of inputs. The equation for an FIR filter is

$$y(n)=\sum_{k=0}^{m-1} a_k x(n-k)$$

where

x(n – k) is a previous history of inputs
y(n) is the filter output at time *n*
a_k is a vector of filter coefficients

The FIR code is a software implementation of this equation.

FIR filtering is a convolution in time. The FIR filter equation is similar to the convolution equation:

$$y(n)=\sum_{k=0}^{\infty} h(k)\, x(n-k)$$

FIR & IIR Filters 4

FIR filters have several advantages that make them more desirable than IIR filters for certain design criteria

- FIR filters can be designed to have linear phase. In many applications, phase is a critical component of the output. For example, in video processing, if the phase information is corrupted the image becomes unrecognizably distorted.

- FIR filters are always stable because they are made up solely of zeros in the complex plane.

- Overflow errors are not problematic because the sum of products operation in the FIR filter is performed on a finite set of data.

- FIR filters are easy to understand and implement. They can provide quick solutions to engineering problems.

4.1.1 Implementation

The FIR filter program is presented in this section as an example, but it easily can be modified and used in a real world system. It is helpful to read the actual program listing along with the following description.

The FIR filtering code uses predefined input samples stored in a buffer and coefficients that are stored in a data file. The code executes a five-tap filter on nine predefined samples. After the program processes the ninth sample it enters an infinite loop.

The code for the FIR filter consists of two assembly-language modules, `firtest.asm` and `fir.asm`. The `firtest.asm` module sets up buffers and pointers and calls `fir.asm`, the module that performs the multiply-accumulate operations. Two other files are needed: an architecture file and a data file. The file `generic.ach` is the architecture file, which defines the hardware in terms of memory spaces and peripherals. The `fircoefs.dat` file contains a list of filter coefficients for the filter.

The code in `firtest.asm` is executed first. The `firtest.asm` code starts by defining the constants `TAPS` and `SAMPLES`.

- `TAPS` is the number of taps of the filter. To change the number of taps, change the value of `TAPS` and add or delete coefficients to the file `FIRCOEFS.DAT`.

- `SAMPLES` is the constant that represents the number of samples that are input into the filter. If you need to change the number of samples, change `SAMPLES` and add or delete the number of predefined samples to the input buffer.

The next part of `firtest.asm` defines four buffers, `COEFS`, `DLINE`, `INBUF`, and `OUTBUF`:

- `COEFS`, a circular buffer in program memory, is filled with the filter coefficients data from the `fircoefs.dat` file.

  Note: The coefficients in `fircoefs.dat` are ordered so the *k=max* coefficient comes *first*, because the FIR loop processes the oldest input sample first and then the next oldest and so on. This means that the coefficient buffer must store the coefficient associated with the oldest value in the delay line in the first position in the coefficient buffer.

- The buffer `DLINE`, in data memory, contains the delay line, a circular buffer of past samples. The `COEFS` and `DLINE` buffers are used in the `fir.asm` module and hold the values that are multiplied and accumulated.

- `INBUF`, a non-circular buffer, contains the predefined input samples and is initialized when it is defined.

- `OUTBUF`, a non-circular buffer, holds the output samples.

The executable code starts at the label `INITIAL_SETUP`. It is placed at program memory location 0x8, which is the address of the reset interrupt service routine. The first instruction executed is a delayed branch jump to the label `BEGIN`. In the two cycles that it takes to branch, the memory wait states for program and data memory are set. At the `BEGIN` label, the data address generator registers are assigned to data buffers. The following four registers are defined to point to buffers:

Register	*Buffer Name*	*Length*	*Description*
I0——>	`dline`	L0=TAPS	delay line
I1——>	`inbuf`	L1=0	input buffer
I2——>	`outbuf`	L2=0	outbuffer
I8——>	`coefs`	L8=TAPS	coefficients

If the length of a buffer is not zero, the buffer is circular.

After the buffer is associated with data address registers, the code initializes the delay line. At the label `FIR_INIT` a loop that puts zeros in the delay line buffer starts. This initialization loop is necessary because without it the contents of the buffer would be undefined until five samples are accessed.

The actual filtering algorithm is implemented as a loop. The loop is executed `SAMPLE` number of times (once for each input sample) and ends at the label `FILTERING`. Iterating through this loop is equivalent to varying n in the FIR filter equation. This loop (in `FIRTEST.ASM`) calls `FIR` (in `FIR.ASM`) with a delayed branch. In the two cycles that it takes to branch, two transfers happen:

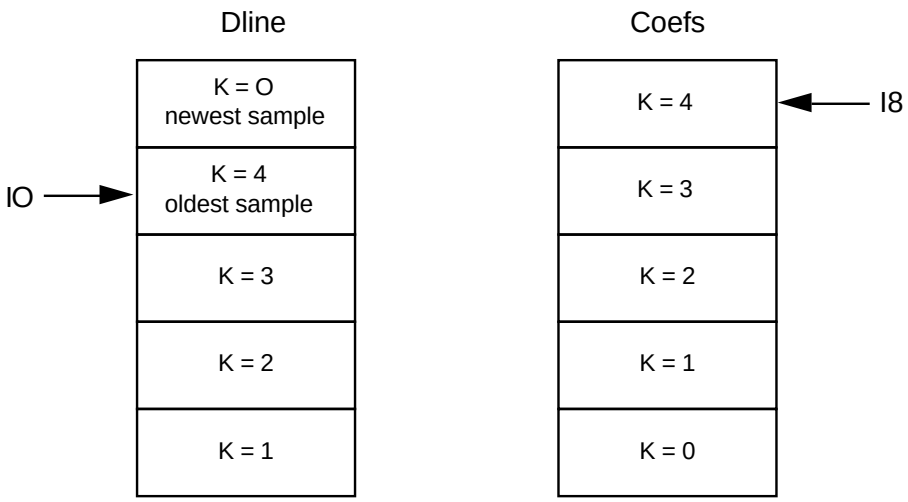
- A value from the input buffer is placed in F0. This is how the input sample is passed to the `FIR.ASM` module.
- R1 is set to a value that is the amount of times that the loop in the FIR module has to repeat. As shown by the FIR equation, R1 is one less than the number of multiplies that has to be executed for a given value of n.

After the code in `FIR.ASM` completes, control returns to `FIRTEST.ASM`. The instruction at the line labeled `FILTERING` writes a result to the output buffer. After the `FILTERING` loop completes, the execution goes into an infinite loop.

The sum of products operations, which comprise the core loop of the FIR filter, are executed by `FIR.ASM`. The code starts by zeroing some registers and loading data from buffers into other registers. At the line labeled `FIR`, a trick is used to zero the R12 register and fetch an input sample to the delay line buffer in a single cycle: use an `XOR` to zero R12 instead of using an immediate move. The `XOR` is an ALU compute operation. The ADSP-21000 family allows for a computation and a data movement to be executed in the same instruction. If an immediate move instruction was used to set R12 to zero, the data move could not be executed in the same instruction.

Note that the pointer is post modifed when the input sample moves to the delay line buffer. Because of this, the coefficients array must be arranged so that the coefficient associated with $K = max$ is first in the array. The modify instruction moves the delay line pointer to the oldest value in the delay line. The input sample just written to the delay line buffer is not accessed until the rest of the buffer is accessed. See Figure 4.1.

4 FIR & IIR Filters



Position of pointers for delay line buffer and coefficent buffer after a new sample has just been added to the delay line. Note the order of the coefficients. It refers to the equation in section 4.4.

Figure 4.1 Delay Line

The `MACS` loop, which computes the sum of products and executes *TAPS – 1* number of times, efficiently uses the multifunction capabilities of the ADSP-21000 architecture. A multifunction instruction can fetch two operands and perform a multiplication and an addition operation in a single cycle.

- The multiplication works on the operands fetched in the previous cycle. Therefore, the coefficients and data must be loaded outside the sum of products loop.

- The operands for the addition are the results of the multiplication; valid operands for the addition are not generated until the loop executes twice. For the first two iterations of the loop, the code uses "dummy" operands of zero. The third time through the loop and after, the multiplication generates two valid operands for the addition.

The code and memory usage for this algorithm is determined by the number of taps the filter has. The code in `FIR.ASM` is common to FIR filter implementations. The code in `FIRTEST.ASM` varies depending on how coefficients and data are input into the filter, and therefore is not included in the benchmark.

The code in `FIR.ASM` takes seven memory locations for instructions. In addition to the instruction memory requirement, there is also a requirement for the delay line and coefficient buffers. The delay line buffer is stored in data memory and the coefficient buffer is stored in program memory. The length of these buffers is equal to the filter's number of taps.

The number of cycles needed to execute the code is *7+ (number of TAPS)*. This number is derived by the following calculation: three instructions plus one cache miss for the first three lines of code; *(TAPS – 1)* plus one cache miss for the `MACS` loop; then three instructions to finish off the code. This formula for the number of cycles is:

cycles = 3+ (TAPS – 1) + 3 + 2 cache misses = 7 + number of taps

A cache miss occurs when an instruction that requires two program memory fetches and a fetch from data memory is executed for the first time. When executing this instruction for the first time, the opcode is loaded in to the cache. The next time that instruction is executed, the opcode can be fetched from the cache, and the program memory bus is free to access data.

4 FIR & IIR Filters

4.1.2 Code Listings

4.1.2.1 Example Calling Routine

```
/************************************************************************

File Name
        FIRTEST.ASM

Version
        .03     11/2/93

Purpose
        This program is a shell program for the FIR.ASM code.  This program
        initializes buffers and pointers.  It also calls the FIR.ASM code.

************************************************************************/

#define SAMPLES 0x9                     /*number of input samples to be
filtered*/
#define TAPS 5
.EXTERN fir;

.SEGMENT /PM pm_data;
.VAR    coefs[TAPS] = “fircoefs.dat”;   /* FIR coefficient stored in file    */
.ENDSEG;

.SEGMENT /DM dm_data;
.VAR    dline[TAPS];                    /* buffer that holds the delay line */
.ENDSEG;

.SEGMENT /DM dm_data;
.VAR    inbuf[SAMPLES]=   1.,  2.,  3.,  4.,  5.,  6.,  7., 8., 9.;
.VAR    outbuf[SAMPLES]=-99.,-99.,-99.,-99.,-99.,-99.,-99.,-99.,-99.;
.ENDSEG;

.SEGMENT /PM rst_svc;
initial_setup:
        jump begin (db);
         pmwait=0x0021;         /*Bank 0 and 1, internal wait states only, 0
wait states*/
         dmwait=0x8421;         /*Banks 0-3, internal wait states only, 0 wait
states*/
```

```
.ENDSEG;

.SEGMENT /PM pm_code;
begin:  l0=TAPS;                /*delay line buffer pointer initialization*/
        b0=dline;
        m0=1;
        l1=0;                   /*input buffer pointer initalization*/
        b1=inbuf;
        l2=0;                   /*output buffer pointer initalization*/
        b2=outbuf;
        l8=TAPS;                /*coefficent buffer pointer initalization*/
        b8=coefs;
        call fir_init (db);     /*initalize delay line buffer to zero */
        m8=1;
        r0=TAPS;
        lcntr=SAMPLES, do filtering until lce;
            call fir (db);      /*input sample passed in F0, output returned in
F0*/
             r1=TAPS-1;
             f0=dm(i1,1);
filtering:   dm(i2,1)=f0;       /*result is stored in outbuf*/
done:   jump done;


fir_init:
    lcntr=r0, do zero until lce;
zero:   dm(i0,m0)=0;            /*initalize the delay line to 0*/
    rts;

.ENDSEG;
```

4 FIR & IIR Filters

Listing 4.1 firtest.asm

4.1.2.2 Filter Code

```
/************************************************************************

File Name
        FIR.ASM

Version
        0.03    11/2/93

Purpose
        This is a subroutine that implements FIR filter code given
        coefficents and samples.

Equation Implemented
        y(n)=Summation from k= 0 to M of H sub k times x(n-k)

Calling Parameters
        f0 = input sample x(n)
        r1 = number of taps in the filter minus 1
        b0 = address of the delay line buffer
        m0 = modify value for the delay line buffer
        l0 = length of the delay line buffer
        b8 = address of the coefficent buffer
        m8 = modify value for the coefficent buffer
        l8 = length of the coefficent buffer

Return Values
        f0 = output sample y(n)

Registers Affected
        f0, f4, f8, f12
        i0, i8

Cycle Count
        6 + (Number of taps -1) + 2 cache misses

# PM Locations
        7 instruction words
        Number of taps locations for coefficents

# DM Locations
        Number of taps of samples in the delay line

************************************************************************/
```

```
.GLOBAL fir;
.EXTERN coefs, dline;
.SEGMENT /PM pm_code;


fir:    r12=r12 xor r12,      dm(i0,m0)=f0;
                        /*set r12 =0 and store input sample in dline*/
        r8=r8 xor r8,         f0=dm(i0,m0), f4=pm(i8,m8);
                        /*set r8=0 and grab data from dline and coef*/
        lcntr=r1, do macs until lce;
                        /*set loop to iterate taps-1 times*/
macs:   f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                        /*perform mult. accumlate and fetch data*/
        rts (db);
        f12=f0*f4, f8=f8+f12;
                        /*perform mult on last pieces of data and 2nd to last
add*/
                    f0=f8+f12;
                        /*perform last add and store result in f0*/

.ENDSEG;
```

4 FIR & IIR Filters

Listing 4.2 fir.asm

4.2 IIR FILTERS

Because IIR digital filters correspond directly to analog filters, one way to design an IIR filter is to create a desired transfer function in the analog domain and then transform it to the *z* domain. Then the coefficients of a direct form IIR filter can be calculated from the *z* domain equation. The following equation is the direct form of the biquad difference equation of an IIR filter:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + a_1y(n-1) + a_2y(n-2)$$

The direct form is not commonly used in IIR filter design. Another form, called the *canonical form*, (also called *direct form II*) uses half as many delay stages, and therefore uses less memory. By assuming a linear time invariant system, the direct form can be mathematically manipulated to get the canonical form. The equation for the canonical form is

$$w(n) = x(n) + b_1 * w(n\text{-}1) + b_2 * w(n-2)$$
$$y(n) = w(n) + a_1 * w(n\text{-}1) + a_2 * w(n-2)$$

The input sample is $x(n)$ and $y(n)$ is the output sample. The term $w(n)$ is called the intermediate value; $w(n-1)$ is the previous value and $w(n-2)$ the one before that. The variables a and b are the coefficients for the filter.

The example in this section uses a variant of the canonical form equation that limits the summation in the canonical form to two. This type of IIR filter is called a *biquad* and is a second order filter. Higher order filters can be constructed by cascading several biquad filters together.

IIR filters have some advantages over FIR filters:

- IIR filters require less memory and fewer instructions to implement a specified transfer function than FIR filters.
- IIR filters are made up of poles and zeros in the complex plane. The poles give IIR filters an ability to realize transfer functions that FIR filters cannot.
- IIR filters translate well into analytical models. An example of this from control theory is the proportional integral differential (PID) controller equation. One implementation of this algorithm uses an

equation that is an implementation of an IIR filter.

The following tradeoffs are the costs of improved performance:

- IIR filters are not necessarily stable—it is the designer's task to ensure stability.
- Processor overflow must be considered. IIR filters are implemented with a sum of products operation that is based on an infinite sum. This construct can produce results that exceeds the maximum value that can be represented by the processor.

4.2.1 Implementation

This section of the chapter describes the assembly code for the biquad IIR filter. The IIR filter code executes a direct form II realization structure.

4.2.1.1 Implementation Overview

Five files are used to assemble the IIR filter; two assembly language modules, the architecture file, the input data file, and the coefficients file. With these five files, you can run the IIR filter code in the ADSP-21000 family simulator and plot the impulse response of the filter.

- The assembly language module `CASCADE.ASM` contains the assembly code that implements a direct form II IIR biquad filter. This code is common to all biquad IIR filters and can be included as part of a library of general DSP functions.
- The assembly language module `IIRMEM.ASM` initializes the input and coefficient buffers and then calls the `CASCADE.ASM` code.
- The architecture file is `GENERIC.ACH`, the same file used in the FIR filter code.
- The file `INPUT.DAT` contains data used to initialize the input buffer for the IIR filter code. This file consists of 300 samples that represent an impulse.
- The coefficients file is called `IIRCOEFS.DAT` and contains filter coefficients generated by a filter design software package. (The filter design software used for this example is FDAS, by Momentum Data Systems.) The coefficients represent a sixth-order elliptical bandpass filter that has a passband from 400 to 500 Hertz with cutoff frequencies of 300 and 600 Hertz. The following listing is an FDAS file that

contains the complete specifications for the FIR filter:

```
FILTER COEFFICIENT FILE
IIR DESIGN
FILTER TYPE                 BAND PASS
ANALOG FILTER TYPE          ELLIPTIC
PASSBAND RIPPLE IN -dB            -.1000
STOPBAND RIPPLE IN -dB           -1.0000
PASSBAND CUTOFF FREQUENCIES  .400000E+03  .500000E+03 HERTZ
STOPBAND CUTOFF FREQUENCIES  .300000E+03  .600000E+03 HERTZ
SAMPLING FREQUENCY           .800000E+04 HERTZ
FILTER DESIGN METHOD: BILINEAR TRANSFORMATION
FILTER ORDER            6 0006h
NUMBER OF SECTIONS      3 0003h
NO. OF QUANTIZED BITS  32 0020h
QUANTIZATION TYPE - FLOATING POINT
COEFFICIENTS SCALED FOR FLOATING POINT IMPLEMENTATION
  .67730926E-02       /* overall gain                    */
  .00000000           /* section  1 coefficient B1       */
 -1.0000000           /* section  1 coefficient B2       */
  1.8039191           /* section  1 coefficient A1       */
 -.92128010           /* section  1 coefficient A2       */
 -1.7640328           /* section  2 coefficient B1       */
  1.0000000           /* section  2 coefficient B2       */
  1.8060702           /* section  2 coefficient A1       */
 -.96266572           /* section  2 coefficient A2       */
 -1.9376569           /* section  3 coefficient B1       */
  1.0000000           /* section  3 coefficient B2       */
  1.8791107           /* section  3 coefficient A1       */
 -.97108089           /* section  3 coefficient A2       */
```

Listing 4.3 Filter Specifications From FDAS

4.2.1.2 Implementation Details

The module `IIRMEM.ASM` starts by declaring and initializing the buffers used by the filter.

The first buffer declared, `INBUF`, is 300 locations long. This buffer is initialized with the data file `INPUT.DAT`, which contains input samples for the filter. These input samples represent an impulse response so that the impulse response of the filter can be examined.

The second buffer declared, `OUTBUF` , is also 300 locations long, and stores

the output samples of the filter.

`DLINE` is the third buffer declared. This buffer holds the delay line that is the w(n-1) and w(n-2) for each biquad stage of the filter. Since there are two past intermediate values for each stage of a biquad, the length of this buffer is twice as long as the number of stages. The delay line buffer is ordered

1st stage $w(n-2)$, 1st stage $w(n-1)$, 2nd stage $w(n-2)$, 2nd stage $w(n-1)$…

The coefficients buffer, which is the next buffer initialized in the code, must be ordered so that it matches the order of the delay line buffer. The order is

1st stage a_2, 1st stage a_1, 1st stage b_2, 1st stage b_1, 2nd stage a_2…

This ordering is used because it is assumed that when the filter sequentially accesses values from the delay line and the coefficient buffer, these values are the ones that should be multiplied together. The coefficient buffer is the only buffer that is stored in program memory; all the others are stored in data memory.

After the buffers are declared, the code in `IIRMEM.ASM` assigns pointers to each data buffer. When assigning pointers to the buffers, a base register (B) and a length (L) register in the data address generators must be defined. When the base register is loaded with a value, the corresponding index register (I) is automatically loaded with a value. Following is a list of index register pointers and their corresponding buffers:

I0—> delay line buffer
I1—> delay line buffer, used to update old intermediate values
I3—> input buffer
I4—> output buffer

I8—> coefficient buffer

The length (L register) of `DLINE` is six and `COEFFS` is twelve; the other buffer lengths are set to zero to indicate that they are not circular. After a buffer is traversed, the index register is specifically written with the base address by an instruction.

Before the filtering starts, the subroutine `CASCADED_BIQUAD_INIT` is called to fill the delay line buffer with zeros. This buffer has to be zeroed out because it is accessed for additions the first time through the `QUADS` loop (in `CASCADE.ASM`). If the buffer was not initialized to zero, the first additions in the filter would be incorrect. The `CASCADED_BIQUAD_INIT` loop is iterated as many times as the number of biquad sections in the filter. Each time the loop is iterated two zeros are written to the buffer. This corresponds to writing zeros to the two intermediate values ($w(n-2)$ and $w(n-1)$) that are stored for each stage.

`CASCADED_BIQUAD` is called by the loop labeled `FILTERING`, which is executed once for each input sample. In this case it is executed 300 times—the length of the input buffer. The input sample is loaded in the F8 register. The pointers to the delay line and to the coefficients buffer are set to the beginning of the buffers. After the `CASCADED_BIQUAD` subroutine returns, the output value of the filter is in F8 and is written to the output buffer.

The file `CASCADE.ASM` contains `CASCADED_BIQUAD`, the actual IIR filtering subroutine. `CASCADED_BIQUAD` expects an input value to be passed in F8 and pointers to be set up as mentioned above. This code uses the `b1=b0` instruction to initialize the I1 pointer, which is the pointer to the delay line that updates the old intermediate values. The subroutine also expects the delay line and the coefficients to be ordered as specified above.

The filtering starts with the first multifunction line

```
f12=f12-f12, f2=dm(i0,m1), f4=pm(i8,m8)
```

This line sets F12 to zero (by performing the subtraction) and fetches a delay line value and a coefficient value. These two values represent $w(n-2)$ and a_2 for the first biquad stage. This instruction is executed outside of the `QUADS` loop so that data is available to be multiplied in the first instruction of the loop. Each iteration of the `QUADS` loop represents one biquad stage of the filter. If there are three biquad stages of the filter, this loop is repeated three times. The register R0 (which was assigned a value equal to the number of stages of the filter in the `IIRMEM.ASM` module) is

used to set the number of times that the loop will be repeated.

The computational registers used by the loop are F2, F3, F4, F8, and F12. Following is a list of what these registers hold for each stage:

Reg.	*Contents*	*Comment*
F2	$w(n-2)$	
F3	$w(n-1)$	
F4	a_2, a_1, b_2, b_1	*held in this order*
F8	sum of multiplications	*output of stage at end*
F12	$a_2 * w(n-2)$, $a_1 * w(n-1)$, $b_2 * w(n-2)$, $b_1 * w(n-1)$	*held in this order*

The `QUADS` loop is an implementation of the IIR filter direct form II difference equation previously given . This equation is implemented by first fetching two pieces of data outside the loop. Then in each cycle of the loop a multiplication of the data fetched in the previous cycle, a running total of the summation of the multiplications, and a fetch of two more pieces of data is executed. After the instruction at the label `QUADS` is executed, one of two things can happen:

- The next biquad stage must be calculated and the loop executed again. The addition in the first line of the loop calculates the final result of the output of the previous stage.

- The biquad stages of filter executes and the addition (on the line with

the RTS) calculates the final result of the filter.

4.2.2 Code Listings

4.2.2.1 iirmem.asm

```
/************************************************************************

File Name
   IIRMEM.ASM

Version
   created      4/25/91
   modified     12/1/93

Purpose
   This program declares and initializes input, delay line, and output
   buffers that are used in IIR filter calculations.  The program then
   calls the module CASCADE.ASM which calculates the IIR filter outputs.
   The coefficients for the IIR filter are read from the file IIRCOEFS.DAT
   which represent a sixth order bandpass elliptical filter.

Equation Implemented

Calling Parameters

Return Values

Registers Affected

Cycle Count

# PM Locations

# DM Locations
```

```
*************************************************************************/

.EXTERN cascaded_biquad;        /*Used to reference a lable in another module*/
.PRECISION=40;                  /*40 bit mode is uses to achieve greater
 precision*/
.ROUND_NEAREST;               /*Rounding mode for constants when they don't fit
 into*/
                              /*destination format*/

#define SAMPLES 300             /*Number of samples in input buffer*/
#define SECTIONS 3              /*Number of biquad sections in filter.  In this
 case there*/
                        /*are three sections which make up sixth order filter*/

.SEGMENT /DM    dm_data;
.VAR    inbuf[SAMPLES] = "input.dat";            /*Input buffer initalized with
*/
                                              /* impulse made up of 300 samples*/
.VAR    outbuf[SAMPLES];                         /*Output buffer declared,
holds*/
                                              /* impulse response of filter */
                                              /*after execution  */
.VAR    dline[SECTIONS*2];                    /*Delay line buffer declared */
.ENDSEG;

.SEGMENT /PM    pm_data;
.VAR    coefs[SECTIONS*4]="iircoefs.dat";  /*Buffer of coefficents declared and
*/
                                              /* initalized*/
.ENDSEG;
.SEGMENT /PM    rst_svc;    /*Reset vector*/
   jump begin(db);         /*Jump to begin with a delayed branch*/
   pmwait=0x000021;        /*PM internal wait states only, 0 wait states*/
   dmwait=0x008421;        /*DM internal wait states only, 0 wait states*/
.ENDSEG;

.SEGMENT /PM    pm_code;
begin:  b3=inbuf;                             /*I3 points to input buffer*/

   l3=0;
   b4=outbuf;                                 /*I4 points to output buffer*/
   l4=0;
   l1=0;                                      /* Updates new values in delay
line,*/
                                              /*  b1 is defined in CASCADE.ASM   */
   m1=1;
   l8=12;                                     /*Length for coefficents, B8 is
defined
 in filtering loop*/
   m8=1;
   b0=dline;                                  /*I0 points to delay line buffer*/
   call cascaded_biquad_init (db);            /*Zero the delay line */
    l0=6;
    r0=SECTIONS;                              /* r0=number of times through
```

(listing continues on next page)

```
                                                /* delay line init. loop*/
   lcntr=SAMPLES, do filtering until lce;       /* Execute filtering loop for
                                                /* all 300 samples*/
       f8=dm(i3,1);                             /*Get a sample from the input
buffer*/
       call cascaded_biquad (db);               /*Input=F8, Output=F8 */
        b0=dline;                                      /*Set i0 to begining of delay
line*/
        b8=coefs;                                      /*Set i8 to begining of
coefficents*/
filtering:  dm(i4,1)=f8;                               /* Store filtered value in
                                                /* output buffer*/
done:   idle;


cascaded_biquad_init:               /*Called once to initalize the delay line*/
    f2=0;
    lcntr=r0, do clear until lce;   /* Each section contains a w(n-1) and*/
                                    /* a w(n-2) in the delay line*/
    dm(i0,1)=f2;                    /* Set w(n-2) equal to 0*/
clear:  dm(i0,1)=f2;                /* Set w(n-1) equal to 0*/
    rts;

.ENDSEG;
```

Listing 4.4 iirmem.asm

4.2.2.2 cascade.asm

```
/************************************************************************

File Name
        CASCADE.ASM

Version
        created         4/25/91
        modified        12/1/93

Purpose
        This module implements a biquad, canonical form, IIR filter.  It needs
        to passed an input sample, a coefficent buffer, and a delay line
```

```
        buffer. The coefficent buffer should be in the order a2, a1, b2, b1
        for each biquad section section. The delay line buffer should be ordered
        w(n-2), w(n-1) for each biquad section.  See the next section, equations
        implemented, to see what the aformentioned variables mean.


Equation Implemented
        w(n) = x(n) + a1*w(n-1) + a2*w(n-2)
        y(n) = w(n) + b1*w(n-1) + b2*w(n-2)

        Note: 1. The usual form for the w(n) equation is

                     w(n)=x(n)-a1*w(n-1)-a2*w(n-2)

                 Since the code implements the w(n) equation with the additions
                 the coefficents for the filter have to take into account the
                 minus sign.

              2. The equation that calculates y(n) has a factor associated with
                 the w(n) term.
                 This factor is 1 in this case because b1 and b2 are normalized
                 to the factor.
                 By normalizing the filter coefficents ahead of time a
                 multiplication is saved in the execution of the filter.

Calling Parameters
        f8 = input sample x(n)
        r0 = number of biquad sections
        b0 = address of DELAY LINE BUFFER
        b8 = address of COEFFICENT BUFFER
        m1 = 1, modify value for delay line buffer
        m8 = 1, modify value for coefficent buffer
        l0 = 0
        l1 = 0
        l8 = 0

Return Values
        f8 = output sample y(n)

Registers Affected
        f2, f3, f4, f8, f12
        i0, b1, i1, i8

Cycle Count
        6 + 4*(number of biquad sections) + 5 cache misses
```

(listing continues on next page)

```
# PM Locations
        10 instruction words
        4 * (number of biquad sections) locations for coefficents

# DM Locations
        2 * (number of biquad sections) locations for the delay line

************************************************************************/


.GLOBAL cascaded_biquad;
.SEGMENT/PM     pm_code;
cascaded_biquad:
   /*Call this for every sample to be filtered*/
b1=b0;
   /*I1 used to update delay line with new values*/
f12=f12-f12, f2=dm(i0,m1), f4=pm(i8,m8);
   /*set f12=0, get a2 coefficent, get w(n-2)*/
lcntr=r0, do quads until lce;
   /*execute quads loop once for each biquad section */
   f12=f2*f4, f8=f8+f12, f3=dm(i0,m1), f4=pm(i8,m8);
   /* a2*w(n-2), x(n)+0 or y(n) for a section, get w(n-1), get a1*/

f12=f3*f4, f8=f8+f12, dm(i1,m1)=f3, f4=pm(i8,m8);
   /*a1*w(n-1), x(n)+[a2*w(n-2)], store new w(n-2), get b2*/

   f12=f2*f4, f8=f8+f12, f2=dm(i0,m1), f4=pm(i8,m8);
   /*b2*w(n-2), new w(n), get w(n-2) for next section, get b1*/
quads:
f12=f3*f4, f8=f8+f12, dm(i1,m1)=f8, f4=pm(i8,m8);
        /*b1*w(n-1), w(n)+[b2*w(n-1)], store new w(n-1), get a2 for next
```

```
section*/

rts (db),f8=f8+f12;
    /*return, compute y(n)*/
      nop;
      nop;
.ENDSEG;
```

Listing 4.5 cascade.asm

4.3 SUMMARY

FIR and IIR filters each have their own characteristics and advantages that makes one or the other suitable for a particular application. If linear phase is a critical factor, then an FIR filter is best. If a design has tight memory and instruction constraints, then choose an IIR filter.

4.4 REFERENCES

[ADI90] Analog Devices, Inc, Amy Mar, ed.. 1990. *Digital Signal Processing Applications Using the ADSP-2100 Family.* Englewood Cliffs, NJ: Prentice Hall.

Multirate Filters 5

Multirate filters change the sampling rate of a signal—they convert the input samples of a signal to a different set of data that represents the same signal sampled at a different rate. Some examples of applications of multirate filters and systems are:

- Sample-rate conversion between digital audio systems
- Narrow-band and band-pass filters
- Sub-band coding for speech processing in vocoders
- Quadrature modulation

Decimation and *interpolation* are the two basic types of multirate filtering processes.

- Decimation is the process of reducing the sample rate of the signal. A decimation filter removes redundant information from the original sampled signal thereby compacting the data.

- Interpolation is the process of increasing the sample rate of the signal. An interpolation filter inserts missing data between the existing samples.

Multirate systems, such as sample-rate conversions, are combinations of decimation and interpolation filters.

The six multirate filters described in this chapter are:

- single-stage decimation filter
- single-stage interpolation filter
- rational rate changer (timer based)
- rational rate changer (interrupt based)
- two-stage decimation filter
- two-stage interpolation filter

5 Multirate Filters

The descriptions of the implementation of these filters for the ADSP-21000 family assume that you already understand the theory behind them. For more details on the theory of operation of multirate filters, refer to the Multirate Filter section in [ADI90].

5.1 SINGLE-STAGE DECIMATION FILTER

A single-stage decimation filter simply resamples the digital signal with anti-aliasing. The code `decimate.asm` (Listing 5.1) uses an *N*-tap Direct Form Finite Impulse Response Filter, and decimates the signal data in real time by a factor of *M*. This process yields a decrease in the input sample rate of $1/M$.

The input signal to the decimator is a signal that is oversampled by *M*; that is, it is sampled at M times the desired output frequency. The decimation filter algorithm consists of two operations. First, a lowpass FIR filter bandlimits the input signal to one-half of the output frequency to prevent aliasing. Then an decimation removes $M - 1$ samples of every *M* samples to create an output signal that is $1/M$ times the input sample rate. Both of the operations are performed in a single FIR routine.

Data acquisition systems take advantage of decimation filters to avoid using expensive, high-performance analog anti-aliasing filters. A system oversamples the input signal by a factor of *M* and then decimates to the desired sample rate. By oversampling the input signal, the transition band of the front end filter to prevent aliasing, thus an inexpensive analog filter can be used. The decimation filter will then reduce the input single to the required sample rate.

5.1.1 Implementation

The `decimate.asm` code starts by initializing the data buffers (`input_buf`, `data`, and `coef`) and the respective DAG registers. The input buffer, `input_buf`, is *M* long where *M* is the decimation factor. The data buffer, `data`, and filter coefficient buffer, `coef`, are both *N* locations long, where *N* is the number of taps for the filter. The I7 and L7 registers are specifically used for the `input_buf` so that the circular buffer overflow interrupt can be used. The timer period is also programmed to the desired input frequency. Since the timer interrupt happens during the circular buffer overflow service routine, interrupt nesting is enabled.

The timer interrupt service routine (`tmzh_svc`) gets the input sample from an A/D converter port (`adc`) and stores the samples in the input buffer `input_buf` . When *M* samples have been acquired, the input buffer is full and a circular buffer overflow interrupt is generated.

The circular buffer overflow interrupt service routine (`cb7_svc`) calls the decimator routine (`decimate`) only every *M* times, not at every sample. This removes *M–1* samples in between decimator calls.

The decimate routine transfers the input buffer to the data buffer as input for the decimation filter, converting from the sampled 16-bit fixed point number to an equivalent 32-bit floating point number. This double buffering allows the filter computations to proceed in parallel with the acquisition of the next M inputs, allowing a larger order filter than if all the filter calculations are made between input sample periods.

To save cycles, the data transfer is performed in a rolled loop to allow the delayed branch (db) call. The fix-to-float and scaling operations are performed in parallel, also to save cycles. If the decimation factor is less than four, the data conversion and transfer is more efficient if performed unrolled.

After the `input_buf` buffer is transferred to the data buffer, the decimate routine runs the FIR filter on the sampled data. The FIR filter is also a rolled loop where the first three data acquisitions are performed outside the loop. Therefore, the loop counter is set to $N - 3$, where N is the number of taps in the filter. The output of the FIR is then converted back to a 16-bit fixed point format and written to the output D/A converter (`dac`) . The STKY bit for the circular buffer overflow is also cleared to prevent reentry to the interrupt service routine.

Since the acquisition of the input data and calculation of the FIR filter are in parallel, the decimator must calculate one pass of the FIR filter while *M* samples are being read into the input buffer, `input_buf` . The following equation determines the value of the timer period:

*Minimum Timer Period = max(ceil[(N + 2*M + 11 + 5*M)/M], 20)*

The FIR filter is invoked by `cb7_svc` , which requires (N + M * 2 + 11) cycles to execute. To obtain M samples requires an additional (5 * M) cycles. Dividing by M yields the timer period, but we must take the greatest integer value larger than this number (ceiling, or ceil function). Additionally, the input buffer must be completely moved to the data

buffer at the start of the `cb7_svc` routine before the first timer interrupt occurs again. Because the entire decimate routine (19 cycles) must be executed between each sampling, there won't be enough time for this if the timer period is not at least 20.

Here is an example calculation for a 20 MHz ADSP-210xx processor:

FP = processor frequency = 33.3
N = number of taps = 64
M = decimation factor = 8
Minimum Timer Period =max(ceil[(64+2*8+11+5*8)/8], 20)
= max(ceil[(131/8)], 20)
= max(ceil[(131/8)], 20)
= max(17, 20)
= 20

Maximum Input Frequency = FP/20 = 33.3MHz/20 = 1.1.665 MHz
Maximum Output Frequency = FP/(M*20) = 33.3mHz/(8*20) = 208 kHz

If N=128, and M=8, 25 cycles is the minimum timer period.

5.1.2 Code Listings–decimate.asm

```
/************************************************************************
File Name
   DECIMATE.ASM

Version
   3/4/91

Purpose
   Real Time Decimator

Calling Parameters
   Input:  adc
   r15 = ADC input data

Return Values
   Output: dac
   r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC fltg-pt data
   r4 = FIR coefficients / temporary reg
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scaling value

Start Labels:
   init_dec        reset-time initialization
   decimate        called by CB7 interrupt

Computation Time:
   cb7_svc = 6 + decimate = N + M*2 + 11
   decimate = N + M*2 + 5 + [5 misses]
   tmzh_svc = 5

Minimum Timer Period:       max( ceil[(N+2*M+11+5*M)/M], 20 )

# PM Locations
   47 Words Code, N Words Data

# DM Locations
   N + M Words Data
************************************************************************/
```

(listing continues on next page)

5 Multirate Filters

Creating & Using the Test Case:

Running the test case requires two data files:

coef.dat	32-tap FIR filter coefficients (floating point)
wave.1	waveform data file

coef.dat contains a 32-tap FIR filter which bandlimits the input signal to 1/8 of the input frequency. Since the decimator in the test case decimates by a factor M=4, this bandlimitation is equivalent to the required limit of 1/2 the output frequency. The filter is a Parks-McClellan filter with a passband ripple of 1.5dB out to about 1/10 the input frequency, and a stopband with greater than 70dB of attenuation above 1/8 the input frequency.

As an example, if the oversampled input frequency is 64kHz, the passband extends out to about 6.4kHz. The stopband starts at 8kHz, and the output frequency is 16kHz.

The example data file is wave.1. It contains 512 signed fixed-point data which emulate a 16-bit A/D converter whose bits are read from Data Memory bits 39:24. wave.1 contains a composite waveform generated from 3 sine waves of differing frequency and magnitude. The cb7_svc routine arithmetically shifts this data to the right by 16 bits, effectively zero-ing out the garbage data which would be found in bits 23:0 if a real 16-bit ADC port were read.

The test case writes the decimated output to a dac port. Since there are 512 samples in wave.1, and the test case performs decimation by 4, there will be 128 data values written to the dac port if all data samples are read in and processed. The values written out are left-shifted by 16 bits in parallel with the float—>fixed conversion, based again on the assumption that the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

```
asm21k -DTEST decimate
ld21k -a generic decimate
sim21k -a generic -e decimate
```

2. In the simulator,
   a. Go to the program memory (disassembled) window,
   d. Go to symbol cb7_done, set to break on 129th occurrence, and
   c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

```
#include "def21020.h"
#define N 32                /*  number of taps                          */
#define M 4                 /*  decimation factor                   */
#define FP 20.0e6           /*  Processor Frequency = 20MHz             */
#define FI 64.0e3           /*  Interrupt Frequency = 64KHz             */

#ifndef TEST /*————————————————————————*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1                    */
#else /*———————————————————————————————*/
#define TPER_VAL 19         /* interrupt every 20 cycles            */
#endif /*——————————————————————————————*/

.SEGMENT /dm dm_data;
.VAR    data[N];            /* FIR input data                           */
.VAR    input_buf[M];       /* raw A/D Converter data                   */
.ENDSEG;

.SEGMENT /pm pm_data;
.VAR    coef[N]="coef.dat"; /* FIR floating-point coefficients      */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter      */
.PORT   dac;                /* Digital to Analog (output) converter     */
.ENDSEG;

/*————————————————————————————————
    RESET Service Routine
————————————————————————————————*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only     */
                DMWAIT=0x8421;      /* no wait states,internal ack only     */
                jump init_dec;      /* initialize the test case         */
.ENDSEG;

/*————————————————————————————————
    TMZH Service Routine
————————————————————————————————*/
.SEGMENT /pm    tmzh_svc;
/*  sample input: this code occurs at the sample rate  */
tmzh_svc:       rti(db);
                    r15=dm(adc);                /* get input sample         */
                    dm(i7,m0)=r15;              /* load in M long buffer    */
.ENDSEG;
```

(listing continues on next page)

5 Multirate Filters

```
/*_____________________________________
    DAG1 Circular Buffer 7 Overflow Service Routine
_____________________________________*/
.SEGMENT /pm    cb7_svc;
/*  process input data when circular buffer 7 wraps around (“overflows”)     */
/*       this code occurs at 1/M times the sample rate                        */
cb7_svc:        call decimate (db);             /* call the decimator        */
                    r12=dm(i7,m0);
                    r4=ashift r12 by -16;
                rti (db);                       /* return from interrupt     */
                    r0 = fix f0 by r14;         /* convert to fixed point    */
cb7_done:           dm(dac)=r0;                 /* output data sample        */
.ENDSEG;

/*_____________________________________
    efficient decimator initialization
_____________________________________*/
.SEGMENT /pm    pm_code;
init_dec:       b0=data; l0=@data; m0=1;        /* data buffer               */
                b7=input_buf; l7=@input_buf;    /* input buffer              */
                b8=coef; l8=@coef; m8=1;        /* coefficient buffer        */

                tperiod=TPER_VAL;               /* program timer             */
                tcount=TPER_VAL;

                /* enable Timer high priority, CB7 interrupts */
                bit set imask TMZHI|CB7I;
                /* clear interrupts before setting global enable */
                bit clr irptl TMZHI|CB7I;

                r14=16;                         /* scale factor float->fixed */
                r15=0;                          /* clear data buffer         */
                lcntr=N, do clrbuf until lce;
clrbuf:             dm(i0,m0)=r15;

                /* enable interrupts, nesting mode, ALU saturation */
                bit set mode1 IRPTEN|NESTM|ALUSAT;
                bit set mode2 TIMEN;            /* enable the timer          */

wait:           idle;
                jump wait;                      /* infinite wait loop        */

.ENDSEG;
```

```
/*_____________________________________
    efficient decimator routine
    executes at 1/M times the sample rate
_____________________________________*/
.SEGMENT /pm    pm_code;
decimate:       /*   Transfer input samples to the FIR data memory space     */
                /* using the input buffer working pointer. Perform           */
                /* fix—>float conversion and right-shifting in parallel.     */
                /*   Note: this loop MUST be unrolled if M<3. It would be    */
                /* more efficient if unrolled for any M<4. The 1st two       */
                /* instructions of this loop have been executed in the delay */
                /* field of the call to this subroutine                      */
                f0=float r4, r12=dm(i7,m0);
                lcntr=M-2, do load_data until lce;
                    r4=ashift r12 by -16, dm(i0,m0)=f0;
load_data:          f0=float r4, r12=dm(i7,m0);
                r4=ashift r12 by -16, dm(i0,m0)=f0;
                f0=float r4;
                dm(i0,m0)=f0;

                /* N-tap FIR filter */
fir:                      f0=dm(i0,m0), f4=pm(i8,m8);
                f8=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                lcntr=N-3, do taps until lce;
taps:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);

                rts(db), f12=f0*f4, f8=f8+f12;
                    f0=f8+f12;                  /* final sum in || w/ rts    */
                    bit clr stky CB7S;          /* clear sticky bit to       */
                                                /*  prevent re-interrupt     */

.ENDSEG;
```

Listing 5.1 decimate.asm

5 Multirate Filters

5.2 SINGLE-STAGE INTERPOLATION FILTER

A single-stage interpolation filter reconstructs a discrete-time signal from another discrete-time signal. This is analogous to the decimator where a sampled signal is sampled again at a different rate. By inserting more data points between the originally sampled data points, interpolation increases the output sample rate. Interpolation performs the function of a digital anti-imaging filter on the original signal. Many digital audio systems, such as compact disc and digital audio tape players, use interpolation (oversampling) to avoid using high-performance analog reconstruction filters. The anti-imaging functions in interpolators allow the use of low-order analog filters on the D/A outputs.

To increase the input sampling frequency by a factor of L, L – 1 zeroes are inserted between the original samples and then low-pass filtered. The insertion of zeros between samples and the smoothing filter fill in the data missing between the original samples. The filter must bandlimit the output signal to one-half the output frequency to prevent imaging. Since one input sample now corresponds to *L* output samples, the sample rate is increased by *L*.

5.2.1 Implementation

The `interpol.asm` code (Listing 5.2) uses an *N*-tap Direct Form Finite Impulse Response Filter in an efficient algorithm to interpolate by *L*=4. This increases the input sample rate by a factor of four. The number of taps is restricted such that N/L must be an integer. An undesirable by-product of the FIR filter is the attenuation of the input signal by a factor of *L*. To correct this, the input data is scaled up by *L* before it is filtered.

Because most of the delay line inputs are zeros, calculation time would be wasted if the zero inputs were multiplied by their corresponding coefficients. Instead, the coefficients are interleaved to perform only the non-trivial calculations. This technique is known as Poly-phase filtering. Thus, the delay line must be only N/L locations long. In this example N=32 and L=4, so the delay line, `data`, is eight location long. The coefficient buffer, `coef`, is 32 locations long. The example initializes the delay line address modifier to one and the coefficient buffer address modifier to *L* = 4, for the proper interleave.

The internal timer generates interrupts at l times the output sample rate, which is the desired output frequency. The register R2 is used as a counter in the timer service routine (`tmzh_svc`) to get a new sample from the A/D converter (`sample`) every *Lth* pass of the interrupt. When a new

sample is read in, it is converted from a 16-bit fixed point value to a 32-bit floating point value and scaled up by *L* to compensate for the attenuation effect. The routine also resets the counter to *L* and modifies the coefficient pointer by *L* for proper interleave.

The timer service routine calls the interpolator (`interpol`). The interpolator runs N/L taps of the FIR filter at the output frequency rate. To achieve the proper interleaving of coefficients with input data, the coefficient pointer is modified by -1 before the interpolation takes place. During the interpolation filter, the coefficient pointer is then modified by L after every access. This interleaving process is equivalent to running an *N*-tap FIR filter on data generated by adding *L– 1* zero samples to each data input.

The filter output is converted back to a 16-bit fixed point value and written to a D/A converter (`dac`) after each interpolation. The technique of rolled loops and delayed branches are used to conserve cycles.

5.2.2 Code Listing–interpol.asm

```
/************************************************************************
File Name
   INTERPOL.ASM

Version
   3/6/91

Purpose
   Real time Interpolator

Calling Parameters
   Input:  adc
   r15 = ADC input data

Return Values
   Output: dac
   r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC data / temp register
   r1 = scale factor L
   r2 = counter
   r4 = FIR coefficients
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels
   init_int    reset-time initialization
   interpol    called by Timer Zero High Priority interrupt

Computation Time
   tmzh_svc = 5 + interpol
               = N/L + 9                  (no sample)
               = N/L + 17                 (with sample)

   interpol = N/L + 4                  (no sample)
               = N/L + 12 + [5 misses]    (with sample)

Minimum Timer Period:       N/L+20
Maximum Output Frequency:   FP/(N/L+20)
Maximum Input Frequency:    FP/(L*(N/L+20))

# PM Locations
   44 Words Code, N Words Data

# DM Locations
   N/L Words Data
************************************************************************/
```

```
_____________________________________

Creating & Using the Test Case:

Running the test case requires two data files:
    coef.dat        32-tap FIR filter coefficients (floating point)
    wave.1          waveform data file

coef.dat contains a 32-tap FIR filter which bandlimits the output signal to
1/8 of the output frequency.  Since the interpolator in the test case
interpolates by a factor M=4, this bandlimitation is equivalent to the
required limit of 1/2 the input frequency.  The filter is a Parks-McClellan
filter with a passband ripple of 1.5dB out to about 1/10 its input frequency,
and a stopband with greater than 70dB of attenuation above 1/8 the input
frequency.

As an example, if the interpolated output frequency is 64kHz (==input
frequency of 16kHz), the passband extends out to about 6.4kHz.  The stopband
starts at 8kHz, and the output frequency is 16kHz.

The example data file is wave.1.  It contains 512 signed fixed-point data
which emulate a 16-bit A/D converter whose bits are read from Data Memory bits
39:24.  wave.1 contains a composite waveform generated from 3 sine waves of
differing frequency and magnitude.  The cb7_svc routine arithmetically shifts
this data to the right by 16 bits, effectively zero-ing out the garbage data
which would be found in bits 23:0 if a real 16-bit ADC port were read.

The test case writes the interpolated output to a dac port.  Since there are
512 samples in wave.1, and the test case performs interpolation by 4, there
will be 2048 data values written to the dac port if all data samples are read
in and processed.  The values written out are left-shifted by 16 bits in
parallel with the float—>fixed conversion, based again on the assumption that
the D/A converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

        asm21k -DTEST interpol
        ld21k -a generic interpol
        sim21k -a generic -e interpol -k port1

    Note: the keystroke file port1.ksf opens two ports:
        1st port - DM, F0000000, Fixed-Pt., auto-wrap, wave.1
        2nd port - DM, F0000001, Fixed-Pt., no wrap,   out.1

2. In the simulator,
        a. Go to the program memory (disassembled) window,
        d. Go to symbol sample, set to break on 513th occurence, and
        c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

_____________________________________*/
```

(listing continues on next page)

5 Multirate Filters

```
#include “def21020.h”
#define N       32          /* number of taps                                  */
#define L       4           /* interpolate by factor of L                      */
#define FP 20.0e6           /* Processor Frequency = 20MHz                     */
#define FI 64.0e3           /* Interrupt Frequency = 64KHz                     */

#ifndef TEST /*———————————————————————*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1                           */
#else /*——————————————————————————————*/
#define TPER_VAL 27         /* interrupt every 28 cycles              */
#endif /*—————————————————————————————*/

.SEGMENT /dm    dm_data;
.VAR    data[N/L];              /* ADC fixed-point data buffer        */
.ENDSEG;

.SEGMENT /pm    pm_data;
.VAR    coef[N]=”coef.dat”;     /* fltg-pt. FIR coefficients                  */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter            */
.PORT   dac;                /* Digital to Analog (output) converter           */
.ENDSEG;

/*———————————————————————————————
    RESET Service Routine
———————————————————————————————*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only       */
                DMWAIT=0x8421;      /* no wait states,internal ack only       */
                jump init_int;      /* initialize the test case       */
.ENDSEG;

/*———————————————————————————————
    TMZH Service Routine
———————————————————————————————*/
.SEGMENT /pm tmzh_svc;
/*  sample input: this code occurs at the L*input rate  */
tmzh_svc:       r2=r2-1,modify(i8,m9);          /* decrement counter, and     */
                                                /*   shift coef pointer back */
                if eq call sample;              /* test and input if L times */

                jump interpol (db);             /* perform the interpolation */
                    /* filter pass, occurs at L times the input input rate   */
                    f0=dm(i0,m0), f4=pm(i8,m8);
                    f8=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);

.ENDSEG;
```

```
/*_____________________________________
    Initialize Interpolation Routine
_____________________________________*/
.SEGMENT /pm    pm_code;
init_int:       /*  initialize buffer index registers  */
                b0=data; l0=@data; m0=1;
                b8=coef; l8=@coef; m8=L;        /* modifier for coef is L    */
                m9=-1;

                /* Register Initialization */
                r1=L;                   /* value for upscaling sample */
                f1=float r1;            /* fix -> float conversion           */
                r2=1;                   /* set  counter to 1 for 1st sample  */
                r14=16;                 /* exponent scale factor             */

                tperiod=TPER_VAL;       /* program timer              */
                tcount=TPER_VAL;
                bit set imask TMZHI;    /* enable TMZH interrupt             */
                bit clr irptl TMZHI;    /* clear any pending interupts       */
                bit set mode1 IRPTEN|ALUSAT; /* enable interrupts, ALU sat   */
                bit set mode2 TIMEN;    /* turn timer on              */

                r0=0;                   /* clear data buffer                 */
                lcntr=N, do clrbuf until lce;
clrbuf:             dm(i0,m0)=r0;

wait:           idle;
                jump wait;              /* infinite wait loop                */
.ENDSEG;

/*_____________________________________
    sample & interpolate
    code executes at the (L*input_rate)
_____________________________________*/
.SEGMENT /pm    pm_code;
interpol:       /* FIR Filter, occurs at L times the input input rate.       */
                /*    First two instructions of this filter have been run    */
                /* in the delay field of the jump to this routine            */
                f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                lcntr=N/L-3, do taps until lce;
taps:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4, f8=f8+f12;
                f0=f8+f12;

                rti (db);
                    r0 = fix f0 by r14;         /* float -> fixed            */
                    dm(dac)=r0;                 /* output data sample */
.ENDSEG;
```

(listing continues on next page)

5 Multirate Filters

```
/*_____________________________________
    get sample
    code executes at the input_rate
_____________________________________*/
.SEGMENT /pm    pm_code;
/*  input data sample once every L times (i.e., at the input rate)          */
sample:         r0=dm(adc);                     /* input data sample        */
                /* input sample occuppies bits 39:24; shift to 15:0         */
                r0 = ashift r0 by -16;          /* shift w/ sign extend     */
                /* do fix->float, and shift coef pointer up by L            */
                f0=float r0, modify(i8,m8);
                rts(db), f0=f0*f1;              /* upscale sample by L      */
                    dm(i0,m0)=f0;               /* update delay line        */
                    r2=m8;                      /* reset counter to L       */
.ENDSEG;
```

Listing 5.2 interpol.asm

5.3 RATIONAL RATE CHANGER (TIMER-BASED)

This section describes a rational rate changer implemented by cascading an interpolating filter and a decimating filter. Interpolating filters and decimating filters can only change the rate of a signal by a factor that is an integer. This restriction makes these filters useless for many real world applications when used by themselves. By cascading these filters, the frequency may be changed by a factor that is a rational number.

For example, to transfer data between a compact disc with a sampling rate of 44.1 kHz to a digital audio tape player with a sampling rate of 48.0 kHz, the sampling rate of the compact disc must be increased by a factor of 48/44.1, which is obviously not an integer. By using a combination of an interpolation filter and decimation filter, a rational number can be determined that approximates the desired frequency change.

The interpolation must take place first so as not to lose some of the desired output frequency bandwidth that otherwise is filtered out by the decimator. The anti-imaging filter (which operates after interpolation) and anti-aliasing filter (which operates before decimation) can be combined into a single low-pass filter.

5.3.1 Implementation

This example, `ratiobuf.asm` (Listing 5.3), uses an N-tap (in this case, $N = 32$) real time direct FIR filter that performs interpolation by $L = 4$ and decimation by $M = 3$ for a $L/M = 4/3$ change in the input sample rate. The same techniques used in both interpolation and decimation filters are utilized to dedicate an entire output period for calculating one output sample. The ratio N/L is restricted to be an integer.

Like the interpolation algorithm, the ratio changer samples the incoming signal at L times the sampling rate using the timer interrupt, and only updates the input buffer every Lth pass. Also, every zero-valued multiplication is skipped by interleaving the coefficients. Like the decimator algorithm, the input is double-buffered so that the filter computations can occur in parallel with the input and output of data. This allows the use of a larger order filter for a given input sample rate.

This ratio changer implementation is counter based. The counter for the inputs, `IN_CNTR`, is set to L, and the counter for the outputs, `OUT_CNTR`, is set to M. If the `IN_CNTR` expires, a new input sample is read into the input buffer. If the `OUT_CNTR` expires, a flag is set to indicate that the main routine must calculate a new output sample.

The initialization routine sets the input buffer (`input_buf`) length to an integer value larger than M/L. For this example, the interpolation factor (L) is four and the decimation factor (M) is three. Therefore, the input buffer length is equal to one (the integer $\geq M/L$; $1 \geq 3/4$). This means that an output occurs at least once before the next input is received. The data buffer length is set to $N/L = 32/4 = 8$ just as in the single-stage interpolation. To properly interleave the coefficients for the filter performing both interpolation and decimation, the buffer pointer I9 is used to keep track of the coefficient updates. Both counters (`IN_CNTR` and `OUT_CNTR`) are initialized to one so that the first sample is read in and an output is generated.

The program then waits in an idle loop for the timer interrupt. Every timer interrupt, the counters are decremented and checked to see if either or both have expired:

- If `IN_CNTR` has not expired, the service routine jumps to `skipin` where it checks to see if the `OUT_CNTR` has expired.
- If `IN_CNTR` has expired, the program jumps to the routine `input` where the next value is read in from the A/D, scaled up by L, and stored in `input_buf` . The counter is reset and the `OUT_CNTR` is then checked to see if an output sample needs calculation.
- If the `OUT_CNTR` has not expired, the program returns to the idle statement to wait for the next timer interrupt.
- If the `OUT_CNTR` has expired, the User Software Interrupt 0 (bit 24 in the IRPTL register) is set to force a software interrupt. This interrupt service routine jumps to the routine `calc_out` .

The routine `calc_out` first determines the number of elements in `input_buf` :

- If the value is zero, a new A/D value was not input since the last time an output was calculated. The data transfer from `input_buf` to `data_buf` is skipped.
- If the `input_buf` is not empty, the stored samples are transferred to the `data_buffer` .

The `calc_out` routine then adjusts the pointer to the coefficients for the proper interleave, and executes the low-pass filter code to generate an output. The low-pass filter is an N/L tap filter implemented as a rolled loop, similar to both the single-stage interpolation and decimation filters.

At the end of the `calc_out` routine, the `input_buf` pointer and `coef` are reset to their original values. The final output is converted back to a 16-bit fixed-point value and sent to the DAC. The routine then returns to the idle loop to wait for the next timer interrupt.

There are several instances of unusual instruction ordering in the `calc_out` subroutine. This ordering is used in order to minimize total execution time, but requires explanation.

If an instruction modifies an I or M register and then the next instruction uses the same DAG, a single NOP cycle is inserted. As explained in the *ADSP-21020 User's Manual*, this dead cycle is inserted because the DAG needs to calculate indirect addresses for a given cycle during the previous cycle. To avoid this dead cycle, modifications of an I or M register are separated from use of a DAG where possible, even though the code ordering becomes more difficult to follow.

For example, the index register I9 is used to track coefficient updates, and must be modified in the `output` subroutine. These instructions are executed:

```
modify(i9,-M);
m10=i9;
modify(i8,m10);
i9=0;
```

A dead cycle occurs after the second instruction and possibly after the fourth instruction (if the fifth instruction uses DAG2). To avoid dead cycles, the first two instructions are moved away from the third, and the fourth instruction is moved to a place where a DAG2 operation does not follow it.

Additionally, there are two places in the `output` subroutine where `i7=b7` must be executed. Performing this ureg-to-ureg transfer by itself wastes cycles, so the transfer is combined with unrelated computations to form a multifunction instruction. The first `i7=b7` transfer must occur before the input buffer is transferred to the delay line. This transfer is moved earlier in the code in order to combine it with an unrelated

computation, saving one cycle. The second `i7=b7` can occur immediately after the input buffer is transferred, but it is moved later in the code so it can be combined with an unrelated computation.

There are two reasons why the user software interrupt 0 (USI0) is used to call the `calc_out` routine. The first reason is that the `calc_out` routine must be allowed to be interrupted by the timer interrupt service routine. The timer interrupt occurs more frequently than the `calc_out` routine. If the timer interrupt routine called the `calc_out` routine, the timer interrupt would occur again before the `calc_out` routine had completed. An interrupt routine cannot interrupt itself. Therefore, the `calc_out` routine is called from a lower priority interrupt (USI0).

This code structure also allows an easier transition from the counter-based rational rate changer algorithm to the external interrupt based algorithm described in the next section.

5.3.2 Code Listings–ratiobuf.asm

```
/************************************************************
File Name
   RATIOBUF.ASM

Version
   3/4/91

Purpose
   Timer-Driven Real Time Rational Rate Changer

Calling Parameters
   Input:  adc
   r0 = ADC input data

Return Values
   Output: dac
   r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC data / temp register
   r1 = temp register
   r2 = input counter
   r3 = output counter
   r4 = FIR coefficients
   r5 = scale value L
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels
   init_rat    reset-time initialization
   input      called by Timer Zero High Priority interrupt
   calc_out    called by Software Interrupt 0

Computation Time
   tmzh_svc = 15
   sft0_svc = NoverL + 2+intMoverL + 16

# PM Locations
   67 Words Code, N Words Data

# DM Locations
   N/L + M/L Words Data
************************************************************/
```

(listing continues on next page)

5 Multirate Filters

```
/*_____________________________________________
Creating & Using the Test Case:

Running the test case requires two data files:
    coef.dat        32-tap FIR filter coefficients (floating point)
    wave.1          waveform data file

Descriptions of coef.dat and wave.1 can be found in decimate.asm &
interpol.asm.

The test case writes the rate-changed output to a dac port.  Since there are
512 samples in wave.1, and the test case performs interpolation by 4 followed
by decimation by 3, there will be 682 data values written to the dac port if
all data samples are read in and processed.  The values written out are
left-shifted by 16 bits in parallel with the float—>fixed conversion, based
again on the assumption that the D/A converter port is located in the upper 16
bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

        asm21k -DTEST ratiobuf
        ld21k -a generic ratiobuf
        sim21k -a generic -e ratiobuf -k port1

    Note: the keystroke file port1.ksf opens two ports:
        1st port - DM, F0000000, Fixed-Pt., auto-wrap, wave.1
        2nd port - DM, F0000001, Fixed-Pt., no wrap,   out.1

2. In the simulator,
        a. Go to the program memory (disassembled) window,
        d. Go to symbol input, set to break on 513th occurence, and
        c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

_____________________________________________*/


#include “def21020.h”

#define N               32      /* N taps, N coefficients               */
#define L               4       /* interpolate by factor of L           */
#define NoverL          8       /* N must be an integer multiple of L   */
#define M               3       /* decimate by factor of M              */
#define intMoverL       2       /* smallest integer GE M/L              */
#define FP              20.0e6  /* Processor Frequency = 20MHz          */
#define FI              64.0e3  /* Interrupt Frequency = 64KHz          */

#define IN_CNTR         r2      /* input counter register               */
#define OUT_CNTR        r3      /* output counter register              */

#ifndef TEST
/*_____________________________________*/
```

```
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1                        */
#else
/*_____________________________*/
#define TPER_VAL 39         /* interrupt every 40 cycles              */
#endif
/*_____________________________*/

.SEGMENT /pm    pm_data;
.VAR    coef[N]=”coef.dat”;     /* coefficient circular buffer        */
.ENDSEG;

.SEGMENT /dm    dm_data;
.VAR    data[NoverL];           /* data delay line                    */
.VAR    input_buf[intMoverL];   /* input buffer is not circular       */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter    */
.PORT   dac;                /* Digital to Analog (output) converter   */
.ENDSEG;

/*_____________________________________
    RESET Service Routine
_____________________________________*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only   */
                DMWAIT=0x8421;      /* no wait states,internal ack only   */
                jump init_rat;      /* initialize the test case       */
.ENDSEG;

/*_____________________________________
   Timer=zero high priority Service Routine
_____________________________________*/
.SEGMENT /pm    tmzh_svc;
tmzh_svc:       IN_CNTR=IN_CNTR-1;          /* test if time for input */
                if ne jump skipin(db);
                    OUT_CNTR=OUT_CNTR-1;    /* test if time for output    */
                    nop;
                jump input (db);            /* calculate the input sample */
                    r0=dm(adc);             /* get input sample       */
                    r0=ashift r0 by -16;    /* right-justify the data */
.ENDSEG;

/*_____________________________________
   Software Interrupt 0 Service Routine
_____________________________________*/
.SEGMENT /pm    sft0_svc;
sft0_svc:       jump calc_out (db);     /* calculate the output sample    */
                    r0=i7;              /* get current ptr to input buffer */
                    r1=input_buf;       /* get start addr of input buffer */

.ENDSEG;
```

(listing continues on next page)

```
/*_____________________________________
    Initialize Interpolation Routine
_____________________________________*/
.SEGMENT /pm    pm_code;
init_rat:       /*  initialize buffer index registers  */
                b0=data; l0=@data; m0=1;    /* data buffer                  */
                b7=input_buf; l7=0;         /* input buffer                 */
                b8=coef; l8=@coef; m8=L;    /* modifier for coef is L!      */
                b9=0;l9=0;m9=-M;            /* track coefficient updates    */

                IN_CNTR = 1;                /* initialize flags, counters   */
                OUT_CNTR = 1;
                r5=L;                       /* scale register for interp.   */
                f5=float r5;
                r14=16;                     /* exponent scale factor        */

                r0=0;                   /* clear data buffer                */
                lcntr=NoverL, do clrbuf until lce;
clrbuf:             dm(i0,m0)=r0;

                tperiod=TPER_VAL;           /* program timer                */
                tcount=TPER_VAL;
                bit set mode2 TIMEN;

                /* enable timer zero high priority, software 0 interrupts   */
                bit set imask TMZHI|SFT0I;
                bit clr irptl TMZHI|SFT0I;      /* clear any pending irpts  */
                bit set mode1 IRPTEN|ALUSAT;    /* enable irpts, ALU sat    */

wait:           idle;
                jump wait;                      /* infinite wait loop       */

.ENDSEG;

/*_____________________________________
    Calculate output
        initiated by software interrupt 0, code below occurs at the
    output sample rate
_____________________________________*/
.SEGMENT /pm    pm_code;
calc_out:       r1=r0-r1, i7=b7;        /* calculate amount in inbuffer     */

                if eq jump modify_coef; /* skip do loop if buffer empty     */
                modify(i9,-M);          /* modify coef update by -M         */
                m10=i9;

                /* dump the input buffer into delay line if L inputs have   */
                /*   been acquired                                          */
                lcntr=r1, do load_data until lce;
                    f1=dm(i7,m0);
load_data:          dm(i0,m0)=f1;

modify_coef:    modify(i8,m10);         /* modify coef ptr by coef update   */
```

```
/* filter pass, occurs at L times the input sample rate                    */
fir:                        f0=dm(i0,m0), f4=pm(i8,m8);
                f8=f0*f4,   f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
                lcntr=NoverL-3, do taps until lce;
taps:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4, f8=f8+f12;
                f0=f8+f12,i7=b7;        /* reset input buffer in || w/comp   */

                i9=0;                   /* reset coef update                 */
                rti (db);
                    r0 = fix f0 by r14;     /* float -> fixed                */
                    dm(dac)=r0;             /* output data sample            */

.ENDSEG;

/*_____________________________________
    Acquire input
        initiated by Timer Interrupt, code occurs at L times the input rate
_____________________________________*/
.SEGMENT /pm    pm_code;
input:          /* convert fixed->float, modify the coef update by L        */
                f0=float r0,modify(i9,m8);
                f0=f0*f5;               /* scale by L                        */
                dm(i7,m0)=f0;           /* load in M long buffer             */
                /* set AZ flag w/ OUT_CNTR, reset the input count to L       */
                OUT_CNTR=pass OUT_CNTR, IN_CNTR=m8;

skipin:         if ne rti;              /* return if OUT_CNTR!=0             */

output:         rti(db);
                    /* cause output routine if OUT_CNTR==0 */
                    bit set irptl SFT0I;
                    OUT_CNTR=M;         /* reset output counter to M         */
.ENDSEG;
```

Listing 5.3 ratiobuf.asm

5 Multirate Filters

5.4 RATIONAL RATE CHANGER (EXTERNAL INTERRUPT-BASED)

The rational rate changer shown in `rat2int.asm` (Listing 5.4) performs the same operation as the rational rate changer (counter based) algorithm. The interrupt based algorithm, however, uses two external interrupts instead of the counters to generate the proper interpolation and decimation interrupts.

5.4.1 Implementation

The IRQ3 interrupt service routine performs the input function of the rational rate changer. An external timer is necessary to cause this interrupt to occur at the input sample rate. This interrupt replaces the timer interrupt and `IN_CNTR` in the counter based changer. The IRQ2 interrupt service routine performs the output function of the rational rate changer. Again, another external timer is necessary to cause the interrupt to occur at the output sample rate. This interrupt routine replaces the User Software Interrupt 0 routine and `OUT_CNTR` in the counter based changer.

Other aspects of the code, initialization of buffers, modifying coefficient pointers, and filter calculations, are executed in the same manner as in the previous example.

5.4.2 Code Listing–rat_2_int.asm

```
/************************************************************************
File Name
   RAT_2INT.ASM

Version
   3/4/91

Purpose
   Interrupt-Driven Real Time Rational Rate Changer

Calling Parameters
   Input:  adc
   r0 = ADC input data

Return Values
   Output: dac
   r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC data / temp register
   r1 = temp register
   r4 = FIR coefficients
   r5 = scale value L
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels
   init_rat    reset-time initialization
   irq3_svc    IRQ3 interrupt service routine for input
   output      called by IRQ2 interrupt service routine

Computation Time
   irq2_svc = 6
   irq3_svc = 3 + output
   output   >= NoverL + 14                    ( Notes: 1,3 )
               <= NoverL + 2*intMoverL + 13        ( Notes: 2,3 )

   Notes:
         1.  w/ no input samples to transfer
         2.  w/max input samples to transfer
         3.  maximum of 5 cache misses on the 1st pass

# PM Locations
   53 Words Code, N Words Data

# DM Locations
   N/L + M/L Words Data
************************************************************************/
```

(listing continues on next page)

5 Multirate Filters

```
_____________________________________________
Creating & Using the Test Case:

Running the test case requires two data files:
    coef.dat        32-tap FIR filter coefficients (floating point)
    wave.1          waveform data file

Descriptions of coef.dat and wave.1 can be found in decimate.asm &
interpol.asm.

The test case writes the rate-changed output to a dac port.  Since there are
512 samples in wave.1, and the test case performs interpolation by 4 followed
by decimation by 3, there will be 682 data values written to the dac port if
all data samples are read in and processed.  The values written out are
left-shifted by 16 bits in parallel with the float—>fixed conversion, based
again on the assumption that the D/A converter port is located in the upper 16
bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

        asm21k -DTEST rat_2int
        ld21k -a generic rat_2int
        sim21k -a generic -e rat_2int

2. In the simulator,
        a. Run keystroke file setint.ksf, which programs periodic interrupts,
        a. Go to the program memory (disassembled) window,
        d. Go to symbol irq3_svc, set to break on 513th occurence, and
        c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

_____________________________________________*/

#include "def21020.h"

#define N               32      /* N taps, N coefficients                   */
#define L               4       /* interpolate by factor of L               */
#define NoverL          8       /* N must be an integer multiple of L */
#define M               3       /* decimate by factor of M                  */
#define intMoverL       2       /* smallest integer GE M/L                  */

.SEGMENT /pm    pm_data;
.VAR    coef[N]="coef.dat";     /* coefficient circular buffer        */
.ENDSEG;

.SEGMENT /dm    dm_data;
.VAR    data[NoverL];           /* data delay line                          */
.VAR    input_buf[intMoverL];   /* input buffer is not circular       */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter          */
```

```
.PORT   dac;                /* Digital to Analog (output) converter          */
.ENDSEG;

/*_____________________________________
    RESET Service Routine
_____________________________________*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only      */
                DMWAIT=0x8421;      /* no wait states,internal ack only      */
                jump init_rat;      /* initialize the test case       */
.ENDSEG;

/*_____________________________________
    IRQ3 Interrupt Service Routine
    occurs at input rate
_____________________________________*/
.SEGMENT /pm irq3_svc;
irq3_svc:       r0=dm(adc);             /* get input sample                  */
                r0=ashift r0 by -16;    /* right-justify the data            */
                /* convert fixed->float, modify the coef update by L         */
                f0=float r0,modify(i9,m8);
                rti(db);
                    f0=f0*f5;           /* scale by L                        */
                    dm(i7,m0)=f0;       /* load in input buffer       */

.ENDSEG;

/*_____________________________________
    IRQ2 Interrupt Service Routine
    occurs at output rate
_____________________________________*/
.SEGMENT /pm irq2_svc;
irq2_svc:       jump output (db);       /* go to the output routine          */
                    r0=i7;              /* get input buffer pointer          */
                    r1=input_buf;       /* get start addr of input buffer    */
.ENDSEG;

/*_____________________________________
    Initialize Interpolation Routine
_____________________________________*/
.SEGMENT /pm    pm_code;
init_rat:       b0=data; l0=@data; m0=1;    /* data buffer                   */
                b7=input_buf; l7=0;         /* input buffer                  */
                b8=coef; l8=@coef; m8=L;    /* modifier for coef is L!       */
                b9=0;l9=0;m9=-M;            /* track coefficient updates     */

                r5=L;                       /* scale reg for interpolator    */
                f5=float r5;                /* fix -> float conversion       */
                r14=16;                     /* exponent scale factor         */

                r0=0;                       /* clear data buffer             */
                lcntr=NoverL, do clrbuf until lce;
clrbuf:             dm(i0,m0)=r0;
```

(listing continues on next page)

```
                /* enable timer zero high priority, software 0 interrupts    */
                bit set imask IRQ3I|IRQ2I;
                bit clr irptl IRQ3I|IRQ2I;      /* clear any pending irpts   */
                bit set mode1 IRPTEN|ALUSAT;    /* enable interrupts, ALU sat*/

wait:           idle;                           /* infinite wait loop        */
                jump wait;

.ENDSEG;

/*_____________________________________________
    Calculate output
        initiated by IRQ2, occurs at output sample rate
_____________________________________________*/
.SEGMENT /pm    pm_code;
output:         r1=r0-r1,i7=b7;         /* r1 = # samples in input buffer    */
                                        /*   also reset input buffer         */

                if eq jump mod_coef(db);/* skip do loop if buffer empty      */
                modify(i9,-M);          /* modify coef update by -M          */
                m10=i9;

                /* dump the buffer into delay line   */
                lcntr=r1, do load_data until lce;
                    f1=dm(i7,m0);
load_data:          dm(i0,m0)=f1;

mod_coef:       modify(i8,m10);         /* modify coef ptr by coef update    */

/* filter pass, occurs at L times the input sample rate */
fir:                        f0=dm(i0,m0), f4=pm(i8,m8);
                f8=f0*f4,   f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
                lcntr=NoverL-3, do taps until lce;
taps:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4, f8=f8+f12;
                f0=f8+f12, i7=b7;       /* reset input buffer in || w/comp   */

                i9=0;                   /* reset coef update                 */
                rti (db);
                    r0 = fix f0 by r14;     /* float -> fixed                */
                    dm(dac)=r0;             /* output data sample            */

.ENDSEG;
```

Listing 5.4 rat2int.asm

5.5 TWO-STAGE DECIMATION FILTER

In the previous example of a single-stage decimation filter, a single low-pass filter is used to prevent the aliasing that otherwise occurs with the rate compression. The filter presented here uses two stages to reduce the number of calculations required with no loss of precision.

If the output frequency from the decimation filter is much smaller than the input oversampled frequency, the transition band of the filter needs to be very small. A small transition band requires a large number of taps, which increases the computation time.

By cascading filters, the number of taps per stage can be reduced, thereby increasing the computational efficiency. For example, the first stage filter may not reduce the input sampling rate by very much, but the filter has a very large transition band requiring fewer taps. The second stage filter may then require a smaller transition band, but the output sample rate is smaller as well, still only needing a few taps. This two-stage implementation can be easily extended to more stages. See [CROCH83] for design curves to help determine optimal rat changed factors for intermediate stages. A two- or three-stage design provides a substantial reduction in filter computations; further reductions in computations come from adding more stages. Because the filters are cascaded, each stage must have a passband ripple equal to the final passband ripple divided by the number of stages used. The stopband ripple does not have to be less than the final stopband ripple because each successive stage attenuates the stopband ripple of the previous stage.

5.5.1 Implementation

This example (`dec2stg.asm`, Listing 5.5) uses two real time direct-form FIR filters of `N_1` and `N_2` taps to decimate by `M_1 * M_2` for a decrease of `1/(M_1 * M_2)` times the input sample rate. The use of an input buffer allows the filter computations to proceed in parallel with the acquisition of the next `M_1 * M_2` inputs, allowing a larger order filter than if all calculations are made between samples.

The timer interrupt is set at a rate equal to the input sample rate to store all the incoming values in the input buffer. A counter (`CNTR`) is set to `M_1 * M_2` . This allows the input buffer to have enough samples for both stages of filters. When the counter expires, User Service Interrupt 0 (USI0) is set. The USI0 interrupt service routine calls the `dec2stg` routine that performs the data transfer and two-stage filter.

The length of the input buffer is twice as long as the decimation rate (that is, `2 * M_1 * M_2`). This double-length buffer acts as two separate input buffers. While one buffer is being transferred to the data buffer for filtering, the second buffer stores values read in from the A/D. This double buffering is accomplished by utilizing two pointers (I7 and I6) that are offset from each other by `M_1 * M_2`. The buffer is circular, so that the pointers "ping-pong" between being the input pointer and the transfer pointer.

Since the input buffer is twice as long as needed, the input samples are now stored in the lower half of the input buffer with pointer I7, while the transfer from the input buffer to the data buffer is done with pointer I6. Since the double-buffer is circular, after the first time through the stages, the pointers "ping-pong," reversing roles. For more details, see [ADI90].

Because `M_1` is less than three for this test case, the movement of data from the input buffer to the buffer data1 is put at the start of the subroutine `dec2stg` as straight-line (i.e., not looped) code. If `M_1` is greater than three, the following code is used:

```
            r12=dm(i6,m0);
            r4=ashift r12 by -16;
            f0=float r4, r12=dm(i6,m0);
            lcntr=M_1-2, do load_data until lce;
                  r4=ashift r12 by -16, dm(i0,m0)=f0;
load_data:        f0=float r4, r12=dm(i6,m0);
            r4=ashift r12 by -16, dm(i0,m0)=f0;
            f0=float r4;
            dm(i0,m0)=f0;
```

The filter operations and coefficient interleaving are performed in the same manner as with the single-stage filters.

5.5.2 Code Listing–dec2stg.asm

```
/************************************************************************
File Name
   DEC2STG.ASM

Version
   3/18/91

Purpose
   Two Stage Decimator

Calling Parameters
   Input:  adc
   r15 = ADC input data

Return Values
   Output: dac
   r0 = DAC output data

Registers Affected
   r0 = FIR data in / ADC fltg-pt data
   r3 = counter
   r4 = FIR coefficients
   r8  = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r14 = 16 = exponent scale factor

Start Labels:
   init_dec    reset-time initialization
   dec2stg     called by software 0 interrupt

Computation Time:
   sft0_svc = N_2 + 10 + M_2*(N_1 + 2*M_1 + 7)
   tmzh_svc = 7

# PM Locations
   73 Words Code, N_1 + N_2 Words Data

# DM Locations
   N_1 + N_2 + M_1*M_2*2 Words
************************************************************************/
```

(listing continues on next page)

5 Multirate Filters

```
Creating & Using the Test Case:

Running the test case requires two data files:
    coef1.dat       32-tap FIR filter #1 coefficients (floating point)
    coef2.dat       128-tap FIR filter #2 coefficients (floating point)
    wave.1          waveform data file

coef1 & coef2 are the two parks-McClellan FIR filters used in this two-stage
decimator.  Assume that the input sample rate is 192kHz, and the decimated
output is 192/4 = 48kHz.  The first FIR filter has 32 taps, and is designed to
have maximum of 1dB of attenuation from 0 - 48kHz, then >90dB attenuation at
>96kHz.  The second FIR filter has 128 taps, and is designed to have maximum
of 1dB of attenuation from 0 - 20kHz, then >90dB attenuation at >24kHz.

Descriptions of wave.1 can be found in decimate.asm & interpol.asm.

The test case writes the decimated output to a dac port.  Since there are 512
samples in wave.1, and the test case performs decimation by 4, there will be
128 data values written to the dac port if all data samples are read in and
processed.  The values written out are left-shifted by 16 bits in parallel
with the float—>fixed conversion, based again on the assumption that the D/A
converter port is located in the upper 16 bits of data memory.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:

        asm21k -DTEST dec2stg
        ld21k -a generic dec2stg
        sim21k -a generic -e dec2stg

2. In the simulator,
        a. Go to the program memory (disassembled) window,
        d. Go to symbol output, set to break on 129th occurence, and
        c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

_____________________________________*/

#include “def21020.h”
#define N_1     32          /*  number of taps in FIR #1                    */
#define N_2     128         /*  number of taps in FIR #2                    */
#define M_1     2           /*  1st decimation factor                       */
#define M_2     2           /*  2nd decimation factor                       */
#define CNTR    r3          /*  counter                                     */
#define FP      20.0e6      /*  Processor Frequency = 20MHz                 */
#define FI      192.0e3     /*  Input Frequency = 192KHz                    */

#ifndef TEST    /*_____________________________*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1                        */
#else           /*_____________________________*/
#define TPER_VAL 104        /* interrupt every 160 cycles                   */
#endif          /*_____________________________*/
```

```
.SEGMENT /dm dm_data;
.VAR    data1[N_1];                 /* FIR input data #1                    */
.VAR    data2[N_2];                 /* FIR input data #2                    */
.VAR    input_buf[M_1*M_2*2];       /* raw ADC data
*/
.ENDSEG;

.SEGMENT /pm pm_data;
.VAR    coef1[N_1]=”coef1.dat”;     /* FIR floating-point coefficients
*/
.VAR    coef2[N_2]=”coef2.dat”;     /* FIR floating-point coefficients
*/
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                /* Analog to Digital (input) converter
*/
.PORT   dac;                /* Digital to Analog (output) converter
*/
.ENDSEG;

/*_____________________________________
    RESET Service Routine
_____________________________________*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only
*/
                DMWAIT=0x8421;      /* no wait states,internal ack only
*/
                call init_dec;      /* initialize the test case             */
wait:           idle;               /* infinite wait loop
*/
                jump wait;
.ENDSEG;

/*_____________________________________
    TMZH Service Routine
_____________________________________*/
.SEGMENT /pm    tmzh_svc;
/* sample input: this code occurs at the sample rate
*/
tmzh_svc:       /* decrement counter and get a sample
*/
                CNTR=CNTR-1,r15=dm(i2,m2);
                if eq jump zero_yes;
zero_no:        rti(db);
                    nop;
                    dm(i7,m0)=r15;              /* load in input buffer
*/
zero_yes:       rti(db);
                    irptl=SFT0I;                /* if counter==0, set interrupt
*/
                    dm(i7,m0)=r15;              /* load in input buffer
```

(listing continues on next page)

```
*/
.ENDSEG;

/*_____________________________________________
    Interrupt Request 3 Service Routine
        Dummy procedure so that the TMZH Service Routine does not fetch
    instructions from empty memory
_____________________________________________*/
.SEGMENT /pm    irq3_svc;
/* dummy procedure */
irq3_svc:       rti;
.ENDSEG;

/*_____________________________________________
    Software Interrupt 0 Service Routine
_____________________________________________*/
.SEGMENT /pm    sft0_svc;
/* process input data: this code occurs at 1/M times the sample rate
*/
sft0_svc:       jump dec2stg (db);              /* call the 2-stage decimator
*/
                    CNTR=M_1*M_2;               /* reset input counter
*/
                    nop;
.ENDSEG;

/*_____________________________________________
    efficient decimator initialization
_____________________________________________*/
.SEGMENT /pm    pm_code;
init_dec:       b0=data1; l0=@data1; m0=1;      /* data buffer #1
*/
                b1=data2; l1=@data2;            /* data buffer #2
*/
                b8=coef1; l8=@coef1; m8=1;      /* coefficient buffer #1
*/
                b9=coef2; l9=@coef2;            /* coefficient buffer #2
*/
                b7=input_buf; l7=@input_buf;    /* input buffer sample ptr
*/
                b6=input_buf; l6=@input_buf;    /* inp. buffer working ptr
*/
                b2=adc; l2=1; m2=0;             /* set A/D conv. pointer
*/

                r14=16;                         /* exponent scale factor
*/
```

```
                CNTR=M_1*M_2;                   /* set input counter
*/

                r0=0;                           /* clear data buffers       */
                lcntr=N_1, do clrbuf1 until lce;
clrbuf1:            dm(i0,m0)=r0;
                lcntr=N_2, do clrbuf2 until lce;
clrbuf2:            dm(i1,m0)=r0;

                tperiod=TPER_VAL;               /* program timer
*/
                tcount=TPER_VAL;
                bit set mode2 TIMEN;
                bit clr irptl TMZHI|SFT0I;      /* clear any pending irpts
*/

                rts (db);
                    /* enable Timer high priority, software 0 interrupts
*/
                    bit set imask TMZHI|SFT0I;
                    /* enable interrupts, nesting, ALU saturation
*/
                    bit set mode1 IRPTEN|NESTM|ALUSAT;
.ENDSEG;

/*_____________________________________
    efficient decimator routine
    executes at 1/(M_1*M_2) times the sample rate
_____________________________________*/
.SEGMENT /pm    pm_code;
dec2stg:    lcntr=M_2, do stage_1 until lce;
                /* transfer input samples to the FIR data memory space      */
                /* using the input buffer working pointer. Perform          */
                /* fix->float conversion in parallel                        */
                r12=dm(i6,m0);
                r4=ashift r12 by -16;
                f0=float r4, r12=dm(i6,m0);
                r4=ashift r12 by -16, dm(i0,m0)=f0;
                f0=float r4;
                dm(i0,m0)=f0;

                /* 1st FIR filter */
```

```
fir_1:                      f0=dm(i0,m0), f4=pm(i8,m8);
                f8=f0*f4,   f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
                lcntr=N_1-3, do taps_1 until lce;
taps_1:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                f12=f0*f4, f8=f8+f12;
                f0=f8+f12;

stage_1:        dm(i1,m0)=f0;   /* output to next filter data buffer
*/

            /* 2nd FIR filter */
fir_2:                  f0=dm(i1,m0), f4=pm(i9,m8);
            f8=f0*f4,   f0=dm(i1,m0), f4=pm(i9,m8);
            f12=f0*f4,  f0=dm(i1,m0), f4=pm(i9,m8);
            lcntr=N_2-3, do taps_2 until lce;
taps_2:         f12=f0*f4, f8=f8+f12, f0=dm(i1,m0), f4=pm(i9,m8);
            f12=f0*f4, f8=f8+f12;
            f0=f8+f12;

            rti (db);
                r0 = fix f0 by r14;     /* float –> fix
*/
output:         dm(dac)=r0;             /* output data sample
*/
.ENDSEG;
```

Listing 5.5 dec2stg.asm

5.6 TWO-STAGE INTERPOLATION FILTER

The two-stage interpolation filter (Listing 5.6) uses two real time N-tap direct form FIR filters to interpolate by `L_1 * L_2` for an increase of `L_1 * L_2` times the input sample rate. The number of taps is restricted such that (`N/L_1 * L_2`) must be an integer. The internal timer is used to generate interrupts at `L_1 * L_2` times the sample rate. Two counters are set up so that the delay line is only updated when a sample is ready at the ADC, that is, both counters have expired.

5.6.1 Implementation

Two delay lines are used, one for each filter stage. The delay line for the first stage filter, `int2stg`, is loaded from the ADC. The delay line for the

second stage filter, `do_fir2`, is loaded from the output of the first stage filter.

After the initialization of the buffers, the main program sits in an idle loop waiting for timer interrupts. The timer interrupts occur at an interval of `L_1 * L_2` times the sample rate. In the timer service routine, the counters for each stage are decremented. The counter with the smaller interpolation factor is decremented first (`CNT2=2`) . If the counter has not expired, the service routine jumps to the `do_fir2` routine. Therefore, the second stage of the filter is executed at an interval of `L_1 * L_2` times the input sample rate. If the counter has expired, the routine decrements the second counter.

The second counter is set to the higher interpolation factor (`CNT=4`) . If the counter has not expired, the service routine jumps to the `int2stg` routine. This routine is called at an interval of `L_1` times the input sample rate. When the `int2stg` routine is called, both the first stage and second stage filtering is performed.

If both counters have expired, the next input value is read from the ADC and then both filter stages are performed. The filter computations and coefficient interleaving are performed in the same manner as the single-stage interpolation filter.

5.6.2 Code Listing–int2stg.asm

```
/***************************************************************************
File Name
    INT2STG.ASM

Version
    3/18/91

Purpose
    Two Stage Interpolator

Calling Parameters
    Input:  adc
    r15 = ADC input data

Return Values
```

```
   Output: dac
   r0 = DAC output data

Registers Affected
   r1 = scale factor L
   r2 = counter #1
   r3 = counter #2
   r4 = FIR coefficients
   r8 = FIR accumulate result reg
   r12 = FIR multiply result reg / temporary reg
   r13 = 1  = counter compare register
   r14 = 16 = exponent scale factor

Start Labels
   init_int    reset-time initialization
   int2stg     called by Timer Zero High Priority interrupt

Computation Time
   tmzh_svc    = N_1/L_1 + N_2/L_2 + 25

# PM Locations
   67 Words Code, N_1 + N_2 Words Data

# DM Locations
   N_1/L_1 + N_2/L_2 Words Data
************************************************************************/
_____________________________________
Creating & Using the Test Case:

Running the test case requires two data files:
    coef.dat        32-tap FIR filter coefficients (floating point)
    sinX.dat        sine-wave data files (X=1,2,3,4,5)

coef.dat contains a 32-tap FIR filter which bandlimits the input signal to 1/8
of the input frequency.  Since the decimator in the test case decimates by a
factor M=4, this bandlimitation is equivalent to the required limit of 1/2 the
output frequency.  The filter is a Parks-McLellan filter with a passband
ripple of 1.5dB out to about 1/20 the input frequency, and a stopband with
greater than 70dB of attenuation above 1/8 the input frequency.

As an example, if the oversampled input frequency is 64kHz, the passband extends
out to about 3.2kHz. The stopband starts at 4kHz, and the output frequency is
16kHz.

The data files are all of the form sin.X, where X ranges from 1 to 5.  Each data
file contains 512 signed fixed-point data values in the range +/- 32767.  These
data points are meant to resemble data read from a 16-bit A/D converter which
```

```
generates signed data.  sin.1 through sin.4 contain simple sine waves at
different frequencies.  sin.5 contains a composite waveform generated from 3
sine wave of differing frequency and magnitude.

The test case writes the interpolated output to a dac port.  Since there are 512
samples in sin.X, and the test case performs decimation by 4, there will be 128
data values written to the dac port if all data samples are read in and
processed.

Armed with this information, you are ready to run:

1. Assemble & Start Simulator:
        asm21k -DTEST int2stg
        ld21k -a generic.ach int2stg
        sim21k -a generic.ach -e int2stg

2. In the simulator,
        a. Go to the program memory (disassembled) window,
        d. Go to symbol output, set to break on 4097th occurence, and
        c. Run

3. Compare wave.1 to out.1 on the graphing program of your choice.

_____________________________________*/


#include "def21020.h"
#define N_1     32          /* number of taps, FIR #1                      */
#define N_2     32          /* number of taps, FIR #2                      */
#define L_1     4           /* interpolate by factor of L_1                */
#define L_2     2           /* interpolate by factor of L_2                */
#define CNT1    r2
#define CNT2    r3

#define FP 20.0e6           /*  Processor Frequency = 20MHz                */
#define FI 64.0e3           /*  Input Frequency = 64KHz              */

#ifndef TEST    /*_____________________________*/
#define TPER_VAL 312        /*  TPER_VAL = FP/FI - 1                       */
#else           /*_____________________________*/
#define TPER_VAL 53         /* interrupt every 54 cycles             */
#endif          /*_____________________________*/

.SEGMENT /dm    dm_data;
```

(listing continues on next page)

```
.VAR    data1[N_1/L_1];         /* ADC fixed-point data buffer #1           */
.VAR    data2[N_2/L_2];         /* ADC fixed-point data buffer #2           */
.ENDSEG;

.SEGMENT /pm    pm_data;
.VAR    coef1[N_1]="coef1.dat"; /* fltg-pt. FIR #1 coefficients       */
.VAR    coef2[N_2]="coef2.dat"; /* fltg-pt. FIR #2 coefficients       */
.ENDSEG;

.SEGMENT /dm ports;
.PORT   adc;                    /* Analog to Digital (input) converter      */
.PORT   dac;                    /* Digital to Analog (output) converter     */
.ENDSEG;

/*_____________________________________________
    RESET Service Routine
_____________________________________________*/
.SEGMENT /pm    rst_svc;
rst_svc:        PMWAIT=0x0021;      /* no wait states,internal ack only     */
                DMWAIT=0x8421;      /* no wait states,internal ack only     */
                jump init_int;      /* initialize the test case       */
.ENDSEG;

/*_____________________________________________
    TMZH Service Routine
_____________________________________________*/
.SEGMENT /pm tmzh_svc;
/*  sample input: this interrupt occurs at the L_1*L_2*input rate           */
tmzh_svc:   CNT2=CNT2-1,modify(i9,m15); /* decrement counter, and           */
                                        /*    shift coef #2 pointer back    */
            if ne jump do_fir2;         /* do second FIR if CNT2!=0         */

            CNT1=CNT1-1,modify(i8,m15); /* decrement counter, and           */
                                        /*    shift coef #1 pointer back    */
            if ne jump int2stg;         /* go to first FIR if CNT1!=0       */
            jump sample (db);           /* get sample if CNT1==CNT1==0      */
                r15=dm(adc);            /* input data sample                */
                r15=ashift r15 by -16;  /* right-justify & zero garbage bits */

.ENDSEG;

/*_____________________________________________
    Initialize Interpolation Routine
_____________________________________________*/
.SEGMENT /pm    pm_code;
init_int:       /*  initialize buffer index registers  */
                b0=data1; l0=@data1; m0=1;      /* data buffer #1           */
                b1=data2; l1=@data2;            /* data buffer #2           */
                b8=coef1; l8=@coef1; m8=L_1;    /* modifier for coef1 = L_1  */
                b9=coef2; l9=@coef2; m9=L_2;    /* modifier for coef2 = L_2  */
                m15=-1;

                /* Register Initialization */
                r1=L_1*L_2;             /* value for upscaling sample       */
                f1=float r1;            /* fix –> float conversion          */
```

```
                CNT1=1;                 /* set interpolate cntrs to 1        */
                CNT2=1;                 /*    for first data sample          */
                r13=1;                  /* counter compare reg               */
                r14=16;                 /* exponent scale factor             */

                r0=0;                   /* clear data buffers                */
                lcntr=N_1/L_1, do clrbuf1 until lce;
clrbuf1:            dm(i0,m0)=r0;
                lcntr=N_2/L_2, do clrbuf2 until lce;
clrbuf2:            dm(i1,m0)=r0;

                tperiod=TPER_VAL;       /* program timer                     */
                tcount=TPER_VAL;

                bit set imask TMZHI;    /* enable TMZH interrupt             */
                bit clr irptl TMZHI;    /* clear any pending irpts           */

                /* enable interrupts, ALU sat */
                bit set mode1 IRPTEN|ALUSAT;
                bit set mode2 TIMEN;    /* turn timer on                     */

wait_interrupt: idle;
                jump wait_interrupt;    /* infinite wait loop                */

.ENDSEG;

/*_____________________________________
    Two-stage Interpolate
    code executes at L_1*L_2*(sample_rate)
_____________________________________*/
.SEGMENT /pm    pm_code;
int2stg:    /* filter pass, occurs at L_1 times the input sample rate */
                        f0=dm(i0,m0), f4=pm(i8,m8);
            f8=f0*f4,   f0=dm(i0,m0), f4=pm(i8,m8);
            f12=f0*f4,  f0=dm(i0,m0), f4=pm(i8,m8);
            lcntr=N_1/L_1-3, do taps1 until lce;
taps1:          f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
            f12=f0*f4, f8=f8+f12;
            f0=f8+f12;
            dm(i1,m0)=f0;

            modify(i9,m9);
            CNT2=m9;                    /* reset counter #2 to L_2           */

/* filter pass, occurs at L_1*L_2 times the input sample rate */
do_fir2:                f0=dm(i1,m0), f4=pm(i9,m9);
            f8=f0*f4,   f0=dm(i1,m0), f4=pm(i9,m9);
            f12=f0*f4,  f0=dm(i1,m0), f4=pm(i9,m9);
            lcntr=N_2/L_2-3, do taps2 until lce;
taps2:          f12=f0*f4, f8=f8+f12, f0=dm(i1,m0), f4=pm(i9,m9);
```

```
            f12=f0*f4, f8=f8+f12;
            f0=f8+f12;
            dm(i1,m0)=f0;

            rti (db);
                r0 = fix f0 by r14;             /* float -> fixed
*/
output:         dm(dac)=r0;                     /* output data sample
*/
.ENDSEG;

/*_____________________________________
    Acquire Sample
_____________________________________*/
.SEGMENT /pm pm_code;
/*  sample input: this code occurs at the input rate
*/
sample:     /* do fix->float, and scale data up by L_1*L_2
*/
            f0=float r15, modify(i8,m8);
            f0=f0*f1;                   /* upscale sample
*/
            jump int2stg (db);
                dm(i0,m0)=f0;           /* update delay line w/latest
*/
                CNT1=m8;                /* reset counter #1 to L_1
*/

.ENDSEG;
```

Listing 5.6 int2stg.asm

5.7 REFERENCES

[ADI90] Analog Devices Inc., Amy Mar, ed. 1990. *Digital Signal Processing Applications Using The ADSP-2100 Family*. Englewood Cliffs, NJ: Prentice Hall.

[CROCH83] Crochiere, Ronald E. and Lawrence L. Rabiner. 1983. *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice Hall.

Adaptive Filters 6

6.1 INTRODUCTION

Fixed-frequency-response digital filters were discussed in the two previous chapters. This chapter looks at filters with a frequency response, or transfer function, that can change over time to match desired system characteristics.

Many computationally efficient algorithms for adaptive filtering have been developed within the past twenty years. They are based on either a statistical approach, such as the least-mean square (LMS) algorithm, or a deterministic approach, such as the recursive least-squares (RLS) algorithm. The major advantage of the LMS algorithm is its computational simplicity. The RLS algorithm, conversely, offers faster convergence, but with a higher degree of computational complexity.

The adaptive filter algorithms discussed in this chapter are implemented with FIR filter structures. Since adaptive FIR filters have only adjustable zeros, they are free of stability problems that can be associated with adaptive IIR filters where both poles and zeros are adjustable. Of the various FIR filter structures available, the direct form (transversal), the symmetric transversal form, and the lattice form are the ones often employed in adaptive filtering applications.

6.1.1 Applications Of Adaptive Filters

Adaptive filters are widely used in telecommunications, control systems, radar systems, and in other systems where minimal information is available about the incoming signal.

Due to the variety of implementation options for adaptive filters, many aspects of adaptive filter design, as well as the development of some of the adaptive algorithms, are governed by the applications themselves. Several applications of adaptive filters based on FIR filter structures are described below.

6 Adaptive Filters

6.1.1.1 System Identification

You can design controls for a dynamic system if you have a model that describes the system in motion. Modeling is not easy with complex physical phenomena, however. You can get information about the system to be controlled from collecting experimental data of system responses to given excitations. This process of constructing models and estimating the best values of unknown parameters from experimental data is called *system identification.*

Figure 6.1 shows a block diagram of the system identification model. The unknown system is modeled by an FIR filter with adjustable coefficients. Both the unknown time-variant system and FIR filter model are excited by an input sequence *u(n).* The adaptive FIR filter output *y(n)* is compared with the unknown system output *d(n)* to produce an estimation error *e(n).* The estimation error represents the difference between the unknown system output and the model (estimated) output. The estimation error *e(n)* is then used as the input to an adaptive control algorithm which corrects the individual tap weights of the filter. This process is repeated through several iterations until the estimation error *e(n)* becomes sufficiently small in some statistical sense. The resultant FIR filter response now represents that of the previously unknown system.

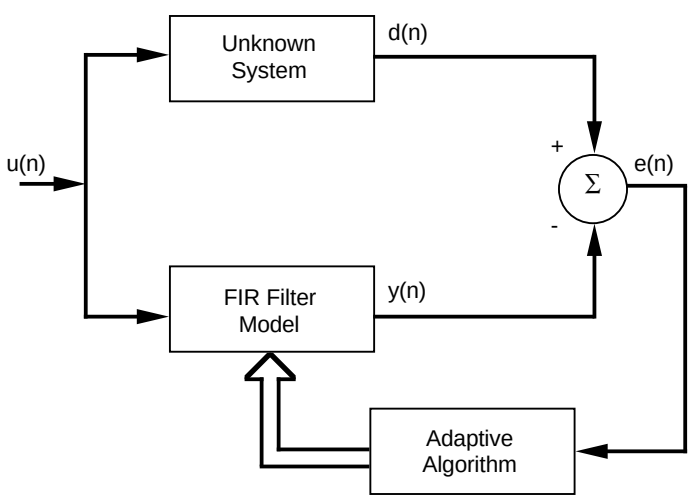


Figure 6.1 System Identification Model

6.1.1.2 Adaptive Equalization For Data Transmission

Adaptive filters are used widely to provide equalization in data modems that transmit data over speech-band and wider bandwidth channels. An adaptive equalizer is employed to compensate for the distortion caused by the transmission medium. Its operation involves a training mode followed by a tracking mode.

Initially, the equalizer is trained by transmitting a known test data sequence *u(n).* By generating a synchronized version of the test signal in the receiver, the adaptive equalizer is supplied with a desired response *d(n).* The equalizer output *y(n)* is subtracted from this desired response to produce an estimation error, which is in turn used to adaptively adjust the coefficients of the equalizer to their optimum values. When the initial training period is completed, the adaptive equalizer tracks possible time variations in channel characteristics during transmission by using a receiver estimate of the transmitted sequence as a desired response. The receiver estimate is obtained by applying the equalizer output *y(n)* to a decision device.

6.1.1.3 Echo Cancellation For Speech-Band Data Transmission

Dial-up switched telephone networks are used for low-volume infrequent data transmission. A device called a "hybrid" provides full-duplex operation, transmit and receive channels, from a two-wire telephone line. Due to impedance mismatch between the hybrid and the telephone channel, an "echo" is generated which can be suppressed by adaptive echo cancellers installed in the network in pairs. The cancellation is achieved by making an estimate of the echo signal components using the transmitted sequence *u(n)* as input data, and then subtracting the estimate *y(n)* from the sampled received signal *d(n).* The resulting error signal can be minimized, in the least-squares sense, to adjust optimally the weights of the echo canceller.

Similar applications include the suppression of narrowband interference in a wideband signal, adaptive line enhancement, and adaptive noise cancellation.

6.1.1.4 Linear Predictive Coding of Speech Signals

The method of linear predictive coding (LPC) is an example of a source coding algorithm used for the digital representation of speech signals. So called source coders are model dependent: they use previous knowledge of how the speech signal was generated at its source. Source coders for speech are generally referred to as *vocoders* and can operate at data rates of 4.8 Kbits/s or below. In LPC, the source vocal tract is modeled as a linear all-pole filter whose parameters are determined adaptively from speech samples by means of linear prediction. The speech samples *u(n)* are, in this case, the desired response, while *u(n-1)* forms the inputs to the adaptive FIR filter known as a prediction error filter. The error signal between u(n) and the output of the FIR filter, *y(n)*, is then minimized in the least-squares sense to estimate the model parameters. The error signal and the model parameters are encoded into a binary sequence and transmitted to the destination. At the receiver, the speech signal is synthesized from the model parameters and the error signal.

6.1.1.5 Array Processing

Adaptive antenna arrays use processing techniques that are very similar to those of adaptive filters. They use the spatial separation between the antenna elements to provide a parallel set of signal samples rather than using the time-delayed or partly processed versions of a one-dimensional input signal. Their applications include bearing estimation and adaptive beamforming.

6.1.2 FIR Filter Structures

An FIR system has a finite-duration impulse response that is zero outside of some finite time interval. Thus, an FIR system has a finite memory of length-*N* samples. Three basic structures for realizing the FIR filter (transversal, symmetric, and lattice) are described below.

6.1.2.1 Transversal Structure

Figure 6.2 shows the structure of a transversal FIR filter with N tap weights (adjustable during the adaptation process) with values at time n denoted as

$$w_0(n),\ w_1(n),\ \ldots,\ w_{N-1}(n).$$

The tap-weight vector, w(n), is represented as

$$\underline{w}(n) = [w_0(n)\ w_1(n)\ \ldots\ w_{N-1}(n)]T$$

the tap-input vector, $\underline{u}$(n), as

$$\underline{u}(n) = [u(n)\ u(n-1)\ \ldots\ u(n-N+1)]T$$

The FIR filter output, $y(n)$, can then be expressed as

$$y(n) = \underline{w}^T(n)\ \underline{u}(n) = \sum_{i=0}^{N-1} w_i(n)\ u(n-i)$$

where T denotes transpose, n is the time index, and N is the order of the filter.

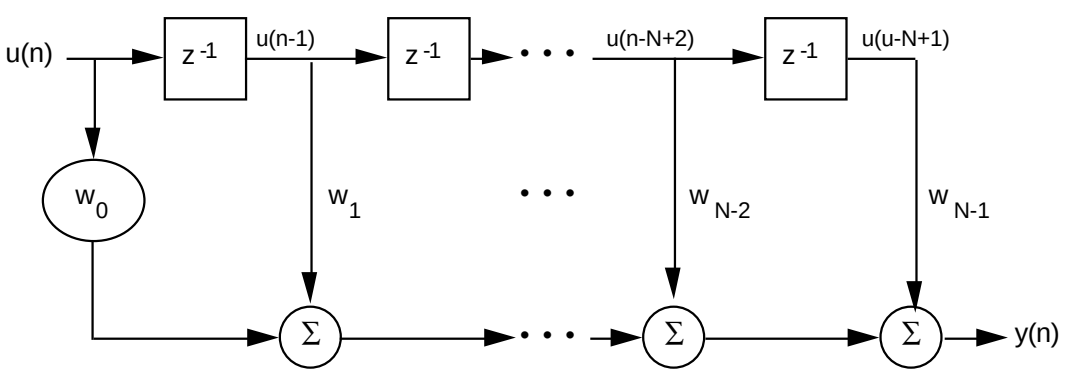


Figure 6.2 Transversal FIR Filter Structure

6.1.2.2 Symmetric Transversal Structure

The characteristic of linear phase response in a filter is sometimes desirable because it allows a system to reject or shape energy bands of the spectrum and still maintain the basic pulse integrity with a constant filter group delay. Imaging and digital communications are examples of applications where this characteristic is desirable.

An FIR filter with time domain symmetry, such as

$$w_0(n) = w_{N-1}(n),\ w_1(n) = w_{N-2}(n)...$$

has a linear phase response in the frequency domain. Consequently, the number of weights is reduced by a half in a transversal structure, as shown in Figure 6.3 with an even N tap weights. The tap-input vector becomes

$$u(n) = [u(n) + u(n-N+1),\ u(n-1) + u(n-N+2),\ ...,\ u(n-N/2+1) + u(n-N/2)]\ T$$

As a result, the filter output $y(n)$ becomes

$$y(n) = \sum_{i=0}^{N/2} w_i(n)[u(n-i) + u(n-N+1+i)]$$

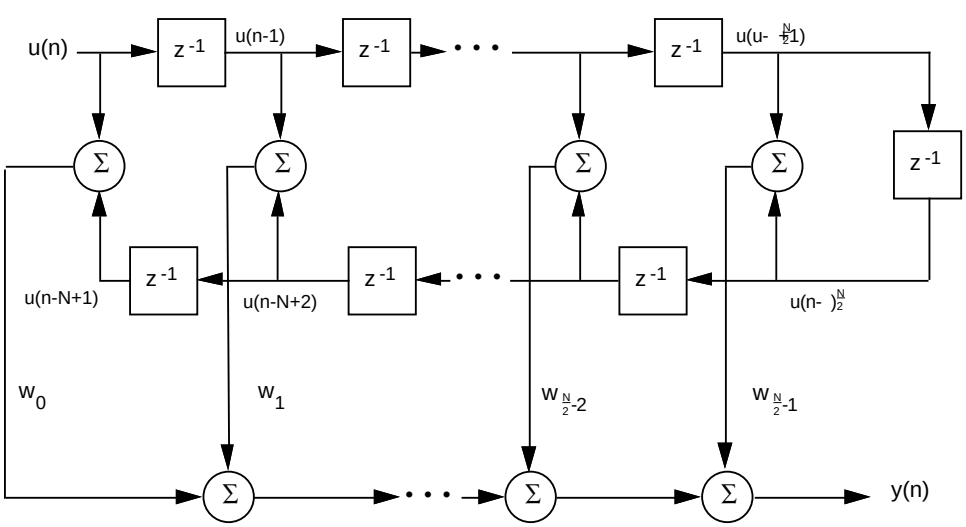


Figure 6.3 Symmetric Transversal Filter Structure

6.1.2.3 Lattice Structure

The lattice filter has a modular structure with cascaded identical stages. Figure 6.4 shows one stage of a lattice FIR structure.

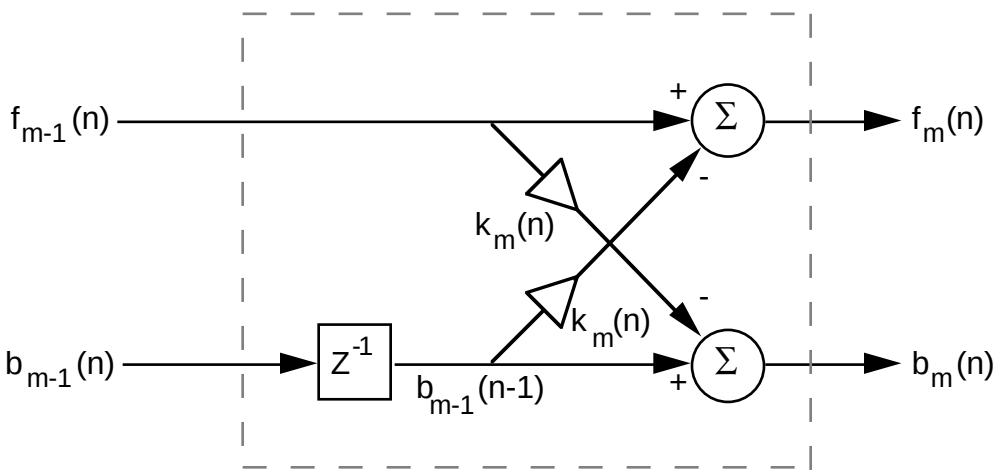


Figure 6.4 One Stage Of Lattice FIR

The lattice structure offers several advantages over the transversal structure:

- The lattice structure has good numerical round-off characteristics that make it less sensitive than the transversal structure to round-off errors and parameter variations.
- The lattice structure orthogonalizes the input signal stage-by-stage, which leads to fast convergence and efficient tracking capabilities when used in an adaptive environment.
- The various stages are decoupled from each other, so it is relatively easy to increase the prediction order if required.
- The lattice filter (predictor) can be interpreted as wave propagation in a stratified medium. This can represent an acoustical tube model of the human vocal tract, which is extremely useful in digital speech processing.

These advantages, however, come at the expense of an increased number of multiplies and adds for a given transfer function realization.

The following equations represent the dynamics of the mth stage of an order M lattice structure as derived from Figure 6.4:

$$f_m(n)=f_{m-1}(n) - K_m(n)\, b_{m-1}(n-1),\ 0 < m < M$$
$$b_m(n)=b_{m-1}(n-1)\, K_m(n)\, f_{m-1}(n),\ 0 < m < M$$

where $f_m(n)$ represents the forward prediction error, $b_m(n)$ represents the backward prediction error, $K_m(n)$ is the reflection coefficient, m is the stage index, and M is the number of cascaded stages. $K_m(n)$ has a magnitude less than one. The terms $f_m(n)$ and $b_m(n)$ are initialized as

$$f_0(n) = b_0(n) = u(n)$$

where $u(n)$ is the input signal.

Speech analysis is usually performed by using the lattice structure and the reflection coefficients $K_m(n)$. Since the dynamic range of $K_m(n)$ is significantly smaller than that of the tap weights, $w(n)$, of a transversal filter, the reflection coefficients require fewer bits to represent them. Hence, $K_m(n)$ are transmitted over the channel.

6.1.3 Adaptive Filter Algorithms

Two types of adaptive algorithms are discussed in this section: least-mean square (LMS) and recursive least-squares (RLS). LMS algorithms are based on a gradient-type search for tracking time-varying signal characteristics. RLS algorithms provide faster convergence and better tracking of time-variant signal statistics than LMS algorithms, but are more complex computationally.

6.1.3.1 The LMS Algorithm

The LMS algorithm is initialized by setting all the weights to zero at time n=0. Tap weights are updated using the relationship

$$\underline{w}(n+1) = \underline{w}(n) + \mu e(n)\underline{u}(n)$$

where $w(n)$ represents the tap weights of the transversal filter, $e(n)$ is the error signal, $u(n)$ represents the tap inputs, and the factor μ is the adaptation parameter or step-size. To ensure convergence, μ must satisfy the condition

$$0 < \mu < (2 \,/\, \textit{total input power})$$

where the *total input power* refers to the sum of the mean-square values of the tap inputs u*(n), u(n-1), ..., u(n-N+1)*. Moreover, the LMS convergence time depends on the ratio of maximum to minimum eigenvalues of the autocorrelation matrix R of the input signal.

To insure that μ does not become sufficiently large to cause filter instability, a Normalized LMSalgorithm can be employed. The normalized LMS employs a time-varying *mu* defined as

$$mu = \frac{x}{\underline{u}T(n).\ \underline{u}(n))}$$

where x is the normalized step-size chosen between 0 and 2. Tap weights are updated according to the relationship

$$w(n+1) = w(n) + \frac{xe(n)\underline{u}(n)}{r+\underline{u}T(n)\underline{u}(n)}$$

The term *x* is the new normalized adaptation constant, while *r* is a small positive term included to ensure that the update term does not become excessively large when $u_T(n)u(n)$ temporarily becomes small.

A problem can occur when the autocorrelation matrix associated with the input process has one or more zero eigenvalues. In this case, the adaptive filter will not converge to a unique solution. In addition, some uncoupled coefficients (weights) may grow without bound until hardware overflow or underflow occurs. This problem can be remedied by using coefficient leakage. This "leaky" LMS algorithm can be written as

$$\underline{w}(n+1) = (1-\mu r)\underline{w}(n) + \mu e(n)\underline{u}(n)$$

where the adaptation constant μ and the leakage coefficient *r* are a small positive values.

6.1.3.2 The RLS Algorithm

The LMS algorithm has many advantages (due to its computational simplicity), but its convergence rate is slow. The LMS algorithm has only one adjustable parameter that affects convergence rate, the step-size parameter μ, which has a limited range of adjustment in order to insure stability.

For faster rates of convergence, more complex algorithms with additional parameters must be used. The RLS algorithm uses a least-squares method to estimate correlation directly from the input data. The LMS algorithm uses the statistical mean-squared-error method, which is slower.

The RLS algorithm uses a transversal FIR filter implementation. The order of operations the algorithm takes is

1. Compute the filter output (tap weights initialized to zero)
2. Find the error signal
3. Compute the Kalman Gain Vector (defined below)
4. Update the inverse of the correlation matrix
5. Update the tap weights

The *Kalman Gain Vector* is based on input-data autocorrelation results, the input data itself, and a factor called the *forgetting factor*. The forgetting factor ranges between zero and one and provides a time-weighting of the input data such that the most recent data points are weighted more heavily than past data. This allows the filter coefficients to adapt to time-varying statistical characteristics of the input data.

The tap weight update is based on the error signal and the Kalman Gain Vector and is expressed as

$$w(n) = w(n-1) + Ke(n)$$

where K is the Kalman Gain Vector and e(n) represents the error signal.

6.2 IMPLEMENTATIONS

The adaptive filter routines have two inputs

- input sample
- desired response

and three outputs

- filter output
- filter error signal
- filter weights.

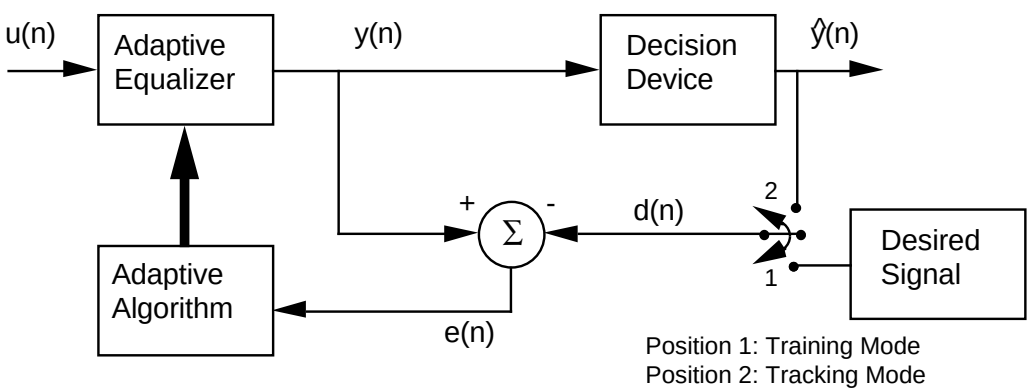


Figure 6.5 Generic Adaptive Filter

6 Adaptive Filters

6.2.1 Transversal Filter Implementation

The transversal and symmetric transversal FIR filter structures require the use of a delay line of input samples *u(n).* The samples *u(n), u(n-1), ..., u(n-N+1)* are stored in a circular Data Memory buffer in a reverse order. The filter weights $w_0, w_1,...,w_{N-1}$ are placed in a circular Program Memory Data buffer in forward order. Note that *u(n + 1)* replaces *u(n– N + 1)* at time *n + 1*; therefore, the index of the delay line buffer must point to *u(n - N + 1)* before the next filter iteration.

6.2.2 LMS (Transversal FIR Filter Structure)

The routine accepts two inputs, the input sample, u(n), and the desired output, d(n).

The filter output, *y(n)*, is calculated in terms of the difference equation

$$y(n) = \sum_{i=0}^{N-1} w_i(n)\, u(n-i)$$

The calculation is made in the single-instruction `macs` loop, which stores the *u(n)* value in a delay line when input. Once the output *y(n)* is calculated, its value is subtracted from *d(n)* to yield the error signal, *e(n)*. The tap weights are updated according to the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * u(n-i)$$

where $0 \le i \le N-1$.The `weights` buffer is then updated with the new value. `STEPSIZE` is a constant set to 0.005. Increasing this value decreases adaptation time, but has a negative effect on the system's steady-state mean-squared error.

6.2.2.1 Code Listing—lms.asm

```
/
*************************************************************************

File Name
   LMS.ASM

Version
   April 2 1991

Purpose
   Performs LMS algorithm implemented with a transversal FIR filter structure.

Equations Implemented
   ***************************************************************
   * 1)  y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
   * where  w= [w0(n) w1(n) ... wN-1(n)]= filter weights           *
   * and  u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
   * n= time index, N= number of filter weights (taps)            *
   * 2)  e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
   * 3)  wi(n+1)= wi(n)+STEPSIZE*e(n)*u(n-i),  0 =<i<= N-1         *
   ***************************************************************

Calling Parameters
   f0= u(n) = input sample
   f1= d(n) = desired output

Return Values
   f13= y(n)= filter output
   f6= e(n)= filter error signal
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f4, f6, f7, f8, f12, f13

Cycle Count
   lms_alg: 3N+8 per iteration, lms_init: 12+N

# PM Locations
   pm code= 29 words, pm data= N words

# DM Locations
   dm data= N words

*************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   lms_init, lms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;
```

(listing continues on next page)

6 Adaptive Filters

```
.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
lms_init: b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0;
          rts;            /* clear delay line & weights */



lms_alg:  dm(i0,m0)=f0, f4=pm(i8,m8);
                                  /* store u(n) in delay line, f4=w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8); /*f8= u(n)*w0(n)
                                     f0= u(n-1), f4= w1(n) */
          f12=f0*f4, f0=dm(i0,m0), f4= pm(i8,m8);
                        /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */
          lcntr=TAPS-3, do macs until lce;
macs:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4= pm(i8,m8);
                                    /* f12= u(n-i)*wi(n), f8= sum of prod,
                                          f0= u(n-i-1), f4= wi+1(n)   */
          f12=f0*f4, f8=f8+f12;   /* f12= u(n-N+1)*wN-1(n) */
          f13=f8+f12;             /* f13= y(n) */
          f6=f1-f13;              /* f6= e(n) */
          f1=f6*f7, f4=dm(i0,m0); /* f1= STEPSIZE*e(n), f4= u(n) */
          f0=f1*f4, f12=pm(i8,m8);/* f0= STEPSIZE*e(n)*u(n),
                                     f12= w0(n) */

          lcntr=TAPS-1, do update_weights until lce;
            f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8); /* f8= wi(n+1) */
                                   /* f4= u(n-i-1), f12= wi+1(n) */

update_weights:   f0=f1*f4, pm(i9,m8)=f8;     /* f0= STEPSIZE*e(n)*u(n-i-1) */
                                   /* store wi(n+1) */
          rts(db);
          f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1) */
                       /* i0 -> u(n+1) location in delay line */
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.1 lms.asm

6.2.3 llms.asm—Leaky LMS Algorithm (Transversal)

Implementation of this algorithm is similar to the standard LMS, except the tap weight update calculation takes into account the constant LEAK_COEF , such that

$$w_i(n+1) = \text{LEAK\_COEF} * w_i(n) + \text{STEPSIZE} * e(n) * u(n-i)$$

6.2.3.1 Code Listing

```
/
*****************************************************************************

File Name
    LLMS.ASM

Version
    April 2 1991

Purpose
    Performs the “leaky” LMS algorithm implemented with a
    transversal FIR filter structure


Equations Implemented
    ***************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights           *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)           *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= LEAK_COEF*wi(n)+STEPSIZE*e(n)*u(n-i), 0 =<i<= N-1 *
    ***************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
    f13= y(n)= filter output
    f6= e(n)= filter error signal
    f8 -> Program Memory Buffer of the filter weights

Registers Affected
    f0, f1, f2, f4, f6, f7, f8, f9, f12, f13

Cycle Count
    llms_alg: 3N+8 per iteration, llms_init: 13+N

# PM Locations
    pm code= 30 words, pm data= N words

# DM Locations
    dm data= N words
```

(listing continues on next page)

6 Adaptive Filters

```
*************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.0045
#define LEAK_COEF       0.9995

.GLOBAL   llms_init, llms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
llms_init:       b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f2=LEAK_COEF;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0; /* clear delay line & weights */
          rts;

llms_alg: f9= pass f1, dm(i0,m0)=f0, f4=pm(i8,m8);
                   /* store u(n) in delay line, f4= w0(n), f9= d(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8); /* f8= u(n)*w0(n) */
                                        /* f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                          /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:             f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                                  /* f12= u(n-i)*wi(n), f8= sum of prod,
                                     f0= u(n-i-1), f4= wi+1(n) */

          f12=f0*f4, f8=f8+f12;               /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f4=pm(i8,m8);           /* f13= y(n), f4= w0(n) */
          f12=f2*f4, f6=f9-f13, f4=dm(i0,m0); /* f12= LEAK_COEF*w0(n)
                                                  f6= e(n), f4= u(n)*/
          f0=f6*f7;                           /* f0= STEPSIZE*e(n) */
          f9=f0*f4, f4=dm(i0,m0);             /* f9= STEPSIZE*e(n)*u(n),
                                                  f4= u(n-1) */
```

```
          lcntr=TAPS-1, do update_weights until lce;
f9=f0*f4, f8=f9+f12, f4=dm(i0,m0), f12=pm(i8,m8); /*
f9=STEPSIZE*e(n)*u(n-i-1), f8=wi(n+1),
          f4=u(n-i-2),f12=wi+1(n) */
update_weights:   f12=f2*f12, pm(i9,m8)=f8;
/* f12= LEAK_COEF*wi+1(n), store wi(n+1) */ rts(db);
          f8=f9+f12, f0=dm(i0,2);
/* f8= wN-1(n+1) i0 -> u(n+1) location in delay line */
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.2 llms.asm

6.2.4 Normalized LMS Algorithm (Transversal)

This implementation is similar to the standard LMS of `lms.asm`, but adds calculation for the normalized stepsize. The normalized stepsize is calculated using the constants `ALPHA` and `GAMMA` and the value *E(n)*, which represents the energy in the delay line. *E(n)*is calculated recursively in the outer `nlms_alg` loop, just prior to the main filter calculation loop, `macs`.

6 Adaptive Filters

6.2.4.1 Code Listing—nlms.asm

```
/*************************************************************************

    File Name
        NLMS.ASM

    Version
        April 2 1991

    Purpose
        Performs the normalized LMS algorithm implemented
        with a transversal FIR filter structure.


    Equations Implemented
        ***************************************************************
        * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
        * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
        * and u=[ u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
        * n= time index, N= number of filter weights (taps)            *
        * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
        * 3) wi(n+1)= wi(n)+normalized_stepsize*e(n)*u(n-i), 0 =<i<= N-1*
        * 4) normalized_stepsize= ALPHA/(GAMMA+E(n))                    *
        * where E(n)= u.u = energy in delay line                        *
        * E(n) is computed recursively as follows                       *
        * 5) E(n)= E(n-1)+u(n)**2-u(n-N)**2                             *
        ***************************************************************

    Calling Parameters
        f0= u(n) = input sample
        f1= d(n) = desired output

    Return Values
        f2= y(n)= filter output
        f6= e(n)= filter error signal
        i8 -> Program Memory Data buffer of the filter weights

    Registers Affected
        f0, f1, f2, f4, f5, f6, f7, f8, f9, f11, f12, f13, f14

    Cycle Count
        nlms_alg: 3N+16 per iteration, nlms_init: 14+N

    # PM Locations
        pm code= 32 words, pm data= N words

    # DM Locations
        dm data= N words

    *************************************************************************/
```

```
#define   TAPS         5
#define   ALPHA        0.1
#define   GAMMA        0.1

#include “a:\global\macros.h”

.GLOBAL   nlms_init, nlms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;



.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
nlms_init:       b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=weights;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=TAPS;
          f5=ALPHA;
          f11=GAMMA;      /* f11= E(0)= GAMMA */
          f9= 2.0;        /* f9= 2.0  for DIVIDE_ macro */
          f13=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f13, pm(i8,m8)=f13;
                                /* clear delay line & weights */
          rts;

nlms_alg: f14=f0*f0, dm(i0,m0)=f0, f4=pm(i8,m8);
                /* f14= u(n)**2, store u(n) in delay line, f4= w0(n) */
   f8=f0*f4, f11=f11+f14, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f11= E(n-1)+u(n)**2, f8=u(n)*w0(n) */
          f12=f0*f4, f11=f11-f13, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f12= u(n-1)*w1(n), f11= E(n), f0= u(n-2),
                    f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
                /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1),
                         f4= wi+1(n) */
```

(listing continues on next page)

```
            f12=f0*f4, f8=f8+f12, f14=f11; /* f12= u(n-N+1)*wN-1(n),
                                              f14= E(n) */
            f2=f8+f12;                /* f2= y(n) */
            f6=f1-f2, f4=dm(i0,m0);     /* f6= e(n), f4= u(n) */
            f7=f6*f5;                 /* f7= ALPHA*e(n) */
            DIVIDE(f1,f7,f14,f9,f0); /* f1= normalized_stepsize*e(n) */
            f0=f1*f4, f12=pm(i8,m8); /* f0= f1*u(n), f12= w0(n) */

            lcntr=TAPS-1, do update_weights until lce;
              f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8); /* f8= wi(n+1)
                                                f4= u(n-i-1), f12= wi+1(n) */
update_weights:   f0=f1*f4, pm(i9,m8)=f8;
                                          /* f0=normalized_stepsize*e(n)*u(n-i-1)
                                                store wi(n+1)*/
            rts(db);
            f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1)
                                    i0 -> u(n+1) location in delay line */
            f13=f4*f4, pm(i9,m8)=f8; /* f13= u(n-N)**2, store wN-1(n+1) */

.ENDSEG;
```

Listing 6.3 nlms.asm

6.2.5 Sign-Error LMS (Transversal)

The sign-error LMS, along with sign-data and sign-sign implementations, represent attempts to simplify the computational requirements of the LMS by reducing the number of multiplies required. While this approach has benefits for discrete IC or VLSI implementations, there is no computational benefit for programmable DSP processor implementations. The filter implementations are similar to the standard LMS, but the tap weight updates use information related to the sign of the error signal and/or the input data.

For the sign-error algorithm, taps are updated according to the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * sgn\{e(n)\} * u(n-i)$$

where

$sgn\{x\} = 1$ for $x \geq 0$
$sgn\{x\} = -1$ for $x < 0$

This function is implemented after the main filter calculation loop, `macs`, and prior to the `update_weights` loop, as the error signal is used across all weights *i*, for any given time, *n*.

6.2.5.1 Code Listing—selms.asm

```
/
*******************************************************************************

File Name
     SELMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-error LMS algorithm implemented
    with a transversal FIR filter structure

Equations Implemented
    ********************************************************************
    * 1) y(n)= w.u  ( . = dot_product) , y(n)= FIR filter output       *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights               *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line   *
    * n= time index, N= number of filter weights (taps)               *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output    *
    * 3) wi(n+1)= wi(n)+STEPSIZE*sgn[e(n)]*u(n-i), 0 =<i<= N-1         *
    * where sgn[e(n)]= +1 if e(n) >= 0 and -1 if e(n) < 0              *
    ********************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
     f13= y(n)= filter output
     f1= e(n)= filter error signal
     i8 -> Program Memory Data buffer of the filter weights

Registers Affected
    f0, f1, f2, f4, f7, f8, f12, f13

Cycle Count
    selms_alg: 3N+8 per iteration, selms_init: 12+N

# PM Locations
    pm code: 29 words, pm data= N words

# DM Locations
    dm data= N words
```

(listing continues on next page)

6 Adaptive Filters

```
*************************************************************************/

#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   selms_init;
.GLOBAL selms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
selms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;                /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;                /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                                  /* clear delay line & weights */
          rts;


selms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                                  /* store u(n) in delay line, f4= w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                  /* f8= u(n)*w0(n)f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                  /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/
```

```
           lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);

            /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n) */
            f12=f0*f4, f8=f8+f12;   /* f12=u(n-N+1)*wN-1(n) */
            f13=f8+f12, f2=f7;      /* f13= y(n), f2= step-size */
            f1=f1-f13, f4=dm(i0,m0);/* f1= e(n), f4= u(n) */
            if lt f2=-f7;
                                    /* if e(n) < 0 then f2= -step-size */
           f0=f2*f4, f12=pm(i8,m8);/* f0= f2*u(n), f12= w0(n) */

           lcntr=TAPS-1, do update_weights until lce;
             f8=f0+f12, f4=dm(i0,m0), f12=pm(i8,m8);
                                   /* f8= wi(n+1), f4= u(n-i-1), f12= wi+1(n) */
update_weights:   f0=f2*f4, pm(i9,m8)=f8;
                                    /* store wi(n+1) */

            rts(db);
            f8=f0+f12, f0=dm(i0,1); /* f8= wN-1(n+1) */
                                    /* i0 -> u(n+1) location in delay line */
            pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.4 selms.asm

6.2.6 Sign-Data LMS (Transversal)

Another sign variation, this time relying on the sign of the data input, where

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * sgn\{u(n - i)\}.$$

The sign function here is implemented in the `update_weights` loop, as

previous input values have influence across the taps, *i*, for a given time, *n*.

6.2.6.1 Code Listing—sdlms.asm

```
/
*************************************************************************

File Name
     SDLMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-data LMS algorithm implemented with
    a transversal FIR filter structure

Equations Implemented
    ****************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)            *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= wi(n)+STEPSIZE*e(n)*sgn[u(n-i)], 0 =<i<= N-1      *
    * where sgn[u(n)]= +1 if u(n) >= 0 and -1 if u(n) < 0           *
    ****************************************************************

Calling Parameters
    f0= u(n)= input sample
    f1= d(n)= desired output

Return Values
    f13= y(n)= filter output
    f6= e(n)= filter error signal
    i8 -> Program Memory Data buffer of the filter weights

Registers Affected
    f0, f1, f4, f5, f6, f7, f8, f9, f12, f13

Cycle Count
    sdlms_alg: 4N+8 per iteration, sdlms_init: 13+N

# PM Locations
    pm code= 32 words, pm data= N words

# DM Locations
```

```
   dm data= N words
*****************************************************************************/


#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   sdlms_init, sdlms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sdlms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;                  /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;                  /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f5=1.0;
          f0=0.0;
          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                                    /* clear delay line & weights */
          rts;


sdlms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                                    /* f4=w0(n),store u(n) in delay line */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                    /* f8= u(n)*w0(n) f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                   /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/
```

(listing continues on next page)

```
          lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
              /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n)
*/

          f12=f0*f4, f8=f8+f12;     /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f12=pm(i8,m8);/* f13= y(n), f12= w0(n) */
          f6=f1-f13, f8=dm(i0,m0);  /* f6= e(n), f8= u(n) */
          f1=f6*f7, f4=dm(i0,m0);   /* f1= STEPSIZE*e(n),f4=u(n-1) */
          f0=pass f8, f9=f1;        /* set ALU flags for sign of u(n) */

          lcntr=TAPS-1, do update_weights until lce;
            if lt f9=-f1;
                                     /* if u(n-i) < 0 then f9= -STEPSIZE*e(n) */
            f9=f1*f5, f8=f9+f12, f12=pm(i8,m8);
                                   /* f8= wi-1(n+1) f12= wi(n), f9= STEPSIZE*e(n)
*/
update_weights:   f0=pass f4, f4=dm(i0,m0), pm(i9,m8)=f8;
               /*store wi-1(n+1) set ALU flags for sign of u(n-i), f4= u(n-i-1)
*/

          if lt f9=-f1;
          rts(db);
          f8=f9+f12, f0=dm(i0,2);
```

```
                              /* f8= wN-1(n+1) i0 -> u(n+1) location in delay line
*/
          pm(i9,m8)=f8;             /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.5 sdlms.asm

6.2.7 Sign-Sign LMS (Transversal)

This sign variation uses sign information from both the error signal and input value to calculate the tap weight updates, using the relationship

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * sgn\{e(n)\} * sgn\{u(n-i)\}$$

The error value, *e(n)*, and the input value, *u(n)*, are multiplied together in the `update_weights` loop in order to set the appropriate multiplier sign flags. The resultant flags determine whether the `STEPSIZE` value is added or subtracted from the past tap weight.

6.2.7.1 Code Listing—sslms.asm

```
/
*************************************************************************

File Name
     SSLMS.ASM

Version
    April 2 1991

Purpose
    Performs the sign-sign LMS algorithm implemented with
    a transversal FIR filter structure

Equations Implemented
    ***************************************************************
    * 1) y(n)= w.u  ( . = dot_product), y(n)= FIR filter output     *
    * where w= [w0(n) w1(n) ... wN-1(n)]= filter weights            *
    * and u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line*
    * n= time index, N= number of filter weights (taps)            *
    * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
    * 3) wi(n+1)= wi(n)+STEPSIZE*sgn[e(n)]*sgn[u(n-i)], 0 =<i<= N-1 *
    * where sgn[x]= +1 if x >= 0 and -1 if x < 0                    *
    ***************************************************************

Calling Parameters
    f0= u(n)= input sample
```

(listing continues on next page)

```
   f1= d(n)= desired output

Return Values
   f13= y(n)= filter output
   f1= e(n)= filter error signal
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f4, f7, f8, f9, f12, f13

Cycle Count
   sslms_alg: 4N+7 per iteration, sslms_init: 13+N

# PM Locations
   pm code= 31 words, pm data= N words

# DM Locations
   dm data= N words

*****************************************************************************/


#define   TAPS         5
#define   STEPSIZE     0.005

.GLOBAL   sslms_init, sslms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sslms_init:      b0=deline_data;
          m0=-1;
          l0=TAPS;        /* circular delay line buffer */
          b8=weights;
          b9=b8;
          m8=1;
          l8=TAPS;        /* circular weight buffer */
          l9=l8;
          f7=STEPSIZE;
          f2=1.0;
          f0=0.0;

          lcntr=TAPS, do clear_bufs until lce;
clear_bufs:       dm(i0,m0)=f0, pm(i8,m8)=f0; /* clear delay line & weights */
          rts;


sslms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
```

```
                                    /* f4=w0(n), store u(n) in delay line */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                    /* f8= u(n)*w0(n), f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                                   /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n)
*/

          lcntr=TAPS-3, do macs until lce;
macs:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
               /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4= wi+1(n)
*/

          f12=f0*f4, f8=f8+f12;     /* f12=u(n-N+1)*wN-1(n) */
          f13=f8+f12, f8=dm(i0,m0), f12=pm(i8,m8);
                                     /* f13= y(n), f12= w0(n) */
           f1=f1-f13, f9=f7;         /* f1= e(n), f8= u(n), f9= STEPSIZE */
          f0=f1*f8, f4=dm(i0,m0);
        /* f0= u(n)*e(n) to set multiplier flags for sign of product, f4=u(n-1)
*/

          lcntr=TAPS-1, do update_weights until lce;
            if ms f9=-f7;          /* if u(n)*e(n)<0 then f9=-STEPSIZE */
            f9=f2*f7, f8=f9+f12, f12=pm(i8,m8);
                            /* restore f9 to STEPSIZE f8= wi-1(n+1), f12= wi(n)
*/
update_weights:   f0=f1*f4, f4=dm(i0,m0), pm(i9,m8)=f8;
                         /* store wi-1(n+1),set multiplier flags, f4= u(n-i-1)
*/
          if ms f9=-f7;
          rts(db);
          f8=f9+f12, f0=dm(i0,2);
                         /* i0 -> u(n+1) location in delay line,.f8= wN-1(n+1)
*/
          pm(i9,m8)=f8;           /* store wN-1(n+1) */

.ENDSEG;
```

Listing 6.6 sslms.asm

6.2.8 Symmetric Transversal Filter Implementation LMS

In this routine, the filter output is defined as

$$N-1$$

6 Adaptive Filters

$$y(n) = \sum_{i=0} w_i(n) * [u(n-i) + u(n-M+i+1)].$$

The filter calculation is broken up into two loops. The `macs` loop is similar to the one in `lms.asm`, but this algorithm adds a second calculation loop, `macs1`, to account for the second term in the sum-of-products. The `update_weights` loop has an added multiply to account for the extra term in the weight update equation

$$w_i(n+1) = w_i(n) + \texttt{STEPSIZE} * e(n) * [u(n-i) + u(n-M+i+1)]$$

6.2.8.1 Code Listing—sylms.asm

```
/
*******************************************************************************

File Name
   SYLMS.ASM

Version
   April 2 1991

Purpose
   Even order symmetric transversal filter structure implementation of the LMS
algorithm

Equations Implemented
   ***************************************************************
   * 1) y(n)= SUM[wi(n)*[u(n-i)+u(n-M+i+1)]] for 0 =<i<= N-1       *
   * where y(n)= FIR filter output, wi= filter weights             *
   *   u= input samples in delay line, N= number of weights (taps),*
   *   n= time index, and M= 2N= length of delay line              *
   * 2) e(n)= d(n)-y(n), e(n)= error signal & d(n)= desired output *
   * 3) wi(n+1)= wi(n)+STEPSIZE*e(n)*[u(n-i)+u(n-M+i+1)], 0=<i<=N-1*
    ***************************************************************

Calling Parameters
   Calling parameters (inputs):
   f0= u(n)= input sample
   f1= d(n)= desired output

Return Values
   f13= y(n)= filter output
   f6= e(n)= filter error signal
```

```
   i8 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f3, f4, f6, f7, f8, f9, f12, f13

Cycle Count
   sylms_alg: 2M+10 per iteration, sylms_init: 17+M

# PM Locations
   pm code= 39 words, pm data= N words

# DM Locations
   dm data= 2N

*************************************************************************/


#define   TAPS          4
#define   STEPSIZE     0.005

.GLOBAL   sylms_init, sylms_alg;

.SEGMENT/DM     dm_data;
.VAR      deline_data[2*TAPS];
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      weights[TAPS];
.ENDSEG;

.SEGMENT/PM     pm_code;
sylms_init:      b0=deline_data;
          b3=deline_data;
          m0=-1;
          m1=1;
          m2=TAPS+2;
          l0=2*TAPS;   /* circular delay line buffer of length M= 2N */
          l3=2*TAPS;
          b8=weights;
          b9=weights;
          m8=1;
          m9=-1;
          m10=-2;
          l8=TAPS;     /* circular weight buffer */
          l9=TAPS;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=2*TAPS, do clear_bufs until lce;
clear_bufs:        dm(i0,m0)=f0, pm(i8,m8)=f0;
                        /* clear delay line & weights */
```

(listing continues on next page)

```
          rts;

sylms_alg:       dm(i0,m0)=f0, f4=pm(i8,m8);
                          /* store u(n) in delay line, f4=w0(n) */
          f8=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                          /* f8= u(n)*w0(n), f0= u(n-1), f4= w1(n) */
   f12=f0*f4, f0=dm(i0,m0), f4=pm(i8,m8);
                        /* f12= u(n-1)*w1(n), f0= u(n-2), f4= w2(n) */

          lcntr=TAPS-3, do macs until lce;
macs:               f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m8);
          /* f12= u(n-i)*wi(n), f8= sum of prod, f0= u(n-i-1), f4=wi+1(n) */

          f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f9=pm(i8,m10);
                           /* f12= u(n-N+1)*wN-1(n), i8 -> wN-2(n) */

          lcntr=TAPS-1, do macs1 until lce;
macs1:              f12=f0*f4, f8=f8+f12, f0=dm(i0,m0), f4=pm(i8,m9);
                           /* f4=wi-1(n) */
          f12=f0*f4, f8=f8+f12, i3=i0;
          f13=f8+f12, f9=dm(i0,m0), f12=pm(i8,m8);
                          /* f13= y(n), i0 -> u(n-1), i8 -> w0(n) */
          f6=f1-f13, f4=dm(i3,m1);
                           /*f6= e(n), f4= u(n), i3 -> u(n-M+1)     */
          f1=f6*f7, f3=dm(i3,m1);  /* f1= STEPSIZE*e(n), f3= u(n-M+1) */
          f4=f3+f4, f3=dm(i3,m1), f12=pm(i8,m8);
                          /* f4= u(n)+u(n-M+1), f3= u(n-M+2), f12=w0(n) */
          f9=f1*f4, f0=dm(i0,m0);
                           /* f9= STEPSIZE*e(n)*[u(n)+u(n-M+1)] f0= u(n-1) */
          f4=f0+f3, f3=dm(i3,m1);
                           /* f4= u(n-1)+u(n-M+2), f3= u(n-M+3)*/
```

```
          lcntr=TAPS-1, do update_weights until lce;
            f9=f1*f4, f8=f9+f12, f0=dm(i0,m0), f12=pm(i8,m8)
                        /* f9= STEPSIZE*e(n)*[u(n-i-1)+u(n-M+2+i)], */
                        /* f8= wi(n+1), f0= u(n-2-i), f12= wi+1(n) */
update_weights:   f4=f0+f3, f3=dm(i3,m1), pm(i9,m8)=f8;
                       /* f4=u(n-2-i)+u(n-M+3+i), f3=u(n-M+4+i), store
wi(n+1) */

          rts(db);
          f8=f9+f12, f0=dm(i0,m2);
                       /* i0 ->u(n+1) location in delay line */
          pm(i9,m8)=f8;
.ENDSEG;
```

Listing 6.7 sylms.asm

6.2.9 Lattice Filter LMS With Joint Process Estimation

The LMS algorithm is implemented using a lattice structure with Joint Process estimation. The utility of a multistage lattice predictor is extended by using the resulting sequence of backward prediction errors, $b_m(n)$, as inputs to a corresponding set of tap coefficients $g_m(n)$, to produce the minimum mean-square estimate of some desired response $d(n)$. This is referred to as a Joint Process estimator since it provides simultaneous calculation of both forward prediction, $f_m(n)$, as well as backward prediction, providing the optimum estimation of the desired response. This joint process LMS technique allows very fast system adaptation for channel equalization and noise cancellation applications. This technique is also known as the gradient lattice-ladder algorithm.

The lattice parameters are described by the relationship

$$k_m(n+1) = K_m(n) + \mu[f_m(n)\, b_{m-1}(n-1) + b_m(n)\, f_{m-1}(n)]$$

where $0 < m \leq M$. The first-stage error is set as

$$e_0(n) = d(n) - b_0(n)\, g_0(n)$$

and subsequent stages set as

$$e_m(n) = e_{m-1}(n) - b_m(n)\, g_m(n)$$

where 0 < m < M. The tap coefficients are updated using the relationship

$$g_m(n+1) = g_m(n) + \mu e_m(n)\, b_m(n)$$

where $0 < m \leq M$.

The output value, $y(n)$, is calculated using the difference equation

$$y(n) = \sum^{M} g_m(n)\, b_m(n)$$

$m = 0$

The gradient lattice algorithm provides significantly faster convergence than the LMS algorithm. Unlike the LMS algorithm, the convergence rate of the gradient lattice algorithm does not depend on the eigenvalue spread of the autocorrelation matrix.

In addition to the filter weight, filter error, and filter output, this routine also provides reflection coefficient, forward prediction error, and backward prediction error as outputs. The lattice algorithm is implemented in the `update_coefs` loop. This loop is `STAGES` long, where `STAGES` is a constant set to the length of the lattice structure (`STAGES = 3` in this example).

6.2.9.1 Code Listing—latlms.asm

```
/
*******************************************************************************

File Name
   LATLMS.ASM

Version
   April 9 1991

Purpose
   Performs the LMS algorithm implemented with a lattice FIR filter structure
with Joint       Process estimation

Equations Implemented
   ****************************************************************
   * 1) f0(n)= b0(n)= u(n)                                        *
   * 2) fm(n)= fm-1(n) - km(n)*bm-1(n-1), 0 <m<= M                *
   * 3) bm(n)= bm-1(n-1) - km(n)*fm-1(n), 0 <m<= M                *
   * 4) km(n+1)= km(n) + STEPSIZE*[fm(n)*bm-1(n-1) + bm(n)*fm-1(n)],*
   *                                  0 <m<= M                    *
   * where u(n)= input sample, n= time index, M= Number of stages *
   *       fm(n)= forward prediction error of the mth stage       *
   *       bm(n)= backward prediction error of the mth stage      *
   *       km(n)= reflection coefficient of the mth stage         *
   *                                                              *
   * 5) e0(n)= d(n) - b0(n)*g0(n)                                 *
   * 6) em(n)= em-1(n) - bm(n)*gm(n), 0 <m<= M                    *
   * 7) gm(n+1)= gm(n) + STEPSIZE*em(n)*bm(n), 0 <=m<= M          *
   * 8) y(n)= SUM [gm(n)*bm(n)] for 0 <=m<= M                     *
   * where d(n)= desired output, y(n)= FIR filter output          *
   *       em(n)= output error at the mth stage                   *
```

(listing continues on next page)

6 Adaptive Filters

```
*       gm(n)= filter weight at the mth stage                    *
*****************************************************************


Calling Parameters
   f0= u(n)= input sample
   f9= d(n)= desired output

Return Values
   f0= bm(n)= mth backward prediction error
   f2= fm(n)= mth forward prediction error
   f6= em(n)= mth output error
   f12= y(n)= filter output
   i8 -> Program Memory Data buffer of the reflection coefficients
   i10 -> Program Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f12, f13

Cycle Count
      LATLMS_ALG: 7+11M  per iteration    , LATLMS_INIT: 16+2M

# PM Locations
      pm code= 36 words, pm data= 2M+1

# DM Locations
   dm data= M+1

*******************************************************************************/

#define   STAGES       3
#define STEPSIZE 0.005

.GLOBAL   latlms_init, latlms_alg;

.SEGMENT/DM      dm_data;
.VAR      bpe_coef[STAGES+1];
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      weights[STAGES+1];
.VAR      ref_coef[STAGES];
.ENDSEG;

.SEGMENT/PM      pm_code;
```

```
latlms_init:     b0=bpe_coef;
          m0=0;
          m1=1;
          l0=STAGES+1;
          b8=ref_coef;
          b10=weights;
          m8=0;
          m9=1;
          l8=STAGES;
          l10=STAGES+1;
          f7=STEPSIZE;
          f0=0.0;
          lcntr=STAGES, do clear_bufs until lce;
            dm(i0,m1)=f0, pm(i8,m9)=f0;
clear_bufs:        pm(i10,m9)=f0;
          rts(db);
          dm(i0,m1)=f0, pm(i10,m9)=f0;
          nop;

latlms_alg:      f10=pass f0, f1=dm(i0,m0), f4=pm(i10,m8);
                 /* f0= b0(n), f10= f0(n), f1= b0(n-1), f4= g0(n) */
          f12=f0*f4, dm(i0,m1)=f0, f5=pm(i8,m9);
                 /* f12= y0(n)= b0(n)*g0(n), store b0(n), f5= k1(n) */
          f13=f1*f5, f6=f9-f12;
                  /* f13= b0(n-1)*k1(n), f6= e0(n) */

          lcntr=STAGES, do update_coefs until lce;
            f3=f0*f7, f2=f10-f13, f8=f4;
                /* f2= fm(n), f3= bm-1(n)*stepsize, f8= gm-1(n) */
            f13=f3*f6, f3=f10;
                /* f13= stepsize*bm-1(n)*em-1(n), f3= fm-1(n) */
            f13=f3*f5, f4=f8+f13, f8=f5;
                /* f4= gm-1(n+1), f13= fm-1(n)*km(n), f8= km(n) */
            f0=f1-f13, pm(i10,m9)=f4;
                 /* f0= bm(n), store gm-1(n+1) */
            f10=f0*f3, f9=dm(i0,m0);
                 /* f10= bm(n)*fm-1(n), f9= bm(n-1) */
            f13=f1*f2, dm(i0,m1)=f0, f4=pm(i10,m8);
                /* f13= bm-1(n-1)*fm(n), store bm(n), f4= gm(n) */
            f13=f0*f4, f1=f10+f13, f10=f6;
                 /* f10= em-1(n)*/
                /* f13= bm(n)*gm(n), f1= fm(n)*bm-1(n-1)+bm(n)*fm-1(n) */
            f12=f12+f13, f5=pm(i8,-1);
```

```
                    /* f12= y(n) of mth stage, f5= km+1(n) */
                f13=f1*f7, f6=f10-f13, f1=f9;
                    /* f6= em(n), f13= stepsize*f1, f1= bm(n-1) */
                f13=f1*f5, f8=f8+f13, f10=f2;
                    /* f13= bm(n-1)*km+1(n), f8= km(n+1), f10= fm(n) */
update_coefs:     f3= f0*f6, pm(i8,2)=f8;
                   /* f3= bm(n)*em(n), store km(n+1) */

            rts(db), f3=f3*f7;
                   /* f3= stepsize*em(n)*bm(n) */
            f4=f3+f4, f3=pm(i8,-1);
                   /* f4= gm(n+1), i8 -> k1(n+1) */
            pm(i10,m9)=f4;
                   /* store gm(n+1) */
.ENDSEG;
```

Listing 6.8 latlms.asm

6.2.10 RLS (Transversal Filter)

The routine calculates the transversal filter output, *y(n)*, much like the lms.asm routine. The Kalman Gain Vector is computed in two steps. The first step computes an intermediate value, *x*. Since *x* deals with the *N*-by-*N* matrix of input autocorrelation data, it is calculated using a nested loop. The second step takes *x* and performs a dot product with the input vector, *u*, in the `mac3` loop, to create $k_0(n)$, the first element in the vector. The subsequent vector values, $k_i(n)$, along with the tap weight vector, $w_i(n)$, are computed in the loop `comp_kn_wn`. Finally, the autocorrelation matrix, *Z*, is updated in a nested loop, where the inner loop updates the rows, and the outer loop updates the columns.

During initialization, the *Z*-matrix is set up using the forgetting factor to create the time decay structure. The forgetting factor is set as a constant, `FORGET_FACT`.

6.2.10.1 Code Listing—rls.asm

```
/
*******************************************************************************

File Name
   RLS.ASM

Version
   April 18 1991

Purpose
   Performs the Recursive Least-Squares (RLS) algorithm
   implemented with a transversal FIR filter structure


Equations Implemented
   ********************************************************************
   * 1) x= s*Z(n-1).u  , where s= 1/forgetting factor, n= time index   *
   *        u= [u(n) u(n-1) ... u(n-N+1)]= input samples in delay line *
   *        Z is an N-by-N matrix, N= number of filter weights         *
   * 2) k= x/[1+u.x] where k is an N-by-1 vector                       *
   * 3) Z(n)= s*Z(n-1) - k.xT , xT is the transpose of vector x        *
   * 4) e(n)= d(n) - w(N-1).u , e(n)= filter "a priori" error signal   *
   *        w= [w0 w1 ... wN-1]= filter weights, d(n)= desired output  *
   * 5) w(n)= w(n-1) + k*e(n)                                          *
   ********************************************************************

Calling Parameters
   f0, f1, f2, f3, f4, f5, f8, f9, f10, f11, f12, f13, f14

Return Values
```

(listing continues on next page)

```
     f13= y(n)= filter output
     f1= e(n)= filter “a priori” error signal
     i0 -> Data Memory Data buffer of the filter weights

Registers Affected
   f0, f1, f2, f4, f7, f8, f9, f12, f13

Cycle Count
   rls_alg: 3N**2+9N+20 per iteration, rls_init: N**2+3N+25

# PM Locations
   pm code= 81 words, pm data= 2N words

# DM Locations
   dm data= N**2+2N

*************************************************************************/

#define   TAPS         5
#define FORGET_FACT    0.9
#define INIT_FACT      1000.

#include “b:\global\macros.h”

.GLOBAL   rls_init, rls_alg;

.SEGMENT/DM      dm_data;
.VAR      weights[TAPS];
.VAR      zmatrix[TAPS*TAPS];
.VAR      xvector[TAPS];
.ENDSEG;

.SEGMENT/PM      pm_data;
.VAR      deline_data[TAPS];
.VAR      kvector[TAPS];
.ENDSEG;

.SEGMENT/PM      pm_code;
rls_init: b0=weights;
          l0=TAPS;
          b1=zmatrix;
          l1=TAPS*TAPS;
          b3=b1;
          l3=l1;
          b2=xvector;
          l2=TAPS;
          m0=1;
          m2=0;
          m3=-3;
          b8=deline_data;
          l8=TAPS;
          b9=kvector;
          l9=TAPS;
```

```
          m8=-1;
          m9=1;
          m11=3;
          f10=2.0;
          f14=1.0;
          f5=1/FORGET_FACT;
          f0=0.0;
          f1=INIT_FACT;
          lcntr=TAPS, do clear_bufs until lce;
            dm(i0,m0)=f0, pm(i8,m9)=f0;
clear_bufs:       dm(i2,m0)=f0, pm(i9,m9)=f0;
          dm(i1,m0)=f1;
          lcntr=TAPS-1, do init_zmatrix until lce;
            lcntr=TAPS, do clear_zel until lce;
clear_zel:         dm(i1,m0)=f0;
init_zmatrix:     dm(i1,m0)=f1;
          rts;

rls_alg:  f4=dm(i0,m0), pm(i8,m8)=f0;
                           /* f4=w0(n-1), store u(n) */
          f8=f0*f4, f4=dm(i0,m0), f0=pm(i8,m8);
                          /* f8=u(n)*w0(n-1), f4=w1(n-1), f0=u(n-1) */
          f12=f0*f4, f4=dm(i0,m0), f0=pm(i8,m8);
                         /* f12=u(n-1)*w1(n-1), f4=w2(n-1), f0=u(n-2) */

          lcntr=TAPS-3, do mac1 until lce;
mac1:              f12=f0*f4, f8=f8+f12, f4=dm(i0,m0), f0=pm(i8,m8);
               /* f12=u(n-i)*wi(n-1), f8=sum of prod,f4=wi+1(n-1), f0=u(n-i-1)
*/

          f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
                /* f12=u(n-N+1)*wN-1(n-1), f4=Z0(0,n-1), f0=u(n) */
          f8=f0*f4, f13=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
                /* f13=y(n), f8= u(n)*Z0(0,n-1), f4=Z0(1,n-1), f0=u(n-1) */
          f12=f0*f4, f1=f9-f13, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f1=e(n), f12=u(n-1)*Z0(1,n-1), f4=Z0(2,n-1), f0=u(n-2) */

          lcntr=TAPS, do compute_xn until lce;
            lcntr=TAPS-3, do mac2 until lce;
mac2:               f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
           /* f12=u(n-i)*Zk(i,n-1), f8=sum of prod,f4=Zk(i+1,n-1), f0=u(n-i-1)
*/

            f12=f0*f4, f8=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f12=u(n-N+1)*Zk(N-1,n-1), f4=Zk+1(0,n-1), f0=u(n) */
            f8=f0*f4, f2=f8+f12, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f2=xk(n),f8=u(n)*Zk+1(0,n-1),f4=Zk+1(1,n-1),
                                                  f0=u(n-1) */
            f12=f0*f4, f4=dm(i1,m0), f0=pm(i8,m8);
               /* f12=u(n-1)*Zk+1(1,n-1), f4=Zk+1(2,n-1), f0=u(n-2) */
compute_xn:  dm(i2,m0)=f2;
               /* store xk(n) */

          f4=dm(i1,m3), f0=pm(i8,m11);
```

(listing continues on next page)

```
                /* i1 -> Z0(0,n-1), i8 -> u(n) */
          f4=dm(i2,m0), f0=pm(i8,m8);
                /* f4= x0(n), f0= u(n) */
          f8=f0*f4, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f8=x0(n)*u(n), f4=x1(n), f0=u(n-1) */
          f12=f0*f4, f8=f8+f14, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f12=x1(n)*u(n-1), f8=1+x0(n)*u(n),f4=x2(n), f0=u(n-
2) */

          lcntr=TAPS-3, do mac3 until lce;
mac3:              f12=f0*f4, f8=f8+f12, f4=dm(i2,m0), f0=pm(i8,m8);
               /* f12=xi(n)*u(n-i), f8=1+sum of prod, f4=xi+1(n),
f0=u(n-i-1) */
          f12=f0*f4, f8=f8+f12, f4=f14;
                /* f12=u(n-N+1)*xN-1(n), f4=1.*/
          f12=f8+f12, f3=dm(i2,m0);
                /* f12=1+u.x, f3= x0(n) */
          DIVIDE(f2,f4,f12,f10,f0);
                /* f2=1/(1+u.x) */
          f0=f2*f3, modify(i8,m9);
                /* f0=k0(n), i8 -> u(n+1) location in delay line */

          lcntr=TAPS-1, do comp_kn_wn until lce;
            f8=f0*f1, f12=dm(i0,m2), pm(i9,m9)=f0;
               /* f8= ki(n)*e(n), f12=wi(n-1), store ki(n) */
            f8=f8+f12, f3=dm(i2,m0);
                /* f8=wi(n), f3=xi+1(n) */
comp_kn_wn:        f0=f2*f3, dm(i0,m0)=f8;
                /* f0=ki+1(n), store wi(n) */

          f8=f0*f1, f12=dm(i0,m2), pm(i9,m9)=f0;
               /* f8=kN-1(n)*e(n), f12=wN-1(n-1), store kN-1(n) */
          f11=f8+f12, f2=dm(i2,m0), f4=pm(i9,m9);
               /* f11= wN-1(n), f2= x0(n), f4= k0(n) */

          f12=f2*f4, f8=dm(i1,m0), f4=pm(i9,m9);
               /* f12= x0(n)*k0(n), f8=Z0(0,n-1), f4= k1(n) */
```

```
          lcntr=TAPS-1, do update_zn until lce;
            f12=f2*f4, f0=f8-f12, f8=dm(i1,m0);
            lcntr=TAPS-2, do update_zrow until lce;
              f3=f0*f5, f0=f8-f12, f8=dm(i1,m0), f4=pm(i9,m9);
update_zrow: f12=f2*f4, dm(i3,m0)=f3;
            f3=f0*f5, f0=f8-f12, f2=dm(i2,m0);
            f0=f0*f5, dm(i3,m0)=f3, f4=pm(i9,m9);
            f12=f2*f4, dm(i3,m0)=f0, f4=pm(i9,m9);
update_zn: f8=dm(i1,m0);
          f12=f2*f4, f0=f8-f12, f8=dm(i1,m0);
          lcntr=TAPS-2, do update_zlastrow until lce;
            f3=f0*f5, f0=f8-f12, f8=dm(i1,m0), f4=pm(i9,m9);
update_zlastrow:  f12=f2*f4, dm(i3,m0)=f3;
          f3=f0*f5, f0=f8-f12, dm(i0,m0)=f11;
          rts(db);
          f0=f0*f5, dm(i3,m0)=f3;
          dm(i3,m0)=f0;
.ENDSEG;
```

Listing 6.9 rls.asm

6.2.11 Testing Shell For Adaptive Filters

The program acts as the signal-generating plant, and can call any of the adaptive algorithms. This module must be edited for use with the specific routine it will call.

6.2.11.1 Code Listing—testafa.asm

```
/
*************************************************************************

File Name
   TESTAFA.ASM

Version
   April 2 1991

Purpose
   This is a testing shell for the Adaptive Filtering Algorithms.
```

(listing continues on next page)

6 Adaptive Filters

```
   This file has to be edited to conform with the algorithm employed.


Equations Implemented

*************************************************************************
   *   This program generates a moving average sequence of real
samples, *
   * and employs the adaptive filter for System Identification based
on  *
   * a transversal FIR filter structure. The generating plant is
*
   * described by the following difference equation:                *
   *               y(n)= x(n-1)-0.5x(n-2)-x(n-3)
*
   * where the plant impulse response is 0, 1, -0.5, -1, 0, 0, ... .
*
   *   The filter is allowed five weight coefficients. The input data
*
   * sequence is a pseudonoise process with a period of 20.
*
   *   This program is also used to test adaptive filters based on
both  *
   * symmetric transversal FIR and lattice FIR structures. The output
*
   * filter error signal is stored in a data buffer named [flt_err]
*
   * for comparison.
*
   *   Place * s in this file with the corresponding code
*

*************************************************************************



*************************************************************************/


#define   SAMPLES      **
   /* ** = 200 for RLS and 1000 for various LMS */

.EXTERN ***_init, ***_alg;
/* *** = lms, llms, nlms, selms, sdlms, sslms, */
                        /*       sylms, latlms, rls .
*/

.SEGMENT/DM     dm_data;
.VAR      flt_err[SAMPLES];
.VAR      input_data[20]= 0.038, -0.901, 0.01, -0.125, -1.275, 0.877,
```

```
-0.881,
                0.930, 1.233, -1.022, 1.522, -0.170, 1.489, -1.469,
                1.068, -0.258, 0.989, -2.891, -0.841, -0.355;
.GLOBAL   flt_err;
.ENDSEG;

.SEGMENT/PM     pm_data;
.VAR      plant_hn[3]= -1.0, -0.5, 1.0;
.ENDSEG;

.SEGMENT/PM     rst_svc;
          dmwait=0x21;
          pmwait=0x21;
          jump begin;
.ENDSEG;

.SEGMENT/PM     pm_code;
begin:          b6=flt_err;
          l6=0;
          b7=input_data;
```

```
                l7=20;
                b15=plant_hn;
                l15=3;
                m15=1;
                m7=1;
                m6=-3;

                call ***_init;/* *** = algorithm codenames listed above */

                lcntr=SAMPLES, do adapt_filter until lce;
                                /* generate data through the plant */
                  f0=dm(i7,m7), f4=pm(i15,m15); /* f0= x(n-3), f4= -1.0 */
                  f8=f0*f4, f0=dm(i7,m7), f4=pm(i15,m15);
                                /* f8= -x(n-3), f0= x(n-2), f4= -0.5 */
                  f12=f0*f4, f0=dm(i7,m7), f4=pm(i15,m15);
                                /* f12= -0.5x(n-2), f0= x(n-1), f4= 1.0 */
                  f12=f0*f4, f8=f8+f12, f0=dm(i7,m7);
                                /* f12= x(n-1), f8= -x(n-3)-0.5x(n-2), f0=
    x(n)= u(n) */
                  f*=f8+f12, modify(i7,m6);
                     /* f*= y(n)= d(n), i7 -> x(n-3) of next iteration */
                     /* f*= f1 for lms, nlms, llms, selms, sdlms,
    sslms,sylms */
                      /* f*= f9 for latlms and RLS */

                  call ***_alg; /* *** = algorithm codenames listed above */

                  dm(i6,m7)=f*; /* store filter error */
                     /* f*= f6 for lms, llms, nlms, sdlms, sylms, latlms */
                     /* f*= f1 for selms, sslms, rls */
                  nop;
    adapt_filter:     nop;
                idle;
    .ENDSEG;
```

Listing 6.10 testafa.asm

6.3 CONCLUSION

Table 6.1 lists the number of instruction cycles and the memory usage relating to the various LMS algorithms implemented with a transversal FIR filter structure. It is interesting to note that the sign-LMS algorithms, originally designed to ease implementation in fixed-function silicon, require more instruction cycles in a programmable DSP due to their sign checking routines.

		Memory Usage		
Algorithm	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
LMS	3N + 8	29	N	N
"Leaky" LMS	3N + 8	30	N	N
Normalized LMS	3N + 16	40	N	N
Sign-Error LMS	3N + 8	29	N	N
Sign-Data LMS	4N + 8	32	N	N
Sign-Sign LMS	4N + 8	31	N	N

Table 6.1 Transversal FIR LMS Performance & Memory Benchmarks For Filters Of Order N

Table 6.2 shows the performance of the LMS algorithm implemented with three different FIR filter structures. The final application will determine the structure used.

		Memory Usage		
Structure	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
Transversal	3N + 8	29	N	N
Symmetric Transversal	2N + 10	39	0.5N	N
Lattice-Ladder	11N + 7	36	2N + 1	N + 1

Table 6.2 LMS Algorithm Benchmarks For Different Filter Structures

Finally, the performance of the RLS and LMS algorithms implemented with a transversal FIR filter structure are compared in Table 6.3. Evidently the fast convergence of the RLS occurs at the expense of instruction cycles.

Algorithm	**Cycles per Iteration**	**PM Code**	**PM Data**	**DM Data**
LMS	3N + 8	29	N	N
RLS	$3N^2 + 9N + 20$	89	2N	$N^2 + 2N$

Fourier Transforms 7

The Discrete Fourier Transform (DFT) is the decomposition of a sampled signal in terms of sinusoidal (complex exponential) components. (If the signal is a function of time, this decomposition results in a frequency domain signal.) The DFT is a fundamental digital signal processing algorithm used in many applications, including frequency analysis and frequency domain processing.

- Frequency analysis provides spectral information for signals that are examined or used in further processing, such as in speech compression.

- Frequency domain processing allows for the efficient computation of the convolution integral (for linear filtering) and of the correlation integral (for correlation analysis).

Because of its computational requirements, the DFT algorithm usually is not used for real time signal processing. Research has developed more efficient ways to compute the DFT. The symmetry and periodicity properties of the DFT are exploited to significantly lower its computational requirements. The resulting algorithms are known collectively as Fast Fourier Transforms (FFTs).

This chapter gives an overview of the DFT algorithm and discusses the features of the ADSP-21000 family architecture that enable fast execution of FFTs. It presents implementations of the DFT when its size is a power of two (a radix-2 FFT) and when its size is a power of four (a radix-4 FFT). Details of these implementations are discussed, including optimization, bit and digit-reversal, coefficient generation, and inverse transforms. This chapter also provides benchmarks and code listings of implementations of the algorithms.

7 Fourier Transforms

7.1 COMPUTATION OF THE DFT

The DFT resembles the correlation operation in that it measures the similarity of an unknown signal with a complex exponential. The resulting spectrum yields the complex information (phase and amplitude) for N frequencies. The resulting values are commonly called *frequency bins* because they fill up with amplitude information for each frequency.

An *N-point* DFT computes a sequence $X(k)$ of N complex-valued numbers given another sequence of data $x(n)$ of length N according to the formula

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi nk/N} \qquad k = 0 \text{ to } N-1$$

To simplify the notation, the complex-valued phase factor $e^{-j2\pi nk/N}$ is usually defined as W_N where:

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

Direct computation of the DFT requires approximately N^2 complex multiplications and N^2 complex accumulations. The DFT is inefficient because it does not exploit the symmetry and periodicity properties of the phase factor W_N. These properties are

Symmetry property: $W_N^{k+N/2} = -W_N^k$
Periodicity property: $W_N^{k+N} = W_N^k$

The FFT algorithms take advantage of the symmetry and periodicity properties to greatly reduce the number of calculations that the DFT requires. In an FFT implementation the real and imaginary components of W_N are frequently called *twiddle factors*. These sine and cosine constants are usually precalculated and stored in a table.

7.1.1 Derivation Of The Fast Fourier Transform

The basis of the FFT is that a DFT can be divided into smaller DFTs. A radix-2 FFT divides the FFT 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.

An N-point DFT can be computed by executing 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 $(N/2)^2 = N^2/4$ complex multiplications and complex 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%. Each of these smaller DFTs can be divided in half, yielding four $N/4$-point DFTs. If the N-point calculations are divided into smaller DFTs until only two-point DFTs remain, the total number of complex multiplications and additions is reduced to $Nlog^2N$. 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%.

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, resulting in a combination of four-point DFTs.

Two methods are used repeatedly to split the DFTs into smaller (two-point or four-point) core calculations:

- The decimation-in-time (DIT) FFT divides the input (time) sequence into two groups: one of even samples and the other of odd samples.
- The decimation-in frequency (DIF) FFT divides the output (frequency) sequence into even and odd portions.

See the references listed at the end of this chapter for more detail on the derivation of these and other FFT algorithms.

7 Fourier Transforms

7.1.2 Butterfly Calculations

The two-point DFT at the core of a radix-2 DIT FFT is called a butterfly calculation. The equations for the radix-2 DIT butterfly are:

$$X0' = X0 + (CX1 + SY1)$$
$$X1' = X0 - (CX1 + SY1)$$
$$Y0' = Y0 + (CY1 - SX1)$$
$$Y1' = Y0 - (CY1 - SX1)$$

The variables *Xn* and *Yn* represent the real and imaginary parts, respectively, of a data sample. The flow graph representing this butterfly calculation is shown in Figure 7.1. Figure 7.2 shows a complete 32-point radix-2 DIT FFT, which is the FFT structure used in the ADSP-210xx radix-2 implementation shown later in this chapter.

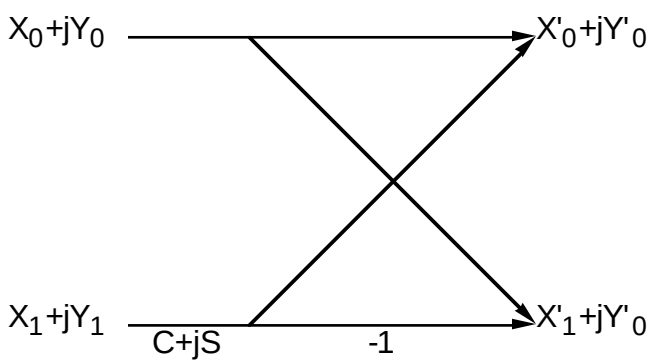


Figure 7.1 Flow Graph Of Butterfly Calculation

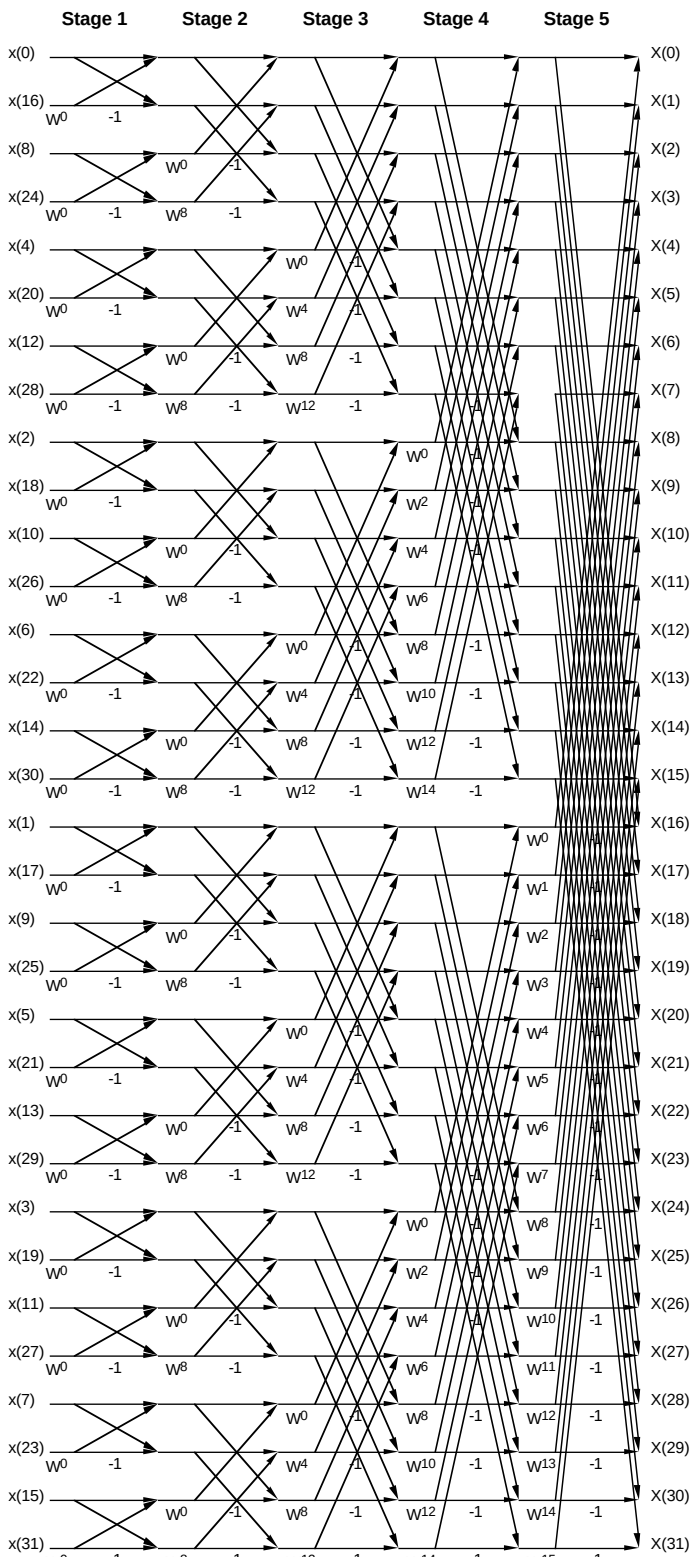


Figure 7.2 32-Point Radix-2 DIT FFT

7 Fourier Transforms

The radix-2 FFT is partitioned into repeated structures called *stages* and *groups*, as well as individual butterflies. There are $log_2(N)$ stages in an N-point radix-2 FFT. This 32-point FFT example contains five stages. The number of groups decreases by 1/2 per stage. The number of butterflies per group increases by two per stage. The number of butterflies per stage is a constant, $N/2$.

Note that in this implementation, the output array *X(k)* is in a normal sequential order and the input array x(n) is in a *bit-reversed* order. This non-sequential order is a result of the FFT algorithm's repeated subdivision of the data sequences. Therefore, the locations of the data, locations of the twiddle factors, or locations of the both may be scrambled (in bit-reversed order). Bit-reversal is an addressing technique used in FFT calculations to order the results sequentially. A bit-reversed address is generated by reversing the order of the bits in a binary representation of the address. (The address is read right to left, with the least significant bit becoming the most significant bit.) Bit-reversal of a data array is accomplished by swapping each position in a data array with the position of its corresponding bit-reversed address.

7.2 ARCHITECTURAL FEATURES FOR FFTS

The ADSP-21000 family of processors has many architectural features that allow for fast implementation of the FFT. These features not only benefit FFT implementations but are also common requirements of all digital signal processing.

The FFT butterfly calculation requires fast arithmetic computations and the generation of a complex sequence of addresses. It also requires a high transfer rate of data between memory and the computation units. The ADSP-210xx meets these needs with its capability to execute a multiplication, an addition and a subtraction, two memory transfers, and two address pointer modifications, all in a single instruction cycle. The parallel addition and subtraction instructions on the ADSP-210xx support the frequent $A + B$ and $A - B$ operations used within butterfly calculations.

A bit-reversed addressing mode is available on the ADSP-210xx to allow low- or no-overhead unscrambling of the FFT results.

The Radix-4 FFT algorithm, in particular, requires that the computation units keep many intermediate values. The ADSP-210xx's 16-location register file prevents bottlenecks from forming between the computation units and memory. It can store all intermediate values needed for an efficient 14 cycle radix-4 butterfly.

The nested structure of a memory-efficient FFT requires a powerful looping support mechanism. The ADSP-210xx allows up to six nested DO UNTIL loops without overhead. Counter decrement, testing and branching all take place in parallel with computations.

7.3 COMPLEX FFTS

The program `FFTRAD2.ASM` (Listing 7.2) is an implementation of a complex input radix-2 FFT for the ADSP-21020 , while the program `FFTRAD4.ASM` (Listing 7.3) is an implementation of the complex input radix-4 FFT. To execute these programs on other ADSP-21000 family members, you must only change the initialization code at the reset vector, rst_svc.

These implementations are optimized to minimize execution time, with only a small increase in code space as a tradeoff. The optimization process uses several techniques, among them are use of simplified butterflies for certain sections of the flow graph and the elimination of overhead through the consolidation of short iterating butterfly loops.

7.3.1 Architecture File Requirements

The FFT bit-reversal operation places a specific requirement on the absolute address of a data array, which is accessed with bit-reversed addressing. The starting address of the data array must be an integer multiple of the FFT size, (*0, N, 2N,* ...). To force an array to start at a specific location, define a segment in the architecture file that starts at an absolute location, and then assign the array to that segment from within the assembly file.

Listing 7.1 shows an example architecture file for the ADSP-21020 which supports bit-reversal. It assigns the segments `dm_rdat` and `dm_idat` to support bit-reversal of the real and imaginary data for FFTs up to 16K points long.

The radix-2 program in Listing 7.2 reads the input arrays, `redata` and `imdata`, using bit-reversed addressing. The `.SEGMENT` directive is used to place these arrays at absolute locations in the `dm_rdat` and `dm_idat` segments.

The radix-4 program in Listing 7.3 writes to the output arrays, `refft` and `imfft`, using bit-reversed addressing. The `.SEGMENT` directive is used to place these arrays at absolute locations in the `dm_rdat` and `dm_idat` segments.

7.3.2 The Radix-2 DIT FFT Program

For maximum efficiency, this implementation of the radix-2 FFT is broken up into a minimum of five stages: the first two stages, the middle stages, the second to the last, and the last stage. Since there is a minimum of five stages, the smallest FFT calculable by this implementation is $N = 2^5 = 32$ points.

The first two stages are performed together in a single loop. They are combined into a loop that implements what is essentially a radix-4 butterfly. The twiddle factor for this butterfly is $W_N^0 = cos(0) - j\ sin(0) = 1$; therefore no multiplications are required. The first two stages execute quickly because there is no group overhead and because the no-multiply radix-4 butterfly requires an equivalent of two cycles per radix-2 butterfly compared to the normal four cycles required by the middle stages.

The addressing of data input to all butterflies is from the bottom of the working array to the top, in other words, they are accessed backwards.

The flow graph for the FFT is structured so that all butterflies within a particular group require the same twiddle factors. As a result, a new value must be read into the register file only at the start of each new group, which improves the FFT's performance by about 20%.

Most of the FFT execution time is spent within the middle stage butterflies. The ADSP-210xx architecture executes these butterflies very efficiently. They require only four cycles each to execute, because a multiply and add operation is performed in all four cycles and a subtract is done on two of the four cycles. The two address generators support four memory reads and four memory writes in each butterfly with a total of eight address pointer updates.

The second to the last stage has only two butterflies per group. It is implemented with one loop to reduce the overhead between groups. Each butterfly takes only 4.5 cycles in this stage.

The last stage has only one butterfly per group, is implemented within one loop, and takes five cycles per butterfly.

7.3.3 The Radix-4 DIF FFT Program

This implementation of the radix-4 FFT executes about 10% faster than the radix-2 program. The only drawbacks are that the number of points in the radix-4 FFT is restricted to a power of four, the program is a bit longer, and that there are N/2 more twiddle factors required.

Instead of bit-reversal, digit-reversal is required for the radix-4 algorithm. Digit-reversal for a radix-4 FFT is defined by taking the addresses in groups of 4 bits and by reversing the order of the groups. Normally the generation of digit-reversed addresses requires either a lookup table in memory or the use the barrel shifter. There is a very simple trick to perform digit-reversal with little or no overhead. Swap the results of the middle two nodes in all radix-4 butterflies, and write the results in a data array in bit-reversed order. Then the ADSP-210xx's built in bit reversal feature can generate a result in normal order.

The radix-4 program is broken up into a minimum of three stages: the first stage, the middle stages, and the last stage. Since there is a minimum of three stages, the smallest FFT calculable in this implementation is N=43=64 points.

The first stage uses a simplified butterfly that requires no multiplications and takes only eight cycles to execute. The middle and last stages are identical and require 14 cycles per butterfly. The last stage is separate so that bit reverse mode can be enabled. The real data is then bit-reversed as it is written out to the `refft` output array resulting in normal order data. The imaginary array is then bit-reversed by reading it with DAG2 and writing the result with the DAG1 I0 register. This also results in normal order data.

The following equations are used in the ADSP-210xx implementation of the radix-4 DIF butterfly:

$$X_0' = X_0 + (C_3X_2 + S_3Y_2) + (C_2X_3 + S_2Y_3) + (C_1X_1 + S_1Y_1)$$
$$X_1' = X_0 + (C_3X_2 + S_3Y_2) - (C_2X_3 + S_2Y_3) + (C_1X_1 + S_1Y_1)$$
$$Y_0' = Y_0 + C_3Y_2 - (C_3X_2 + S_3Y_2) + (C_1Y_1 - S_1X_1) + (C_2Y_3 - S_2X_3)$$
$$Y_1' = Y_0 + C_3Y_2 - (C_3X_2 + S_3Y_2) - (C_1Y_1 - S_1X_1) + (C_2Y_3 - S_2X_3)$$
$$Y_2' = Y_0 - C_3Y_2 - (C_3X_2 + S_3Y_2) + (C_2X_3 + S_2Y_3) - (C_1X_1 + S_1Y_1)$$
$$Y_3' = Y_0 - C_3Y_2 - (C_3X_2 + S_3Y_2) - (C_2X_3 + S_2Y_3) - (C_1X_1 + S_1Y_1)$$
$$X_2' = X_0 - (C_3X_2 + S_3Y_2) + (C_1Y_1 - S_1X_1) - (C_2Y_3 - S_2X_3)$$
$$X_3' = X_0 - (C_3X_2 + S_3Y_2) - (C_1Y_1 - S_1X_1) - (C_2Y_3 - S_2X_3)$$

See the references at the end of this chapter for more detail on radix-4 FFT structures.

7.3.4 FFTs On The ADSP-21060

The ADSP-21060 can bit-reverse with the DAG2 I8 register. By incorporating both the DAG1 and DAG2 bit-reversal in a parallel instruction, approximately N cycles are saved in an N-point FFT. Both the `FFTRAD2.ASM` and `FFTRAD4.ASM` programs can be modified easily to take advantage of this feature.

7.3.5 FFT Twiddle Factor Generation

The radix-2 FFT requires two twiddle factor tables stored in memory. These tables consist of 1/2 cycle of a sine wave and a cosine wave. Each table is *N/2* samples long. A C language utility for generating the two tables, called `TWIDRAD2.C`, is shown in Listing 7.4.

The radix-4 FFT also requires two twiddle factor tables stored in memory. They are different from the radix-2 tables in that they are stored in a bit-reversed-like order. These tables consist of *3N/4* samples. A C language utility for generating the two tables called `twidrad4.c`, is shown in Listing 7.5. Radix-4 tables generated for an N-point FFT also can be used for any radix-4 FFT of length less than N, because of the bit-reversed nature of the radix-4 tables.

7.4 INVERSE COMPLEX FFTs

The inverse relationship for obtaining a sequence from its DFT is called the inverse DFT (IDFT). The transformation is described by the equation

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)\, ej2\pi nk/N \qquad n = 0 \text{ to } N-1$$

You can use the complex FFTs algorithms described in this chapter to implement both the IDFT and the DFT. The only difference between the two transformations is the normalization factor *1/N* and the phase signs of the twiddle factors. Consequently, an FFT algorithm for computing the DFT are converted into an IFFT algorithm by using a reversed (upside down) twiddle factor table. Because the cosine table is symmetrical about zero, only the sine table must be reversed.

Instead of reversing the twiddle factors, a simpler method to convert the FFT into an IFFT is to swap the complex parts of input and output arrays with this algorithm:

1. Swap the input data array's real and imaginary parts.
2. Run the forward FFT.
3. Swap the result's real and imaginary parts.
4. Scale the result by *1/N*.

The data swapping steps need no overhead if they are incorporated into data transfers that are already required.

7 Fourier Transforms

7.5 BENCHMARKS

Complex Radix-2 FFT with Bit -Reversal

Butterfly Calculation Performance:

First 2 Stages	8 cycles per 4 (radix-2) butterflies
Middle Stages	4 cycles per butterfly
2nd to Last Stage	9 cycles per 2 butterflies
Last Stage	5 cycles per butterfly group

Memory Usage:

pm code = 158 words, pm data = 1.5 * *N* words, dm data = 3.5 * *N* words

Complex Radix-4 FFT with Digit-Reversal

Butterfly Calculation Performance:

First Stage	8 cycles per radix-4 butterfly
Other Stages	14 cycles per radix-4 butterfly

Memory Usage:

pm code = 192 words, pm data = 1.75 * *N* words, dm data = 3.75 * *N* words

7.6 CODE LISTINGS

7.6.1 FFT.ACH - Architecture File

```
!   FFT.ACH
!   Example (ADSP-21020) architecture file for FFTs. Supports proper multiple
!   of N base addresses for bit reversing of real and imaginary arrays of sizes
!  up to 16K words.

.SYSTEM   fft;
.PROCESSOR = ADSP21020;

.SEGMENT /ROM  /BEGIN=0x000008 /END=0x00000F /PM rst_svc;

.SEGMENT /ROM  /BEGIN=0x000100 /END=0x003FFF /PM pm_code;
.SEGMENT /RAM  /BEGIN=0x004000 /END=0x00BFFF /PM pm_data;

.SEGMENT /RAM  /BEGIN=0x00000000 /END=0x00003FFF /DM dm_rdat;
.SEGMENT /RAM  /BEGIN=0x00004000 /END=0x00007FFF /DM dm_idat;
.SEGMENT /RAM  /BEGIN=0x00008000 /END=0x0000BFFF /DM dm_data;

.ENDSYS;
```

Listing 7.1 FFT.ACH

7.6.2 FFTRAD2.ASM - Complex Radix2 FFT

```
/*_____________________________________________________________________
FFTRAD2.ASM    ADSP-21020 Radix-2 DIT Complex Fast Fourier Transform

Calculates a radix-2 FFT. The FFT length (N) must be a power of 2 and a
minimum of 32 points. Input data is not destroyed during the course of this
routine. The input and output arrays are normal ordered. The real array is
stored in DM, the imaginary array is stored in PM. The real twiddle factors
are in an N/2 long Cosine table stored in PM, and the imaginary twiddle
factors are in an N/2 long Sine Table in stored in DM. The twiddle factors
are generated by the program TWIDRAD2.

To implement a inverse FFT, one only has to (1) swap the real and imaginary
of the incoming data, (2) take the forward FFT, (3) swap the real and
imaginary of the outgoing data, and (4) scale the data by 1/N.

Version:
          10-SEP-90 Original
          25-APR-91 Revision
           26-MAY-93,  in FSTAGE drain pipe without dummy dm access
           14-DEC-93, Cleaned up format, added benchmarks

Calling Parameters:
    pm(cosine[N/2]) - real twiddle factors from TWIDRAD2 program
    dm(sine[N/2])   - imaginary twiddle factors from TWIDRAD2 program
    dm(redata[N])    - real input array, bitreversed to a working array
    dm(imdata[N])    - imaginary input array, bitreversed to a working array

    (Note: Because the bit reversed address mode is used with the arrays
    refft and imfft, they must start at addresses that are integer
    multiples of the length (N) of the transform, (i.e. 0,N,2N,3N,...).
    This is accomplished by specifing two segments starting at those addresses
    in the architecture file and placing the variables alone in their
    respective segments. These addresses must also be reflected in the
    preprocessor variables ORE and OIM in bit reversed format.)

Return Values:
    dm(refft[N])   - real working array and output
    dm(imfft[N])   - imaginary working array and output

Altered Registers:
    Most I, M, L, and R registers.
    Three levels of loop nesting.
```

```
Benchmarks: Radix-2, complex with bit reversal

    FFT Length    cycles   ms @ 25 MHz CLK  ms @ 33 MHz CLK  ms @ 40 MHz CLK
    ______    ___   ________  ________  ________
         64         1002        .040            .031            .021
        128         2088        .083            .063            .052
        256         4486        .179            .135            .112
        512         9764        .391            .293            .244
       1024        21314        .853            .639            .533
       2048        46432       1.857           1.393           1.161
       4096       100734       4.029           4.022           2.518
       8192       217504       8.700           6.535           5.437

   First 2 Stages    - 8 cycles per 4 (radix-2) butterflies
   Middle Stages     - 4 cycles per butterfly
   2nd to Last Stage - 9 cycles per 2 butterflies
   Last Stage        - 5 cycles per butterfly group

Memory Usage:
    pm code = 158 words, pm data = 1.5*N words, dm data = 3.5*N words
_____________________________________________________________________________*/

/* Include for symbolic definition of system register bits */
#include “def21020.h”

/*_________The constants below must be changed for different length FFTs______
N        = number of points in the FFT, must be a power of 2
STAGES   = log2(N)
BRMODIFY  = bitrev(32 bit N/2)
ORE      = bitrev(32 bit addr of input real in dm), addr is 0,N,2N,3N,...
OIM      = bitrev(32 bit addr of input imag in dm), addr is 0,N,2N,3N,...
_____________________________________________________________________________*/
#define N               256
#define STAGES          8
#define BRMODIFY        0x01000000
#define ORE             0x00000000
#define OIM             0x00020000

/*________These constants are independent of the number of points____________*/
#define BFLY8           4     /*Number of bttrfly in a group of 8*/

.SEGMENT/DM             dm_data;
.VAR    sine[N/2]= “ts2.dat”;     /*imag twiddle factors, from TWIDRAD2 */
.VAR    refft[N];                 /* real result */
.GLOBAL refft;
.ENDSEG;

.SEGMENT/DM             dm_rdat; /* This segment is an integer multiple of N */
.VAR    redata[N]= “inreal.dat”; /* input real array */
.GLOBAL redata;
.ENDSEG;
```

(listing continues on next page)

7 Fourier Transforms

```
.SEGMENT/DM             dm_idat; /* This segment is an integer multiple of N */
.VAR    imdata[N]= “inimag.dat”; /* input image array */
.GLOBAL imdata;
.ENDSEG;

.SEGMENT/PM             pm_data;
.VAR    cosine[N/2]= “tc2.dat”;   /* real twiddle factors, from TWIDRAD2 */
.VAR    imfft[N];                 /* imag result */
.GLOBAL imfft;
.ENDSEG;

/*_____________ADSP-21020 reset vector test call of fft_____________________*/
.SEGMENT/PM             rst_svc; /* program starts at the reset vector */
        pmwait=0x0021;           /*pgsz=0,pmwtstates=0,intrn.wtstates only*/
        dmwait=0x8421;           /*pgsz=0,pmwtstates=0,intrn.wtstates only*/
        call fftrad2;
stop:   idle;
.ENDSEG;

.SEGMENT/PM             pm_code;
/*_____________________________begin FFT_____________________________________*/
fftrad2:
        bit set mode1 BR0;  /* enable bit reverse of i0 */
        B0=OIM;             /* Points to input imaginary array */
        l0=0;
        m0=BRMODIFY;        /* Modifier for bitreverse counter*/

        b8=imfft;           /* Working array and output */
        l8=N;
        m8=1;

/*First read imag and bit reverse to pm space*/
        f0=dm(i0,m0);
        lcntr=N-1, do pmbr until lce;   /* Bit reverse from dm to pm */
pmbr:     f0=dm(i0,m0), pm(i8,m8)=f0;
        pm(i8,m8)=f0;                   /* Do last transfer */

/*Now do bitrev real within first two stages*/
        b0=ORE;         /* Points to input real array to be read in */
                        /* bit reversed order */
        b2=refft;
        l2=N;       /* Circ pointer limits loopend pointer overflow */
        m1=1;       /* This loop increments forward +1*/

        b8=imfft;
        l8=N;       /* Circ pointer limits loopend pointer overflow */
        b10=imfft;
        l10=N;      /* Circ pointer limits loopend pointer overflow */
```

```
    /*Do the first two stages (actually a radix-4 FFT stage)*/

                                              f0=dm(i0,m0),   f1=pm(i8,m8);
                                              f2=dm(i0,m0),   f3=pm(i8,m8);
                f0=f0+f2,       f2=f0-f2,     f4=dm(i0,m0),   f5=pm(i8,m8);
                f1=f1+f3,       f3=f1-f3,     f6=dm(i0,m0),   f7=pm(i8,m8);
                f4=f6+f4,       f6=f6-f4;
                f5=f5+f7,       f7=f5-f7;
                f8=f0+f4,       f9=f0-f4;
                f10=f1+f5,      f11=f1-f5;

lcntr=N/4-1,  do FSTAGE until lce;    /* do N/4 simple radix-4 butterflies */
                f12=f2+f7,      f13=f2-f7,    f0=dm(i0,m0),   f1=pm(i8,m8);
                f14=f3+f6,      f15=f3-f6,    f2=dm(i0,m0),   f3=pm(i8,m8);
                f0=f0+f2,       f2=f0-f2,     f4=dm(i0,m0),   f5=pm(i8,m8);
                f1=f1+f3,       f3=f1-f3,     f6=dm(i0,m0),   f7=pm(i8,m8);
                f4=f6+f4,       f6=f6-f4,     dm(i2,m1)=f8,   pm(i10,m8)=f10;
                f5=f5+f7,       f7=f5-f7,     dm(i2,m1)=f12,  pm(i10,m8)=f14;
                f8=f0+f4,       f9=f0-f4,     dm(i2,m1)=f9,   pm(i10,m8)=f11;
FSTAGE:         f10=f1+f5,      f11=f1-f5,    dm(i2,m1)=f13,  pm(i10,m8)=f15;

                f12=f2+f7,      f13=f2-f7; /* change on 5/26/93, drain pipe*/
                f14=f3+f6,      f15=f3-f6; /* without out of range dm xfer*/
                                              dm(i2,m1)=f8,   pm(i10,m8)=f10;
                                              dm(i2,m1)=f12,  pm(i10,m8)=f14;
                                              dm(i2,m1)=f9,   pm(i10,m8)=f11;
                                              dm(i2,m1)=f13,  pm(i10,m8)=f15;

    /*middle stages loop */

        bit clr mode1 BR0;     /*finished with bitreversal*/

        b0=refft;
        l0=N;       /* Circ pointer limits loopend pointer overflow */
        b1=sine;
        l1=@sine;

        b9=cosine;
        l9=@cosine;
        b11=imfft;
        l11=N;      /* Circ pointer limits loopend pointer overflow */

        m0=-BFLY8;
        m1=-N/8;
        m2=-BFLY8-1;
        m9=-N/8;
        m11=-1;

        r2=2;
        r3=-BFLY8;            /*initializes m0,10 - incr for butterf branches*/
        r5=BFLY8;             /*counts # butterflies per a group             */
        r9=(-2*BFLY8)-1;      /*initializes m12 - wrap around to next grp + 1*/
        r10=-2*BFLY8;         /*initializes m8 - incr between groups   */
        r13=-BFLY8-1;         /*initializes m2,13 - wrap to bgn of 1st group */
        r15=N/8;                /*# OF GROUPS IN THIRD STAGE*/
```

(listing continues on next page)

```
        f1=dm(i1,m1),   f7=pm(i9,m9); /*set pointers to tables to 1st coeff. */

lcntr=STAGES-4, do end_stage until lce; /*# OF STAGES TO BE HANDLED = LOG2N-4*/
        m8=r10;
        m10=r3;
        m12=r9;
        i0=refft+N-1;
        i2=refft+N-1;
        i8=imfft+N-1;
        i10=imfft+N-1;
        i11=imfft+N-1;
        r15=r15-r2,     m13=r13;        /*CALCULATE # OF CORE */
                                        /*BFLIES/GROUP IN THIS STAGE*/

                                        f0=dm(i1,m1),   f7=pm(i8,m8);
f12=f0*f7,                              f6=dm(i0,m0),   f1=pm(i9,m9);
f8=f1*f6,                                               modify(i11,m10);
f11=f1*f7,                                              f7=pm(i8,m8);
f14=f0*f6,  f12=f8+f12,                 f8=dm(i0,m0);
f12=f0*f7,  f13=f8+f12, f10=f8-f12,     f6=dm(i0,m0);

/*Each iteration does another set of bttrflys in each group*/

lcntr=r5,   do end_group until lce;    /*# OF BUTTERFLIES/GROUP IN THIS STAGE*/

/*core butterfly loop*/

lcntr=r15,  do end_bfly until lce;     /*Do a butterfly in each group - 2*/
    f8=f1*f6,   f14=f11-f14,                    dm(i2,m0)=f10,  f9=pm(i11,m8);
    f11=f1*f7,  f3=f9+f14,      f9=f9-f14,      dm(i2,m0)=f13,  f7=pm(i8,m8);
    f14=f0*f6,  f12=f8+f12,                     f8=dm(i0,m0),   pm(i10,m10)=f9;
end_bfly:
    f12=f0*f7,  f13=f8+f12,     f10=f8-f12,     f6=dm(i0,m0),   pm(i10,m10)=f3;

/*finish up last bttrfly and set up for next stage*/

f8=f1*f6,               f14=f11-f14,    dm(i2,m0)=f10,  f9=pm(i11,m8);
f11=f1*f7,  f4=f9+f14,  f9=f9-f14,      dm(i2,m0)=f13,  f14=pm(i8,m11);
f14=f0*f6,  f12=f8+f12,                 f8=dm(i0,m2),   pm(i10,m10)=f9;
            f13=f8+f12, f10=f8-f12,     f0=dm(i1,m1),   f7=pm(i8,m8);/*dm:sin*/
                        f14=f11-f14,    dm(i2,m0)=f10,  f9=pm(i11,m12);
  /*start on next butterfly in each group*/
f12=f0*f7,  f3=f9+f14,  f9=f9-f14,      f6=dm(i0,m0),   f1=pm(i9,m9);/*pm:cos*/
f8=f1*f6,                               dm(i2,m2)=f13,  pm(i10,m10)=f4;
f11=f1*f7,                                              pm(i10,m10)=f9;
f14=f0*f6,  f12=f8+f12,                 f8=dm(i0,m0),   f7=pm(i8,m8);
end_group:
f12=f0*f7,  f13=f8+f12, f10=f8-f12,     f6=dm(i0,m0),   pm(i10,m13)=f3;
```

```
            r4=r15+r2,          i1=b1;       /*PREPARE R4 FOR #OF BFLIES CALC*/
            r15=ashift r4 by -1;             /*# OF BFLIES/GRP IN NEXT STAGE*/
            r4=-r15,            i9=b9;
            m1=r4;                                /*update inc for sin & cos */
            m9=r4;
            r5=ashift r5 by 1,  f1=dm(i1,m1);     /*update # bttrfly in a grp*/
            r3=-r5;                               /*   inc for bttrfly branch*/
            r13=r3-1,           m0=r3;            /*   wrap to 1st grp       */
            r10=ashift r3 by 1, f7=pm(i9,m9);     /*   inc between grps      */
end_stage:  r9=r10-1,           m2=r13;           /*   wrap to grp +1        */

/*_________ next to last stage__________*/
        m1=-2;                  /*modifier to sine table pntr    */
        m8=r10;                 /*incr between groups            */
        m9=-2;                  /*modifier to cosine table pntr  */
        m10=r3;                 /*incr between bttrfly branches  */
        m12=r9;                 /*wrap around to next grp + 1    */
        m13=r13;                /*wrap to bgn of 1st group       */

        i0=refft+N-1;
        i1=sine+(N/2)-2;        /*pntr to 1st sine coeff         */
        i2=refft+N-1;
        i8=imfft+N-1;
        i9=cosine+(N/2)-2;      /*pntr to 1st cosine coeff       */
        i10=imfft+N-1;
        i11=imfft+N-1;
                                        f0=dm(i1,m1),   f7=pm(i8,m8);
f12=f0*f7,                              f6=dm(i0,m0),   f1=pm(i9,m9);
f8=f1*f6,                                               modify(i11,m10);
f11=f1*f7,                                              f7=pm(i8,m12);
f14=f0*f6, f12=f8+f12,                  f8=dm(i0,m0);
f12=f0*f7, f13=f8+f12,  f10=f8-f12,     f6=dm(i0,m0);

/*Do the N/4 butterflies in the two groups of this stage*/

lcntr=N/4,  do end_group2 until lce;
    f8=f1*f6,                   f14=f11-f14,    dm(i2,m0)=f10,  f9=pm(i11,m8);
    f11=f1*f7,  f3=f9+f14,      f9=f9-f14,      dm(i2,m0)=f13,  f1=pm(i9,m9);
    f14=f0*f6,  f12=f8+f12,                     f8=dm(i0,m2),   pm(i10,m10)=f9;
                f13=f8+f12,     f10=f8-f12,     f0=dm(i1,m1),   f7=pm(i8,m8);
    f12=f0*f7,                  f14=f11-f14,    f6=dm(i0,m0),   f9=pm(i11,m12);

    f8=f1*f6,   f3=f9+f14,      f9=f9-f14,      dm(i2,m0)=f10,  pm(i10,m10)=f3;
    f11=f1*f7,                                  dm(i2,m2)=f13,  pm(i10,m10)=f9;
    f14=f0*f6, f12=f8+f12,                      f8=dm(i0,m0),   f7=pm(i8,m12);
end_group2:
    f12=f0*f7, f13=f8+f12,      f10=f8-f12,     f6=dm(i0,m0),   pm(i10,m13)=f3;
```

(listing continues on next page)

```
/*   The last stage      */

        m0=-N/2;
        m2=-N/2-1;
        m10=m0;
        m13=m2;
        i0=refft+N-1;
        i1=sine+(N/2)-1;        /*pntr to 1st sine coeff         */
        i2=refft+N-1;
        i8=imfft+N-1;
        i9=cosine+(N/2)-1;      /*pntr to 1st cosine coeff       */
        i10=imfft+N-1;
        i11=imfft+N-1;
        m1=-1;                  /*modifiers to coeff tables      */
        m9=-1;

/*start first bttrfly*/
                                        f0=dm(i1,m1),   f7=pm(i8,m11);
f12=f0*f7,                              f6=dm(i0,m0),   f1=pm(i9,m9);
f8=f1*f6,                                               modify(i11,m10);
f11=f1*f7;
f14=f0*f6, f12=f8+f12,                  f8=dm(i0,m2),   f9=pm(i11,m11);

/*do N/2 bttrflys in the last stage, one butterfly per group*/

lcntr=N/2,  do last_stage until lce;
                f13=f8+f12,     f10=f8-f12,     f0=dm(i1,m1),  f7=pm(i8,m11);
    f12=f0*f7,  f14=f11-f14,                    f6=dm(i0,m0),  f1=pm(i9,m9);
    f8=f1*f6,   f3=f9+f14,      f15=f9-f14,     dm(i2,m0)=f10, f9=pm(i11,m11);
    f11=f1*f7,                                  dm(i2,m2)=f13, pm(i10,m10)=f15;
last_stage:
    f14=f0*f6, f12=f8+f12,                      f8=dm(i0,m2),  pm(i10,m13)=f3;

rts;    /*finished*/
/*_____________________________________________________________________________*/
.ENDSEG;
```

Listing 7.2 fftrad2.asm

7.6.3 FFTRAD4.ASM - Complex Radix-4 FFT

```
/*_____________________________________________________________________________
FFTRAD4.ASM     ADSP-21020 Radix-4 Complex Fast Fourier Transform

This routine performs a complex, radix 4 Fast Fourier Transform (FFT). The FFT
length (N) must be a power of 4 and a minimum of 64 points. The real part of
the input data is placed in DM and the complex part in PM. This data is
destroyed during the course of the computation. The real and complex output of
the FFT is placed in separate locations in DM.

Since this routine takes care of all necessary address digit-reversals, the
input and output data are in normal order. The digit reversal is accomplished
by using a modified radix 4 butterfly throughout which swaps the inner two
nodes resulting with bit reversed data. The digit reversal is completed by
bit reversing the real data in the final stage and then bit reversing the
imaginary so that it ends up in DM.

To implement an inverse FFT, you only have to (1) swap the incoming datas real
and imaginary parts, (2) run the forward FFT, (3) swap the outgoing datas
real and imaginary parts and (4) scale the data by 1/N.

For this routine to work correctly, the program "twidrad4.C" must be used to
generate the special twiddle factor tables for this program.

Author:     Karl Schwarz & Raimund Meyer, Universitaet Erlangen Nuernberg
Version:
          25-APR-91, Rev. 1
          18-JUN-91, Rev. 2
          14-DEC-93, Cleaned up comments

Calling Parameters:
    costwid table at DM : cosine    length 3*N/4
    sintwid table at PM : sine      length 3*N/4
    real  input  at DM : redata     length N, normal order
    imag  input  at PM : imdata     length N, normal order

Return Values:
    real  output at DM : refft      length N, normal order
    imag  output at DM : imfft      length N, normal order

    (Note:  Because the bit reversed addressing mode is used with the arrays
    refft and imfft, they must start at addresses that are integer
    multiples of the length (N) of the transform, (i.e. 0,N,2N,3N,...).
    This is accomplished by specifying two segments starting at those addresses
    in the architecture file and placing the variables alone in their
    respective segments. These addresses must also be reflected in the
    preprocessor variables ORE and OIM in bit reversed format.)

Altered Registers:
    All I, M, L and R registers.
    Three levels of looping.
```

(listing continues on next page)

7 Fourier Transforms

```
Benchmarks: radix-4, complex with digit reversal

    FFT Length   cycles   ms @ 25 MHz CLK  ms @ 33 MHz CLK  ms @ 40 MHz CLK
    ______   ___   _______  _______  _______
        64          920        .037             .028             .023
       256         4044        .162             .121             .101
      1024        19245        .770             .577             .481
      4096        90702       3.628            2.722            2.268
     16384       419434      16.777           12.583           10.486

    First Stage  -   8 cycles per radix-4 butterfly
    Other Stages -  14 cycles per radix-4 butterfly

Memory Usage:
    pm code = 192 words, pm data = 1.75*N words, dm data = 3.75*N words
_____________________________________________________________________________*/

/*________The constants below must be changed for different length FFTs________
    N       Number of points in FFT.  Must be a power of four, minimum of 64.
    STAGES  Set to log4(N) or (log(N)/log(4))
    OST     = bitrev(32 bit N/2)
    ORE     = bitrev(32 bit addr of output real in dm), addr is 0,N,2N,3N,...
    OIM     = bitrev(32 bit addr of output imag. in dm), addr is 0,N,2N,3N,...
_____________________________________________________________________________*/
#define N       256
#define STAGES  4
#define OST     0x01000000
#define ORE     0x00000000
#define OIM     0x00020000

/* include for symbolic definition of system regster bits */
#include “def21020.h”

.SEGMENT/DM             dm_data;
.VAR    cosine[3*N/4]=”tc4.dat”;/*Cosine twiddle factors, from FFTTR4TBL prog*/
.VAR    redata[N]=”inreal.dat”; /* Input real data */
.GLOBAL redata;
.ENDSEG;

.SEGMENT/DM             dm_rdat; /* this segment is an integer multiple of N */
.VAR    refft[N];               /* Output real data */
.GLOBAL refft;
.ENDSEG;

.SEGMENT/DM             dm_idat; /* this segment is an integer multiple of N */
.VAR    imfft[N];               /* Output imaginary data */
.GLOBAL imfft;
.ENDSEG;

.SEGMENT/PM             pm_data;
.VAR    sine[3*N/4]=”ts4.dat”;  /* Sine twiddle factors, from FFTTR4TBL prog*/
.VAR    imdata[N]=”inimag.dat”; /* Input imaginary data */
.GLOBAL imdata;
.ENDSEG;
```

```
.SEGMENT/PM             rst_svc; /* program starts at the reset vector */
        pmwait=0x0021;           /*pgsz=0,pmwtstates=0,intrn.wtstates only*/
        dmwait=0x008421;         /*pgsz=0,dmwtstates=0,intrn.wtstates only*/
        call    fft;
stop:   idle;
.ENDSEG;

.SEGMENT/PM             pm_code;
fft:
/*_______first stage radix-4 butterfly without twiddles______*/
                        i0=redata;
                        i1=redata+N/4;
                        i2=redata+N/2;
                        i3=redata+3*N/4;
                        i4=i0;
                        i5=i1;
                        i6=i2;
                        i7=i3;
                        m0=1;
                        m8=1;
                        i8=imdata;
                        i9=imdata+N/4;
                        i10=imdata+N/2;
                        i11=imdata+3*N/4;
                        i12=i8;
                        i13=i9;
                        i14=i10;
                        i15=i11;
                        l0 = 0;
                        l1 = l0;
                        l2 = l0;
                        l3 = N;     /* circular to prevent pointer overflow */
                        l4 = l0;
                        l5 = l0;
                        l6 = l0;
                        l7 = l0;
                        l8 = l0;
                        l9 = l0;
                        l10 = l0;
                        l11 = N;    /* circular to prevent pointer overflow */
                        l12 = l0;
                        l13 = l0;
                        l14 = l0;
                        l15 = l0;
                                                f0=dm(i0,m0),   f1=pm(i8,m8);
                                                f2=dm(i2,m0),   f3=pm(i10,m8);
                f0=f0+f2,       f2=f0-f2,       f4=dm(i1,m0),   f5=pm(i9,m8);
                f1=f1+f3,       f3=f1-f3,       f6=dm(i3,m0),   f7=pm(i11,m8);
                f4=f6+f4,       f6=f6-f4;
                f5=f5+f7,       f7=f5-f7;
                f8=f0+f4,       f9=f0-f4;
                f10=f1+f5,      f11=f1-f5;
```

(listing continues on next page)

7 Fourier Transforms

```
lcntr=N/4,      do fstage until lce;    /* do N/4 simple radix-4 butterflies */
                f12=f2+f7,      f13=f2-f7,      f0=dm(i0,m0),   f1=pm(i8,m8);
                f14=f3+f6,      f15=f3-f6,      f2=dm(i2,m0),   f3=pm(i10,m8);
                f0=f0+f2,       f2=f0-f2,       f4=dm(i1,m0),   f5=pm(i9,m8);
                f1=f1+f3,       f3=f1-f3,       f6=dm(i3,m0),   f7=pm(i11,m8);
                f4=f6+f4,       f6=f6-f4,       dm(i4,m0)=f8,   pm(i12,m8)=f10;
                f5=f5+f7,       f7=f5-f7,       dm(i5,m0)=f9,   pm(i13,m8)=f11;
                f8=f0+f4,       f9=f0-f4,       dm(i6,m0)=f12,  pm(i14,m8)=f14;
fstage:
                f10=f1+f5,      f11=f1-f5,      dm(i7,m0)=f13,  pm(i15,m8)=f15;

                l3=l0;      /* pointer overflow only problem in 1st stage */
                l11=l0;     /* pointer overflow only problem in 1st stage */

/*_____________Middle stages with radix-4 main butterfly_____________________*/

/* m0=1 and m8=1 is still preset */
                        m1=-2;          /* reverse step for twiddles */
                        m9=m1;
                        m2=3;           /* forward step for twiddles */
                        m10=m2;
                        m5=4;           /* first there are 4 groups */
                        r2=N/16;        /* with N/16 bflies in each group*/
                        r3=N/16*3;      /* step to next group */

lcntr=STAGES-2, do mstage until lce;    /* do STAGES-2 stages */

                        i7=cosine;      /* first real twiddle */
                        i15=sine;       /* first imag twiddle */

                        r8=redata;
                        r9=imdata;

                        i0=r8;          /* upper real      path */
        r10=r8+r2;      i8=r9;          /* upper imaginary path */
                        i1=r10;         /* second real input path */
        r10=r10+r2,     i4=r10;         /* second real output path */
                        i2=r10;         /* third real input path */
        r10=r10+r2,     i5=r10;         /* third real output path */
                        i3=r10;         /* fourth real input path */
        r10=r9+r2,      i6=r10;         /* fourth real output path */
                        i9=r10;         /* second imag input path */
        r10=r10+r2,     i12=r10;        /* second imag output path */
                        i10=r10;        /* third imag input path */
        r10=r10+r2,     i13=r10;        /* third imag output path */
                        i11=r10;        /* fourth imag input path */
                        i14=r10;        /* fourth imag output path */
                        m4=r3;
                        m12=r3;
        r4=r3+1,        m6=r2;
                        m3=r4;
        r2=r2-1,        m11=r4;
                        m7=r2;
```

```
lcntr=m5,       do mgroup until lce;    /* do m5 groups */
                                                f0=dm(i7,m0),   f5=pm(i9,m8);
f8=f0*f5,                                       f4=dm(i1,m0),   f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5,                                      f0=dm(i7,m0),   f5=pm(i11,m8);
f13=f1*f4,      f12=f9+f12,                     f4=dm(i3,m0),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13;
f13=f1*f5;
f9=f0*f5,       f8=f8+f13,                      f0=dm(i7,m1),   f5=pm(i10,m8);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,     f4=dm(i2,m0),   f1=pm(i15,m9);
f11=f0*f4;
f13=f1*f5,                      f6=f9-f13;
f9=f0*f5,       f13=f11+f13,                    f11=dm(i0,0);
f13=f1*f4,      f8=f11+f13,     f10=f11-f13;
          /*___________Do m7 radix-4 butterflies___________*/
lcntr=m7,       do mr4bfly until lce;
                                f13=f9-f13,     f4=dm(i1,m0),   f5=pm(i9,m8);
                f2=f2+f6,       f15=f2-f6,      f0=dm(i7,m0),   f1=pm(i15,m8);
f8=f0*f4,       f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,0);
f12=f1*f5,      f9=f9+f13,      f11=f9-f13,     f13=f2;
f8=f0*f5,       f12=f8+f12,                     f0=dm(i7,m0),   f5=pm(i11,m8);
f13=f1*f4,      f9=f9+f13,      f6=f9-f13,      f4=dm(i3,m0),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13,      dm(i0,m0)=f3,   pm(i8,m8)=f9;
f13=f1*f5,      f11=f11+f14,    f7=f11-f14,     dm(i4,m0)=f7,   pm(i12,m8)=f6;
f9=f0*f5,       f8=f8+f13,                      f0=dm(i7,m1),   f5=pm(i10,m8);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,     f4=dm(i2,m0),   f1=pm(i15,m9);
f11=f0*f4,      f3=f10+f15,     f8=f10-f15,                     pm(i13,m8)=f11;
f13=f1*f5,                      f6=f9-f13,      dm(i6,m0)=f8,   pm(i14,m8)=f7;
f9=f0*f5,       f13=f11+f13,                    f11=dm(i0,0);
mr4bfly:
f13=f1*f4,      f8=f11+f13,     f10=f11-f13,    dm(i5,m0)=f3;
          /*___________End radix-4 butterfly_____________*/
/*                       dummy for address update      *             * */
                                f13=f9-f13,     f0=dm(i7,m2),   f1=pm(i15,m10);
                f2=f2+f6,       f15=f2-f6,      f0=dm(i1,m4),   f1=pm(i9,m12);
                f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,0);
                f9=f9+f13,      f11=f9-f13,     f0=dm(i2,m4);
                f9=f9+f2,       f6=f9-f2,       f0=dm(i3,m4),   f1=pm(i10,m12);
                                                dm(i0,m3)=f3,   pm(i8,m11)=f9;
                f11=f11+f14,    f7=f11-f14,     dm(i4,m3)=f7,   pm(i12,m11)=f6;
                f3=f10+f15,     f8=f10-f15,                     pm(i13,m11)=f11;
                                                dm(i6,m3)=f8,   pm(i14,m11)=f7;
mgroup:                                         dm(i5,m3)=f3,   f1=pm(i11,m12);

                                r3=m4;
                                r1=m5;
                                r2=m6;
        r3=ashift r3 by -2;             /* groupstep/4 */
        r1=ashift r1 by 2;              /* groups*4 */
                                m5=r1;
mstage: r2=ashift r2 by -2;             /* butterflies/4 */

/*_____________________Last radix-4 stage_____________________________________*/
/* Includes bitreversal of the real data in dm */
```

(listing continues on next page)

```
        bit set mode1 BR0;              /* bitreversal in i0 */
/* with: m0=m8=1 preset */
        i4=redata;                      /* input */
        i1=redata+1;
        i2=redata+2;
        i3=redata+3;
        i0=ORE;  /* real output array base must be an integer multiple of N */
        m2=OST;
        i7=cosine;
        i8=imdata;                      /* input */
        i9=imdata+1;
        i10=imdata+2;
        i11=imdata+3;
        i12=imdata;                     /* output */
        i15=sine;
        m1=4;
        m9=m1;
                                                f0=dm(i7,m0),   f5=pm(i9,m9);
f8=f0*f5,                                       f4=dm(i1,m1),   f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5,                                      f0=dm(i7,m0),   f5=pm(i11,m9);
f13=f1*f4,      f12=f9+f12,                     f4=dm(i3,m1),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13;
f13=f1*f5;
f9=f0*f5,       f8=f8+f13,                      f0=dm(i7,m0),   f5=pm(i10,m9);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,     f4=dm(i2,m1),   f1=pm(i15,m8);
f11=f0*f4;
f13=f1*f5,                      f6=f9-f13;
f9=f0*f5,       f13=f11+f13,                    f11=dm(i4,m1);
f13=f1*f4,      f8=f11+f13,     f10=f11-f13;
            /*________Do N/4-1 radix-4 butterflies_______*/
lcntr=N/4-1, do lstage until lce;
                                f13=f9-f13,     f4=dm(i1,m1),   f5=pm(i9,m9);
                f2=f2+f6,       f15=f2-f6,      f0=dm(i7,m0),   f1=pm(i15,m8);
f8=f0*f4,       f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,m9);
f12=f1*f5,      f9=f9+f13,      f11=f9-f13,     f13=f2;
f8=f0*f5,       f12=f8+f12,                     f0=dm(i7,m0),   f5=pm(i11,m9);
f13=f1*f4,      f9=f9+f13,      f6=f9-f13,      f4=dm(i3,m1),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13,      dm(i0,m2)=f3,   pm(i12,m8)=f9;
f13=f1*f5,      f11=f11+f14,    f7=f11-f14,     dm(i0,m2)=f7,   pm(i12,m8)=f6;
f9=f0*f5,       f8=f8+f13,                      f0=dm(i7,m0),   f5=pm(i10,m9);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,     f4=dm(i2,m1),   f1=pm(i15,m8);
f11=f0*f4,      f3=f10+f15,     f8=f10-f15,                     pm(i12,m8)=f11;
f13=f1*f5,                      f6=f9-f13,      dm(i0,m2)=f3,   pm(i12,m8)=f7;
f9=f0*f5,       f13=f11+f13,                    f11=dm(i4,m1);
lstage:
```

```
f13=f1*f4,      f8=f11+f13,     f10=f11-f13,    dm(i0,m2)=f8;

                                f13=f9-f13;
                f2=f2+f6,       f15=f2-f6;
                f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,m9);
                f9=f9+f13,      f11=f9-f13,     dm(i0,m2)=f3;
                f9=f9+f2,       f6=f9-f2,       dm(i0,m2)=f7;
                                                                pm(i12,m8)=f9;
                f11=f11+f14,    f7=f11-f14,                     pm(i12,m8)=f6;
                f3=f10+f15,     f8=f10-f15,                     pm(i12,m8)=f11;
                                                dm(i0,m2)=f3,   pm(i12,m8)=f7;
                                                dm(i0,m2)=f8;

/*________Do the bitreversal of the imaginary part from pm to dm___________*/
        i8=imdata;
        i0=OIM;  /* image output array base must be an integer multiple of N */
        f0=pm(i8,m8);

        lcntr=N-1, do pmbr until lce;    /* do N-1 bitreversals */
pmbr:     dm(i0,m2)=f0, f0=pm(i8,m8);

        rts (db);
        dm(i0,m2)=f0;
        bit clr mode1 BR0;              /* no bitreversal in i0 any more */
.ENDSEG;
```

Listing 7.3 fftrad4.asm

7.6.4 TWIDRAD2.C - Radix2 Coefficient Generator

```
/*_____________________________________________________________________
TWIDRAD2.C  Twiddle Factor Generator for ADSP-21020 Radix-2 FFT

Written:  18-FEB-91, Analog Devices
Modified:  25-MAR-91
          12-JUN-91
_____________________________________________________________________*/

#include <stdio.h>
#include <math.h>

main()
{
  int i, n;
  double freq, c, s;
  double pi;
  FILE *s_file;
  FILE *c_file;
  char filename1[25];
  char filename2[25];

/* initialize pi */
  pi = 4.0*atan(1.0);

  printf("%c%c%c%c",27,91,50,74); /*clear screen*/
  printf("%c%c%c",27,91,72);      /*curser home*/
  printf("_____________Radix-2 FFT Twiddle Factor Generator_____________\n");
  printf("Generates a 1/2 a sine and cosine for use with the ADSP-21020\n");
  printf("radix-2 FFT code. Use with the radix-2 FFT - FFTRAD2.ASM only.\n");
  printf("12-JUN-91, Analog Devices\n");
  printf("_____________________________________________________________\n");
```

```
    printf("Enter the number of points -> ");
    scanf(" %d",&n);
    n=n/2;

    printf("\nEnter the real twiddles filename ———> ");
    scanf(" %s",filename1);
    c_file=fopen(filename1,"w");

    printf("\nEnter the imaginary twiddles filename —> ");
    scanf(" %s",filename2);
    s_file=fopen(filename2,"w");

    freq=2.0*pi*0.5/(double)n;
    for (i=0; i <= n-1; i++)
    {
      s=sin((double)i * freq);
      c=cos((double)i * freq);
      fprintf(s_file,"%22.14e\n",s);
      fprintf(c_file,"%22.14e\n",c);
      printf("%d : %22.14e : %22.14e\r",i,s,c);
    }

    fclose(s_file);
    fclose(c_file);
    printf("\nFinished\n");
  }
```

Listing 7.4 twidrad2.c

7 Fourier Transforms

7.6.5 TWIDRAD4.C - Radix4 Coefficient Generator

```
/*_____________________________________________________________________________
TWIDRAD4.C
 C program to generate the radix-4 twiddle factor table for the
 fast FFT-program RAD4.ASM for the ADSP21020.

 Based on MATLAB-File by Karl Schwarz, Erlangen-Nuernberg Universitaet

 Written  12-MAR-1991  Analog Devices
 Revised  25-MAR-1991  Analog Devices
_____________________________________________________________________________*/

#include <stdio.h>
#include <math.h>
#define   MAX_LEN      16384

int b[MAX_LEN];
main() {
    double tc;
    double ts;
    double  k1, k2, k3;           /* coeff for table calculation      */
    double  pi;
    char    cosfn[20];  /* strings for file names */
    char    sinfn[20];
    int       length,             /* length of FFT */
           iter,           /* length/4          */
           i, j, k;
    FILE *s_file, *c_file;


/* initialize pi */
    pi=4.0*atan(1.0);

/* User interface */

    printf("%c%c%c%c",27,91,50,74);  /* clears ibm screen */
    printf("%c%c%c",27,91,72);             /* homes ibm cursor  */

printf("\n                          FFTR4TBL \n");
printf("   Cosine and Sine Table generator for FFTRAD4.ASM \n");
printf("\nThis program generates the cosine and sine tables for the fast\n");
printf("FFT program FFTRAD4.ASM on the ADSP-21020.  The length of the FFTs\n");
printf("can be any power of four equal or greater than 64. This program is\n");
printf("configured to generate tables for FFTs as long as %d.\n\n",MAX_LEN);

    printf("\n Enter the FFT length (max %d): ",MAX_LEN);
    scanf(" %d",&length);

    printf("\n Name of Cosine table to be created: ");
    scanf(" %s",cosfn);
    c_file=fopen(cosfn,"w");
```

```
    printf(" Name of Sine table to be created: ");
    scanf(" %s",sinfn);
    s_file=fopen(sinfn,"w");

/* Start Calculations  */
    if (length > 1024)
    printf("\n\n     Thinking hard . . .");
    else
    printf("\n\n     Thinking . . .");

    iter = length/4;

    /* generate array for bit reversed addressing */
    for (i=1,j=1;i<=iter;i++) {
      b[i-1] = j-1;
      k = iter/2;
      while (k<j && k!=0) {
        j =j-k;
        k = k/2;
      }
      j += k;
    }

    /* calculate tables */

    k1 = (2*pi/(double)length);
    k2 = (2*pi/(double)length)*2;
    k3 = (2*pi/(double)length)*3;

    for (i=0;i<iter;i++) {
      tc = cos(b[i]*k1);
      ts = sin(b[i]*k1);
      fprintf(c_file,"%22.14e\n",tc);
      fprintf(s_file,"%22.14e\n",ts);

      tc = cos(b[i]*k3);
      ts = sin(b[i]*k3);
      fprintf(c_file,"%22.14e\n",tc);
      fprintf(s_file,"%22.14e\n",ts);

      tc = cos(b[i]*k2);
      ts = sin(b[i]*k2);
      fprintf(c_file,"%22.14e\n",tc);
      fprintf(s_file,"%22.14e\n",ts);
    }

   fclose(c_file);
   fclose(s_file);
   printf("\n    Done!\n\n");
}
```

Listing 7.5 twidrad4.c

7.7 REFERENCES

[ADI92] Analog Devices Inc., A. Mar, ed. 1992. Digital Signal Processing Applications Using *The ADSP-2100 Family*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

[BRIGHAM74] Brigham, E. O. 1974. *The Fast Fourier Transform*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

[BURRUS85] Burrus, C.S. and T.W. Parks. 1985. *DFT and Convolution Algorithms*. New York, NY: John Wiley and Sons.

[DUDGEON84] Dudgeon, D. and R. Mersereau. 1984. *Multidimensional Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[EMBREE91] Embree, P. and B. Kimble. 1991. *C Language Algorithms For Digital Signal Processing.* , Englewood Cliffs, NJ: Prentice-Hall, Inc.

[GONZALEZ77] Gonzalez, T. and P. Wintz. 1977. *Digital Image Processing*. Reading, MA: Addision-Wesley Publishing Company.

[HKMSHD88] Hakimmashhadi, H. 1988. "Discrete Fourier Transform and FFT." *Signal Processing Handbook.* New York, NY: Marcel Dekker, Inc.

[HAYKIN83] Haykin, S. 1983. *Communications Systems.* New York: John Wiley and Sons.

[OPPENHEIM75] Oppenheim, A.V. and R.W. Schafer. 1975. *Digital Signal Processing*. Engelwood Cliffs, NJ: Prentice-Hall, Inc.

[PROAKIS88] Proakis, J.G. and D.G. Manolakis. 1988. *Introduction To Digital Signal Processing*, New York, NY: Macmillan Publishing Company.

[RABINER75] Rabiner, L.R. and B. Gold. 1975. *Theory And Applications Of Digital Signal Processing.* Engelwood Cliffs, NJ: Prentice-Hall, Inc.

Graphics 8

A computer graphics display system synthesizes pictures of real or imaginary objects from computer-based models. A computer graphics system consists of several subsystems, or layers, which may be implemented in hardware or software. Each layer processes the input passed to it from the layer above it and feeds its output to the input of the next lower layer. The image is constructed as it passes through the pipeline. The bottom layer is a hardware display device that transforms the constructed image into a viewable picture, such as a picture on a CRT monitor or line plotter drawings on paper.

8.1 3-D GRAPHICS LINE ACCEPT/REJECT

Depending on an image's size and position, a display device may be incapable of displaying the entire image all at once, so device-dependent constraints must be put on the image before it is rendered. One of the processes that determines which portion of an image resides within the "view volume" of the display device is called the *3-D Graphics Line Accept/Reject. Algorithm.*

This subroutine performs trivial accept/reject checking on lines. The input is a list in data memory of line endpoint coordinates—a pair of *x, y, z* coordinates for each line. The subroutine performs clipping to a window defined by a list in program memory of the *Xmin, Xmax, Ymin, Ymax, Zmin,* and *Zmax* coordinates of the viewing volume. The output of this subroutine is a display list of lines for the line renderer.

The subroutine calculates the *outcodes* for both of the endpoints of each line. The outcode is a six-bit value that represents this truth table:

OUTCODE bit	*Meaning*
5	$Z > Zmax$
4	$Z < Zmin$
3	$Y > Ymax$
2	$Y < Ymin$
1	$X > Xmax$
0	$X < Xmin$

The algorithm performs these six tests and sets the outcode bits accordingly. If an outcode is zero, the point lies within the view volume.

Each line is classified into one of three categories:

trivial acceptance	both endpoints inside view volume, the line can be simply rendered
trivial rejection	both endpoints outside the view volume, the line is not rendered at all
neither	the line must be clipped prior to rendering

See [FOLEY90] p. 145-149 for further information about 3-D Graphics Line Accept/Reject.

8.1.1 Implementation

Each point is subjected to six test to determine its outcode. After the six compares are performed on each point, the Compare Accumulator (upper six bits of ASTAT) have these values:

OUTCODE bit	*ASTAT bit*	*Meaning*
5	31	*Z1 >Zmax*
4	30	*Z1 < Zmin*
3	29	*Y1 > Ymax*
2	28	*Y1 < Ymin*
1	27	*X1 > Xmax*
0	26	*X1 < Xmin*

A point is only in the view volume if all six of these bits are zero.

Both outcodes must be all zeroes for the line to be trivially accepted. If the logical OR of the outcodes is not equal to all zeroes, the line can be trivially rejected.

When the line is neither trivially accepted nor rejected, it is clipped. There will be at least one bit and at most three bits in the outcode which are non-zero. Each non-zero bit requires the line to be clipped to its corresponding window coordinate as shown Table 10.1 above. For example, if outcode bit 2 is non-zero, the line must be clipped to the *Ymin* boundary.

When a line requires clipping, the program calls the `clipline` subroutine. Calling `clipline` at this point is preferable to clipping after trivially accepting/rejecting all the lines since the coordinates do not have to be fetched again and the outcodes do not have to be recalculated.

The `clipline` subroutine is application dependent and so is not given here. Keep in mind when you write `clipline` that the endpoints of the unclipped line and the outcodes for those endpoints are available in registers zero through seven. The `clipline` routine should write a two (2) to the display list and the new endpoints to the clipped endpoint list. DAG1 index register I1 points to the display list and I2 points to the clipped endpoint list.

After each line is classified, and possibly clipped, a value is added to the display list. A zero (0) in the display list represents a trivially rejected line, a one (1) represents a trivially accepted line, and a two (2) indicates the line required clipping. For lines that required clipping, the clipped endpoint values (two points, i.e. six values) are written to the clipped endpoint list.

8 Graphics

8.1.2 Code Listing

```
/
*****************************************************************************

File Name
   accrej.asm:

Version
    1.0

Purpose
   Performs trivial accept/reject checking on lines.

Equations Implemented

Calling Parameters
   REGISTER FILE
   r7     number of lines to be examined
   r12    preload with 0x3F (six ones)
   r13    preload with 2
   r14    preload with 1
   r15    preload with 0

   DAG1 (Data Memory)
   i0     pointer to line endpoint list (X1,Y1,Z1,X2,Y2,Z2,...)
   m0     +1
   i1     pointer to display list
   i2     pointer to clipped endpoint list

   DAG2 (Program Memory)
   i8     pointer to Xmin,Xmax,Ymin,Ymax,Zmin,Zmax
   l8     6
   m8     +1

Return Values

Registers Affected
   r0-r9

Cycle Count
   25*N + 28  worst case (all lines require clipping, 24 cache misses),
   21*N + 1   best case  (all lines trivially accepted, PMDAs in cache)

   where N = number of lines, and  worst case does NOT include time to perform
clipping

   840ns to classify one line @ 40ns instruction cycle
   1.2 million lines classified/second @ 40ns instruction cycle

# PM Locations
     45 instructions + 6 data

# DM Locations
     10*N, where N = Number of lines to be examined.
*****************************************************************************/
```

```
.EXTERN     clipline;
.EXTERN     accrej;

/* Defines related to the Compare Accumulator (CACC) */
#define CACC6_MSK       0xfc000000                  /* top 6 bits of CACC   */
#define test_cacc(msk)  bit tst astat msk           /* CACC test instr      */

/* static registers */
#define REJECT          r15     /* = 0      */
#define ACCEPT          r14     /* = 1      */
#define CLIP            r13     /* = 2      */
#define SIX_ONES        r12     /* = 0x3f   */

/* First Endpoint Coordinates */
#define X1              f0
#define Y1              f1
#define Z1              f2

/* Second Endpoint Coordinates */
#define X2              f3
#define Y2              f4
#define Z2              f5

/* Outcodes for Each Endpoint */
#define OC1             r6
#define OC2             r7

/* indices to lists */
#define DSP_LIST        i1
#define CLIP_LIST       i2

.SEGMENT /pm pm_code;
.GLOBAL accrej;
accrej:
        /* Perform OUTCODE #1 Comparisons */
                        X1=dm(i0,m0),   f8=pm(i8,m8);
        comp (f8,X1),                   f8=pm(i8,m8);       /* X1 < Xmin ?  */
        comp (X1,f8),   Y1=dm(i0,m0),   f8=pm(i8,m8);       /* X1 > Xmax ?  */
        comp (f8,Y1),                   f8=pm(i8,m8);       /* Y1 < Ymin ?  */
        comp (Y1,f8),   Z1=dm(i0,m0),   f8=pm(i8,m8);       /* Y1 > Ymax ?  */
        comp (f8,Z1),                   f8=pm(i8,m8);       /* Z1 < Zmin ?  */
        comp (Z1,f8),   X2=dm(i0,m0),   f8=pm(i8,m8);       /* Z1 > Zmax ?  */

        /* r7 contains number of lines - decrement it for loop */
        r7 = r7 - 1, OC1 = astat;   /* get outcode 1, store in a register   */
        OC1 = fext OC1 by 26:6;     /* extract 6 MSBs                       */
```

(listing continues on next page)

```
        /* START OF LOOP */
        lcntr = r7, do arlp-1 until lce;
            /* Perform OUTCODE #2 Comparisons */
            comp (f8,X2),                   f8=pm(i8,m8);   /* X2 < Xmin ?  */
            comp (X2,f8),   Y2=dm(i0,m0),   f8=pm(i8,m8);   /* X2 > Xmax ?  */
            comp (f8,Y2),                   f8=pm(i8,m8);   /* Y2 < Ymin ?  */
            comp (Y2,f8),   Z2=dm(i0,m0),   f8=pm(i8,m8);   /* Y2 > Ymax ?  */
            comp (f8,Z2),                   f8=pm(i8,m8);   /* Z2 < Zmin ?  */
            comp (Z2,f8);                                   /* Z2 > Zmax ?  */

            OC2 = astat;            /* get outcode 2, store in a register   */
            OC2 = fext OC2 by 26:6; /* extract 6 MSBs                       */

            r9 = OC1 and OC2;
reject:
            if ne jump nxt (db);
                if ne dm(i1,m0) = REJECT;
                r9 = OC1 or OC2;
accept:
            if eq dm(i1,m0) = ACCEPT;

clipit:
            if ne call clipline;

nxt:        /* Perform OUTCODE #1 Comparisons */
            /* NOTE: Cannot pre-fetch X1 because the old value in X1 must   */
            /*       be preserved until after clipline has been called      */
                        X1=dm(i0,m0),   f8=pm(i8,m8);
            comp (f8,X1),                   f8=pm(i8,m8);   /* X1 < Xmin ?  */
            comp (X1,f8),   Y1=dm(i0,m0),   f8=pm(i8,m8);   /* X1 > Xmax ?  */
            comp (f8,Y1),                   f8=pm(i8,m8);   /* Y1 < Ymin ?  */
            comp (Y1,f8),   Z1=dm(i0,m0),   f8=pm(i8,m8);   /* Y1 > Ymax ?  */
            comp (f8,Z1),                   f8=pm(i8,m8);   /* Z1 < Zmin ?  */
            comp (Z1,f8),   X2=dm(i0,m0),   f8=pm(i8,m8);   /* Z1 > Zmax ?  */

            OC1 = astat;
            OC1 = fext OC1 by 26:6;

arlp:       /* END OF LOOP */
```

```
        comp (f8,X2),                   f8=pm(i8,m8);       /* X2 < Xmin ?  */
        comp (X2,f8),   Y2=dm(i0,m0),   f8=pm(i8,m8);       /* X2 > Xmax ?  */
        comp (f8,Y2),                   f8=pm(i8,m8);       /* Y2 < Ymin ?  */
        comp (Y2,f8),   Z2=dm(i0,m0),   f8=pm(i8,m8);       /* Y2 > Ymax ?  */
        comp (f8,Z2),                   f8=pm(i8,m8);       /* Z2 < Zmin ?  */
        comp (Z2,f8);                                       /* Z2 > Zmax ?  */

        OC2 = astat;
        OC2 = fext OC2 by 26:6;

        r9 = OC1 and OC2;

rej2:
        if ne rts (db);
            if ne dm(i1,m0) = REJECT;
            r9 = OC1 or OC2;
acc2:
        if eq dm(i1,m0) = ACCEPT;

clip2:
        if ne call clipline;

        rts;

.ENDSEG;
```

Listing 8.1 accej.asm

8 Graphics

8.2 CUBIC BEZIER POLYNOMIAL EVALUATION

To generate a one-dimensional image that describes a three-dimensional object, a mathematical relationship must be developed. Traditionally, three-dimensional curved surfaces are graphically represented by either of two methods: as functions of the standard *x*, *y*, and *z* variables, or as a function of another parameter (such as *t*). A problem inherent with the parametric method is that it may require a representation of an infinite slope. Conversely, since the parametric method describes a surface as a third order polynomial, the slope problem is easily handled. Both the Bezier and B-spline polynomial use parametric equations to represent curved surfaces.

The cubic Bezier polynomial uses four points to define a curve (or surface): both endpoints and two other points not on the curve which define tangents on the curve's endpoints. These points (the *p* terms in the following equation, are referred to as control points or *knots* (see Figure 8.1).

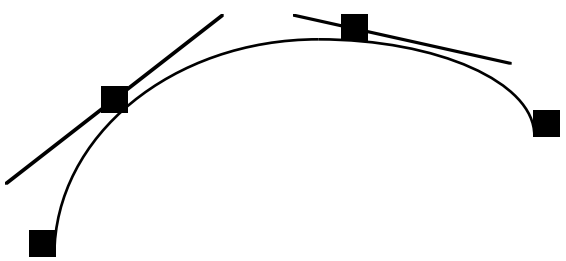


Figure 8.1 Cubic Bezier Polynomial

The assembly language routine in this section evaluates the Cubic Bezier polynomial

$$x(t) = p_{1x}(1-t)^3 + 3p_2t(1-t)^2 + 3p_{3x}\,t^2(1-t) + p_{4x}t^3$$

after it has been factored to the form

$$x(t) = (t-1)[\,-(t-1)^2p_{1x} + 3t\,[(t-1)p_{2x} - p_{3x}t\,)]] + p_{4x}t^3$$

See [FOLEY90] p. 519 - 521 for further information.

8.2.1 Implementation

The code given is not looped. For a similar example taking advantage of hardware-supported DO loops, see `bspline.asm`.

The register definitions used in the code below refer to the various sections of the factored equation being assembled, as shown in Figure 8.2:

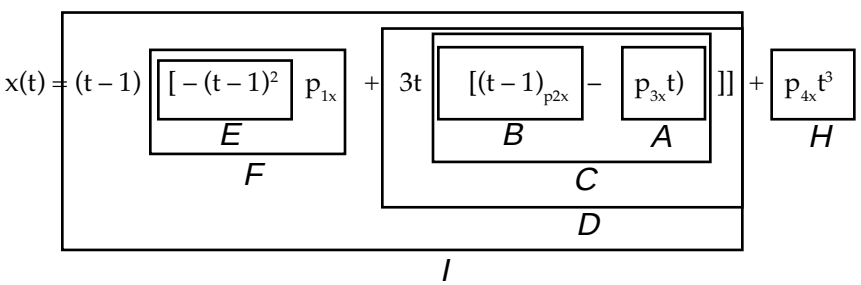


Figure 8.2 Register Assignments For Cubic Bezier Polynomial

8 Graphics

8.2.2 Code Listing

```
/
*******************************************************************************

File Name
   bezier.asm

Version

Purpose
   Define a curve (or surface).

Equations Implemented
                     3                2           2               3
    x(t) = p   (1-t)  +  3 p   t (1-t)  +  3 p   t (1-t)  +  p   t
            1x              2x                3x              4x

Calling Parameters
   f0    = input value (t)
   f1    = t - 1
   f9    = P1
   f5    = P2
   f6    = P3
   f7    = P4
   f14   = 3.0

Return Values
   f0    = x(t)

Registers Affected
   f8, f12, f13

Cycle Count
   11 cycles per bezier evaluation, straight-line code
   10 cycles per bezier evaluation, looped
   400ns per bezier evaluation @ 40ns instruction cycle
2.5 million bezier evaluations/sec @ 40ns instruction cycle

# PM Locations
    11

# DM Locations
    0
```

```
*****************************************************************************/

.GLOBAL bezier;

#define T       f0      /* T, the parametric value */
#define T_M1    f1      /* T - 1 */

#define P1      f9
#define P2      f5
#define P3      f6
#define P4      f7

#define TMP     f4
#define A       f12
#define B       f8
#define C       f8
#define D       f8
#define E       f12
#define F       f12
#define G       f8
#define H       f13
#define I       f12

#define THREE   f14     /* 3.0 */

.SEGMENT /pm    pm_code;
bezier:
        A = T * P3, TMP = T_M1;
        B = T_M1 * P2;
        E = T_M1 * TMP, C = B - A;
        C = THREE * C;
        D = T * C;

        F = E * P1;
        H = T * P4, G = D - F;
        H = H * T;
        rts (db), H = H * T;
            I = G * T_M1;
            f0 = H + I;
.ENDSEG;
```

Listing 8.2 bezier.asm

8 Graphics

8.3 CUBIC B-SPLINE POLYNOMIAL EVALUATION

The Cubic B-spline polynomial, like the Bezier, is commonly used to represent curves and surfaces as a compact list of control points, or *knots*. (see Figure 8.3). In contrast to the Bezier polynomial, the Cubic B-spline polynomial does not pass through the endpoints or any of the other control points.

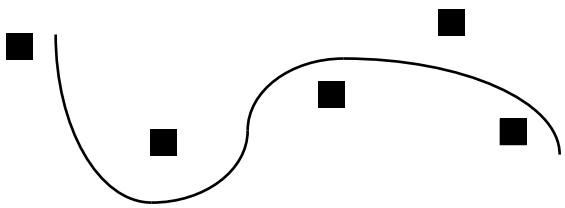


Figure 8.3 Cubic B-Spline Polynomial

See [FOLEY90] p. 521-523 for further information about Cubic B-Spline Polynomials.

8.3.1 Implementation

This routine evaluates the Cubic Bspline polynomial:

$$\overline{-(u-1)^3 p_1 + [\,3u^2(u-2)+4\,]\,p_2 - [\,3(u3+u^2+u)+1\,]\,p_3 + u^3 p_4}$$

The register definitions used in the code below refer to the various sections of the factored equation, as shown in Figure 8.4:

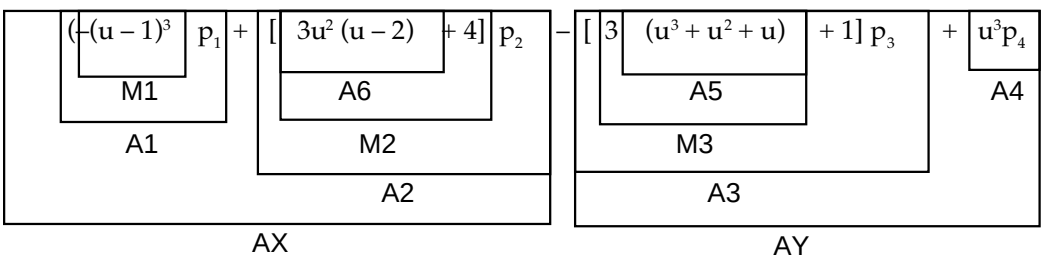


Figure 8.4 Register Assignments For Cubic B-Spline Polynomial

Upon entering the subroutine the algorithm reads the input value twice from the same data memory location. For readability of the equation, the variable `U` holds the input value for the multiplication and the variable `AU` holds the same input value for the additions.

The loop counter is set with R0, which holds the number of values for which the Cubic B-Spline Equation will be executed.

8 Graphics

8.3.2 Code Listing

```
/
*****************************************************************************

File Name
   bspline.asm

Version

Purpose
   Represent curves and surfaces as a compact list of control points.

Equations Implemented
                  3         2                      3    2                   3
           -(u-1) p  + [ 3u (u-2) + 4 ] p  - [ 3(u  + u  + u) + 1) ] p  + u p
                   1                    2                             3      4
   x(u) = ———————————————————————————————————
                                    6.0

Calling Parameters
   r0  = number of x(u) equations to be calculated.
   f14 = 1.0
   i8  = PM  pointer to P1 —> P4
   i0  = DM pointer to 1-D array of input U values
   i1  = DM pointer to constants { 3.0, 4.0, 0.1666666666 }
   i2  = DM pointer to results x(U)
    m1 =  0
    m0 =  1

Return Values
   i2  = pointer to results x(U)

Registers Affected
   F0, F2, F4, F7, F8, F12

Cycle Count
   15N + 11 cycles per bspline evaluation
   600ns per bspline evaluation @ 40ns instruction cycle
   1.7 million bspline evaluations/sec @ 40ns instruction cycle

# PM Locations
    17 instruction + 4 data

# DM Locations
    4 + 2 * N, where N = number of times x(U) will be calculated
```

```
*****************************************************************************/

.GLOBAL bspline;

/* Multiplier Input defines */

#define U       f10     /* U, the parametric value  */

#define X       f0      /* result x(U)              */

#define M1      f0
#define M2      f1
#define M3      f2

#define MUS     f5      /* U^2, MULT version        */
#define MUM1    f6      /* U - 1 */
#define MUM2    f3      /* U - 2 */

#define P1      f4      /* parametric coef #1       */
#define P2      f4      /* parametric coef #2       */
#define P3      f4      /* parametric coef #3       */
#define P4      f4      /* parametric coef #4       */

#define THREE   f7      /* 3.0                      */
#define ISIX    f15     /* 0.16666666666            */
#define ONE     f14


/* ALU Input defines */

#define AUC     f12     /* U^3, ALU version         */
#define AUM1    f9      /* U-1, ALU version         */
#define A1      f15
#define A2      f11
#define A3      f9
#define A4      f10
#define A5      f9
#define A6      f11
#define FOUR    f15     /* 4.0                      */
#define AU      f8      /* U, ALU version           */
#define AUS     f13     /* U^2, ALU version         */
#define AX      f8      /* sum of 1st 2 terms       */
```

(listing continues on next page)

```
#define AY      f12     /* sum of last 2 terms      */

.SEGMENT /pm    pm_code;
bspline:

    U  = dm(i0,m1);
    AU = dm(i0,m0);
    MUM1 = U - ONE;

    lcntr = r0, do bslp until lce;

        M1   = MUM1 * MUM1,                             AUM1 = MUM1;
        M1   = M1 * MUM1,       MUM2 = AUM1 - ONE;
        MUS  = U * U;
        AUC  = U * MUS,         AUS  = MUS;
        M2   = MUM2 * MUS,      A5  = AU + AUS,         THREE = dm(i1,m0);
        A6   = M2 * THREE,      M3  = A5 + AUC,         FOUR = dm(i1,m0);
        A5   = M3 * THREE,      M2  = A6 + FOUR,        P1 = pm(i8,m8);
        A1   = M1 * P1,         M3  = A5 + ONE,         P2 = pm(i8,m8);
        A2   = M2 * P2,                                 P3 = pm(i8,m8);
        A3   = M3 * P3,         AX  = A2 - A1,          P4 = pm(i8,m8);
        A4   = AUC * P4;
                                AY  = A4 - A3,          U = dm(i0,m1);
                                X  = AX + AY,           ISIX = dm(i1,m0);
        X = X * ISIX,                                   AU = dm(i0,m0);
bslp:                           MUM1 = U - ONE,         dm(i2,m0) = X;

    rts;

.ENDSEG;
```

Listing 8.3 B-spline.asm

8.4 BIT BLOCK TRANSFER

Speed of operation is critical for graphics and image processing applications. Even while an image is displayed, the display processor is working to update that image and to create new ones—it is often necessary for the processing unit to perform tasks simultaneously.

Bit-Blt (short for Bit Boundary Block Transfer) is an algorithm for the transfer of image data (pixels) from one location to another. The location from which the data is transferred is called the *source*; the location to which the data goes is called the *destination*. In addition to performing the transfers, Bit-blt can perform logical or mathematical operations on the data.

8.4.1 Implementation

BitBlt, or Bit Block Transfer, is the transferring of *N* words of data to a destination where each destination word is formed by taking bits OFFSET–1 to 0 from the one source datum, and bits 31 down to OFFSET of the next source datum. For example, Figure 8.5 shows BitBlt between two areas in memory, with OFFSET = *b* :

Source Data		*Destination Data*
31 *b*	*b*-1 0	31 0
XXXXXXXXX	AAAAAAA	AAAAAAAAAAAAAAAA
AAAAAAAAA	BBBBBBB	BBBBBBBBBBBBBBBB
BBBBBBBBB	CCCCCCC	CCCCCCCCCCCCCCCC
CCCCCCCCC	DDDDDDD	DDDDDDDDDDDDDDDD
etc...		*etc...*

Figure 8.5 BitBlit

The BitBlt algorithm alternately uses two registers, *X* and Y ,to store the source data to be transferred and two registers, *P* and *Q* , to hold the results of shifting the bit patterns in X and Y. The offset is denoted as *b*. Figure 8.6 shows a the usage of *X* and *Y:*

	31 *b*	*b* – 1 0
X —>	XXXXXXXXX	PPPPPPP
Y —>	PPPPPPPPP	QQQQQQQ
X —>	QQQQQQQQQ	PPPPPPP
Y —>	PPPPPPPPP	QQQQQQQ

Figure 8.6 Register Usage For BitBlt

The algorithm consists of these steps:

1. Read source word into *X*.
2. Shift left *X* by 32 – *OFFSET*.
3. Read source word into *Y*.
4. *Y* Shift right *Y* by *OFFSET*.
5. *X* OR *Y* -> *DM_DEST*.
6. Shift *Y* left by 32 – *OFFSET*. Read new value into *X*, Shift *X* right by OFFSET.
7. *Y* OR *X* -> *DM_DEST*.
8. Repeat 2 - 7 *N* times, where *N* is the number of 32-bit words to move.

8.4.2 Code Listing

```
/
*************************************************************************

File Name
   bitblt.asm

Version

Purpose
   Bit Block Transfer.

Equations Implemented
    None

Calling Parameters
   r0      N
   r1      OFFSET
   i0      pointer to start of source
   i1      pointer to start of destination
   m0      +1

Return Values
   none

Registers Affected
   r0, r2-r6

Cycle Count
   2N + 7 cycles
   12.5 million words / second @ 40ns instruction cycle
   50   million bytes / second @ 40ns instruction cycle

# PM Locations
   15

# DM Locations
   2 * N, where N is number of words to transfer
```

(listing continues on next page)

```
*************************************************************************/

#define N           r0          /* number of 32-bit words to transfer       */
#define OFFSET      r1          /* bit offset for each word                 */
#define A           r2          /* register used to assemble transfer       */
#define B           r2          /* register used to assemble transfer       */
#define DM_SRC      dm(i0,m0)   /* transfer source                          */
#define DM_DEST     dm(i1,m0)   /* transfer destination                     */
#define X           r3          /* a source word                            */
#define Y           r4          /* another source word                      */
#define LREG        r5          /* register storing left-shift value        */
#define RREG        r6          /* register storing right-shift value       */
#define LEFT        by LREG     /* a more descriptive syntax for left-shift */
#define RIGHT       by RREG     /* "                      " for right-shift */

.SEGMENT /pm    pm_code;
bitblt:     N = lshift N by -1;         { iterations = N/2                 }
            N = N - 1;                  { iterations = N/2 -1              }
            RREG = -OFFSET;
            LREG = 32;
            LREG = LREG + RREG,         X = DM_SRC;

            A = lshift X LEFT,          Y = DM_SRC;
            A = A or lshift Y RIGHT,    X = DM_SRC;

            lcntr=N, do blt until lce;
                B = lshift Y LEFT,              DM_DEST = A;
                B = B or lshift X RIGHT,        Y = DM_SRC;
                A = lshift X LEFT,              DM_DEST = B;
blt:            A = A or lshift Y RIGHT,        X = DM_SRC;

            B = lshift Y LEFT,          DM_DEST = A;
            B = B or lshift X RIGHT;
            DM_DEST = B;
.ENDSEG;
```

Listing 8.4 bitblt.asm

8.5 BRESENHAM LINE DRAWING

The Bresenham Line Drawing Algorithm draws pixels approximating a line between two specified endpoints. Since the algorithm is working in pixel coordinates, it is basically an integer operation. The `set_pixel` macro, which turns on the chosen pixel, is application-dependent.

[GLASSNER90] presents an implementation of this algorithm in which both pixel calculations and accumulation of an error term are done in fixed-point, for a total of four fixed-point operations in the inner loop.

8.5.1 Implementation

Although the ADSP-21020 is a floating-point processor, the ability to use both the `ALU` and Data Address Generators in parallel allow the required four fixed-point operations to be executed in two cycles.

If the inner loop of the Bresenham algorithm is implemented in a straightforward way, with all calculations performed in the ALU, it requires five instructions (not including the `set_pixel` operation). In this implementation, all macro registers (ERR, Y, X, SX, SY, AX, AY) are in the register file, and four fixed-point ALU operations are required, two of which are conditional, two unconditional. The five instructions necessary are:

```
        do lp until lce;
           if lt jump noyinc;
               Y = Y + SY;
               ERR = ERR - AX;
noyinc:
           X = X + SX;
           ERR = ERR + AY;
lp:        set_pixel(X,Y);/* application-dependent macro */
```

If the inner loop is implemented so that the ALU calculates the error term, `ERR`, and DAG1 calculates the X and Y coordinates, the loop can be implemented in two instructions:

```
          do lp until lce;
              if ge ERR = ERR - AX, modify(Y,SY);
              ERR = ERR + AY, modify (X,SX);
lp:           set_pixel(X,Y);
```

8 Graphics

For some graphics applications, the `set_pixel` macro may be nothing more than setting a memory location to a constant value. The constant value may simply be 1, or it may be a 24-bit integer indicating the line color. The inner loop used in this application can be as simple as a single instruction:

```
do lp-1 until lce;
   if ge ERR = ERR - AX, modify(P,SY);
   ERR = ERR + AY, modify (P,SX);
   dm(P) = CONST;              /* set_pixel —> one instruction */
lp:
```

If image memory is in data memory and is organized as *M* rows each *N* pixels long, *P* is a DAG1 index register, SY is ±*N*, SX is ±1, and `CONST` is a register file register containing the constant value that the pixel is set to.

This implies that the entire Bresenham inner loop, including setting the chosen pixel, can be performed in as few as three instructions.

8.5.2 Code Listing

```
/
*****************************************************************************

File Name
   bresen.asm:

Version
    1.0

Purpose
   Draws pixels approximating a line between two specified endpoints.

Equations Implemented
   Y=SIN(X) or
   Y=COS(X)

Calling Parameters
   r0      = x1    = x coord on line start
   r1      = y1    = y coord on line start
   r2      = x2    = x coord on line end
   r3      = y2    = y coord on line end
   l2      = 0
   l3      = 0

Return Values
   F0 = Sine (or Cosine) of input Y=[-1,1]

Registers Affected
   r0-r11, i2, m2, i3, m3

Cycle Count

   2N + set_pixel*(N+1) + 25 cycles
                where N = number of points
                                = max (abs(x2-x1),abs(y2-y1))

326 cycles for a 100-pixel line
76,700  100-pixel lines/second @ 40ns instruction cycle

# PM Locations
   34 words

# DM Locations
   11 Words
```

(listing continues on next page)

8 Graphics

```
*****************************************************************************/

/* set_pixel is a user-defined macro. Currently, it just writes out the
    generated X,Y pairs to data memory for inspection.*/

#define set_pixel(x,y)  r0=x; dm(i0,m0)=r0; r0=y; dm(i0,m0)=r0

#define X1      r0
#define Y1      r1
#define X2      r2
#define Y2      r3

#define X       i2
#define Y       i3
#define XLEN    l2
#define YLEN    l3
#define XSTP    1
#define YSTP    1

#define SX      m2
#define SY      m3

#define DX      r4
#define DY      r5
#define AX      r6
#define AY      r7
#define RSX     r8
#define RSY     r9
#define ERR     r11

#define NEG_ONE r10
#define LEFT    by 1
#define RIGHT   by NEG_ONE

/*_____________________________________
Bresenham Line Drawing Function

    Label           Meaning
    _____       _________________
    bresen          start of function
    oct1            octant 1 line-drawing algorithm
    lp1             octant 1 loop
    oct2            octant 2 line-drawing algorithm
```

```
    lp2             octant 2 loop
_____________________________________*/
.SEGMENT /pm pm_code;
.GLOBAL bresen;
bresen:
        NEG_ONE = -1;
        XLEN = 0; YLEN = 0;
        RSX = 0; RSY = 0;
        DX = X2 - X1;
        if ge RSX = RSX + XSTP; /* if ( x2>x1 ) step x-coord up            */
        if lt RSX = RSX - XSTP; /* if ( x2>x1 ) step x-coord down          */
        DY = Y2 - Y1;
        if ge RSY = RSY + YSTP; /* if ( y2>y1 ) step y-coord up            */
        if lt RSY = RSY - YSTP; /* if ( y2>y1 ) step y-coord down          */
        AX = abs DX;
        AX = ashift AX LEFT;    /* X error term = 2 * ( abs(x2-x1) )       */
        AY = abs DY, SX = RSX;  /* save register-SX into modify register   */
        AY = ashift AY LEFT;    /* Y error term = 2 * ( abs(y2-y1) )       */
        comp(AX,AY), SY = RSY;  /* save register-SY into modify register   */
        if lt jump oct2(db);    /* if ( abs(x2-x1) > abs(y2-y1) ) {        */
            X = X1;             /*      computeOctant1;                    */
            Y = Y1;             /*   } else { computeOctant2; }            */

oct1:
        set_pixel(X,Y);

        /* initial error = 2 * ( abs(y2-y1) - abs(x2-x1) )      */
        ERR = ashift AX RIGHT;
        ERR = AY - ERR, lcntr=ERR;

        do lp1-1 until lce;
            if ge ERR = ERR - AX, modify(Y,SY);
            ERR = ERR + AY, modify (X,SX);
            set_pixel(X,Y);
lp1:
        rts;

oct2:
        set_pixel(X,Y);

        /* initial error = abs(x2-x1) - abs(y2-y1))*2   */
        ERR = ashift AY RIGHT;
        ERR = AX - ERR, lcntr = ERR;

        do lp2-1 until lce;
            if ge ERR = ERR - AY, modify(X,SX);
            ERR = ERR + AX, modify (Y,SY);
            set_pixel(X,Y);
lp2:
        rts;
.ENDSEG;
```

Listing 8.5 bresen.asm

8 Graphics

8.6 3-D GRAPHICS TRANSLATION, ROTATION, & SCALING

Translation, rotation, and *scaling* are pixel manipulations which move images.

- An image is translated when its position is changed (the figure seems to move) without affecting size.
- An image is scaled (stretched or compressed) when it is made larger or smaller.
- Rotation of an image is angular movement of an image with respect to its origin.

Scaling, rotation, and repositioning are done by multiplying the input coordinates (a column vector of size 4×1) by a 4×4 transformation matrix This subroutine performs the 3-D graphics transformation:

$$P' = M * P$$

where

P = a 3-D point, a column vector with elements x, y, z, and w
M = a 4 x 4 transformation matrix (combination of translation, scaling and/or rotation)
P' = The translation of point P

8.6.1 Implementation

To implement 3-D Translation, Rotation and Scaling, as a 4x4 by 4x1 matrix multiply requires 16 multiplications and 12 additions. In this routine, assume that M is a matrix of the form:

$$M = \begin{bmatrix} r11 & r12 & r13 & tx \\ r21 & r22 & r23 & ty \\ r31 & r32 & r33 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since the bottom row contains mostly zeroes, this matrix it can be broken down into a 3×3 matrix of the *r* terms and a 3×1 matrix of the *t* terms; when added together the two matrices yield the same values. The transformation can be decomposed as

$$P' = R * P + T$$

where

$$R = \begin{bmatrix} r11 & r12 & r13 \\ r21 & r22 & r23 \\ r31 & r32 & r33 \end{bmatrix}$$

$$P = \begin{bmatrix} px \\ py \\ pz \end{bmatrix}$$

$$T = \begin{bmatrix} tx \\ ty \\ tz \end{bmatrix}$$

reducing the operations to nine multiplications and nine additions.

For the true 4 ×4 by 4 ×1 multiply, see `mul44x41.asm`.

For a description of 3-D transformations and [*x, y ,z, w*] coordinates, see Chapter 5 of [FOLEY90].

In this implementation, the original list of points in data memory (pointed to by i0) contains x, *y, z,* and *w* coordinates, while the transformed points in data memory (pointed to by I1) only have *x, y,* and *z* coordinates. To transfer the *w* coordinate required two data memory transfers (one to read *w*, one to write *w*), and there is only a single data memory move slot left in the inner loop in this implementation. Preserving *w* in the transformed point list costs an additional cycle.

It is not necessary to actually move the *w* coordinates; they are unchanged by the translate/scale/rotate operations. When the transformed points are later used, the transformed *x, y, z* coordinates can be accessed by one DM index register, with a modify value of +1, and the *w* coordinates can be accessed by another DM index register, with a modify value of +4.

Specifically, when a final transformation to the canonical view volume is required, and a true 4×4 by 4×1 operation must be performed, the `mul44x41.asm` code shows how to index the transformed points with separate [x, y, z] and [w] indices.

The order of operations has been rearranged slightly to reduce cycle time and also to take advantage of "empty" instruction slots that are available during multifunction instructions. Instead of first performing all multiplications of *R * P* and then adding *T*, multifunction capability is utilized. For example, the F12 term in the last instruction before the loop holds

r11px + r12py + r13 pz + tx

The program is organized so that the code before the loop reads the coordinates for the first point to be processed, the first two rows of matrix *R*, and enough operands to supply the registers within the loop with data. After the loop is processed, the remaining code calculates the last point coordinates.

In the ADSP-21000 family, registers are read at the beginning of a cycle and are written to at the end of the cycle. For example, in the first parallel multiply accumulate

```
f8 = f3 * f4, f12 = f8 + f15, etc.
```

It is important to remember that F8 used as an addend will reflect the value that F8 contained at the beginning of this cycle, not the product of F3 * F4.

There is a tradeoff between leaving values in registers versus dedicating a circular buffer and register to read the nine values of matrix *R*. Choosing the latter is advantageous because multifunction operations can support the memory reads. Also, since there are nine less registers to keep track of, the code is easier to comprehend.

For more information, see [FOLEY90] pp. 256-261.

8.6.2 Code Listing

```
/
*****************************************************************************

File Name
   transf.asm
Version

Purpose
   Scaling, rotation, and repositioning  with basic matrix operations.

Equations Implemented
   P' = M * P where

   P  = 3-D point, a column vector with elements x,y,z, and w
   M  = 4 x 4 transformation matrix (combination of translation,
        scaling and/or rotation)
   P' = Translation of point P

Calling Parameters
   r0     = number of points to transform
   i0     = index to point P (x,y,z,w)
   m0     = +1
   m1     = +2
   i1     = index to transformed point area
   i2     = index to matrix T
   l2     = 3 (length of T)
   i8     = index to matrix R
   l8     = 9 (length of R)
   m8     = +1

Return Values

Registers Affected
    r0-r6, r8-r10, r12-r15

Cycle Count
   10N+5 cycles, where N 3-D points are transformed
   400ns per point @ 40ns instruction cycle
   2.5 million points/sec @ 40ns instruction cycle

# PM Locations
    27 instructions + 9 words of PM data
# DM Locations
    10 words
```

(listing continues on next page)

```
*****************************************************************************/

.GLOBAL transf;

.SEGMENT /pm    pm_code;
transf:
    r0 = r0-1, f1=dm(i0,m0), f4=pm(i8,m8);         /* first point is processed
*/
                                                   /* outside loop, so
decrement*/
                                                   /* r0, f1 = px, f4 = r11 */
    f15=f1*f4, f2=dm(i0,m0), f4=pm(i8,m8);         /* f2=py,  f4=r12 */
    f8=f2*f4, f3=dm(i0,m1), f4=pm(i8,m8);          /* f3=pz,  f4=r13 */
    f8=f3*f4, f12=f8+f15, f9=dm(i2,m0), f4=pm(i8,m8); /* f9=tx,  f4=r21 */
    f15=f1*f4, f12=f8+f12, f4=pm(i8,m8);           /* f12 holds running sum */
                                                   /* of products, f4=r22 */
    f8=f2*f4,  f12=f9+f12, f4=pm(i8,m8);           /* f4=r23 */

    lcntr=r0, do trlp until lce;

    f8=f3*f4, f13=f8+f15, f10=dm(i2,m0), f4=pm(i8,m8); /*f10=ty, f4=r31*/
    f15=f1*f4, f13=f8+f13, f11=dm(i2,m0), f4=pm(i8,m8);/*f11=tz, f4=r32*/
    f8=f2*f4, f13=f10+f13,f4=pm(i8,m8);               /* f4=r33*/
    f8=f3*f4, f14=f8+f15, dm(i1,m0)=f12;              /*f12 = r11px + r12 py */
                                                      /* + r13 pz + tx */

                  f14=f8+f14,  f1=dm(i0,m0),  f4=pm(i8,m8);
                                                   /*cont with circ buf reads*/
        f15=f1*f4, f14=f11+f14, f2=dm(i0,m0),  f4=pm(i8,m8);
        f8=f2*f4,               f3=dm(i0,m1),  f4=pm(i8,m8);
        f8=f3*f4,  f12=f8+f15,  f9=dm(i2,m0),  f4=pm(i8,m8);
        f15=f1*f4,f12=f8+f12,dm(i1,m0)=f13,f4=pm(i8,m8);
                                                   /*f13=pxr21+pyr22+pzr23+ty*/
trlp:   f8=f2*f4,  f12=f9+f12,  dm(i1,m0)=f14, f4=pm(i8,m8);
                                                   /*f14=pxr31+pyr32+pzr33+tz*/


    f8=f3*f4,  f13=f8+f15, f10=dm(i2,m0), f4=pm(i8,m8);
    f15=f1*f4, f13=f8+f13, f11=dm(i2,m0), f4=pm(i8,m8);
    f8=f2*f4,  f13=f10+f13,               f4=pm(i8,m8);
    f8=f3*f4,  f14=f8+f15, dm(i1,m0)=f12;          /*f12 = pxr11+pyr12+pzr13
+tx*/

    f14=f8+f14;
    rts(db), f14=f11+f14;
        dm(i1,m0)=f13;                             /*f13 = pxr21 +pyr22+pzr23
+ty*/
        dm(i1,m0)=f14;                             /*f14 = pxr31+ pyr32+
pzr33+tz*/
```

8.7 MULTIPLY 4×4 BY 4×1 MATRICES (3D GRAPHICS TRANSFORMATION)

```
.ENDSEG;
```

Listing 8.6 transf.asm

This subroutine performs a 4×4 by 4×1 3D graphics transformation on N 4×1 points. Each point requires 16 multiplications and 12 additions. Even though this is a graphics transformation that operates in three dimensions (the x-, y-, and z-axes), the w coordinates are needed to provide a scaling point of reference for the other three coordinates.

Note that if the transformation is a combination of translation, rotation, and scaling operations, the routine `transf.asm` can be used, saving five multiplications and three additions per point. The `mul44x41` module is intended for use with other types of transformations, such as transforming to the canonical clipping volume, which can't be done with the matrix in `transf.asm`.

See [FOLEY90], pp. 256-261, for further information

8.7.1 Implementation

This code example can be used in two ways. If the list of points is a sequence of [x ,y, z, w] coordinates, assemble this code normally (`asm21k mul44x41`). If the list of points is [x, y, z] coordinates only, and the w coordinate is buried in another list of points, define the preprocessor variable `WOUT` with the syntax `asm21k -DWOUT mul44x41` . This may be needed if `transf.asm` has been used in previous transformations, since the `transf.asm` code doesn't keep the w coordinate embedded in the [x, y, z] coordinate list.

Remember, in the ADSP-21000 family, registers are read at the beginning

of a cycle and written to at the end of the cycle. When register values are used as input operands in algebraic expressions, the value that is operated upon is the value contained in the register before the current cycle is executed.

8.7.2 Code Listing

```
/
*************************************************************************************

File Name
   mul44x41.asm

Version


Purpose
   General 4x4 by 4x1 3D graphics transform on N 4x1 points.

Equations Implemented

Calling Parameters
   r9 = number of points to transform
   i0 = index to [x,y,z] coordinates
   m0 = +1
   m1 = +4
   i1 = index to transformed point area
   i2 = index to [w] coordinates
   i8 = index to 4x4 matrix
   l8 = 16
   m8 = +1

Return Values

Registers Affected
   r1, r4-r9, r14-r15

Cycle Count
   16N+4 cycles, where N 3-D points are transformed
   640ns per point @ 40ns instruction cycle
   1.56 million points/sec @ 40ns instruction cycle

# PM Locations
```

```
    36 words of instructions + 16 words of PM data
# DM Locations
    (8 * N) + 4, where N is the number of points to transform

*****************************************************************************/

.GLOBAL mul44x41;

#ifdef WOUT

#define IP  i0      /* Principal Index, to [x,y,z] list */
#define IS  i2      /* Secondary Index, to [w] list */
#define MS  m1      /* Secondary Modify, for stepping from w to w */

#else

#define IP  i0      /* Principal Index, to [x,y,z] list */
#define IS  i0      /* Secondary Index, to [w] list */
#define MS  m0      /* Secondary Modify, for stepping from w to w */

#endif

.SEGMENT /pm    pm_code;
mul44x41:
    r9 = r9-1,  f4=dm(IP,m0),  f1=pm(i8,m8);
                                /*read px,read r11 of transform matrix*/
    f15=f1*f4,  f5=dm(IP,m0),               f1=pm(i8,m8);
                               /*f15 = pxr11, read py and r12*/
    f8=f1*f5,  f6=dm(IP,m0),                f1=pm(i8,m8);
                               /*f8 = pyr12, read pz and r13*/
    f8=f1*f6,  f15=f8+f15,  f7=dm(IS,MS),   f1=pm(i8,m8);
                               /*f8=pzr13, f15 = pxr11 + pyr12*/
                               /* read pw and r14*/
    f8=f1*f7,  f15=f8+f15,                  f1=pm(i8,m8);
                               /*f8=pwr14, f15 = pxr11 + pyr12*/
                               /*+pzr13, read r21*/
    lcntr=r9, do trlp until lce;  /*continue w/ rest of points*/
        f14=f1*f4, f15=f8+f15,              f1=pm(i8,m8);
        f8=f1*f5,              dm(i1,m0)=f15,f1=pm(i8,m8);
                               /*f15 = pxr11 + pyr12+ pzr13 + pwr14*,write
it*/
        f8=f1*f6,  f14=f8+f14,              f1=pm(i8,m8);
        f8=f1*f7,  f14=f8+f14,              f1=pm(i8,m8);
        f15=f1*f4, f14=f8+f14,              f1=pm(i8,m8);
        f8=f1*f5,             dm(i1,m0)=f14, f1=pm(i8,m8);
        f8=f1*f6,  f15=f8+f15,              f1=pm(i8,m8);
        f8=f1*f7,  f15=f8+f15,              f1=pm(i8,m8);
        f14=f1*f4, f15=f8+f15,              f1=pm(i8,m8);
        f8=f1*f5,             dm(i1,m0)=f15, f1=pm(i8,m8);
        f8=f1*f6,  f14=f8+f14,              f1=pm(i8,m8);
```

(listing continues on next page)

```
        f8=f1*f7,  f14=f8+f14, f4=dm(IP,m0),  f1=pm(i8,m8);
        f15=f1*f4, f14=f8+f14, f5=dm(IP,m0),  f1=pm(i8,m8);
        f8=f1*f5,              f6=dm(IP,m0),  f1=pm(i8,m8);
        f8=f1*f6,  f15=f8+f15, f7=dm(IS,MS),  f1=pm(i8,m8);
trlp:   f8=f1*f7,  f15=f8+f15, dm(i1,m0)=f14, f1=pm(i8,m8);


    f14=f1*f4, f15=f8+f15,                 f1=pm(i8,m8);
    f8=f1*f5,              dm(i1,m0)=f15,  f1=pm(i8,m8);
    f8=f1*f6,  f14=f8+f14,                 f1=pm(i8,m8);
    f8=f1*f7,  f14=f8+f14,                 f1=pm(i8,m8);
    f15=f1*f4, f14=f8+f14,                 f1=pm(i8,m8);
    f8=f1*f5,              dm(i1,m0)=f14,  f1=pm(i8,m8);
    f8=f1*f6,  f15=f8+f15,                 f1=pm(i8,m8);
    f8=f1*f7,  f15=f8+f15,                 f1=pm(i8,m8);
    f14=f1*f4, f15=f8+f15,                 f1=pm(i8,m8);
    f8=f1*f5,              dm(i1,m0)=f15,  f1=pm(i8,m8);
    f8=f1*f6,  f14=f8+f14,                 f1=pm(i8,m8);

    rts(db), f8=f1*f7,  f14=f8+f14;
        f14=f8+f14;
        dm(i1,m0)=f14;


.ENDSEG;
```

Listing 8.7 mul44x41.asm

8.8 TABLE LOOKUP WITH INTERPOLATION

Images that can be represented by coordinate pairs (*x, y*) can be stored in memory. The *x*-coordinate is the index into the array (assuming constant spacing), and the *y* -coordinate is the value of the function. This arrangement is valid for most simple graphs. If greater precision is required, interpolation must be used. (For example, if the corresponding function to a point that is in between the indices.)

8.8.1 Implementation

This code performs a table lookup with interpolation to approximate *f(x)* from a table of values. An image (or a value on a graph) can be calculated with the following data:

- input value for the index (or x-coordinate)

- values of the two points that surround the needed value (two *y*-coordinates)
- initial *x* value (starting index) $\frac{x_1 - X_0}{S}$
- spacing (step size) between the indices

The interpolation formula used is

$$f'(x) = ((F[X + 1] - F[X]) * (\qquad - X) \} + F[X]$$

where

f'(x) = the approximation of f(x)
x = input value x
F[x], F[x + 1] = two consecutive points on the graph between which the

needed value lies
S = x spacing in the table (step size)
X_0 = value of the first index of the table
X = floor of ((x - X0) / S)

8.8.2 Code Listing

```
/
*******************************************************************************

File Name
   tblllkup.asm:

Version

Purpose
   performs a table lookup with interpolation to approximate f(x) from a table
of values.

Equations Implemented
                                        x - X0
   f'(x) = { ( F[X+1] - F[X] ) *     (---  - X ) } + F[X]
                                          S

   where:

   f'(x)        =     the approximation of f(x)
   x            =     input value x
   F[x], F[x+1] =     two consecutive points on the graph between which our
         needed value lies
   S            =     x spacing in the table (step size)
   X0           =     value of the first index of the table
   X            =     floor of (( x - X0 ) / S )


Calling Parameters
   f0     = x
   f7     = 1/S
   f15    = X0
   i0     = start of table
   b0     = i0 (base address for circular buffer)
   l0     = length of table
   m0     = 1

Return Values
   f1 = interpolated f(x)

Registers Affected
   MODE1 Register Bits:
```

```
          TRUNC   = 1

Cycle Count
   11 cycles
   440ns @ 40ns instruction cycle
   Note: 11 cycles includes the DAG hold after load of m1 register.

# PM Locations
    10 words of instructions
# DM Locations
    N words data memory, where N is the number of values in the table

*****************************************************************************/


.GLOBAL tbllkup;

#define     X       f0          /* x, the input value                      */
#define     XF      f2          /* (float) floor(X)                        */
#define     X0      f15         /* x(0), first index in table              */
#define     IS      f7          /* 1/S inverse of table stepsize           */
#define     IDX     r3          /* (int) floor(X)                          */
#define     INTRP   f2          /* interpolation factor                    */
#define     FI      f4          /* F(x)                                    */
#define     FI1     f1          /* F(x+1)                                  */
#define     RES     f1          /* the interpolated result                 */

.SEGMENT /pm    pm_code;
tbllkup:
        X = X - X0, i0 = b0;    /*subtract starting index from input,verify that
i0 is*/
                                /*pointing to top of array */

        X = X * IS;             /*=divide by step size  */
        IDX = fix X;            /*take integer part for spacing*/
        XF = float IDX, m1 = IDX;/*float effectively gives floor function*/
        INTRP = X - XF, modify(i0,m1);/*finding interpolation value */
        FI = dm(i0,m0);         /*read y-coordinates that surround needed value
*/
        FI1 = dm(i0,m0);
        rts (db), RES = FI1 - FI;/*subtract y-coordinates*/
```

```
            RES = RES * INTRP;   /*multiply by interpolation factor*/
            RES = RES + FI;      /*add to known value */
.ENDSEG;
```

Listing 8.8 tblllkup.asm

8.9 VECTOR CROSS PRODUCT

The Vector Cross Product, a commonly used algorithm in 3D graphics illumination and back-face culling, produces a vector perpendicular to its two vector operands.

The cross product of vectors A and B is C; $C = A \times B$

can be written as:

$$cx = ay * bz - az * by$$
$$cy = az * bx - ax * bz$$
$$cz = ax * by - ay * bx$$

8.9.1 Implementation

To maximize looped performance, this subroutine simultaneously computes two vector cross products within a twelve-instruction loop. Since each cross product requires six multiplications, the effective rate of six cycles per cross product is the best rate that can be achieved on a

processor with a single parallel multiplier.

Preprocessor `#define` statements are used for register assignments to make reading the code easier.

8.9.2 Code Listing

```
/
*******************************************************************************

File Name
   xprod.asm

Version

Purpose
   Vector Cross Product: produces a vector perpendicular to the two vector
operands.

Equations Implemented
   cx = ay*bz  -  az*by
   cy = az*bx  -  ax*bz
   cz = ax*by  -  ay*bx

Calling Parameters
   F0 = Input Value X=[6E-20, 6E20]
   l_reg=0

Return Values

Registers Affected
   r11    = number of cross products to perform (must be even and greater than
4)
   i0     = index to list of A vectors
   l0     = 0
   m0     = +1
   i1     = index to C output vectors
   l1     = 0
   i8     = index to list of B vectors
   l8     = 0
   m8     = +1

Cycle Count
   6N+7 cycles, where N Cross Products are performed
   240ns per Cross Product @ 40ns instruction cycle
```

(listing continues on next page)

8 Graphics

```
   4.2 million Cross Products/sec @ 40ns instruction cycle

# PM Locations
    29 words of PM instructions = 3 * N words of PM data,
    where N is number of cross products to perform
# DM Locations
    6 * N words of DM, where N is number of cross products to perform

*****************************************************************************/

.GLOBAL xprod;

#define AX1     f1
#define AY1     f2
#define AZ1     f3

#define AX2     f0
#define AY2     f3
#define AZ2     f2

#define BX1     f5
#define BY1     f6
#define BZ1     f7

#define BX2     f4
#define BY2     f7
#define BZ2     f6

#define AXBY    f8
#define AYBX    f12
#define AYBZ    f9
#define AZBY    f13
#define AZBX    f10
#define AXBZ    f14

#define CX      f15
#define CY      f15
#define CZ      f15

.SEGMENT /pm    pm_code;
xprod:

    r11 = lshift r11 by -1;     /* 2 vector cross prods per loop, so divide */
                                /* original number of points by 2 */
    r11 = r11-1,           AX1=dm(i0,m0), BX1=pm(i8,m8);
                                /* first multply occurs */
                                /*outside before loop,so subtract 1 */
                                /* from num of points */
                           AY1=dm(i0,m0), BY1=pm(i8,m8); /* read AY1, BY1 */
                           AZ1=dm(i0,m0), BZ1=pm(i8,m8); /* read AZ1, BZ1 */
```

```
    AYBZ=AY1*BZ1;
    AZBY=AZ1*BY1,          AX2=dm(i0,m0), BX2=pm(i8,m8); /*read data of next pt
*/

    AZBX=AZ1*BX1, CX=AYBZ-AZBY; /* continue multiplies, calc. CX term */
    AXBZ=AX1*BZ1,               AY2=dm(i0,m0), BY2=pm(i8,m8);
    AXBY=AX1*BY1, CY=AZBX-AXBZ, dm(i1,m0)=CX;
                                /* first write of CX term, cal. CY term */
    AYBX=AY1*BX1,               AZ2=dm(i0,m0), BZ2=pm(i8,m8);
                                /*cont multipy and read*/

    lcntr=r11, do xlp until lce;
        AYBZ=AY2*BZ2, CZ=AXBY-AYBX, dm(i1,m0)=CY;
        AZBY=AZ2*BY2,               AX1=dm(i0,m0), BX1=pm(i8,m8);
        AZBX=AZ2*BX2, CX=AYBZ-AZBY, dm(i1,m0)=CZ;
        AXBZ=AX2*BZ2,               AY1=dm(i0,m0), BY1=pm(i8,m8);
        AXBY=AX2*BY2, CY=AZBX-AXBZ, dm(i1,m0)=CX;
        AYBX=AY2*BX2,               AZ1=dm(i0,m0), BZ1=pm(i8,m8);
        AYBZ=AY1*BZ1, CZ=AXBY-AYBX, dm(i1,m0)=CY;
        AZBY=AZ1*BY1,               AX2=dm(i0,m0), BX2=pm(i8,m8);
        AZBX=AZ1*BX1, CX=AYBZ-AZBY, dm(i1,m0)=CZ;
        AXBZ=AX1*BZ1,               AY2=dm(i0,m0), BY2=pm(i8,m8);
        AXBY=AX1*BY1, CY=AZBX-AXBZ, dm(i1,m0)=CX;
xlp:    AYBX=AY1*BX1,               AZ2=dm(i0,m0), BZ2=pm(i8,m8);


    AYBZ=AY2*BZ2, CZ=AXBY-AYBX, dm(i1,m0)=CY;
                                 /*continue multipies and writes outside*/
    AZBY=AZ2*BY2;                /*last loop, no more reads needed */
    AZBX=AZ2*BX2, CX=AYBZ-AZBY, dm(i1,m0)=CZ;
    AXBZ=AX2*BZ2;
    AXBY=AX2*BY2, CY=AZBX-AXBZ, dm(i1,m0)=CX;
    rts(db), AYBX=AY2*BX2;
        CZ=AXBY-AYBX, dm(i1,m0)=CY;
        dm(i1,m0)=CZ;
.ENDSEG;
```

Listing 8.9 xprod.asm

8.10 REFERENCES

[FOLEY90] Foley, J., A. VanDam, et. al. 1990.*Computer Graphics, Principles and Practice*. Addison-Wesley.

[GLASSNER90] Glassner, Andrew S. 1990. *Graphics Gems*. Boston: Academic Press, Inc.

Image Processing 9

Graphics and imaging are two-dimensional applications. Just as in the one-dimensional case, it is often desirable to manipulate the data through filtering. This chapter describes the implementation of two-dimensional filtering using 3×3 kernals, as well as median filtering. Histogram equalization, a technique for enhancing images, is also described.

9.1 TWO-DIMENSIONAL CONVOLUTION

According to digital signal processing theory, a linear time-invariant system (LTI system) is completely characterized by the impulse response function, *h(n)*, which is the system's response to the unit sample sequence delta (n). From this principle, the response *y(n)* of an LTI system is the *convolution sum* of an input to the system *x(n)* with the impulse response, *h(n)*. The following equation defines convolution in one dimension:

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

A two dimensional LTI system can again be described by it's impulse response $h(n_1, n_2)$.The corresponding two-dimensional convolution sum can be describe by the following equation:

$$y(n_1, n_2) = h(n_1, n_2) ** x(n_1, n_2)$$

where

$x(n_1,n_2)$ describes the two-dimensional input
$y(n_1,n_2)$ describes the two-dimensional output

Note: The " **" symbol signifies a two-dimensional convolution.

This equation expands to the following equation:

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x(k_1, k_2)\, h(n_1 - k, n_2 - k_2)$$

9 Image Processing

A matrix is a natural data structure for storing two-dimensional data. For the two dimensional convolution, the data input values (x-values) are stored in the data matrix (in data memory), and the impulse response values, h-values, are stored in the transfer function matrix (in program memory).

9.1.1 Implementation

FIR filters are used in the two-dimensional signal space just as they are in the one-dimension signal space. For two-dimensional signals kernal convolution performs FIR filtering on an image matrix. The kernal convolution implements the following equation:

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x(k_1, k_2)\, h(n_1 - k, n_2 - k_2)$$

where

$y(n_1, n_2)$ is the output filtered image
$h(k_1, k_2)$ is the filter kernal
$x(n_1,n_2)$ is the input image

The kernel convolves a 3×3 coefficient matrix by an $M \times N$ image matrix. This code segment can use any size data buffer; the `#DEFINE` statements at the start of the program determine the size of the input data matrix. The input data (in data memory) is assumed to be in row major format. The kernel (convolution coefficients) are loaded from the file `coeff.dat` into program memory.

Figure 9.1 shows A graphical depiction of the 3×3 kernel convolution.

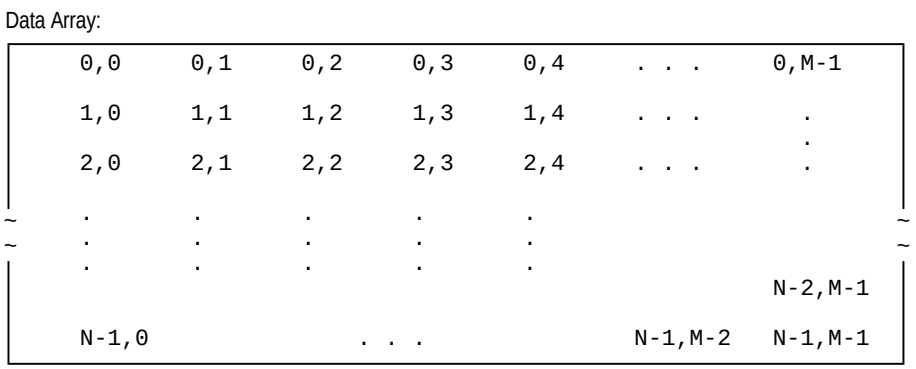


Figure 9.1 3x3 Convolution Matrix

The data array elements in the upper left are referred to as {d00,d01...}.
The convolution kernel elements are referred to as { c00, c01, ... , c32, c33 }.
The first two convolutions are formed from these products:

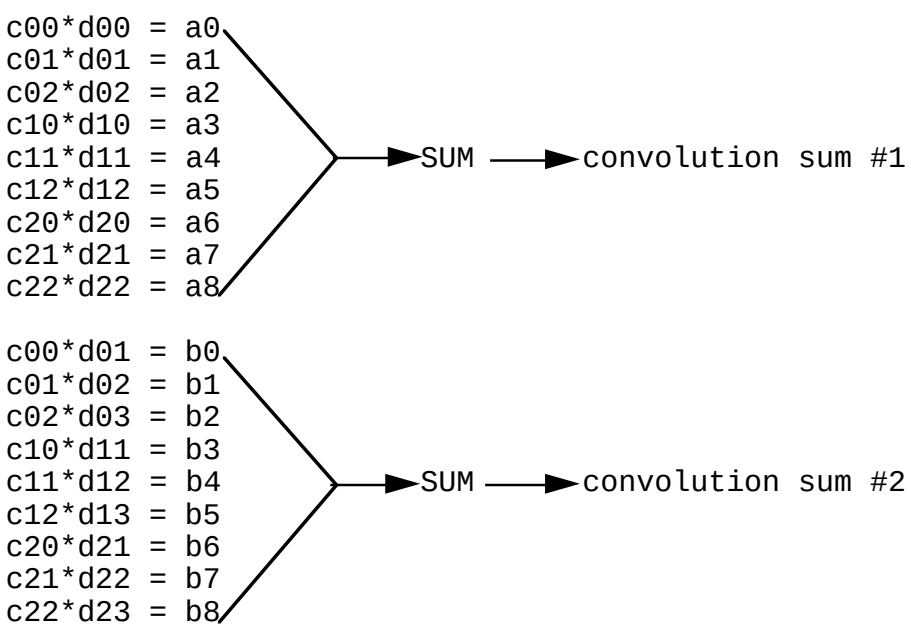


Figure 9.2 3x3 Convolution Operation

Figure 9.2 shows that for every convolution sum there are

- nine reads from data memory (input data)
- nine reads from program memory (kernel coefficients)
- eight additions
- one write to program memory for the convolution sum

The inner loop of the algorithm performs these 27 operations.

The program executes the outer loop *m-2* times because the size the data matrix is 3×3 the data starts at the upper left hand corner of the matrix. At the *m-2*th iteration through the outer loop, the 3×3 kernel matrix covers the data values through m; further loop iterations beyond *m-2* times are redundant. Therefore, the outer loop of the program determines the starting point of the convolution and manages the update of the array indices from row to row. The advantages of the automatic post modify DAGs are obvious; one modify statement and one read statement can manage the indices to process the data array.

The inner loop of the algorithm performs all of the calculations. While the outer loop determines the starting point of the convolution and manages the update of the array indices from row to row, the inner loop of the algorithm performs the calculations. The third kernel coefficient is stored in F5 in the `setup` segment of the code so as to avoid an extra cycle for program memory read within the loop. (The ADSP-210xx multifunction operations allow a data memory read/write and a program memory read/write in one cycle, and the algorithm requires 18 reads and one write.)

The calculations start at the upper left hand corner of the image matrix. The first iteration of the kernel matrix by the image matrix performs the operations over the first , second, and third rows. The second iteration covers the second, third, and fourth rows, etc. Therefore, at the $M-2$ iteration, the $M-1$ and M rows have already been dealt with, and, in the interest of time and space, we set the outer loop equal to $M-2$.

The write operation to the PM location that contains the convolution sum is at the second instruction in the inner loop, which may seem like an unusual place for it. To minimize the cycle count, this write of the partial sum cannot occur at the bottom of the loop. F12 is used to hold the partial products, and F8 is used to hold the running total of the partial sums.

The first time through the loop, F8 (which only contains one product) is written to program memory at the relative bottom of the circular buffer output (where the base register is currently pointing). When the circular buffer is written to again, with the valid sum of products, the memory is correctly written at the relative top of the circular buffer.

9.1.2 Code Listing

```
/
*************************************************************************

File Name
   comv3x3.asm
Version

Purpose
   This code performs the Kernel Convolution.

Equations Implemented
      y(n1, n2) =  SUM (SUM ( x(k1, k2) h(n1 – k, n2 – k2))

Calling Parameters
      none
Return Values

Registers Affected

Cycle Count
   Setup: This code initializes register constants and address pointers needed.
               cycles=10+1
               time=11•50ns=550ns

   Benchmark: This code performs the Kernel Convolution.
               cycles=(9*(n-2)+3)*(m-2)+5+11    (note: 11 cache misses)
               time (at m=780, n=1024, cycletime=50ns) =7158394•50ns=.3579197s
               MFLOPS sustained in core loop=20MIPS(9mpy+8add)/9=37.77

# PM Locations
     27 words of instructions + 9 words of PM data
# DM Locations
     m * n + (m -2) * (n -2) where m = rows and n = collumns

*************************************************************************/
```

(listing continues on next page)

9 Image Processing

```
/*The third coefficient is stored in the register file to free up a cycle to
output a result. The first write to the output buffer is unused, the pointer
then wraps around to the proper location at the start of output.*/

#DEFINE   m 780        /*m by n image*/
#DEFINE n 1024

.SEGMENT/DMDATA dmsegment;
.VAR                                input[m*n];
.VAR                                output[(m-2)*(n-2)];
.ENDSEG;

.SEGMENT/PMDATA pmdataseg;
.VAR                                kernal[3*3]=”coef.dat;
.ENDSEG;

.SEGMENT/PMCODE pmcodeseg;
setup:          M0=1;
          M1=n-2;
          M2=-(2*n+1);
          M3=2;
          M8=1;
          M9=2;
          B0=input;                 L0=0;
          B8=kernal;          L8=@kernal;
          B9=output+(m-2)*(n-2);    L9=@output;
          F5=PM(kernal+2);          /*store in register file to save cycle*/

kern_conv:      F0=DM(I0,M0), F4=PM(I8,M8);
          LCNTR=m-2, DO in_row UNTIL LCE;
                LCNTR=n-2, DO in_col UNTIL LCE;
                      F8=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M9);
                      F12=F0*F4,           F0=DM(I0,M1), PM(I9,M8)=F8;
                      F12=F0*F5, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
                      F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
                      F12=F0*F4, F8=F8+F12, F0=DM(I0,M1), F4=PM(I8,M8);
                      F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
                      F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
                      F12=F0*F4, F8=F8+F12, F0=DM(I0,M2), F4=PM(I8,M8);
in_col:               F12=F0*F4, F8=F8+F12, F0=DM(I0,M0), F4=PM(I8,M8);
                MODIFY(I0,M0);
in_row:               F0=DM(I0,M0);
          RTS (D), F8=F8+F12;
          PM(I9,M8)=F8;
          NOP;
.ENDSEG;
```

Listing 9.1 CONV3x3.ASM

9.2 MEDIAN FILTERING (3X3)

Like any other data transmitted through a channel, images are subject to noise corruption. One type of noise is *shot* or *impulse noise*—a strong spike-like noise scattered evenly through the image. *Median filtering* can remove shot noise and is particularly useful when the image sharpness must be preserved. It replaces a given data value with the median value of its neighboring elements. The median is the value *m* such that half the values in an array are less than *m* and half the values are greater than *m*.

The program presented here implements a 3×3 median filter (a nine-element array) which is applied to every pixel in the image and replaces that pixel with the median value.

In any six element subset of nine data values, at least one of the values will be larger than the median (i.e., at least one value will have a rank of sixth largest). This algorithm of median filtering considers the first six values in the order that they appear in memory—no presorting is required. The highest and lowest values are discarded and a new data value is read. Performing this compare iteratively on successively smaller groups yields a median of three values ranked in the middle of the array—the median is the "middle" value of these three.

For further information see [GLASSNER90], p.171, 711; [GONZALEZ87], p. 162-163; [JAIN89], p. 246-247.

9.2.1 Implementation

The median filter algorithm has a simple implementation in ADSP-21000 family assembly language. Most of the program consists of a few instructions repeated over and over again— the use of macros leads to code that is easy to read and visualize.

The `comp` (compare) operator is central to the macro invocation: it sets the appropriate flag in the arithmetic status (ASTAT) register depending on the relative value of the operands. This conditional operation is always valid because the ADSP-210xx updates the ASTAT register after every operation. The AZ flag is set (ALU result is 0) if the operands of `comp` are equal, and the AN flag is set (ALU result is negative) if the first operator is smaller than the second. The state of the ASTAT, if neither AZ or AN is set, is positive; operands a and b will be swapped if and only if a is greater than b.

9 Image Processing

The ADSP-210xx has a sufficient number of registers to perform the memory read and write operations and the comparisons and swaps that follow them. If registers were not available, it would not be possible to read and compare using the same register; an intermediate storage area in memory and overhead cycles used for memory management would be needed.

9.2.2 Code Listing

```
/
*************************************************************************

File Name
   med3x3.asm

Version
    1.0

Purpose
   Median filtering: replace a given data value with the median value of its
neighboring            elements.

Equations Implemented
   Y=SIN(X) or
   Y=COS(X)

Calling Parameters
   i0           index to data values
   m0           column offset (usually 1)
   m1           row offset (usually [#pixels/row - 2])
   m2           offset to next 3x3 block (usually -[2*m1 + 1])

Return Values
   r0     median value

Registers Affected
   r0-r6   tmp values

Cycle Count

   56 cycles
   2.24µs per 3x3
   587ms for median filtering a 512x512 imag

# PM Locations
   16 words of instruction
# DM Locations
   9 words
```

```
*************************************************************************/

#define s2(a,b)        comp(a,b); if gt b = pass a, a = b;
                                                  /*one macrdefinition*/
                                                  /*that gets repeated*/

#define mnmx3(a,b,c)   s2(b,c); s2(a,c); s2(a,b)

#define mnmx4(a,b,c,d)  s2(a,b); s2(c,d); s2(a,c); s2(b,d)

#define mnmx5(a,b,c,d,e) s2(a,b); s2(c,d); s2(a,c); s2(a,e); \
                              s2(d,e); s2(b,e)

#define mnmx6(a,b,c,d,e,f) s2(a,d); s2(b,e); s2(c,f); \
                                s2(a,b); s2(a,c); s2(e,f); s2(d,f)

.SEGMENT /pm  pm_code;
.GLOBAL med3x3;                                   /*make code visible*/
                                                  /*to other modules*/

med3x3:   r1=dm(i0,m0);                           /*initial 6 reads*/
          r2=dm(i0,m0);
          r3=dm(i0,m1);
          r4=dm(i0,m0);
          r5=dm(i0,m0);
          r6=dm(i0,m1);
strt:
          mnmx6(r1, r2, r3, r4, r5, r6);          /*start the compare*/
mm6:
          r1 = dm(i0,m0);                         /*another read; overwrite*/
                                                         /*r1 (the smallest)*/

          mnmx5(r1, r2, r3, r4, r5);                     /*smallest and greatest
*/
                                                  /*values have dropped out*/
mm5:
          r1 = dm(i0,m0);                         /*r1 replaced again, */
                                                  /*r5 drops out*/
          mnmx4(r1, r2, r3, r4);
mm4:
          r1 = dm(i0,m2);                         /*a 3 value sort here*/
          mnmx3(r1, r2, r3);
mm3:
          rts (db);                               /*delayed branch means*/
                                                  /*2 more instructions*/
                                                  /*executed before return*/


                r0 = r2;                          /*r2 was median of last 3*/
                nop;
.ENDSEG;
```

Listing 9.2 med3x3.asm

9 Image Processing

9.3 HISTOGRAM EQUALIZATION

A histogram tallies the intensities of all 8-bit pixels of an image into the 256 bins of a histogram array. If the pixel resolution is increased to 10-bits or 12-bits, a larger histogram array is required (1024 or 4096 bins, respectively). An intensity of zero (0x00) is the darkest pixel and is tallied in bin 0, the first bin. An intensity of 255 (0xFF) is the brightest pixel and is stored in bin 255, the last bin. The ADSP-210xx supports a 32-bit integer data type, and thus can tally large numbers of pixels without overflowing the histogram array.

After all pixel intensities have been tallied, the histogram array can be analyzed to determine the darkness or brightness of the image. If the image is too dark, most of the pixels will record in the first 128 bins (Figure 9.3). If the image is too bright, most of the pixels will record in the second 128 bins (Figure 9.4).

The process of *histogram equalization* enhances the contrast of the image by applying a gain and offset to each pixel, which produces an image with pixels that cover all intensities.

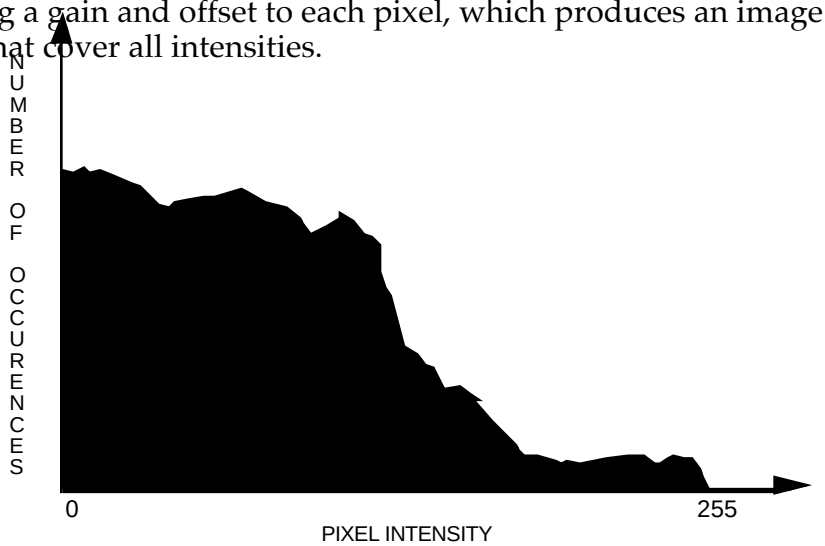


Figure 9.3 Histogram Of Dark Picture

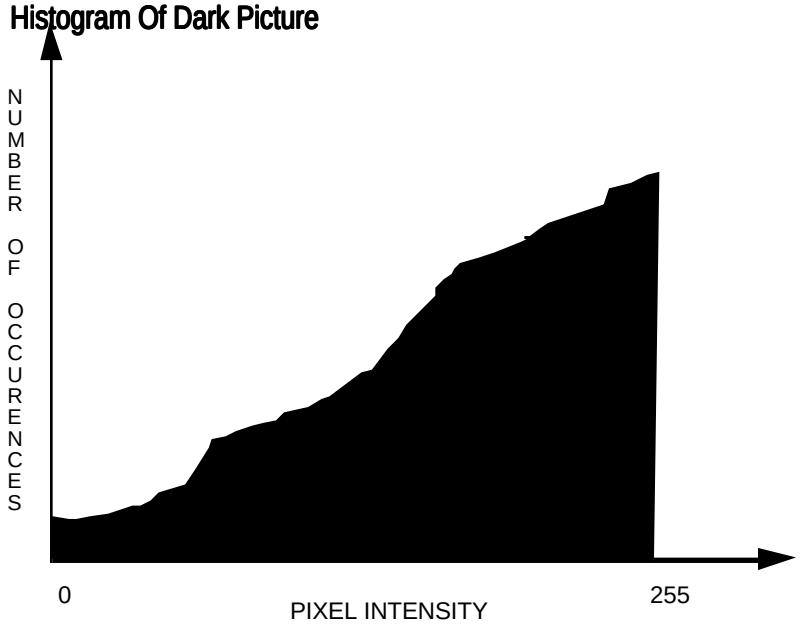


Figure 9.4 Histogram Of Bright Picture

9.3.1 Implementation

The following sequence of tasks generalize the calculation of the histogram array:

1. Read next pixel from memory.
2. Add the pixel value to the base address of the histogram array.
3. Use this address value as a pointer to read the current number stored in the corresponding bin in the temporary histogram array.
4. Increment that bin value.
5. Write the bin value back to the temporary histogram.

The `histo.asm` program (Listing 9.3) is an optimized implementation of the pseudo-code listed above. The core histogram loop, `hloop`, processes two pixels per iteration. This technique yields an average calculation rate of 3.5 instruction cycles per pixel. Due to register pipelining, the two pixels processed in the loop must be tallied in two different histogram arrays. After all pixels are processed, these two temporary histograms in program memory are added together to produce a composite histogram in data memory. The subroutine declares the two

temporary and the final composite histogram arrays.

9.3.2 Code Listing

```
/
*****************************************************************************

File Name
   histo.asm

Version

Purpose
   An image histogram is an array summarizing the number of occurrences of each
   grey level in an image. If a pixel in a typical monochrome image can take on
   any of 256 distinct values, the histogram would need 256 bins. In each bin
   the number of ocurrences of this grey level is stored.

   The algorithm used here assumes the monochrome image is stored in a buffer in
   Data Memory, and the histogram is formed in Data Memory.

Equations Implemented
    None

Calling Parameters
   b0,i0   = data buffer start
   l0           = data buffer length
   m0           = 1
   m15          = 0

   Note: l1 = l8 = l9 = N = # of bins. N must be even, positive, and >= 4

Return Values
   histogram output bins pointed to by i1

Registers Affected
   r0           input data 1
   r1           bin value 1
   r2           input data 2
   r3           bin value 2, tmp register
   r14          temp bin buffer #1 start
   r15          temp bin buffer #2 start

Cycle Count
   3.5N + 2B + 30 cycles
          where N = number of data values
          B = number of bins

# PM Locations
    32 words instructions + 512 words data
# DM Locations
    N * 256, where N = number of data values
```

```
*****************************************************************************/

* declare two temporary histogram buffers in program memory */

SEGMENT /pm pm_data;
VAR  temp1[256];
VAR  temp2[256];
ENDSEG;

* declare histogram result buffer in data memory */

SEGMENT /dm dm_data;
VAR  histogram[256];
GLOBAL  histogram;
ENDSEG;

SEGMENT /pm  pm_code;
GLOBAL histo;

* Initialize data addresses, loop count */

histo:     b8 = temp1; l8=@temp1; r14=i8;
           b9 = temp2; l9=@temp2;
           r3 = l1;                        /* r3 = N = # of bins */
           r3 = lshift r3 by -1;           /* r3 = N/2   */
           r3 = r3 - 1, r15=i9;            /* r3 = N/2-1, initialize r15  */

* Do the histogram into 2 temporary bins in PM */

           r0=dm(i0,m0);
           r0=r0+r14, r2=dm(i0,m0);
           lcntr = r3, do hloop until lce;
                r2=r2+r15, i8=r0;
                i9=r2;
                r1=pm(i8,m15);  /* 2 cycles due to I register load latency */
                r1=r1+1, r3=pm(i9,m15);
                r3=r3+1, r0=dm(i0,m0), pm(i8,m15)=r1;
hloop:          r0=r0+r14, r2=dm(i0,m0), pm(i9,m15)=r3;

           r2=r2+r15, i8=r0;
           i9=r2;
           r1=pm(i8,m15);
           r1=r1+1, r3=pm(i9,m15);
           r3=r3+1, pm(i8,m15)=r1;
           pm(i9,m15)=r3;

* Now combine the bins back into DM */

           b1=histogram; l1=@histogram;
           i8=b8;
           i9=b9;
           r2=l1;
           r2=r2-1, r0=pm(i8,m8);
```

(listing continues on next page)

```
            r1=pm(i9,m8);
            lcntr=r2, do combine until lce;
                  r2=r0+r1,     r0=pm(i8,m8);
ombine:      dm(i1,m0)=r2,  r1=pm(i9,m8);

            rts (db);
            r2=r0+r1;
            dm(i1,m0)=r2;

ENDSEG;
```

Listing 9.3 histo.asm

9.4 ONE-DIMENSIONAL MEDIAN FILTERING

A median filter is designed to sort samples in an array by magnitude, lowest to highest. The middle sorted sample is the median value. Median filters require an odd number of samples to guarantee a middle position.

Median filters are used in image processing to average the image without blurring edges, like low pass and mean average filters do. Median filters are non-linear functions and are not used in speech or audio signal processing.

Some median filters are calculated on samples that cover a two-dimensional area. The median filter discussed in this section is one-dimensional; it finds the median of a horizontal line of samples. One-dimensional median filters may be used if the image scanning device is line array or if it necessary to reduce the processing power required of the DSP.

9.4.1 Implementation

Median filters have a delay line similar to the FIR filter that works on the last N samples. After the median filter processes a sample, it outputs the results and waits for the next input sample. When the next sample is received, it replaces the oldest sample in the delay line.

Figure 9.5 demonstrates the first pass through the median filter to resolve the lowest magnitude sample. The median filter result is four in this example.

Listing 9.4 is a fixed point implementation of the median filter for an ADSP-210xx family DSP. The first task is to transfer the samples from the delay line to the median filter buffer, because they are shuffled. Next, the

median filter is performed, as demonstrated in figure 9.5. Each pass of the outer loop resolves the next highest magnitude sample in the median filter buffer. Each pass through the inner loop compares the current lowest sample in the outer loop pass to the next sample in the median filter buffer. If the next sample is less than the current lowest, they are swapped. The last outer loop pass resolves the middle or median sample in the median filter buffer. After the median sample has been resolved, it is not necessary to resolve the remaining higher magnitude samples.

Listing 9.5 is the floating-point version of listing 9.4.

Outer Loop Pass 1. R4 = 4

cmp R4 = 4, r2 = 6; 6 < 4, no, R4 = 4
cmp R4 = 4, r2 = 7; 7 < 4, no, R4 = 4
cmp R4 = 4, r2 = 5; 5 < 4, no, R4 = 4
cmp R4 = 4, r2 = 1; 1 < 4, yes, R4 = 1, base_adr + 4 = 4
cmp R4 = 1, r2 = 2; 2 < 1, no, R4 = 1
cmp R4 = 1, r2 =3; 3 < 4, no, R4 = 1

First pass resolves lowest in buffer, since lowest is not median
in is not restored into the median filter buffer. Next pass starts with
r4 = base_adr + 1 =5. The next pass will resolve the next lowest.
The third pass of the outer loop wil resolve the next lowest. The
fourth pass will resolve the median.

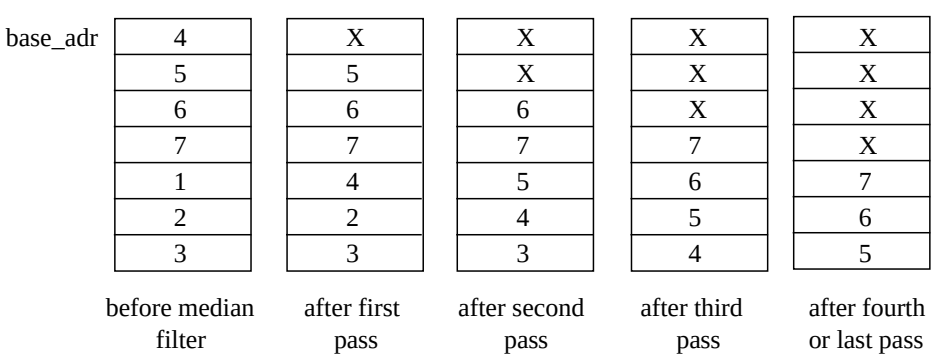


Figure 9.5 Median Filter Algorithm

9.4.2 Code Listings

```
/
*************************************************************************

File Name
   med_fix.asm

Version

Purpose
   N tap one-dimensional median filter subroutine for fixed point data
   Equations Implemented

Calling Parameters
   b1, i1 = start address of input delay line in data memory
   l1           = length of delay line buffer
   m1           =1 - to modify index registers
   b8, i8 = start address of median filter buffer in program mem
   l8           =length of delay line buffer
   m9           =1 - to modify index registers

Return Values
   r4 = median of values in delay line

Registers Affected
    r0, r2, r4, r5, r15

Cycle Count
   FILTER_LEN + 6*[(FILTER_LEN+1)/2] + 14 + 3*sum[N]
   where N=(FILTER_LEN-1)/2 to FILTER_LEN-1
   99 cycles for FILTER_LEN=7

# PM Locations
    16 Instruction + N Words of PM Data, where N is the order of the median
filter
# DM Locations
    N Words, where N is the order of the median filter
```

```
*************************************************************************/

#define  FILTER_LEN 7                  /* must be an odd number */

.segment/pm pmcode;

.GLOBAL start_median;

start_median:

 /* xfer loop - transfer delay line data in DM to median filter buffer in PM */

    r0=dm(i1,m1);
    lcntr=FILTER_LEN-1, do xfer until lce;
xfer:    r0=dm(i1,m1), pm(i8,m9)=r0;   /* transfer to filter buffer */
                                       /* read from input buffer */
    pm(i8,m9)=r0;                      /* transfer to filter buffer */

/*
    median filter loop - find median value in delay line using this technique:

              for N=0 to (FILTER_LEN+1)/2    - outer loop

                   for M=N to FILTER_LEN     - inner loop

                        if (median[N] < median[M])

                             median[M]=median[N], median[M]=median[N]

                        inc M

                   inc N
*/

    b9=b8;                             /* i8, i9 point to median filter data */
    r15=FILTER_LEN;                    /* r15 is loop count for inner loop */

   /* each pass through the outer loop resolves the next greatest magnitude */
   /* in the median filter buffer - median[N]                                */

    lcntr=(FILTER_LEN+1)/2, do outer_loop until lce;

         r4=pm(i8,m9);                 /* read median[N] */
         modify(i9,1);                 /* i9 points to median[M] */
                                       /* where M=N+1 first */
         r15=r15-1, r5=pm(i9,m9);      /* decrement inner loop count */
                                       /* read median[M] */

 /* inner loop finds minimum of the remaining values in median filter buffer */

         lcntr=r15, do inner_loop until lce;
              r2=pass r5;                        /* f2=median[M] */
              comp(r2,r4), r5=pm(i9,m9);         /* cmp median[M], median[N] */
```

(listing continues on next page)

```
inner_loop:   if lt r4=pass r2, pm(-2,i9)=r4;    /* if median[M] < median[N] */
                                                 /* median[N]=median[M] */
                                                 /* median[M]=median[N] */
outer_loop:

         i9=i8;                                  /* init i9 to median[N+1] */

    rts;                                         /* return is non-delayed */
                                                 /* 3 cycles              */

.endseg;
```

Listing 9.4 Fixed-Point 1-D Median Fillter

```
/
*************************************************************************

File Name
   med_flt.asm

Version
    1.0
Purpose
   N tap one-dimensional median filter subroutine for floating point data

Calling Parameters
   b1, i1 = start address of input delay line in data memory
   l1           = length of delay line buffer
   m1           =1 - to modify index registers
   b8, i8       = start address of median filter buffer in program mem
   l8           =length of delay line buffer
   m9           =1 - to modify index registers

Return Values
   f4 = median of values in delay line

Registers Affected
    f0, f2, f4, f5, r15
Cycle Count
   FILTER_LEN + 6*[(FILTER_LEN+1)/2] + 14 + 3*sum[N]
   where N=(FILTER_LEN-1)/2 to FILTER_LEN-1
   99 cycles for FILTER_LEN=7

# PM Locations
    16 instructions + N Words of PM Data,
      where N is the order of the median filter

# DM Locations
```

```
        N Words, where N is the order of the median filter

*****************************************************************************/

    Execution Time:

*/

#define  FILTER_LEN 7                  /* must be an odd number */

.segment/pm pmcode;

.GLOBAL start_median;

start_median:

 /* xfer loop - transfer delay line data in DM to median filter buffer in PM */

    f0=dm(i1,m1);
    lcntr=FILTER_LEN-1, do xfer until lce;
xfer:    f0=dm(i1,m1), pm(i8,m9)=f0;   /* transfer to filter buffer */
                                       /* read from input buffer */
    pm(i8,m9)=f0;                      /* transfer to filter buffer */

/*
    median filter loop - find median value in delay line using this technique:

              for N=0 to (FILTER_LEN+1)/2    - outer loop

                   for M=N to FILTER_LEN     - inner loop

                        if (median[N] < median[M])

                             median[M]=median[N], median[M]=median[N]

                        inc M

                   inc N
*/

    b9=b8;                             /* i8, i9 point to median filter data */
    r15=FILTER_LEN;                    /* r15 is loop count for inner loop */

   /* each pass through the outer loop resolves the next greatest magnitude */
   /* in the median filter buffer - median[N]                                */

    lcntr=(FILTER_LEN+1)/2, do outer_loop until lce;

         f4=pm(i8,m9);                 /* read median[N] */
         modify(i9,1);                 /* i9 points to median[M] */
```

(listing continues on next page)

```
                                        /* where M=N+1 first */
        r15=r15-1, f5=pm(i9,m9);        /* decrement inner loop count */
                                        /* read median[M] */

 /* inner loop finds minimum of the remaining values in median filter buffer */

        lcntr=r15, do inner_loop until lce;
             f2=pass f5;                        /* f2=median[M] */
             comp(f2,f4), f5=pm(i9,m9);         /* cmp median[M], median[N] */
inner_loop:  if lt f4=pass f2, pm(-2,i9)=f4;    /* if median[M] < median[N] */
                                                /* median[N]=median[M] */
                                                /* median[M]=median[N] */
outer_loop:

        i9=i8;                                  /* init i9 to median[N+1] */

    rts;                                        /* return is non-delayed */
                                                /* 3 cycles              */

.endseg;
```

Listing 9.5 Floating-Point 1-D Median Fillter

9.5 REFERENCES

[GLASSNER90] Glassner, Andrew S., ed. 1990. *Graphics Gems*. San Diego, CA: Academic Press, Inc.

[GONZALEZ87] Gonzalez, R.C. and P. Wintz. 1987. *Digital Image Processing*. Reading, MA: Addison Wesley.

[JAIN89] Jain, Anil K. 1989. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall.

JTAG Downloader 10

The software application described in this chapter is no longer recommended or supported by Analog Devices. This application provided some guidelines for developing a boot loader that used the processor's Test Access Port (TAP) and minimal external hardware to load a small kernel program. When executed, the kernel program would load the remainder of the application into the processor.

For more information about the JTAG Downloader, contact applications support by email at processor.support@analog.com.

Index

Index

G

H

I

Index

J

L

M

Index

V

ADSP-21000 Family Development Software Release 3.2 Installation Guide and Release Note

#83-000855-03

SHARC
SUPER HARVARD ARCHITECTURE COMPUTER

1 GENERAL INFORMATION

1.1 Documentation

To order additional copies of any of the following publications, contact an Analog Devices sales agent. Documentation for use with this release includes the following:

- ***ADSP-21000 Family Assembler Tools and Simulator Manual***
- ***ADSP-21000 Family C Tools Library Manual***
- ***ADSP-21000 Family C Runtime Library Manual***
- ***ADSP-21020/ADSP-21010 User's Manual***
- ***ADSP-2106x SHARC User's Manual***
- ***ADSP-2106x SHARC EZ-ICE Hardware Installation Manual***
- ***3.2 Release Note***

1.2 What's Covered in this Release Note?

This release note contains information on the following software development tools for ADSP-21000 Family DSP microprocessors that is not described in, or has changed from, the descriptions in the corresponding user's manuals.

- ***Assembler***
- ***Linker***
- ***Librarian***
- ***Simulator***
- ***Emulator (EZ-ICE® In-Circuit Emulator)***
- ***PROM Splitter***
- ***Loader***
- ***CSWAP***
- ***CDUMP***
- ***G21K Optimizing C Compiler***
- ***G21K C Runtime Library***

1.3 Who are you?

We can better support you if you register. Please register your software by returning the enclosed registration form.

1.4 For Technical Assistance

Call DSP Applications Engineering at (617) 461-3672 (Massachusetts), (408) 879-3037 (California), (404) 263-3722 (Georgia), or (323) 2371672 (Europe).

1.5 For Bug Fixes and Documentation Errata

- Contact the BBS at (617) 461-4258, 8 bits data/no parity/1 stop bit/300/1200/2400/ 9600/14400 baud. Select the following menu items: (D) DSP Processor Information/(2) 210xx Family Information/(D) Development Tools/(N) New Bug Fixes

- Contact the FTP site at `ftp.analog.com` . The information is in the `/pub/dsp/dev_tool/21k_tool` directory in `readme.dev` .

1.6 Warranty

Analog Devices warrants G21K for users who have purchased this program as part of an ADSP-21000 Family Development Software package according to the terms and conditions of the ADSP-2100 and ADSP-21000 Family Development Software License Agreement provided with this software distribution, subject to the GNU General Public License, also provided therein.

2 SALES PACKAGE

This software package is available for both PC and SUN workstations. The part numbers are:

Part Number	*Includes this software...*
ADDS-210xx-SW-PC	Assembler, Linker, Librarian, Assembly Library, ADSP-21020 Simulator, ADSP-21020 Emulator, ADSP-2106x Simulator, ADSP-2106x Emulator, PROM Splitter, G21K C Compiler, C Runtime Library
ADDS-210xx-SW-SUN	Assembler, Linker, Librarian, Assembly Library, ADSP-21020 Simulator, ADSP-2106x Simulator, PROM Splitter, G21K C Compiler, C Runtime Library

3 SOFTWARE INSTALLATION

The installation procedures for the PC version and the Sun version of these tools differ. Note the requirements for your package and follow the appropriate instructions.

3.1 PC Version

The PC package contains a set of 3-1/2" high-density diskettes. Make sure that your disk drive is capable of reading the diskettes. If you require 5-1/4" diskettes, please specify this on your registration form to receive 5-1/4" diskettes.

3.1.1 PC Requirements

This software requires up to 13 MB of hard disk storage. The required system configuration to run the development software is a 386, or higher, based PC with hard disk, high-density floppy disk drive, a color video card and VGA monitor, and a minimum of 2 MB extended RAM. DOS 3.1 or higher. Microsoft® Windows™ 3.1 or higher is required to run Windows-based tools. A mouse is highly recommended.

You must have at least 450 Kb of memory available in the lower 640 Kb in order for the software to run properly. Most device drivers and TSRs consume this lower memory. Maximize the low memory available to the software by moving as many of these device drivers and TSRs into high memory.

The software opens many files and accesses these files frequently. Performance can be degraded, or fail in some cases, if DOS is not set up to support this. In the config.sys file, be sure the following directives are present and the values for these directives are at least equal to the minimum shown below:

- BUFFERS=25
- FILES=40

If you will be using the development software in native DOS, both Extended Memory Support (XMS) and Expanded Memory Support (EMS) are required. Use memory managers that support both standards and be sure not to disable EMS memory in your memory manager if you will be

running native DOS. If you will not be using the development software in native DOS, but will use the software in DOS boxes under Windows, only XMS memory support is required.

3.1.2 PC Installation Procedure

The software uses a Windows installation utility to load the software onto your hard drive. To install the software:

1. Insert the floppy into your PC.
2. Start Windows if it is not already running.
3. From the Program Manager or File Manager, select Run... from the File menu.
4. In the dialog box that appears, type the following
   "a:\install" (or "b:\install")
5. Follow the directions on the screen.
6. Reboot your PC after completing the installation for all changes to take effect.

3.1.3 PC Environment Variables

The development software requires several environment variables in order to run properly and efficiently. Because the installation procedure is Windows based, these environment variables must be set manually. This section describes what these environment variables are used for and how to change them. In addition to these environment variables, remember to include the development tools executable directory in your search path. All environment variables are typically setup in your autoexec.bat file. See the readme.txt file on the installation disk, and your installation directory, for more information.

Variable Name	*Typical Value*	*Description*
ADI_DSP	C:\ADI_DSP	Points to the root directory of the development tools software. Insure there is no trailing whitespace after the pathname.
TMP	C:\TMP	Points to a directory that all tools, except the compiler, will use for temporary file storage. For optimum performance from the tools, this should point to a sufficiently large RAM drive.
GO32	EMU C:\ADI_DSP\21K\ETC\EMU387	Points to directory that contains floating point emulation routines. This is only needed on PCs whose CPU does not support native floating point operation (ie: 386SX, 486SX)
GO32TMP	C:\TMP	Points to a directory that the compiler will use for temporary file storage. For optimum performance from the tools, this should point to a sufficiently large RAM drive.

3.2 Sun Version

The Sun package contains a 1/4" cartridge tape. Make sure that your tape drive is capable of reading this format.

3.2.1 Sun Requirements

The following are required for Sun installations:

System:	Sun4, Sparc based
Operating System:	SunOS 4.1.3
X Server:	X11R5 or OpenWindows 3.0
Window Manager:	mwm (Motif 1.2), olwm or twm
Disk Drive:	22 MB plus 48M swap space
Tape Drive:	quarter inch cartridge (QIC)
RAM:	12 MB
Misc:	Color monitor highly recommended, keyboard and mouse required

In addition, the development software, specifically the simulators, use the gnuplot graphics package from the Free Software Foundation for plotting memory. If you plan on using the plotting feature of the simulators, be sure you have version 3.5 of gnuplot in your executable search path. You can obtain gnuplot from the official distribution site; ftp.dartmouth.edu [129.170.16.4]. The file is called /pub/gnuplot/gnuplot3.5.tar.Z. Official mirror sites are, in Australia, monu1.cc.monash.edu.au [130.194.1.101] and, in Europe, irisa.irisa.fr [131.254.254.2].

Lastly, if using the Xnews server, be sure Xnews patch #100444-66 is installed. This patch can be obtained from Sun, via anonymous FTP, at sunsolve.sun.com. The file is called /pub/patches/100444-66.tar.Z.

3.2.2 Sun Installation Procedure

The software uses an installation utility to load the software, from the tape drive, onto your hard drive. To install the software:

1. Insert the tape into the tape drive.
2. Create a directory where the software will be installed, typically named adi_dsp.
3. Use the cd command to make this the default directory.
4. Extract the software from the tape with the following command:
   "tar xvf 'tape_drive' " (where 'tape_drive' is the name of your tape drive device, typically /dev/rmt0)
5. Move the "windows" directory, that was created in the ADI_DSP subdirectory to your home directory. This can be done by entering the following at the command line:
   mv "full pathname of the adi_dsp directory"/windows ~/.
6. Create the ADI_DSP environment variable and add the executable directory to your default path. This is typically done in your .cshrc file by adding the lines:
   setenv ADI_DSP "full pathname of the adi_dsp directory"
   set path = ($path $ADI_DSP/21k/bin)
   setenv MWHOME ${ADI_DSP}/mw

source ${MWHOME}/setup-mwuser.csh

7. Source your .cshrc file or, log out and then log back in again, for the changes to take effect.

3.2.3 Sun Environment Variables

When you followed the above installation procedure, you created all the environment variable needed by the software. This section describes what these environment variables are used for and how to change them.

Variable Name	*Default*	*Description*
ADI_DSP	adi_dsp	Points to the root directory of the development tools software. Insure there is no trailing whitespace after the pathname.
MWHOME	adi_dsp/mw	Points to the directory that contains the windowing library routines needed to run the simulators.

4 CHANGES FROM PREVIOUS RELEASE

This section describes the changes from release 3.1 to release 3.2.

4.1 Assembler

- Minor bug fixes.
- The ADSP-21060's memory-mapped IOP Registers are programmed by writing to the appropriate address in internal memory. The symbolic names of the IOP registers can also be used in ADSP-21060 programs—the `#define` definitions for these names are contained in the file def21060.h which is provided in the `INCLUDE` directory of the ADSP-21000 Family Development Software. The def21060.h file is also described in the ADSP-21060 User's Manual.

4.2 Assembly Run-Time Library

- Minor bug fixes.

4.3 Linker

- The -r switch is no longer supported.
- Minor bug fixes.

4.4 Loader

- Minor bug fixes.

4.5 Librarian

- The librarian now accepts absolute path names.
- Minor bug fixes.

4.6 Simulators

4.6.1 Version Specific Changes

4.6.1.1 PC Version

- All simulators are now true 16 bit Windows applications. DOS versions are no longer provided.

4.6.1.2 Sun Version

- The simulators no longer use a character based interface. Instead, they use a commercial windowing package, included with the software, that allows the user to select between a Windows look and feel, or a Motif look and feel.

4.6.2 Common Changes

- Full SHARC IOP bitfield decoding and editing.
- Enhanced memory window tracking by PC, DAG index registers and DMA index registers.
- Format and tracking info shown in memory window title bars.
- Font selection now available via pulldown menu and initialization file variable “font_size”.
- Capability to load only symbols from an executable file via the pulldown menu.
- Boot loader file support now available via the initialization file variable “ldr_file”.
- An automatic run mode now available via the initialization file variable “batch_mode”. If “batch_mode” is nonzero, the simulator will exit after ‘n’ “validation_cycles”. (see below)
- An automatic termination of simulation after ‘n’ cycles by specifying a nonzero value for the initialization file variable “validation_cycles”.
- The MS bits of the MODE2 register now reflect the “.PROCESSOR” directive for the simulator.
- Memory mapped port and IOP serial port file input and output now supported from the pulldown menu.
- Memory mapped port description file support now available from the pulldown menu and the initialization file variable “prt_file”.
- CBUG now halts properly when running and a key is pressed.
- Profile data dumps now work properly.
- The initialization files used by the simulators, wsim060.ini and wsim020.ini, are located in the Windows directory. The settings in these files replace the command line parameters used in traditional DOS programs.
- The simulators use the GNU plot utility, “wgnuplot”, to perform plotting of data. Memory plot commands result in the creation of plot files that are passed to the wgnuplot program, that is spawned to display the plot windows. See the files wgnuplot.txt and wgnuplot.hlp for more information.

- The simulators can generate and send debug information to the standard AUX device. The initialization file contains a verbose mode variable, which enables or disables this information. In order to display this information, you may either run the dbwin application, supplied with this development software, or connect a monochrome monitor. The system.ini file, located in your windows directory, has been modified to insure this information is turned off by default. Specifically, the [debug] section has been modified to include the line “OutputTo=NUL”.

4.7 Emulator

- All emulators are now true 16 bit Windows applications. DOS versions are no longer provided.
- Full SHARC IOP bitfield decoding and editing.
- Enhanced memory window tracking by PC, DAG index registers and DMA index registers.
- Format and tracking info shown in memory window title bars.
- Font selection now available via menu pulldown and initialization file variable “font_size”.
- Capability to load only symbols from an executable file via the menu pulldown.
- Significantly improved speed of memory transfers between host and target.
- Control of background DMA activity, when target is halted, via initialization file variable “dma_stop”.
- The display can now be refreshed by the Ctrl-Alt-Shift G key.
- The initialization files used by the emulators, wice060.ini and wice020.ini, are located in the Windows directory. The settings in these files replace the command line parameters used in traditional DOS programs.
- The emulators use the GNU plot utility, “wgnuplot”, to perform plotting of data. Memory plot commands result in the creation of plot files that are passed to the wgnuplot program, that is spawned to display the plot windows. See the files wgnuplot.txt and wgnuplot.hlp for more information.
- The emulators can generate and send debug information to the standard AUX device. The initialization file contains a verbose mode variable, which enables or disables this information. In order to display this information, you may either run the dbwin application, supplied with this development software, or connect a monochrome monitor. The system.ini file, located in your windows directory, has been modified to insure this information is turned off by default. Specifically, the [debug] section has been modified to include the line “OutputTo=NUL”.

4.8 Splitter

Minor bug fixes.

4.9 C Language Tools

- Better error detection for command line switches.

4.9.1 G21K Preprocessor

- C++ style comments (“//”) are now accepted by the preprocessor.
- G21K now takes `.i` and `.is` files as inputs to skip pre-processing.

4.9.2 G21K Optimizing C Compiler

- The register allocation function of the compiler has been overhauled. The compiler now makes much better use, and selection of, registers. This improvement will significantly improve the compiler's code generation and efficiency.
- The compiler no longer inserts `NOPs` after jumps in situations where they are not required.
- Compiler generated code that modified DAG registers to span from internal to external memory has been improved to work around ADSP-2106x silicon restrictions. See restrictions section for related limitations.
- All compiler generated symbols now are prefixed by `_L$` for easier identification.
- Code selection of `Fx = Fx -Fx` for zeroing a register has been changed to `Rx = Rx - Rx`. The previous code selection caused problems with non-IEEE formatted numbers.
- There exists a silicon restriction that does not allow branch instructions, call or jump, to be placed in the last two locations of a DO loop. If the compiler generates a branch instruction in the third location from the end of a DO loop and cannot fill the last two instructions in the DO loop with meaningful code, the compiler will fill the last two locations with NOPs.
- The compiler generates all branches as PC-relative operations rather than direct addressing. This feature facilitates relocatable code. The compiler does not use PC-relative operations for the `cjump` instruction. The ADSP-21060 silicon does not currently support relative addressing with `cjump`.
- In order to better support operating systems, the compiler now supports the -mainrts switch. This switch forces the function `main()` to be treated as a true function call, with appropriate prolog and epilog code, rather than a simple jump from the run time header.

4.9.3 G21K Run Time Library

- `stdlib.h` has been fixed so that RAND_MAX is defined to be (2^32 - 1) instead of (2^16).
- `fir` routine now restores registers correctly.
- `frexp` and `frexpf` now return properly if an invalid argument was passed in.
- `tan`, `tanf`, `pow`, `powf` and `sinf` now function correctly when passed negative arguments.

4.10 COFF Tools

4.10.1 CSWAP

- No changes.

4.10.2 CDUMP

- No changes.

5 RESTRICTIONS

This section details restrictions in this version of the ADSP-21000 Family Development Software. The restrictions described in this section are supplemental to those described in the corresponding user's manual.

5.1 Assembler

- When using the backslash (\) line continuation character, do not put any characters after the backslash on the same line; it must be followed immediately by a carriage return. Otherwise, an assembler error occurs. For example, a comment or a space after a backslash causes an assembler error.
- When using the `Rn=FDEP Rx BY <bit6>:<len6>` instruction, use only positive numbers for `<bit6>`. A negative number for `<bit6>` results in a syntax error that is not properly reported.
- When multiple instructions are placed on a single line in the source file, the listing file shows only the last instruction on the line.
- Unprintable special characters in the source file may cause the assembler to crash.
- Macro substitutions made with a `#define` statement cannot begin with a number. The following example will fail:

```
#define 2me     r0 = 1
```

- In the architecture file, the END value, must be greater than the BEGIN value when defining a segment. If not, when computing the size of a segment, the assembler incorrectly interprets the resulting negative number as a large positive number.
- In the include directory, there is a file called asm_sprt.h, in which are defined various macros to aid the user in writing assembly level implementations of 'C' functions. The use of this file is described in the 'C' Tools Manual. Omitted from the manual is a statement indicating this file is to be used only as an include file in 'C' programs. Unfortunately, when the file is included from an assembly level routine, the pre-processor defines that are automatically generated by the compiler, are not asserted. As such, what get's included from the file are ADSP-2106x definitions, which is fine if that is your target processor. If your target processor is an ADSP-21020, you must manually define the processor define, __ADSP21020__, in order to include the proper ADSP-21020 definitions.

5.2 Assembly Run-Time Library

- None.

5.3 Linker

- The -s switch, used to strip symbolic information from an executable file does not work reliably.
- The linker will generate an error if the architecture file defines a multiprocessor memory space segment for processor 1, ID = 001, that defines locations 0x80000 thru 0x9ffff. This is because that memory space is in fact locations 0x00000 thru 0x7ffff of that processor.
- In the architecture file, the END value, must be greater than the BEGIN value when defining a segment. If not, when computing the size of a segment, the linker incorrectly interprets the resulting negative number as a large positive number.

5.4 Loader

- For link booting to function properly, the file "060_link.exe" must be copied from the 21k/examples/06x/linkboot/linkr0 or linkr1 directory into the 21k/etc directory.
- The boot loader does not make special provision for an executable that can be loaded by the automatic hardware boot alone. It creates a boot file that contains the boot loader, just to load

over itself. Although this works without a problem, it generates a boot loader file that is significantly larger (more than twice the size) than necessary.

- The Loader does not efficiently pack the final initialization. Regardless of the size of the last initialization, the loader places 256 48-bit words in the boot file.
- When using the loader, do not use the mem21k utility. Any executable that is going to be processed by ldr21k should not be processed with mem21k. See the "-nomem' compiler switch.
- The loader requires the use of address 0x20004 for booting an ADSP-2106x target. Any executable file that is going to be processed by ldr21k, should have a `NOP` or an `IDLE` instruction at address 0x20004. This only applies to ADSP-2106x targets.
- As noted in the users manual, the DMAC6 control register will not contain a zero when the program begins execution.
- If a single ADSP-2106x has a processor ID that does not equal zero, a dummy executable file must be placed on the command line or else the boot will not occur properly. When the boot loader is executed on an ADSP-2106x with a non-zero ID, it expects to read a table of contents before the boot data begins. This table of contents points to the proper place in the ROM that holds the code for each different processor. The ldr21k tool only places a table of contents at the beginning of a ROM file if more than one executable is given on the command line. A dummy executable can be created by assembling and linking a file that contains a single `NOP`.
- The -fbinary switch has not been fully tested and, although implemented, is not supported for this release.

5.5 Librarian

- None.

5.6 Simulators

5.6.1 Version Specific Restrictions

5.6.1.1 PC Version

- The source file sizes in CBUG are restricted to less than 64Kb.
- The Program Manager icons for the simulators were inadvertantly omitted.

5.6.1.2 Sun Version

- The input and output filenames used for simulated SPORT data transfers must be in lower case.
- When selecting menu items, the choice currently selected does not highlight.
- When using the "click and drag" method of traversing menu hierarchies, traversing backwards in the hierarchies does not work. Using the "click and release" method does work.

5.6.2 Common Restrictions

- The following pulldown menus are not implemented for this release:

  - Save Layout
  - Execute Instruction
  - Counting Breakpoints
  - Auto-Interrupt
  - Auto-Flag
  - Backtrace
- The ADSP-2106x silicon prioritizes DMA channels as follows: 0, 1, 2, 3, ChainLoad, ExtRW, 4, 5, 6, 7, 8, 9. The simulator has the priority of ChainLoad and ExtRW reversed.
- When entering instructions into a PM window, “if ... else” instructions must be entered with a comma (,) before the else.
- The simulator allows writing to the following read-only and reserved bits in the IOP memory space:

  LSRQ: 31-20 and 19-16

  STCTLx: 28-24

  LCTL: 31-30

  LCOM: 31-26, 25-23, 11-0

  LAR: 31-30

  SYSCON: 7

  STKY: 26-21
- The simulator incorrectly initializes the multiprocessing vector interrupt register, VIRPT, to 0x40014 (external). The correct value should be 0x20014 (internal).
- The simulator’s display for the bus timeout counter, BCNT, incorrectly displays the bus timeout maximum value, BMAX.
- The simulator does not automatically clear it’s symbol table when a new executable is loaded.
- The simulator does not honor the short word sign extend bit, SSE, of the MODE1 register.
- Displays for the serial port FIFO buffers will be available in a future release.
- Variable names which are identical to the segment name in which they reside cause the simulator to omit the variable name from the symbol table. The practice of naming symbols and segments with identical names should be avoided.
- The architecture file “.BANK” directive has no affect. The various “.BANK” qualifiers are not asserted by the simulator. In addition, wait states MSIZE, IMDWx bits etc., are not set automatically. These bits should be set by the user’s code at run time.
- The ADSP-2106x simulator does not support link ports. SPORT DMA transfers are supported. External port master mode DMA transfers with external memory functionality is included but not tested nor supported.
- Host and PROM booting options are the only boot loading operation that the ADSP-2106x simulator supports. Host boot files must be in ASCII format as created by the loader, ldr21k, with the -fascii switch.
- Arithmetic loops (i.e. non-counter-based loops) containing one or two instructions that terminate on the first iteration of the loop are not properly simulated. Workaround: Insert `NOP` instructions into the body of the loop to make it at least 3 instructions long.

- When enabling or disabling the timer through the MODE2 register, the timer may exhibit an extra cycle of latency before the change takes effect. This extra cycle of latency may also occur when the Auto-Interrupt Control and Auto-Flag Control functions are used.
- If an assembly label coincides with other labels, only the last label linked will be seen by the simulator symbols display and search.
- In the DAGs, when setting the base registers via the user interface, the corresponding index register gets set also. This operation differs from the emulators where the index register does not get set.
- The Buffer Hang Disable (BHD) bit in the ADSP-2106x SYSCON register defaults to a clear state. This means the processor core will hang by default if memory-mapped FIFOs are read-when-empty or written-when-full.
- The simulator does not support both serial ports running in loopback simultaneously. The simulator does support loopback of either serial port, provided only 1 serial port is in loopback at a time.
- DMA SPORT transfers do not halt properly in the simulator. When a DMA count register decrements to zero, and no DMA chaining is called for, the DMA channel should disable itself. However, the DMA activity bit, in the DMASTAT register, is not cleared and the SPORT buffer status bits are incorrect.
- The Memory Dump command only supports hexadecimal syntax. If dumps in floating point format are required, use the plot memory command, and examine the resulting .plt file.
- In the ADSP-2106x simulator, if the multiplier underflow bit, MUS, of the STKY register is set, the simulator incorrectly clears the floating point underflow bit, FLTUI, of the IRPTL register when the FLTUI interrupt is serviced.
- In the status stack windows, the simulator allows the user to modify the FLAG bits, bits 19 through 22, of the ASTAT registers. These simulator should not allow modification of these bits.
- In the ADSP-21020 simulator, Program Memory windows do not support the floating point display mode.
- In the ADSP-21020 simulator, the BSET instruction will not simulate properly if the bit that is to be set lies in bit position 15 through 31. Note the following run-time library functions may not simulate properly as they employ use of this function; A-Law/ U-Law encoding, sin, atof, strtod, dtoi, dmult, dadd and dsub.
- In the ADSP-2106x simulator, memory references to reserved locations in IO space will result in the “Error --- Instruction Timed Out” message.
- The simulator does not support 40 bit fixed point values with the LOAD command; fixed point values are limited to 32 bits.
- After a chip reset, all simulators, include the 2 clock cycles that are used to fill the fetch/decode/execute pipeline, in their cycle counters. The EZ-ICEs do not count these 2 cycles.

5.7 Emulator

This section describes the restrictions for each ADSP-210xx EZ-ICE emulator.

5.7.1 Common Restrictions

This section describes restrictions that are common to all ADSP-210xx EZ-ICE emulators.

- The following pulldown menus are not implemented for this release:
  - Save Layout
  - Execute Instruction
  - Counting Breakpoints
  - Auto-Interrupt
  - Auto-Flag
  - Backtrace
- The Program Manager icons for the emulators were inadvertantly omitted.
- The emulator does not automatically clear it's symbol table when a new executable is loaded.
- The source file sizes in CBUG are restricted to less than 64Kb.
- The architecture file ".BANK" directive has no affect. The various ".BANK" qualifiers are not asserted by the emulator. In addition, wait states MSIZE, IMDWx bits etc., are not set automatically. These bits should be set by the user's code at run time.
- Variable names which are identical to the segment name in which they reside cause the emulator to omit the variable name from the symbol table. The practice of naming symbols and segments with identical names should be avoided.
- The Memory Dump command only supports hexadecimal syntax. If dumps in floating point format are required, use the plot memory command, and examine the resulting .plt file.
- In the DAGs, when setting the base registers via the user interface, the corresponding index register does not get set. This operation differs from the simulator where the corresponding index register's value is set when the base register is set.
- Software breakpoints can not be placed anywhere that a `CALL` instruction would be invalid (i.e., either of the two instructions following a Delayed Branch jump or call, or last three instructions of a `DO` loop).
- When loading an executable with `Memory Verify ON`, any ports that are defined may report a readback error. Ignore this or turn `Memory Verify OFF`.
- PM locations, 0 through 7 for ADSP-21020 and 0x20000 through 0x20003 for ADSP-2106x, must not contain any user code and should be set to 0 (`NOP`).
- Interrupts to the 21020 are not latched when the emulator is halted.
- Single-step does not wait on `IDLE` instructions (i.e., executes as a `NOP`).
- Three locations of the PC stack are reserved for the emulator.
- IRPTL and IMASKP have bit 0 set while stepping.
- Timer continues to operate while software breakpoints are encountered (for approximately 4 cycles with zero wait states).
- If the timer interrupt gets serviced while a software breakpoint is executed, the status stack may not get cleaned up properly; there may be an extra MODE1/ASTAT pair on the stack.
- PMDA instructions perform both program memory accesses when single-stepping (for example, if the cache contains the instruction to be fetched, it may fetch the instruction anyway).
- Software breakpoints remain in memory if you select the `RUN` command then `EXIT` or `QUIT`. This results in the opcode at the breakpoint address being lost (breakpoint address will contain a `CALL`).

- When entering instructions into a PM window, “if ... else” instructions must be entered with a comma (,) before the else.
- ***Do not*** set the IMASKP bit 0 (EMUI). This causes the processor to enter emulator space.
- The [user_sect] “base_address” variable in the wice0x0.ini file must be set correctly according to the actual jumper configuration on the emulator PC plug-in board. The default address for the board is 0x340. See the EZ-ICE Emulator Manual for complete hardware installation details.
- FADDR and DADDR always display the current PC value in the Program Counters window.
- After a chip reset, all simulators, include the 2 clock cycles that are used to fill the fetch/decode/execute pipeline, in their cycle counters. The EZ-ICEs do not count these 2 cycles.

5.7.2 ADSP-21010/ADSP-21020 EZ-ICE Specific Restrictions

This section describes restrictions that are common to all ADSP-21010/ADSP-21020 EZ-ICE emulators.

- Stepping over, or setting breakpoints on, the `push loop` instruction, or the two instructions following a `push loop` instruction, can cause unpredictable results.
- When any of the ADSP-21020 stacks (PC, status, loop) overflow, the ADSP-21020 must be reset to use those stacks. The stack contents displayed after an overflow are all bits set for the PC stack and all bits cleared for Loop and Status stacks.
- When the target is ***not*** running, and the target asserts $\overline{BR}$, the ADSP-21020 asserts $\overline{BG}$. None of the address or control pins are 3-stated as expected, however.
- Read-only registers (i.e., FADDR, DADDR, PMADR, DMADR, etc.) cannot be set from the user interface.
- PMI hardware breakpoint placed on the instruction immediately following a program memory data access is ignored until the instruction is cached. Workaround: Use a software breakpoint.
- The cache is loaded in a different, but valid, sequence. This ***does not*** cause the execution sequence to be different.
- Software breakpoints should not be placed on an instruction immediately following an instruction that unmasks a pending interrupt.
- Program Memory windows do not support the floating point display mode.

5.7.3 SHARC EZ-ICE Specific Restrictions

- DMA transfers which are halted by the emulator (see [dma_stop] variable in wice060.ini) are not resumed correctly.
- The input clock rate of the ADSP-2106x target must be >= 20 MHz.
- The ADSP-2106x must be powered by a nominal 5 VDC. The emulator interface pod does not support a 3.3 VDC supply.
- The JTAG port must be reset after powerup by either an active emulator probe or by the TRST pin. The emulator probe leaves the TRST open which will cause the TRST pin to float high. If the probe is inactive, ie; the emulator application is not running, the JTAG port will not be reset. If the target power supply is cycled in this situation, the JTAG port will come up in an undefined state and may cause improper operation of the processor. The part will remain in this state until the emulator application is started.

- If, upon startup, the architecture file listed in the initialization file contains a ".processor" statement that conflicts with the target processor, the error message displayed does not include the filename of the architecture file. For example, the error message displayed is

  "`Invalid .PROCESSOR directive in .`"

  rather than

  "`Invalid .PROCESSOR directive in c:\21k.ach.`"

- When using the "command_timeout" feature to automatically stop the emulator after 'n' milliseconds, there is a slight chance that the emulator will halt and then automatically restart. This can happen if the emulator encounters a breakpoint at the same instant as the 'n' milliseconds timer expires.
- When using the EZ-ICE with the EZ-LAB board and DspHost software library, resetting the 2106x, while the library is in the middle of a PC to EZ-LAB data transfer may cause the PC to hang.
- The EZ-ICE does not save and restore the instruction cache contents during emulator halts/runs and single steps. As a result, when benchmarking code, run the code under test from start to finish with no breakpoints, halts or single steps.

5.8 Splitter

- None.

5.9 C Language Tools

5.9.1 G21K Preprocessor

- None, other than those described in the user's manual.

5.9.2 G21K Optimizing C Compiler

- The -mpcrel switch, to generate PC relative code, does not generate PC relative code in all cases. As such, this switch should be avoided.
- Although the compiler generated code that modified DAG registers to span from internal to external memory has been improved to work around ADSP-2106x silicon restrictions, certain code sequences can cause problems. For example, the following sequence will cause the compiler to generate code that violates the ADSP-2106x restriction:

```
int i, j;
int *ptr;
ptr = (j + (int *)i);
```

- Although the compiler has been enhanced to minimize the modification of DAG registers that span
- With Release 3.1, a new function calling convention was implemented. To maintain the Release 3.0 calling convention, use the `-mkeepi13` switch.

5.9.3 G21K Numerical 'C' Extensions

- None, other than those described in the user's manual.

5.9.4 G21K Run Time Library

- The files strspendp.asm and set_proc21k.asm were incorrectly named on the install disks for the PC version. After they are installed on the user's disk, they appear as strspzba.asm and set_pzaq.asm.
- This release does not include optimized FFT routines. These routines will be placed on the BBS when available.
- The `atof()` and `strtod()` library functions require you to use the `-fno-short-double` switch.
- The `frexp()` and `modf()` library functions require you to use the `-fno-short-double` switch, or rename them as `frexpf()` and `modff()`.
- The compiler's built-in code does not check for large numbers; this could result in compiler errors. When `sqrtf()` input exceeds `FLT_MAX/2` (1.7e38) and the `-O2` switch is used, a negative value may be returned. Use the `-fno-builtin` switch and `-O2` switch to avoid this.
- The `sqrt()` function returns negative values when the input exceeds `FLT_MAX/2 (1.7e38)`. Use the `-mdont-inline-sqrt` switch or use `sqrtf()` with the `-fno-builtin` and `-O2` switch instead. The compiler's built-in code does not check for large numbers; this could result in compiler errors.
- The `rsqrt()` and `rsqrtf()` functions do not work.
- The `qsort()` function is not in the library.
- The `matinv()` function is no longer supported.
- The `acosf()` function is unreliable. Use `acos()` instead.
- The `clear_interrupt()` routine does not work for interrupts that set STKY register bits, such as floating point overflow.
- The `zero_cross()` function does not work.
- The `expf()`, `exp()` , and `cexpf()` functions do not return `HUGE_VAL` or set `ERANGE` in `errno` when there is an overflow error.
- When polymorphic functions (functions that use `pm` pointers) are used, and the function returns a pointer to program memory, cast the output of the function to pm. For example: `(char pm *)`

5.10 COFF Tools

5.10.1 CSWAP

- None.

5.10.2 CDUMP

- None.