TMS320C54x DSP Programmer's Guide

Literature Number: SPRU538
July 2001

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

This manual provides basic examples and optimization techniques for use when writing code for the TMS320C54x™ DSPs.

Notational Conventions

This document uses the following conventions.

- The device number TMS320C54x is often abreviated as C54x.
- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

  Here is an example of a system prompt and a command that you might enter:

```
C:  csr –a /user/ti/simuboard/utilities
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect "*section name***",** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value$_1$ [, ... , value$_n$]*

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

- In most cases, hexadecimal numbers are shown with the suffix h. For example, the following number is a hexadecimal 40 (decimal 64):

  40h

  Similarly, binary numbers are shown with the suffix b. For example, the following number is the decimal number 4 shown in binary form:

  0100b

- Bits are sometimes referenced with the following notation:

Notation	Description	Example
Register(n–m)	Bits n through m of Register	AC0(15–0) represents the 16 least significant bits of the register AC0.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C54x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set (literature number SPRU172) describes the TMS320C54x™ digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set (literature number SPRU179) describes the TMS320C54x™ digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 4: Applications Guide (literature number SPRU173) describes software and hardware applications for the TMS320C54x™ digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510™ emulator.

TMS320C54x Simulator Getting Started Guide (literature number SPRU137) describes how to install the TMS320C54x simulator and the C source debugger for the C54x. The installation for Windows 3.1, SunOS™, and HP-UX™ systems is covered.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C54x™ generation of devices.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the TMS320C54x™ C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for the TMS320C54x generation of devices.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x™ assembly language tools and the C compiler for the TMS320C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x DSP Library Programmer's Reference (literature number SPRU518) describes the optimized DSP Function Library for C programmers on the TMS320C54x DSP.

Trademarks

Code Composer Studio, TMS320C54x, C54x, TMS320C55x, and C55x are trademarks of Texas Instruments.

Contents

Figures

Tables

Examples

Equations

Chapter 1

TMS320C54x Architectural Overview

This chapter lists some of the key features of the TMS320C54x™ (C54x) DSP architecture.

Topic **Page**

1.1 TMS320C54x Overview

The C54x has a high degree of operational flexibility and speed. It combines an advanced modified Harvard architecture (with one program memory bus, three data memory buses, and four address buses), a CPU with application-specific hardware logic, on-chip memory, on-chip peripherals, and a highly specialized instruction set. Spinoff devices that combine the C54x CPU with customized on chip memory and peripheral configurations have been, and continue to be, developed for specialized areas of the electronics market.

The C54x devices offer these advantages:

- ❏ Enhanced Harvard architecture built around one program bus, three data buses, and four address buses for increased performance and versatility
- ❏ Advanced CPU design with a high degree of parallelism and application-specific hardware logic for increased performance
- ❏ A highly specialized instruction set for faster algorithms and for optimized high-level language operation
- ❏ Modular architecture design for fast development of spinoff devices
- ❏ Advanced IC processing technology for increased performance and low power consumption
- ❏ Low power consumption and increased radiation hardness because of new static design techniques

1.2 TMS320C54x Key Features

Key CPU core and instruction set features of the C54x DSPs include:

- CPU
  - Advanced multibus architecture with one program bus, three data buses, and four address buses
  - 40-bit arithmetic logic unit (ALU), including a 40-bit barrel shifter and two independent 40-bit accumulators
  - 17-bit × 17-bit parallel multiplier coupled to a 40-bit dedicated adder for nonpipelined single-cycle multiply/accumulate (MAC) operation
  - Compare, select, store unit (CSSU) for the add/compare selection of the Viterbi operator
  - Exponent encoder to compute the exponent of a 40-bit accumulator value in a single cycle
  - Two address generators, including eight auxiliary registers and two auxiliary register arithmetic units
  - Dual-CPU/core architecture on the 5420
- Instruction set
  - Single-instruction repeat and block repeat operations
  - Block memory move instructions for better program and data management
  - Instructions with a 32-bit long operand
  - Instructions with 2- or 3-operand simultaneous reads
  - Arithmetic instructions with parallel store and parallel load
  - Conditional-store instructions
  - Fast return from interrupt

Chapter 2

Improving System Performance

This chapter introduces features of the TMS320C54x™ (C54x) DSP that improve system performance. These features allow you to conserve power and manage memory. You can improve the performance of any application through efficient memory management.

Topic **Page**

2.1 Tips for Efficient Memory Allocation

- **Tip: Carefully plan your SARAM vs DARAM data allocation.**

  The C54x can access minimum 64K words of program and 64K words of data memory. On-chip memory accesses are more efficient than off-chip memory access, since there are eight different internal buses on the C54x but there is only one external bus for off-chip accesses. This means that an off-chip operation requires more cycles than that of an on-chip operation.

  In cases where the DSP uses wait-state generators to interface to slower memories, the system, cannot run at full speed. If on-chip memory consists of dual access RAM (DARAM), accessing two operands from the same block does not incur a penalty. Using single access RAM (SARAM), however, incurs a cycle penalty.

- **Tip: For random-access variables, use direct addressing and allocate them in the same 128-word page.**

  Random-access variables use direct addressing mode. Data-page relative direct memory addressing makes efficient use of memory resources. Allocating all the random variables on a single data page saves some extra CPU cycles.

  Sometimes data variables have an associated lifetime. When that lifecycle is over, the data variables become useless.. Thus, if two data variables have non-overlapping lifetimes, both can occupy the same physical memory. The **UNION** directive in the linker command file allows two or more data variables share the same physical memory location

- **Tip: If required, reserve CPU resources for the exclusive use of interrupts.**

  The actual lifetime of a variable determines whether it is retained across the application or only in the function. By careful organization of the code in an application, resources can be used optimally. Aggregate variables, such as arrays and structures, are accessed via pointers located within that program's data page, but the actual aggregate variables reside else where in the data memory. Depending upon the lifetime of the arrays or structures, these can also form unions accordingly.

  Interrupt driven tasks require careful memory management. Often, programmers assume that all CPU resources are available when required. This may not be the case if tasks are interrupted periodically. These interrupts do not require many CPU resources, but they force the system to respond within a certain time. To ensure that interrupts occur within the specified time and the interrupted code resumes as soon as possible, you

must use low overhead interrupts. If the application requires frequent interrupts, you can set aside some of the CPU resources for these interrupts. When all CPU resources are used, simply saving and restoring the CPU's contents increases the overhead for an interrupt service routine (ISR).

Dedicated auxiliary registers are useful for servicing interrupts. Allowing interrupts at certain places in the code permits the various tasks of an application to reuse memory. If the code is fully interruptible (that is, interrupts can occur anywhere and interrupt response time is assured within a certain period), memory blocks must be kept separate from each other. On the other hand, if a context switch occurs at the completion of a function rather than in the middle of execution, the variables can be overlapped for efficiency. This allows variables to use the same physical memory addresses at different times.

2.2 Memory Alignment Requirements

C54x data placement in memory must comply with the following requirements:

- ❑ Long words must be aligned at even boundaries for double-precision operations; that is, the most significant word at an even address and the least significant word at an odd address.

- ❑ Circular buffers should be aligned at a K boundary, where K is the smallest integer that satisfies $2^K > R$ and R is the size of the circular buffer. Use the **align** directive to align buffers to correct sizesIf an application uses circular buffers of different sizes, allocate the largest buffer size as the first alignment, the next highest as the second alignment, and so forth. Example 2–1 shows the memory management alignment feature where the largest circular buffer is 1024 words, and therefore, is assigned first. A 256-word buffer is assigned next. Unused memory can be used for other functions without conflict.

Example 2–1. Memory Alignment Example

```
DRAM      : origin = 0x0100, length = 0x1300
inpt_buf  : {} > DRAM,align(1024)PAGE 1
outdata   : {} > DRAM,align(1024)PAGE 1
UNION     : > DRAM align(1024) PAGE 1
{
   fft_bffr
   adpt_sct:
   {
   *(bufferw)
   .+=80h;
   *(bufferp)
   }
}
UNION     : > DRAM align(256) PAGE 1
{
   fir_bfr
   cir_bfr
   coff_iir
   bufferh
   twid_sin
}
UNION     : > DRAM align(256) PAGE 1
{
   fir_coff
   cir_bfr1
   bufferx
   twid_cos
}
```

2.3 Stack Initialization

Stack allocation can also benefit from efficient memory management. The stack grows from high to low memory addresses. The stack pointer (SP) decrements by 1 before pushing a new element onto the stack and post increments after a pop. The bottom location of the stack added to the stack size gives the actual starting location of the stack pointer. The last element on the stack is always empty. Whether the stack is on chip or off chip affects the cycle count during the stack accesses.

Example 2–2 shows stack initialization when the application is written in assembly. The variable SYSTEM_STACK holds the size of the stack. It is loaded into the SP, which points to the end of the stack.

Example 2–2. Stack Initialization for Assembly Applications

```
K_STACK_SIZE .set      100
STACK        .usect    “stack”, K_STACK_SIZE
SYSTEM_STACK .set      STACK+K_STACK_SIZE
       .ref   SYSTEM_STACK
       STM    #SYSTEM_STACK, SP  ; initialization
                                 ; of SP– this is done
                                 ; vectors.asm
stack : {} DRAM    PAGE 1        ; initialization of stack
                                 ; in linker command file
```

Example 2–3 shows stack initialization by c_int00 routine from the C runtime support library(rts.lib) when the application is written in C.The compiler uses the stack to allocate local variables, pass arguments, and save the processor status. The stack size is set by the linker and the default size is 1 K words. If 1K words of stack is more than necessary, allocate a smaller size stack by using the **stack** directive in the linker command file and utilize the freed up memory for other data variables.

Example 2–3. Stack Initialization c_int00 routine

```
.text
_c_int00:
*****************************************************************************
*    Init Stack Pointer. Remember stack grows from high to low address      *
*****************************************************************************

   STM #_stack, SP                ; set to begging of stack memory
   ADDM #(_STACK_SIZE – 1), *(SP) ; add size to get to the top
   ANDM #0FFFEh, *(SP)            ; make sure it is an even address
```

2.4 Overlay Management

Some systems use a memory configuration in which all or part of the memory space is overlaid. This allows the system to map different banks of physical memory into and out of a single address range. Multiple banks of physical memory can overlay each other at the same address space. In the C54x, you can:

- Overlay on-chip program and data memory.

  This is achieved by setting the OVLY bit in the PMST register. This is particularly useful in loading the coefficients of a filter, since program and data use the same physical memory.

- Overlay off-chip memory to achieve more than 64K words.

  If an application needs more than 64K words of either data or program memory, two options are available: The first one is to use one of the C54x derivatives that provides more than 16 address lines to access more than 64K words of program space. The other option is to use an external device that provides upper addresses beyond the 16-bit memory range. The DSP writes a value to a register located in its I/O space, whose data lines are the higher address bits. It implements bank switching to cross the 64K boundary. Some devices have Bank Switch Control Register to select memory bank boundary size. Since the bank switch requires action from the DSP, frequent switching between the banks is not very efficient. It is more efficient to partition tasks within a bank and switch banks only when starting new tasks.

2.5 Memory-to-Memory Moves

There are various reasons for performing memory-to-memory moves. These reasons include making copies of buffers to preserve the original, moving contents from ROM to RAM, and moving copies of code from their load location to their execution location. Example 2–4 implements memory-to-memory moves on the C54x using single-instruction repeat loops.

Example 2–4. Memory-to-Memory Block Moves Using the RPT Instruction

```
;
      .mmregs
      .text
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVDD instruction to move
; information in data memory to other data memory
; locations.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MOVE_DD:
   STM   #4000h,AR2   ;Load pointer to source in
                      ;data memory.
   STM   #100h,AR3    ;Load pointer to
                      ;destination in data memory.
   RPT   #(1024–1)    ;Move 1024 value.
   MVDD  *AR2+,*AR3+
   RET
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVDP instruction to move external
; data memory to internal program memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
MOVE_DP:
   STM   #0E000h,AR1  ;Load pointer to source in
                      ;data memory.
   RPT   #(8192–1)    ;Move 8K to program memory space.
   MVDP  *AR1+,#800h
   RET
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the MVPD instruction to move external
; program memory to internal data memory.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MOVE_PD:
   STM   #0100h,AR1   ;Load pointer to
                      ;destination in data memory.
   RPT   #(128–1)     ;Move 128 words from external
   MVPD  #3800h,*AR1+ ;program to internal data
                      ;memory.
   RET
```

Example 2–4.Memory-to-Memory Block Moves Using the RPT Instruction (Continued)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the READA instruction to move external
; program memory to internal data memory. This differs
; from the MVPD instruction in that the accumulator
; contains the address in program memory from which to
; transfer. This allows for a calculated, rather than
; pre-determined, location in program memory to be
; specified. READA can access locations in program memory
; beyond 64K word boundary
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
READ_A:
      STM   #0100h,AR1   ;Load pointer to
                         ;destination in data memory.
      RPT   #(128–1)     ;Move 128 words from external
      READA *AR1+        ;program to internal data
                         ;memory.
      RET
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine uses the WRITA instruction to move data
; memory to program memory. The calling routine must
; contain the destination program memory address in the
; accumulator. WRITA can access program memory address
; beyond 64K word boundary
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
WRITE_A:
      STM   #380h,AR1    ;Load pointer to source in
                         ;data memory.
      RPT   #(128–1)     ;Move 128 words from data
      WRITA *AR1+        ;memory to program memory.
      RET
```

2.6 Efficient Power Management

The C54x family of DSPs exhibits very low power dissipation and flexible power management. This is important in developing applications for portable systems, particularly wireless and hand-held systems. Three aspects of power management are discussed here: on- versus off-chip memory, the use of $\overline{\text{HOLD}}$, and the use of IDLE modes.

To fetch and execute instructions from on-chip memory requires less power than accessing them from off-chip memory. The difference between these two accesses becomes noteworthy if a large piece of code resides off chip and is used more frequently than the on-chip code. The code can be partitioned so that the code that consumes the most power and is used most frequently is placed on-chip. (Masked ROM devices are another alternative for very high-performance applications.)

If the program is executed from internal memory, activities on the external bus during code access cycles can be disabled with the AVIS bit in the PMST register. This feature saves a significant amount of power. However, once the AVIS bit is set, the address bus is still driven in its previous state. The external bus interface bit (EXIO) in the bank-switching control register (BSCR) controls the states of the address, control, and data lines. If the function is disabled, the address and data buses, along with the control lines, become inactive after the current bus cycle.

The $\overline{\text{HOLD}}$ signal and the HM bit of the Status Register 1 (ST1) initiate a power-down mode by either shutting off CPU execution or continuing internal CPU execution if external access is not necessary. This makes external memory available for other processors. The timers and serial ports are not used, and the device can be interrupted and serviced.

Using the IDLE1, IDLE2, and IDLE3 modes can cut down the device power consumption significantly The system clock and peripherals are not halted in IDLE1 mode, but CPU activities are stopped. In IDLE1 mode peripherals and timers can bring the device out of power-down mode. The system can use the timer interrupt as a wake-up if the device needs to be in power-down mode periodically. In IDLE2 mode both CPU and peripherals are halted. The IDLE2 mode saves a significant amount of power, compared to IDLE1. The IDLE3 mode shuts off the entire chip along with the PLL circuitry and save even more power than IDLE2 mode. Unlike the IDLE1 mode, an external interrupt is required to wake up the processor in IDLE2 or IDLE3 mode.

Chapter 3

Arithmetic and Logical Operations

This chapter shows how the TMS320C54x™ (C54x) supports typical arithmetic and logical operations, including multiplication, addition, division, square roots, and extended-precision operations.

Also, the C54x DSP Library (DSPLIB) (see Chapter 5) contains additional math routines.

Topic **Page**

3.1 Division and Modulus Algorithm

The C54x implements division operations by using repeated conditional subtraction. Example 3–1 uses four types of integer division and modulus:

- ❏ Type I: 32-bit by 16-bit unsigned integer division and modulus
- ❏ Type II: 32-bit by 16-bit signed integer division and modulus
- ❏ Type III: 16-bit by 16-bit unsigned integer division and modulus
- ❏ Type IV: 16-bit by 16-bit signed integer division and modulus

SUBC performs binary division like long division. For 16-bit by 16-bit integer division, the dividend is stored in low part accumulator A. The program repeats the SUBC command 16 times to produce a 16-bit quotient in low part accumulator A and a 16-bit remainder in high part accumulator B. For each SUBC subtraction that results in a negative answer, you must left-shift the accumulator by 1 bit. This corresponds to putting a 0 in the quotient when the divisor does not go into the dividend. For each subtraction that produces a positive answer, you must left shift the difference in the ALU output by 1 bit, add 1, and store the result in accumulator A. This corresponds to putting a 1 in the quotient when the divisor goes into the dividend.

Similarly, 32-bit by 16-bit integer division is implemented using two stages of 16-bit by 16-bit integer division. The first stage takes the upper 16 bits of the 32-bit dividend and the 16-bit divisor as inputs. The resulting quotient becomes the higher 16 bits of the final quotient. The remainder is left shifted by 16 bits and adds the lower 16 bits of the original dividend. This sum and the 16-bit divisor become inputs to the second stage. The lower 16 bits of the resulting quotient is the final quotient and the resulting remainder is the final remainder.

Both the dividend and divisor must be positive when using SUBC. The division algorithm computes the quotient as follows:

1) The algorithm determines the sign of the quotient and stores this in accumulator B.
2) The program determines the quotient of the absolute value of the numerator and the denominator, using repeated SUBC commands.
3) The program takes the negative of the result of step 2, if appropriate, according to the value in accumulator B.

For unsigned division and modulus (types I and III), you must disable the sign extension mode (SXM = 0). For signed division and modulus (types II and IV), turn on sign extension mode (SXM = 1). The absolute value of the numerator must be greater than the absolute value of the denominator.

Example 3–1. Unsigned/Signed Integer Division Examples

```
;;=========================================================================== ;;
;; File Name: DIV_ASM.ASM
;;
;; Title: Divide & Modulus – Assembly Math Utilities.
;;
;; Original draft: Alex Tessaralo
;; Modified for C54x: Simon Lau & Philip Jones
;;  Texas Instruments Inc.
;; ;;=========================================================================
;;
;; Target:   C54X
;; ;;=========================================================================
;;
;; Contents:  DivModUI32             ; 32–bit By 16–bit Unsigned Integer Divide
;;                                      ; And Modulus.
;;               DivModUI16             ; 16–bit By 16–bit Unsigned Integer Divide
;;                                      ; And Modulus.
;;               DivModI32           ; 32–bit By 16–bit Signed Integer Divide
;;                                      ; And Modulus.
;;               DivModI16           ; 16–bit By 16–bit Signed Integer Divide
;;                                      ; And Modulus.
;; ;;=========================================================================
;;
;; History:   mm/dd/yy | Who      | Description Of Changes.
;; ---------+----------+---------------------------------------
;; 08/01/96 | Simon L. | Original draft.
;;
;;=========================================================================
;;=========================================================================== ;;
;; Module Name: DivModUI32
;; ;;=========================================================================
;;
;; Description: 32 Bit By 16 Bit Unsigned Integer Divide And Modulus
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;  .bss      d_NumH,1                ; 00000000h to FFFFFFFFh
;;  .bss      d_NumL,1
;;  .bss      d_Den,1                 ; 0000h to FFFFh
;;  .bss      d_QuotH,1               ; 00000000h to FFFFFFFFh
;;  .bss      d_QuotL,1
;;  .bss      d_Rem,1                 ; 0000h to FFFFh
;;
;;  CALL      DivModUI32
;; ;;-------------------------------------------------------------------------;;
;; Input: d_NumH
;;           d_NumL
;;           d_Den
;;
;; Modifies:  SXM
;;  accumulator A
;;
;; Output:   d_QuotH
;;           d_QuotL
```

Example 3–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;           d_Rem
;;
;;-------------------------------------------------------------------------;;
;; Algorithm: Quot         = Num/Den
;;     Rem                 = Num%Den
;;;;   NumH                   = n3|n2      QuotH  = q3|q2
;;     NumL                   = n1|n0      QuotL  = q1|q0
;;     Den                 = d1|d0      Rem = r1|r0
;;
;;     Phase1: t1|t0|q3|q2 = A     (after repeating SUBC 16 times)
;;              ___________
;;     d1|d0 ) 00|00|n3|n2 = A     (before)
;;
;;
;;     Phase2: r1|r0|q1|q0    = A     (after repeating SUBC 16 times)
;;              ___________
;;     d1|d0 )   t1|t0|n1|n0      = A     (before)
;;
;;     NOTES:    Sign extension mode must be turned off.
;; ;;=========================================================================
    .def          DivModUI32
    .ref          d_NumH
    .ref          d_NumL
    .ref          d_Den
    .ref          d_QuotH
    .ref          d_QuotL
    .ref          d_Rem
    .textDivModUI32:
    RSBX          SXM                ; sign extention mode off
    LD            d_NumH,A
    RPT#(16-1)
    SUBC          d_Den,A
    STLA,     d_QuotH
    XOR       d_QuotH,A          ; clear AL
    OR            d_NumL,A               ; AL = NumL
    RPT#(16-1)
    SUBC          d_Den,A
    STLA,     d_QuotL
    STHA,d_Rem
    RET
;;========================================================================= ;;
;; Module Name: DivModUI16
;; ;;=========================================================================
;;
;; Description: 16 Bit By 16 Bit Unsigned Integer Divide And Modulus
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;     .bss      d_Num,1               ; 0000h to FFFFh
;;     .bss      d_Den,1               ; 0000h to FFFFh
;;     .bss      d_Quot,1              ; 0000h to FFFFh
;;     .bss      d_Rem,1               ; 0000h to FFFFh
;;
;;     CALL      DivModUI16
```

Example 3–1. Unsigned/Signed Integer Division Examples (Continued)

```
;; ;;-------------------------------------------------------------------------;;
;; Input:    d_Num
;;              d_Den
;;
;; Modifies:  SXM
;;              accumulator A
;;
;; Output:      d_Quot
;;              d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm: Quot   = Num/Den
;;     Rem= Num%Den
;;
;;     Num= n1|n0             Quot    = q1|q0
;;     Den= d1|d0             Rem     = r1|r0
;;
;;        r1|r0|q1|q0 = A     (after repeating SUBC 16 times)
;;
;;     d1|d0 ) 00|00|n1|n0 = A     (before)
;;
;;     NOTES:  Sign extension mode must be turned off.
;; ;;=========================================================================
       .def      DivModUI16
       .ref      d_Num
       .ref      d_Den
       .ref      d_Quot
       .ref      d_Rem
       .text
DivModUI16:
       RSBX      SXM                      ; sign extention mode off
       LD        @d_Num,A
       RPT       #(16-1)
       SUBC      @d_Den,A
       STL    A,@d_Quot
       STH    A,@d_Rem
       RET
;;=========================================================================== ;;
;; Module Name: DivModI32
;; ;;=========================================================================
;;
;; Description: 32 Bit By 16 Bit Signed Integer Divide And Modulus.
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;     .bss      d_NumH,1                    ; 80000001h to 7FFFFFFFh
;;     .bss      d_NumL,1
;;     .bss      d_Den,1                     ;     8000h to 7FFFh
;;     .bss      d_QuotH,1                 ; 80000001h to 7FFFFFFFh
;;     .bss      d_QuotL,1
;;     .bss      d_Rem,1                     ;     8000h to 7FFFh
;;
;;     CALL   DivModI32
; ;;-------------------------------------------------------------------------;;
;; Input:    d_NumH
```

Example 3–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;              d_NumL
;;              d_Den
;;
;; Modifies:  SXM
;;              T
;;              accumulator A
;;              accumulator B
;;
;; Output:      d_QuotH
;;              d_QuotL
;;              d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm: Quot   = Num/Den
;;     Rem          = Num%Den
;;;;      Signed division is similar to unsigned division except that
;;     the sign of Num and Den must be taken into account.
;;     First the sign is determined by multiplying Num by Den.
;;     Then division is performed on the absolute values.
;;
;;     NumH   = n3|n2         QuotH   = q3|q2
;;     NumL   = n1|n0         QuotL   = q1|q0
;;     Den = d1|d0         Rem     = r1|r0
;;
;;     Phase1: t1|t0|q3|q2 = A       (after repeating SUBC 16 times)
;;              ___________
;;     d1|d0 ) 00|00|n3|n2 = A       (before)
;;
;;;;   Phase2: r1|r0|q1|q0 = A       (after repeating SUBC 16 times)
;;              _____________
;;     d1|d0 ) t1|t0|n1|n0 = A       (before)
;;
;;     NOTES:  Sign extension must be turned on.
;; ;;=========================================================================
       .def      DivModI32
       .ref      d_NumH
       .ref      d_NumL
       .ref      d_Den
       .ref      d_QuotH
       .ref      d_QuotL
       .ref      d_Rem
       .text
DivModI32:
       SSBX      SXM                        ; sign extention mode on
       LD        d_Den,16,A
       MPYA      d_NumH                     ; B has sign of quotient
       ABS    A
       STH A  ,d_Rem                    ; d_Rem = abs(Den) temporarily
       LD        d_NumH,16,A
       ADDS      d_NumL,A
       ABS    A
       STH    A,d_QuotH                  ; d_QuotH = abs(NumH) temporarily
       STL    A,d_QuotL                  ; d_QuotL = abs(NumL) temporarily
       LD        d_QuotH,A
```

Example 3–1. Unsigned/Signed Integer Division Examples (Continued)

```
       RPT    #(16–1)
       SUBC       d_Rem,A
       STL    A,d_QuotH                  ; AH = abs(QuotH)
       XOR    d_QuotH,A                  ; clear AL
       OR         d_QuotL,A                  ; AL = abs(NumL)
       RPT    #(16–1)
       SUBC       d_Rem,A
       STL    A,d_QuotL                  ; AL = abs(QuotL)
       STH    A,d_Rem                        ; AH = Rem
       BCD    DivModI32Skip,BGEQ         ; if B neg, then Quot =
                                                 ; –abs(Quot)
       LD         d_QuotH,16,A
       ADDS       d_QuotL,A
       NEG    A
       STH    A,d_QuotH
       STL    A,d_QuotL
DivModI32Skip:
       RET
;;=========================================================================== ;;
;; Module Name: DivModI16
;; ;;=========================================================================
;;
;; Description: 16 Bit By 16 Bit Signed Integer Divide And Modulus.
;; ;;-------------------------------------------------------------------------;;
;; Usage ASM:
;;        .bss      d_Num,1              ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Den,1              ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Quot,1             ; 8000h to 7FFFh (Q0.15 format)
;;        .bss      d_Rem,1              ; 8000h to 7FFFh (Q0.15 format)
;;
;;        CALL      DivModI16
;; ;;-------------------------------------------------------------------------;;
;; Input:        d_Num
;;                  d_Den
;;
;; Modifies:     AR2
;;                  T
;;                  accumulator A
;;                  accumulator B
;;                  SXM
;;
;; Output:          d_Quot
;;                  d_Rem
;; ;;-------------------------------------------------------------------------;;
;; Algorithm:    Quot      = Num/Den
;;                  Rem    = Num%Den
;;
;;     Signed division is similar to unsigned division except that
;;     the sign of Num and Den must be taken into account.
;;     First the sign is determined by multiplying Num by Den.
;;     Then division is performed on the absolute values.
;;
;;                  Num    = n1|n0        Quot    = q1|q0
```

Example 3–1. Unsigned/Signed Integer Division Examples (Continued)

```
;;                  Den    = d1|d0        Rem     = r1|r0
;;
;;            r1|r0|q1|q0     = A       (after repeating SUBC 16 times)
;;             ___________
;;        d1|d0 )   00|00|n1|n0  = A     (before)
;;
;;     NOTES:  Sign extension mode must be turned on.
;;
;;=============================================================================
          .def      DivModI16
          .ref      d_Num
          .ref      d_Den
          .ref      d_Quot
          .ref      d_Rem
          .text
DivModI16:
          SSBX      SXM                 ; sign extention mode on
          STM    #d_Quot,AR2
          LD        d_Den,16,A
          MPYA      d_Num                  ; B has sign of quotient
          ABS    A
          STH    A,d_Rem                ; d_Rem = abs(Den) temporarily
          LD        d_Num,A
          ABS    A                      ; AL = abs(Num)
          RPT    #(16–1)          SUBC      d_Rem,A
          STL    A,d_Quot               ; AL = abs(Quot)
          STH    A,d_Rem                ; AH = Rem
          LD        #0,A
          SUB       d_Quot,16,A         ; AH = –abs(Quot)
          SACCD     A,*AR2,BLT             ; If B neg, Quot = –abs(Quot)
          RET
;;============================================================================= ;;
;;End Of File.
;;=============================================================================
```

3.2 Sines and Cosines

Sine-wave generators are used in signal processing systems, such as communications, instrumentation, and control. In general, there are two methods to generate sine and cosine waves. The first is the table look-up method, which is used for applications not requiring extreme accuracy. This method uses large tables for precision and accuracy and requires more memory. The second method is the Taylor series expansion, which is more efficient. This method determines the sine and cosine of an angle more accurately and uses less memory than table look-up, and it is discussed here.

The first four terms of the expansion compute the angle. The Taylor series expansions for the sine and cosine of an angle are:

$$
\begin{aligned}
sin(\theta) &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \\
&= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}\left(1 - \frac{x^2}{8.9}\right) \\
&= x - \frac{x^2}{3!} + \frac{x^5}{5!}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right) \\
&= x - \frac{x^2}{3!}\left(1 - \frac{x^2}{4.5}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right)\right) \\
&= x\left(1 - \frac{x^2}{2.3}\left(1 - \frac{x^2}{4.5}\left(1 - \frac{x^2}{6.7}\left(1 - \frac{x^2}{8.9}\right)\right)\right)\right)
\end{aligned}
$$

$$
\begin{aligned}
cos(\theta) &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \\
&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}\left(1 - \frac{x^2}{7.8}\right) \\
&= 1 - \frac{x^2}{2!} + \frac{x^4}{4!}\left(1 - \frac{x^2}{5.6}\left(1 - \frac{x^2}{7.8}\right)\right) \\
&= 1 - \frac{x^2}{2}\left(1 - \frac{x^2}{3.4}\left(1 - \frac{x^2}{5.6}\left(1 - \frac{x^2}{7.8}\right)\right)\right)
\end{aligned}
$$

The following recursive formulas generate the sine and cosine waves:

$$sin\ n\theta = 2\ cos(\theta)sin\{(n-1)\theta\} - sin\{(n-2)\theta\}$$

$$cos\ n\theta = 2\ cos(\theta)cos\{(n-1)\theta\} - cos\{(n-2)\theta\}$$

These equations use two steps to generate a sine or cosine wave. The first evaluates cos(θ) and the second generates the signal itself, using one multiply and one subtract for a repeat counter, n.

Example 3–2 and Example 3–3 assume that the delayed cos((n–1)) and cos((n–2)) are precalculated and are stored in memory. The Taylor series expansion to evaluate the delayed cos((n–1)), cos((n–2))/sin((n–1)), and sin((n–2)) values for a given θ can also be used.

Example 3–2. Generation of a Sine Wave

```
; Functional Description
;  This function evaluates the sine of an angle using the Taylor series
;  expansion.
;  sin(theta) = x(1–x^2/2*3(1–x^2/4*5(1–x^2/6*7(1–x^2/8*9))))
;
          .mmregs
          .def         d_x,d_squr_x,d_coff,d_sinx,C_1
d_coff .sect         ”coeff”
          .word        01c7h
          .word        030bh
          .word        0666h
          .word        1556h
d_x    .usect    ”sin_vars”,1
d_squr_x  .usect     ”sin_vars”,1
d_temp .usect    ”sin_vars”,1
d_sinx .usect    ”sin_vars”,1
C_1    .usect    ”sin_vars”,1
          .text
sin_start:
          STM    #d_coff,AR3         ; c1=1/72,c2=1/42,c3=1/20,
                                       ; c4=1/6
          STM    #d_x,AR2            ; input value
          STM    #C_1,AR4            ; A1, A2, A3, A4
          sin_angle:
          LD         #d_x,DP
          ST         #6487h,d_x          ; pi/4
          ST         #7fffh,C_1
          SQUR       *AR2+,A             ; let x^2 = P
          ST         A,*AR2              ; AR2  – > x^2
          || LD      *AR4,B              ;
          MASR       *AR2+,*AR3+,B,A  ; (1–x^2)/72
          MPYA       A                   ; 1–x^2(1–x^2)/72
                                         ; T = x^2
          STH    A,*AR2
          MASR       *AR2–,*AR3+,B,      ; A = 1–x^2/42(1–x^2/72)
                                         ; T =x^2(1–x^2/72)
          MPYA       *AR2+               ; B = A(32–16)*x^2
          ST         B,*AR2              ;
          || LD      *AR4,B              ; B = C_1
          MASR       *AR2–,*AR3+,B,      ; A = 1–x^2/20(1–x^2/42(1–x^2/72)
          MPYA       *AR2+               ; B = A(32–16)*x^2
          ST         B,*AR2
          || LD      *AR4,B
          MASR       *AR2–,*AR3+,B,A  ; AR2 – > d_squr_x
          MPYA       d_x
          STH    B, d_sinx        ; sin(theta)
          RET
```

Example 3–2.Generation of a Sine Wave (Continued)

```
          .end
; Functional Description
; This function generates the sine of angle. Using the recursive given above, the
; cosine of the angle is found and the recursive formula is used to generate the
; sine wave. The sin(n–1) and sin(n–2) can be calculated using the Taylor
; series expansion or can be pre–calculated.
                .mmregs
                .ref          cos_prog,cos_start
d_sin_delay1  .usect    ”cos_vars”,1
d_sin_delay2  .usect    ”cos_vars”,1
K_sin_delay_1.set          0A57Eh ; sin(–pi/4)
K_sin_delay_2.set          8000h     ; sin(–2*pi/4);
K_2           .set          2h        ; cicular buffer size
K_256            .set       256       ; counter
K_THETA          .set       6487h        ; pi/4
                .text
start:
       LD     #d_sin_delay1,DP
       CALL   cos_start
       STM #d_sin_delay1,AR3          ; intialize the buffer
       RPTZ       A,#3h
       STL A,*AR3+
       STM #1,AR0
       STM #K_2,BK
       STM #K_256–1,BRC
       STM #d_sin_delay1,AR3
       ST     #K_sin_delay_1,*AR3+%  ; load calculated initial values of sin((n–1) )
       ST     #K_sin_delay_2,*AR3+%  ; load calculated initial values of sin((n–2) )
                                        ; this generates the sine_wave
sin_generate:
       RPTB   end_of_sine
       MPY *AR2,*AR3+0%,A             ; cos(theta)*sin{(n–1)theta}
       SUB *AR3,15,A               ; 1/2*sin{(n–2)theta)
       SFTA   A,1,A                      ; sin(n*theta)
       STH A,*AR3                  ; store
end_of_sine
       NOP
       NOP
       B      sin_generate
       .end
```

Example 3–3. Generation of a Cosine Wave

```
; Functional Description
; this computes the cosine of an angle using the Taylor Series Expansion
       .mmregs
       .def   d_x,d_squr_x,d_coff,d_cosx,C_7FFF
       .def   cos_prog,cos_start
       STH A,*AR3                    ; store
       .word  024ah                        ; 1/7.8
       .word  0444h                        ; 1/5.6
       .word  0aa9h                        ; 1/3.4
d_x .usect ”cos_vars”,1
d_squr_x      .usect ”cos_vars”,1
d_cosx    .usect ”cos_vars”,1
C_7FFF    .usect ”cos_vars”,1
K_THETA       .set      6487h           ; pi/4
K_7FFF    .set      7FFFh
             .text
cos_start:
             STM    #d_coff,AR3         ;c1=1/56,c2=1/30,c3=1/12
             STM    #d_x,AR2            ; input theta
             STM    #C_7FFF,AR4         ; A1, A2, A3, A4
cos_prog:
             LD        #d_x,DP
             ST        #K_THETA,d_x     ; input theta
             ST        #K_7FFF,C_7FFF
             SQUR      *AR2+,A             ; let x^2 = P
             ST        A,*AR2           ; AR2  – > x^2
             || LD     *AR4,B           ;
             MASR      *AR2+,*AR3+,B,A  ; (1–x^2)/72
             MPYA      A                ; 1–x^2(1–x^2)/72
                                        ; T = x^2
             STH       A,*AR2
             MASR      *AR2–,*AR3+,B,A  ; A = 1–x^2/42(1–x^2/72)
                                        ; T =x^2(1–x^2/72)
             MPYA      *AR2+            ; B = A(32–16)*x^2
             ST        B,*AR2           ;
             || LD     *AR4,B           ; B = C_1
             MASR      *AR2–,*AR3+,B,A  ; A = 1–x^2/20(1–x^2/42(1–x^2/72))
             SFTA      A,–1,A           ; –1/2
             NEG       A
             MPYA      *AR2+            ; B = A(32–16)*x^2
             RETD
             ADD       *AR4,16,B
             STH       B,*AR2           ; cos(theta)
             .end
             .mmregs
             .ref      cos_prog,cos_start
d_cos_delay1  .usect ”cos_vars”,1
d_cos_delay2  .usect cos_vars”,1
d_theta          .usect ”cos_vars”,1
K_cos_delay_1 .set      06ed9h              ; cos(–pi/6)
K_cos_delay_2 .set      4000h                  ; cos(–2*pi/6);
K_2           .set      2h                     ; cicular buffer size
K_256            .set      256                 ; counter
```

Example 3–3.Generation of a Cosine Wave (Continued)

```
K_theta          .set      4303h                  ; sin(pi/2–pi/6)= cos(pi/6)
                                                  ; cos(pi/2–pi/x)
                                                  ; .052= 4303h
                 .text
start:
       LD        #d_cos_delay1,DP
       CALL      cos_start
       CALL      cos_prog                         ; calculate cos(theta)
       STM    #d_cos_delay1,AR3
       RPTZ      A,#3h
       STL    A,*AR3+
       STM    #d_cos_delay1,AR3
       ST        #K_cos_delay_1,*AR3+
       ST        #K_cos_delay_2,*AR3
       STM    #d_cos_delay1,AR3               ; output vaues
       ST        #K_theta,d_theta
       STM    #1,AR0
       STM    #K_2,BK
       STM    #K_256–1,BRC
cos_generate:
       RPTB      end_of_cose
       MPY    *AR2,*AR3+0%,A                   ; cos(theta)*cos{(n–1)theta}
       SUB    *AR3,15,A                     ; 1/2*cos{(n–2)theta)
       SFTA      A,1,A                            ; cos(n*theta)
       STH    A,*AR3                        ; store
       PORTW     *AR3,56h                         ; write to a port
end_of_cose
       NOP
       NOP
       B         cos_generate                  ; next sample
       .end
```

3.3 Square Roots

Example 3–4 uses a 6-term Taylor series expansion to approximate the square root of a single-precision 32-bit number. A normalized, 32-bit, left-justified number is passed to the square root function. The output is stored in the upper half of the accumulator, and the EXP and NORM instructions normalize the input value. The EXP instruction computes an exponent value in a single cycle and stores the result in T, allowing the NORM instruction to normalize the number in a single cycle. If the exponent is an odd power, the mantissa is (multiplied by 1 divided by the square root of 2) to compensate after finding the square root of the 32-bit number. The exponent value is negated to denormalize the number.

$$y^{0.5} = (1 + x)^{0.5}$$

where :

$$x = y–1$$

$$= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \frac{7x^5}{256}$$

$$= 1 + \frac{x}{2} - 0.5\left(\frac{x}{2}\right)^2 + 0.5\left(\frac{x}{2}\right)^3 - 0.625\left(\frac{x}{2}\right)^4 + 0.875\left(\frac{x}{2}\right)^5$$

where :

$$0.5 \leqq x < 1$$

Example 3–4. Square Root Computation

```
*************************************************************************
* Six term Taylor Series is used here to compute the square root of a number
* y^0.5 = (1+x)^0.5 where x = y–1
*   = 1+(x/2)–0.5*((x/2)^2+0.5*((x/2)^3–0.625*((x/2)^4+0.875*((x/2)^5)
* 0.5 <= x < 1
*************************************************************************
       .mmregs
       .sect        ”squr_var”
d_part_prod         .word  0
d_part_shift     .word  0
C_8000           .word  0
C_sqrt_one_half  .word  0
d_625               .word  0
d_875               .word  0
tmp_rg1             .word  0
K_input             .set   800h            ; input # = 0.0625
K_8000           .set   8000h           ; –1 or round off bit
K_4000           .set   4000h           ; 0.5 coeff
K_SQRT_HALF         .set   5a82h             ; 1/sqrt2
K_625               .set   –20480       ; coeff 0.625
K_875               .set   28672             ; coeff 0.875
                    .text
```

Example 3–4. Square Root Computation (Continued)

```
sqroot:
              LD     #d_part_prod,DP
              ST     #K_8000,C_8000
              ST     #K_input,d_part_prod
              ST     #K_SQRT_HALF,C_sqrt_one_half
              ST     #K_875,d_875
              ST     #K_625,d_625
              LD     d_part_prod,16,A    ; load the #
              EXP A
              nop                        ; dead cycle
              NORM   A
              ADDS   C_8000,A               ; round off bit
              STH A, d_part_prod      ; normalized input
              LDM T,B
              SFTA   B,–1,B              ; check for odd or even power
              BCD res_even,NC
              NE     B                      ; negate the power
              STL B,d_part_shift          ; this shift is used to denormalize the #
              LD     d_part_prod,16,B    ; load the normalized input #
              CALLD  sq_root                ; square root program
              ABS B
              NOP                        ; cycle for delayed slot
              LD     B,A                 ;
              BD     res_common
              SUB B,B                 ; zero B
              MACAR  C_sqrt_one_half,B      ; square root of 1/2
                                            ; odd power
res_even
              LD     d_part_prod,16,B
              CALLD  sq_root
              ABS B
              NOP                        ; cycle for the delayed slot
res_common
              LD     d_part_shift,T         ; right shift value
              RETD
              STH B,d_part_prod
              LD     d_part_prod,TS,A    ; denormaliize the #
sq_root:
              SFTA   B,–1,B              ; x/2 = y–1/2
              SUB #K_4000,16,B,B
              STH B,tmp_rg1           ; tmp_rg1 = x/2
              SUB #K_8000,16,B        ; B = 1+x/2
              SQUR   tmp_rg1,A           ; A (x/2)^2, T = x/2
              NEG A                      ; A = –A
              ADD A,–1,B              ; B = 1+x/2–.5(x/2)^2
              SQUR   A,A                 ; A = (x/2)^4
              MACA   d_625,B                ; 0.625*A+B
                                            ; T =0.625
              LD     tmp_rg1,T           ; T = x/2
              MPYA   A                      ; (x/2)^4*x/2
              MACA   d_875,B                ; 0.875*A+B
              SQUR   tmp_rg1,A           ; x/2^2; T = x/2
              MPYA   A                      ; A = x/2*x/2^2
              RETD
              ADD A,–1,B
              ADDS   C_8000,B               ; round off bit
              .end
```

3.4 Extended-Precision Arithmetic

Numerical analysis, floating-point computations, and other operations may require arithmetic operations with more than 32 bits of precision. Since the C54x devices are 16/32-bit fixed-point processors, software is required for arithmetic operations with extended precision. These arithmetic functions are performed in parts, similar to the way in which longhand arithmetic is done.

The C54x has several features that help make extended-precision calculations more efficient. One of the features is the carry bit, which is affected by most arithmetic ALU instructions, as well as the rotate and shift operations. The carry bit can also be explicitly modified by loading ST0 and by instructions that set or reset status register bits. For proper operation, the overflow mode bit should be reset (OVM = 0) to prevent the accumulator from being loaded with a saturation value.

The two C54x internal data buses, CB and DB, allow some instructions to handle 32-bit operands in a single cycle. The long-word load and double-precision add/subtract instructions use 32-bit operands and can efficiently implement multi-precision arithmetic operations.

The hardware multiplier can multiply signed/unsigned numbers, as well as multiply two signed numbers and two unsigned numbers. This makes 32-bit multiplication efficient.

3.4.1 Addition and Subtraction

The carry bit, C, is set in ST0 if a carry is generated when an accumulator value is added to:

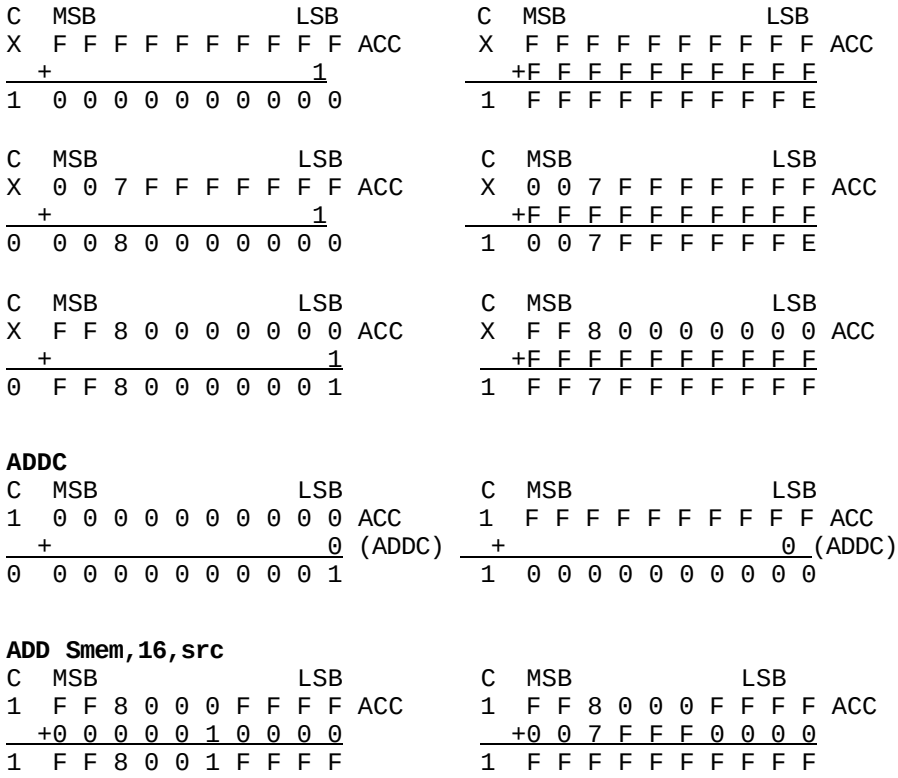
- The other accumulator
- A data-memory operand
- An immediate operand

A carry can also be generated when two data-memory operands are added or when a data-memory operand is added to an immediate operand. If a carry is not generated, the carry bit is cleared.

The ADD instruction with a 16-bit shift is an exception because it only sets the carry bit. This allows the ALU to generate the appropriate carry when adding to the lower or upper half of the accumulator causes a carry.

Figure 3–1 shows several 32-bit additions and their effect on the carry bit.

Figure 3–1. 32-Bit Addition



Example 3–5 adds two 64-bit numbers to obtain a 64-bit result. The partial sum of the 64-bit addition is efficiently performed by the DLD and DADD instructions, which handle 32-bit operands in a single cycle. For the upper half of a partial sum, the ADDC (ADD with carry) instruction uses the carry bit generated in the lower 32-bit partial sum. Each partial sum is stored in two memory locations by the DST (long-word store) instruction.

Example 3–5. Lit Number

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; 64-bit Addition
;
;       X3 X2 X1 X0
;     + Y3 Y2 Y1 Y0
;     -------------
;       W3 W2 W1 W0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ADD64: DLD   @X1,A     ;A = X1 X0
       DADD  @Y1,A     ;A = X1 X0 + Y1 Y0
       DST   A,@W1
       DLD   @X3,A     ;A = X3 X2
       ADDC  @Y2,A     ;A = X3 X2 + 00 Y2 + C
       ADD   @Y3,16,A  ;A = X3 X2 + Y3 Y2 + C
       DST   A,@W3
       RET
```

Similar to addition, the carry bit is reset if a borrow is generated when an accumulator value is subtracted from:

- ❏ The other accumulator
- ❏ A data-memory operand
- ❏ An immediate operand

A borrow can also be generated when two data-memory operands are subtracted or when an immediate operand is subtracted from a data-memory operand. If a borrow is not generated, the carry bit is set.

The SUB instruction with a 16-bit shift is an exception because it only resets the carry bit. This allows the ALU to generate the appropriate carry when subtracting from the lower or the upper half of the accumulator causes a borrow.

Figure 3–2 shows several 32-bit subtractions and their effect on the carry bit.

Figure 3–2. 32-Bit Subtraction

```
C  MSB             LSB          C  MSB             LSB
X  0 0 0 0 0 0 0 0 0 0 ACC      X  F F 0 0 0 0 0 0 0 0 ACC
  –                   1           –F F F F F F F F F F
0  F F F F F F F F F F          0  0 0 0 0 0 0 0 0 0 1

C  MSB             LSB          C  MSB             LSB
X  0 0 7 F F F F F F F ACC      X  0 0 7 F F F F F F F ACC
  –                   1           –F F F F F F F F F F
1  0 0 7 F F F F F F E          C  F F 8 0 0 0 0 0 0 0

C  MSB             LSB          C  MSB             LSB
X  F F 8 0 0 0 0 0 0 0 ACC      X  F F 8 0 0 0 0 0 0 0 ACC
  –                   1           –F F F F F F F F F F
1  F F 7 F F F F F F F          0  F F 8 0 0 0 0 0 0 1
```

SUBB

```
C  MSB             LSB          C  MSB             LSB
0  0 0 0 0 0 0 0 0 0 0 ACC      0  F F F F F F F F F F ACC
  –                   0 (SUBB)    –                   0(SUBB)
0  F F F F F F F F F F          1  F F F F F F F F F E
```

SUB Smem,16,src

```
C  MSB             LSB          C  MSB             LSB
1  F F 8 0 0 0 F F F F ACC      0  F F 8 0 0 0 F F F F ACC
  –0 0 0 0 0 1 0 0 0 0            –F F F F F F 0 0 0 0
0  0 0 7 F F F F F F F          0  F F 8 0 0 1 F F F F
```

Example 3–6 subtracts two 64-bit numbers on the C54x. The partial remainder of the 64-bit subtraction is efficiently performed by the DLD (long word load) and the DSUB (double precision subtract) instructions, which handle 32-bit operands in a single cycle. For the upper half of a partial remainder, the SUBB (SUB with borrow) instruction uses the borrow bit generated in the lower 32-bit partial remainder. Each partial remainder is stored in two consecutive memory locations by a DST.

Example 3–6. 64-Bit Subtraction

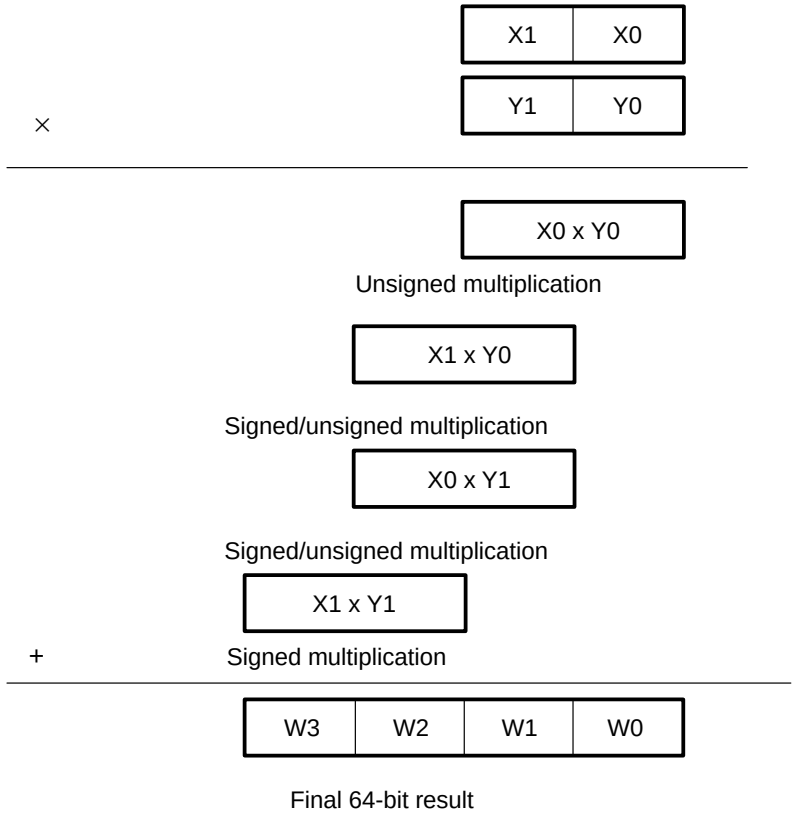
```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; 64 bit Subtraction
;
;       X3 X2 X1 X0
;     – Y3 Y2 Y1 Y0
;     –––––––––––––
;       W3 W2 W1 W0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
      DLD    @X3,A     ;A = X3 X2
      SUBB   @Y2,A     ;A = X3 X2 – 00 Y2 – (inv C)
      DST    A,@W1
      SUB    @Y3,16,A  ;A = X3 X2 – Y3 Y2 – (inv C)
      DST    A,@W3
      RET
```

3.4.2 Multiplication

The MPYU (unsigned multiply) and MACSU (signed/unsigned multiply and accumulate) instructions can also handle extended-precision calculations.

Figure 3–3 shows how two 32-bit numbers obtain a 64-bit product. The MPYU instruction multiplies two unsigned 16-bit numbers and places the 32-bit result in one of the accumulators in a single cycle. The MACSU instruction multiplies a signed 16-bit number by an unsigned 16-bit number and accumulates the result in a single cycle. Efficiency is gained by generating partial products of the 16-bit portions of a 32-bit (or larger) value instead of having to split the value into 15-bit (or smaller) parts.

Figure 3–3. 32-Bit Multiplication



The program in Example 3–7 shows that a multiply of two 32-bit integer numbers requires one multiply, three multiply/accumulates, and two shifts. The product is a 64-bit integer number. Note in particular, the use of MACSU, MPYU and LD instructions. The LD instruction can perform a right-shift in the accumulator by 16 bits in a single cycle.

Example 3–7. 32-Bit Integer Multiplication

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two 32-bit signed integers
; resulting; in a 64-bit product. The operands are fetched

; from data memory and the result is written back to data
; memory.
; Data Storage:
;    X1,X0           32-bit operand
;    Y1,Y0           32-bit operand
;    W3,W2,W1,W0     64-bit product
; Entry Conditions:
;    SXM = 1, OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   STM   #X0,AR2          ;AR2 = X0 addr
   STM   #Y0,AR3          ;AR3 = Y0 addr
   LD    *AR2,T           ;T = X0
   MPYU  *AR3+,A          ;A = X0*Y0
   STL   A,@W0            ;save W0
   LD    A,–16,A          ;A = A >> 16
   MACSU *AR2+,*AR3–,A    ;A = X0*Y0>>16 + X0*Y1
   MACSU *AR3+,*AR2,A     ;A = X0*Y0>>16 + X0*Y1 + X1*Y0
   STL   A,@W1            ;save W1
   LD    A,–16,A          ;A = A >> 16
   MAC   *AR2,*AR3,A      ;A = (X0*Y1 + X1*Y0)>>16 + X1*Y1
   STL   A,@W2            ;save W2
   STH   A,@W3            ;save W3
```

Example 3–8 performs fractional multiplication. The operands are in Q31 format, while the product is in Q30 format.

Example 3–8. 32-Bit Fractional Multiplication

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; This routine multiplies two Q31 signed integers
; resulting in a Q30 product. The operands are fetched
; from data memory and the result is written back to data
; memory.
; Data Storage:
;    X1,X0           Q31 operand
;    Y1,Y0           Q31 operand
;    W1,W0           Q30 product
; Entry Conditions:
;    SXM = 1, OVM = 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   STM   #X0,AR2         ;AR2 = X0 addr
   STM   #Y1,AR3         ;AR3 = Y1 addr
   LD    #0,A            ;clear A
   MACSU *AR2+,*AR3–,A   ;A = X0*Y1
   MACSU *AR3+,*AR2,A    ;A = X0*Y1 + X1*Y0
   LD    A,–16,A         ;A = A >> 16
   MAC   *AR2,*AR3,A     ;A = A + X1*Y1
   STL   A,@W0           ;save lower product
   STH   A,@W1           ;save upper product
```

3.5 Floating-Point Arithmetic

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if a 32-bit number places the binary point after the most significant bit (which is also the sign bit), only fractional numbers (numbers with absolute values less than 1), can be represented. The fixed-point system, although simple to implement in hardware, imposes limitations in the dynamic range of the represented number. You can avoid this difficulty by using floating-point numbers.

A floating-point number consists of a mantissa, m, multiplied by a base, b, raised to an exponent, e, as follows:

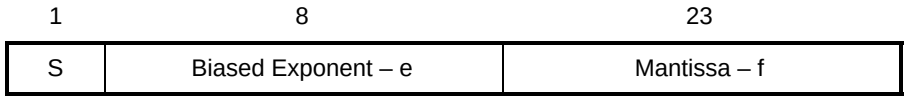
$$m * b^e$$

To implement floating-point arithmetic on the C54x, operands must be converted to fixed-point numbers and then back to floating-point numbers. Fixed-point values are converted to floating-point values by normalizing the input data.

Floating-point numbers are generally represented by mantissa and exponent values. To multiply two numbers, add their mantissas, multiply the exponents, and normalize the resulting mantissa. For floating-point addition, shift the mantissa so that the exponents of the two operands match. Left-shift the lower-power operand by the difference between the two exponents. Add the exponents and normalize the result.

Figure 3–4 illustrates the IEEE standard format to represent floating-point numbers. This format uses sign-magnitude notation for the mantissa, and the exponent is biased by 127. In a 32-bit word representing a floating-point number, the first bit is the sign bit, represented by s. The next eight bits correspond to the exponent, which is expressed in an offset-by-127 format (the actual exponent is e–127). The following 23 bits represent the absolute value of the mantissa, with the most significant 1 implied. The binary point is placed after this most significant 1. The mantissa, then, has 24 bits.

Figure 3–4. IEEE Floating-Point Format



The values of the numbers represented in the IEEE floating-point format are as follows:

$$(-1)^s * 2^{e-127} * (01.f) \qquad \text{If } 0 < e < 255$$

Special Cases:

$(-1)^S * 0.0$	If e = 0, and f = 0 (zero)
$(-1)^S * 2-126 * (0.f)$	If e = 0 and f <> 0 (denormalized)
$(-1)^S *$ infinity	If e = 255 and f = 0 (infinity)
NaN (not a number)	If e = 255 and f <> 0

Example 3–9 through Example 3–11 illustrate how the C54x performs floating-point addition, multiplication, and division.

Example 3–9. Add Two Floating-Point Numbers

```
*;*************************************************************************
 ;
*; FLOAT_ADD – add two floating point numbers
*; Copyright (c) 1993–1994 Texas Instruments Incorporated
*; NOTE: The ordering of the locals are placed to take advantage of long word
*; loads and stores which require the hi and low words to be at certain addresses.
*; Any future modifications which involve the stack must take this quirk into
*; account
*;*************************************************************************
 ;
*************************************************************************
;Operand 1 (OP1) and Operand (OP2) are each packed into sign, exponent, and the
;words of mantissa. If either exponent is zero special case processing is initiated.
;In the general case, the exponents are compared and the mantissa of the lower
;exponent is renormalized according to the number with the larger exponent. The
;mantissas are also converted to a two’s complement format to perform the actual
;addition. The result of the addition is then renormalized with the corresponding
;adjustment in the exponent. The resulting mantissa is converted back to its
;original sign–magnitude format and the result is repacked into the floating point
;representation.
*************************************************************************
*;*********************************************************************
 ;
*;       resource utilization:  B accumulator, T–register
*;       status bits affected: TC, C, SXM, OVM,
*;       entry requirements : CPL bit set
*;*********************************************************************
 ;
 ; Floating Point Format – Single Precision
*-------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*-------------------------------------------------------------------------------
*-------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*-------------------------------------------------------------------------------
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
*; Single precision floating point format is a 32 bit format consisting of a 1 bit
sign field, an 8 bit exponent field, and a 23 bit mantissa field. The fields are
defined as follows
*;  Sign <S>          : 0 = positive values; 1 = negative value
*;  Exponent <E7–E0>  : offset binary format
*;              00 = special cases (i.e. zero)
*;                 01 = exponent value + 127 = –126
*;                 FE = exponent value + 127 = +127
*;                 FF = special cases (not implemented)
*;  Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                           1.M22M21...M1M0
*;            Range          : –1.9999998 e+127 to –1.0000000 e–126
*;                             +1.0000000 e–126 to +1.9999998 e+127
*;                             (where e represents 2 to the power of)
*;                             –3.4028236 e+38  to –1.1754944 e–38
*;                             +1.1754944 e–38  to +3.4028236 e+38
*;                             (where e represents 10 to the power of)
*;*****************************************************************************
res_hm        .usect flt_add”,1      ; result high mantissa
res_lm        .usect ”flt_add”,1     ; result low mantissa
res_exp       .usect ”flt_add”,1     ; result exponent
res_sign      .usect ”flt_add”,1     ; result sign
op2_hm        .usect  ”flt_add”,1    ; OP2 high mantissa
op2_lm        .usect ”flt_add”,1     ; OP2 low mantissa
op2_se        .usect ”flt_add”,1     ; OP2 sign and exponent
op1_se        .usect ”flt_add”,1     ; OP1 sign and exponent
op1_hm        .usect ”flt_add”,1     ; OP1 high mantissa
op1_lm        .usect ”flt_add”,1     ; OP1 low mantissa
op1_msw       .usect ”flt_add”,1     ; OP1 packed high word
op1_lsw       .usect ”flt_add”,1     ; OP1 packed low word
op2_msw       .usect ”flt_add”,1     ; OP2 packed high word
op2_lsw       .usect ”flt_add”,1     ; OP2 packed low word
err_no        .usect ”flt_add”,1     ;
        .mmregs
*****************************************************************
* Floating point number 12.0 can be represented as 1100 = 1.100 x 23 => sign =0
*   biased exponent = 127+3 = 130
*             130 = 10000010
*         Mantissa 10000000000000000000000
* Thus 12.0 can be represented as 01000001010000000000000000000000= 4140h
*******************************************************************************
*
K_OP1_HIGH    .set   4140h         ; floating point number 12.0
K_OP1_LOW .set   0000h
K_OP2_HIGH    .set   4140h         ; floating point number 12.0
K_OP2_LOW .set   0000h
       .mmregs
       .text
start_flt:
       RSBX   C16
       LD     #res_hm,DP           ; initialize the page pointer
       LD     #K_OP2_HIGH,A    ; load floating #2 – 12
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
       STL A,op2_msw
       LD     #K_OP2_LOW,A
       STL A,op2_lsw
       LD     #K_OP1_HIGH,A    ; load floating #1 –  12
       STL A,op1_msw
       LD     #K_OP1_LOW,A
       STL A,op1_lsw
*
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;    Test OP1 for special case treatment of zero.
*;    Split the MSW of OP1 in the accumulator.
*;    Save the exponent on the stack [xxxx xxxx EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store the mantissa as a signed value
*;*****************************************************************************
*
       DLD op1_msw,A           ; load the OP1 high word
       SFTA   A,8                 ; shift right by 8
       SFTA   A,–8
       BC     op1_zero,AEQ        ; If op1 is 0, jump to special case
       LD     A,B                 ; Copy OP1 to acc B
       RSBX   SXM                 ; Reset for right shifts used for masking
       SFTL   A,1                 ; Remove sign bit
       STH    A,–8,op1_se         ; Store exponent to stack
       SFTL   A,8                    ; Remove exponent
       SFTL   A,–9
       ADD #080h,16,A              ; Add implied 1 to mantissa
       XC     1,BLT                   ; Negate OP1 mantissa for negative values
       NEG A
       SSBX   SXM                 ; Make sure OP2 is sign–extended
       DST A,op1_hm               ; Store mantissa
*
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;    Test OP1 for special case treatment of zero.
*;    Split the MSW of OP1 in the accumulator.
*;    Save the exponent on the stack [xxxx xxxx EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store the mantissa as a signed value
*;*****************************************************************************
*
       DLD op2_msw,A           ; Load acc with op2
       BC     op2_zero,AEQ        ; If op2 is 0, jump to special case
       LD     A,B                 ; Copy OP2 to acc B
       SFTL   A,1                 ; Remove sign bit
       STH A,–8,op2_se            ; Store exponent to stack
       RSBX   SXM                    ; Reset for right shifts used for masking
       SFTL   A,8                 ; Remove exponent
       SFTL   A,–9
       ADD #080h,16,A              ; Add implied 1 to mantissa
       XC  1,BLT                   ; Negate OP2 mantissa for negative values
       NEG A
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
       SSBX   SXM                ; Set sign extension mode
       DST A,op2_hm              ; Store mantissa
**;*************************************************************************
*;       EXPONENT COMPARISON
*;  Compare exponents of OP1 and OP2 by subtracting: exp OP2 – exp OP1
*;  Branch to one of three blocks of processing
*;     Case 1:  exp OP1 is less than exp OP2
*;     Case 2:  exp OP1 is equal to exp OP2
*;     Case 3:  exp OP1 is greater than exp OP2
*;*************************************************************************
*
       LD     op1_se,A           ; Load OP1 exponent
       LD     op2_se,B           ; Load OP2 exponent
*
       SUB A,B             ; Exp OP2 – exp OP1 –> B
       BC     op1_gt_op2,BLT     ; Process OP1 > OP2
       BC     op2_gt_op1,BGT     ; Process OP2 > OP2
*
*;*************************************************************************
*;       exp OP1 = exp OP2
*;  Mantissas of OP1 and OP2 are normalized identically.
*;  Add mantissas:  mant OP1 + mant OP2
*;  If result is zero, special case processing must be executed.
*;  Load exponent for possible adjustment during normalization of result
*;*************************************************************************
a_eq_b
       DLD op1_hm,A            ; Load OP1 mantissa
       DADD   op2_hm,A            ; Add OP2 mantissa
       BC     res_zero,AEQ     ; If result is zero, process special case
       LD     op1_se,B            ; Load exponent in preparation for normalizing
*
*;*************************************************************************
*;    normalize THE RESULT
*;    Take the absolute value of the result.
*;    Set up to normalize the result.
*;    The MSB may be in any of bits 24 through 0.
*;    Left shift by six bits; bit 24 moves to bit 30, etc.
*;    Normalize resulting mantissa with exponent adjustment.
*;*************************************************************************
*
normalize
       STH A,res_sign          ; Save signed mantissa on stack
       ABS A                   ; Create magnitude value of mantissa
       SFTL   A,6              ; Pre–normalize adjustment of mantissa
       EXP A                   ; Get amount to adjust exp for normalization
       NOP
       NORM   A                   ; Normalize the result
       ST     T,res_exp        ; Store exp adjustment value
       ADD #1,B                ; Increment exp to account for implied carry
       SUB res_exp,B        ; Adjust exponent to account for normalization
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
*
*;*************************************************************************
*;       POST–NORMALIZATION ADJUSTMENT AND STORAGE
*;  Test result for underflow and overflow.
*;  Right shift mantissa by 7 bits.
*;  Mask implied 1
*;  Store mantissa on stack.
*;*************************************************************************
*
normalized
       STL B,res_exp        ; Save result exponent on stack
       BC     underflow,BLEQ      ; process underflow if occurs
       SUB #0FFh,B             ; adjust to check for overflow
       BC     overflow,BGEQ    ; process overflow if occurs
       SFTL   A,–7                ; Shift right to place mantissa for splitting
       STL A,res_lm            ; Store low mantissa
       AND #07F00h,8,A         ; Eliminate implied one
       STH A,res_hm            ; Save result mantissa on stack**
;*************************************************************************
*;*************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
*;*************************************************************************
*
       LD     res_sign,9,A     ; 0000 000S 0000 0000 0000 0000 0000 0000
       AND #100h,16,A
       ADD res_exp,16,A     ; 0000 000S EEEE EEEE 0000 0000 0000 0000
       SFTL   A,7              ; SEEE EEEE E000 0000 0000 0000 0000 0000
       DADD   res_hm,A            ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
*
*;*************************************************************************
*;       CONTEXT RESTORE
*;  Pop local floating point variables.
*;  Restore contents of B accumulator, T Register
*;*************************************************************************
*
return_value
       NOP
       NOP
       RET
*
*;*************************************************************************
*;       exp OP1 > exp OP2
*;  Test if the difference of the exponents is larger than 24 (precision of the mantissa)
*;  Return OP1 as the result if OP2 is too small.
*;  Mantissa of OP2 must be right shifted to match normalization of OP1
*;  Add mantissas:  mant OP1 + mant op2
*;*************************************************************************
*
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
op1_gt_op2
       ABS B                       ; If exp OP1 >= exp OP2 + 24 then return OP1
       SUB #24,B
       BC     return_op1,BGEQ
       ADD #23,B                   ; Restore exponent difference value
       STL B,res_sign              ; Store exponent difference to be used as RPC
       DLD op2_hm,A                ; Load OP2 mantissa
       RPT res_sign                ; Normalize OP2 to match OP1
       SFTA   A,–1
       BD     normalize            ; Delayed branch to normalize result
       LD     op1_se,B                ; Load exponent value to prep for normaliza-
tion
       DADD   op1_hm,A                ; Add OP1 to OP2
*
*;*****************************************************************************
*;
*;       OP1 < OP2
*; Test if the difference of the exponents is larger than 24 (precision of the mantissa).
*; Return OP2 as the result if OP1 is too small.
*; Mantissa of OP1 must be right shifted to match normalization of OP2.
*; Add mantissas:  mant OP1 + mant OP2
*;*****************************************************************************
*;
op2_gt_op1
       SU     B  #24,B                ; If exp OP2 >= exp OP1 + 24 then return OP2
       BC     return_op2,BGEQ
       ADD #23,B                   ; Restore exponent difference value
       STL B,res_sign              ; Store exponent difference to be used as RPC
       DLD op1_hm,A                ; Load OP1 mantissa
       RPT res_sign                ; Normalize OP1 to match OP2
       SFTA   A,–1   BD normalize ; Delayed branch to normalize result
       LD     op2_se,B                ; Load exponent value to prep for normalization
       DADD   op2_hm,A                ; Add OP2 to OP1
*;*****************************************************************************
*;
*;       OP1 << OP2  or OP1 = 0
*;*****************************************************************************
*;
*
return_op2
op1_zero
       BD     return_value
       DLD op2_msw,A           ; Put OP2 as result into A
       NOP
*
*;*****************************************************************************
*;
*;       OP1 << OP2  or  OP1 = 0
*;*****************************************************************************
*;
*
op2_zero
return_op1
       DLD op1_hm,A                   ; Load signed high mantissa of OP1
       BC     op1_pos,AGT                ; If mantissa is negative . . .
       NEG A                          ; Negate it to make it a positive value
       ADDM   #100h,op1_se            ; Place the sign value back into op1_se
```

Example 3–9.Add Two Floating-Point Numbers (Continued)

```
op1_pos
       SUB #80h,16,A              ; Eliminate implied one from mantissa
       LD     op1_se,16,B               ; Put OP1 back together in acc A as a re-
sult
       BD     return_value
       SFTL   B,7
       ADD B,A
*;*****************************************************************************
*;       overflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
*
overflow
       ST     #2,err_no              ; Load error no
       LD     res_sign,16,A          ; Pack sign of result
       AND #8000,16,A                ; Mask to get sign
       OR     #0FFFFh,A              ; Result low mantissa = 0FFFFh
       BD     return_value           ; Branch delayed
       ADD #07F7Fh,16,A           ; Result exponent = 0FEh
                                     ; Result high mant = 07Fh
*****************************************************************************
*;       underflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
*
underflow
       ST     #1,err_no              ; Load error no
       RET
res_zero
       BD     return_value           ; Branch delayed
       SUB A,A                       ; For underflow result = 0
       NOP
```

Example 3–10. Multiply Two Floating-Point Numbers

```
*;*****************************************************************************
*;  Float_MUL – multiply two floating point numbers
*;  Copyright (c) 1993–1994 Texas Instruments Incorporated
*;*****************************************************************************
*;*****************************************************************************
;This routine multiplies two floating point  numbers. OP1 and OP2 are each unpacked
;into sign, exponent, and two words of mantissa. If either exponent is zero
;special case processing is initiated. The exponents are summed. If the result is
;less than zero underflow has occurred. If the result is zero, underflow may have
;occurred. If the result is equal to 254 overflow may have occurred.  If the result
;is greater than 254 overflow has occurred. Underflow processing returns a value
;of zero. Overflow processing returns the largest magnitude value along with the
;appropriate sign. If no special cases are detected, a 24x24–bit multiply is
;executed. The result of the exclusive OR of the sign bits, the sum of the
;exponents and the ;24 bit truncated mantissa are packed and returned
*;*****************************************************************************
```

Example 3–10. Multiply Two Floating-Point Numbers (Continued)

```
*;       resource utilization:  B accumulator, T–register
*;       status bits affected: TC, C, SXM, OVM, C16
*;       entry requirements : CPL bit set
*;*****************************************************************************
; Floating Point Format – Single Precision
*------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*------------------------------------------------------------------------------
*------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*------------------------------------------------------------------------------
*; Single precision floating point format is a 32 bit format  consisting of a    *
*; 1 bit sign field, an 8 bit exponent field, and a 23 bit mantissa field. The *
*; fields are defined as follows.                                                *
*;            Sign <S>          : 0 = positive values; 1 = negative values
*;            Exponent <E7–E0>  : offset binary format
*;                                00 = special cases (i.e. zero)
*;                                01 = exponent value + 127 = –126
*;                           FE = exponent value + 127 = +127
*;                                FF = special cases (not implemented)
*;  Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                                1.M22M21...M1M0
*;            Range             : –1.9999998 e+127 to –1.0000000 e–126
*;                                +1.0000000 e–126 to +1.9999998 e+
*;                                (where e represents 2 to the power of)
*;                                –3.4028236 e+38  to –1.1754944 e–
*;                                +1.1754944 e–38  to +3.4028236 e+38
*;                                (where e represents 10 to the power of)
*;*****************************************************************************
res_hm    .usect "flt_add",1            ;result high mantissa
res_lm    .usect "flt_add",1            ;result low mantissa
res_exp      .usect "flt_add",1            ;result exponent
res_sign     .usect "flt_add",1            ; result sign
op2_hm    .usect "flt_add",1            ; OP2 high mantissa
op2_lm    .usect "flt_add",1            ; OP2 low mantissa
op2_se    .usect "flt_add",1            ; OP2 sign and exponent
op1_se    .usect "flt_add",1            ; OP1 sign and exponent
op1_hm    .usect "flt_add",1            ; OP1 high mantissa
op1_lm    .usect "flt_add",1            ; OP1 low mantissa
op1_msw      .usect "flt_add",1            ; OP1 packed high word
op1_lsw      .usect "flt_add",1            ; OP1 packed low word
op2_msw      .usect "flt_add",1            ; OP2 packed high word
op2_lsw      .usect "flt_add",1            ; OP2 packed low word
err_no    .usect "flt_add",1            ;
*****************************************************************
* Floating point number 12.0 can be represented as 1100 = 1.100 x 23 => sign =0
*                                      biased exponent = 127+3 = 130
*                                     130 = 10000010
*                                     Mantissa 10000000000000000000000
```

Example 3–10. Multiply Two Floating-Point Numbers (Continued)

```
*  Thus 12.0 can be represented as 01000001010000000000000000000000= 4140h
*******************************************************************************
*
K_OP1_HIGH          .set      4140h           ; floating point number 12.0
K_OP1_LOW       .set      0000h
K_OP2_HIGH          .set      4140h           ; floating point number 12.0
K_OP2_LOW       .set      0000h
          .mmregs
          .text
start_flt:
          RSBX   C16                        ; Insure long adds for later
          LD     #res_hm,DP                    ; initialize the page pointer
          LD     #K_OP2_HIGH,A              ; load floating #2 – 12
          STL A,op2_msw
          LD     #K_OP2_LOW,A
          STL A,op2_lsw
          LD     #K_OP1_HIGH,A              ; load floating #1 –  12
          STL A,op1_msw
          LD     #K_OP1_LOW,A
          STL A,op1_lsw
*
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value
*;    Store entire mantissa with a long word store
*;*****************************************************************************
          DLD op1_msw,A           ; OP1
          SFTA   A,8
          SFTA   A,–8
          BC     op_zero,AEQ             ; if op1 is 0, jump to special case
          STH A,–7,op1_se         ; store sign AND exponent to stack
          STL A,op1_lm               ; store low mantissa
          AND #07Fh,16,A             ; mask off sign & exp to get high mantissa
          ADD #080h,16,A             ; ADD implied 1 to mantissa
          STH A,op1_hm               ; store mantissa to stack
*;*****************************************************************************
*;       CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP2 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
          DLD op2_msw,A           ; load acc a with OP2
          BC     op_zero,AEQ             ; if OP2 is 0, jump to special case
          STH A,–7,op2_se         ; store sign and exponent to stack
          STL A,op2_lm            ; store low mantissa
          AND #07Fh,16,A             ; mask off sign & exp to get high mantissa
          ADD #080h,16,A             ; add implied 1 to mantissa
          STH A,op2_hm            ; store mantissa to stack
```

Example 3–10. Multiply Two Floating-Point Numbers (Continued)

```
*;*************************************************************************
*;       SIGN EVALUATION
*;  Exclusive OR sign bits of OP1 and OP2 to determine sign of result.
*;*************************************************************************
         LD  op1_se,A               ; load sign and exp of op1 to acc
         XOR op2_se,A               ; xor with op2 to get sign of result
         AND #00100h,A           ; mask to get sign
         STL A,res_sign             ; save sign of result to stack
*;*************************************************************************
*;       EXPONENT SUMMATION
*;  Sum the exponents of OP1 and OP2 to determine the result exponent.  Since
*;  the exponents are biased (excess 127) the summation must be decremented
*;  by the bias value to avoid double biasing the result
*;  Branch to one of three blocks of processing
*;    Case 1:  exp OP1 + exp OP2 results in underflow (exp < 0)
*;    Case 2:  exp OP1 + exp OP2 results in overflow (exp >= 0FFh)
*;    Case 3:  exp OP1 + exp OP2 results are in range (exp >= 0 & exp < 0FFh)
*;      NOTE:  Cases when result exp = 0 may result in underflow unless there
*;             is a carry in the result that increments the exponent to 1.
*;             Cases when result exp = 0FEh may result in overflow if there
*;             is a carry in the result that increments the exponent to 0FFh.
*;*************************************************************************
         LD     op1_se,A               ; Load OP1 sign and exponent
         AND #00FFh,A               ; Mask OP1 exponent
         LD     op2_se,B               ; Load OP2 sign and exponent
         AND #0FFh,B                ; Mask OP2 exponent
         SUB #07Fh,B                ; Subtract offset (avoid double bias)
         ADD B,A                    ; Add OP1 exponent
         STL A,res_exp           ; Save result exponent on stack
         BC     underflow,ALT       ; branch to underflow handler if exp < 0
         SUB #0FFh,A             ; test for overflow
         BC     overflow,AGT     ; branch to overflow is exp > 127
*;*************************************************************************
*;       MULTIPLICATION
*; Multiplication is implemented by parts.  Mantissa for OP1 is three bytes
*; identified as Q, R, and S
*; (Q represents OP1 high mantissa and R and S represent the two bytes of OP1 low
*; mantissa).  Mantissa for
*; OP2 is also 3  bytes identified as X, Y, and Z (X represents OP2 high mant and
*; Y and Z represent the two bytes
*; of OP2 low mantissa).  Then
*;                       0  Q  R  S      (mantissa of OP1)
*;                    x  0  X  Y  Z      (mantissa of OP2)
*;                      ===========
*;                       RS*YZ         <-- save only upper 16 bits of result
*;                       RS*0X
*;                       0Q*YZ
*;                       0Q*0X         <-- upper 16 bits are always zero
*;                     ===========
*;                       result        <-- result is always in the internal 32 bits
*;(which ends up in the accumulator) of the possible 64 bit product
*;*************************************************************************
```

Example 3–10. Multiply Two Floating-Point Numbers (Continued)

```
          LD     op1_lm,T            ; load low mant of op1 to T register
          MPYU   op2_lm,A            ; RS * YZ
          MPYU   op2_hm,B            ; RS * 0X
          ADD A,–16,B             ; B = (RS * YZ) + (RS * 0X)
          LD     op1_hm,T            ; load high mant of op1 to T register
          MPYU   op2_lm,A         ; A = 0Q * YZ
          ADD B,A                 ; A = (RS * YZ) + (RS * 0X) + (0Q * YZ)
          MPYU   op2_hm,B            ; B = 0Q * 0X
          STL B,res_hm            ; get lower word of 0Q * 0X
          ADD res_hm,16,A         ; A = final result
*;*****************************************************************************
*;       POST–NORMALIZATION ADJUSTMENT AND STORAGE
*;  Set up to adjust the normalized result.
*;    The MSB may be in bit 31.  Test this case and increment the exponent
*;    and right shift mantissa 1 bit so result is in bits 30 through 7
*;  Right shift mantissa by 7 bits.
*;  Store low mantissa on stack.
*;  Mask implied 1 and store high mantissa on stack.
*;  Test result for underflow and overflow.
*******************************************************************************
          ADD #040h,A             ; Add rounding bit
          SFTA   A,8              ; sign extend result to check if MSB is in 31
          SFTA   A,–8
          RSBX   SXM              ; turn off sign extension for normalization
          LD     res_exp,B        ; load exponent of result
          BC     normalized,AGEQ  ; check if MSB is in 31
          SFTL   A,–1                ; Shift result so result is in bits 30:7
          ADD    #1,B                ; increment exponent
          STL B,res_exp        ; save updated exponent normalized
          BC     underflow,BLEQ      ; check for underflow
          SUB #0FFh,B             ; adjust to check for overflow
          BC     overflow,BGEQ    ; check for overflow
          SFTL   A,–7                ; shift to get 23 msb bits of mantissa result
          STL A,res_lm            ; store low mantissa result
          AND #07F00h,8,A         ; remove implied one
          STH A,res_hm         ; store the mantissa result
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
;*****************************************************************************
          LD     res_sign,16,A    ; 0000 000S 0000 0000 0000 0000 0000 0000
          ADD res_exp,16,A     ; 0000 000S EEEE EEEE 0000 0000 0000 0000
          SFTL   A,7              ; SEEE EEEE E000 0000 0000 0000 0000 0000
          DADD   res_hm,A            ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
*;*****************************************************************************
*;        CONTEXT RESTORE
*;*****************************************************************************
return_value
```

Example 3–10. Multiply Two Floating-Point Numbers (Continued)

```
op_zero
          nop
          nop
          ret
*;*****************************************************************************
*;
*;  overflow PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
overflow
          ST     #2,err_no        ; Load error no
          LD     res_sign,16,B    ; Load sign of result
          LD     #0FFFFh,A        ; Result low mantissa = 0FFFFh
          OR     B,7,A               ; Add sign bit
          BD     return_value     ; Branch delayed
          ADD #07F7Fh,16,A     ; Result exponent = 0FEh
                                     ; Result high mant = 07Fh
*;*****************************************************************************
*;       UNDERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*****************************************************************************
underflow
          ST     #1,err_no        ; Load error no
          BD     return_value     ; Branch delayed
          SUB A,A              ; For underflow result = 0
          NOP
```

Example 3–11. Divide a Floating-Point Number by Another

```
*;*****************************************************************************
*;  FLOAT_DIV – divide two floating point numbers
*;  Copyright (c) 1993–1994 Texas Instruments Incorporated
*;*****************************************************************************
;Implementation: OP1 and OP2 are each unpacked into sign, exponent, and two words
;of mantissa.  If either exponent is zero special case processing is initiated.
;The difference of the exponents are taken. IF the result is less than zero underflow
;has occurred. If the result is zero, underflow may have occurred. If the result
;is equal to 254 overflow may have occurred.  If the result is greater than 254
;overflow has occurred.
; Underflow processing returns a value of zero. Overflow processing returns the
;largest magnitude value along with the appropriate sign.  If no special cases are
;detected, a 24x24–bit divide is ;executed.  The result of the exclusive OR of the
;sign bits, the difference of the exponents and the 24 bit truncated mantissa are
;packed and returned.
*;*****************************************************************************
*;*****************************************************************************
*;
*;       resource utilization:  B accumulator , T register
*;       status bits affected: TC, C, SXM, OVM, C16
*;       entry requirements : CPL bit set
*;*****************************************************************************
```

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
; Floating Point Format – Single Precision
*-------------------------------------------------------------------------------
*| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*|  S | E7 | E6 | E5 | E4 | E3 | E2 | E1 | E0 | M22| M21| M20| M19| M18| M17| M16|
*-------------------------------------------------------------------------------
*-------------------------------------------------------------------------------
*| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
*|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
*| M15| M14| M13| M12| M11| M10| M9 | M8 | M7 | M6 | M5 | M4 | M3 | M2 | M1 | M0 |
*-------------------------------------------------------------------------------
*; Single precision floating point format is a 32 bit format  consisting of a 1
bit sign field, an 8 bit exponent  *
*; field, and a 23 bit mantissa field. The fields are defined as follows
*
*;            Sign <S>          : 0 = positive values; 1 = negative values
*;            Exponent <E7–E0>  : offset binary format
*;                                00 = special cases (i.e. zero)
*;                             01 = exponent value + 127 = –126
*;                                FE = exponent value + 127 = +127
*;                                FF = special cases (not implemented)
*; Mantissa <M22–M0> : fractional magnitude format with implied 1
*;                                1.M22M21...M1M0
*;            Range             : –1.9999998 e+127 to –1.0000000 e–126
*;                                +1.0000000 e–126 to +1.9999998 e+127
*;                                (where e represents 2 to the power of)
*;                                –3.4028236 e+38  to –1.1754944 e–38
*;                                +1.1754944 e–38  to +3.4028236 e+
*;                                (where e represents 10 to the power of)
*;*****************************************************************************
res_hm              .usect ”flt_div”,1
res_lm              .usect ”flt_div”,1
res_exp                .usect ”flt_div”,1
res_sign               .usect ”flt_div”,1
op2_hm              .usect ”flt_div”,1
op2_lm              .usect ”flt_div”,1
op2_se              .usect ”flt_div”,1
op1_se              .usect ”flt_div”,1
op1_hm              .usect ”flt_div”,1
op1_lm              .usect ”flt_div”,1
op1_msw                .usect ”flt_div”,1
op1_lsw                .usect ”flt_div”,1
op2_msw                .usect ”flt_div”,1
op2_lsw                .usect ”flt_div”,1
err_no              .usect ”flt_div”,1
       .mmregs
*
*
```

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
K_divisor_high          .set      4140h
K_divisor_low       .set      0000h
K_dividend_high     .set      4140h
K_dividend_low          .set      0000h
                        .sect     ”vectors”
       B                float_div
       NOP
       NOP
       .text
float_div:
       LD     #res_hm,DP                ; initialize the page pointer
       LD     #K_divisor_high,A         ; load floating #2 – 12
       STL A,op2_msw
       LD     #K_divisor_low,A
       STL A,op2_lsw
       LD     #K_dividend_high,A     ; load floating #1 –  12
       STL A,op1_msw
       LD     #K_dividend_low,A
       STL A,op1_lsw
*********************************************************
       RSBX      C16                 ; Insure long adds for later
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
       DLD op1_msw,A              ; load acc a with OP1
       SFTA   A,8
       SFTA   A,–8
       BC     op1_zero,AEQ           ; if op1 is 0, jump to special case
       STH A,–7,op1_se               ; store sign and exponent to stack
       STL A,op1_lm                  ; store low mantissa
       AND #07Fh,16,A                ; mask off sign & exp to get high mantissa
       ADD #080h,16,A                ; ADD implied 1 to mantissa
       STH A,op1_hm                  ; store mantissa to stack
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – UNPACK
*;  Test OP1 for special case treatment of zero.
*;  Split the MSW of A in the accumulator.
*;    Save the sign and exponent on the stack [xxxx xxxS EEEE EEEE].
*;    Add the implied one to the mantissa value.
*;    Store entire mantissa with a long word store
*;*****************************************************************************
       DLD op2_msw,A              ; load acc a with OP2
       BC     op2_zero,AEQ           ; if OP2 is 0, divide by zero
       STH A,–7,op2_se               ; store sign and exponent to stack
       STL A,op2_lm                  ; store low mantissa
       AND #07Fh,16,A             ; mask off sign & exp to get high mantissa
```

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
      ADD #080h,16,A            ; ADD implied 1 to mantissa
      STH A,op2_hm              ; store mantissa to stack
*
*;*****************************************************************************
*;        SIGN EVALUATION
*;  Exclusive OR sign bits of OP1 and OP2 to determine sign of result.
*;************************   ************************************************
*     LD    op1_se,A              ; load sign and exp of op1 to acc
      XOR op2_se,A              ; xor with op2 to get sign of result
      AND #00100h,A           ; mask to get sign
      STL A,res_sign            ; save sign of result to stack
*
*;*****************************************************************************
*;        EXPONENT SUMMATION
*;  Find difference between operand exponents to determine the result exponent. *
*   Since the subtraction process removes the bias it must be re-added in.      *
*
*;  Branch to one of three blocks of processing
*;    Case 1:  exp OP1 + exp OP2 results in underflow (exp < 0)
*;    Case 2:  exp OP1 + exp OP2 results in overflow (exp >= 0FFh)
*;    Case 3:  exp OP1 + exp OP2 results are in range (exp >= 0 & exp < 0FFh)
*;        NOTE:  Cases when result exp = 0 may result in underflow unless there *
*         is a carry in the result that increments         the exponent to 1.   *
*         Cases when result exp = 0FEh may result in overflow if there is a carry  *
*         in the result that increments the exponent to 0FFh.
*;*****************************************************************************
*
      LD    op1_se,A              ; Load OP1 sign and exponent
      AND #0FFh,A               ; Mask OP1 exponent
*
      LD    op2_se,B              ; Load OP2 sign and exponent
      AND #0FFh,B               ; Mask OP2 exponent
*
      ADD #07Fh,A               ; Add offset (difference eliminates offset)
      SUB B,A                 ; Take difference between exponents
      STL A,res_exp           ; Save result exponent on stack
*
      BC    underflow,ALT       ; branch to underflow handler if exp < 0
      SUB #0FFh,A               ; test for overflow
      BC    overflow,AGT        ; branch to overflow is exp > 127
*
*;*****************************************************************************
*;        DIVISION
*; Division is implemented by parts. The mantissas for both OP1 and OP2 are left shifted
*  in the  32 bit field to reduce the effect of secondary and tertiary contributions to
*  the final result. The left shifted results are identified as OP1'HI, OP1'LO, OP2'HI,
*  and OP2'LO where OP1'HI and OP2'HI have the xx most significant bits of the mantissas
*  and OP1'LO and OP2'LO contain the remaining bits *  of each mantissa. Let QHI and QLO
*  represent the two portions of the resultant mantissa. Then
```

$$QHI + QLO = \frac{OPI'HI + OPI'LO}{OP2'HI + OP2'LO} = \frac{OPI'HI + OPI'LO}{OP2'HI} * \frac{1}{\left(1 + \frac{OP2'LO}{OP2'HI}\right)}$$

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
*;     Now let   X = OP2'LO/OP2'HI
*;     Then by Taylor's Series Expansion
*      1/(1 + x) = 1–x + x²–x³ + ........
*;     Since OP2'HI contains the first xx significant bits of the OP2 mantissa,*
X = OP2'LO/OP2'HI < 2–yy*;    Therefore the X2 term and all subsequent terms are less
than the least significant
*      bit of the 24–bit result and can be dropped. The result then becomes
*      QHI + QLO = (OPI'HI + OPI'LO)/(OP2'HI + OP2'LO) * (1 – OP2'LO/OP2'HI)

                 = (QHI + QLO) * (1 – OP2'LO/OP2'HI)
*;     where Q'HI and Q'LO represent the first approximation of the result. Also since
*      Q'LO and OP2'LO/OP2'HI are less significant the 24th bit of the result, this
*      product term can be dropped so
*      QHI + QLO = (OPI'HI + OPI'LO)/(OP2'HI + OP2'LO) = (OPI'HI + OPI'LO)/OP2'HI * 1/(1 + OP2'LO/OP2'HI)
that
*;*****************************************************************************
             DLD op1_hm,A                ; Load dividend mantissa
       SFTL   A,6                  ; Shift dividend in preparation for division
*
       DLD op2_hm,B                ; Load divisor mantissa
       SFTL   B,7                  ; Shift divisor in preparation for division
       DST B,op2_hm                ; Save off divisor
*
       RPT #14                   ; QHI = OP1'HI/OP2'HI
       SUBC   op2_hm,A
       STL A,res_hm                ; Save QHI
*
       SUBS   res_hm,A                 ; Clear QHI from ACC
       RPT #10                   ; Q'LO = OP1'LO / OP2'HI
       SUBC   op2_hm,A
       STL A,5,res_lm              ; Save Q'LO*
       LD     res_hm,T                 ; T = Q'HI
       MPYU   op2_lm,A                 ; Store Q'HI * OP2'LO in acc A
       SFTL   A,–1                     ;*
       RPT #11                   ; Calculate Q'HI * OP2'LO / OP2'HI
       SUBC   op2_hm,A                 ;   (correction factor)
       SFTL   A,4                  ; Left shift to bring it to proper range
       AND #0FFFFh,A             ; Mask off correction factor
*
       NEG A                       ; Subtract correction factor
       ADDS   res_lm,A                 ; Add Q'LO
       ADD res_hm,16,A             ; Add Q'HI
*
```

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
*;*****************************************************************************
*;        POST–NORMALIZATION ADJUSTMENT AND STORAGE
*;  Set up to adjust the normalized result. The MSB may be in bit 31. Test this
case and increment the exponent and right shift mantissa 1 bit so result is in
bits 30 through 7. Right shift mantissa by 7 bits. Store low mantissa on stack.
Mask implied 1 and store high mantissa on stack. Test result for underflow and
overflow.
*;*****************************************************************************
*
       LD     res_exp,B           ; Load result exponent
       EXP    A                      ; Get amount to adjust exp for normalizationNOP
       NORM   A                      ; Normalize the result
       ST     T,res_exp              ; Store the exponent adjustment value
       SUB    res_exp,B              ; Adjust exponent (add either zero or one)
       SFTL   A,–1                   ; Pre–scale adjustment for rounding
       ADD    #1,B                   ; Adjust exponent
       ADD    #020h,A             ; Add rounding bit
       EXP    A                      ; Normalize after rounding     NOP
       NORM   A                      ;
       ST     T,res_exp              ; Adjust exponent for normalization
       SUB    res_exp,B              ;
       STL    B,res_exp              ; Save exponent
       BC     underflow,BLEQ      ; process underflow if occurs
       SUB    #0FFh,B             ; adjust to check for overflow
       BC     overflow,BGEQ       ; process overflow if occurs
       SFTL   A,–7                   ; Shift right to place mantissa for splitting
       STL    A,res_lm               ; Save result low mantissa
       AND    #07F00h,8,          ; Eliminate implied one
       STH    A,res_hm               ; Save result mantissa on stack
*
*;*****************************************************************************
*;        CONVERSION OF FLOATING POINT FORMAT – PACK
*;  Load sign.
*;  Pack exponent.
*;  Pack mantissa.
*;*****************************************************************************
*
       LD     res_sign,16,A       ; 0000 000S 0000 0000 0000 0000 0000 0000
       ADD res_exp,16,A        ; 0000 000S EEEE EEEE 0000 0000 0000 0000
       SFTL   A,7                 ; SEEE EEEE E000 0000 0000 0000 0000 0000
       DADD   res_hm,A               ; SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
**;*****************************************************************************
*;        CONTEXT RESTORE
*;*****************************************************************************
return_value
op1_zero
   ret
*
```

Example 3–11. Divide a Floating-Point Number by Another (Continued)

```
*;*************************************************************************
*;        OVERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*************************************************************************
overflow
       ST     #2,err_no           ; Load error no
       SAT A                      ; Result exponent = 0FEh
       SUB #081h,16,A             ; Result high mant = 07Fh
       BD     return_value        ; Branch delayed
       LD     res_sign,16,B       ; Load sign of result
       OR     B,7,A                 ; Pack sign*
*;*************************************************************************
*;
UNDERFLOW PROCESSING
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*************************************************************************
*
underflow
       ST     #1,err_no           ; Load error no
       BD     return_value        ; Branch delayed
       sub A,A                ; For underflow result = 0
       nop
**;
*************************************************************************
*; DIVIDE BY ZERO
*;  Push errno onto stack.
*;  Load accumulator with return value.
*;*************************************************************************
op2_zero
       ST     #3,err_no           ; Load error no
       SAT A                      ; Result exponent = FEh
                                     ; Result low mant = FFFFh
       LD     op1_se,16,B            ; Load sign and exponent of OP1
       AND #100h,16,B             ; Mask to get sign of OP1
       OR  B,7,A                  ; Pack sign
       BD     return_value        ; Branch delayed
       SUB #081h,16,A             ; Result high mant = 7Fh
       NOP
```

3.6 Logical Operations

DSP-application systems perform many logical operations, including bit manipulation and packing and unpacking data. A digital modem uses a scrambler and a descrambler to perform bit manipulation. The input bit stream is in a packed format of 16 bits. Each word is unpacked into 16 words of 16-bit data, with the most significant bit (MSB) as the original input bit of each word. The unpack buffer contains either 8000h or 0000h, depending upon the bit in the original input-packed 16-bit word. The following polynomial generates a scrambled output, where the $\oplus$ sign represents modulus 2 additions from the bitwise exclusive OR of the data values:

$$\text{Scrambler output} = 1 \oplus x^{-18} \oplus x^{-23}$$

The same polynomial sequence in the descrambler section reproduces the original 16-bit input sequence. The output of the descrambler is a 16-bit word in packed format.

Example 3–12. Pack/Unpack Data in the Scrambler/Descrambler of a Digital Modem

```
; TEXAS INSTRUMENTS INCORPORATED
          .mmregs
                        .asg            AR1,UNPACK_BFFR
                        .asg            AR3,SCRAM_DATA_18
                        .asg            AR4,SCRAM_DATA_23
                        .asg            AR2,DE_SCRAM_DATA_18
                        .asg            AR5,DE_SCRAM_DATA_23
d_scram_bffr        .usect       ”scrm_dat”,30
d_de_scram_bffr     .usect       ”dscrm_dt”,30
d_unpack_buffer     .usect       ”scrm_var”,100
d_input_bit            .usect       ”scrm_var”,1
d_pack_out             .usect       ”scrm_var”,1
d_asm_count            .usect       ”scrm_var”,1
K_BFFR_SIZE            .set            24
K_16                   .set            16
                       .def            d_input_bit
                       .def            d_asm_count
; Functional Description
;  This routine illustrates the pack and unpack of a data stream and
;  also bit manipulation. A digital scrambler and descrambler does the
;  bit manipulation and the input to the scrambler is in unpacked format
;  and the output of the descrambler is in packed 16–bit word.
;  scrambler_output = 1+x^–18+x^–23
;  additions are modulus 2 additions or bitwise exclusive OR of data
;  values. The same polynomial is used to generate the descrambler
;  output.
      .sect     ”scramblr”
```

Example 3–12. Pack/Unpack Data in the Scrambler/Descrambler of a Digital Modem (Continued)

```
scrambler_init:
       STM    #d_unpack_buffer,UNPACK_BFFR
       STM    #d_scram_bffr,SCRAM_DATA_23
       RPTZ       A,#K_BFFR_SIZE
       STL    A,*SCRAM_DATA_23+
       STM    #d_scram_bffr+K_BFFR_SIZE–1,SCRAM_DATA_23
       STM    #d_scram_bffr+17,SCRAM_DATA_18
       STM    #d_de_scram_bffr+K_BFFR_SIZE–1,DE_SCRAM_DATA_23
       STM    #d_de_scram_bffr+17,DE_SCRAM_DATA_18
       LD         #d_input_bit,Dp
       ST         #–K_16+1,d_asm_count
scramler_task:
;  the unpack data buffer has either 8000h or 0000h since the bit stream
;  is either 1 or 0
unpack_data:
       STM    #K_16–1,BRC
       RPTB       end_loop–1                        ; unpack the data into 16–bit
                                                    ; word
       PORTR      1h,d_input_bit                    ; read the serial bit stream
       LD         d_input_bit,15,A                ; mask thelower 15 bits
                                                    ; the MSB is the serial bit
                                                    ; stream
       STL    A,*UNPACK_BFFR                    ; store the 16 bit word
unpack_16_words
scrambler:
       LD         *SCRAM_DATA_18–%,A
       XOR    *SCRAM_DATA_23,A              ; A = x^–18+x^–23
       XOR    *UNPACK_BFFR,A                   ; A = A+x^0
       STL    A,*SCRAM_DATA_23–%            ; newest sample, for next
                                                    ; cycle it will be x(n–1)
       STL    A,*UNPACK_BFFR                   ; store the scrambled data
scramble_word
descrambler:
       LD         *DE_SCRAM_DATA_18–%,A
       XOR    *DE_SCRAM_DATA_23,A           ; A = x^–18+x^–23
       XOR    *UNPACK_BFFR,A                   ; A = A+x^0
       STL    A,*DE_SCRAM_DATA_23–%         ; newest sample, for next
                                                    ; cycle it will be x(n–1)
       STL    A,*UNPACK_BFFR                   ; store the scrambled data
de_scramble_word
; ASM field shifts the descrambler output MSB into proper bit position
;
pack_data
       RSBX   SXM                                  ; reset the SXM bit
       LD     d_asm_count,ASM
       LD     *UNPACK_BFFR+,A
       LD     A,ASM,A
       OR     d_pack_out,A                      ; start pack the data
       STL A, d_pack_out
       ADDM   #1,d_asm_count
```

Example 3–12. Pack/Unpack Data in the Scrambler/Descrambler of a Digital Modem (Continued)

```
pack_word
       SSBX   SXM                              ; enable SXM mode
end_loop
       NOP                                     ; dummy instructions nothing
                                                  ; with the code
       NOP
       .end
```

Chapter 4

Application-Specific Instructions and Examples

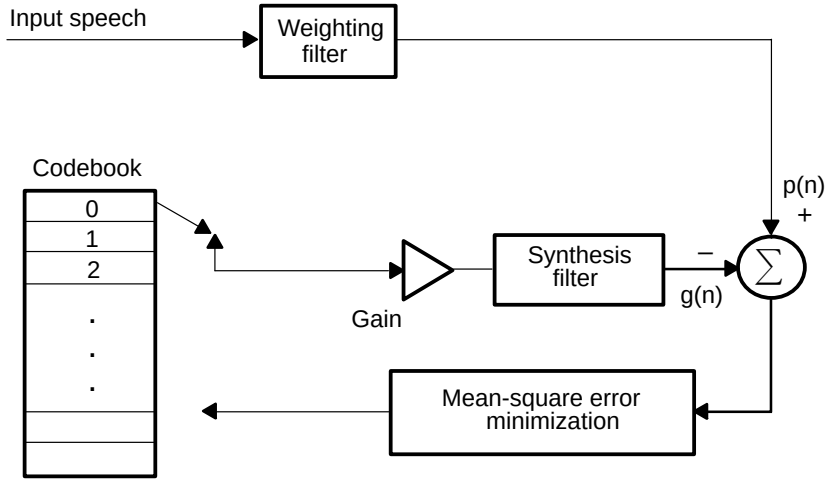
This chapter shows examples of application-specific instructions that the TMS320C54x™ (C54x) offers and the typical functions where they are used. Functions like codebook search and viterbi are widely used for speech coding and telecommunications.

Topic **Page**

4.1 Codebook Search for Excitation Signal in Speech Coding

A code-excited linear predictive (CELP) speech coder is widely used for applications requiring speech coding with a bit rate under 16K bps. The speech coder uses a vector quantization technique from codebooks to an excitation signal. This excitation signal is applied to a linear predictive-coding (LPC) synthesis filter. To obtain optimum code vectors from the codebooks, a codebook search is performed, which minimizes the mean-square error generated from weighted input speech and from the zero-input response of a synthesis filter. Figure 4–1 shows a block diagram of a CELP-based speech coder.

Figure 4–1. CELP-Based Speech Coder



To locate an optimum code vector, the codebook search uses Equation 4–1 to minimize the mean-square error.

Equation 4–1. Optimum Code Vector Localization

$$E_i = \sum_{i=0}^{N-1} \{p(n) - \gamma_i g_i(n)\}^2 \qquad \text{N : Subframe}$$

The variable $p(n)$ is the weighted input speech, $g_i(n)$ is the zero-input response of the synthesis filter, and $_i$ is the gain of the codebook.

The cross-correlation (c_i) of $p(n)$ and $g_i(n)$ is represented by Equation 4–2. The energy (G_i) of $g_i(n)$ is represented by Equation 4–3.

Equation 4–2. Cross Correlation Variable (c_i)

$$c_i = \sum_{i=0}^{N-1} g_i * p(n)$$

Equation 4–3. Energy Variable (G_i)

$$G_i = \sum_{i=0}^{N-1} g_i^2$$

Equation 4–1 is minimized by maximizing $\frac{c_i^2}{G_i}$. Therefore, assuming that a code vector with i = opt is optimal, Equation 4–4 is always met for any i. The codebook search routine evaluates this equation for each code vector and finds the optimum one.

Equation 4–4. Optimal Code Vector Condition

$$\left(\frac{c_i^2}{G_i}\right) \leq \left(\frac{c_{opt}^2}{G_{opt}}\right)$$

Example 4–1 shows the implementation algorithm for codebook search on C54x. The square (SQUR), multiply (MPYA), and conditional store (SRCCD, STRCD, SACCD) instructions are used to minimize the execution cycles. AR5 points to c_i and AR2 points to G_i. AR3 points to the locations of G_{opt} and c_{opt}^2. The value of i(*opt*) is stored at the location addressed by AR4.

Example 4–1. Codebook Search

```
                .title   ”CODEBOOK SEARCH”
                .mmregs
                .text
SEARCH:
                STM   #C,AR5          ;Set C(i) address
                STM   #G,AR2          ;Set G(i) address
                STM   #OPT,AR3        ;Set OPT address
                STM   #IOPT,AR4       ;Set IOPT address
                ST    #0,*AR4         ;Initialize lag
                ST    #1,*AR3+        ;Initialize Gopt
                ST    #0,*AR3–        ;Initialize C2opt
                STM   #N–1,BRC
                RPTB  Srh_End–1
                SQUR  *AR5+,A         ;A = C(i) * C(i)
                MPYA  *AR3+           ;B = C(i)^2 * Gopt
                MAS   *AR2+,*AR3–,B   ;B = C(i)^2 * Gopt –
                                      ;G(i) * C2opt,T = G(i)
                SRCCD *AR4,BGEQ       ;if(B >= 0) then
                                      ;iopt = BRC
                STRCD *AR3+,BGEQ      ;if(B >= 0) then
                                      ;Gopt = T
                SACCD A,*AR3–,BGEQ    ;if(B >= 0) then
                                      ;C2opt = A NOP
                NOP                   ;To save current BCR
Srh_End:                              ;*AR4 –> optimal index
                RET
                .end
```

4.2 Viterbi Algorithm for Channel Decoding

Convolutional encoding with the Viterbi decoding algorithm is widely used in telecommunication systems for error control coding. The Viterbi algorithm requires a computationally intensive routine with many add-compare-select (ACS) iterations. The C54x can perform fast ACS operations because of dedicated hardware and instructions that support the Viterbi algorithm on chip. This implementation allows the channel decoder and the equalizer in communication systems to be used efficiently.

In the global system for mobile communications (GSM) cellular radio, the polynomials in Equation 4–5 are used for convolutional encoding.

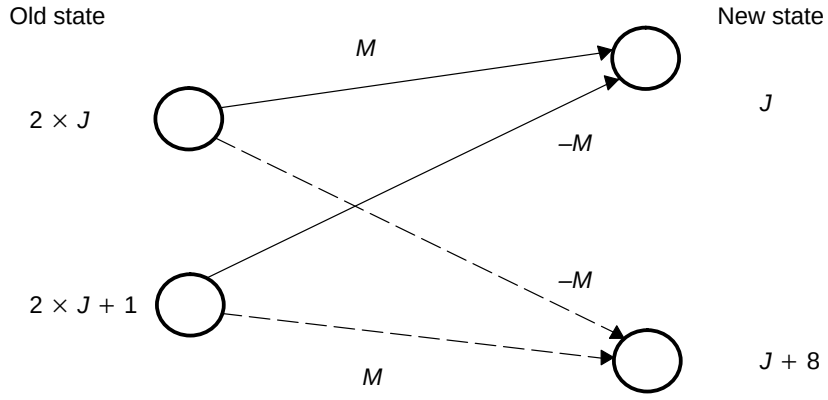
Equation 4–5. Polynomials for Convolutional Encoding

$$G1(D) = 1 + D^3 + D^4 \qquad G2(D) = 1 + D + D^3 + D^4$$

This convolutional encoding can be represented in a trellis diagram, which forms a butterfly structure as shown in Figure 4–2. The trellis diagram illustrates all possible transformations of convolutional encoding from one state to another, along with their corresponding path states. There are 16 states, or eight butterflies, in every symbol time interval. Two branches are input to each state. Decoding the convolutional code involves finding the optimal path by iteratively selecting possible paths in each state through a predetermined number of symbol time intervals. Two path metrics are calculated by adding branch metrics to two old-state path metrics and the path metric (J) for the new state is selected from these two path metrics.

Equation 4–6 defines a branch metric.

Figure 4–2. Butterfly Structure of the Trellis Diagram



Equation 4–6. Branch Metric

$$M = SD(2\times i) \times B(J,0) + SD(2\times i+1) \times B(J,1)$$

SD(2×*i*) is the first symbol that represents a soft-decision input and *SD*(2×*i*+1) is the second symbol. *B*(*J*,0) and *B*(*J*,1) correspond to the code generated by the convolutional encoder as shown in Table 4–1.

Table 4–1. Code Generated by the Convolutional Encoder

J	***B*(*J*,0)**	***B*(*J*,1)**
0	1	1
1	–1	–1
2	1	1
3	–1	–1
4	1	–1
5	–1	1
6	1	–1
7	–1	1

The C54x can compute a butterfly quickly by setting the ALU to dual 16-bit mode. To determine the new path metric (J), two possible path metrics from 2×*J* and 2×*J*+1 are calculated in parallel with branch metrics (*M* and –*M*) using the DADST instruction. The path metrics are compared by the CMPS instruction.

To calculate the new path metric (*J*+8), the DSADT instruction calculates two possible path metrics using branch metrics and old path metrics stored in the upper half and lower half of the accumulator. The CMPS instruction determines the new path metric.

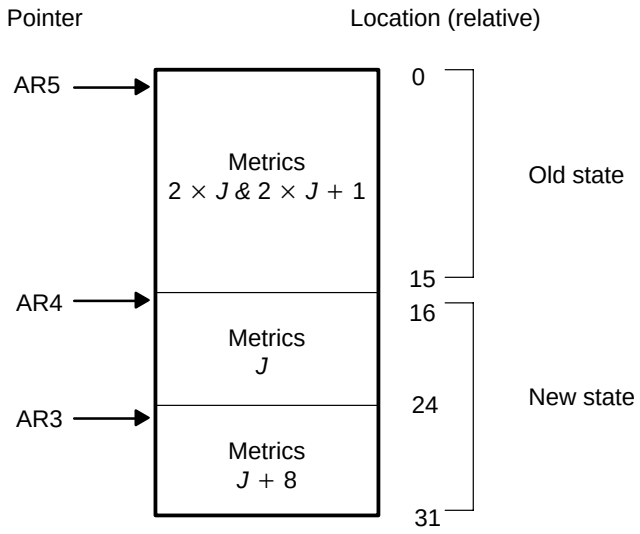
The CMPS instruction compares the upper word and the lower word of the accumulator and stores the larger value in memory. The 16-bit transition register (TRN) is updated with every comparison so you can track the selected path metric. The TRN contents must be stored in memory locations after processing each symbol time interval. The back-track routine uses the information in memory locations to find the optimal path.

Example 4–2 shows the Viterbi butterfly macro. A branch metric value is stored in *T* before calling the macro. During every butterfly cycle, two macros prevent T from receiving opposite sign values of the branch metrics. Figure 4–3 illustrates pointer management and the storage scheme for the path metrics used in Example 4–2.

In one symbol time interval, eight butterflies are calculated for the next 16 new states. This operation repeats over a number of symbol time intervals. At the

end of the sequence of time intervals, the back-track routine is performed to find the optimal path out of the 16 paths calculated. This path represents the bit sequence to be decoded.

Figure 4–3. Pointer Management and Storage Scheme for Path Metrics



Example 4–2. Viterbi Operator for Channel Coding

```
VITRBF .MACRO     ;
       DADST  *AR5,A  ;A = OLD_M(2*J)+T//OLD_(2*J+1)–T
       DSADT  *AR5+,B    ;B = OLD_M(2*J)–T//OLD_(2*J+1)+T
       CMPS   A,*AR4+    ;NEW_M(J) = MAX(A_HIGH,A_LOW)
              ;TRN<<1, TRN(0,0) = TC
       CMPS   B,*AR3+    ;NEW_M(J+8) = MAX(B_HIGH,B_LOW)
              ;TRN<<1, TRN(0,) = TC
       .ENDM
VITRBR .MACRO     ;
       DSADT  *AR5,A  ;A = OLD_M(2*J)–T//OLD_(2*J+1)+T
       DADST  *AR5+,B    ;B = OLD_M(2*J)+T//OLD_(2*J+1)–T
       CMPS   A,*AR4+    ;NEW_M(J) = MAX(A_HIGH,A_LOW)
              ;TRN<<1, TRN(0,0) = TC
       CMPS   B,*AR3+    ;NEW_M(J+8) = MAX(B_HIGH,B_LOW)
              ;TRN<<1, TRN(0,) = TC
       .ENDM
```

Chapter 5

TI C54x DSPLIB

The TI C54x DSPLIB is an optimized DSP function library for C programmers on TMS320C54x™ (C54x) DSP devices. It includes over 50 C-callable assembly-optimized general-purpose signal processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can shorten significantly your DSP application development time.

The TI DSPLIB includes commonly used DSP routines. **Source code is provided to allow you to modify the functions to match your specific needs and is shipped as part of the C54x Code Composer Studio product under the c:\ti\C5400\dsplib\54x_src directory.**

Full documentation on C54x DSPLIB can be found in the *TMS320C54x DSP Library Programmer's Reference* (SPRU518).

Topic **Page**

5.1 Features and Benefits

- ❏ Hand-coded assembly optimized routines
- ❏ C-callable routines fully compatible with the C54x DSP compiler
- ❏ Fractional Q15-format operands supported
- ❏ Complete set of examples on usage provided
- ❏ Benchmarks (cycles and code size) provided
- ❏ Tested against Matlab™ scripts

5.2 DSPLIB Data Types

DSPLIB functions generally operate on Q15-fractional data type elements:

- ❏ Q.15 (DATA): A Q.15 operand is represented by a *short* data type (16 bit) that is predefined as *DATA*, in the *dsplib.h* header file.

Certain DSPLIB functions use the following data type elements:

- ❏ Q.31 (LDATA): A Q.31 operand is represented by a *long* data type (32 bit) that is predefined as L*DATA*, in the *dsplib.h* header file.
- ❏ Q.3.12: Contains 3 integer bits and 12 fractional bits.

5.3 DSPLIB Arguments

DSPLIB functions typically operate over vector operands for greater efficiency. Though these routines can be used to process short arrays or scalars (unless a minimum size requirement is noted), the execution times will be longer in those cases.

- ❏ **Vector stride is always equal 1:** vector operands are composed of vector elements held in consecutive memory locations (vector stride equal to 1).
- ❏ **Complex elements** are assumed to be stored in a Real-Imaginary (Re-Im) format.
- ❏ **In-place computation is allowed (unless specifically noted):** Source operand can be equal to destination operand to conserve memory.

5.4 Calling a DSPLIB Function from C

In addition to installing the DSPLIB software, to include a DSPLIB function in your code you have to:

- ❏ Include the *dsplib.h* include file
- ❏ Link your code with the DSPLIB object code library, *54xdsp.lib*.
- ❏ Use a correct linker command file describing the memory configuration available in your C54x DSP board.

For example, the following code contains a call to the recip16 and q15tofl routines in DSPLIB:

```
#include ”dsplib.h”

DATA x[3] = { 12398 , 23167, 564};

DATA  r[NX];
DATA  rexp[NX];
float rf1[NX];
float rf2[NX];

void main()
{
        short i;

        for (i=0;i<NX;i++)
         {
               r[i] =0;
               rexp[i] = 0;
         }

        recip16(x, r, rexp, NX);
        q15tofl(r, rf1, NX);

        for (i=0; i<NX; i++)
         {
               rf2[i] = (float)rexp[i] * rf1[i];
         }

        return;
}
```

In this example, the q15tofl DSPLIB function is used to convert Q15 fractional values to floating-point fractional values. However, in many applications, your data is always maintained in Q15 format so that the conversion between floating point and Q15 is not required.

5.5 Calling a DSPLIB Function from Assembly Language Source Code

The DSPLIB functions were written to be used from C. Calling the functions from assembly language source code is possible as long as the calling-function conforms with the C54x DSP C compiler calling conventions. Refer to the *TMS320C54x Optimizing C Compiler User's Guide* (SPRU103), if a more in-depth explanation is required.

Realize that the DSPLIB is not an optimal solution for assembly-only programmers. Even though DSPLIB functions can be invoked from an assembly program, the resulting execution times and code size may not be optimal due to unnecessary C-calling overhead.

5.6 Where to Find Sample Code

You can find examples on how to use every single function in DSPLIB, in the *examples* subdirectory. This subdirectory contains one subdirectory for each function. For example, the *c:\ti\cstools\dsplib\examples* directory contains the following files:

- *araw_t.c*: main driver for testing the DSPLIB acorr (raw) function.
- *test.h*: contains input data(a) and expected output data(yraw) for the acorr (raw) function as. This test.h file is generated by using Matlab scripts.
- *test.c*: contains function used to compare the output of araw function with the expected output data.
- *ftest.c*: contains function used to compare two arrays of float data types.
- *ltest.c*: contains function used to compare two arrays of long data types.
- *54x.cmd*: an example of a linker command you can use for this function.

5.7 DSPLIB Functions

DSPLIB provides functions in the following 8 functional catagories:

- ❑ Fast-Fourier Transforms (FFT)
- ❑ Filtering and convolution
- ❑ Adaptive filtering
- ❑ Correlation
- ❑ Math
- ❑ Trigonometric
- ❑ Miscellaneous
- ❑ Matrix

For specific DSPLIB function API descriptions, refer to the *TMS320C54x DSP Library Programmer's Reference* (SPRU518).