TMS320C55x DSP Programmer's Guide

Preliminary Draft

This document contains preliminary data current as of the publication date and is subject to change without notice.

SPRU376A
August 2001

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

This manual describes ways to optimize C and assembly code for the TMS320C55x™ DSPs and recommends ways to write TMS320C55x code for specific applications.

Notational Conventions

This document uses the following conventions.

❑ The device number TMS320C55x is often abbreviated as C55x.

❑ 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 usually 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.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x 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.

TMS320C55x Technical Overview (literature number SPRU393). This overview is an introduction to the TMS320C55x digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000 DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the mnemonic instructions individually. It also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the algebraic instructions individually. It also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x Optimizing C Compiler User's Guide (literature number SPRU281) describes the C55x C compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for TMS320C55x devices.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) 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 TMS320C55x devices.

TMS320C55x DSP Library Programmer's Reference (literature number SPRU422) describes the optimized DSP Function Library for C programmers on the TMS320C55x DSP.

The CPU, the registers, and the instruction sets are also described in online documentation contained in Code Composer Studio™.

Trademarks

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This chapter lists some of the key features of the TMS320C55x™ (C55x) DSP architecture and shows a recommended process for creating code that runs efficiently.

Topic **Page**

1.1 TMS320C55x Architecture

The TMS320C55x device is a fixed-point digital signal processor (DSP). The main block of the DSP is the central processing unit (CPU), which has the following characteristics:

- A unified program/data memory map. In program space, the map contains 16M bytes that are accessible at 24-bit addresses. In data space, the map contains 8M words that are accessible at 23-bit addresses.
- An input/output (I/O) space of 64K words for communication with peripherals.
- Software stacks that support 16-bit and 32-bit push and pop operations. You can use these stack for data storage and retreival. The CPU uses these stacks for automatic context saving (in response to a call or interrupt) and restoring (when returning to the calling or interrupted code sequence).
- A large number of data and address buses, to provide a high level of parallelism. One 32-bit data bus and one 24-bit address bus support instruction fetching. Three 16-bit data buses and three 24-bit address buses are used to transport data to the CPU. Two 16-bit data buses and two 24-bit address buses are used to transport data from the CPU.
- An instruction buffer and a separate fetch mechanism, so that instruction fetching is decoupled from other CPU activities.
- The following computation blocks: one 40-bit arithmetic logic unit (ALU), one 16-bit ALU, one 40-bit shifter, and two multiply-and-accumulate units (MACs). In a single cycle, each MAC can perform a 17-bit by 17-bit multiplication (fractional or integer) and a 40-bit addition or subtraction with optional 32-/40-bit saturation.
- An instruction pipeline that is protected. The pipeline protection mechanism inserts delay cycles as necessary to prevent read operations and write operations from happening out of the intended order.
- Data address generation units that support linear, circular, and bit-reverse addressing.
- Interrupt-control logic that can block (or mask) certain interrupts known as the maskable interrupts.
- A TMS320C54x-compatible mode to support code originally written for a TMS320C54x™ DSP.

1.2 Code Development Flow for Best Performance

The following flow chart shows how to achieve the best performance and code-generation efficiency from your code. After the chart, there is a table that describes the phases of the flow.

Figure 1–1. Code Development Flow

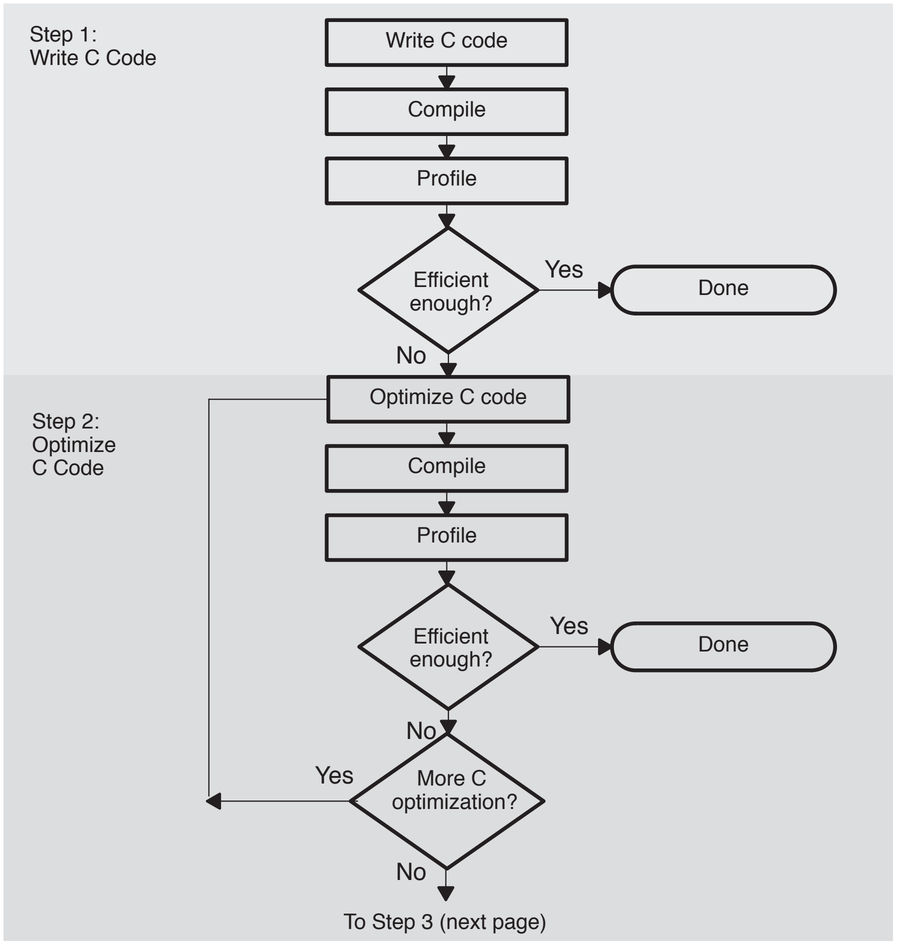
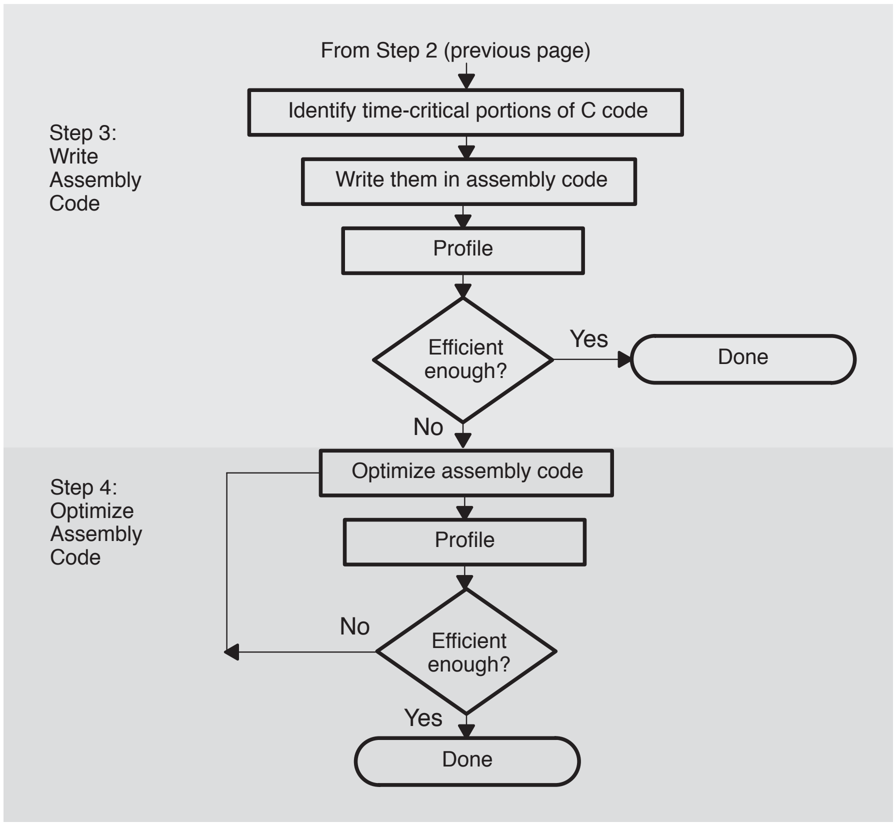


Figure 1–1. Code Development Flow (Continued)


From Step 2 (previous page)
Step 3: Write Assembly Code
Identify time-critical portions of C code
Write them in assembly code
Profile
Efficient enough?
Yes
Done
No
Step 4: Optimize Assembly Code
Optimize assembly code
Profile
No
Efficient enough?
Yes
Done

Step	Goal
1	**Write C Code:** You can develop your code in C using the ANSI-compliant C55x C compiler without any knowledge of the C55x DSP. Use Code Composer Studio™ to identify any inefficient areas that you might have in your C code. After making your code functional, you can improve its performance by selecting higher-level optimization compiler options. If your code is still not as efficient as you would like it to be, proceed to step 2.
2	**Optimize C Code:** Explore potential modifications to your C code to achieve better performance. Some of the techniques you can apply include (see Chapter 3): ❑ Use specific types (register, volatile, const). ❑ Modify the C code to better suit the C55x architecture. ❑ Use an ETSI intrinsic when applicable. ❑ Use C55x compiler intrinsics. After modifying your code, use the C55x profiling tools again, to check its performance. If your code is still not as efficient as you would like it to be, proceed to step 3.
3	**Write Assembly Code:** Identify the time-critical portions of your C code and rewrite them as C-callable assembly-language functions. Again, profile your code, and if it is still not as efficient as you would like it to be, proceed to step 4.
4	**Optimize Assembly Code:** After making your assembly code functional, try to optimize the assembly-language functions by using some of the techniques described in Chapter 4, *Optimizing Your Assembly Code*. The techniques include: ❑ Place instructions in parallel. ❑ Rewrite or reorganize code to avoid pipeline protection delays. ❑ Minimize stalls in instruction fetching.

Chapter 2

Tutorial

This tutorial walks you through the code development flow introduced in Chapter 1, and introduces you to basic concepts of TMS320C55x (C55x) DSP programming. It uses step-by-step instructions and code examples to show you how to use the software development tools integrated under Code Composer Studio (CCS).

Installing CCS before beginning the tutorial allows you to edit, build, and debug DSP target programs. For more information about CCS features, see the CCS Tutorial. You can access the CCS Tutorial within CCS by choosing Help→Tutorial.

The examples in this tutorial use instructions from the mnemonic instruction set, but the concepts apply equally for the algebraic instruction set.

Topic **Page**

2.1 Introduction

This tutorial presents a simple assembly code example that adds four numbers together (y = x0 + x3 + x1 + x2). This example helps you become familiar with the basics of C55x programming.

After completing the tutorial, you should know:

- The four common C55x addressing modes and when to use them.
- The basic C55x tools required to develop and test your software.

This tutorial does not replace the information presented in other C55x documentation and is not intended to cover all the topics required to program the C55x efficiently.

Refer to the related documentation listed in the preface of this book for more information about programming the C55x DSP. Much of this information has been consolidated as part of the C55x Code Composer Studio online help.

For your convenience, all the files required to run this example can be downloaded with the *TMS320C55x Programmer's Guide* (SPRU376) from http://www.ti.com/sc/docs/schome.htm. The examples in this chapter can be found in the 55xprgug_srccode\tutor directory.

2.2 Writing Assembly Code

Writing your assembly code involves the following steps:

- Allocate sections for code, constants, and variables.
- Initialize the processor mode.
- Set up addressing modes and add the following values: x0 + x1 + x2 + x3.

The following rules should be considered when writing C55x assembly code:

- Labels

  The first character of a label must be a letter or an underscore (_) followed by a letter, and must begin in the first column of the text file. Labels can contain up to 32 alphanumeric characters.

- Comments

  When preceded by a semicolon (;), a comment may begin in any column. When preceded by an asterisk (*), a comment must begin in the first column.

The final assembly code product of this tutorial is displayed in Example 2–1, Final Assembly Code of tutor.asm. This code performs the addition of the elements in vector x. Sections of this code are highlighted in the three steps used to create this example.

For more information about assembly syntax, see the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280).

Example 2–1. Final Assembly Code of tutor.asm

```
* Step 1: Section allocation
* ------
       .def x,y,init
x      .usect ”vars”,4         ; reserve 4 uninitalized 16-bit locations for x
y      .usect ”vars”,1         ; reserve 1 uninitialized 16-bit location for y

       .sect ”table”           ; create initialized section ”table” to
init   .int 1,2,3,4            ; contain initialization values for x

       .text                   ; create code section (default is .text)
       .def start              ; define label to the start of the code
start

* Step 2: Processor mode initialization
* ------
    BCLR  C54CM     ; set processor to ’55x native mode instead of
                    ; ’54x compatibility mode (reset value)
    BCLR  AR0LC     ; set AR0 register in linear mode
    BCLR  AR6LC     ; set AR6 register in linear mode

* Step 3a: Copy initialization values to vector x using indirect addressing
* -------
copy
    AMOV  #x, XAR0             ; XAR0 pointing to variable x
    AMOV  #init, XAR6          ; XAR6 pointing to initialization table

    MOV      *AR6+, *AR0+      ; copy starts from ”init” to ”x”
    MOV      *AR6+, *AR0+
    MOV      *AR6+, *AR0+
    MOV      *AR6, *AR0

* Step 3b: Add values of vector x elements using direct addressing
* -------
add
    AMOV  #x, XDP              ; XDP pointing to variable x
    .dp x                      ; and the assembler is notified

    MOV      @x, AC0
    ADD      @(x+3), AC0
    ADD      @(x+1), AC0
    ADD      @(x+2), AC0

* Step 3c. Write the result to y using absolute addressing
* -------
    MOV      AC0, *(#y)

end
    NOP
    B  end
```

2.2.1 Allocate Sections for Code, Constants, and Variables

The first step in writing this assembly code is to allocate memory space for the different sections of your program.

Sections are modules consisting of code, constants, or variables needed to successfully run your application. These modules are defined in the source file using assembler directives. The following basic assembler directives are used to create sections and initialize values in the example code.

- .sect *"section_name"* creates initialized name section for code/data. Initialized sections are sections defining their initial values.
- .usect *"section_name", size* creates uninitialized named section for data. Uninitialized sections declare only their size in 16-bit words, but do not define their initial values.
- .int *value* reserves a 16-bit word in memory and defines the initialization value
- .def *symbol* makes a symbol global, known to external files, and indicates that the symbol is defined in the current file. External files can access the symbol by using the .ref directive. A symbol can be a label or a variable.

As shown in Example 2–2 and Figure 2–1, the example file tutor.asm contains three sections:

- *vars*, containing five uninitialized memory locations
  - The first four are reserved for vector *x* (the input vector to add).
  - The last location, *y*, will be used to store the result of the addition.
- *table*, to hold the initialization values for *x*. The *init* label points to the beginning of section *table.*
- *.text*, which contains the assembly code

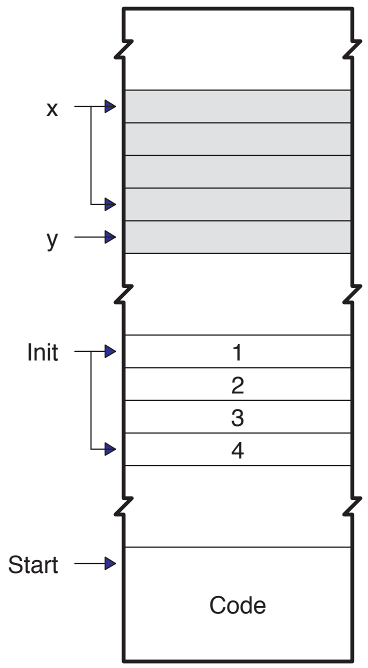
Example 2–2 shows the partial assembly code used for allocating sections.

Example 2–2. Partial Assembly Code of tutor.asm (Step 1)

```
* Step 1: Section allocation
* ------
    .def x, y, init
x   .usect “vars”, 4 ; reserve 4 uninitialized 16-bit locations for x
y   .usect “vars”, 1 ; reserve 1 uninitialized 16-bit location for y

    .sect “table”           ; create initialized section “table” to
init   .int 1, 2, 3, 4      ; contain initialization values for x

    .text            ; create code section (default is .text)
    .def start       ; define label to the start of the code
start
```

Note: The algebraic instructions code example for Partial Assembly Code of tutor.asm (Step 1) is shown in Example B–1 on page B-2.

Figure 2–1. Section Allocation



2.2.2 Processor Mode Initialization

The second step is to make sure the status registers (ST0_55, ST1_55, ST2_55, and ST3_55) are set to configure your processor. You will either need to set these values or use the default values. Default values are placed in the registers after processor reset. You can locate the default register values after reset in the *TMS320C55x DSP CPU Reference Guide* (SPRU371).

As shown in Example 2–3:

- ❑ The AR0 and AR6 registers are set to linear addressing (instead of circular addressing) using bit addressing mode to modify the status register bits.
- ❑ The processor has been set in C55x native mode instead of C54x-compatible mode.

Example 2–3. Partial Assembly Code of tutor.asm (Step 2)

```
* Step 2: Processor mode initialization
* ------
    BCLR  C54CM     ; set processor to ’55x native mode instead of
                    ; ’54x compatibility mode (reset value)
    BCLR  AR0LC     ; set AR0 register in linear mode
    BCLR  AR6LC     ; set AR6 register in linear mode
```

Note: The algebraic instructions code example for Partial Assembly Code of tutor.asm (Step 2) is shown in Example B–2 on page B-2.

2.2.3 Setting up Addressing Modes

Four of the most common C55x addressing modes are used in this code:

- ARn Indirect addressing (identified by *), in which you use auxiliary registers (ARx) as pointers.
- DP direct addressing (identified by @), which provides a positive offset addressing from a base address specified by the DP register. The offset is calculated by the assembler and defined by a 7-bit value embedded in the instruction.
- k23 absolute addressing (identified by #), which allows you to specify the entire 23-bit data address with a label.
- Bit addressing (identified by the bit instruction), which allows you to modify a single bit of a memory location or MMR register.

For further details on these addressing modes, refer to the *TMS320C55x DSP CPU Reference Guide* (SPRU371). Example 2–4 demonstrates the use of the addressing modes discussed in this section.

In Step 3a, initialization values from the *table* section are copied to vector *x* (the vector to perform the addition) using indirect addressing. Figure 2–2 illustrates the structure of the extended auxiliar registers (XAR*n*). The XAR*n* register is used only during register initialization. Subsequent operations use AR*n* because only the lower 16 bits are affected (AR*n* operations are restricted to a 64k main data page). AR6 is used to hold the address of *table*, and AR0 is used to hold the address of *x*.

In Step 3b, direct addressing is used to add the four values. Notice that the XDP register was initialized to point to variable *x.* The .dp assembler directive is used to define the value of XDP, so the correct offset can be computed by the assembler at compile time.

Finally, in Step 3c, the result was stored in the *y* vector using absolute addressing. Absolute addressing provides an easy way to access a memory location without having to make XDP changes, but at the expense of an increased code size.

Example 2–4. Partial Assembly Code of tutor.asm (Part3)

```
* Step 3a: Copy initialization values to vector x using indirect addressing
* -------
copy
    AMOV  #x, XAR0         ; XAR0 pointing to variable x
    AMOV  #init, XAR6      ; XAR6 pointing to initialization table

    MOV      *AR6+, *AR0+  ; copy starts from ”init” to ”x”
    MOV      *AR6+, *AR0+
    MOV      *AR6+, *AR0+
    MOV      *AR6, *AR0

* Step 3b: Add values of vector x elements using direct addressing
* -------
add
    AMOV  #x, XDP          ; XDP pointing to variable x
    .dp x                  ; and the assembler is notified

    MOV      @x, AC0
    ADD      @(x+3), AC0
    ADD      @(x+1), AC0
    ADD      @(x+2), AC0

* Step 3c: Write the result to y using absolute addressing
* -------
    MOV      AC0, *(#y)

end
    NOP
    B  end
```

Note: The algebraic instructions code example for Partial Assembly Code of tutor.asm (Part3) is shown in Example B–3 on page B-3.

Figure 2–2. Extended Auxiliary Registers Structure (XARn)

	22–16	15–0
XAR*n*	AR*n*H	AR*n*

Note: ARnH (upper 7 bits) specifies the 7-bit main data page. ARn (16-bit register) specifies a 16-bit offset to the 7-bit main data page to form a 23-bit address.

2.3 Understanding the Linking Process

The linker (lnk55.exe) assigns the final addresses to your code and data sections. This is necessary for your code to execute.

The file that instructs the linker to assign the addresses is called the linker command file (tutor.cmd) and is shown in Example 2–5. The linker command file syntax is covered in detail in the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280).

- ❑ All addresses and lengths given in the linker command file uses byte addresses and byte lengths. This is in contrast to a TMS320C54x linker command file that uses 16-bit word addresses and word lengths.
- ❑ The MEMORY linker directive declares all the physical memory available in your system (For example, a DARAM memory block at location 0x100 of length 0x8000 bytes). Memory blocks cannot overlap.
- ❑ The SECTIONS linker directive lists all the sections contained in your input files and where you want the linker to allocate them.

When you build your project in Section 2.4, this code produces two files, tutor.out and a tutor.map. Review the test.map file, Example 2–6, to verify the addresses for x, y, and table. Notice that the linker reports byte addresses for program labels such as *start* and *.text*, and 16-bit word addresses for data labels like *x*, *y*, and *table*. The C55x DSP uses byte addressing to acces variable length instructions. Instructions can be 1-6 bytes long.

Example 2–5. Linker command file (tutor.cmd)

```
MEMORY        /* byte address, byte len */
{
   DARAM: org= 000100h, len = 8000h
   SARAM: org= 010000h, len = 8000h
}


SECTIONS      /* byte address, byte len */
{
   vars :> DARAM
   table: > SARAM
   .text:> SARAM
}
```

Example 2–6. Linker map file (test.map)

```
********************************************************************************
TMS320C55xx COFF Linker
********************************************************************************
>> Linked Mon Feb 14 14:52:21 2000

OUTPUT FILE NAME:   <tutor.out>
ENTRY POINT SYMBOL: "start"  address: 00010008


MEMORY CONFIGURATION

        name      org (bytes)  len (bytes)  used (bytes)  attributes  fill
        ----      -----------  -----------  ------------  ----------  ----
         DARAM     00000100     000008000    0000000a      RWIX
         SARAM     00010000     000008000    00000040      RWIX


SECTION ALLOCATION MAP

output                                                          attributes/
section   page  orgn(bytes) orgn(words) len(bytes) len(words)   input sections
--------  ----  ----------- ----------- ---------- ----------   --------------
vars        0               00000080               00000005     UNINITIALIZED
                               00000080               00000005   test.obj (vars)

table       0               00008000               00000004
                               00008000               00000004   test.obj
(table)

.text       0  00010008                00000038
                  00010008                00000037               test.obj
(.text)
                  0001003f                00000001               --HOLE-- [fill
= 2020]

.data       0               00000000               00000000     UNINITIALIZED
                               00000000               00000000   test.obj
(.data)

.bss        0               00000000               00000000     UNINITIALIZED
                               00000000               00000000   test.obj (.bss)
```

Example 2–6. Linker map file (test.map), (Continued)

```
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name

abs. value/
byte addr   word addr   name
---------   ---------   ----
            00000000    .bss
            00000000    .data
00010008                .text
            00000000    ___bss__
            00000000    ___data__
            00000000    ___edata__
            00000000    ___end__
00010040                ___etext__
00010008                ___text__
            00000000    edata
            00000000    end
00010040                etext
            00008000    init
00010008                start
            00000080    x
            00000084    y


GLOBAL SYMBOLS: SORTED BY Symbol Address

abs. value/
byte addr   word addr   name
---------   ---------   ----
            00000000    ___end__
            00000000    ___edata__
            00000000    end
            00000000    edata
            00000000    ___data__
            00000000    .data
            00000000    .bss
            00000000    ___bss__
            00000080    x
            00000084    y
            00008000    init
00010008                start
00010008                .text
00010008                ___text__
00010040                ___etext__
00010040                etext

[16 symbols]
```

2.4 Building Your Program

At this point, you should have already successfully installed CCS and selected the C55x Simulator as the CCS configuration driver to use. You can select the configuration driver to be used in the CCS setup.

Before building your program, you must set up your work environment and create a .pjt file. Setting up your work environment involves the following tasks:

- ❑ Creating a project
- ❑ Adding files to the work space
- ❑ Modifying the build options
- ❑ Building your program

2.4.1 Creating a Project

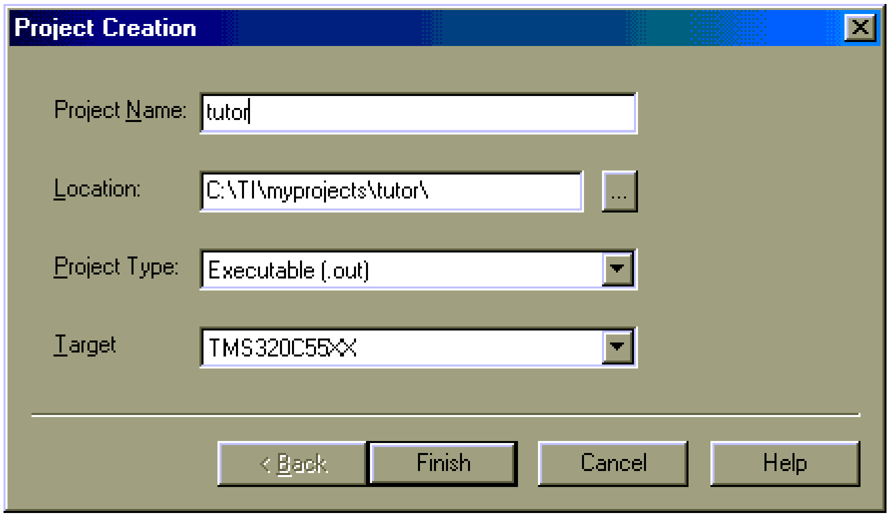
Create a new project called tutor.pjt.

1) From the Project menu, choose New and enter the values shown in Figure 2–3.

2) Select Finish.

You have now created a project named tutor.pjt and saved it in the new c:\ti\myprojects\tutor folder.

Figure 2–3. Project Creation Dialog Box


Project Creation

Project Name: tutor

Location: C:\TI\myprojects\tutor\

Project Type: Executable (.out)

Target TMS320C55XX

2.4.2 Adding Files to the Workspace

Copy the tutorial files (tutor.asm and tutor.cmd) to the *tutor* project directory.

1) Navigate to the directory where the tutorial files are located (the 55xprgug_srccode\tutor directory) and copy them into the c:\ti\myprojects\tutor directory. As an alternative, you can create your own source files by choosing File→New→Source File and typing the source code from the examples in this book.

2) Add the two files to the tutor.pjt project. Highlight tutor.pjt, right-click the mouse, select Add Files, browse for the tutor.asm file, select it, and click Open, as shown in Figure 2–4. Do the same for tutor.cmd, as shown in Figure 2–5.

Figure 2–4. Add tutor.asm to Project

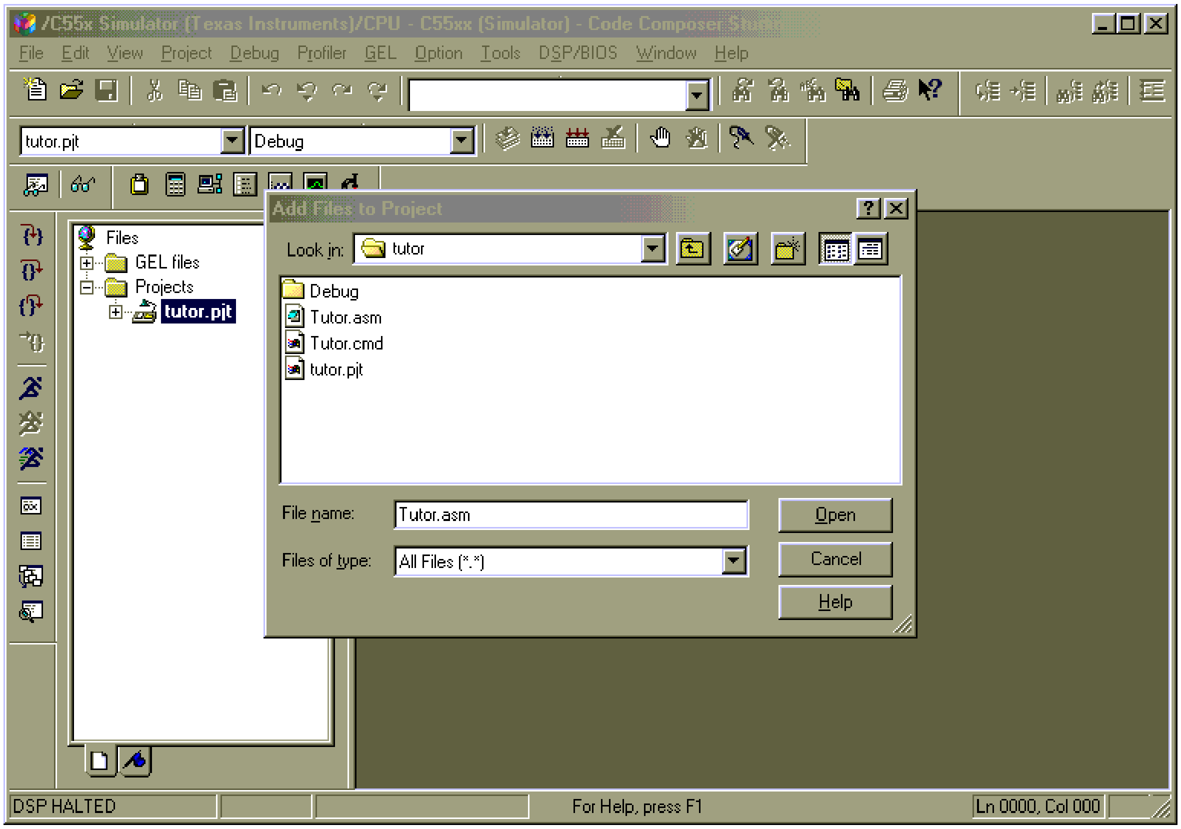
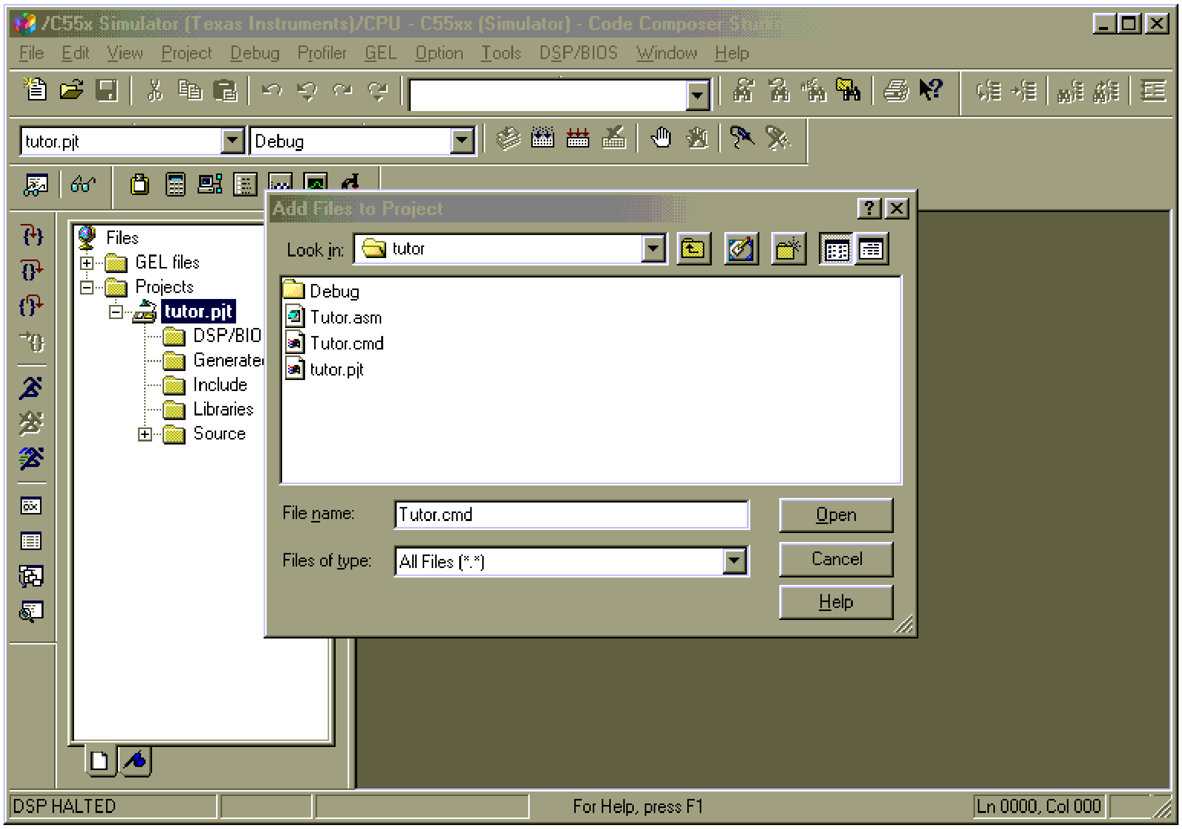


Figure 2–5. Add tutor.cmd to Project



File name: Tutor.cmd

Files of type: All Files (*.*)

2.4.3 Modifying Build Options

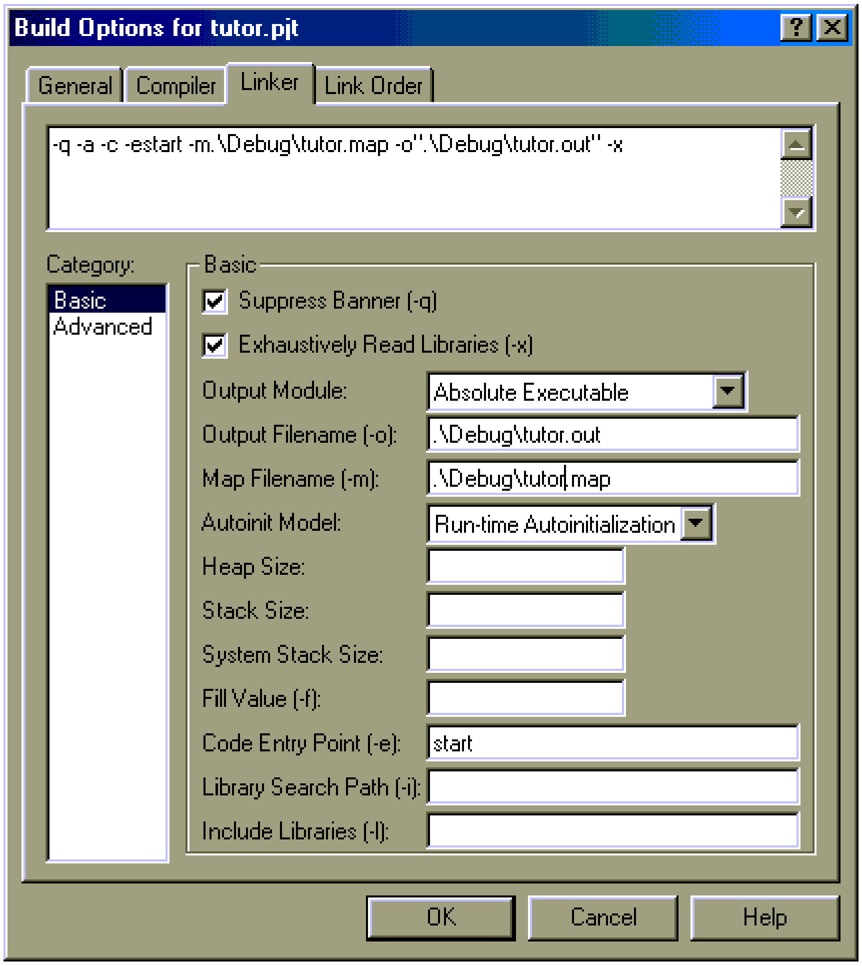
Modify the Linker options.

1) From the Project menu, choose Build Options.

2) Select the Linker tab and enter fields as shown in Figure 2–6.

3) Click OK when finished.

Figure 2–6. Build Options Dialog Box



```
-q -a -c -estart -m.\Debug\tutor.map -o".\Debug\tutor.out" -x
```

[✓] Suppress Banner (-q)
[✓] Exhaustively Read Libraries (-x)

Output Module:	Absolute Executable
Output Filename (-o):	.\Debug\tutor.out
Map Filename (-m):	.\Debug\tutor.map
Autoinit Model:	Run-time Autoinitialization
Heap Size:	
Stack Size:	
System Stack Size:	
Fill Value (-f):	
Code Entry Point (-e):	start
Library Search Path (-i):	
Include Libraries (-l):	

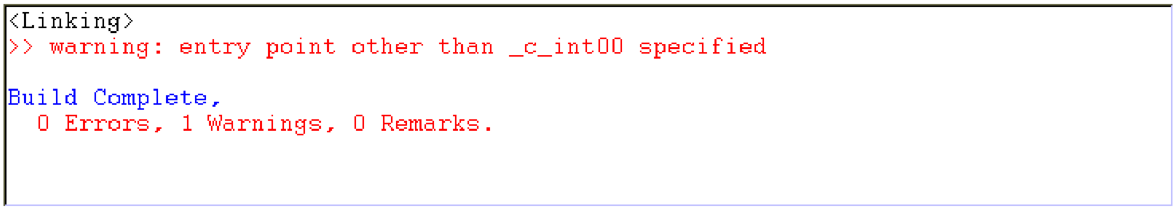
2.4.4 Building the Program

From the Project menu, choose Rebuild All. After the Rebuild process completes, the screen shown in Figure 2–7 should display.

When you build your project, CCS compiles, assembles, and links your code in one step. The assembler reads the assembly source file and converts C55x instructions to their corresponding binary encoding. The result of the assembly processes is an object file, *tutor.obj*, in industry standard COFF binary format. The object file contains all of your code and variables, but the addresses for the different sections of code are not assigned. This assignment takes place during the linking process.

Because there is no C code in your project, no compiler options were used.

Figure 2–7. Rebuild Complete Screen



```
<Linking>
>> warning: entry point other than _c_int00 specified

Build Complete,
  0 Errors, 1 Warnings, 0 Remarks.
```

2.5 Testing Your Code

To test your code, inspect its execution using the C55x Simulator.

Load tutor.out

1) From the File menu, choose Load program.
2) Navigate to and select tutor.out (in the \debug directory), then choose Open.

CCS now displays the tutor.asm source code at the beginning of the start label because of the entry symbol defined in the linker command file (-e start). Otherwise, it would have shown the location pointed to by the reset vector.

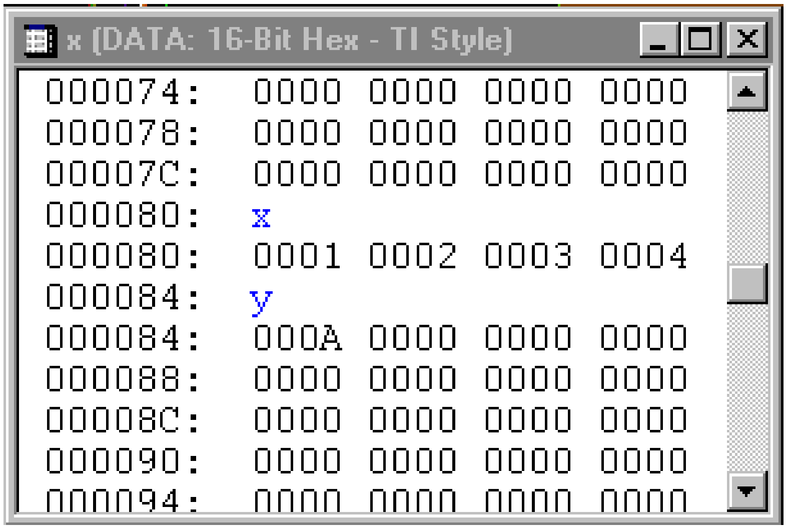
Display arrays *x, y,* and *init* by setting Memory Window options

1) From the View menu, choose Memory.
2) In the Title field, type x.
3) In the Address field, type *x.*
4) Repeat 1–3 for *y.*
5) Display the *init* array by selecting View→ Memory.
6) In the Title field, type *Table*.
7) In the Address field, type *init*.
8) Display AC0 by selecting View→CPU Registers→CPU Registers.

The labels *x*, *y*, and *init* are visible to the simulator (using View→ Memory) because they were exported as symbols (using the .def directive in tutor.asm). The -g option was used to enable assembly source debugging.

Now, single-step through the code to the *end* label by selecting Debug→Step Into. Examine the X Memory window to verify that the table values populate *x* and that *y* gets the value 0xa (1 + 2 + 3 + 4 = 10 = 0xa), as shown in Example 2–7.

Example 2–7. x Memory Window



```
000074:   0000 0000 0000 0000
000078:   0000 0000 0000 0000
00007C:   0000 0000 0000 0000
000080:   x
000080:   0001 0002 0003 0004
000084:   y
000084:   000A 0000 0000 0000
000088:   0000 0000 0000 0000
00008C:   0000 0000 0000 0000
000090:   0000 0000 0000 0000
000094:   0000 0000 0000 0000
```

2.6 Benchmarking Your Code

After verifying the correct functional operation of your code, you can use CCS to calculate the number of cycles your code takes to execute.

Reload your code

From the File menu, choose Reload Program.

Enable clock for profiling

1) From the Profiler menu, choose Enable Clock.

2) From the Profiler menu, choose View Clock.

Set breakpoints

1) Select the *tutor.asm* window.

2) Set one breakpoint at the beginning of the code you want to benchmark (first instruction after *start*): Right-click on the instruction next to the *copy* label and choose Toggle Breakpoint.

3) Set one breakpoint marking the end: Right-click on the instruction next to the *end* label and choose Toggle Breakpoint.

Benchmark your code

1) Run to the first breakpoint by selecting Debug→ Run.

2) Double-click in the Clock Window to clear the cycle count.

3) Run to the second breakpoint by selecting Debug→ Run.

4) The Clock Window displays the number of cycles the code took to execute between the breakpoints, which was approximately 17.

Chapter 3

Optimizing C Code

You can maximize the performance of your C code by using certain compiler options, C code transformations, and compiler intrinsics. This chapter discusses features of the C language relevant to compilation on the TMS320C55x (C55x) DSP, performance-enhancing options for the compiler, and C55x-specific code transformations that improve C code performance. All assembly language examples were generated for the large memory model via the `−ml` compiler option.

Topic **Page**

3.1 Introduction to Writing C/C++ Code for a C55x DSP

This section describes some general issues to keep in mind when writing C/C++ code for the TMS320C55x (C55x) architecture (or any DSP architecture). Keep this information in mind when working in Step 1 of code development as described in Chapter 1. Refer to *TMS320C55x Optimizing C/C++ Compiler User's Guide* (SPRU281) for additional language issues.

3.1.1 Tips on Data Types

Give careful consideration to the data type size when writing your code. The C55x compiler defines a size for each C data type (signed and unsigned):

`char`	16 bits
`short`	16 bits
`int`	16 bits
`long`	32 bits
`long long`	40 bits
`float`	32 bits
`double`	64 bits

Floating point values are in the IEEE format. Based on the size of each data type, follow these guidelines when writing your code:

- ❑ Avoid code that assumes that `int` and `long` types are the same size.
- ❑ Use the `int` data type for fixed-point arithmetic (especially multiplication) whenever possible. Using type `long` for multiplication operands will result in calls to a run-time library routine.
- ❑ Use `int` or `unsigned int` types rather than `long` for loop counters. The C55x has mechanisms for efficient hardware loops, but hardware loop counters are only 16 bits wide.
- ❑ Avoid code that assumes `char` is 8 bits or `long long` is 64 bits.

When writing code to be used on multiple DSP targets, it may be wise to define "generic" types for the standard C types. For example, one could use the types `Int16` and `Int32` for a 16 bit integer type and 32 bit integer type respectively. When compiling for the C55x DSP, these types would be type defined to `int` and `long`, respectively.

In general it is best to use the type `int` *for* loop index variables and other integer variables where the number of bits is unimportant as `int` typically represents the most efficient integer type for the target to manipulate, regardless of architecture.

3.1.2 How to Write Multiplication Expressions Correctly in C Code

Writing multiplication expressions in C code so that they are both correct and efficient can be confusing, especially when technically illegal expressions can, in some circumstances, generate the code you wanted in the first place. This section will help you choose the correct expression for your algorithm.

The correct expression for a 16x16–>32 multiplication on a C55x DSP is:

```
long res = (long)(int)src1 * (long)(int)src2;
```

According to the C arithmetic rules,this is actually a 32x32–>32 multiplication, but the compiler will notice that each operand fits in 16 bits, so it will issue an efficient single-instruction multiplication.

A 16-bit multiplication with a 32-bit result is an operation which does not directly exist in the C language, but does exist on C55x hardware, and is vital for multiply-and-accumulate (MAC)-like algorithm performance.

Example 3–1 shows two incorrect ways and a correct way to write such a multiplication in C code.

Example 3–1. Generating a 16x16–>32 Multiply

```
long mult(int a, int b)
{
    long result;
    /* incorrect */
    result = a * b;
    /* incorrect */
    result = (long)(a * b);
    /* correct */
    result = (long)a * b;
    return result;
}
```

Note that the same rules also apply for other C arithmetic operators. For example, if you want to add two 16-bit numbers and get a full 32 bit result, the correct syntax is:

```
(long) res = (long)(int)src1 + (long)(int)src;
```

3.1.3 Memory Dependences

To maximize the efficiency or your code, the C55x compiler reorders instructions to minimize pipeline stalls, puts certain assembly instructions in parallel, and generates dual multiply-and-accumulate (dual-MAC) instructions. These transformations require the compiler to determine the relationships, or dependences, between instructions. Dependence means that one instruction must occur before another. For example, a variable may need to be loaded from memory before it can be used. Because only independent instructions can be scheduled in parallel or reordered, dependences inhibit parallelism and code movement. If the compiler cannot prove that two instructions are independent, it must assume that instructions must remain in the order they originally appeared and must not be scheduled in parallel.

Often it is difficult for the compiler to determine whether instructions that access memory are independent. The following techniques help the compiler determine which instructions are independent:

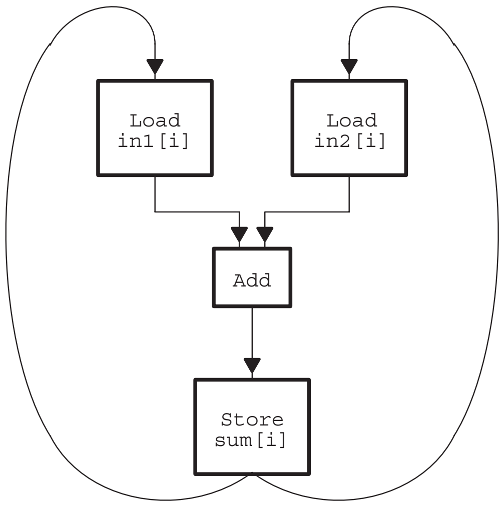
- ❑ Use the `restrict` keyword to indicate that a pointer is the only pointer than can point to a particular object in the scope in which the pointer is declared.
- ❑ Use the `–pm` option which gives the compiler global access to the whole program and allows it to be more aggressive in ruling out dependences.

To illustrate the concept of memory dependences, it is helpful to look at the algorithm code in a dependence graph. Example 3–2 shows code for a simple vector sum. Figure 3–1 shows a simplified dependence graph for that piece of code.

Example 3–2. C Code for Vector Sum

```
void vecsum(int *sum, short *in1, short *in2, int N)
{
   int i;
   for (i = 0; i < N; i++)
      sum[i] = in1[i] + in2[i];
}
```

Figure 3–1. Dependence Graph for Vector Sum



The dependence graph in Figure 3–1 shows that:

- ❑ The paths from the store of `sum[i]` back to the loads of `in1[i]` and `in2[i]` indicate that writing to `sum` may have an effect on the memory pointed to by either `in1` or `in2`.
- ❑ A read from `in1` or `in2` cannot begin until the write to `sum` finishes, which creates an aliasing problem. Aliasing occurs when two pointers can point to the same memory location. For example, if `vecsum()` is called in a program with the following statements, `in1` and `sum` alias each other because they both point to the same memory location:

```
short a[10], b[10];
vecsum(a, a, b, 10);
```

To help the compiler resolve memory dependences, you can qualify a pointer or array with the `restrict` keyword. Its use represents a guarantee by the programmer that within the scope of the pointer declaration, the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the behavior of the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In the declaration of the vector sum function you can use the `restrict` keyword to tell the compiler that `sum` is the only pointer that points to that object:

```
void vecsum(int * restrict sum, int *in1, int *in2, int N)
```

(Likewise, you could add `restrict` to `in1` and `in2` as well.) The next piece of code shows how to use `restrict` with an array function parameter instead of a pointer:

```
void vecsum(int sum[restrict], int *in1, int *in2, int N)
```

Caution must be exercised when using restrict. Consider this call of `vecsum()` (with the `sum` parameter qualified by `restrict`):

```
vecsum(a, a, b, 10);
```

Undefined behavior would result because `sum` and `in1` would point to the same object, which violates `sum`'s declaration as `restrict`.

3.1.4 Analyzing C Code Performance

Use the following techniques to analyze the performance of specific code regions:

❑ Use the `clock()` and `printf()` functions in C/C++ code to time and display the performance of specific code regions. You can use the stand-alone simulator (load55) for this purpose. Remember to subtract out the overhead time of calling the `clock()` function.

❑ Enable the clock and use profile points and the RUN command in the Code Composer Studio debugger to track the number of CPU clock cycles consumed by a particular section of code.

❑ Put each loop into a separate file that can be rewritten, recompiled, and run with the stand-alone simulator (load55).The critical performance areas in your code are most often loops.

As you use the techniques described in this chapter to optimize your C/C++ code, you can then evaluate the performance results by running the code and looking at the instructions generated by the compiler. More detail on performance analysis can be found in section 3.3.

3.2 Compiling the C/C++ Code

The C55x compiler offers high-level language support by transforming your C/C++ code into assembly language source code. The compiler tools include a shell program (cl55), which you use to compile, optimize, assemble, and link programs in a single step. To invoke the compiler shell, enter:

cl55 [*options*] [*filenames*] [**–z** [*linker options*] [*object files*]]

For a complete description of the C/C++ compiler and the options discussed in this section, see the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).

3.2.1 Compiler Options

Options control the operation of the compiler. This section introduces you to the recommended options for performance, information gathering, and code size.

First make note of the options to avoid using on performance critical code.The options described in Table 3–1 are intended for debugging, and could potentially *decrease* performance and increase code size.

Table 3–1. Compiler Options to Avoid on Performance-Critical Code

Option	Description
`–g`, `–s`, `–ss`, `–gp`	These options are intended for debugging and can limit the amount of optimization across C statements leading to larger code size and slower execution.
`–o1`, `–o0`	Always use `–o2/–o3` to maximize compiler analysis and optimization
`–mr`	Prevents generation of hardware loops to reduce context save/restore for interrupts. As hardware loops greatly improve performance of loop code, avoid this option on performance critical code.

The options in Table 3–2 can be used to improve performance. The options `–o3`, `–pm`, `–mb`, `–oi50`, and `–op2` are recommended for maximum performance.

Table 3–2. Compiler Options for Performance

Option	Description
`–o3`	Represents the highest level of optimization available. Various loop optimizations are performed, and various file-level characteristics are also used to improve performance.
`–pm`	Combines source files to perform program-level optimization by allowing the compiler visibility to the entire application source.
`–oi<size>`	Enables inlining of functions based on a maximum size. (Enabled with `–o3`.) Size here is determined internally by the optimizer and does not correspond to bytes or any other known standard unit. Use a `–onx` option to check sizes of individual functions.
`–mb`	Asserts to the compiler that all data is on-chip. This option is used to enable the compiler to generate dual-MAC. See section 3.4.2.2 for more details.
`–op2`	When used with `–pm`, this option allows the compiler to assume that the program being compiled does not contain any functions or variables called or modified from outside the current file. The compiler is free to remove any functions or variables that are unused in the current file.
`–mn`	Re-enables optimizations disabled when using `–g` option (symbolic debugging). Use this option when it is necessary to debug optimized code.

The options described in Table 3–3, can be used to improve code size with a possible degradation in performance.

Table 3–3. Compiler Options That May Degrade Performance and Improve Code Size

Option	Description
`–ms`	Encourages the compiler to optimize for code space. (Default is to optimize for performance.)
`–oi0`	Disables all automatic size-controlled inlining enabled by `–o3`. User specified inlining of functions is still allowed.

The options described in Table 3–4 provide information to the programmer. Some of them may negatively affect performance and/or code size.

Table 3–4. Compiler Options for Information

Option	Description
`–k`	The assembly file is not deleted. This allows you to inspect the generated code. This option has no impact on performance or code size.
`–s/–ss`	Interlists optimizer comments/C source in the assembly. The `–s` option may show minor performance degradation. The `–ss` option may show more severe performance degradation.
`–mg`	Generate algebraic assembly code. (Default is mnemonic.) There is no performance or code size impact.
`–on`*x*	When *x* is 1, the optimizer generates an information file (.nfo filename extension). When *x* is 2, a more verbose information file is generated. There is no performance or code size impact.

3.2.2 Performing Program-Level Optimization (`–pm` Option)

You can specify program-level optimization by using the `–pm` option with the `–o3` option. With program-level optimization, all your source files are compiled into one intermediate file giving the compiler complete program view during compilation. Because the compiler has access to the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- ❑ If the number of iterations of a loop is determined by a value passed into the function and the compiler can determine what the value is from the calling function, the compiler will have more information about the number of iterations of the loop, resulting in more efficient loop code.
- ❑ If a particular argument to a function always has the same value, the compiler replaces the occurrences of the argument in the function with the value.
- ❑ If a return value of a function is never used, the compiler deletes the return code in the function.
- ❑ If a function is not called, directly or indirectly, the compiler removes the code in the function.

Program-level optimization increases compilation time because the compiler performs more complex optimizations on a larger amount of code. For this reason you may not want to use this option for every build of large programs.

Example 3–3 and Example 3–4 show the content of two files. One file contains the source for the `main` function and the second file contains source for a small function called `sum`.

Example 3–3. Main Function File

```
extern int sum(const int *a, int n);
const int a[10] = {1,2,3,4,5,6,7,8,9,10};
const int b[10] = {11,12,13,14,15,16,17,18,19,20};
int sum1, sum2;
int main(void)
{
   sum1 = sum(a,10);
   sum2 = sum(b,10);
   return(0);
}
```

Example 3–4. Sum Function File

```
int sum(const int *a, int n)
{
   int total = 0;
   int i;
   for(i=0; i<n; i++)
   {
      total += a[i];
   }
   return total;
}
```

When this code is compiled with `–o3` and `–pm` options, the optimizer has enough information about the calls to `sum` to determine that the same loop count is used for both calls. It therefore eliminates the argument `n` from the call to the function and explicitly uses the count in the repeat single instruction as shown in Example 3–5.

Example 3–5. Assembly Code Generated With `–o3` *and* `–pm` *Options*

```
_sum:
;** Parameter deleted n == 9u
        MOV #0, T0 ; |3|
        RPT #9
            ADD *AR0+, T0, T0

        return    ; |11|
_main:
        AADD #–1, SP
        AMOV #_a, XAR0 ; |9|
        call #_sum ; |9|
                                ; call occurs [#_sum]   ; |9|
        MOV T0, *(#_sum1) ; |9|
        AMOV #_b, XAR0 ; |10|
        call #_sum ; |10|
                                ; call occurs [#_sum]   ; |10|
        MOV T0, *(#_sum2) ; |10|
        AADD #1, SP
        return
                                ; return occurs
```

Note: The algebraic instructions code example for Assembly Code Generated With –o3 and –pm Options is shown in Example B–4 on page B-4.

Caution must be exercised when using program mode (–pm) compilation on code that consists of a mixture of C/C++ and assembly language functions. These issues are described in detail in the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).

3.2.3 Using Function Inlining

There are two ways to enable the compiler to inline functions:

- ❑ Inlining controlled by the `inline` keyword. To enable this mode you must run the optimizer (that is, you must choose at least `–o0`.)
- ❑ Automatic inlining of small functions that are not declared as `inline` in your C/C++ code. To enable this mode use the `–o3` and `–oi`*`<size>`* compiler options.

The `-oi<size>` option may be used to specify automatic inlining of small functions even if they have not been declared with the `inline` keyword. The size of a function is an internal compiler notion of size. To see the size of a particular function use the `-onx` options described in Table 3–4 on page 3-9.

Example 3–6 shows the resulting assembly instructions when the code in Example 3–3 and Example 3–4 is compiled with `-o3`, `-pm`, and `-oi50` options.

In `main`, the function calls to `sum` have been inlined. However, code for the body of function `sum` has still been generated. The compiler must generate this code because it does not have enough information to eliminate the possibility that the function `sum` may be called by some other externally defined function. If no external function calls `sum`, it can be declared as `static inline`. The compiler will then be able to eliminate the code for `sum` after inlining.

Example 3–6. Assembly Generated Using -o3, -pm, and -oi50

```
_sum:
        MOV #0, T0 ; |3|
        RPT #9
            ADD *AR0+, T0, T0
        return    ; |11|
_main:
        AMOV #_a, XAR3 ; |9|
        RPT #9
||      MOV #0, AR1 ; |3|
            ADD *AR3+, AR1, AR1

        MOV AR1, *(#_sum1) ; |11|
        MOV #0, AR1 ; |3|
        AMOV #_b, XAR3 ; |10|
        RPT #9
            ADD *AR3+, AR1, AR1

        MOV AR1, *(#_sum2) ; |11|
        return
```

Note: The algebraic instructions code example for Assembly Generated Using –o3, –pm, and –oi50 is shown in Example B–5 on page B-5.

3.3 Profiling Your Code

In large applications, it makes sense to optimize the most important sections of code first. You can use the information generated by profiling options to get started. This section describes profiling methods to determine whether to move from Step 1 to Step 2 of the code development flow as described in Chapter 1 (or from Step 2 to Step 3). You can use several different methods to profile your code.

3.3.1 Using the `clock()` Function to Profile

To get cycle count information for a function or region of code with the stand-alone simulator, embed the `clock()` function in your C code. Example 3–7 demonstrates this technique.

Example 3–7. Using the `clock()` Function

```
#include <stdio.h>
#include <time.h>  /* Need time.h in order to call clock() */
int main()
{
   clock_t start, stop, overhead;
   start = clock();  /* Calculate the overhead of calling clock */
   stop = clock();   /* and subtract this amount from the results. */
   overhead = stop – start;
   start = clock();
   /* Function or Code Region to time goes here */
   stop = clock();
   printf(”cycles: %ld\n”,(long)(stop – start – overhead));
   return(0);
}
```

Caution: Using `clock()` to time a region of code could increase the cycle count of that region due to the extra variables needed to hold the timing information (the `stop`, `start`, and `overhead` variables above). Wrapping `clock()` around a function call should not affect the cycle count of that function.

3.3.2 Using CCS 2.0 to Profile

Code Composer Studio (CCS) 2.0 has extensive profiling options that can be used to profile your C code. First you must enable the clock by selecting Enable Clock from the Profiler menu. Selecting Start New Session from the Profiler menu starts a new profiling session. To profile all functions, click on the Profile All Functions button in the profiler session window. To profile certain functions or regions of code, click the Create Profile Area and enter the starting and ending line numbers of the code you wish to profile. (Note that you must build your code for debugging (–g option) to enable this feature.) Then, run your program and the profile information will be updated in the profiler session window.

More information on profiling with CCS 2.0 can be found in the online documentation.

3.4 Refining the C/C++ Code

This section describes C55x-specific optimization techniques that you can use to improve your C/C++ code performance. These techniques should be used in Step 2 of the code development flow as described in Chapter 1. Consider these tips when refining your code:

- Create loops that efficiently use C55x hardware loops, MAC hardware, and dual-MAC hardware.
- Use intrinsics to replace complicated C/C++ code
- Avoid the modulus operator when simulating circular addressing
- Use `long` accesses to reference 16-bit data in memory
- Write efficient control code

It is recommended that the following optimization techniques be applied in the order presented here. The code can be profiled after implementing the optimization described in each section to determine if further optimization is needed. If so, proceed to the next optimization. The six techniques presented in this section are summarized in Table 3–5. The indications (high, moderate, low, easy, many, some, few) in the table apply to typical DSP code. **Potential performance gain** estimates the performance improvement over no modifications to the code. **Ease of implementation** reflects both the required amount of change to the code and the complexity of the optimization. **Opportunities** are the number of places the optimization can be applied in typical DSP code.

Table 3–5. Summary of C/C++ Code Optimization Techniques

Optimization Technique	Potential Performance Gain	Ease of Implementation	Opportunities	Issues
Generate efficient loop code	High	Easy	Many	
Use MAC hardware efficiently	High	Moderate	Many	
Use Intrinsics	High	Moderate	Many	Reduces portability
Avoid modulus in circular addressing	Moderate	Easy	Some	
Use `long` accesses for 16-bit data	Low	Moderate	Few	
Generate efficient control code	Low	Easy	Few	

3.4.1 Generating Efficient Loop Code

You can realize substantial gains from the performance of your C/C++ loop code by refining your code in the following areas:

- Avoid function calls within the body of repeated loops. This enables the compiler to use very efficient hardware looping constructs (`repeat`, `localrepeat`, and `blockrepeat`, or `RPT`, `RPTBLOCAL`, and `RPTB` in mnemonic syntax).
- Keep loop code small to enable the compiler to use `localrepeat`.
- Analyze trip count issues.
- Use the `MUST_ITERATE` pragma.
- Use the `−o3` and `−pm` compiler options.

3.4.1.1 Avoid Function Calls within Loops

Whenever possible avoid using function calls within loops. Because repeat labels and counts would have to be preserved across calls, the compiler decides never to generate hardware loops that contain function calls. This leads to inefficient loop code.

3.4.1.2 Keep Loops Small to Enable `localrepeat`

Keeping loop code small enables the compiler to make use of the native `localrepeat` instruction. The compiler will generate `localrepeat` for small loops that do not contain any control flow structures other than forward conditionals. `Localrepeat` loops consume less power than other looping constructs. An example of a small loop that can use `localrepeat` is shown in Example 3−8 and Example 3−9. Example 3−8 shows C code and Example 3−9 shows the assembly code generated by the compiler.

Example 3−8. Simple Loop That Allows Use of `localrepeat`

```
void vecsum(const short *a, const short *b, short *c, unsigned int n)
{
   unsigned int i;

   for (i=0; i<=n−1; i++)
   {
      *c++ = *a++ + *b++;
   }
}
```

Example 3–9. Assembly Code for `localrepeat` Generated by the Compiler

```
_vecsum:
        SUB #1, T0, AR3
        MOV AR3, BRC0
        RPTBLOCAL L2-1
            ADD *AR0+, *AR1+, AC0 ; |7|
            MOV HI(AC0), *AR2+ ; |7|
L2:
        return
```

Note: The algebraic instructions code example for Assembly Code for localrepeat Generated by the Compiler is shown in Example B–6 on page B-5.

3.4.1.3 Trip Count Issues

A trip count is the number of times that a loop executes; the trip counter is the variable used to count each iteration. When the trip counter reaches the limit equal to the trip count, the loop terminates. Maximum performance for loop code is gained when the compiler can determine the exact minimum and maximum for the trip count. To this end, use the following techniques to convey trip count information to the compiler:

- ❑ Use `int` (or `unsigned int`) type for trip counter variable, whenever possible.
- ❑ Use the `MUST_ITERATE` pragma to eliminate code to skip around loops and help the compiler generate efficient hardware loops. This pragma can also be used to aid in loop unrolling.
- ❑ Be sure to use the `-o3` and `-pm` compiler options to allow the optimizer access to the whole program or large parts of it and to characterize the behavior of loop trip counts.

Using `int` Type. Using the type `int` for the trip counter is important to allow the compiler to generate hardware looping constructs.

In Example 3–10, consider this simple *for* loop:

```
for(i = 0; i<n; i++)
```

If, for example, `i` and `n` were declared to be of type `long`, no hardware loop could be generated. This is because the C55x internal loop iteration count register is only 16 bits wide. If `i` and `n` are declared as type `int`, then the compiler will generate a hardware loop.

3.4.1.4 Using the MUST_ITERATE Pragma

The MUST_ITERATE pragma is used to convey programmer knowledge about loops to the compiler. It should be used as much as possible to aid the compiler in the optimization of loops.

Example 3–10 shows code to compute the sum of a vector. The corresponding assembly code is shown in Example 3–11. Notice the conditional branch that jumps around the loop body in the generated assembly code. The compiler must insert this additional code if there is any possibility that the loop could execute zero times. In this particular case the loop upper bound n is an integer. Thus, n could be zero or negative in which case C semantics would dictate that the *for* loop body would not execute. A hardware loop must execute at least once, so the jump around code ensures correct execution in cases where *n <= 0*.

Example 3–10. Inefficient Loop Code for Loop Variable and Constraints (C)

```
int sum(const short *a, int n)
{
   int sum = 0;
   int i;
   for(i=0; i<n; i++)
   {
      sum += a[i];
   }
  return sum;
}
```

Example 3–11. Inefficient Loop Code for Variable and Constraints (Assembly)

```
_sum:
        MOV #0, AR1 ; |3|
        BCC L2,T0 <= #0 ; |6|
                                        ; branch occurs  ; |6|
        SUB #1, T0, AR2
        MOV AR2, CSR
        RPT CSR
            ADD *AR0+, AR1, AR1

        MOV AR1, T0 ; |11|
        return    ; |11|
```

Note: The algebraic instructions code example for Inefficient Loop Code for Variable and Constraints (Assembly) is shown in Example B–7 on page B-6.

If it is known that the loop always executes at least once, this fact can be communicated to the compiler via the `MUST_ITERATE` pragma. Example 3–12 shows how to use the pragma for this piece of code. Example 3–13 shows the more efficient assembly code that can now be generated because of the pragma.

Example 3–12. Using the `MUST_ITERATE` Pragma

```
int sum(const short *a, int n)
{
   int sum = 0;
   int i;

#pragma MUST_ITERATE(1)
   for(i=0; i<n; i++)
   {
      sum += a[i];
   }
   return sum;
}
```

(Note that the same effect could be generated by using an `_nassert`, to assert to the compiler that `n` is greater than zero: `_nassert(n>0)`).

Example 3–13. Assembly Code Generated With the `MUST_ITERATE` Pragma

```
_sum:
        SUB #1, T0, AR2
        MOV AR2, CSR
        MOV #0, AR1 ; |3|
        RPT CSR
            ADD *AR0+, AR1, AR1

        MOV AR1, T0 ; |12|
        return    ; |12|
```

Note: The algebraic instructions code example for Assembly Code Generated With the `MUST_ITERATE` Pragma is shown in Example B–8 on page B-6.

`MUST_ITERATE` can be used to communicate several different pieces of information to the compiler. The format of the pragma is:

```
#pragma MUST_ITERATE(min, max, mult)
```

All fields are optional. `min` is the minimum number of iterations of the loop, `max` is the maximum number of iterations of the loop, and `mult` tells the compiler that the loop always executes a multiple of `mult` times. If some of these values are not known until run time, do not include them in the pragma. Incorrect information communicated via the pragma could result in undefined program behavior. The `MUST_ITERATE` pragma must appear immediately before the loop that it is meant to describe in the C code. `MUST_ITERATE` can be used in the following ways:

- It can convey that the trip count will be greater than some minimum value.

```
/* This loop will always execute at least 30 times */
#pragma MUST_ITERATE(30)
for(j=0; j<x; j++)
```

- It can convey the maximum trip count.

```
/* The loop will execute no more than 100 times */
#pragma MUST_ITERATE(,100)
for (j=0; j<x; j++)
```

- It can convey that the trip count is always divisible by a value.

```
/* The loop will execute some multiple of 4 times */
#pragma MUST_ITERATE(,,4)
for (j=0; j<x; j++)
```

Consider the following loop header (from the ETSI gsmefr benchmark):

```
for(i=a[0]; i < 40; i +=5)
```

To generate a hardware loop, the compiler would need to emit code that would determine the number of loop iterations at run time. This code would require an integer division. Since this is computationally expensive, the compiler will not generate such code and will not generate a hardware loop. However, if the programmer knows that, for example, `a[0]` is always less than or equal to 4, then the loop always executes exactly eight times. This can be communicated via a `MUST_ITERATE` pragma enabling the compiler to generate an efficient hardware loop:

```
#pragma MUST_ITERATE(8,8)
```

3.4.2 Efficient Use of MAC hardware

Multiply-and-accumulate (MAC) is a very common paradigm in DSP algorithms, and a C55x DSP has hardware to perform MAC operations efficiently. It can perform a single MAC (or multiply, multiply and subtract) operation or two MAC operations in a single cycle (a dual-MAC operation). The next section describes how to write efficient, small loops that use a single MAC operation. Section 3.4.2.2 describes how to enable the compiler to generate dual-MAC operations from your C/C++ code.

3.4.2.1 Special Considerations When Using MAC Constructs

The compiler can generate a very efficient single repeat MAC construct (that is, a `repeat` (`RPT`) loop with a MAC as its only instruction.) To facilitate the generation of single repeat MAC constructs, use local rather than global variables for the summation, as shown in Example 3–14. If a global variable is used, the compiler is obligated to perform an intervening storage to the global object. This prevents it from generating a single repeat.

In the case where Q15 arithmetic is being simulated, the result of the MAC operation may be accumulated into a `long` object. The result may then be shifted and truncated before the return, as shown in Example 3–15.

Example 3–14. Use Local Rather Than Global Summation Variables

```
/* Not recommended */
int gsum=0;
void dotp1(const int *x, const int *y, unsigned int n)
{
   unsigned int i;
   for(i=0; i<=n–1; i++)
      gsum += x[i] * y[i];
}
/* Recommended */
int dotp2(const int *x, const int *y, unsigned int n)
{
   unsigned int i;
   int lsum=0;
   for(i=0; i<=n–1; i++)
      lsum += x[i] * y[i];
   return lsum;
}
```

Example 3–15. Returning Q15 Result for Multiply Accumulate

```
int dotp(const int *x, const int *y, unsigned int n)
{
   unsigned int i;
   long sum=0;
   for(i=0; i<=n–1; i++)
      sum += (long)x[i] * y[i];
   return (int)((sum>>15) & 0x0000FFFFL);
}
```

3.4.2.2 Generating Dual-MAC Operations

A dual-MAC operation (2 multiply-and-accumulate/subtract instructions in a single cycle) is one of the most important hardware features of a C55x DSP. (Note, the term dual-MAC will be used to refer to dual multiplies, dual multiply-and-accumulates and dual multiply-and-subtracts.) You must follow several guidelines in your C code to help the compiler generate dual-MAC operations.

In order for the compiler to generate a dual-MAC operation, the code must have two consecutive MAC (or MAS/multiply) instructions that get all their multiplicands from memory and share one multiplicand. The two operations must not write their results to the same variable or location. The compiler can easily turn this example into a dual-MAC:

```
int *a,*b, onchip *c;
long  s1,s2;
[...]
s1 = s1 + (*a++ * *c);
s2 = s2 + (*b++ * *c++);
```

This is a sequence of two MAC instructions that share the `*c` memory reference. Intrinsics can also be transformed into dual-MACs:

```
s1 = _smac(s1,*a++,*c);
s2 = _smac(s2,*b++,*c++);
```

You must inform the compiler that the memory pointed to by the shared dual-MAC operand is on chip (a requirement for the addressing mode used for the shared operand). There are two ways to do this. The first (and preferred) way involves the use of the `onchip` type qualifier. It is used like this:

```
void foo(int onchip *a)
{
   int onchip b[10];
   ...
}
```

This keyword can be applied to any pointer or array and indicates that the memory pointed to by that pointer or array is always on chip.

The second technique is to compile with the `–mb` switch (passed to cl55). This asserts to the compiler that all data pointed to by the shared dual-MAC pointer will be on chip. This switch is a shortcut. Instead of putting many `onchip` qualifiers into the code, `–mb` can be used instead. You must ensure that all required data will be on chip. If `–mb` is used and some data pointed to by a shared dual-MAC pointer is not on chip, undefined behavior may result. Remember, this is a shortcut. The `onchip` keyword should be used to enable dual-MAC operations in most circumstances. Using `–mb` could result in dual-MACs being generated in unexpected or undesirable places.

Unfortunately, a lot of C code that could benefit from using dual-MACs is not written in such a way as to enable the compiler to generate them. However, the compiler can sometimes transform the code in such a way to generate a dual-MAC. For example, look at Example 3–16 which shows a C version of a simple

FIR filter. (Notice the `onchip` keyword used for the pointer parameter `h`.) In order to generate a dual-MAC in this case, the compiler must somehow generate two consecutive MAC operations from the single MAC operation in the code. This is done via a loop transformation called unroll-and-jam. This transformation replicates the outer loop and then fuses the two resulting inner loops back together. Example 3–17 shows what the code in Example 3–16 would look like if unroll-and-jam were applied manually.

Example 3–16. C Code for an FIR Filter

```
void fir(short onchip *h, short *x, short * y, short m, short n)
{
   short i,j;
   long y0;
   for (j = 0; j < m; j++)
   {
      y0 = 0;

      for (i = 0; i < n; i++)
         y0 += (long)x[i + j] * h[i];

      y[j] = (short) (y0 >> 16);
   }
}
```

Example 3–17. FIR C Code After Unroll-and-Jam Transformation

```
void fir(short onchip *h, short *x, short *y, short m, short n)
{
   short i,j;
   long y0,y1;
   for (j = 0; j < m; j+=2)
   {
      y0 = 0;
      y1 = 0;

      for (i = 0; i < n; i++)
      {
         y0 += (long)x[i + j] * h[i];
         y1 += (long)x[i + j+1] * h[i];
      }

      y[j] = (short) (y0 >> 16);
      y[j+1] = (short) (y1 >> 16);
   }
}
```

Notice that now we are computing two separate sums (`y0` and `y1`) for each iteration of the outer loop. If this C code were fed to the compiler, it would generate a dual-MAC in the inner loop. The compiler can perform the unroll-and-jam transformation automatically, but the programmer must provide additional information to ensure that the transformation is safe.

- The compiler must determine that the outer loop repeats an even number of times. If the loop bounds are provably constant, the compiler can determine this automatically. Otherwise, if the user knows that the loop always repeats an even number of times, a `MUST_ITERATE` pragma can be used immediately preceding the outer loop:

  ```
  #pragma MUST_ITERATE(1,,2)
  ```

  (Note that the first parameter (1) indicates that the outer loop always executes at least once. This is to eliminate loop jump-around code as described in section 3.4.1.4 on page 3-18.)

- ❑ The compiler must also know that the inner loop executes at least once. This can be specified by inserting the following `MUST_ITERATE` pragma just before the `for` statement of the inner loop:

  ```
  #pragma MUST_ITERATE(1)
  ```

- ❑ The compiler must also know that there are no memory conflicts in the loop nest. In our example, that means the compiler must know that all the writes to array `y` cannot affect the values in array `x` or `h`. Consider the code in Example 3–17 on page 3-25. We have changed the order of memory accesses by performing unroll-and-jam. In the transformed code, twice as many reads from `x` (and `h`) occur before any writes to `y`. If writes to `y` could affect the data pointed to by `x` (or `h`), the transformed code could produce different results. If these three arrays were locally declared arrays, the compiler would not have a problem. In this case we pass the arrays into the function via pointer parameters. If the programmer is sure that writes to `y` will not affect the arrays `x` and `h` within the function, the `restrict` keyword can be used in the function declaration:

  ```
  void fir(short onchip *h, short *x, short * restrict
  y, short m, short n)
  ```

  The `restrict` keyword tells the compiler that no other variable will point at the memory that `y` points to. (See section 3.1.3 for more information on memory dependences and `restrict`.) The final C code is shown in Example 3–18, and the corresponding assembly code in Example 3–19.

  Even using the `MUST_ITERATE` pragma and `restrict` qualifiers, some loops may still be too complicated for the compiler to generate as dual-MACs. If there is a piece of code you feel could benefit from dual-MAC operations, it may be necessary to transform the code by hand. This process is similar to the transformations described for writing dual-MAC operations in assembly code as described in section 4.1.

Example 3–18. FIR Filter With `MUST_ITERATE` Pragma and `restrict` Qualifier

```
void fir(short onchip *h, short *x, short * restrict y, short m,
         short n)
{
   short i,j;
   long y0;
#pragma MUST_ITERATE(1,,2)
   for (j = 0; j < m; j++)
   {
      y0 = 0;
#pragma MUST_ITERATE(1)
      for (i = 0; i < n; i++)
         y0 += (long)x[i + j] * h[i];

      y[j] = (short) (y0 >> 16);
   }
}
```

Example 3–19. Generated Assembly for FIR Filter Showing Dual-MAC

```
_fir:
        ADD #1, T0, AR3
        SFTS AR3, #-1
        SUB #1, AR3
        MOV AR3, BRC0
        PSH T3, T2
        MOV #0, T3 ; |6|
||      MOV XAR0, XCDP
        AADD #-1, SP
        RPTBLOCAL L4-1

            SUB #1, T1, T2
            MOV XAR1, XAR3
            MOV T2, CSR
            ADD T3, AR3
            MOV XAR3, XAR4
            ADD #1, AR4
            MOV #0, AC0 ; |8|
            RPT CSR
||          MOV AC0, AC1 ; |8|
                MAC *AR4+, *CDP+, AC0 :: MAC *AR3+, *CDP+, AC1
                                            ; loop ends
L3:
            MOV XCDP, XAR0
            ADD #2, T3
            SUB T1, AR0
||          MOV HI(AC0), *AR2(short(#1))
            ADD #2, AR2
||          MOV HI(AC1), *AR2
            MOV XAR0, XCDP

        AADD #1, SP
        POP T3,T2
        return
```

Note: The algebraic instructions code example for Generated Assembly for FIR Filter Showing Dual-MAC is shown in Example B–9 on page B-7.

3.4.3 Using Intrinsics

The C55x compiler provides intrinsics, special functions that map directly to inlined C55x instructions, to optimize your C code quickly. Intrinsics are specified with a leading underscore (_) and are accessed by calling them as you would call a function.

For example, without intrinsics, saturated addition can only be expressed in C code by writing a multicycle function, such as the one in Example 3–20.

Example 3–21 shows the resultant inefficient assembly language code generated by the compiler.

Example 3–20. Implementing Saturated Addition in C

```
int sadd(int a, int b)
{
   int result;
   result = a + b;

   // Check to see if ’a’ and ’b’ have the same sign
   if (((a^b) & 0x8000) == 0)
   {
      // If ’a’ and ’b’ have the same sign, check for underflow
      // or overflow

      if ((result ^ a) & 0x8000)
      {
         // If the result has a different sign than ’a’
         // then underflow or overflow has occurred.
         // if ’a’ is negative, set result to max negative
         // If ’a’ is positive, set result to max  positive
         // value
         result = ( a < 0) ? 0x8000 : 0x7FFF;
      }
   }
   return result;
}
```

Example 3–21. Inefficient Assembly Code Generated by C Version of Saturated Addition

```
_sadd:
        MOV T1, AR1 ; |5|
        XOR T0, T1 ; |9|
        BTST @#15, T1, TC1 ; |9|
        ADD T0, AR1
        BCC L2,TC1 ; |9|
                                ; branch occurs  ; |9|
        MOV T0, AR2 ; |9|
        XOR AR1, AR2 ; |9|
        BTST @#15, AR2, TC1 ; |9|
        BCC L2,!TC1 ; |9|
                                ; branch occurs  ; |9|
        BCC L1,T0 < #0 ; |22|
                                ; branch occurs  ; |22|
        MOV #32767, T0 ; |22|
        B L3      ; |22|
                                ; branch occurs  ; |22|
L1:
        MOV #-32768, AR1 ; |22|
L2:
        MOV AR1, T0 ; |25|
L3:
        return    ; |25|
                                ; return occurs  ; |25|
```

Note: The algebraic instructions code example for Inefficient Assembly Code Generated by C Version of Saturated Addition is shown in Example B–10 on page B-8.

The code for the C simulated saturated addition can be replaced by a single call to the `_sadd` intrinsic as is shown in Example 3–22. The assembly code generated for this C source is shown in Example 3–23.

Note that using compiler intrinsics reduces the portability of your code. You may consider using ETSI functions instead of intrinsics. These functions can be mapped to intrinsics for various targets. For C55x code, the file gsm.h defines the ETSI functions using compiler intrinsics. (The actual C code ETSI functions can be used when compiling on the host or other target without intrinsics.) For example, the code in Example 3–22 could be rewritten to use the ETSI `add` function as shown in Example 3–24. The ETSI `add` function is mapped to the `_sadd` compiler intrinsic in the header file gsm.h. (Of course, you probably want to replace calls to the `sadd` function with calls to the ETSI `add` function.)

Table 3–6 lists the intrinsics supported by the C55x compiler. For more information on using intrinsics, please refer to the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).

Example 3–22. Single Call to `_sadd` Intrinsic

```
int sadd(int a, int b)
{
   return _sadd(a,b);
}
```

Example 3–23. Assembly Code Generated When Using Compiler Intrinsic for Saturated Add

```
_sadd:
        BSET ST3_SATA
        ADD T1, T0 ; |3|
        BCLR ST3_SATA
        return    ; |3|
                                                ; return occurs  ; |3|
```

Note: The algebraic instructions code example for Assembly Code Generated When Using Compiler Intrinsic for Saturated Add is shown in Example B–11 on page B-9.

Example 3–24. Using ETSI Functions to Implement `sadd`

```
#include <gsm.h>
int sadd(int a, int b)
{
   return add(a,b);
}
```

Table 3–6. TMS320C55x C Compiler Intrinsics

Intrinsic	C Compiler Intrinsic Description
int _sadd(int src1, int src2);	Adds two 16-bit integers, producing a saturated 16-bit result (SATA bit set)
long _lsadd(long src1, long src2);	Adds two 32-bit integers, producing a saturated 32-bit result (SATD bit set)
long long _llsadd(long long src1, long long src2);	Adds two 40-bit integers, producing a saturated 40-bit result (SATD bit set)
int _ssub(int src1, int src2);	Subtracts src2 from src1, producing a saturated 16-bit result (SATA bit set)
long _lssub(long src1, long src2);	Subtracts src2 from src1, producing a saturated 32-bit result (SATD bit set)
long long _llssub(long long src1, long long src2);	Subtracts src2 from src1, producing a saturated 40-bit result (SATD bit set)
int _smpy(int src1, int src2);	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 16-bit result. (SATD and FRCT bits set)
long _lsmpy(int src1, int src2);	Multiplies src1 and src2, and shifts the result left by 1. Produces a saturated 32-bit result. (SATD and FRCT bits set)
long _smac(long src, int op1, int op2);	Multiplies op1 and op2, shifts the result left by 1, and adds it to src. Produces a saturated 32-bit result. (SATD, SMUL, and FRCT bits set)
long _smas(long src, int op1, int op2);	Multiplies op1 and op2, shifts the result left by 1, and subtracts it from src. Produces a 32-bit result. (SATD, SMUL and FRCT bits set)
int _abss(int src);	Creates a saturated 16-bit absolute value. _abss(8000h) results in 7FFFh (SATA bit set)
long _labss(long src);	Creates a saturated 32-bit absolute value. _labss(8000000h) results in 7FFFFFFFh (SATD bit set)
long long _llabss(long long src);	Creates a saturated 40-bit absolute value. _llabss(800000000h) results in 7FFFFFFFFFh (SATD bit set)
int _sneg(int src);	Negates the 16-bit value with saturation. _sneg(8000h) results in 7FFFh
long _lsneg(long src);	Negates the 32-bit value with saturation. _lsneg(80000000h) results in 7FFFFFFFh
long long _llsneg(long long src);	Negates the 40-bit value with saturation. _llsneg(8000000000h) results in 7FFFFFFFFFh

Table 3–6. TMS320C55x C Compiler Intrinsics (Continued)

Intrinsic	C Compiler Intrinsic Description
long _smpyr(int src1, int src2);	Multiplies src1 and src2, shifts the result left by 1, and rounds by adding 2^{15} to the result and zeroing out the lower 16 bits. (SATD and FRCT bits set)
long _smacr(long src, int op1, int op2);	Multiplies op1 and op2, shifts the result left by 1, adds the result to src, and then rounds the result by adding 2^{15} and zeroing out the lower 16 bits. (SATD , SMUL, and FRCT bits set)
long _smasr(long src, int op1, int op2);	Multiplies op1 and op2, shifts the result left by 1, subtracts the result from src, and then rounds the result by adding 2^{15} and zeroing out the lower 16 bits. (SATD , SMUL and FRCT bits set)
int _norm(int src);	Produces the number of left shifts needed to normalize 16-bit value.
int _lnorm(long src);	Produces the number of left shifts needed to normalize 32-bit value.
long _rnd(long src);	Rounds src by adding 2^{15}. Produces a 32-bit saturated result with the lower 16 bits zeroed out. (SATD bit set)
int _sshl(int src1, int src2);	Shifts src1 left by src2 and produces a 16-bit result. The result is saturated if src2 is less than or equal to 8. (SATD bit set)
long _lsshl(long src1, int src2);	Shifts src1 left by src2 and produces a 32-bit result. The result is saturated if src2 is less than or equal to 8. (SATD bit set)
int _shrs(int src1, int src2);	Shifts src1 right by src2 and produces a 16-bit result. Produces a saturated 16-bit result. (SATD bit set)
long _lshrs(long src1, int src2);	Shifts src1 right by src2 and produces a 32-bit result. Produces a saturated 32-bit result. (SATD bit set)

3.4.4 Using Long Data Accesses for 16-Bit Data

The primary use of treating 16-bit data as `long` is to transfer data quickly from one memory location to another. Since 32-bit accesses also can occur in a single cycle, this could reduce the data-movement time by half. The only limitation is that the data must be aligned on a double word boundary (that is, an even word boundary). The code is even simpler if the number of items transferred is a multiple of 2. To align the data use the `DATA_ALIGN` pragma:

```
short x[10];
#pragma DATA_ALIGN(x,2)
```

Example 3–25 shows a memory copy function that copies 16-bit data via 32-bit pointers.

Example 3–25. Block Copy Using Long Data Access

```
void copy(const short *a, const short *b, unsigned short n)
{
   unsigned short i;
   unsigned short na;
   long *src, *dst;
   // This code assumes that the number of elements to transfer ’n’
   // is a multiple of 2. Divide the number of 1 word transfers
   // by 2 to get the number of double word transfers.
   na = (n>>1) –1;
   // Set beginning address of SRC and DST for long transfer.
   src = (long *)a;
   dst = (long *)b;
   for (i=0; i<= na; i++)
   {
      *dst++ = *src++;
   }
}
```

3.4.5 Simulating Circular Addressing in C

When simulating circular addressing in C, avoid using the modulus operator (%). Modulus can take several cycles to implement and often results in a call to a run-time library routine. Instead, use the macros shown in Example 3–26. Use of these macros is only valid when the index increment amount (`inc`) is less than the buffer size (`size`), and when the code of `CIRC_UPDATE` is always used to update the array index. Example 3–28 shows the same example using modulus.

Notice that `CIRC_REF` simply expands to `(var)`. In the future, using modulus will be the more efficient way to implement circular addressing in C. The compiler will be able to transform certain uses of modulus into efficient C55x circular addressing code. At that time, the `CIRC_UPDATE` and `CIRC_REF` macros can be updated to use modulus. Use of these macros will improve current performance and minimize future changes needed to take advantage of improved compiler functionality with regards to circular addressing.

Example 3–27 displays the resulting assembly code generated by the compiler.

The (much less efficient) resulting assembly code is shown in Example 3–29.

Example 3–26. Simulating Circular Addressing in C

```
#define CIRC_UPDATE(var,inc,size)\
     (var) +=(inc); if ((var)>=(size)) (var)-=(size);
#define CIRC_REF(var,size) (var)
long circ(const int *a, const int *b, int nb, int na)
{
   int i,x=0;
   long sum=0;
   for(i=0; i<na; i++)
   {
      sum += (long)a[i] * b[CIRC_REF(x,nb)];
      CIRC_UPDATE(x,1,nb)
   }
   return sum;
}
```

Example 3–27. Assembly Output for Circular Addressing C Code

```
_circ:
        MOV #0, AC0 ; |7|
        BCC L2,T1 <= #0 ; |9|
                                        ; branch occurs  ; |9|
        SUB #1, T1, AR3
        MOV AR3, BRC0
        MOV #0, AR2 ; |6|
        RPTBLOCAL L2-1
                                            ; loop starts
L1:
            MACM *AR1+, *AR0+, AC0, AC0 ; |11|
||          ADD #1, AR2
            CMP AR2 < T0, TC1 ; |12|
            XCCPART !TC1 ||
                SUB T0, AR1
            XCCPART !TC1 ||
                SUB T0, AR2
                                        ; loop ends   ; |13|
L2:
        return    ; |14|
                                        ; return occurs  ; |14|
```

Note: The algebraic instructions code example for Assembly Output for Circular Addressing C Code is shown in Example B–12 on page B-9.

Example 3–28. Circular Addressing Using Modulus Operator

```
long circ(const int *a, const int *b, int nb, int na)
{
   int i,x=0;
   long sum=0;
   for(i=0; i<na; i++)
   {
      sum += (long)a[i] * b[x % mb];
      x++;
   }
   return sum;
}
```

Example 3–29. Assembly Output for Circular Addressing Using Modulus Operator

```
_circ:
        PSH T3, T2
        AADD #-7, SP
        MOV XAR1, dbl(*SP(#0))
||      MOV #0, AC0 ; |4|
        MOV AC0, dbl(*SP(#2)) ; |4|
||      MOV T0, T2 ; |2|
        BCC L2,T1 <= #0 ; |6|
                                            ; branch occurs  ; |6|
        MOV #0, T0 ; |3|
        MOV T1, T3
||      MOV XAR0, dbl(*SP(#4))
L1:
        MOV dbl(*SP(#0)), XAR3
        MOV *AR3(T0), T1 ; |8|
        MOV dbl(*SP(#4)), XAR3
        MOV dbl(*SP(#2)), AC0 ; |8|
        ADD #1, T0, T0
        MACM *AR3+, T1, AC0, AC0 ; |8|
        MOV AC0, dbl(*SP(#2)) ; |8|
        MOV XAR3, dbl(*SP(#4))
        call #I$$MOD ; |9|
||      MOV T2, T1 ; |9|
                                            ; call occurs [#I$$MOD] ; |9|
        SUB #1, T3
        BCC L1,T3 != #0 ; |10|
                                            ; branch occurs  ; |10|
L2:
        MOV dbl(*SP(#2)), AC0
        AADD #7, SP ; |11|
        POP T3,T2
        return    ; |11|
                                            ; return occurs  ; |11|
```

Note: The algebraic instructions code example for Assembly Output for Circular Addressing Using Modulo is shown in Example B–13 on page B-10.

3.4.6 Generating Efficient Control Code

Control code typically tests a number of conditions to determine the appropriate action to take.

The compiler generates similar constructs when implementing nested if-then-else and switch/case constructs when the number of case labels is fewer than eight. Because the first true condition is executed with the least amount of branching, it is best to allocate the most often executed conditional first. When the number of case labels exceeds eight, the compiler generates a .switch label section. In this case, it is still optimal to place the most often executed code at the first case label.

In the case of single conditionals, it is best to test against zero. For example, consider the following piece of C code:

```
if (a!=1)       /* Test against 1 */
   <inst1>
else
   <inst2>
```

If the programmer knows that `a` is always 0 or 1, the following more efficient C code can be used:

```
if (a==0)       /* Test against 0 */
   <inst1>
else
   <inst2>
```

In most cases this test against zero will result in more efficient compiled code.

3.4.7 Summary of Coding Idioms for C55x

Table 3–7 shows the C coding methods that should be used for some basic DSP operations to generate the most efficient assembly code for the C55x.

Table 3–7. C Coding Methods for Generating Efficient C55x Assembly Code

Operation	Recommended C Code Idiom

16bit * 16bit => 32bit (multiply)	`int a,b;`
	`long c;`
	`c = (long)a * b;`

Q15 * Q15 => Q15 (multiply)
Fractional mode with saturation
	`int a,b,c;`
	`c = _smpy(a,b);`

Q15 * Q15 => Q31 (multiply)
Fractional mode with saturation
	`int a,b;`
	`long c;`
	`c = _lsmpy(a,b);`

32bit + 16bit * 16bit => 32 bit (MAC)	`int a,b;`
	`long c;`
	`c = c + ((long)a * b));`

Q31 + Q15 * Q15 => Q31 (MAC)
Fractional mode with saturation
	`int a,b;`
	`long c;`
	`c = _smac(c,a,b);`

32bit – 16bit * 16bit => 32 bit (MAS)	`int a,b;`
	`long c;`
	`c = c – ((long)a * b));`

Q31 – Q15 * Q15 => Q31 (MAS)
Fractional mode with saturation
	`int a,b;`
	`long c;`
	`c = _smas(c,a,b);`

16bit +/– 16bit => 16bit
32bit +/– 32bit => 32bit
40bit +/– 40bit => 40bit (addition or subtraction)
	`<int, long, long long> a,b,c;`
	`c = a + b;`
	`/* or */`
	`c = a – b;`

16bit + 16bit => 16bit (addition)
with saturation
	`int a,b,c;`
	`c = _sadd(a,b);`

32bit + 32bit => 32bit (addition)
with saturation
	`long a,b,c;`
	`c = _lsadd(a,b);`

40bit + 40bit => 40bit (addition)
with saturation
	`long long a,b,c;`
	`c = _llsadd(a,b);`

Table 3–7. C Coding Methods for Generating Efficient C55x Assembly Code (Continued)

Operation	Recommended C Code Idiom

16bit – 16bit => 16bit (subtraction)
with saturation
	`int a,b,c;`
	`c = _ssub(a,b);`

32bit – 32bit => 32bit (subtraction)
with saturation
	`long a,b,c;`
	`c = _lssub(a,b);`

40bit – 40bit => 40bit (subtraction)
with saturation
	`long long a,b,c;`
	`c = _llssub(a,b);`

I16bitI => 16bit
I32bitI => 32bit
I40bitI => 40bit (absolute value)
	`<int, long, long long> a,b;`
	`b = abs(a);  /* or */`
	`b = labs(a); /* or */`
	`b = llabs(a);`

I16bitI => 16bit
I32bitI => 32bit
I40bitI => 40bit (absolute value)
with saturation
	`<int, long, long long> a,b;`
	`b = _abss(a);  /* or */`
	`b = _labss(a); /* or */`
	`b = _llabss(a);`

round(Q31) = > Q15 (rounding towards infinity)
with saturation
	`long a;`
	`int b;`
	`b = _rnd(a)>>16;`

Q39 => Q31 (format change)	`long long a;`
	`long b;`
	`b = a >> 8;`

Q30 = > Q31 (format change)
with saturation
	`long a;`
	`long b;`
	`b = _lsshl(a,1);`

40bit => 32bit both Q31 (size change)	`long long a;`
	`long b;`
	`b = a;`

3.5 Memory Management Issues

This section provides a brief discussion on managing data and code in memory. Memory usage and subsequent code speed may be affected by a number of factors. The discussion in this section will focus on the following areas that affect memory usage. The information in this section is valid regardless of object code format (COFF or DWARF).

❑ Avoiding holes caused by data alignment

❑ Local versus global symbol declarations

❑ Stack configuration

❑ Allocating code and data in the C55x memory map

3.5.1 Avoiding Holes Caused by Data Alignment

The compiler requires that all values of type `long` be stored on an even word boundary. When declaring data objects (such as structures) that may contain a mixture of multi-word and single-word elements, place variables of type `long` in the structure definition first to avoid holes in memory. The compiler automatically aligns structure objects on an even word boundary. Placing these items first takes advantage of this alignment.

Example 3–30. Considerations for Long Data Objects in Structures

```
/* Not recommended */
typedef struct abc{
   int a;
   long b;
   int c;
} ABC;
/* Recommended */
typedef struct abc{
   long a;
   int b,c;
} ABC;
```

3.5.2 Local vs. Global Symbol Declarations

Locally declared symbols (symbols declared within a C function), are allocated space by the compiler on the software stack. Globally declared symbols (symbols declared at the file level) and static local variables are allocated space in the compiler generated .bss section by default. The C operating environment created by the C boot routine, _c_int00, places the C55x DSP in CPL mode. CPL mode enables stack-based offset addressing and disables DP offset addressing. The compiler accesses global objects via absolute addressing modes. Because the full address of the global object is encoded as part of the instruction in absolute addressing modes, this can lead to larger code size and potentially slower code. CPL mode favors the use of locally declared objects, since it takes advantage of stack offset addressing. Therefore, if at all possible, it is better to declare and manipulate local objects rather than global objects. When function code requires multiple uses of a non-volatile global object, it is better to declare a local object and assign it the appropriate value:

```
extern int Xflag;
int function(void)

{
  int lflag = Xflag;

   .
   x = lflag ? lflag & 0xfffe : lflag;
   .
    .
   return x;
}
```

3.5.3 Stack Configuration

The C55x has two software stacks: the data stack (referenced by the pointer SP) and the system stack (referenced by the pointer SSP). These stacks can be indexed independently or simultaneously depending on the chosen operating mode. There are three possible operating modes for the stack:

❑ Dual 16-bit stack with fast return

❑ Dual 16-bit stack with slow return

❑ 32-bit stack with slow return

In the 32-bit mode, SSP is incremented whenever SP is incremented. The primary use of SSP is to hold the upper 8 bits of the return address for context saving. It is not used for data accesses. Because the C compiler allocates space on the data stack for all locally declared objects, operating in this mode doubles the space allocated for each local object. This can rapidly increase memory usage. In the dual 16-bit modes, the SSP is only incremented for context saving (function calls, interrupt handling). Allocation of memory for local objects does not affect the system stack when either of the dual 16-bit modes is used.

Additionally, the selection of fast return mode enables use of the RETA and CFCT registers to effect return from functions. This potentially increases execution speed because it reduces the number of cycles required to return from a function.

It is recommended to use dual 16-bit fast return mode to reduce memory space requirements and increase execution speed. The stack operating mode is selected by setting bits 28 and 29 of the reset vector address to the appropriate values. Dual 16-bit fast return mode may be selected by using the .ivec assembler directive when creating the address for the reset vector. For example:

```
   .ivec      reset_isr_addr, USE_RETA
```

(This is the default mode for the compiler as setup by the supplied runtime support library.) The assembler will automatically set the correct value for bits 28 and 29 when encoding the reset vector address. For more information on stack modes, see the *TMS320C55x DSP CPU Reference Guide* (SPRU371).

3.5.4 Allocating Code and Data in the C55x Memory Map

The compiler groups generated code and data into logical units called sections. Sections are the building blocks of the object files created by the assembler. They are the logical units operated on by the linker when allocating space for code and data in the C55x memory map.

The compiler/assembler can create any of the sections described in Table 3–8.

Table 3–8. Section Descriptions

Section	Description
.cinit	Initialization record table for global and static C variables
.pinit	A list of constructor function pointers called at boot time
.const	Explicitly initialized global and static const symbols
.text	Executable code and constants
.bss	Global and static variables
.ioport	Uninitialized global and static variables of type ioport
.stack	Data stack (local variables, lower 16 bits of return address, etc.)
.sysstack	System stack (upper 8 bits of 24 bit return address)
.sysmem	Memory for dynamic allocation functions
.switch	Labels for switch/case
.cio	For CIO Strings and buffers

These sections are encoded in the object file produced by the assembler. When linking the objects, it is important to pay attention to where these sections are linked in memory to avoid as many memory conflicts as possible. Following are some recommendations:

❑ Allocate .stack and .sysstack in DARAM (dual-access RAM): the .stack and .sysstack sections are often accessed at the same time when a function call/return occurs. If these sections are allocated in the same SARAM (single-access RAM) block, then a memory conflict will occur, adding additional cycles to the call/return operation. If they are allocated in DARAM or separate SARAM blocks, this will avoid such a conflict.

❑ The start address of the .stack and .sysstack sections are used to initialize the data stack pointer (SP) and the system stack pointer (SSP), respectively. Because these two registers share a common data page pointer register (SPH) these sections must be allocated on the same 64K-word memory page.

❑ Allocate the .bss and .stack sections in a single DARAM or separate SARAM memory spaces. Local variable space is allocated on the stack. It is possible that there may be conflicts when global variables, whose allocation is in .bss section, are accessed within the same instruction as a locally declared variable.

- ❑ Use the `DATA_SECTION` pragma: If an algorithm uses a set of coefficients that is applied to a known data array, use the `DATA_SECTION` pragma to place these variables in their own named section. Then explicitly allocate these sections in separate memory blocks to avoid conflicts. Example 3–31 shows sample C source for using the `DATA_SECTION` pragma to place variables in a user defined section.

Most of the preceding memory allocation recommendations are based on the assumption that the typical operation accesses at most two operands. Table 3–9 shows the possible operand combinations.

Example 3–31. Declaration Using DATA_SECTION Pragma

```
#pragma DATA_SECTION(h, ”coeffs”)
short h[10];
#pragma DATA_SECTION(x, ”mydata”)
short x[10];
```

Table 3–9. Possible Operand Combinations

Operand 1	Operand 2	Comment
Local var (stack)	Local var (stack)	If stack is in DARAM then no memory conflict will occur
Local var(stack)	Global var(.bss)	If stack is in separate SARAM block or is in same DARAM block, then no conflict will occur
Local var(stack)	Const symbol (.const)	If .const is located in separate SARAM or same DA-RAM no conflict will occur
Global var(.bss)	Global var(.bss)	If .bss is allocated in DARAM, then no conflict will occur
Global var(.bss)	Const symbol(.const)	If .const and .bss are located in separate SARAM or same DARAM block, then no conflict will occur

When compiling with the small memory model (compiler default) allocate all data sections, .data, .bss, .stack, .sysmem, .sysstack, .cio, and .const, on the first 64K word page of memory (Page 0).

Example 3–32 contains a sample linker command file for the small memory model. For extensive documentation on the linker and linker command files, see the *TMS320C55x Assembly Language Tools User’s Guide* (SPRU280).

Example 3–32. Sample Linker Command File

```
/**********************************************************
         LINKER command file for LEAD3 memory map.
         Small memory model
**********************************************************/
-stack    0x2000      /* Primary stack size   */
-sysstack 0x1000      /* Secondary stack size */
-heap     0x2000      /* Heap area size       */
-c                /* Use C linking conventions: auto-init vars at runtime */
-u _Reset         /* Force load of reset interrupt handler                */
MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */
  RAM  (RWIX) : origin = 0x000100, length = 0x01ff00 /* 128Kb page of RAM */
  ROM  (RIX)  : origin = 0x020100, length = 0x01ff00 /* 128Kb page of ROM */
  VECS (RIX)  : origin = 0xffff00, length = 0x000100 /*256-byte int vector*/
 PAGE 1:  /* -------- 64K-word I/O Address Space -------- */
  IOPORT (RWI) : origin = 0x000000, length = 0x020000
}

SECTIONS
{
   .text     > ROM    PAGE 0  /* Code                          */
   /* These sections must be on same physical memory page   */
   /* when small memory model is used                        */
   .data     > RAM    PAGE 0  /* Initialized vars              */
   .bss      > RAM    PAGE 0  /* Global & static vars          */
   .const    > RAM    PAGE 0  /* Constant data                 */
   .sysmem   > RAM    PAGE 0  /* Dynamic memory (malloc)       */
   .stack    > RAM    PAGE 0  /* Primary system stack          */
   .sysstack > RAM    PAGE 0  /* Secondary system stack        */
   .cio      > RAM    PAGE 0  /* C I/O buffers                 */
   /* These sections may be on any physical memory page      */
   /* when small memory model is used                        */
   .switch   > RAM    PAGE 0  /* Switch statement tables       */
   .cinit    > RAM    PAGE 0  /* Auto-initialization tables */
   .pinit    > RAM    PAGE 0  /* Initialization fn tables      */
    vectors  > VECS   PAGE 0  /* Interrupt vectors             */
   .ioport   > IOPORT PAGE 1  /* Global & static IO vars       */
```

3.5.5 Allocating Function Code to Different Sections

The compiler provides a pragma to allow the placement of a function's code into a separate user defined section. The pragma is useful if it is necessary to have some granularity in the placement of code in memory.

The pragma, in Example 3–33, defines a new section called .myfunc. The code for the function `myfunction`() will be placed by the compiler into this newly defined section. The section name can then be used within the SECTIONS directive of a linker command file to explicitly allocate memory for this function. For details on how to use the SECTIONS directive, see the *TMS320C55x Assembly Language Tools User's Guide* (SPRU280).

Example 3–33. Allocation of Functions Using `CODE_SECTION` Pragma

```
#pragma CODE_SECTION(myfunction, ".myfunc")
void myfunction(void)
{
    .
    .
}
```

Chapter 4

Optimizing Assembly Code

This chapter offers recommendations for producing TMS320C55x (C55x) assembly code that:

- Makes good use of special architectural features, like the dual multiply-and-accumulate (MAC) hardware, parallelism, and looping hardware.
- Produces no pipeline conflicts, memory conflicts, or instruction-fetch stalls that would delay CPU operations.

This chapter shows ways you can optimize TMS320C55x assembly code, so that you have highly-efficient code in time-critical portions of your programs.

Topic **Page**

4.1 Efficient Use of the Dual-MAC Hardware

Two of the most common operations in digital signal processing are the Multiply and Accumulate (MAC) and the Multiply and Substract (MAS). The C55x architecture can implement two multiply/accumulate (or two multiply/substract) operations in one cycle as shown in the typical C55x dual-MAC instruction below:

```
MAC *AR2+, *CDP+, AC0
:: MAC*AR3+, *CDP+, AC1
```

that performs

AC0 = AC0 + xmem * cmem

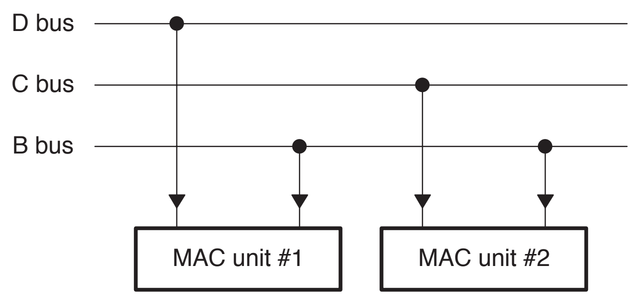
AC1 = AC1 + ymem * cmem

where xmem, ymem, and cmem are operands in memory pointed by registers AR2, AR3, and CDP, respectively. Notice the following characteristics of C55x dual-MAC instructions:

1) **The dual-MAC/MAS operation can be performed using three operands only**, which implies that one of the operands (cmem) should be common to both MAC/MAS operations.

   The two MAC units on the C55x DSP are economically fed data via three independent data buses: BB (the B bus), CB (the C bus), and DB (the D bus). During a dual-MAC operation, each MAC unit requires two data operands from memory (four operands total). However, the three data buses are capable of providing at most three independent operands. To obtain the required fourth operand, the data value on the B bus is used by both MAC units. This is illustrated in Figure 4–1. With this structure, the fourth data operand is not independent, but rather is dependent on one of the other three operands.

Figure 4–1. Data Bus Usage During a Dual-MAC Operation



In the most general case of two multiplications, one would expect a requirement of four fully independent data operands. While this is true on the surface, in most cases one can get by with only three independent operands and avoid degrading performance by specially structuring the DSP code at either the algorithm or application level. The special structuring, covered in sections 4.1.1 through 4.1.4, can be categorized as follows:

- ❑ Implicit algorithm symmetry (e.g., symmetric FIR, complex vector multiply)
- ❑ Loop unrolling (e.g., block FIR, single-sample FIR, matrix multiply)
- ❑ Multi-channel applications
- ❑ Multi-algorithm applications

2) **The common operand (cmem) has to be addressed using XCDP (Coefficient Data Pointer) and should be kept in internal memory** since the bus used to fetch this operand (B bus) is not connected to external memory. For xmem and ymem operands, any of the eight auxiliary registers (XAR0–XAR7) can be used.

3) **In order to perform a dual-MAC/MAS operation in one cycle, the common operand (cmem) should not be residing in the same memory block with respect to the other two operands,** as the maximum bandwidth of a C55x memory block is two accesses in one cycle (internal DARAM block).

   If the cmem, xmem, and ymem operands point to the same data (for example, during the autocorrelation of a signal), one option is to temporarily copy the cmem data to a different block.

   The programmer should make the appropriate decision whether the dual-MAC cycle savings compensate the extra cycles and extra memory required by the data-copy process. If many functions need extra buffers in order to use the dual-MAC, then an excess amount of data memory can be consumed. One possibility for alleviating this problem is to allocate one buffer (of maximum required size) and use it commonly across all the functions.

4.1.1 Implicit Algorithm Symmetry

When an algorithm has internal symmetry, it can sometimes be exploited for efficient dual-MAC implementation. One such example is a symmetric FIR filter. This filter has coefficients that are symmetrical with respect to delayed values of the input signal. The mathematical expression for a symmetric FIR filter can be described by the following discrete-time difference equation:

$$y(k) = \sum_{j=0}^{\frac{N}{2}-1} a_j \left[x(k-j) + x(k+j-N+1)\right]$$

where

N = Number of filter taps (even)
x() = Element in the vector of input values
y() = Element in the vector of output values
k = Time index

Similar in form to the symmetrical FIR filter is the anti-symmetrical FIR filter:

$$y(k) = \sum_{j=0}^{\frac{N}{2}-1} a_j \left[x(k-j) - x(k+j-N+1)\right]$$

Both the symmetrical and anti-symmetrical FIR filters can be implemented using a dual-MAC approach because only three data values need be fetched per inner loop cycle: a_j, x(k – j), and x(k + j – N + 1). The coefficient a_j is delivered to the dual-MAC units using the B bus and using XCDP as the pointer. The C bus and the D bus are used along with two XARx registers to access the independent elements x(k – j) and x(k + j – N + 1).

A second example of an algorithm with implicit symmetry is an element-by-element complex vector multiply. Let {A}, {B}, and {C} be vectors of length N, and let j be the imaginary unit value (i.e., square root of –1). The complex components of {A}, {B}, and {C} can be expressed as

$$a_i = a_i^{RE} + ja_i^{IM} \quad b_i = b_i^{RE} + jb_i^{IM} \quad c_i = c_i^{RE} + jc_i^{IM} \quad \text{for } 1 \le i \le N$$

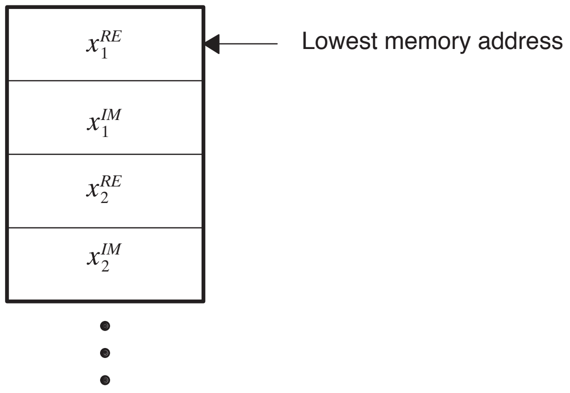
and the expression for each element in {C} is computed as

$$\begin{aligned} c_i &= a_i * b_i \\ &= \left(a_i^{RE} + ja_i^{IM}\right) * \left(b_i^{RE} + jb_i^{IM}\right) \\ &= \left(a_i^{RE} b_i^{RE} - a_i^{IM} b_i^{IM}\right) + j\left(a_i^{RE} b_i^{IM} + a_i^{IM} b_i^{RE}\right) \qquad \text{for } 1 \le i \le N \end{aligned}$$

The required four multiplications in the above expression can be implemented with two dual-MAC instructions by grouping the multiplications as follows:

- ❑ 1st multiplication group: $a_i^{RE} b_i^{RE}$ and $a_i^{IM} b_i^{RE}$
- ❑ 2nd multiplication group: $a_i^{IM} b_i^{IM}$ and $a_i^{RE} b_i^{IM}$

Each dual-multiply grouping requires only three independent operands. An assembly code example for the complex vector multiply is given in Example 4–1 (showing mnemonic instructions). Note that this particular code assumes the following arrangement in memory for a complex vector:



In addition, the code stores both portions of the complex result to memory at the same time. This requires that the results vector be long-word aligned in memory. One way to achieve this is through use of the alignment flag option with the .bss directive, as was done with this code example. Alternatively, one could place the results array in a separate uninitialized named section using a .usect directive, and then use the linker command file to force long-word alignment of that section.

Example 4–1. Complex Vector Multiplication Code

```
N      .set   3                    ; Length of each complex vector

       .data
A      .int   1,2,3,4,5,6          ; Complex input vector #1
B      .int   7,8,9,10,11,12       ; Complex input vector #2

;Results are: 0xfff7, 0x0016, 0xfff3, 0x0042, 0xffef, 0x007e

       .bss C, 2*N, ,1             ; Results vector, long-word aligned

       .text
       BCLR ARMS                   ; Clear ARMS bit (select DSP mode)
       .arms_off                   ; Tell assembler ARMS = 0

cplxmul:
       AMOV #A, XAR0               ; Pointer to A vector
       AMOV #B, XCDP               ; Pointer to B vector
       AMOV #C, XAR1               ; Pointer to C vector
       MOV #(N-1), BRC0            ; Load loop counter
       MOV #1, T0                  ; Pointer offset
       MOV #2, T1                  ; Pointer increment

       RPTBLOCAL endloop           ; Start the loop

       MPY *AR0, *CDP+, AC0
       :: MPY *AR0(T0), *CDP+, AC1

       MAS *AR0(T0), *CDP+, AC0
       :: MAC *(AR0+T1), *CDP+, AC1

endloop:
       MOV pair(LO(AC0)), dbl(*AR1+) ; Store complex result
                                   ; End of loop
```

Note: The algebraic instructions code example for Complex Vector Multiplication is shown in Example B–14 on page B-11.

4.1.2 Loop Unrolling

Loop unrolling involves structuring computations to exploit the reuse of data among different time or geometric iterations of the algorithm. Many algorithms can be structured computationally to provide for such reuse and allow a dual-MAC implementation.

In filtering, input and/or output data is commonly stored in a delay chain buffer. Each time the filter is invoked on a new data point, the oldest value in the delay chain is discarded from the bottom of the chain, while the new data value is added to the top of the chain. A value in the chain will get reused (for example, multiplied by a coefficient) in the computations over and over again as successive time-step outputs are computed. The reuse will continue until such a time that the data value becomes the oldest value in the chain and is discarded. Dual-MAC implementation of filtering should therefore employ a time-based loop unrolling approach to exploit the reuse of the data. This scenario is presented in sections 4.1.2.1 and 4.1.2.2.

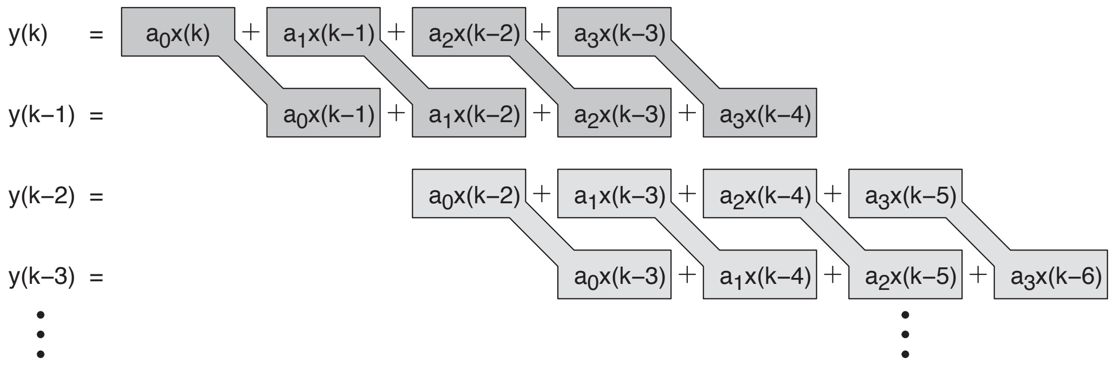
An application amenable to geometric based loop unrolling is matrix computations. In this application, successive rows in a matrix get multiplied and accumulated with the columns in another matrix. In order to obtain data reuse within the loop kernel, the computations using two different rows of data should be handled in parallel. This will be presented in section 4.1.2.3.

4.1.2.1 Temporal Loop Unrolling: Block FIR Filter

To efficiently implement a block FIR filter with the two MAC units, loop unrolling must be applied so that two time-based iterations of the algorithm are computed in parallel. This allows reuse of the coefficients.

Figure 4–2 illustrates the coefficient reuse for a 4-tap block FIR filter with constant, real-value coefficients. The implementation computes two sequential filter outputs in parallel so that only a single coefficient, a_i, is used by both MAC units. Consider, for example, the computation of outputs y(k) and y(k – 1). For the first term in each of these two rows, one MAC unit computes a_0x(k), while the second MAC unit computes a_0x(k – 1). These two computations combined require only three different values from memory: a_0, x(k), and x(k – 1). Proceeding to the second term in each row, a_1x(k – 1) and a_1x(k – 2) are computed similarly, and so on with the remaining terms. After fully computing the outputs y(k) and y(k – 1), the next two outputs, y(k – 2) and y(k – 3), are computed in parallel. Again, the computation begins with the first two terms in each of these rows. In this way, DSP performance is maintained at two MAC operations per clock cycle.

Figure 4–2. Computation Groupings for a Block FIR (4-Tap Filter Shown)


y(k) =
a0x(k) + a1x(k–1) + a2x(k–2) + a3x(k–3)
y(k–1) =
a0x(k–1) + a1x(k–2) + a2x(k–3) + a3x(k–4)
y(k–2) =
a0x(k–2) + a1x(k–3) + a2x(k–4) + a3x(k–5)
y(k–3) =
a0x(k–3) + a1x(k–4) + a2x(k–5) + a3x(k–6)

Note that filters with either an even or odd number of taps are handled equally well by this method. However, this approach does require one to compute an even number of outputs y(). In cases where an odd number of outputs is desired, one can always zero-pad the input vector x() with one additional zero element, and then discard the corresponding additional output.

Note also that not all of the input data must be available in advance. Rather, only two new input samples are required for each iteration through the algorithm, thereby producing two new output values.

A non-optimized assembly code example for the block FIR filter is shown in Example 4–2 (showing mnemonic instructions). An optimized version of the same code is found in Example 4–3 (showing mnemonic instructions). The following optimizations have been made in Example 4–3:

- The first filter tap was peeled out of the inner loop and implemented using a dual-multiply instruction (as opposed to a dual-multiply-and-accumulate instruction). This eliminated the need to clear AC0 and AC1 prior to entering the inner loop each time.

- The last filter tap was peeled out of the inner loop. This allows for the use of different pointer adjustments than in the inner loop, and eliminates the need to explicitly rewind the CDP, AR0, and AR1 pointers.

  The combination of these first two optimizations results in a requirement that N_TAPS be a minimum of 3.

- Both results are now written to memory at the same time using a double store instruction. Note that this requires the results array (OUT_DATA) to be long-word aligned. One way to achieve this is through use of the alignment flag option with the .bss directive, as was done in this code example. As an alternative, you could place the results array in a separate uninitialized named section using a .usect directive, and then use the linker command file to force long-word alignment of that section.

- The outer loop start instruction, RPTBLOCAL, has been put in parallel with the instruction that preceded it.

Example 4–2. Block FIR Filter Code (Not Optimized)

```
N_TAPS    .set  4                           ; Number of filter taps
N_DATA    .set  11                          ; Number of input values

          .data
COEFFS    .int   1,2,3,4                    ; Coefficients
IN_DATA   .int   1,2,3,4,5,6,7,8,9,10,11    ; Input vector

;Results are:    0x0014, 0x001E, 0x0028, 0x0032,
;                0x003C, 0x0046, 0x0050, 0x005A

          .bss   OUT_DATA, N_DATA - N_TAPS + 1 ; Output vector

          .text
          BCLR ARMS                ; Clear ARMS bit (select DSP mode)
          .arms_off                ; Tell assembler ARMS = 0

bfir:
          AMOV #COEFFS, XCDP                ; Pointer to coefficient array
          AMOV #(IN_DATA + N_TAPS - 1), XAR0   ; Pointer to input vector
          AMOV #(IN_DATA + N_TAPS), XAR1    ; 2nd pointer to input vector
          AMOV #OUT_DATA, XAR2              ; Pointer to output vector
          MOV  #((N_DATA - N_TAPS + 1)/2 - 1), BRC0
                                            ; Load outer loop counter
          MOV  #(N_TAPS - 1), CSR           ; Load inner loop counter

          RPTBLOCAL endloop                 ; Start the outer loop

          MOV #0, AC0                       ; Clear AC0
          MOV #0, AC1                       ; Clear AC1

          RPT CSR                           ; Start the inner loop
          MAC *AR0-, *CDP+, AC0             ; All taps
          :: MAC *AR1-, *CDP+, AC1

          MOV AC0, *AR2+                    ; Write 1st result
          MOV AC1, *AR2+                    ; Write 2nd result
          MOV #COEFFS, CDP                  ; Rewind coefficient pointer
          ADD #(N_TAPS + 2), AR0            ; Adjust 1st input vector
                                            ;   pointer
endloop:
          ADD #(N_TAPS + 2), AR1            ; Adjust 2nd input vector
                                            ;   pointer
                                            ; End of outer loop
```

Note: The algebraic instructions code example for Block FIR Filter Code (Not Optimized) is shown in Example B–15 on page B-12.

Example 4–3. Block FIR Filter Code (Optimized)

```
N_TAPS    .set   4                            ; Number of filter taps
N_DATA    .set   11                           ; Number of input values

          .data
COEFFS    .int   1,2,3,4                      ; Coefficients
IN_DATA   .int   1,2,3,4,5,6,7,8,9,10,11      ; Input vector

;Results are:    0x0014, 0x001E, 0x0028, 0x0032,
;                0x003C, 0x0046, 0x0050, 0x005A

          .bss   OUT_DATA, N_DATA - N_TAPS + 1, ,1
                                              ; Output vector, long word
aligned

          .text
          BCLR ARMS                     ; Clear ARMS bit (select DSP mode)
          .arms_off                     ; Tell assembler ARMS = 0

bfir:
          AMOV #COEFFS, XCDP                  ; Pointer to coefficient array
          AMOV #(IN_DATA + N_TAPS - 1), XAR0  ; Pointer to input vector
          AMOV #(IN_DATA + N_TAPS), XAR1      ; 2nd pointer to input vector
          AMOV #OUT_DATA, XAR2                ; Pointer to output vector
          MOV  #((N_DATA - N_TAPS + 1)/2 - 1), BRC0
                                              ; Load outer loop counter
          MOV  #(N_TAPS - 3), CSR             ; Load inner loop counter
          MOV  #(-(N_TAPS - 1)), T0           ; CDP rewind increment

          MOV  #(N_TAPS + 1), T1              ; ARx rewind increment
          ||RPTBLOCAL endloop                 ; Start the outer loop

          MPY *AR0-, *CDP+, AC0               ; 1st tap
          :: MPY *AR1-, *CDP+, AC1

          RPT CSR                             ; Start the inner loop
          MAC *AR0-, *CDP+, AC0               ; Inner taps
          :: MAC *AR1-, *CDP+, AC1

          MAC *(AR0+T1), *(CDP+T0), AC0       ; Last tap
          :: MAC *(AR1+T1), *(CDP+T0), AC1

endloop:
          MOV pair(LO(AC0)), dbl(*AR2+)       ; Store both results
                                              ; End of outer loop
```

Note: The algebraic instructions code example for Block FIR Filter Code (Optimized) is shown in Example B–16 on page B-13.

4.1.2.2 Temporal Loop Unrolling: Single-Sample FIR Filter

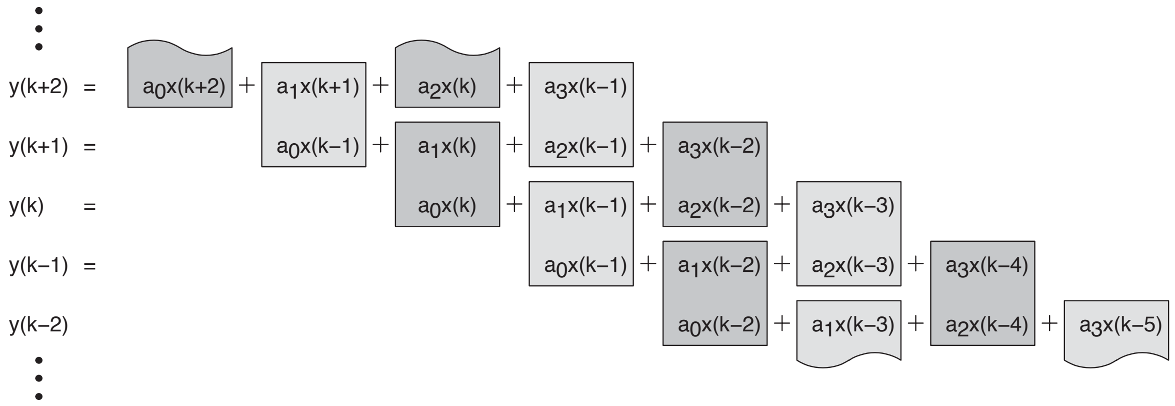
The temporally unrolled block FIR filter described in section 4.1.2.1 maintains dual-MAC throughput by sharing a common coefficient between the two MAC units. In some algorithms, the loop unrolling needs to be performed so that a common data variable is shared instead. The single-sample FIR filter is an example of such an algorithm. In the single-sample FIR filter, the calculations for the current sample period are interlaced with those of the next sample period in order to achieve a net performance of two MAC operations per cycle.

Figure 4–3 shows the needed computation groupings for a 4-tap FIR filter. At any given time step, one multiplies and accumulates every other partial product in the corresponding row, beginning with the first partial product in the row. In addition, one also multiplies and accumulates every other term in the next row (that is, the row above the current row) in advance of that time step, beginning with the second partial product in the next row. In this way, each row is fully computed over the course of two sample periods.

For example, at time step k, it is desired to compute y(k). The first term in the y(k) row is $a_0x(k)$, which is computed using one of the two MAC units. In addition, the second MAC unit is used to compute the second term in the y(k+1) row, $a_1x(k)$, in advance of time step k + 1. These two computations combined require only three different values from memory: a_0, a_1, and x(k). Note that the term x(k) is not available until time k. This is why calculations at each time step must begin with the first term in the corresponding row.

The second term in the y(k) row is $a_1x(k – 1)$. However, this would have been already computed during the first computation at time step k – 1 (similar to how $a_1x(k)$ was just pre-computed for time step k+1) , so it can be skipped here. The third term in the y(k) row, $a_2x(k – 2)$, is computed next, and at the same time, the term $a_3x(k – 2)$ is computed in the y(k + 1) row in advance of time step k+1.

Figure 4–3. Computation Groupings for a Single-Sample FIR With an Even Number of TAPS (4-Tap Filter Shown)


y(k+2) = a0x(k+2) + a1x(k+1) + a2x(k) + a3x(k–1)
y(k+1) = a0x(k–1) + a1x(k) + a2x(k–1) + a3x(k–2)
y(k) = a0x(k) + a1x(k–1) + a2x(k–2) + a3x(k–3)
y(k–1) = a0x(k–1) + a1x(k–2) + a2x(k–3) + a3x(k–4)
y(k–2) a0x(k–2) + a1x(k–3) + a2x(k–4) + a3x(k–5)

Notice that two separate running sums are maintained, one with partial products for the current time step, the other with pre-computed terms for the next time step. At the next time step, the pre-computed running sum becomes the current running sum, and a new pre-computed running sum is started from zero. At the end of each sample period, the current running sum contains the current filter output, which can be dispatched as required by the application.

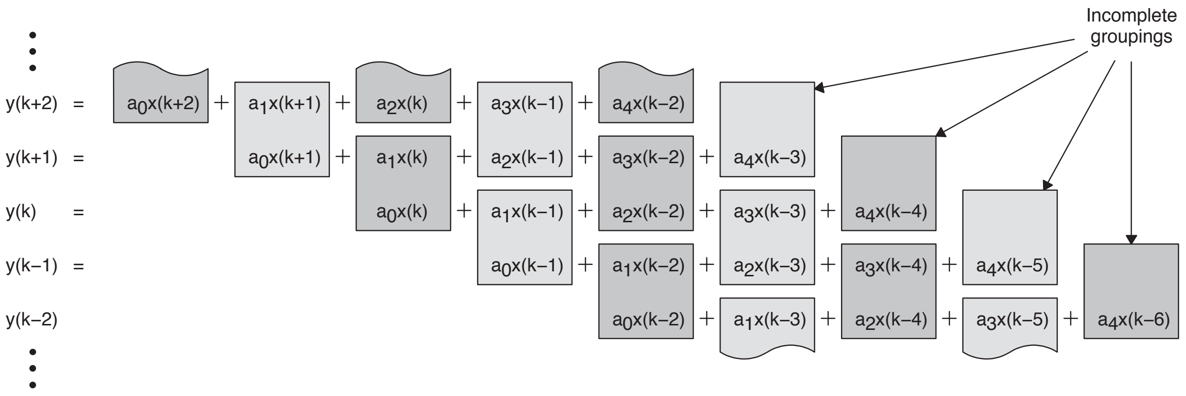
The above approach is not limited to the 4-tap filter illustrated in Figure 4–3. Any other filter with an even number of taps is a straightforward extension. For filters with an odd number of taps, the computation groupings become problematic, in that the last grouping in each row is missing the pre-calculation term in the row above it.

Figure 4–4 depicts this problem for a 5-tap filter. To overcome this problem, one should pad the filter to the next higher even number of taps by using a zero coefficient for the additional term. For example, the five tap filter is augmented to

$$y(k) = a_0x(k) + a_1x(k-1) + a_2x(k-2) + a_3x(k-3) + a_4x(k-4) + 0[x(k-5)]$$

In this way, any filter with an odd number of taps can be implemented as a filter with an even number of taps but retain the frequency response of the original odd-number-tap filter.

Figure 4–4. Computation Groupings for a Single-Sample FIR With an Odd Number of TAPS (5-Tap Filter Shown)


Incomplete groupings
y(k+2) =
y(k+1) =
y(k) =
y(k–1) =
y(k–2)
a0x(k+2) + a1x(k+1) + a2x(k) + a3x(k–1) + a4x(k–2)
a0x(k+1) + a1x(k) + a2x(k–1) + a3x(k–2) + a4x(k–3)
a0x(k) + a1x(k–1) + a2x(k–2) + a3x(k–3) + a4x(k–4)
a0x(k–1) + a1x(k–2) + a2x(k–3) + a3x(k–4) + a4x(k–5)
a0x(k–2) + a1x(k–3) + a2x(k–4) + a3x(k–5) + a4x(k–6)

4.1.2.3 Geometric Loop Unrolling: Matrix Mathematics

Matrix mathematics typically involves considerable data reuse. Consider the general case of multiplying two matrices:

$$[C] = [A] \times [B]$$

where

[A] = m × n matrix
[B] = n × p matrix
[C] = m × p matrix
m ≥ 1, n ≥ 1, p ≥ 1

The expression for each element in matrix C is given by:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \qquad (1 \le i \le m,\ 1 \le j \le p)$$

where the conventional notation $x_{i,j}$ is being used to represent the element of matrix X in the ith row and jth column. There are basically two different options for efficient dual-MAC implementation. First, one could compute $c_{i,j}$ and $c_{i,j+1}$ in parallel. The computations made are:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \qquad c_{i,j+1} = \sum_{k=1}^{n} a_{i,k} b_{k,j+1}$$

The element $a_{i,k}$ is common to both expressions. The computations can therefore be made in parallel, with the common data $a_{i,k}$ delivered to the dual-MAC units using the B bus and using XCDP as the pointer. The C bus and the D bus are used along with two XARx registers to access the independent elements $b_{k,j}$ and $b_{k,j+1}$.

Alternatively, one could compute $c_{i,j}$ and $c_{i+1,j}$ in parallel. The computations made are then:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}\, b_{k,j} \qquad\qquad c_{i+1,j} = \sum_{k=1}^{n} a_{i+1,k}\, b_{k,j}$$

In this case, the element $b_{k,j}$ is common to both expressions. They can therefore be made in parallel, with the common data $b_{k,j}$ delivered to the dual-MAC units using the B bus and using XCDP as the pointer. The C bus and the D bus are used along with two XARx registers to access the independent elements $a_{i,k}$ and $a_{i+1,k}$.

The values of m and p determine which approach one should take. Because the inner loop will compute two elements in matrix C each iteration, clearly it is most efficient if an even number of elements can be computed. Therefore, if p is even, one should implement the first approach: compute $c_{i,j}$ and $c_{i,j+1}$ in parallel. Alternatively, if m is even, the second approach is more efficient: compute $c_{i,j}$ and $c_{i+1,j}$ in parallel. If both m and p are even, either approach is appropriate. Finally, if neither m nor p is even, there will be an extra element c that will need to be computed individually each time through the inner loop. One could add additional single-MAC code to handle the final element in the inner loop. Alternatively, one could pad either matrix A or matrix B with a row or column or zeros (as appropriate) to make either m or p even. The elements in matrix C computed using the pad row or column should then be discarded after computation.

4.1.3 Multichannel Applications

In multichannel applications, the same signal processing is often done on two or more independent data streams. Depending on the specific type of processing being performed, it may be possible to process two channels of data in parallel, one channel in each MAC unit. In this way, a common set of constant coefficients can be shared.

An application example readily amenable to this approach is non-adaptive filtering where the same filter is being applied to two different data streams. Both channels are processed in parallel, one channel in each of the two MAC units. For example, the same FIR filter applied to two different data streams can be represented mathematically by the following expressions:

$$y_1(k) = \sum_{j=0}^{N-1} a_j x_1(k-j) \qquad\qquad y_2(k) = \sum_{j=0}^{N-1} a_j x_2(k-j)$$

where

N = Number of filter taps
a_j = Element in the coefficient array
$x_i()$ = Element in the ith vector of input values
$y_i()$ = Element in the ith vector of output values
k = Time index

The value a_j is common to both calculations. The two calculations can therefore be performed in parallel, with the common a_j delivered to the dual-MAC units using the B bus and using XCDP as the pointer. The C bus and the D bus are used along with two XARx registers to access the independent input elements $x_1(k - j)$ and $x_2(k - j)$.

A second example is the correlation computation between multiple incoming data streams and a fixed data vector. Suppose it is desired to compute the correlation between the vectors X_1 and Y, and also between the vectors X_2 and Y. One would need to compute the following for each element in the correlation vectors R_1 and R_2:

$$r_{x_1 y}(j) = \sum_k x_1(k + j)\, y(k) \qquad r_{x_2 y}(j) = \sum_k x_2(k + j)\, y(k)$$

The element y(k) is common to both calculations. The two calculations can therefore be performed in parallel, with the common data y(k) delivered to the dual-MAC units via the B bus with XCDP as the address pointer. The C bus and the D bus are used along with two XARx registers to access the independent elements $x_1(k + j)$ and $x_2(k + j)$.

4.1.4 Multi-Algorithm Applications

When two or more different processing algorithms are applied to the same data stream, it may be possible to process two such algorithms in parallel. For example, consider a statistical application that computes the autocorrelation of a vector X, and also the correlation between vector X and vector Y. One would need to compute the following for each element in the correlation vectors R_{xx} and R_{xy}:

$$r_{xx}(j) = \sum_k x(k + j)\, x(k) \qquad r_{xy}(j) = \sum_k x(k + j)\, y(k)$$

The element x(k + j) is common to both calculations. The two calculations can therefore be made in parallel, with the common data x(k + j) delivered to the dual-MAC units using the B bus and using XCDP as the pointer. The C bus and the D bus are used along with two XARx registers to access the independent elements x(k) and y(k).

4.2 Using Parallel Execution Features

The C55x architecture allows programmers to place two operations or instructions in parallel to reduce the total execution time. There are two types of parallelism: Built-in parallelism within a single instruction and user-defined parallelism between two instructions. Built-in parallelism (see section 4.2.1) is automatic; as soon as you write the instruction, it is put in place. User-defined parallelism is optional and requires decision-making. Sections 4.2.2 through 4.2.8 present the rules and restrictions associated with the use of user-defined parallelism, and give examples of using it.

4.2.1 Built-In Parallelism

Instructions that have built-in parallelism perform two different operations in parallel. In the algebraic syntax, they can be identified by the comma that separates the two operations, as in the following example:

```
AC0 = *AR0 * coef(*CDP),      ; The data referenced by AR0 is multiplied by
AC1 = *AR1 * coef(*CDP)       ; a coefficient referenced by CDP. At the same time
                              ; the data referenced by AR1 is multiplied by the
                              ; same coefficient.
```

In the mnemonic syntax, they can be identified by a double colon (::) that separates the two operations. The preceding example in the mnemonic syntax is:

```
MPY *AR0, *CDP, AC0           ; The data referenced by AR0 is multiplied by
:: MPY *AR1, *CDP, AC1        ; a coefficient referenced by CDP. At the same time
                              ; the data referenced by AR1 is multiplied by the
                              ; same coefficient.
```

4.2.2 User-Defined Parallelism

Two instructions may be placed in parallel to have them both execute in a single cycle. The two instructions are separated by the II separator. One of the two instructions may have built-in parallelism. The following algebraic code example shows a user-defined parallel instruction pair. One of the instructions in the pair also features built-in parallelism.

```
AC0 = AC0 + (*AR3+ * coef(*CDP+)),   ; 1st instruction (has built-in parallelism)
AC1 = AC1 + (*AR4+ * coef(*CDP+))
|| repeat(CSR)                       ; 2nd instruction
```

The equivalent mnemonic code example is:

```
MPY *AR3+, *CDP+, AC0                ; 1st instruction (has built-in parallelism)
:: MPY *AR4+, *CDP+, AC1
|| RPT CSR                           ; 2nd instruction
```

4.2.3 Architectural Features Supporting Parallelism

The C55x architecture provides three main, independent computation units that are controlled by the instruction buffer unit (I unit):

- Program flow unit (P unit)
- Address-data flow Unit (A unit)
- Data computation unit (D unit)

The C55x instructions make use of dedicated operative resources (or operators) within each of the units. In total, there are 14 operators available across the three computation units, and the parallelism rules enable the use of two independent operators in parallel within the same cycle. If all other rules are observed, two instructions that independently use any two of the independent operators may be placed in parallel.

Figure 4–5 shows a matrix that reflects the 14 operators mentioned and the possible operator combinations that may be used in placing instructions in parallel. The operators are ordered from rows 1 through 14 as well as columns 1 though 14. A blank cell in any given position (row I, column J) in the matrix indicates that operator I may be placed in parallel with operator J, and an X in any given position indicates that the two operators cannot be placed in parallel. For example, a D-Unit MAC operation (row 7) may be placed in parallel with a P-Unit Load operation (column 13) but cannot be placed in parallel with a D-Unit ALU operation (column 5).

Figure 4–5. Matrix to Find Operators That Can Be Used in Parallel

		A-unit ALU	A-unit Swap	A-unit Load	A-unit Store	D-unit ALU	D-unit Shifter	D-unit MAC	D-unit Load	D-unit Store	D-unit Shift, Store	D-unit Swap	P-unit Control	P-unit Load	P-unit Store
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
A-unit ALU	1	X													
A-unit Swap	2		X												
A-unit Load	3														
A-unit Store	4														
D-unit ALU	5					X	X	X							
D-unit Shifter	6					X	X	X			X				
D-unit MAC	7					X	X	X							
D-unit Load	8														
D-unit Store	9														
D-unit Shift, Store	10						X				X				
D-unit Swap	11											X			
P-unit Control	12												X		
P-unit Load	13														
P-unit Store	14														

Note: X in a table cell indicates that the operator in that row and the operator in that column cannot be used in parallel with each other. A blank table cell indicates that the operators can be used in parallel.

Bus resources also play an important part in determining whether two instructions may be placed in parallel. Typically, programmers should be concerned with the data buses and the constant buses. Table 4–1 lists and describes the main CPU buses of interest and gives examples of instructions that use the different buses. These may also be seen pictorially in Figure 4–6. Figure 4–6 also shows all CPU buses and the registers/operators in each of the three functional units.

Table 4–1. CPU Data Buses and Constant Buses

Bus Type	Bus(es)	Description of Bus(es)	Example: Instruction That Uses The Bus(es)

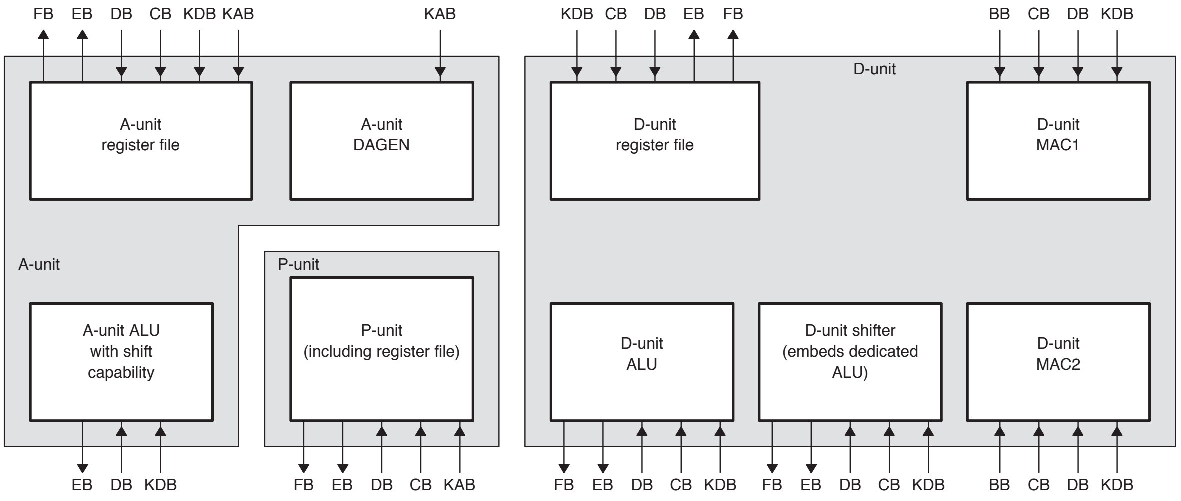
Data	BB	Special coefficient read bus	`MPY *AR1+, *CDP+, AC0`
			`:: MPY *AR3+, *CDP+, AC1`
			The operand referenced by CDP is carried to the CPU on BB.

	CB, DB	Data-read buses	`MOV *AR3+, AC0` The operand referenced by AR3 is carried to the low half of AC0 on DB.
	EB, FB	Data-write buses	`MOV AC0, *AR3` The low half of AC0 is carried on EB to the location referenced by AR3.

Constant	KAB	Constant bus used in the address phase of the pipeline to carry addresses:
		❑ The P unit uses KAB to generate program-memory addresses.
		❑ The A unit uses KAB to generate data-memory addresses.
			P-unit use:
			`B #Routine2`
			The constant Routine2 is carried on KPB to the P unit, where it is used for a program-memory address.
			A-unit use:
			`MOV *SP(#7), BRC0`
			The immediate offset (7) is carried to the data-address generation unit (DAGEN) on KAB.

	KDB	Constant bus used by the A unit or D unit for computations. This bus is used in the execute phase of the instruction pipeline.	`ADD #1234h, AC0` The constant 1234h is carried on KDB to the D-unit ALU, where it is used in the addition.

Figure 4–6. CPU Operators and Buses



4.2.4 User-Defined Parallelism Rules

This section describes the rules that a programmer must follow to place two instructions in parallel. It is essential to note here that all the rules must be observed for the parallelism to be valid. However, this section begins with a set of four basic rules (Table 4–2) that a programmer may use when implementing user-defined parallelism. If these are not sufficient, the set of advanced rules (Table 4–3) needs to be considered.

Table 4–2. Basic Parallelism Rules

Consideration	Rule
Hardware resource conflicts	Two instructions in parallel cannot compete for operators (see Figure 4–5, page 4-18) or buses (see Table 4–1, page 4-19).
Maximum instruction length	The combined length of the instruction pair cannot exceed 6 bytes.

Parallel enable bit
OR
Soft dual encoding
	If either of the following cases is true, the instructions can be placed in parallel:
	❑ **Parallel enable bit is present:** At least one of two instructions in parallel must have a parallel enable bit in its instruction code. The instruction set reference guides (see *Related Documentation from Texas Instruments* in the preface) indicate whether a given instruction has a parallel enable bit.
	❑ **Soft dual encoding is present:** For parallel instructions that use Smem or Lmem operands, each instruction must use one of the indirect operands allowed for the dual AR indirect addressing mode:
	`*ARn`
	`*ARn+`
	`*ARn–`
	`*(ARn + T0)   (Available if C54CM bit = 0)`
	`*(ARn + AR0)  (Available if C54CM bit = 1)`
	`*(ARn – T0)   (Available if C54CM bit = 0)`
	`*(ARn – AR0)  (Available if C54CM bit = 1)`
	`*ARn(T0)      (Available if C54CM bit = 0)`
	`*ARn(AR0)     (Available if C54CM bit = 1)`
	`*(ARn + T1)`
	`*(ARn – T1)`

Table 4–3. Advanced Parallelism Rules

Consideration	Rule

Byte extensions for constants	An instruction that uses one of the following addressing-mode operands cannot be placed in parallel with another instruction. The constant (following the # symbol) adds 2 or 3 bytes to the instruction.
	`*abs16(#k16)`
	`*port(#k16) (algebraic syntax)`
	`port(#k16) (mnemonic syntax)`
	`*(#k23)`
	`*ARn(#K16)`
	`*+ARn(#K16)`
	`*CDP(#K16)`
	`*+CDP(#K16)`

mmap() and port() qualifiers	An instruction that uses the mmap() qualifier to indicate an access to a memory-mapped register or registers cannot be placed in parallel with another instruction. The use of the mmap() qualifier is a form of parallelism already. Likewise an instruction that uses a port() qualifier to indicate an access to I/O space cannot be placed in parallel with another instruction. The use of a port() qualifier is a form of parallelism already.
Parallelism among A unit, D unit, and P unit	Parallelism among the three computational units is allowed without restriction (see Figure 4–5). An operation executed within a single computational unit can be placed in parallel with a second operation executed in one of the other two computational units.
Parallelism within the P unit	Two program-control instructions cannot be placed in parallel. However, other parallelism among the operators of the P unit is allowed.
Parallelism within the D unit	Certain restrictions apply to using operators of the D unit in parallel (see Figure 4–5).
Parallelism within the A unit	Two A-unit ALU operations or two A-unit swap operations cannot be performed in parallel. However, other parallelism among the operators of the A unit is allowed.

4.2.5 Process for Implementing User-Defined Parallelism

This section presents a process that may be used to simplify the process of using user-defined parallelism to produce optimized assembly language code. Figure 4–7 is a flow chart outlining this process, and the steps are also described in Table 4–4.

Figure 4–7. Process for Applying User-Defined Parallelism

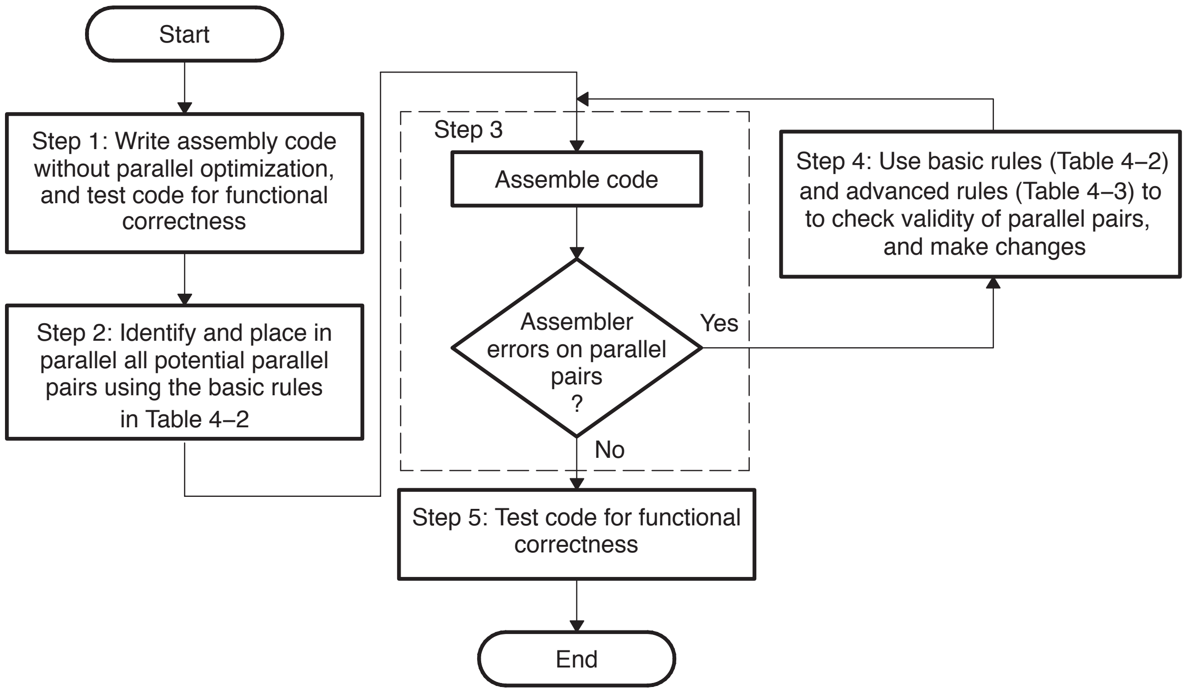

Start
Step 1: Write assembly code without parallel optimization, and test code for functional correctness
Step 2: Identify and place in parallel all potential parallel pairs using the basic rules in Table 4–2
Step 3
Assemble code
Assembler errors on parallel pairs ?
Yes
No
Step 4: Use basic rules (Table 4–2) and advanced rules (Table 4–3) to to check validity of parallel pairs, and make changes
Step 5: Test code for functional correctness
End

Table 4–4. Steps in Process for Applying User-Defined Parallelism

Step	Description
1	Write assembly code without the use of user-defined parallelism, and verify the functionality of the code. Note that in this step, you may take advantage of instructions with built-in parallelism.
2	Identify potential user-defined parallel instruction pairs in your code, and, using the basic rules outlined in Table 4–2 as guidelines, place instructions in parallel. Start by focusing on heavily used kernels of the code.
3	Run the optimized code through the assembler to see if the parallel instruction pairs are valid. The assembler will indicate any invalid parallel instruction pairs. If you have invalid pairs, go to step 4; otherwise go to step 5.
4	Refer to the set of parallelism rules in section 4.2.4 to determine why failing parallel pairs may be invalid. Make necessary changes and return to step 3.
5	Once all your parallel pairs are valid, make sure your code still functions correctly.

4.2.6 Parallelism Tips

As you try to optimize your code with user-defined parallelism, you might find the following tips helpful (these tips use mnemonic instructions):

- Place all load and store instructions in parallel. For example:

```
MOV   *AR2, AC1 ; Load AC1
||MOV BRC0, *AR3   ; Store BRC0
```

- The A-Unit ALU can handle (un)saturated 16-bit processing in parallel with the D-Unit ALU, MAC, and shift operators. For example:

```
; Modify AR1 in A unit, and perform an accumulator
; shift, saturate, and store operation in the D unit.
    ADD     T0, AR1
    ||MOV uns(rnd(HI(saturate(AC1 << #1)))), *AR2
```

- Accumulator shift, saturate, and store operations can be placed in parallel with D-Unit ALU or MAC operations. For example:

```
; Shift, saturate, and store AC1 while
; modifying AC2
    MOV     uns(rnd(HI(saturate(AC1 << #1)))), *AR2
    ||ADD AC1, AC2
```

- Control operations can be placed in parallel with DSP operations. For example:

```
; Switch control to a block-repeat loop, and
; Perform the first computation of the loop
    ||MACMR  *AR1+, *AR3, AC0
    RPTBLOCAL   loop-1
```

- Instructions with built-in parallelism increase the bandwidth of instructions paired by user-defined parallelism. For example:

```
; Place parallel accumulator load operations
; in parallel with an auxiliary register store
; operation
    ADD     dual(*AR2), AC0, AC1
    ||MOV AC2, dbl(*AR6)
```

- You can fill a buffer with a constant value efficiently. For example:

```
MOV   #0, AC0         ; Clear AC0
||RPT #9              ; Switch control to repeat loop
MOV   AC0, dbl(*AR1+) ; Store 32-bit constant to buffer
                      ; and increment pointer
```

- Instructions to be executed conditionally can be placed in parallel with the *if* instruction. For example:

```
XCC   check, T0 == #0 ; If T0 contains 0, ...
||MOV #0, AR0      ; ... Load AR0 with 0
check
```

4.2.7 Examples of Parallel Optimization Within CPU Functional Units

This section provides examples to show how to make code more efficient by using parallelism within the A unit, the P unit, and D unit.

4.2.7.1 A-Unit Example of Parallel Optimization

Example 4–4 shows a simple Host-DSP application in which a host sends a single command to tell the DSP which set of coefficients to use for a multiply-and-accumulate (MAC) operation. The DSP calls a COMPUTE function to perform the computation and returns the result to the host. The communication is based on a very simple handshaking, with the host and DSP exchanging flags (codes). The code in Example 4–4 does not use parallelism. Example 4–5 shows the code optimized through the use of parallel instruction pairs.

As mentioned, Example 4–5 shows the optimized code for Example 4–4. In Example 4–5, the parallel instruction pairs are highlighted. Notice the following points:

❏ The first four instructions (ARn loads) are immediate loads and cannot be placed in parallel due to constant bus conflicts and total instruction sizes.

❏ The first parallel pair shows an immediate load of CSR through the bus called KDB. This load is executed in parallel with the setting of the SXMD mode bit, which is handled by the A-unit ALU.

❏ The second parallel pair is a SWAP instruction in parallel with an *if* instruction. Despite the fact that the SWAP instruction is executed conditionally, it is valid to place it in parallel with the *if* instruction.

❏ The third parallel pair stores AR4 to memory via the D bus (DB), and stores a constant (BUSY) to memory via the bus called KDB.

❏ The fourth parallel pair loads AC1 with a constant that is carried on the bus called KDB and, in parallel, switches program control to a single-repeat loop.

❏ The last parallel pair stores AC1 to AR4 via a cross-unit bus and, in parallel, returns from the COMPUTE function.

Example 4–4. A-Unit Code With No User-Defined Parallelism

```
; Variables
      .data

COEFF1   .word   0x0123        ; First set of coeffiecients
      .word   0x1234
      .word   0x2345
      .word   0x3456
      .word   0x4567

COEFF2   .word   0x7654        ; Second set of coefficients
      .word   0x6543
      .word   0x5432
      .word   0x4321
      .word   0x3210

HST_FLAG  .set   0x2000        ; Host flag address
HST_DATA  .set   0x2001        ; Host data address

CHANGE   .set 0x0000           ; "Change coefficients" command from host
READY    .set 0x0000           ; "REARY" Flag from Host
BUSY     .set 0x1111           ; "BUSY" Flag set by DSP

      .global  start_a1

      .text

start_a1:
      MOV  #HST_FLAG, AR0      ; AR0 points to Host Flag
      MOV  #HST_DATA, AR2      ; AR2 points to Host Data
      MOV  #COEFF1, AR1        ; AR1 points to COEFF1 buffer initially
      MOV  #COEFF2, AR3        ; AR3 points to COEFF2 buffer initially
      MOV  #4, CSR             ; Set CSR = 4 for repeat in COMPUTE
      BSET FRCT                ; Set fractional mode bit
      BSET SXMD                ; Set sign-extension mode bit
```

Note: The algebraic instructions code example for A-Unit Code With No User-Defined Parallelism is shown in Example B–17 on page B-14.

Example 4–4. A-Unit Code With No User-Defined Parallelism (Continued)

```
LOOP:
      MOV  *AR0, T0                ; T0 = Host Flag
      BCC  PROCESS, T0 == #READY   ; If Host Flag is “READY”, continue
      B LOOP                       ; process – else poll Host Flag again

PROCESS:
      MOV  *AR2, T0                ; T0 = Host Data

      XCC  Check, T0 == #CHANGE
                                   ; The choice of either set of
                                   ; coefficients is based on the vlaue
                                   ; of T0.  COMPUTE uses AR3 for
                                   ; computation, so we need to
                                   ; load AR3 correctly here.

      SWAP AR1, AR3                ; Host message was “CHANGE”, so we
                                   ; need to swap the two coefficient
                                   ; pointers.

Check:
      CALL COMPUTE                 ; Compute subroutine
      MOV  AR4, *AR2               ; Write result to Host Data
      MOV  #BUSY, *AR0             ; Set Host Flag to Busy
      B LOOP                       ; Infinite loop continues
END

COMPUTE:
      MOV  #0, AC1                 ; Initialize AC1 to 0
      RPT  CSR                     ; CSR has a value of 4
      MACM *AR2, *AR3+, AC1        ; This MAC operation is performed
                                   ; 5 times
      MOV  AC1, AR4                ; Result is in AR4
      RET

HALT:
      B HALT
```

Note: The algebraic instructions code example for A-Unit Code With No User-Defined Parallelism is shown in Example B–17 on page B-14.

Example 4–5. A-Unit Code in Example 4–4 Modified to Take Advantage of Parallelism

```
; Variables
      .data

COEFF1   .word   0x0123           ; First set of coeffiecients
      .word   0x1234
      .word   0x2345
      .word   0x3456
      .word   0x4567

COEFF2   .word   0x7654           ; Second set of coefficients
      .word   0x6543
      .word   0x5432
      .word   0x4321
      .word   0x3210

HST_FLG  .set 0x2000              ; Host flag address
HST_DATA  .set   0x2001           ; Host data address

CHANGE   .set 0x0000              ; “Change coefficients” command from host
READY    .set 0x0000              ; “READY” Flag from Host
BUSY     .set 0x1111              ; “BUSY” Flag set by DSP

      .global  start_a2

      .text

start_a2:
      MOV  #HST_FLAG, AR0         ; AR0 points to Host Flag
      MOV  #HST_DATA, AR2         ; AR2 points to Host Data
      MOV  #COEFF1, AR1           ; AR1 points to COEFF1 buffer initially
      MOV  #COEFF2, AR3           ; AR3 points to COEFF2 buffer initially
      MOV  #4, CSR                ; Set CSR = 4 for repeat in COMPUTE
    ||BSET FRCT                   ; Set fractional mode bit
      BSET SXMD                   ; Set sign-extension mode bit

LOOP:
      MOV  *AR0, T0               ; T0 = Host Flag
      BCC  PROCESS, T0 == #READY  ; If Host Flag is “READY”, continue
      B LOOP                      ; process – else poll Host Flag again
```

Note: The algebraic instructions code example for A-Unit Code in Example 4–4 Modified to Take Advantage of Parallelism is shown in Example B–18 on page B-16.

Example 4–5. A-Unit Code in Example 4–4 Modified to Take Advantage of Parallelism (Continued)

```
PROCESS:
      MOV  *AR2, T0              ; T0 = Host Data

      XCC  Check, T0 == #CHANGE
                                 ; The choice of either set of
                                 ; coefficients is based on the vlaue
                                 ; of T0.  COMPUTE uses AR3 for
                                 ; computation, so we need to
                                 ; load AR3 correctly here.

    ||SWAP AR1, AR3              ; Host message was "CHANGE", so we
                                 ; need to swap the two coefficient
                                 ; pointers.

Check:
      CALL COMPUTE               ; Compute subroutine
      MOV  AR4, *AR2             ; Write result to Host Data
    ||MOV  #BUSY, *AR0           ; Set Host Flag to Busy
      B LOOP                     ; Infinite loop continues

END

COMPUTE:
      MOV  #0, AC1               ; Initialize AC1 to 0
    ||RPT  CSR                   ; CSR has a value of 4
      MACM *AR2, *AR3+, AC1      ; This MAC operation is performed
                                 ; 5 times
      MOV  AC1, AR4              ; Result is in AR4
    ||RET

HALT:
      B HALT
```

Note: The algebraic instructions code example for A-Unit Code in Example 4–4 Modified to Take Advantage of Parallelism is shown in Example B–18 on page B-16.

4.2.7.2 P-Unit Example of Parallel Optimization

Example 4–6 demonstrates a very simple nested loop and some simple control operations with the use of P-unit registers. This example shows the unoptimized code, and Example 4–7 shows the code optimized through the use of the P-unit parallel instruction pairs (parallel instruction pairs are highlighted).

Notice the following about Example 4–7:

- ❑ The first three register loads are immediate loads using the same constant bus and, therefore, cannot be placed in parallel.
- ❑ The fourth register load (loading BRC0) can be placed in parallel with the next load (loading BRC1), which does not use the constant bus, but the data bus DB to perform the load.
- ❑ The next instruction, which is a control instruction (blockrepeat) is placed in parallel with the load of AC2. There are no conflicts and this pair is valid. The loading of AC2 is not part of the blockrepeat structure.
- ❑ The final parallel pair is a control instruction (localrepeat) and a load of AR1. The loading of AR1 is not part of the localrepeat structure but is part of the outer blockrepeat structure.

Example 4–6. P-Unit Code With No User-Defined Parallelism

```
    .CPL_off              ; Tell assembler that CPL bit is 0
                          ; (Direct addressing is done with DP)

; Variables
    .data

var1   .word  0x0004
var2   .word  0x0000

    .global   start_p1

    .text

start_p1:
    AMOV   #var1, XDP
    MOV #var2, AR3

    MOV #0007h, BRC0      ; BRC0 loaded using KPB
    MOV *AR3, BRC1        ; BRC1 loaded using DB

    MOV #0006h, AC2

    RPTB      Loop1-1
    MOV AC2, AC1
    MOV #8000h, AR1
    RPTBLOCAL Loop2-1
    SUB #1, AC1
    MOV AC1, *AR1+
Loop2:
    ADD #1, AC2
Loop1:

    MOV BRC0, @AC0L       ; AC0L loaded using EB
    MOV BRC1, @AC1L       ; AC1L loaded using EB

    BCC start_p1, AC0 >= #0
    BCC start_p1, AC1 >= #0
end_p1
```

Note: The algebraic instructions code example for P-Unit Code With No User-Defined Parallelism is shown in Example B–19 on page B-18.

Example 4–7. P-Unit Code in Example 4–6 Modified to Take Advantage of Parallelism

```
    .CPL_off                  ; Tell assembler that CPL bit is 0
                              ; (Direct addressing is done with DP)

; Variables
    .data

var1   .word  0x0004
var2   .word  0x0000

    .global   start_p2

    .text

start_p2:
    AMOV   #var1, XDP
    MOV #var2, AR3

    MOV #0007h, BRC0          ; BRC0 loaded using KPB
    ||MOV  *AR3, BRC1         ; BRC1 loaded using DB

    MOV #0006h, AC2
    ||RPTB    Loop1-1
    MOV AC2, AC1
    MOV #8000h, AR1
    ||RPTBLOCAL  Loop2-1
    SUB #1, AC1
    MOV AC1, *AR1+
Loop2:
    ADD #1, AC2
Loop1:

    MOV BRC0, @AC0L           ; AC0L loaded using EB
    MOV BRC1, @AC1L           ; AC1L loaded using EB

    BCC start_p1, AC0 >= #0
    BCC start_p1, AC1 >= #0
end_p2
```

Note: The algebraic instructions code example for P-Unit Code in Example 4–6 Modified to Take Advantage of Parallelism is shown in Example B–20 on page B-19.

4.2.7.3 D-Unit Example of Parallel Optimization

Example 4–8 demonstrates a very simple load, multiply, and store function with the use of D-unit registers. Example 4–9 shows this code modified to take advantage of user-defined parallelism (parallel instruction pairs are highlighted).

The following information determined the optimizations made in Example 4–9:

- ❑ As in the P-unit example on page 4-32, we cannot place the immediate register loads in parallel due to constant-bus conflicts.
- ❑ The instructions that load AC0 and AC2 have been placed in parallel because they are not both immediate loads and as such, there are no constant-bus conflicts.
- ❑ It is not possible to place the two single-MAC instructions in parallel since the same operator is required for both and as such a conflict arises. However, placing the second MAC instruction in parallel with the SWAP instruction is valid.
- ❑ The two 16-bit store operations at the end of the code are placed in parallel because there are two 16-bit write buses available (EB and FB).

Example 4–8. D-Unit Code With No User-Defined Parallelism

```
; Variables
    .data

var1   .word  0x8000
var2   .word  0x0004

    .global   start_d1

    .text

start_d1:
    MOV #var1, AR3
    MOV #var2, AR4

    MOV #0004h, AC0      ; AC0 loaded using KDB
    MOV *AR3, AC2        ; AC2 loaded using DB

    MOV #5A5Ah, T0       ; T0 loaded with constant, 0x5A5A

    MAC AC0, T0, AC2     ; MAC
    MAC AC2, T0, AC1     ; MAC
    SWAP   AC0, AC2      ; SWAP

    MOV HI(AC1), *AR3    ; Store result in AC1
    MOV HI(AC0), *AR4    ; Store result in AC0
end_d1
```

Note: The algebraic instructions code example for D-Unit Code With No User-Defined Parallelism is shown in Example B–21 on page B-20.

Example 4–9. D-Unit Code in Example 4–8 Modified to Take Advantage of Parallelism

```
; Variables
   .data

var1   .word  0x8000
var2   .word  0x0004

   .global   start_d2

   .text

start_d2:
   MOV #var1, AR3
   MOV #var2, AR4

   MOV #0004h, AC0         ; AC0 loaded using KDB
   ||MOV  *AR3, AC2        ; AC2 loaded using DB

   MOV #5A5Ah, T0          ; T0 loaded with constant, 0x5A5A

   MAC AC0, T0, AC2        ; MAC
   MAC AC2, T0, AC1        ; MAC
   ||SWAP AC0, AC2         ; SWAP

   MOV HI(AC1), *AR3       ; Store result in AC1
   ||MOV  HI(AC0), *AR4    ; Store result in AC0
end_d2
```

Note: The algebraic instructions code example for D-Unit Code in Example 4–8 Modified to Take Advantage of Parallelism is shown in Example B–22 on page B-21.

4.2.8 Example of Parallel Optimization Across the A-Unit, P-Unit, and D-Unit

Example 4–10 shows unoptimized code for an FIR (finite impulse response) filter. Example 4–11 is the result of applying user-defined parallelism to the same code. It is important to notice that the order of instructions has been altered in a number of cases to allow certain instruction pairs to be placed in parallel. The use of parallelism in this case has saved about 50% of the cycles outside the inner loop.

Example 4–10. Code That Uses Multiple CPU Units But No User-Defined Parallelism

```
   .CPL_ON             ; Tell assembler that CPL bit is 1
                       ; (SP direct addressing like *SP(0) is enabled)

; Register usage
; -----------------------------------------------------------------

   .asg   AR0, X_ptr   ; AR0 is pointer to input buffer – X_ptr
   .asg   AR1, H_ptr   ; AR1 is pointer to coefficients – H_ptr
   .asg   AR2, R_ptr   ; AR2 is pointer to result buffer – R_ptr
   .asg   AR3, DB_ptr  ; AR3 is pointer to delay buffer – DB_ptr

FRAME_SZ  .set   2

   .global _fir

   .text

; *****************************************************************

_fir

; Create local frame for temp values
; -----------------------------------------------------------------

   AADD   #–FRAME_SZ, SP

; Turn on fractional mode
; Turn on sign–extension mode
; ------------------------------------------------------------------

   BSET   FRCT         ; Set fractional mode bit
   BSET   SXMD         ; Set sign extension mode bit

; Set outer loop count by subtracting 1 from nx and storing into
; block–repeat counter
; -------------------------------------------------------------------

   SUB #1, T1, AC1     ; AC1 = number of samples (nx) – 1
   MOV AC1, *SP(0)     ; Top of stack = nx – 1
   MOV *SP(0), BRC0    ; BRC0 –= nx – 1 (outer loop counter)
```

Note: The algebraic instructions code example for Code That Uses Multiple CPU Units But No User-Defined Parallelism is shown in Example B–23 on page B-22.

Example 4–10. Code That Uses Multiple CPU Units But No User-Defined Parallelism (Continued)

```
; Store length of coefficient vector/delay buffer in BK register
; ------------------------------------------------------------

   BSET   AR1LC             ; Enable AR1 circular configuration
   MOV *(#0011h), BSA01     ;Set buffer (filter) start address
                            ; AR1 used as filter pointer

   BSET   AR3LC             ; Enable AR3 circular configuration]
   MOV DB_ptr, *SP(1)       ; Save pointer to delay buffer pointer
   MOV *DB_ptr, AC1         ; AC1 = delay buffer pointer
   MOV AC1, DB_ptr          ; AR3 (DB_ptr) = delay buffer pointer
   MOV *(#0013h), BSA23     ;Set buffer (delay buffer) start address
                            ; AR3 used as filter pointer
   MOV T0, *SP(0)           ; Save filter length, nh – used as buffer
                            ; size
   MOV *SP(0), BK03         ; Set circular buffer size – size passed
                            ; in T0
   SUB #3, T0, AC1          ; AC1 = nh – 3
   MOV AC1, *SP(0)
   MOV *SP(0), CSR          ; Set inner loop count to nh – 3
   MOV #0, H_ptr            ; Initialize index of filter to 0
   MOV #0, DB_ptr           ; Initialize index of delay buffer to 0

; Begin outer loop on nx samples
; --------------------------------------------------------------

   RPTB   Loop1–1

; Move next input sample into delay buffer
; --------------------------------------------------------------

   MOV *X_ptr+, *DB_ptr

; Sum h * x for next y value
; --------------------------------------------------------------

   MPYM   *H_ptr+, *DB_ptr+, AC0

   RPT CSR
   MACM   *H_ptr+, *DB_ptr+, AC0

   MACMR  *H_ptr+, *DB_ptr, AC0   ; Round result
```

Note: The algebraic instructions code example for Code That Uses Multiple CPU Units But No User-Defined Parallelism is shown in Example B–23 on page B-22.

Example 4–10. Code That Uses Multiple CPU Units But No User-Defined Parallelism (Continued)

```
; Store result
; ------------------------------------------------------------------------

   MOV HI(AC0), *R_ptr+
Loop1:

; Clear FRCT bit to restore normal C operating environment
; Return overflow condition of AC0 (shown in ACOV0) in T0
; Restore stack to previous value, FRAME, etc.
; Update current index of delay buffer pointer
; ------------------------------------------------------------------------

END_FUNCTION:

   MOV *SP(1), AR0          ; AR0 = pointer to delay buffer pointer
   AADD   #FRAME_SZ, SP     ; Remove local stack frame
   MOV DB_ptr, *AR0         ; Update delay buffer pointer with current
                            ; index

   BCLR   FRCT              ; Clear fractional mode bit

   MOV #0, T0               ; Make T0 = 0 for no overflow (return value)
   XCC Label, overflow(AC0)
   MOV #1, T0               ; Make T0 = 1 for overflow (return value)
Label:

   RET
; ************************************************************************
```

Note: The algebraic instructions code example for Code That Uses Multiple CPU Units But No User-Defined Parallelism is shown in Example B–23 on page B-22.

In Example 4–11, parallel pairs that were successful are shown in **bold** type; potential parallel pairs that failed are shown in *italic* type. The first failed due to a constant bus conflict, and the second failed due to the fact that the combined size is greater than 6 bytes. The third pair failed for the same reason, as well as being an invalid soft-dual encoding instruction. This last pair in italics failed because neither instruction has a parallel enable bit. Some of the load/store operations that are not in parallel were made parallel in the first pass optimization process; however, the parallelism failed due to bus conflicts and had to be removed.

Note:

Example 4−11 shows optimization *only* with the use of the parallelism features. Further optimization of this FIR function is possible by employing other optimizations.

Example 4−11. Code in Example 4−10 Modified to Take Advantage of Parallelism

```
    .CPL_ON                 ; Tell assembler that CPL bit is 1
                            ; (SP direct addressing like *SP(0) is enabled)

; Register usage
; -----------------------------------------------------------------

    .asg   AR0, X_ptr       ; AR0 is pointer to input buffer – X_ptr
    .asg   AR1, H_ptr       ; AR1 is pointer to coefficients – H_ptr
    .asg   AR2, R_ptr       ; AR2 is pointer to result buffer – R_ptr
    .asg   AR3, DB_ptr      ; AR3 is pointer to delay buffer – DB_ptr

FRAME_SZ  .set   2

    .global _fir

    .text

; *****************************************************************

_fir

; Create local frame for temp values
; -----------------------------------------------------------------

    AADD   #-FRAME_SZ, SP   ; (Attempt to put this in parallel with
                            ; the following AC1 modification failed)

; Set outer loop count by subtracting 1 from nx and storing into
; block-repeat counter
; Turn on fractional mode
; Turn on sign-extension mode
; -------------------------------------------------------------------

    SUB #1, T1, AC1         ; AC1 = number of samples (nx) – 1

    BSET   FRCT             ; Set fractional mode bit
    ||MOV  AC1, *SP(0)      ; Top of stack = nx – 1

    MOV *SP(0), BRC0        ; BRC0 = nx – 1 (outer loop counter)
    ||BSET SXMD             ; Set sign-extension mode bit
```

Note: The algebraic instructions code example for Code in Example 4−10 Modified to Take Advantage of Parallelism is shown in Example B−24 on page B-25.

Example 4–11. Code in Example 4–10 Modified to Take Advantage of Parallelism (Continued)

```
; Store length of coefficient vector/delay buffer in BK register
; ------------------------------------------------------------

   BSET   AR1LC             ; Enable AR1 circular configuration
   MOV *(#0011h), BSA01     ;Set buffer (filter) start address
                            ; AR1 used as filter pointer

   BSET   AR3LC             ; Enable AR3 circular configuration
   ||MOV  DB_ptr, *SP(1)    ; Save pointer to delay buffer pointer

   MOV *DB_ptr, AC1         ; AC1 = delay buffer pointer
   MOV AC1, DB_ptr          ; AR3 (DB_ptr) = delay buffer pointer
   ||MOV  T0, *SP(0)        ; Save filter length, nh – used as buffer
                            ; size
   MOV *(#0013h), BSA23     ;Set buffer (delay buffer) start address
                            ; AR3 used as filter pointer

   MOV *SP(0), BK03         ; Set circular buffer size – size passed
                            ; in T0

   SUB #3, T0, AC1          ; AC1 = nh – 3
   MOV AC1, *SP(0)

   MOV *SP(0), CSR          ; Set inner loop count to nh – 3
   ||MOV  #0, H_ptr         ; Initialize index of filter to 0

   MOV #0, DB_ptr           ; Initialize index of delay buffer to 0
                            ; (in parallel with RPTB below)

; Begin outer loop on nx samples
; --------------------------------------------------------------

   ||RPTB Loop1-1

; Move next input sample into delay buffer
; --------------------------------------------------------------

   MOV *X_ptr+, *DB_ptr
```

Note: The algebraic instructions code example for Code in Example 4–10 Modified to Take Advantage of Parallelism is shown in Example B–24 on page B-25.

Example 4–11. Code in Example 4–10 Modified to Take Advantage of Parallelism (Continued)

```
; Sum h * x for next y value
; ----------------------------------------------------------------------

   MPYM   *H_ptr+, *DB_ptr+, AC0
   ||RPT  CSR

   MACM   *H_ptr+, *DB_ptr+, AC0

   MACMR  *H_ptr+, *DB_ptr, AC0   ; Round result

; Store result
; ----------------------------------------------------------------------

   MOV HI(AC0), *R_ptr+
Loop1:

; Clear FRCT bit to restore normal C operating environment
; Return overflow condition of AC0 (shown in ACOV0) in T0
; Restore stack to previous value, FRAME, etc.
; Update current index of delay buffer pointer
; ----------------------------------------------------------------------

END_FUNCTION:

   MOV *SP(1), AR0                ; AR0 = pointer to delay buffer pointer
   ||AADD #FRAME_SZ, SP           ; Remove local stack frame

   MOV DB_ptr, *AR0               ; Update delay buffer pointer with current
                                  ; index
   ||BCLR FRCT                    ; Clear fractional mode bit

   MOV #0, T0                     ; Make T0 = 0 for no overflow (return value)
   ||XCC  Label, overflow(AC0)
   MOV #1, T0                     ; Make T0 = 1 for overflow (return value)
Label:

   ||RET
; **********************************************************************
```

Note: The algebraic instructions code example for Code in Example 4–10 Modified to Take Advantage of Parallelism is shown in Example B–24 on page B-25.

4.3 Implementing Efficient Loops

There are four common methods to implement instruction looping in the C55x DSP:

- ❑ Single repeat: *repeat(CSR/k8/k16),[CSR += TA/k4]*
- ❑ Local block repeat: *localrepeat{}*
- ❑ Block repeat: *blockrepeat{}*
- ❑ Branch on auxiliary register not zero: *if (ARn_mod != 0) goto #loop_start*

The selection of the looping method to use depends basically on the number of instructions that need to be repeated and in the way you need to control the loop counter parameter. The first three methods in the preceding list offer zero-overhead looping and the fourth one offers a 5-cycle loop overhead.

Overall, the most efficient looping mechanisms are the *repeat()* and the *localrepeat{}* mechanisms. The *repeat()* mechanism provides a way to repeat a single instruction or a parallel pair of instructions in an interruptible way. *repeat(CSR)*, in particular, allows you to compute the loop counter at runtime. Refer to section 4.3.2, *Efficient Use of* repeat(CSR) *Looping*.

Note:

If you are migrating code from a TMS320C54x DSP, be aware that a single-repeat instruction is interruptible on a TMS320C55x DSP. On a TMS320C54x DSP, a single-repeat instruction cannot be interrupted.

The *localrepeat{}* mechanism provides a way to repeat a block from the instruction buffer queue. Reusing code that has already been fetched and placed in the queue brings the following advantages:

- ❑ Fewer program-memory access pipeline conflicts
- ❑ Overall lower power consumption
- ❑ No repetition of wait-state and access penalties when executing loop code from external RAM

4.3.1 Nesting of Loops

You can create up to two levels of block-repeat loops without any cycle penalty. You can have one block-repeat loop nested inside another, creating an inner (level 1) loop and an outer (level0) loop. In addition, you can put any number of single-repeat loops inside each block-repeat loop. Example 4–12 shows a multi-level loop structure with two block-repeat loops and two single-repeat loops. (The examples in this section use instructions from the algebraic instruction set, but the concepts apply equally for the mnemonic instruction set.)

Example 4−12 shows one block-repeat loop nested inside another block-repeat loop. If you need more levels of multiple-instruction loops, use *branch on auxiliary register not zero* constructs to create the remaining outer loops. In Example 4−13 (page 4-44), a *branch on auxiliary register not zero* construct (see the last instruction in the example) forms the outermost loop of a Fast Fourier Transform algorithm. Inside that loop are two *localrepeat{}* loops. Notice that if you want the outermost loop to execute *n* times, you must initialize AR0 to *(n − 1)* outside the loop.

Example 4−12. Nested Loops

```
    MOV   #(n0−1), BRC0
    MOV   #(n1−1), BRC1
;   ...
    RPTBLOCAL    Loop1−1       ; Level 0 looping (could instead be blockrepeat):
                               ; Loops n0 times
;       ...
        RPT    #(n2−1)
;       ...
        RPTBLOCAL    Loop2−1   ; Level 1 looping (could instead be blockrepeat):
                               ; Loops n1 times
;       ...
            RPT    #(n3−1)
;           ...
Loop2:
;       ...
Loop1:
```

Note: The algebraic instructions code example for Nested Loops is shown in Example B−25 on page B-28.

Example 4–13. Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code)

```
_cfft:

radix_2_stages:
; ...

outer_loop:
; ...
      MOV   T1, BRC0
; ...
      MOV   T1, BRC1
; ...
      SFTS  AR4, #1                           ; outer loop counter
      ||BCC no_scale, AR5 == #0               ; determine if scaling required
; ...

no_scale:

      RPTBLOCAL   Loop1-1

      MOV   dbl(*AR3), AC0                    ; Load ar,ai

      SUB   AC0, dual(*AR2), AC2              ; tr = ar - br
                                              ; ti = ai - bi

      RPTBLOCAL   Loop2-1

      ADD   dual(*AR2), AC0, AC1              ; ar' = ar + br
                                              ; ai' = ai + bi
      ||MOV AC2, dbl(*AR6)                    ; Store tr, ti

      MPY   *AR6, *CDP+, AC2                  ; c*tr
      ::MPY *AR7, *CDP+, AC3                  ; c*ti

      MOV   AC1, dbl(*AR2+)                   ; Store ar, ai
      ||MOV dbl(*AR3(T0)), AC0                ; * load ar,ai

      MASR  *AR6, *CDP-, AC3                  ; bi' = c*ti - s*tr
      ::MACR      *AR7, *CDP-, AC2            ; br' = c*tr + s*ti

      MOV   pair(HI(AC2)), dbl(*AR3+)         ; Store br', bi'
      ||SUB AC0, dual(*AR2), AC2              ; * tr = ar - br
                                              ; * ti = ai - bi
```

Note: The algebraic instructions code example for Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code) is shown in Example B–26 on page B-28.

Example 4–13. Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code) (Continued)

```
Loop2:
       ADD   dual(*AR2), AC0, AC1           ; ar’ = ar + br
                                            ; ai’ = ai + bi
       ||MOV AC2, dbl(*AR6)                 ; Store tr, ti

       MPY   *AR6, *CDP+, AC2               ; c*tr
       ::MPY *AR7, *CDP+, AC3               ; c*ti

       MOV   AC1, dbl(*(AR2+T1))            ; Store ar, ai

       MASR  *AR6, *CDP+, AC3               ; bi’ = c*ti – s*tr
       ::MACR      *AR7, *CDP+, AC3         ; br’ = c*tr + s*ti

       MOV   pair(HI(AC2)), dbl(*(AR3+T1))  ; Store br’, bi’
Loop1:

       SFTS  AR3, #1
       ||MOV #0, CDP                        ; rewind coefficient pointer
       SFTS  T3, #1
       ||BCC outer_loop, AR4 != #0
```

Note: The algebraic instructions code example for Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code) is shown in Example B–26 on page B-28.

To achieve an efficient nesting of loops, apply the following guidelines:

❑ Use a single-repeat instruction for the innermost loop if the loop contains only a single instruction (or a pair of instructions that have been placed in parallel).

❑ Use a local block-repeat instruction (*localrepeat* in the algebraic syntax) for a loop containing more than a single instruction or instruction pair—provided the loop contains no more than 56 bytes of code. If there are more than 56 bytes but you would still like to use the local block-repeat instruction, consider the following possibilities:

  ■ Split the existing loop into two smaller loops.

  ■ Reduce the number of bytes in the loop. For example, you can reduce the number of instructions that use embedded constants.

❑ Use a standard block-repeat instruction (*blockrepeat* in the algebraic syntax) in cases where a local block-repeat instruction cannot be used. The standard block-repeat mechanism always refetches the loop code from memory.

- When you nest a block-repeat loop inside another block-repeat loop, initialize the block-repeat counters (BRC0 and BRC1) in the code outside of both loops. This technique is shown in Example 4–12 (page 4-43).

  Neither counter needs to be re-initialized inside its loop; placing such initializations inside the loops only adds extra cycles to the loops. The CPU uses BRC0 for the outer (level 0) loop and BRC1 for the inner (level 1) loop. BRC1 has a shadow register, BRS1, that preserves the initial value of BRC1. Each time the level 1 loop must begin again, the CPU automatically re-initializes BRC1 from BRS1.

4.3.2 Efficient Use of *repeat(CSR)* Looping

The single-repeat instruction syntaxes allow you to specify the repeat count as a constant (embedded in the repeat instruction) or as the content of the computed single-repeat register (CSR). When CSR is used, it is not decremented during each iteration of the single-repeat loop. Before the first execution of the instruction or instruction pair to be repeated, the content of CSR is copied into the single-repeat counter (RPTC). RPTC holds the active loop count and is decremented during each iteration. Therefore, CSR needs to be initialized only once. Initializing CSR outside the outer loop, rather than during every iteration of the outer loop, saves cycles. There are advantages to using CSR for the repeat count:

- The repeat count can be dynamically computed during runtime and stored to CSR. For example, CSR can be used when the number of times an instruction must be repeated depends on the iteration number of a higher loop structure.
- Using CSR saves outer loop cycles when the single-repeat loop is an inner loop.
- An optional syntax extension enables the repeat instruction to modify the CSR after copying the content of CSR to RPTC. When the single-repeat loop is repeated in an outer loop, CSR contains a new count.

Example 4–14 (page 4-47) uses CSR for a single-repeat loop that is nested inside a block-repeat loop. In the example, CSR is assigned the name inner_cnt.

Example 4–14. Use of CSR (Shown in Real Block FIR Loop Code)

```
;   ...
    .asg   CSR, inner_cnt      ; inner loop count
    .asg   BRC0, outer_cnt     ; outer loop count
;   ...
    .asg   AR0, x_ptr       ; linear pointer
    .asg   AR1, db_ptr1     ; circular pointer
    .asg   AR2, r_ptr       ; linear pointer
    .asg   AR3, db_ptr2     ; circular pointer
    .asg   CDP, h_ptr       ; circular pointer

; ...

_fir2:

; ...
    AMAR   *db_ptr2-    ; index of 2nd oldest db entry
;
; Setup loop counts
; ---------------------------------------------------------------------
    ||SFTS T0, #-1          ; T0 = nx/2

    SUB #1, T0        ; T0 = (nx/2 - 1)
    MOV T0, outer_cnt    ; outer loop executes nx/2 times
    SUB #3, T1, T0       ; T0 = nh-3
    MOV T0, inner_cnt    ; inner loop executes nh-2 times
    ADD #1, T1        ; T1 = nh+1, adjustment for db_ptr1, bd_ptr2
;
; Start of outer loop
; ---------------------------------------------------------------------
    ||RPTBLOCAL   OuterLoop-1   ; start the outer loop

    MOV *x_ptr+, *db_ptr1   ; get 1st new input value
    MOV *x_ptr+, *db_ptr2   ; get 2nd new input value

; 1st interation
    MPY *db_ptr1+, *h_ptr+, AC0 ; part 1 of dual-MPY
    ::MPY  *db_ptr2+, *h_ptr+, AC1 ; part 2 of dual-MPY

; inner loop
    ||RPT  inner_cnt
    MAC *db_ptr1+, *h_ptr+, AC0 ; part 1 of dual-MAC
    ::MAC  *db_ptr2+, *h_ptr+, AC1 ; part 2 of dual-MAC
```

Note: This example shows portions of the file fir2.asm in the TI C55x DSPLIB (introduced in Chapter 8).

Example 4–14. Use of CSR (Shown in Real Block FIR Loop Code) (Continued)

```
; last iteration has different pointer adjustment and rounding
   MACR   *(db_ptr1-T1), *h_ptr+, AC0   ; part 1 of dual-MAC
   ::MACR *(db_ptr2-T1), *h_ptr+, AC1   ; part 2 of dual-MAC

; store result to memory
   MOV HI(AC0), *r_ptr+ ; store 1st Q15 result to memory
   MOV HI(AC1), *r_ptr+ ; store 2nd Q15 result to memory

OuterLoop:              ; end of outer loop

; ...
```

Note: This example shows portions of the file fir2.asm in the TI C55x DSPLIB (introduced in Chapter 8).

4.3.3 Avoiding Pipeline Delays When Accessing Loop-Control Registers

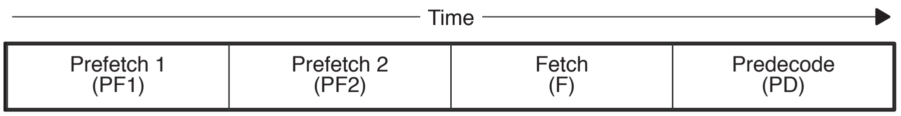
Accesses to loop-control registers like CSR, BRC0, and BRC1 can cause delays in the instruction pipeline if nearby instructions make competing accesses. For recommendations on avoiding this type of pipeline delay, see the “Loop control” section of Table 4–6 (page 4-54) .

4.4 Minimizing Pipeline and IBQ Delays

The C55x instruction pipeline is a protected pipeline that has two, decoupled segments:

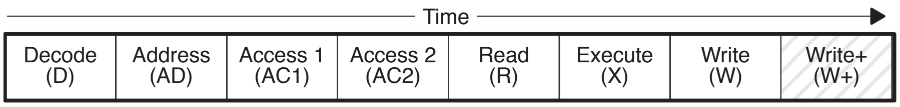
- The first segment, referred to as the *fetch pipeline*, fetches 32-bit instruction packets from memory into the instruction buffer queue (IBQ), and then feeds the second pipeline segment with 48-bit instruction packets. The fetch pipeline is illustrated in Figure 4–8.
- The second segment, referred to as the *execution pipeline*, decodes instructions and performs data accesses and computations. The execution pipeline is illustrated and described in Figure 4–9. Table 4–5 provides examples to help you understand the activity in the key phases of the execution pipeline.

Figure 4–8. First Segment of the Pipeline (Fetch Pipeline)



Pipeline Phase	Description
PF1	Present the program fetch address to memory.
PF2	Wait for memory to respond.
F	Fetch an instruction packet from memory and place it in the IBQ.
PD	Predecode instructions in the IBQ (identify where instructions begin and end; identify parallel instructions).

Figure 4–9. Second Segment of the Pipeline (Execution Pipeline)



Note: ▨ Only for memory write operations.

Pipeline Phase	**Description**
Decode (D)	❏ Read six bytes from the instruction buffer queue. ❏ Decode an instruction pair or a single instruction. ❏ Dispatch instructions to the appropriate CPU functional units. ❏ Read STx_55 bits associated with data address generation: ST1_55(CPL) ST2_55(ARnLC) ST2_55(ARMS) ST2_55(CDPLC) ❏ Read STx bits related to address generation (ARxLC, CPL, ARMS) for address generation in the AD phase.
Address (AD)	❏ Read/modify registers when involved in data address generation. For example: ARx and Tx in *ARx+(T0) BK03 if AR2LC=1 SP during pushes and pops SSP, same as for SP if in 32-bit stack mode ❏ Read A-unit register in the case of a R/W (in AD-phase) conflict. ❏ Perform operations that use the A-unit ALU. For example: Arithmetic using AADD instruction Swapping A-unit registers with a SWAP instruction Writing constants to A-unit registers (BKxx, BSAxx, BRCx, CSR, etc.) ❏ Decrement ARx for the conditional branch instruction that branches on ARx not zero. ❏ (Exception) Evaluate the condition of the XCC instruction (*execute(AD-unit*) attribute in the algebraic syntax).
Access 1 (AC1)	Refer to section 4.4.3.1 for description.
Access 2 (AC2)	Refer to section 4.4.3.1 for description.

Figure 4–9. Second Segment of the Pipeline (Execution Pipeline) (Continued)

Pipeline Phase	Description
Read (R)	❏ Read data from memory, I/O space, and MMR-addressed registers. ❏ Read A-unit registers when executing specific D-unit instructions that “prefetch” A-unit registers (listed in Appendix A) in the R phase rather than reading them in the X phase. ❏ Evaluate the conditions of conditional instructions. Most but not all condition evaluation is performed in the R phase. Exceptions are marked with (Exception) in this table.
Execute (X)	❏ Read/modify registers that are not MMR-addressed. ❏ Read/modify individual register bits. ❏ Set conditions. ❏ (Exception) Evaluate the condition of the XCCPART instruction (execute(D-unit) attribute in the algebraic syntax), **unless** the instruction is conditioning a write to memory (in this case, the condition is evaluated in the R phase). ❏ (Exception) Evaluate the condition of the RPTCC instruction.
Write (W)	❏ Write data to MMR-addressed registers or to I/O space (peripheral registers). ❏ Write data to memory. From the perspective of the CPU, the write operation is finished in this pipeline phase.
Write+ (W+)	Write data to memory. From the perspective of the memory, the write operation is finished in this pipeline phase.

Table 4–5. Pipeline Activity Examples

Example	Pipeline Explanation
AMOV #k23, XARx	XARx is initialized with a constant in the AD phase.
MOV #k, ARx	ARx is not MMR-addressed. ARx is initialized with a constant in the X phase.
MOV #k, mmap(ARx)	ARx is MMR-addressed. ARx is initialized with a constant in the W phase.
AADD #k, ARx	With this special instruction, ARx is initialized with a constant in the AD phase.
MOV #k, *ARx+	The memory write happens in the W+ phase. See special write pending and memory bypass cases in section 4.4.3.

Table 4–5. Pipeline Activity Examples (Continued)

Example	Pipeline Explanation
MOV *ARx+, AC0	ARx is read and updated in the AD phase. AC0 is loaded in the X phase.
ADD #k, ARx	ARx is read at the beginning of the X phase and is modified at the end of the X phase.
ADD ACy, ACx	ACx and ACy read and write activity occurs in the X phase.
MOV mmap(ARx), ACx	ARx is MMR-addressed and so is read in the R phase. ACx is modified in the X phase.
MOV ARx, ACx	ARx is not MMR-addressed and so is read in the X phase. ACx is modified in the X phase.
BSET CPL	The CPL bit is set in the X phase.
PUSH, POP, RET or AADD #K8, SP	SP is read and modified in the AD phase. SSP is also affected if the 32-bit stack mode is selected.
XCCPART overflow(ACx) || MOV *AR1+, AC1	The condition is evaluated in the X phase. Note: AR1 is incremented regardless of whether the condition is true.
XCCPART overflow(ACx) || MOV AC1, *AR1+	The condition is evaluated in the R phase because it conditions a write to memory. Note: AR1 is incremented regardless of whether the condition is true.
XCC overflow(ACx) || MOV *AR1+, AC1	The condition is evaluated in the AD phase. Note: AR1 is incremented only if the condition is true.

Multiple instructions are executed simultaneously in the pipeline, and different instructions perform modifications to memory, I/O space, and register values during different phases of the pipeline. In an unprotected pipeline, this could lead to data-access errors—reads and writes at the same location happening out of the intended order. The pipeline-protection unit of the C55x DSP inserts extra cycles to prevent these errors. If an instruction (say, instruction 3) must access a location but a previous instruction (say, instruction 1) is not done with the location, instruction 3 is halted in the pipeline until instruction 1 is done. To minimize delays, you can take steps to prevent many of these pipeline-protection cycles.

The instruction set reference guides (see *Related Documentation From Texas Instruments* in the preface) show how many cycles an instruction takes to execute when the pipeline is full and experiencing no delays. Pipeline-protection cycles add to that best-case execution time. As it will be shown, most cases of pipeline conflict can be solved with instruction rescheduling.

This section provides examples to help you to better understand the impact of the pipeline structure on the way your code performs. It also provides you with recommendations for coding style and instruction usage to minimize conflicts or pipeline stalls. This section does not cover all of the pipeline potential conflicts, but some of the most common pipeline delays and IBQ delays found when writing C55x code.

4.4.1 Process to Resolve Pipeline and IBQ Conflicts

A pipeline conflict occurs when one instruction attempts to access a resource before a previous instruction is done with that resource. The pipeline-protection unit adds extra cycles to delay the later instruction.The following process is recommended for resolving pipeline conflicts that are causing delays. Try to focus your pipeline optimization effort on your key, inner code kernels first, to achieve a greater payback.

Step 1: Make your code functional.

Write your code first, without pipeline optimization in mind. In the C55x DSP, the pipeline is protected. Code is executed in the order in which it is written, and stalls are automatically inserted by the hardware to prevent incorrect operation. This makes programming the DSP easier and makes the code easier to debug than an open-pipeline device, in which the sequence of the code might not be the sequence of operation.

Step 2: Determine where the pipeline conflicts exist.

If you are using the C55x simulator, take advantage of its pipeline-conflict detection capabilities. Watch the *clock* variable when stepping through your code in the C55x simulator to view the intervention of the pipeline protection unit. If the clock increments by more than 1, there might be a pipeline or memory conflict in the instruction you just single stepped.

The C55x emulator/debugger does not support the *clock* variable, and setting breakpoints before and after may not give you accurate results for a single instruction due to the initial pipeline fill and the final pipeline flush during single stepping. In this case, you should try to benchmark small pieces of looped code.

Also, C55x simulator version 2.1 or higher offers a pipeline visibility plug-in to help you identify and resolve pipeline conflicts.

Step 3: Apply the pipeline optimization coding recommendations summarized in Table 4–6.

After step 2 or if you are not using the simulator but the emulator, you should try to apply the recommendations directly.

Tip: When suspecting a pipeline conflict between two instructions, try to add NOP instructions in between. If the entire code cycle count does not increase by adding NOPs, then you can try to rearrange your code to replace those NOPs with useful instructions.

Software solutions to apply for pipeline and memory conflicts include:

- ❑ Reschedule instructions.
- ❑ Reduce memory accesses by using CPU registers to hold data.
- ❑ Reduce memory accesses by using a local repeat instruction, an instruction that enables the CPU to repeatedly execute a block of code from the instruction buffer queue.
- ❑ Relocate variables and data arrays in memory, or consider temporarily copying arrays to other nonconflicting memory banks at run time.

4.4.2 Recommendations for Preventing Pipeline Delays

Table 4–6 lists recommendations for avoiding common causes for pipeline delays. The rightmost column of the table directs you to the section that contains the details behind each recommendation. (The examples in this section use instructions from the algebraic instruction set, but the concepts apply equally for the mnemonic instruction set.)

Table 4–6. Recommendations for Preventing Pipeline Delays

Recommendation Category	**Recommendation**	**See ...**
General	❑ In the case of a conflict, the front runner wins.	Section 4.4.2.1, page 4-56
Register access	❑ Avoid consecutive accesses to the same register.	Section 4.4.2.2, page 4-56

Table 4–6. Recommendations for Preventing Pipeline Delays (Continued)

Recommendation Category	Recommendation	See ...
	❑ Use MAR type of instructions, when possible, to modify ARx and Tx registers, but avoid read/write register sequences, and pay attention to instruction size.	Section 4.4.2.3, page 4-60
Pipeline-protection granularity	❑ Pay attention to pipeline-protection granularity when updating status register bit fields.	Section 4.4.2.4, page 4-62
	❑ Pay attention to pipeline-protection granularity when accessing different registers in consecutive instructions.	Section 4.4.2.5, page 4-63
Loop control	❑ Understand when the loop-control registers are accessed in the pipeline	Section 4.4.2.6, page 4-64
	❑ Avoid writing the BRC register in the last few instructions of a block-repeat or local block-repeat structure to prevent an unprotected pipeline situation.	Section 4.4.2.7, page 4-65
	❑ Initialize the BRCx or CSR register at least 4 cycles before the repeat instruction, or initialize the register with an immediate value.	Section 4.4.2.8, page 4-66
Condition evaluation	❑ Try to set conditions well in advance of the time that the condition is tested.	Section 4.4.2.9, page 4-67
	❑ When making an instruction execute conditionally, use XCC instead of XCCPART to create fully protected pipeline conditions, but be aware of potential pipeline delays.	Section 4.4.2.10, page 4-68
	❑ Understand the XCCPART condition evaluation exception.	Section 4.4.2.11, page 4-70

Table 4–6. Recommendations for Preventing Pipeline Delays (Continued)

Recommendation Category	Recommendation	See ...
Memory usage	❑ When working with dual-MAC and FIR instructions, put the Cmem operand in a different memory bank.	Section 4.4.3.2, page 4-78
	❑ Map program code to a dedicated SARAM memory block to avoid conflicts with data accesses.	Section 4.4.3.3, page 4-79
	❑ For 32-bit accesses (using an Lmem operand), no performance hit is incurred if you use SARAM (there is no need to use DARAM).	Section 4.4.3.4, page 4-79
IBQ usage	❑ Align PC discontinuities in 32-bit memory boundaries.	Section 4.4.4, page 4-79
	❑ Use short instructions as the first instructions after a PC discontinuity.	
	❑ Use LOCALREPEAT when possible	

4.4.2.1 In the case of a conflict, the front runner wins.

A pipeline conflict arises when two instructions in different phases in the pipeline compete for the use of the same resource. The resource is granted to the instruction that is ahead in terms of pipeline execution, to increase overall instruction throughput.

4.4.2.2 Avoid consecutive accesses to the same register.

As shown in Figure 4–9, registers are not accessed in the same pipeline phase. Therefore, pipeline conflicts can occur, especially in write/read or read/write sequences to the same register. Following are three common register pipeline conflict cases and how to resolve them.

Case 1: ARx write followed by an indirect addressing ARx read/update

Example 4–15 shows an AR write followed by an indirect-addressing AR read and update. In the example, `I2` has a 4-cycle latency due to pipeline protection. `I2` must wait in the AD phase until `I1` finishes its X phase. `I2` requires 5 cycles (minimum 1 cycle + 4 pipeline-protection cycles).

Example 4–15. A-Unit Register (Write in X Phase/Read in AD Phase) Sequence

```
I1: MOV #k, AR1    ; Load AR1 with constant
                   ;  (AR1 modified in X phase of I1)
I2: MOV *AR1+, AC0 ; Load AC0 with value pointed to by AR1
                   ;  (AR1 read in AD phase of I2)
                   ; Results: AC0 = content of memory at
                   ; location #y16, AR1 = #y16 + 1
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	I2 delayed 4 cycles
	I2		I1				4	
	I2			I1			5	
	I2				I1		6	I1 writes to AR1
	I2					I1	7	I2 reads AR1
		I2					8	
			I2				9	
				I2			10	
					I2		11	
						I2	12	

One solution is instruction rescheduling. Between I1 and I2 you can place 4 cycles worth of instructions from elsewhere.

```
I1: MOV #k, AR1
nop                 ; Replace NOPs with useful instructions
nop
nop
nop
I2: MOV *AR1+, AC0
```

Another solution is to use a MAR instruction to write to AR1:

```
I1: AMOV #k, AR1
I2: MOV *AR1+, AC0
```

A AMOV instruction modifies a register in the AD phase, preventing a pipeline conflict. This solution is covered in more detail in section 4.4.2.3 (page 4-60).

Case 2: ARx read followed by an indirect addressing ARx read/update

Example 4–16 shows an AR read followed by an indirect addressing AR update. In the example, `I2` is delayed 2 cycles due to pipeline protection. `I2` must wait in the AD phase until `I1` finishes its R phase. `I2` executes in 3 cycles (minimum 1 cycle + 2 pipeline-protection cycles). *Notice that AR1 is read by `I1` and is incremented by `I2` in the same cycle (cycle 5). This is due to a special A-unit register prefetch mechanism by a D-unit instruction that reads the A-unit register in the R phase instead of the X phase. Appendix A lists the D-unit instructions that offer this feature.*

Example 4–16. A-Unit Register Read/(Write in AD Phase) Sequence

```
I1: MOV AR1, AC1   ; AR1 read in R phase (A-unit register
                     prefetch)
I2: MOV *AR1+, AC0 ; AR1 updated in AD phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	I2 delayed 2 cycles
	I2		I1				4	
	I2			I1			5	I1 reads AR1 at the beginning of cycle 5 (A-unit register pre-fetch); I2 increments AR1 at the end of cycle 5
		I2			I1		6	
			I2			I1	7	
				I2			8	
					I2		9	
						I2	10	

To prevent 1 to 3 of the pipeline-protection cycles, you can reschedule instructions. If possible, take up to 3 cycles worth of instructions from elsewhere in the program and place them between `I1` and `I2`.

```
MOV AR1, AC1
nop          ; Replace NOPs with useful instructions
nop
MOV *AR1+, AC0
```

One could consider that using a MAR for the ARx register update could also be solution. However as Example 4–16 shows, using MAR can cause unnecessary pipeline conflicts. This is covered in detail in section 4.4.2.3.

Case 3: ACx write followed by an ACx read

Accumulators and registers not associated with address generation are read and written in the X phase, if MMR addressing is not used. Otherwise, the read and write happen in the R and W phases, respectively.

The (write in X phase)/(read in R phase) sequence shown in Example 4–17 costs 1 cycle for pipeline protection. AC0 is updated in the X phase of `I1`. AC0 must be read in the R phase of `I2`, but `I2` must wait until `I1` has written to AC0. The 1 cycle can be regained if you move a 1-cycle instruction between `I1` and `I2`:

```
I1:ADD #1, AC0
   nop   ; Replace NOP with useful instruction
I2:MOV mmap(AC0_L), AC2
```

Notice that

```
ADD #1, AC0
MOV AC0, AC2
```

will not cause pipeline conflicts because AC0 is read by (AC2=AC0) in the X phase. When AC0_L is accessed via the memory map (@AC0_L II mmap()), it is treated as a memory access and read in the read phase.

Example 4–17. Register (Write in X Phase)/(Read in R Phase) Sequence

```
I1: ADD #1, AC0          ; AC0 updated in X phase
I2: MOV mmap(AC0_L), AC2 ; AC0_L read in R phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	
		I2	I1				4	
			I2	I1			5	
				I2	I1		6	I1 updates AC0; I2 delayed
				I2		I1	7	I2 reads AC0
					I2		8	
						I2	9	

4.4.2.3 Use MAR type of instructions, when possible, to modify ARx and Tx registers, but avoid read/write register sequences, and pay attention to instruction size.

The MAR type of instructions (AMOV, AMAR, AADD, ASUB) use independent hardware in the data-address generation unit (DAGEN) to update ARx and Tx registers in the AD phase of the pipeline. You can take advantage of this fact to avoid pipeline conflicts, as shown in Example 4–18. Because AR1 is updated by the MAR instruction prior to being used by I2 for addressing generation, no cycle penalty is incurred.

However, using a MAR instruction could increase instruction size. For example, AADD T1, AR1 requires 3 bytes, while ADD T1, AR1 requires 2 bytes. You must consider the tradeoff between code size and speed.

Example 4–18. Good Use of MAR Instruction (Write/Read Sequence)

```
I1: AMOV #k, AR1       ; AR1 updated in AD phase
I2: MOV *AR1+, AC0     ; AR1 read in AD phase
                       ; (No cycle penalty)
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	I1 updates AR1
	I2	I1					3	I2 reads AR1
		I2	I1				4	
			I2	I1			5	
				I2	I1		6	
					I2	I1	7	
						I2	8	

Example 4–19 shows that sometimes, using a MAR instruction can cause pipeline conflicts. The MAR instruction (I2) attempts to write to AR1 in the AD phase, but due to pipeline protection, I2 must wait for AR1 to be read in the R phase by I1. This causes a 2-cycle latency. Notice that AR1 is read by I1 and is updated by I2 in the same cycle (cycle 5). This is made possible by the A-unit register prefetch mechanism activated in the R phase of the C55x DSP (see page 4-58). I1 is one of the D-unit instructions listed in Appendix A.

One way to avoid the latency in Example 4–19 is to use the code in Example 4–20:

```
I1: ADD mmap(AR1), T2, AC1   ; AR1 read in R phase and
I2: ADD T1, AR1              ; AR1 updated in X phase
                             ; (No cycle penalty)
```

Example 4–19. Bad Use of MAR Instruction (Read/Write Sequence)

```
I1: ADD mmap(AR1), T2, AC1   ; AR1 read in R phase
                             ; (A-unit register prefetch)
I2: AADD T1, AR1             ; AR1 updated in AD phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	I2 delayed 2 cycles
	I2		I1				4	
	I2			I1			5	I1 reads AR1 at the beginning of cycle 5 (A-unit register prefetch); I2 updates AR1 at the end of cycle 5
		I2			I1		6	
			I2			I1	7	
				I2			8	
					I2		9	
						I2	10	

Example 4–20. Solution for Bad Use of MAR Instruction (Read/Write Sequence)

```
I1: ADD mmap(AR1), T2, AC1   ; AR1 read in R phase
                             ; (A-unit register prefetch)
I2: ADD T1, AR1              ; AR1 updated in X phase
                             ; (No cycle penalty)
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	
		I2	I1				4	
			I2	I1			5	I1 reads AR1
				I2	I1		6	
					I2	I1	7	I2 updates AR1
						I2	8	

4.4.2.4 Pay attention to pipeline-protection granularity when updating status register bit fields.

A write to STx can generate the following not-so-obvious pipeline stalls:

❑ **Case 1:** ***Stall during the decode phase when followed by an instruction doing a memory access.***

This is illustrated in Example 4–21. `I2` needs to wait in the D phase until the value of CPL is set by `I1`. This causes an 5-cycle pipeline delay.

Example 4–21. Stall During Decode Phase

```
I1: OR #k, mmap(ST1)
I2: MOV *SP(1), AC2
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
I2		I1					3	
I2			I1				4	
I2				I1			5	
I2					I1		6	
I2						I1	7	ST1 writes by I1 and I2 read CPL in same cycle
	I2						8	
		I2					9	
			I2				10	
				I2			11	
					I2		12	
						I2	13	

- **Case 2: *Stall due to coarse pipeline-protection granularity on STx register.***

  STx register bits can be viewed by the pipeline-protection unit as individual bits or as part of the same pipeline bit group (listed in Table 4–7).

Table 4–7. Bit Groups for STx Registers

Registers	Bit Group Members
ST0	AC0V0, AC0V1, AC0V2, AC0V3
ST1	CPL, C54CM, ARMS
ST2	AR0LC to AR7LC, CDPLC
ST1 and ST3	C54CM, C16, SXMD, SATD, M40, SST

An instruction accessing an STx register bit can have a pipeline conflict with another instruction accessing *a different bit* of the same STx register, if:

- *One of instructions makes a direct reference to the STx register name.*
- *The bits being accessed belong to the same group as listed in Table 4–7.*

  Bits within the same group have *coarse granularity*: you cannot access one bit without delaying the access to other bits within the same group.

  In the example below, `I2` gets delayed 4 cycles until AR1LC gets set because AR1LC and AR2LC belong to the same group.

```
I1: BSET AR7LC        ; AR7LC is written in the
                      ; X phase
I2: MOV  *AR1++%, AC1 ; AR1 circular update in the D
                      ; phase conflicts with the
                      ; AR7LC update because AR1LC
                      ; and AR7LC belong to the same
                      ; STx group.
```

4.4.2.5 *Pay attention to pipeline-protection granularity when accessing different registers in consecutive instructions.*

Registers can be viewed by the pipeline-protection unit as individual registers or as part of the same *pipeline register group* (listed in Table 4–8).

A pipeline delay can occur if two different registers belonging to the same group are accessed at the same time.

Table 4–8. Pipeline Register Groups

Register Group	Group Members
Accumulator 0	AC0L, AC0H, AC0G
Accumulator 1	AC1L, AC1H, AC1G
Accumulator 2	AC2L, AC2H, AC2G
Accumulator 3	AC3L, AC3H, AC3G
Block repeat registers	RSA0, REA0, RSA1, REA1
Transition registers	TRN0, TRN1

4.4.2.6 Understand when the loop-control registers are accessed in the pipeline.

As in any register, the h/w loop controller registers (CSR, RPTC, RSA0, REA0, RSA1, REA1, BRC0, BRC1, and BRS1) can be read and written in:

- ❑ The AD phase (example: `MOV #K12, BRC0`)
- ❑ The R or write phase when accessed using MMR addressing (example: `@BRC0`)
- ❑ The X phase when not in MMR addressing mode.

Loop-control registers can also be modified by the repeat instruction. For example:

- ❑ During RPT loop execution:
  - ■ CSR or an instruction constant is loaded into RPTC in the AD phase of the single RPT instruction.
  - ■ RPTC is tested and decremented in the Decode (D) phase of each repeated instruction.
- ❑ During RPTB and RPTBLOCAL loop execution:
  - ■ BRS1 is loaded into BRC1 in the AD phase of the *blockrepeat* or *localrepeat* instructions.
  - ■ RSAx and REAx are loaded in the AD phase of the *blockrepeat* or *localrepeat* instructions.
  - ■ BRCx is decremented in the decode phase of the last instruction of the loop.
  - ■ RSAx and REAx are constantly read throughout the loop operation.

4.4.2.7 Avoid writing the BRC register in the last few instructions of a block-repeat or local block-repeat structure to prevent an unprotected pipeline situation.

Writing to BRC in one of the last few instructions (exact number depends on the specific instruction) of a block-repeat structure could cause an unprotected pipeline situation. On the other hand, reading BRC is pipeline protected and does not insert an extra pipeline stall cycle.

BRC write accesses may not be protected in the last cycles of a block-repeat structure. Do not write to BRC0 or BRC1 within those cycles. This can be seen in Example 4–22. BRC0 is to be written to by `I1` in the W phase (cycle 7),while BRC0 is decremented in the D phase of `I2`. The pipeline-protection unit *cannot* guarantee the proper sequence of these operations (write to BRC0 and then decrement BRC0). BRC0 is decremented by `I2` before BRC0 changed by `I1`. On the other hand, certain instructions are protected (for example, `BRC0 = #k` is pipeline protected).

Example 4–22. Unprotected BRC Write

```
I1: ADD #1, mmap(BRC0)    ;BRC0 written in W phase
I2: Inst2                 ; Last instruction of a
                          ; block-repeat loop BRC0
                          ; decremented in D phase of last
                          ; instruction
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	CPU decrements BRC0
	I2	I1					3	
		I2	I1				4	
			I2	I1			5	
				I2	I1		6	
					I2	I1	7	I1 changes BRC0 late
						I2	8	

4.4.2.8 *Initialize the BRCx or CSR register at least 4 cycles before the repeat instruction, or initialize the register with an immediate value.*

Whenever BRC1 is loaded, BRS1 is loaded with the same value. In Example 4–23, BRC1 and BRS1 are to be loaded by `I1` in X phase of the pipeline (cycle 6), while BRS1 is to be read by `I2` in the AD phase (cycle 3) to initialize BRC1. The pipeline-protection unit keeps the proper sequence of these operations (write to BRS1 and then read BRS1) by delaying the completion of `I2` by 4 cycles. Example 4–24 shows a similar situation with CSR.

Instruction rescheduling or initialization of BRC1/CSR with an immediate value will remove the pipeline conflict in Example 4–23. An instruction that loads BRC1/CSR with an immediate value will do so in the AD phase of the instruction.

Example 4–23. BRC Initialization

```
I1: MOV *AR1, BRC1     ; BRC1 and BRS1 loaded in X phase
I2: RPTB label         ; BRS1 is copied to BRC1 in AD phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	I2 delayed 4 cycles
	I2		I1				4	
	I2			I1			5	
	I2				I1		6	I1 loads BRS1
	I2					I1	7	I2 copies BRS1 into BRC1
		I2					8	
			I2				9	
				I2			10	
					I2		11	
						I2	12	

Example 4–24. CSR Initialization

```
I1: MOV *AR1, CSR  ; CSR written in X phase
I2: RPT CSR        ; CSR read in AD phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	I2 delayed 4 cycles
	I2		I1				4	
	I2			I1			5	
	I2				I1		6	I1 loads CSR
	I2					I1	7	I2 reads CSR
		I2					8	
			I2				9	
				I2			10	
					I2		11	
						I2	12	

4.4.2.9 Try to set conditions well in advance of the time that the condition is tested.

Conditions are typically evaluated in the R phase of the pipeline with the following exceptions:

❑ For XCC instruction (*execute(AD_unit)* keyword used in algebraic syntax), the condition is evaluated in the AD phase; for example:

```
XCC cond
```

❑ For XCCPART instruction (*execute(D_unit)* keyword used in algebraic syntax), the condition is evaluated in the X phase; for example:

```
XCCPART cond
```

The exception is when the XCCPART instruction conditions a write to memory (see section 4.4.2.11). In this case the condition is evaluated in the R phase; for example:

```
XCCPART cond
|| MOV AC1, *AR1+
```

❑ In an RPTCC, the condition is evaluated in the X phase.

Example 4–25 involves a condition evaluation preceded too closely by a write to the register affecting the condition. AC1 is updated by `I1` in the X phase, while `I2` must read AC1 in the R phase to evaluate the condition (AC1 == #0). The pipeline-protection unit ensures the proper sequence of these operations (write to AC1 and then test AC1) by delaying the completion of `I2` by 1 cycle. The solution is to move `I1` within the program, such that it updates AC1 at least 1 cycle sooner.

Example 4–25. Condition Evaluation Preceded by a X-Phase Write to the Register Affecting the Condition

```
I1: ADD #1, AC1           ; AC1 update in X phase
I2: BCC #label, AC1==#0   ; AC1 test in R phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
	I2	I1					3	
		I2	I1				4	
			I2	I1			5	
				I2	I1		6	I1 updates AC1; I2 delayed
				I2		I1	7	I2 tests AC1
					I2		8	
						I2	9	

4.4.2.10 *When making an instruction execute conditionally, use* **XCC** *instead of* **XCCPART** *to create fully protected pipeline conditions, but be aware of potential pipeline delays.*

- **Use of XCC:** Example 4–26 shows a case where a register load operation (MOV *AR3+, AC1) is made conditional with the XCC instruction. **When XCC is used, the condition is evaluated in the AD phase, and if the condition is true, the conditional instruction performs its operations in the AD through W phases.** In Example 4–26, the AR3+ update in I3 depends on whether AC0 > 0; therefore, the update is postponed until I1 updates AC0. As a result, I3 is delayed by 4 cycles.

- **Use of XCCPART:** One solution to the latency problem in Example 4–26 is to move `I1` such that AC0 is updated 4 cycles earlier. Another solution is to use XCCPART, as shown in Example 4–27. **When XCCPART is used, the condition is evaluated in the X phase, and if the condition is true, the conditional instruction performs its X phase operation.**

Operations in the AD through R phases will happen unconditionally. In Example 4–27, AR3 is updated in the AD phase regardless of the condition. Also, the memory read operation in `I2` will always happen. However, the memory value will be written AC1 only if the condition is true. Otherwise, the memory value is discarded. Overall, a zero-latency execution is achieved.

Notice that the advantage of using XCC is that it conditions the entire effect of the instruction, not just part. However, this could come at the expense of added latency cycles.

Example 4–26. Making an Operation Conditional With XCC

```
I1: ADD #1, AC0         ; AC0 updated in X phase
I2: XCC AC0>#0          ; AC0 tested in AD phase
I3: MOV *AR3+, AC1      ; If AC0 > 0, increment AR3 in AD
                        ; phase and load AC1 in X phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1							1	
I2	I1						2	
I3	I2	I1					3	I2 and I3 delayed 4 cycles
I3	I2		I1				4	
I3	I2			I1			5	
I3	I2				I1		6	I1 updates AC0
I3	I2					I1	7	I2 tests AC0
	I3	I2					8	If AC0 > 0, I3 updates AR3
		I3	I2				9	
			I3	I2			10	
				I3	I2		11	
					I3	I2	12	If AC0 > 0, I3 updates AC1
						I3	13	

Example 4–27. Making an Operation Conditional With execute(D_unit)

```
I1: ADD #1, AC0        ; AC0 updated in X phase
I2: XCCPART AC0>#0     ; AC0 tested in X phase
I3: MOV *AR3+, AC1     ; If AC0 > 0, load AC1 in X phase.
                       ; Update AR3 in AD phase regardless
                        of the condition.
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1								
I2	I1							
I3	I2	I1						
	I3	I2	I1					I3 updates AR3 unconditionally
		I3	I2	I1				
			I3	I2	I1			I1 updates AC0
				I3	I2	I1		I2 tests AC0
					I3	I2		If AC0 > 0, I3 updates AC1
						I3		

4.4.2.11 *Understand the* XCCPART *condition evaluation exception.*

Typically, XCCPART causes the CPU to evaluate the condition in the execute (X) phase. The exception is when you make a memory write operation dependent on a condition, in which case the condition is evaluated in the read (R) phase.

In Example 4–28, AR3 is to be read by `I1` in R phase to evaluate the condition, while AR3 is to be modified by `I3` in the AD phase. The pipeline-protection unit keeps the proper sequence of these operations (read AR3 and then write to AR3) by delaying the completion of `I3` by 2 cycles. Notice that AR3 is tested by `I1` and is modified by `I3` in the same cycle (cycle 5). This is enabled by an A-unit register prefetch mechanism activated in the R phase of the C55x DSP.

To prevent the 2-cycle delay in Example 4–28, you can insert two other, non-conflicting instructions between `I2` and `I3`.

Example 4–28. Conditional Parallel Write Operation Followed by an AD-Phase Write to the Register Affecting the Condition

```
I1: XCCPART AR3==#0      ; AR3 tested in R phase
I2: || MOV AC1, *AR1+    ; If AR3 contains 0, write to
                         ; memory in W phase. Update AR1
                         ; regardless of condition.
I3: MOV *(AR3+T0), AC1   ; AR3 updated in AD phase
```

D	AD	AC1	AC2	R	X	W	Cycle	Comment
I1 ||I2							1	
I3	I1 ||I2						2	I2 updates AR1 unconditionally
	I3	I1 ||I2					3	I3 delayed 2 cycles
	I3		I1 ||I2				4	
	I3			I1 ||I2			5	I1 tests AR3; I3 modifies AR3
		I3			I1 ||I2		6	
			I3			I1 ||I2	7	
				I3			8	
					I3		9	
						I3	10	

4.4.3 Memory Accesses and the Pipeline

This section discusses the impact of the memory mapping of variables/array on the execution of C55x instructions. The two factors to consider are the type of memory (DARAM vs SARAM) and the memory block allocation.

- ❑ **DARAM** supports *2 accesses/cycle* to the same memory block by performing one access in the first half-cycle and the other in the next half-cycle. Table 4–9 lists the accesses performed in each half-cycle, and Table 4–11 shows how the different memory accesses are performed. Address bus loads are omitted from Table 4–11 for simplicity.
- ❑ **SARAM** supports *1 access/cycle* to one memory block.

Table 4–9. Memory Accesses

Type of Access	Bus Used (see Table 4–10)	Half-Cycle Used for DARAM
Instruction fetch	P	First and second halves
Data operand read (smem)	D	Second half
First data operand read (xmem)	D	First half
Second data operand read (ymem)	C	Second half
Third data operand (cmem)	B	Second half
Long (32-bit) data operand read (lmem)	C and D (Note 1)	Second half
Data operand write (smem)	E	First half
Long (32-bit) data operand write (lmem)	E and F (Note 2)	First and second halves
First data operand write (xmem)	F	Second half
Second data operand write (ymem)	E	First half
Coefficient read (cmem)	B	Second half

Notes: 1) Address is driven by D bus. Data is driven in C and D buses.
2) Address is driven by E bus. Data is driven in E and F buses.

Table 4–10. C55x Data and Program Buses

Bus	Description
B	BB. This data-read data bus carries a 16-bit coefficient data value (Cmem) from data space to the CPU.
C, D	CB, DB. Each of these data-read data buses carries a 16-bit data value from data space to the CPU. DB carries a value from data space or from I/O-space. In the case of an *Lmem* (32-bit) read, DB presents the data address but each half of the 32-bit data comes in DB and CB.
E, F	EB, FB. Each of these data-write data buses carries a 16-bit data value to data space from the CPU. EB carries a value to data space or to I/O-space. In the case of an *Lmem* (32-bit) write, EB presents the data address but each half of the 32-bit data comes in EB and FB.
P	PB. This program read bus carries 32-bit instructions from program space to the IBQ.

Table 4–11. Half-Cycle Accesses to Dual-Access Memory (DARAM) and the Pipeline (Note 1)

Access Type	D	AD	AC1	AC2		R	X	W	W+	
Smem read					D					
Smem write									E	
Lmem read				C	D					
Lmem write									E	F
Xmem read II Ymem read				C	D					
Xmem write II Ymem write									E	F
Xmem read II Ymem read II Cmem read (Note 2)				C	D B					
Xmem read II Ymem write					D				E	
Lmem read II Lmem write				C	D				E	F

Notes: 1) For detailed memory access description, refer to section 4.4.3.1.
2) B (Cmem) should be in a different DARAM block than the D (Xmem) operand to avoid one delay cycle.

4.4.3.1 Memory Access and the Pipeline

A CPU memory access is pipelined across three clock cycles as described in Table 4–12.

- **For a read operation:** In the first cycle (request cycle), the request and address are placed on the bus. In the second cycle (memory read cycle), the read access is done to the memory. In the third cycle (data read cycle), the data is delivered to the buses.

- **For a write operation:** In the first cycle (request cycle), the request and address are placed on the bus. In the second cycle (data write cycle), the data is written to the buses. In the third cycle (memory write), the write accesses are done to the memory.

Table 4–12. Memory Accesses and the Pipeline

Pipeline Phase	AC1	AC2	R	X	W	W+
Read operation	Request	Memory read	Data read			
Write operation				Request	Data written	Memory written

The *memory access* happens in the AC2 phase for reads and in the W+ phase for writes.

As seen in Table 4–9, two simultaneous acceses can occur to the same DARAM block, and only one access to a SARAM block.

Ideally, we should allocate all data into DARAM due to its higher memory bandwidth (2 accesses/cycle). However, DARAM is a limited resource and should be used only when it is advantageous. Following are recommendations to guide your memory mapping decisions.

❑ Reschedule instructions.

❑ Reduce memory accesses by using CPU registers to hold data.

❑ Reduce memory accesses by using a local repeat instruction, an instruction that enables the CPU to repeatedly execute a block of code from the instruction buffer queue.

❑ Relocate variables and data arrays in memory, or consider temporarily copying arrays to other nonconflicting memory banks at run time.

It is important to note that in case of a memory access contention between CPU and DMA, the C55x has the following bus priority:

❑ CPU data accesses (highest priority)

❑ CPU program access

❑ DMA access (lowest priority)

Following are some typical memory conflicts:

Case 1. A write-pending case: Write conflicting with a Dual-Operand Read

Example 4–29 shows a conflict between a write and a dual-operand read. In this case, there is a conflict between the operand write access (E bus) in W+ and the second data read access (C bus) in AC2. This conflict is known as *write pending* and is resolved automatically by delaying the write access by one cycle (write *pends*).

The actual execution time of these instructions does not increase, because the delayed (pending) write memory access (I1) is performed while the read instruction (I5) is in the R phase.

Example 4–29. A Write Pending Case

```
I1: MOV AC0, *AR3+        ; Write happens in the W+ phase
I2: NOP
I3: NOP
I4: NOP
I5: ADD *AR4+, *AR5+, A   ; Dual read happens in the R
                          ; phase
```

D	AD	AC1	AC2	R	X	W	W+	Cycle	Comment
I1								1	
I2	I1							2	
I3	I2	I1						3	
I4	I3	I2	I1					4	
I5	I4	I3	I2	I1				5	
	I5	I4	I3	I2	I1			6	
		I5	I4	I3	I2	I1		7	
			I5	I4	I3	I2	I1	8	I1 write conflicts with I5 reads. I1 write pends in the memory buffers and reads complete.
				I5	I4	I3	I2	9	I1 write completes.
					I5	I4	I3	10	
						I5	I4	11	
							I5	12	

Example 4–30. A Memory Bypass Case

```
I1: MOV AC0, *AR3+        ; Initially, AR3 and AR4 have the
                          ; same values. Write happens in
                          ; the W+ phase.
I2: ADD *AR4+, *AR5+, A   ; Dual read happens in the R
                          ; phase
```

D	AD	AC1	AC2	R	X	W	W+	Cycle	Comment
I1								1	
I2	I1							2	
	I2	I1						3	
		I2	I1					4	
			I2	I1				5	I2 tries to read from memory. Pipeline detects I1 instruction processing and waits for “memory bypass” in R phase.
				I2	I1			6	I2 tries to read from internal buses, but has to wait for I2 W phase.
				I2		I1		7	I1 writes data to internal buses in W phase. I2 delayed one cycle.
					I2		I1	8	I2 can now proceed.
						I2		9	
							I2	10	

Case 2. A memory-bypass case:

If in Example 4–30, any read access (via D or C buses) is from the *same* memory location in memory where the write access should occur, the CPU bypasses reading the actual memory location; instead, it reads the data directly from the internal buses. This allows the pipeline to perform a memory write access in a later pipeline phase than that in which the next instruction reads from the same memory location. Without this memory bypass feature, the delay would have been 3 cycles.

4.4.3.2 When working with dual-MAC and FIR instructions, put the Cmem operand in a different memory bank.

Provided code is not executed from the same memory block in which you have the data being accessed by that code, the only memory access type which can generate a conflict in a DARAM is the execution of instructions requiring three data operands in 1 cycle: Xmem, Ymem, and Cmem (coefficient operand). Examples of two commonly used instructions that use three data operands are:

- Dual multiply-and-accumulate (MAC) instruction:

```
MAC Xmem, Cmem, ACx
:: MAC Ymem, Cmem, ACy
```

- Finite impulse response filter instructions:

```
FIRSADD Xmem, Ymem, Cmem, ACx, ACy
FIRSSUB X mem, Ymem, Cmem, ACx, ACy
```

This memory conflict can be solved by maintaining the Ymem and Xmem operands in the same DARAM memory bank but putting the Cmem operand into a different memory bank (SARAM or DARAM).

When cycle intensive DSP kernels are developed, it is extremely important to identify and document software integration recommendations stating which variables/arrays must not be mapped in the same DARAM memory block.

The software developer should also document the associated cycle cost when the proposed optimized mapping is not performed. That information will provide the software integrator enough insight to make trade-offs.

When cycle intensive DSP kernels are developed, it is extremely important to identify and document software integration recommendations stating which variables/arrays must not be mapped in the same dual access memory. The software developer should also document the associated cycle cost when the proposed optimized mapping is not performed. That information will provide the software integrator enough insight to make trade-offs. Table 4–13 provides an example of such table: if the 3 arrays named “input.” “output,” and “coefficient” are in the same DARAM, the subroutine named “filter” will have 200 cycle overhead.

Table 4–13. Cross-Reference Table Documented By Software Developers to Help Software Integrators Generate an Optional Application Mapping

Routine	Cycle Weight	Array 1 Name (Xmem)	Size	Array 2 Name (Ymem)	Size	Array 3 Name (Cmem)	Size	Cycle Cost per Routine
filter	10%	input	40	output	40	coefficient	10	10*20

4.4.3.3 Map program code to a dedicated SARAM memory block to avoid conflicts with data accesses.

If a DARAM block maps both program and data spaces of the same routine, a program code fetch will conflict, for example, with a dual (or triple) data operand read (or write) access if they are performed in the same memory block. C55x DSP resolves the conflict by delaying the program code fetch by one cycle. It is therefore recommended to map the program code in a dedicated program memory bank: generally a SARAM memory bank is preferred. This avoids conflicts with data variables mapped in the high bandwidth DARAM banks.

Another way to avoid memory conflicts is to use the 56-byte IBQ to execute blocks of instructions without refetching code after the 1st iteration (see *localrepeat{}* instruction). Conflict will only occur in the first loop iteration.

4.4.3.4 For 32-bit accesses (using an Lmem operand), no performance hit is incurred if you use SARAM (there is no need to use DARAM).

When a 32-bit memory access is performed with Lmem, only one *address* bus (DAB or EAB) is used to specify the most and least significant words of the 32-bit value. Therefore, reading from or writing to a 32-bit memory location in an SARAM bank occurs in 1 cycle.

4.4.4 Recommendations for Preventing IBQ Delays

The IBQ is filled by the *fetch pipeline* in 32-bit instruction packets each aligned in a 32-bit memory boundary. Instructions stored in the IBQ are dispatched to the *execution pipeline* in 48-bit packets and consumed by the decode (D) phase of the pipeline at a rate that depends on the length of the instruction being decoded. The number of bytes waiting in the IBQ to be decoded is known as *IBQ fetch advance* and should be greater than the instruction length to be decoded in order to have enough data to feed the execution pipeline. Otherwise an IBQ stall in the fetch pipeline will occur. IBQ fetch advance can be between 0 and 24 bytes.

4.4.4.1 IBQ Performance on PC Discontinuities

When a PC discontinuity occurs (a subroutine call, a branch, a block repeat loop,...), the IBQ content is flushed and the fetch pipeline starts filling the IBQ with the instructions at the target address. One delay cycle will occur if the first 32-bit aligned memory fetch packet does not contain a complete parallel instruction pair (for example, branching to a 4-byte-aligned nop_16||nop_16 does not cause an extra stall) or a complete single instruction and the first byte of the following instruction.

To avoid IBQ delay cycles and increase the chances of the first 32-bit packet to contain the target address instruction and the first byte of the next instruction, following are recommendations:

- *Align PC discontinuities in a 32-bit memory boundary*

  This means for example, to align

  - the starting address of a subroutine
  - the first instruction inside a block repeat

- *Use short instructions at the target address of PC discontinuities*. However, if the size of the first instruction after the PC discontinuity is 4 bytes or larger there will be a delay of 1 cycle.

To better understand the effect of the IBQ in code cycles, we present 3 typical PC discontinuitites cases:

Case 1: IBQ Behavior During Branches

Examples 1a, 1b, and 1c show branch examples where the IBQ behavior is illustrated.

Example 1a: No Delay

```
         B label1
         .....
         ....
         .align 4              ; force 32-bit alignment
label1:  XOR ac1 << #1, ac0    ; 3 bytes
         MOV t0, ac2           ; 2 bytes
```

By using the .align 4 assembler directive, we are able to avoid an IBQ stall. The first 32-bit fetch packet at the PC discontinuity target address, *label1*, contains the first instruction and the first byte of the next instruction.

Example 1b: One Delay

```
         B label2
         .....
         ....
         .align 4              ; force 32-bit alignment
label2:  MOV AC0, *AR3-
         || MOV #15, BRC1      ; 5 bytes
         MOV t0, ac2           ; 2 bytes
```

In this case, even though we align the target address in a 32-bit boundary, the first packet cannot contain the first instruction and the first byte of the next instruction. Using a shorter instruction at the target address (less than 3 bytes) can prevent this delay cycle. For example, we can rearrange the parallelism as shown in Example 1c.

Example 1c: No Delay

```
         B label2
         .....
         .align 4
label2:  MOV AC0, *AR3-         ; 2 bytes
         MOV #15, BRC1
         || MOV t0, ac2         ; 5 bytes
```

Case 2: IBQ Behavior During Repeat Block

IBQ delays can also occur when the PC discontinuity is caused by a block repeat loop. During a block repeat processing the IBQ fetches sequentially the instructions until it reaches the last instruction. At this point, since the IBQ does not know what the size of the last instruction is (it only knows the address of the last instruction), it fetches 7 more bytes. Thus, the IBQ may be fetching more program bytes than it actually needs. This overhead can cause some delays. The following example will show this:

Example 2

```
   .align 4
4000   nop
4001   rptb label3
4004   amov #0x8000 , xar0
400a   ;;nop_16       ;← commented nop_16 instruction
400a   nop_16 || nop_16  ;#1
400e   nop_16 || nop_16  ;#2
4012   nop_16 || nop_16  ;#3
4016   nop_16 || nop_16  ;#4
401a   nop_16 || nop_16  ;#5
401e   nop_16 || nop_16  ;#6
4022   nop_16 || nop_16  ;#7
4026   nop_16 || nop_16  ;#8
402a   nop_16 || nop_16  ;#9
       label3:
402e   nop_16 || nop_16  ;#10
4032
```

The block repeat loop has 11 instructions. Since the size of the first instruction is 6 bytes, the total cycle count for this loop should be 12 cycles since it requires two 32-bit fetch packets to have the entire first instruction (AMOV) and the first byte of the next instruction. However, this code actually takes 13 cycles, as illustrated below.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Program Address Bus	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	04	08	0c	10	14	18	
Decode PC						04	0a	0e	12	16	1a	1e	22	26	2a	2e	**X**	**X**	04	

Note:
- The Cycle row corresponds to the cycle timeline.
- The Program Address Bus row corresponds to the address of the 32-bit block requested from memory (fetch packet). (For simplifying the notation, the 0x4004 is written as 04.)
- The Decode PC row corresponds to the address of the instruction that is being decoded.

There are two **X** slots that are used to fetch packets that are never decoded. Therefore, we have 11 instructions for 13 cycles. One of the **X** slots is unavoidable because of the first 6-byte instruction (AMOV). The second, however, could be used by another instruction. For example, if we uncomment the nop_16 instruction, the cycle count will be the same—13 cycles—indicating the opportunity to rearrange your code and insert a useful instruction instead of the nop_16. This kind of delay can occur in block repeat loops but it does not occur in local repeat loops. For this reason, the use of RPTBLOCAL is recommended. To avoid this kind of delay, use the following recommendation:

❑ *Use RPTBLOCAL when possible*. It will also save power.

IBQ delays can also occur when there are many 5- or 6-byte instructions consecutively. This will reduce the fetch advance and eventually produce delays. To avoid this kind of delay, use the following recommendation:

❑ *Avoid using too many 5- or 6-byte instructions (or 5- or 6-byte parallel instructions) consecutivel*y. Doing so will eventually starve the IBQ. Instead, mix in short instructions to balance the longer instructions, keeping in mind that the average sustainable fetch, without incurring a stall, is 4 bytes.

Case 3: IBQ Behavior During Local Repeat Block

The RPTBLOCAL loop differs from a RPTB loop in that the size of the content of the loop is able to fit completely in the IBQ. The maximum size of a RPTBLOCAL is 55 bytes between the address of the first instruction and the last instruction of the loop. If the last instruction of the loop is a 6-byte instruction then the size of the loop is 61 bytes. During the processing of the loop the IBQ keeps fetching instructions until it reaches the maximum fetch advance which is 24 bytes.

Since all the instructions of the local loop are in the IBQ, **there are no IBQ delays inside a RPTBLOCAL loop**.

Special Case: IBQ Delays When the size of the Local Loop is Close to 61 Bytes

There is not much space left in the IBQ to fetch additional instructions. **In this case, after the loop completes, IBQ delays in the first instruction outside the loop may occur**. This delay will occur when the IBQ did not have enough space to fetch the first instruction following the loop plus the first byte of the second instruction (see Example 3). This type of IBQ delay could range from 1 to 6, depending on the size of the local loop and the length of the instruction following the local loop.

Example 3: Delays After the Completion of a RPTBLOCAL Loop

```
         RPTBLOCAL  label4

; loop is 61 bytes

0x4000   NOP_16|| NOP_16              ; 4 bytes
0x4004   NOP_16|| NOP_16              ; 4 bytes
0x4008   NOP_16|| NOP_16              ; 4 bytes
0x400c   NOP_16|| NOP_16              ; 4 bytes
0x4010   NOP_16|| NOP_16              ; 4 bytes
0x4014   NOP_16|| NOP_16              ; 4 bytes
0x4018   NOP_16|| NOP_16              ; 4 bytes
0x401c   NOP_16|| NOP_16              ; 4 bytes
0x4020   NOP_16|| NOP_16              ; 4 bytes
0x4024   NOP_16|| NOP_16              ; 4 bytes
0x4028   NOP_16|| NOP_16              ; 4 bytes
0x402c   NOP_16|| NOP_16              ; 4 bytes
0x4030   NOP_16                       ; 2 bytes
0x4032   MAXDIFF AC0, AC1, AC2, AC1   ; 5 bytes
         ||MOV #0, AC3
label4:
0x4037   SUBADD T3,*mold_ptr+,AC0     ; 6 bytes
         ||AADD #1, mnew_ptr
0x403d   NOP_16|| NOP_16              ; 4 bytes
```

The size of the RPTBLOCAL loop in the previous example is 61 bytes. Since the size of the IBQ is 64 bytes, and since, in this case, the content of the loop is aligned on a 32-bit boundary, the IBQ fetches also the first three bytes of the first instruction after the loop. However, that is not sufficient to avoid delays as the first instruction outside the loop (at 0x403d) address is a 4-byte instruction. Therefore, in this case, five IBQ delays will occur after the completion of the loop.

4.4.4.2 The Speculative Pre-Fetch Feature

The C55x offers a speculative pre-fetch feature that can save several execution cycles of conditional control flow instructions when the condition detected is “true.” For example, in the case of a conditional branch (BCC) with an immediate value, the branch target address is known in the decode (D) phase (in the case of an immediate value) or in the address (AD) phase (in the case of a relative offset) of the pipeline but the condition is evaluated later in the Read (R) phase. To avoid the extra cycle delays that could imply to wait for the condition to be known before fetching the target address, the C55x fetches the branch target address speculatively and the fetch packet is stored in the IBQ. If the condition is evaluated as “true,” then the instruction decoder can get the branch target instruction from the IBQ with minimum latency.

Chapter 5

Fixed-Point Arithmetic

The TMS320C55x (C55x) DSP is a 16-bit, fixed-point processor. This chapter explains important considerations for performing standard- and extended-precision fixed-point arithmetic with the DSP. Assembly-language code examples are provided to demonstrate the concepts.

Topic **Page**

5.1 Fixed-Point Arithmetic – a Tutorial

Digital signal processors (DSPs) have been developed to process complex algorithms that require heavy computations. DSPs can be divided into two groups: floating-point DSPs and fixed-point DSPs.

Typically, floating-point DSPs use 32-bit words composed of a 24-bit mantissa and an 8-bit exponent, which together provide a dynamic range from 2^{-127} to $2^{128}(1 - 2^{-23})$. This vast dynamic range in floating-point devices means that dynamic range limitations may be virtually ignored in a design. Floating-point devices are usually more expensive and consume more power than fixed-point devices.

Fixed-point DSPs, like the TMS320C55x DSP, typically use 16-bit words. They use less silicon area than their floating-point counterparts, which translates into cheaper prices and less power consumption. Due to the limited dynamic range and the rules of fixed-point arithmetic, a designer must play a more active role in the development of a fixed-point DSP system. The designer has to decide whether the 16-bit words will be interpreted as integers or fractions, apply scale factors if required, and protect against possible register overflows.

5.1.1 2s-Complement Numbers

In binary form, a number can be represented as a signed magnitude, where the left-most bit represents the sign and the remaining bits represent the magnitude.

+52 = 0 011 0100b

–52 = 1 011 0100b

This representation is not used in a DSP architecture because the addition algorithm would be different for numbers that have the same signs and for numbers that have different signs. The DSP uses the 2s-complement format, in which a positive number is represented as a simple binary value and a negative value is represented by inverting all the bits of the corresponding positive value and then adding 1.

Example 5–1 shows the decimal number 353 as a 16-bit signed binary number. Each bit position represents a power of 2, with 2^0 at the position of the least significant bit and 2^{15} at the position of the most significant bit. The 0s and 1s of the binary number determine how these powers of 2 are weighted (times 0 or times 1) when summed to form 353. Because the number is signed, 2^{15} is given a negative sign. Example 5–2 shows how to compute the negative of a 2s-complement number.

Example 5–1. Signed 2s-Complement Binary Number Expanded to Decimal Equivalent

2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**

$$\begin{aligned} =\ & (0 \times (-2^{15})) + (0 \times 2^{14}) + (0 \times 2^{13}) + (0 \times 2^{12}) + (0 \times 2^{11}) + (0 \times 2^{10}) + (0 \times 2^{9}) + (1 \times 2^{8}) + (0 \times 2^{7}) \\ & + (1 \times 2^{6}) + (1 \times 2^{5}) + (0 \times 2^{4}) + (0 \times 2^{3}) + (0 \times 2^{2}) + (0 \times 2^{1}) + (1 \times 2^{0}) \\ =\ & (1 \times 2^{8}) + (1 \times 2^{6}) + (1 \times 2^{5}) + (1 \times 2^{0}) \\ =\ & 256 + 64 + 32 + 1 = 353 \end{aligned}$$

Example 5–2. Computing the Negative of a 2s-Complement Number

```
Begin with a positive binary number (353 decimal):      0000 0001 0110 0001

Invert all bits to get the 1s complement:                1111 1110 1001 1110
Add 1 to get the 2s complement:                        +                   1
                                                       ---------------------
Result: negative binary number (–353 decimal):           1111 1110 1001 1111
```

5.1.2 Integers Versus Fractions

The most common formats used in DSP programming are integers and fractions. In signal processing, fractional representation is more common. A fraction is defined as a ratio of two integers such that the absolute value of the ratio is less than or equal to 1. When two fractions are multiplied together, the result is also a fraction. Multiplicative overflow, therefore, never occurs. Note, however, that additive overflow can occur when fractions are added. Overflows are discussed in section 5.5, beginning on page 5-24.

Figure 5–1 shows how you can interpret 2s-complement numbers as integers. The most significant bit (MSB) is given a negative weight, and the integer is the sum of all the applicable bit weights. If a bit is 1, its weight is included in the sum; if the bit is 0, its weight is not applicable (the effective weight is 0). For simplicity, the figure shows 4-bit binary values; however, the concept is easily extended for larger binary values. Compare the 4-bit format in Figure 5–1 with the 8-bit format in Figure 5–2. The LSB of a binary integer always has a bit weight of 1, and the absolute values of the bit weights increase toward the MSB. Adding bits to the left of a binary integer does not change the absolute bit weights of the original bits.

Figure 5–1. 4-Bit 2s-Complement Integer Representation

4-bit 2s-complement integer	MSB			LSB	
Bit weights	$-2^3 = -8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
Least positive value	0	0	0	1	= 0 + 0 + 0 + 1 = 1
Most positive value	0	1	1	1	= 0 + 4 + 2 + 1 = 7
Least negative value	1	1	1	1	= –8 + 4 + 2 + 1 = –1
Most negative value	1	0	0	0	= –8 + 0 + 0 + 0 = –8
Other examples:	0	1	0	1	= 0 + 4 + 0 + 1 = 5
	1	1	0	1	= –8 + 4 + 0 + 1 = –3

Figure 5–2. 8-Bit 2s-Complement Integer Representation

MSB							LSB
$-2^7 = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

Figure 5–3 shows how 2s-complement numbers can be interpreted as a fractions. The concept is much the same as that in Figure 5–1, but the bit weights are fractional, meaning that the number cannot have an absolute value larger than 1. Compare the 4-bit format in Figure 5–3 with the 8-bit format in Figure 5–4. The MSB of a binary fraction always has a bit weight of –1, and the absolute values of the bit weights decrease toward the LSB. Unlike adding bits to the left of a binary integer, adding bits to the left of a binary fraction changes the bit weights of the original bits.

Figure 5–3. 4-Bit 2s-Complement Fractional Representation

4-bit binary fraction	MSB			LSB	
Bit weights	$-2^0 = -1$	$2^{-1} = 1/2$	$2^{-2} = 1/4$	$2^{-3} = 1/8$	
Least positive value	0	0	0	1	= 0 + 0 + 0 +1/8 = 1/8
Most positive value	0	1	1	1	= 0 + 1/2+ 1/4+ 1/8 = 7/8
Least negative value	1	1	1	1	= –1 + 1/2 + 1/4 + 1/8 = –1/8
Most negative value	1	0	0	0	= –1 + 0 + 0 + 0 = –1
Other examples:	0	1	0	1	= 0 + 1/2 + 0 + 1/8 = 5/8
	1	1	0	1	= –1 + 1/2 + 0 + 1/8 = –3/8

Figure 5–4. 8-Bit 2s-Complement Fractional Representation

MSB							LSB
$-2^0 = -1$	$2^{-1} = 1/2$	$2^{-2} = 1/4$	$2^{-3} = 1/8$	$2^{-4} = 1/16$	$2^{-5} = 1/32$	$2^{-6} = 1/64$	$2^{-7} = 1/128$

5.1.3 2s-Complement Arithmetic

An important advantage of the 2s-complement format is that addition is performed with the same algorithm for all numbers. To become more familiar with 2s-complement binary arithmetic, refer to the examples in this section. You may want to try a few examples yourself. It is important to understand how 2s-complement arithmetic is performed by the DSP instructions, in order to efficiently debug your program code.

Example 5–3 shows two 2s-complement additions. These binary operations are completely independent of the convention the programmer uses to convert them into decimal numbers. To highlight this fact, an integer interpretation and a fractional interpretation are shown for each addition. For simplicity, the examples use 8-bit binary values; however, the concept is easily extended for larger binary values. For a better understanding of how the integer and fractional interpretations were derived for the 8-bit binary numbers, see Figure 5–2 (page 5-4) and Figure 5–4 (page 5-5), respectively.

Example 5–3. Addition With 2s-Complement Binary Numbers

2s-Complement Addition	Integer Interpretation	Fractional Interpretation
1 (carry)		
0000 0101	5	5/128
+ 0000 0100	+ 4	+ 4/128
––––––––––––––	––––	–––––––
0000 1001	9	9/128
1 1 1 (carries)		
0000 0101	5	5/128
+ 0000 1101	+ 13	+ 13/128
––––––––––––––	–––––	––––––––
0001 0010	18	18/128

Example 5–4 shows subtraction. As with the additions in Example 5–3, an integer interpretation and a fractional interpretation are shown for each computation. It is important to notice that 2s-complement subtraction is the same as the addition of a positive number and a negative number. The first step is to find the 2s-complement of the number to be subtracted. The second step is to perform an addition using this negative number.

Example 5–4. Subtraction With 2s-Complement Binary Numbers

2s-Complement Subtraction	Integer Interpretation	Fractional Interpretation
Original form:		
0000 0101	5	5/128
– 0000 0100	– 4	– 4/128
––––––––––––––	––––	–––––––
2s complement of subtracted term:		
11 (carries)		
1111 1011		
+ 1		
––––––––––––––		
1111 1100		
Addition form:		
11111 1 (carries)		
0000 0101	5	5/128
+ 1111 1100	+ (–4)	+ (–4/128)
––––––––––––––	––––––	–––––––––
0000 0001	1	1/128
(final carry ignored)		
Original form:		
0000 0101	5	5/128
– 0000 1101	– 13	– 13/128
––––––––––––––	–––––	––––––––
2s complement of subtracted term:		
1111 0010		
+ 1		
––––––––––––––		
1111 0011		
Addition form:		
111 (carries)		
0000 0101		
+ 1111 0011	5	5/128
––––––––––––––	+ (–13)	+ (–13/128)
1111 1000	–––––––	––––––––––
	–8	–8/128

Example 5–5 shows 2s-complement multiplication. For simplicity, the example uses a 4-bit by 4-bit multiplication and assumes an 8-bit accumulator for the result. Notice that the 7-bit mathematical result is sign extended to fill the accumulator. The C55x DSP sign extends multiplication results in this way, extending the result to either 32 bits or 40 bits. The effects of this type of sign extension can be seen in the integer and fractional interpretations of Example 5–5. The integer is not changed by sign extension, but the fraction can be misinterpreted. Sign extension adds an extra sign bit. If your program assumes that the MSB of the result is the only sign bit, you must shift the result left by 1 bit to remove the extra sign bit. In the C55x DSP, there is a control bit called FRCT to automate this shift operation. When FRCT = 1, the DSP automatically performs a left shift by 1 bit after a multiplication. You can clear and set FRCT with the following instructions:

```
BCLR FRCT    ; Clear FRCT
BSET FRCT    ; Set FRCT
```

Example 5–5. Multiplication With 2s-Complement Binary Numbers

2s-Complement Multiplication		**Integer Interpretation**	**Fractional Interpretation**
0100	Multiplicand	4	4/8
x 1101	Multiplier	x (–3)	x (–3/8)
–––––––		––––––	–––––––––
0000100			
000000			
00100			
1100	(see **Note**)		
–––––––––			
1110100	7-bit mathematical result	–12	–12/64 (The MSB of the result is the only sign bit)
11110100	8-bit sign-extended result in accumulator	–12	–12/64 if properly interpreted; –12/128 if incorrectly interpreted. To remove extra sign bit in MSB position, shift result left by 1 bit.
Note: Because the MSB is a sign bit, the final partial product is the 2s complement negative of the multiplicand.			

5.2 Extended-Precision Addition and Subtraction

Numerical analysis, floating-point computations, and other operations may require arithmetic operations with more than 32 bits of precision. Because the C55x device is a 16-/32-bit fixed-point processor, 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 C55x DSP has several features that help make extended-precision calculations more efficient.

- **CARRY bit:** One of the features is the CARRY status bit, which is affected by most arithmetic D-unit ALU instructions, as well as the rotate and shift operations. CARRY depends on the M40 status bit. When M40 = 0, the carry/borrow is detected at bit position 31. When M40 = 1, the carry/borrow reflected in CARRY is detected at bit position 39. Your code can also explicitly modify CARRY by loading ST0_55 or by using a status bit clear/set instruction. For proper extended-precision arithmetic, the saturation mode bit should be cleared (SATD = 0) to prevent the accumulator from saturating during the computations.
- **32-bit addition, subtraction, and loads:** Two C55x 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 extended-precision arithmetic.
- **16-bit signed/unsigned multiplication:** The hardware multiplier can multiply 16-bit signed/unsigned numbers, as well as multiply two signed numbers and two unsigned numbers. This makes 32-bit multiplication operations efficient.

5.2.1 A 64-Bit Addition Example

The code in Example 5–6 adds two 64-bit numbers to obtain a 64-bit result. The following are some code highlights:

- The partial sum of the 64-bit addition is efficiently performed by the following instructions, which handle 32-bit operands in a single cycle.

Mnemonic instructions:	MOV40 dbl(Lmem), ACx
	ADD dbl(Lmem), ACx
Algebraic instructions:	ACx = dbl(Lmem)
	ACx = ACx + dbl(Lmem)

- For the upper half of a partial sum, the instruction that follows this paragraph uses the carry bit generated in the lower 32-bit partial sum. Each partial sum is stored in two memory locations using MOV ACx, dbl(Lmem) or dbl(Lmem) = ACx.

Mnemonic instruction:	ADD uns(Smem), CARRY, ACx
Algebraic instruction:	ACx = ACx + uns(Smem) + CARRY

❑ Typically, if a computation does not generate a carry, the CARRY bit is cleared.

As shown in Figure 5–5, the ADD instruction with a 16-bit shift (shown following this paragraph) is an exception because it can only set the CARRY bit. If this instruction does not generate a carry, the CARRY bit is left unchanged. This allows the D-unit ALU to generate the appropriate carry when adding to the lower or upper half of the accumulator causes a carry.

Mnemonic instruction:	ADD Smem << #16, ACx, ACy
Algebraic instruction:	ACy = ACx + (Smem << #16)

For illustration purposes, Figure 5–5 shows the normal effect of several 32-bit additions on the CARRY bit (referred to as C in the figure).

Example 5–6. 64-Bit Addition

```
;*********************************************************************
; 64-Bit Addition         Pointer assignments:
;
;     X3 X2 X1 X0         AR1 -> X3  (even address)
;   + Y3 Y2 Y1 Y0                X2
;   --------------               X1
;     W3 W2 W1 W0                X0
;                         AR2 -> Y3  (even address)
;                                Y2
;                                Y1
;                                Y0
;                         AR3 -> W3  (even address)
;                                W2
;                                W1
;                                W0
;
;*********************************************************************

   MOV40  dbl(*AR1(#2)), AC0            ; AC0 = X1 X0
   ADD    dbl(*AR2(#2)), AC0            ; AC0 = X1 X0 + Y1 Y0
   MOV    AC0,dbl(*AR3(#2))             ; Store W1 W0.
   MOV40  dbl(*AR1), AC0                ; AC0 = X3 X2
   ADD    uns(*AR2(#1)),CARRY,AC0       ; AC0 = X3 X2 + 00 Y2 + CARRY
   ADD    *AR2<< #16, AC0               ; AC0 = X3 X2 + Y3 Y2 + CARRY
   MOV    AC0, dbl(*AR3)                ; Store W3 W2.
```

Note: The algebraic instructions code example for 64-Bit Addition is shown in Example B–27 on page B-30.

Figure 5–5. Effect on CARRY of Addition Operations

```
C  MSB              LSB       C  MSB              LSB
X  F F F F F F F F F F ACx    X  F F F F F F F F F F ACx
   +                  1         +F F F F F F F F F F
1  0 0 0 0 0 0 0 0 0 0        1  F F F F F F F F F E

C  MSB              LSB       C  MSB              LSB
X  0 0 7 F F F F F F F ACx    X  0 0 7 F F F F F F F ACx
   +                  1         +F F F F F F F F F F
0  0 0 8 0 0 0 0 0 0 0        1  0 0 7 F F F F F F E

C  MSB              LSB       C  MSB              LSB
X  F F 8 0 0 0 0 0 0 0 ACx    X  F F 8 0 0 0 0 0 0 0 ACx
  +                   1         +F F F F F F F F F F
0  F F 8 0 0 0 0 0 0 1        1  F F 7 F F F F F F F

           ACy = ACx + Smem + CARRY

C  MSB              LSB       C  MSB              LSB
1  0 0 0 0 0 0 0 0 0 0 ACx    1  F F F F F F F F F F ACx
  +                   0          +                  0
0  0 0 0 0 0 0 0 0 0 1 ACy    1  0 0 0 0 0 0 0 0 0 0

Special Case:   ACy = ACx + (Smem <<16)

C  MSB              LSB       C  MSB              LSB
1  F F 8 0 0 0 F F F F ACx    1  F F 8 0 0 0 F F F F ACx
  +0 0 0 0 0 1 0 0 0 0          +0 0 7 F F F 0 0 0 0
1  F F 8 0 0 1 F F F F ACy    1  F F F F F F F F F F

C keeps a values of 1, even when a CARRY is not generated.
```

5.2.2 A 64-Bit Subtraction Example

Example 5–7 subtracts two 64-bit numbers to obtain a 64-bit result. The following are some code highlights:

- ❑ The partial remainder of the 64-bit subtraction is efficiently performed by the following instructions, which handle 32-bit operands in a single cycle.

Mnemonic instructions:	MOV40 dbl(Lmem), ACx
	SUB dbl(Lmem), ACx
Algebraic instructions:	ACx = dbl(Lmem)
	ACx = ACx – dbl(Lmem)

- ❑ For the upper half of the partial remainder, the instruction that follows this paragraph uses the borrow generated in the lower 32-bit partial remainder. The borrow is not a physical bit in a status register; it is the logical inverse of CARRY. Each partial sum is stored in two memory locations using MOV ACx, dbl(Lmem) or dbl(Lmem) = ACx.

Mnemonic instruction: SUB uns(Smem), BORROW, ACx
Algebraic instruction: ACx = ACx – uns(Smem) – BORROW

- ❑ Typically, if a borrow is not generated, the CARRY is set.

  As shown in Figure 5–6, the SUB instruction with a 16-bit shift (shown following this paragraph) is an exception because it only resets the carry bit. This allows the D-unit ALU to generate the appropriate carry when subtracting to the lower or upper half of the accumulator causes a borrow.

  Mnemonic instruction: SUB Smem << #16, ACx, ACy
  Algebraic instruction: ACy = ACx – (Smem << #16)

  Figure 5–6 shows the effect of subtractions on the CARRY bit.

Example 5–7. 64-Bit Subtraction

```
;*********************************************************************
; 64-Bit Subtraction       Pointer assignments:
;
;    X3 X2 X1 X0           AR1 -> X3 (even address)
;  - Y3 Y2 Y1 Y0                  X2
;  --------------                 X1
;    W3 W2 W1 W0                  X0
;                          AR2 -> Y3 (even address)
;                                 Y2
;                                 Y1
;                                 Y0
;                          AR3 -> W3 (even address)
;                                 W2
;                                 W1
;                                 W0
;
;*********************************************************************

   MOV40  dbl(*AR1(#2)), AC0               ; AC0 = X1 X0
   SUB dbl(*AR2(#2)), AC0                  ; AC0 = X1 X0 - Y1 Y0
   MOV AC0, dbl(*AR3(#2))                  ; Store W1 W0.
   MOV40 dbl (*AR1), AC0                   ; AC0 = X3 X2
   SUB uns(*AR2(#1)), BORROW, AC0          ; AC0 = X3 X2 - 00 Y2 - BORROW
   SUB *AR2 << #16, AC0                    ; AC0 = X3 X2 - Y3 Y2 - BORROW
   MOV AC0, dbl(*AR3)                      ; Store W3 W2.
```

Note: The algebraic instructions code example for 64-Bit Subtraction is shown in Example B–28 on page B-31.

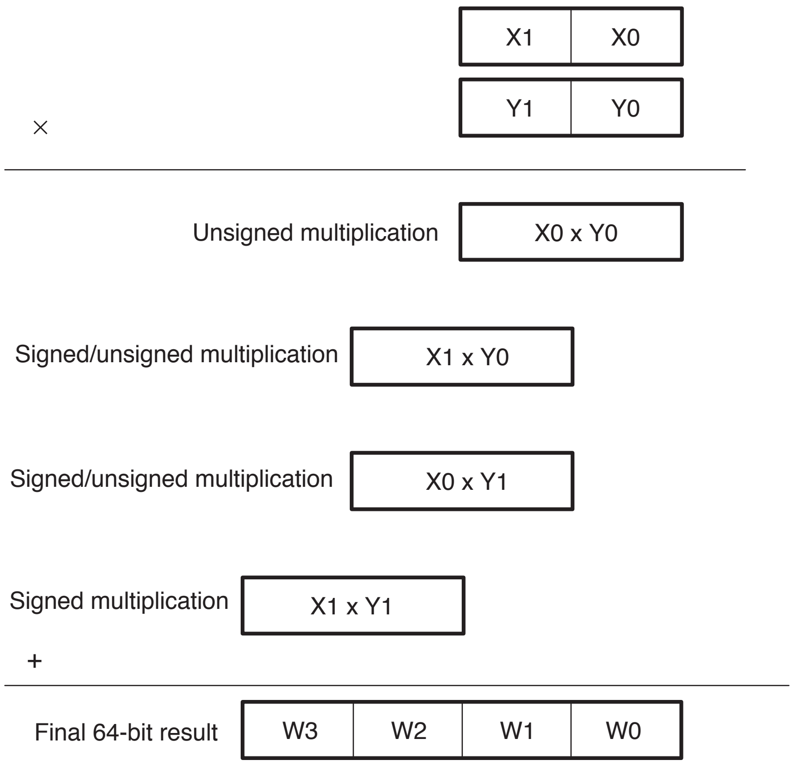
Figure 5–6. Effect on CARRY of Subtraction Operations

```
C  MSB              LSB        C  MSB              LSB
X  0 0 0 0 0 0 0 0 0 0 ACx     X  F F 0 0 0 0 0 0 0 0 ACx
  –                    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 ACx     X  0 0 7 F F F F F F F ACx
  –                    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 ACx     X  F F 8 0 0 0 0 0 0 0 ACx
  –                    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

            ACy = ACx – Smem – BORROW

C  MSB              LSB        C  MSB              LSB
0  0 0 0 0 0 0 0 0 0 0 ACx     0  F F F F F F F F F F ACx
  –                    0         –                    0
0  F F F F F F F F F F ACy     1  F F F F F F F F F E

            ACy = ACx – (Smem << 16)

C  MSB              LSB        C  MSB              LSB
1  F F 8 0 0 0 F F F F ACx     0  F F 8 0 0 0 F F F F ACx
  –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 ACy     0  F F 8 0 0 1 F F F F
                               C keeps a zero value, even
                               when a borrow is generated.
```

5.3 Extended-Precision Multiplication

Extended precision multiplication (operands larger than 16 bit) can be performed using basic C55x instructions. The C55x instruction set provides the user with a very flexible set of 16-bit multiply instructions that accept signed and unsigned operands and with a very efficient set of multiply-and-accumulate instructions that shift the value of the accumulator before adding it to the multiplication result. Figure 5–5 shows how two 32-bit numbers yield a 64-bit product.

Figure 5–7. 32-Bit Multiplication



Example 5–8 shows that a multiplication of two 32-bit integer numbers requires one multiplication, two multiply/accumulate/shift operations, and a multiply/accumulate operation. The product is a 64-bit integer number. Example 5–9 shows a fractional multiplication. The operands are in Q31 format, while the product is in Q31 format.

Example 5–8. 32-Bit Integer Multiplication

```
;*****************************************************************
;   This routine multiplies two 32-bit signed integers, giving a
;   64-bit result. The operands are fetched from data memory and the
;   result is written back to data memory.
;
;   Data Storage:                          Pointer Assignments:
;   X1 X0        32-bit operand            AR0 -> X1
;   Y1 Y0        32-bit operand                   X0
;   W3 W2 W1 W0  64-bit product            AR1 -> Y1
;                                                 Y0
;   Entry Conditions:                      AR2 -> W0
;   SXMD = 1 (sign extension on)                  W1
;   SATD = 0 (no saturation)                      W2
;   FRCT = 0 (fractional mode off)                W3
;
;   RESTRICTION: The delay chain and input array must be
;   long-word aligned.
;****************************************************************

    AMAR *AR0+                             ; AR0 points to X0
    || AMAR *AR1+                          ; AR1 points to Y0
    MPYM uns(*AR0), uns(*AR1), AC0         ; ACO = X0*Y0
    MOV AC0,*AR2+                          ; Save W0
    MACM *AR0+, uns(*AR1-), AC0 >> #16, AC0 ; AC0 = X0*Y0>>16 + X1*Y0
    MACM uns(*AR0-), *AR1, AC0             ; AC0 = X0*Y0>>16 + X1*Y0 + X0*Y1
    MOV AC0, *AR2+                         ; Save W1
    MACM *AR0, *AR1, AC0 >> #16, AC0       ; AC0 = AC0>>16 + X1*Y1
    MOV AC0, *AR2+                         ; Save W2
    MOV HI(AC0), *AR2                      ; Save W3
```

Note: The algebraic instructions code example for 32-Bit Integer Multiplication is shown in Example B–29 on page B-32.

Example 5–9. 32-Bit Fractional Multiplication

```
;***************************************************************************
;   This routine multiplies two Q31 signed integers, resulting in a
;   Q31 result. The operands are fetched from data memory and the
;   result is written back to data memory.
;
;   Data Storage:                          Pointer Assignments:
;   X1 X0     Q31 operand                  AR0 -> X1
;   Y1 Y0     Q31 operand                         X0
;   W1 W0     Q31 product                  AR1 -> Y1
;                                                 Y0
;   Entry Conditions:                      AR2 -> W1 (even address)
;   SXMD = 1 (sign extension on)                  W0
;   SATD = 0 (no saturation)
;   FRCT = 1 (shift result left by 1 bit)
;
;   RESTRICTION: W1 W0 is aligned such that W1 is at an even address.
;***************************************************************************
    AMAR *AR0+                             ; AR0 points to X0
    MPYM uns(*AR0-), *AR1+, AC0            ; AC0 = X0*Y1
    MACM *AR0, uns(*AR1-), AC0             ; AC0 =X0*Y1 + X1*Y0
    MACM *AR0, *AR1, AC0 >> #16, AC0       ; AC0 = AC0>>16 + X1*Y1
    MOV AC0, dbl(*AR2)                     ; Save W1 W0
```

Note: The algebraic instructions code example for 32-Bit Fractional Multiplication is shown in Example B–30 on page B-33.

5.4 Division

Binary division is the inverse of multiplication. Multiplication consists of a series of shift and add operations, while division can be broken into a series of subtract and shift operations. On the C55x DSP you can implement this kind of division by repeating a form of the conditional subtract (SUBC) instruction.

In a fixed-point processor, the range of the numbers we can use is limited by the number of bits and the convention we use to represent these numbers. For example, with a 16-bit unsigned representation, it is not possible to represent a number larger than $2^{16} - 1$ (that is, 65 535). There can be problems with division operations that require computing the inverse of a very small number. Some digital signal processing algorithms may require integer or fractional division operations that support a large range of numbers. This kind of division can be implemented with the conditional subtract (SUBC) instruction.

The difference between integers and fractional numbers is so great in a fixed-point architecture that it requires different algorithms to perform the division operation. Section 5.4.1 shows how to implement signed and unsigned integer division. Section 5.4.2 (page 5-23) describes fractional division.

5.4.1 Integer Division

To prepare for a SUBC integer division, place a 16-bit positive dividend in an accumulator. Place a 16-bit positive divisor in memory. When you write the SUBC instruction, make sure that the result will be in the same accumulator that supplies the dividend; this creates a cumulative result in the accumulator when the SUBC instruction is repeated. Repeating the SUBC instruction 16 times produces a 16-bit quotient in the low part of the accumulator (bits 15–0) and a remainder in the high part of the accumulator (bits 31–16). During each execution of the conditional subtract instruction:

1) The 16-bit divisor is shifted left by 15 bits and is subtracted from the value in the accumulator.

2) If the result of the subtraction is greater than or equal to 0, the result is shifted left by 1 bit, added to 1, and stored in the accumulator. If the result of the subtraction is less than 0, the result is discarded and the value in the accumulator is shifted left by 1 bit.

The following examples show the implementation of the signed/unsigned integer division using the SUBC instruction.

5.4.1.1 Examples of Unsigned Integer Division

Example 5–10 shows how to use the SUBC instruction to implement unsigned division with a 16-bit dividend and a 16-bit divisor.

Example 5–10. Unsigned, 16-Bit By 16-Bit Integer Division

```
;***************************************************************************
; Pointer assignments:                    ___________
;      AR0 -> Dividend        Divisor ) Dividend
;      AR1 -> Divisor
;      AR2 -> Quotient
;      AR3 -> Remainder
;
; Algorithm notes:
;      - Unsigned division, 16-bit dividend, 16-bit divisor
;      - Sign extension turned off. Dividend & divisor are positive numbers.
;      - After division, quotient in AC0(15-0), remainder in AC0(31-16)
;***************************************************************************

   BCLR SXMD                  ; Clear SXMD (sign extension off)
   MOV *AR0, AC0              ; Put Dividend into AC0
   RPT #(16 - 1)              ; Execute subc 16 times
      SUBC *AR1, AC0, AC0     ; AR1 points to Divisor
   MOV AC0, *AR2              ; Store Quotient
   MOV HI(AC0), *AR3          ; Store Remainder
```

Note: The algebraic instructions code example for Unsigned, 16-Bit Integer Division is shown in Example B–31 on page B-33.

Example 5–11 shows how to implement unsigned division with a 32-bit dividend and a 16-bit divisor. The code uses two phases of 16-bit by 16-bit integer division. The first phase takes as inputs the high 16 bits of the 32-bit dividend and the 16-bit divisor. The result in the low half of the accumulator is the high 16 bits of the quotient. The result in the high half of the accumulator is shifted left by 16 bits and added to the lower 16 bits of the dividend. This sum and the 16-bit divisor are the inputs to the second phase of the division. The lower 16 bits of the resulting quotient is the final quotient and the resulting remainder is the final remainder.

Example 5–11. Unsigned, 32-Bit By 16-Bit Integer Division

```
;***************************************************************************
; Pointer assignments:                                   ___________
;      AR0 -> Dividend high half         Divisor ) Dividend
;             Dividend low half
;             ...
;      AR1 -> Divisor
;             ...
;      AR2 -> Quotient high half
;             Quotient low half
;             ...
;      AR3 -> Remainder
;
; Algorithm notes:
;      - Unsigned division, 32-bit dividend, 16-bit divisor
;      - Sign extension turned off. Dividend & divisor are positive numbers.
;      - Before 1st division: Put high half of dividend in AC0
;      - After 1st division:  High half of quotient in AC0(15-0)
;      - Before 2nd division: Put low part of dividend in AC0
;      - After 2nd division:  Low half of quotient in AC0(15-0) and
;                             Remainder in AC0(31-16)
;***************************************************************************

    BCLR SXMD                     ; Clear SXMD (sign extension off)
    MOV *AR0+, AC0                ; Put high half of Dividend in AC0
    ||  RPT #(15 - 1)             ; Execute subc 15 times
       SUBC *AR1, AC0, AC0        ; AR1 points to Divisor
    SUBC *AR1, AC0, AC0           ; Execute subc final time
    || MOV #8, AR4                ; Load AR4 with AC0_L memory address
    MOV AC0, *AR2+                ; Store high half of Quotient
    MOV *AR0+, *AR4               ; Put low half of Dividend in AC0_L
    RPT #(16 - 1)                 ; Execute subc 16 times
       SUBC *AR1, AC0, AC0
    MOV AC0, *AR2+                ; Store low half of Quotient
    MOV HI(AC0), *AR3)            ; Store Remainder
    BSET SXMD                     ; Set SXMD (sign extension on)
```

Note: The algebraic instructions code example for Unsigned, 32-Bit by 16-Bit Integer Division is shown in Example B–32 on page B-34.

5.4.1.2 Examples of Signed Integer Division

Some applications might require doing division with signed numbers instead of unsigned numbers. The conditional subtract instruction works only with positive integers. The signed integer division algorithm computes the quotient as follows:

1) The sign of the quotient is determined and preserved in AC0.

2) The quotient of the absolute values of the dividend and the divisor is determined using repeated conditional subtract instructions.

3) The negative of the result is computed if required, according to the sign of AC0.

Example 5–12 shows the implementation of division with a signed 16-bit dividend and a 16-bit signed divisor, and Example 5–13 extends this algorithm to handle a 32-bit dividend.

Example 5–12. Signed, 16-Bit By 16-Bit Integer Division

```
;***************************************************************************
; Pointer assignments:                          ___________
;      AR0 -> Dividend        Divisor ) Dividend
;      AR1 -> Divisor
;      AR2 -> Quotient
;      AR3 -> Remainder
;
; Algorithm notes:
;      - Signed division, 16-bit dividend, 16-bit divisor
;      - Sign extension turned on. Dividend and divisor can be negative.
;      - Expected quotient sign saved in AC0 before division
;      - After division, quotient in AC1(15-0), remainder in AC1(31-16)
;***************************************************************************

      BSET SXMD                    ; Set SXMD (sign extension on)
      MPYM *AR0, *AR1, AC0         ; Sign of (Dividend x Divisor) should be
                                   ;   sign of Quotient
      MOV *AR1, AC1                ; Put Divisor in AC1
      ABS AC1, AC1                 ; Find absolute value, |Divisor|
      MOV AC1, *AR2                ; Store |Divisor| temporarily
      MOV *AR0, AC1                ; Put Dividend in AC1
      ABS AC1, AC1                 ; Find absolute value, |Dividend|
      RPT #(16 - 1)                ; Execute subc 16 times
         SUBC *AR2, AC1, AC1       ; AR2 -> |Divisor|
      MOV HI(AC1), *AR3            ; Save Remainder
      MOV AC1, *AR2                ; Save Quotient
      SFTS AC1, #16                ; Shift quotient: Put MSB in sign position
      NEG AC1, AC1                 ; Negate quotient
      XCCPART label, AC0 < #0      ; If sign of Quotient should be negative,
      MOV HI(AC1), *AR2            ;   replace Quotient with negative version
label:
```

Note: The algebraic instructions code example for Signed, 16-Bit by 16-Bit Integer Division is shown in Example B–33 on page B-35.

Example 5–13. Signed, 32-Bit By 16-Bit Integer Division

```
;***************************************************************************
;   Pointer assignments:    (Dividend and Quotient are long-word aligned)
;       AR0 -> Dividend high half (NumH) (even address)
;              Dividend low half (NumL)
;       AR1 -> Divisor (Den)
;       AR2 -> Quotient high half (QuotH) (even address)
;              Quotient low half (QuotL)
;       AR3 -> Remainder (Rem)
;
;   Algorithm notes:
;       - Signed division, 32-bit dividend, 16-bit divisor
;       - Sign extension turned on. Dividend and divisor can be negative.
;       - Expected quotient sign saved in AC0 before division
;       - Before 1st division:  Put high half of dividend in AC1
;       - After 1st division:   High half of quotient in AC1(15-0)
;       - Before 2nd division:  Put low part of dividend in AC1
;       - After 2nd division:   Low half of quotient in AC1(15-0) and
;                               Remainder in AC1(31-16)
;***************************************************************************

   BSET SXMD                  ; Set SXMD (sign extension on)
   MPYM  *AR0, *AR1, AC0      ; Sign( NumH x Den ) is sign of actual result
   MOV *AR1, AC1              ; AC1 = Den
   ABS AC1, AC1               ; AC1 = abs(Den)
   MOV AC1, *AR3              ; Rem = abs(Den) temporarily
   MOV40 dbl(*AR0), AC1       ; AC1 = NumH NumL
   ABS AC1, AC1               ; AC1 = abs(Num)
   MOV AC1, dbl(*AR2)         ; QuotH = abs(NumH) temporarily
                              ; QuotL = abs(NumL) temporarily

   MOV *AR2, AC1              ; AC1 = QuotH
   RPT #(15 - 1)              ; Execute subc 15 times
      SUBC *AR3, AC1, AC1
   SUBC *AR3, AC1, AC1        ; Execute subc final time
   || MOV #11, AR4            ; Load AR4 with AC1_L memory address
   MOV AC1, *AR2+             ; Save QuotH
   MOV *AR2, *AR4             ; AC1_L = QuotH
   RPT #(16 - 1)              ; Execute subc 16 times
      SUBC *AR3, AC1, AC1
   MOV AC1, *AR2-             ; Save QuotL
   MOV HI(AC1), *AR3          ; Save Rem

   BCC skip, AC0 >= #0        ; If actual result should be positive, goto skip.
   MOV40 dbl(*AR2), AC1       ; Otherwise, negate Quotient.
   NEG AC1, AC1
   MOV AC1, dbl(*AR2)

skip:
   RET
```

Note: The algebraic instructions code example for Signed, 32-Bit by 16-Bit Integer Division is shown in Example B–34 on page B-36.

5.4.2 Fractional Division

The algorithms that implement fractional division compute first an approximation of the inverse of the divisor (denominator) using different techniques such as Taylor Series expansion, line of best fit, and successive approximation. The result is then multiplied by the dividend (numerator). The C55x DSP function library (see Chapter 8) implements this function under the name ldiv16.

To calculate the value of Y_m this algorithm uses the successive approximation method. The approximations are performed using the following equation:

$$Y_{m+1} = 2\,Y_m - Y_m^{\,2} X_{norm}$$

If we start with an initial estimate of Y_m, then the equation will converge to a solution very rapidly (typically in three iterations for 16-bit resolution). The initial estimate can either be obtained from a look-up table, from choosing a midpoint, or simply from linear interpolation. The ldiv16 algorithm uses linear interpolation. This is accomplished by taking the complement of the least significant bits of the X_{norm} value.

5.5 Methods of Handling Overflows

An overflow occurs when the result of an arithmetical operation is larger than the largest number that can be represented in the register that must hold the result. Due to the 16-bit format, fixed-point DSPs provide a limited dynamic range. You must manage the dynamic range of your application to avoid possible overflows. The overflow depends on the nature of the input signal and of the algorithm in question.

5.5.1 Hardware Features for Overflow Handling

The C55x DSP offers several hardware features for overflow handling:

- Guard bits:

  Each of the C55x accumulators (AC0, AC1, AC2, and AC3) has eight guard bits (bits 39–32), which allow up to 256 consecutive multiply-and-accumulate operations before an accumulator overflow.

- Overflow flags:

  Each C55x accumulator has an associated overflow flag (see the following table). When an operation on an accumulator results in an overflow, the corresponding overflow flag is set.

- Saturation mode bits:

  The DSP has two saturation mode bits: SATD for operations in the D unit of the CPU and SATA for operations in the A unit of the CPU. When the SATD bit is set and an overflow occurs in the D unit, the CPU saturates the result. Regardless of the value of SATD, the appropriate accumulator overflow flag is set. Although no flags track overflows in the A unit, overflowing results in the A unit are saturated when the SATA bit is set.

  Saturation replaces the overflowing result with the nearest range boundary. Consider a 16-bit register which has range boundaries of 8000h (largest negative number) and 7FFFh (largest positive number). If an operation generates a result greater than 7FFFh, saturation can replace the result with 7FFFh. If a result is less than 8000h, saturation can replace the result with 8000h.

5.5.1.1 Overview of Overflow Handling Techniques

There are a number of general methodologies to handle overflows. Among the methodologies are saturation, input scaling, fixed scaling, and dynamic scaling. We will give an overview of these methodologies and will see some examples illustrating their application.

- Saturation:

  One possible way to handle overflows is to use the hardware saturation modes mentioned in section 5.5.1. However, saturation has the effect of clipping the output signal, potentially causing data distortion and non-linear behavior in the system.

- Input scaling:

  You can analyze the system that you want to implement and scale the input signal, assuming worst conditions, to avoid overflow. However, this approach can greatly reduce the precision of the output.

- Fixed scaling:

  You can scale the intermediate results, assuming worst conditions. This method prevents overflow but also increases the system's signal-to-noise ratio.

- Dynamic scaling:

  The intermediate results can be scaled only when needed. You can accomplish this by monitoring the range of the intermediate results. This method prevents overflow but increases the computational requirements.

The next sections demonstrate these methodologies applied to FIR (finite impulse response) filters, IIR (infinite impulse response) filters and FFTs (fast Fourier transforms).

5.5.1.2 Scaling Methods for FIR Filters

The best way to handle overflow problems in FIR (finite impulse response) filters is to design the filters with a gain less than 1 to avoid having to scale the input data. This method, combined with the guard bits available in each of the accumulators, provides a robust way to handle overflows in the filters.

Fixed scaling and input scaling are not used due to their negative impact on signal resolution (basically one bit per multiply-and-accumulate operation). Dynamic scaling can be used for an FIR filter if the resulting increase in cycles is not a concern. Saturation is also a common option for certain types of audio signals.

5.5.1.3 Scaling Methods for IIR Filters

Fixed-point realization of an IIR (infinite impulse response) filter in cascaded second-order phases is recommended to minimize the frequency response sensitivity of high-order filters. In addition to round-off error due to filter coefficient quantization, overflow avoidance is critical due to the recursive nature of the IIR filter.

Overflow between phases can be avoided by maintaining the intermediate value in the processor accumulator. However, overflow can happen at the internal filter state (delay buffer) inside each phase. To prevent overflow at phase k, the filter unit-pulse response f(n) must be scaled (feed forward path) by a gain factor G_k given by

$$Option 1 : G_k = \sum_n abs(f(n))$$

or

$$Option\ 2 : G_k = \sum_n \left(abs(f(n))^2\right)^{1/2}$$

Option 1 prevents overflows, but at the expense of precision. Option 2 allows occasional overflows but offers an improved precision. In general, these techniques work well if the input signal does not have a large dynamic range.

Another method to handle overflow in IIR filters is to use dynamic scaling. In this approach, the internal filter states are scaled down by half only if an overflow is detected at each phase. The result is a higher precision but at the expense of increased MIPS.

5.5.1.4 Scaling Methods for FFTs

In FFT (Fast Fourier Transform) operations, data will grow an average of one bit on the output of each butterfly. Input scaling will require shifting the data input by *log n* (n = size of FFT) that will cause a *6(log n)* dB loss even before computing the FFT. In fixed scaling, the output of the butterfly will be scaled by 2 at each phase. This is probably the most common scaling approach for FFTs because it is simple and has a better sound-to-noise ratio (SNR). However, for larger FFTs this scaling may cause information loss.

Another option is to implement a dynamic scaling approach in which scaling by 2 at each phase occurs only when bit growth occurs. In this case, an exponent is assigned to the entire phase block (block floating-point method). When scaling by 2 happens, the exponent is incremented by 1. At the end of the FFT, the data are scaled up by the resulting exponent. Dynamic scaling provides the best SNR but increases the FFT cycle count because you have to detect bit growth and update the exponent accordingly.

Chapter 6

Bit-Reversed Addressing

This chapter introduces the concept and the syntax of bit-reverse addressing. It then explains how bit-reverse addressing can help to speed up a Fast Fourier Transform (FFT) algorithm. To find code that performs complex and real FFTs (forward and reverse) and bit-reversing of FFT vectors, see Chapter 8, *TI C55x DSPLIB*.

Topic **Page**

6.1 Introduction to Bit-Reverse Addressing

Bit-reverse addressing is a special type of indirect addressing. It uses one of the auxiliary registers (AR0–AR7) as a base pointer of an array and uses temporary register 0 (T0) as an index register. When you add T0 to the auxiliary register using bit-reverse addressing, the address is generated in a bit-reversed fashion, with the carry propagating from left to right instead of from right to left.

Table 6–1 shows the syntaxes for each of the two bit-reversed addressing modes supported by the TMS320C55x (C55x) DSP.

Table 6–1. Syntaxes for Bit-Reverse Addressing Modes

Operand Syntax	Function	Description

*(ARx – T0B)	address = ARx
	ARx = (ARx – T0)
		After access, T0 is subtracted from ARx with reverse carry (rc) propagation.

*(ARx + T0B)	address = ARx
	ARx = (ARx + T0)
		After access, T0 is added to ARx with reverse carry (rc) propagation.

Assume that the auxiliary registers are 8 bits long, that AR2 represents the base address of the data in memory (0110 0000b), and that T0 contains the value 0000 1000b (decimal 8). Example 6–1 shows a sequence of modifications of AR2 and the resulting values of AR2.

Table 6–2 shows the relationship between a standard bit pattern that is repeatedly incremented by 1 and a bit-reversed pattern that is repeatedly incremented by 1000b with reverse carry propagation. Compare the bit-reversed pattern to the 4 LSBs of AR2 in Example 6–1.

Example 6–1. Sequence of Auxiliary Registers Modifications in Bit-Reversed Addressing

```
*(AR2+T0B)    ;AR2   =   0110 0000     (0th value)
*(AR2+T0B)    ,AR2   =   0110 1000     (1st value)
*(AR2+T0B)    ;AR2   =   0110 0100     (2nd value)
*(AR2+T0B)    ;AR2   =   0110 1100     (3rd value)
*(AR2+T0B)    ;AR2   =   0110 0010     (4th value)
*(AR2+T0B)    ;AR2   =   0110 1010     (5th value)
*(AR2+T0B)    ;AR2   =   0110 0110     (6th value)
*(AR2+T0B)    ;AR2   =   0110 1110     (7th value)
```

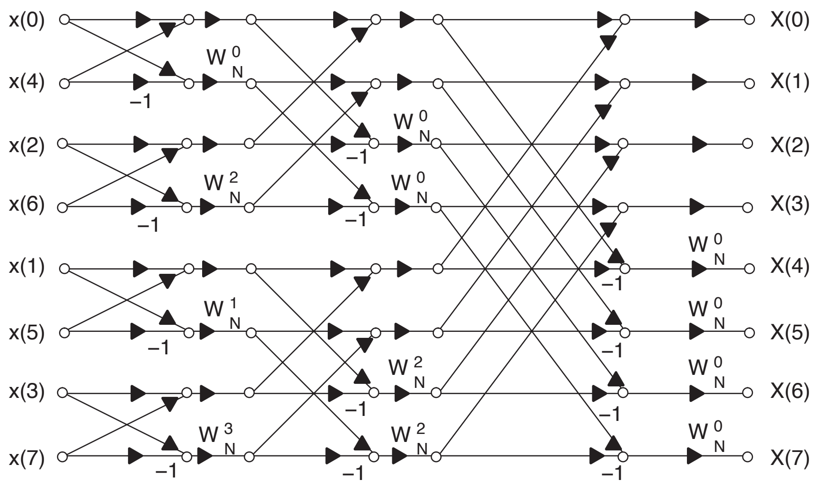
Table 6–2. Bit-Reversed Addresses

Step	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Step
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

6.2 Using Bit-Reverse Addressing In FFT Algorithms

Bit-reversed addressing enhances execution speed for Fast-Fourier Transform (FFT) algorithms. Typical FFT algorithms either take an in-order vector input and produce a bit-reversed vector output or take a bit-reversed vector input and produce an in-order vector output. In either case, bit-reverse addressing can be used to resequence the vectors. Figure 6–1 shows a flow graph of an 8-point decimation-in-frequency FFT algorithm with a bit-reversed input and an in-order output.

Figure 6–1. FFT Flow Graph Showing Bit-Reversed Input and In-Order Output



Consider a complex FFT of size N (that is, an FFT with an input vector that contains N complex numbers). You can bit-reverse either the input or the output vectors by executing the following steps:

1) Write 0 to the ARMS bit of status register 2 to select the DSP mode for AR indirect addressing. (Bit-reverse addressing is not available in the control mode of AR indirect addressing.) Then use the .arms_off directive to notify the assembler of this selection.

2) Use Table 6–3 to determine how the base pointer of the input array must be aligned to match the given vector format. Then load an auxiliary register with the proper base address.

3) Consult Table 6–3 to properly load the index register, T0.

4) Ensure that the entire array fits within a 64K boundary (the largest possible array addressable by the 16-bit auxiliary register).

As an example of how to use Table 6–3, suppose you need to bit-reverse a vector with N = 64 complex elements in the Re-Im-Re-Im format. The (n + 1) least significant bits (LSBs) of the base address must be 0s, where

$(n + 1) = (\log_2 N + 1) = (\log_2 64 + 1) = (6 + 1) = 7$ LSBs

Therefore, AR0 must be loaded with a base address of this form (Xs are don't cares):

AR0 = XXXX XXXX X000 0000b

The index loaded into T0 is equal to the number of elements:

$T0 = 2^n = 2^6 = 64$

Table 6–3. Typical Bit-Reverse Initialization Requirements

Vector Format in Memory	T0 Initialization Value	Alignment of Vector Base Address
Re-Im-Re-Im. The real and imaginary parts of each complex element are stored at consecutive memory locations. For example: Real Imaginary ... Real Imaginary	2^n where $n = \log_2 N$	(n+1) LSBs must be 0s, where $n = \log_2 N$
Re-Re...Im-Im. The real and imaginary data are stored in separate arrays. For example: Real Real ... Imaginary Imaginary	$2^{(n-1)}$ where $n = \log_2 N$	n LSBs must be 0s, where $n = \log_2 N$

6.3 In-Place Versus Off-Place Bit-Reversing

You can have a bit-reversed array write over the original array (in-place bit-reversing) or you can place the bit-reversed array in a separate place in memory (off-place bit-reversing). Example 6–2 shows assembly language code for an off-place bit-reverse operation. An input array of N complex elements pointed to by AR0 is bit-reversed into an output array pointed to by AR1. The vector format for input and output vectors is assumed to be Re-Im-Re-Im. Each element (Re-Im) of the input array is loaded into AC0. Then the element is transferred to the output array using bit-reverse addressing. Each time the index in T0 is added to the address in AR1, the addition is done with carries propagating from left to right, instead of from right to left.

Although it requires twice the memory, off-place bit-reversing is faster than in-place bit-reversing. Off-place bit-reversing requires 2 cycles per complex data point, while in-place bit-reversing requires approximately 4 cycles per complex data point.

Example 6–2. Off-Place Bit Reversing of a Vector Array (in Assembly)

```
;...
 BCLR ARMS                    ; reset ARMS bit to allow bit-reverse addressing
 .arms_off                    ; notify the assembler of ARMS bit = 0
;...
off_place:
 RPTBLOCAL LOOP
       MOV dbl(*AR0+), AC0        ; AR0 points to input array
LOOP:  MOV AC0, dbl(*(AR1+T0B))   ; AR1 points to output array
                              ; T0 = NX = number of complex elements in
                              ; array pointed to by AR0
```

Note: The algebraic instructions code example for Off-Place Bit Reversing of a Vector Array (in Assembly) is shown in Example B–35 on page B-37.

6.4 Using the C55x DSPLIB for FFTs and Bit-Reversing

The C55x DSP function library (DSPLIB) offers C-callable DSP assembly-optimized routines. Among these are a bit-reversing routine (cbrev()) and complex and real FFT routines.

Example 6–3 shows how you can invoke the cbrev() DSPLIB function from C to do in-place bit-reversing. The function bit-reverses the position of in-order elements in a complex vector x and then computes a complex FFT of the bit-reversed vector. The function uses in-place bit-reversing. See Chapter 8 for an introduction to the C55x DSPLIB.

Example 6–3. Using DSPLIB cbrev() Routine to Bit Reverse a Vector Array (in C)

```
#define NX 64
short x[2*NX]        ;
short scale = 1      ;

void main(void)
{
;...
cbrev(x,x,NX)        // in-place bit-reversing on input data (Re-Im format)
cfft(x,NX,scale)     // 64-point complex FFT on bit-reversed input data with
                     // scaling by 2 at each phase enabled
;...
}
```

Note: This example shows portions of the file cfft_t.c in the TI C55x DSPLIB (introduced in Chapter 8).

Chapter 7

Application-Specific Instructions

This chapter presents examples of efficient implementations of some common signal processing and telecommunications functions. These examples illustrate the use of some application-specific instructions on the TMS320C55x (C55x) DSP. (Most of the examples in this chapter use instructions from the mnemonic instruction set, corresponding algebraic instruction set examples are shown in Appendix A.)

Topic **Page**

7.1 Symmetric and Asymmetric FIR Filtering (FIRS, FIRSN)

FIR (finite impulse response) filters are often used in telecommunications applications because they are unconditionally stable and they may be designed to preserve important phase information in the processed signal. A *linear phase* FIR provides a phase shift that varies in proportion to the input frequency and requires that the impulse response be **symmetric**: h(n) = h(N–n).

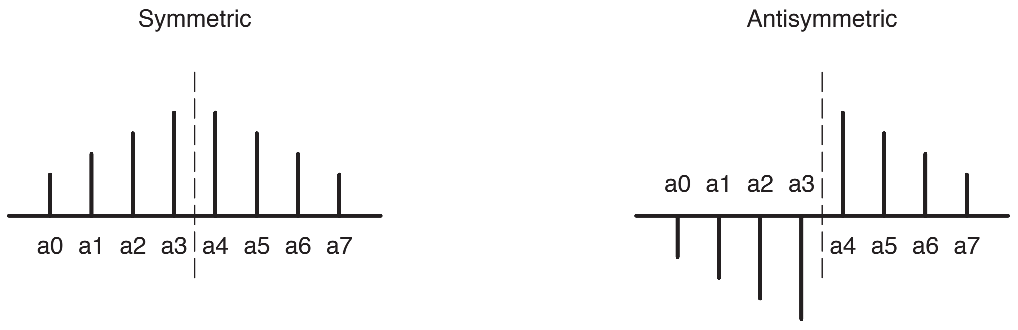
Another class of FIR filter is the **antisymmetric** FIR: h(n) = –h(N–n). A common example is the Hilbert transformer, which shifts positive frequencies by +90 degrees and negative frequencies by –90 degrees. Hilbert transformers may be used in applications, such as modems, in which it is desired to cancel lower sidebands of modulated signals.

Figure 7–1 gives examples of symmetric and antisymmetric filters, each with eight coefficients (a0 through a7). Both symmetric and antisymmetric filters may be of even or odd length. However, even-length symmetric filters lend themselves to computational shortcuts which will be described in this section. It is sometimes possible to reformulate an odd-length filter as a filter with one more tap, to take advantage of these constructs.

Because (anti)symmetric filters have only N/2 distinct coefficients, they may be folded and performed with N/2 additions (subtractions) and N/2 multiply-and-accumulate operations. Folding means that pairs of elements in the delay buffer which correspond to the same coefficient are pre-added(subtracted) prior to multiplying and accumulating.

The C55x DSP offers two different ways to implement symmetric and asymmetric filters. This section shows how to implement these filters using specific instructions, FIRS and FIRSN. To see how to implement symmetric and asymmetric filters using the dual-MAC hardware, see section 4.1.1, *Implicit Algorithm Symmetry*, which begins on page 4-4. The firs/firsn implementation and the dual-MAC implementation are equivalent from a throughput standpoint.

Figure 7–1. Symmetric and Antisymmetric FIR Filters



Symmetric FIR Filter a[n]:

$a[n] = a[N-n]$ where $0 < n < \frac{N}{2}$

Antisymmetric FIR Filter a[n]:

$a[n] = -a[N-n]$ where $0 < n < \frac{N}{2}$

Symmetric FIR filter output:

$$Y(n) = a[0](x[n-7]+x[0]) + a[1](x[n-6]+x[1]) + a[2](x[n-5]+x[2]) + a[3](x[n-4]+x[3])$$

Antisymmetric FIR filter output:

$$Y(n) = a[0](x[n-7]-x[0]) + a[1](x[n-6]-x[1]) + a[1](x[n-5]-x[2]) + a[3](x[n-4]-x[3])$$

Definitions:

a = Filter coefficient	Y = Filter output
n = Sample index	x = Filter input data value
N = Number of filter taps	

7.1.1 Symmetric FIR Filtering With the *firs* Instruction

The C55x instruction for symmetric FIR filtering is:

firs(Xmem,Ymem,Cmem,ACx,ACy)

This instruction performs two parallel operations: a multiply-and-accumulate (MAC) operation, and an addition. The firs() instruction performs the following parallel operations:

ACy = ACy + (ACx * Cmem),
ACx = (Xmem << #16) + (Ymem << #16)

The first operation performs a multiplication and an accumulation in a MAC unit of the CPU. The input operands of the multiplier are the content of ACx(32–16) and a data memory operand, which is addressed using the coefficient addressing mode and is sign extended to 17 bits. Table 7–1 explains the operands necessary for the operation.

Table 7−1. Operands to the firs *or* firsn *Instruction*

Operand(s)	Description
Xmem and Ymem	One of these operands points to the newest value in the delay buffer. The other points to the oldest value in the delay buffer.
Cmem	This operand points to the filter coefficient.
ACx	ACx is one of the four accumulators (AC0−AC3). It holds the sum of the two delayed input values referenced by Xmem and Ymem.
ACy	ACy is one of the four accumulators (AC0−AC3) but is not the same accumulator as ACx. ACy holds the output of each filter tap. After all the filter taps have been performed, ACy holds the final result.

7.1.2 Antisymmetric FIR Filtering With the *firsn* Instruction

The antisymmetric FIR is the same as the symmetric FIR except that the pre-addition of sample pairs is replaced with a pre-subtraction. The C55x instruction for antisymmetric FIR filtering is:

```
firsn(Xmem,Ymem,Cmem,ACx,ACy)
```

This instruction performs two parallel operations: a multiply-and-accumulate (MAC) operation, and a subtraction. The firsn() instruction performs the following parallel operations:

```
ACy = ACy + (ACx * Cmem),
ACx = (Xmem << #16) − (Ymem << #16)
```

The first operation performs a multiplication and an accumulation in a MAC unit of the CPU. The input operands of the multiplier are the content of ACx(32−16) and a data memory operand, which is addressed using the coefficient addressing mode and is sign extended to 17 bits. Table 7−1 (page 7-4) explains the operands necessary for the operation.

7.1.3 Implementation of a Symmetric FIR Filter on the TMS320C55x DSP

The C55x DSPLIB features an efficient implementation of the Symmetric FIR on the C55x device. Example 7−1 presents the kernel of that implementation to illustrate the usage of the *firs* instruction.

Example 7–1. Symmetric FIR Filter

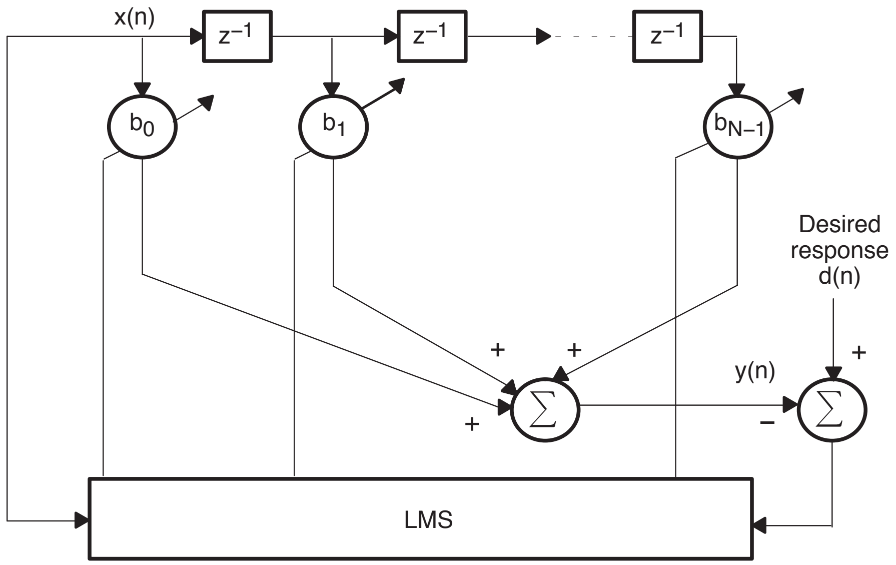
```
;
; Start of outer loop
;----------------------------------------------------------------
      localrepeat {           ; Start the outer loop

; Get next input value

      *db_ptr1 = *x_ptr+      ; x_ptr: pointer to input data buffer
                              ; db_ptr1: pointer to newest input value

; Clear AC0 and pre-load AC1 with the sum of the 1st and last inputs
      ||AC0 = #0;

; 1st and last inputs
      AC1 = (*db_ptr1+ << #16) + (*db_ptr2- << #16)
; Inner loop
      ||repeat(inner_cnt)
      firs(*db_ptr1+, *db_ptr2-, *h_ptr+, AC1, AC0)

; 2nd to last iteration has different pointer adjustment
      firs(*(db_ptr1-T0), *(db_ptr2+T1), coef(*h_ptr+), AC1, AC0)

; Last iteration is a MAC with rounding
      AC0 = rnd(AC0 + (*h_ptr+ * AC1))

; Store result to memory
      *r_ptr+ = HI(AC0)       ;store Q15 value to memory

      }                       ;end of outer loop
```

Note: This example shows portions of the file firs.asm in the TI C55x DSPLIB (introduced in Chapter 8).

7.2 Adaptive Filtering (LMS)

Some applications for adaptive FIR (finite impulse response) and IIR (infinite impulse response) filtering include echo and acoustic noise cancellation. In these applications, an adaptive filter tracks changing conditions in the environment. Although in theory, both FIR and IIR structures can be used as adaptive filters, stability problems and the local optimum points of IIR filters makes them less attractive for this use. Therefore, FIR filters are typically used for practical adaptive filter applications. The least mean square (LMS), local block-repeat, and parallel instructions on the C55x DSP can be used to efficiently implement adaptive filters. The block diagram of an adaptive FIR filter is shown in Figure 7–2.

Figure 7–2. Adaptive FIR Filter Implemented With the Least-Mean-Squares (LMS) Algorithm



Two common algorithms employed for least mean squares adaptation are the non-delayed LMS and the delayed LMS algorithm. When compared to non-delayed LMS, the more widely used delayed LMS algorithm has the advantage of greater computational efficiency at the expense of slightly relaxed convergence properties. Therefore, section 7.2.1 describes only the delayed LMS algorithm.

7.2.1 Delayed LMS Algorithm

In the delayed LMS, the convolution is performed to compute the output of the adaptive filter:

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

where

y = Filter output
n = Sample index
k = Delay index
N = Number of filter taps
b_k = Adaptive coefficient
x = Filter input data value

The value of the error is computed and stored to be used in the next invocation:

$$e(n) = d(n) - y(n)$$

where

e = Error
d = Desired response
y = Actual response (filter output)

The coefficients are updated based on an error value computed in the previous invocation of the algorithm (β is the conversion constant):

$$b_k(n+1) = b_k(n) + 2\beta e(n-1)x(n-k-1)$$

The delayed LMS algorithm can be implemented with the LMS instruction—lms(Xmem, Ymem, ACx, ACy)—which performs a multiply-and-accumulate (MAC) operation and, in parallel, an addition with rounding:

```
ACy = ACy + (Xmem * Ymem),
ACx = rnd(ACx + (Xmem << #16)
```

The input operands of the multiplier are the content of data memory operand Xmem, sign extended to 17 bits, and the content of data memory operand Ymem, sign extended to 17 bits. One possible implementation would assign the following roles to the operands of the LMS instruction:

Operand(s)	**Description**
Xmem	This operand points to the coefficient array.
Ymem	This operand points to the data array.
ACx	ACx is one of the four accumulators (AC0−AC3). ACx is used to update the coefficients.
ACy	ACy is one of the four accumulators (AC0−AC3) but is not the same accumulator as ACx. ACy holds the output of the FIR filter.

An efficient implementation of the delayed LMS algorithm is available in the C55x DSP function library (see Chapter 8). NO TAG shows the kernel of this implementation.

Example 7–2. Delayed LMS Implementation of an Adaptive Filter

```
;   ar_data: index in the delay buffer
;   ar_input: pointer to input vector
;   ar_coef: pointer to coefficient vector

StartSample:

; Clear AC0 for initial error term
   MOV   #0, AC1
   ||RPTBLOCAL OuterLoop-1
   MOV   *ar_input+, *ar_data+    ;copy input -> state(0)

; Place error term in T3
   MOV   HI(AC1), T3

; Place first update term in AC0
;...while clearing FIR value
   MPYM  *ar_data+, T3, AC0
   ||MOV #0, AC1

;AC0 = update coef
;AC1 = start of FIR output
   LMS   *ar_coef, *ar_data, AC0, AC1
   ||RPTLOCAL InnerLoop-1
   MOV   HI(AC0), *ar_coef+
   ||MPYM      *ar_data+, T3, AC0

;AC0 = update coef
;AC1 = update of FIR output
     LMS   *ar_coef, *ar_data, AC0, AC1
InnerLoop:

; Store Calculated Output
   MOV   HI(AC0), *ar_coef+
   ||MOV rnd(HI(AC1)), *ar_output+

; AC2 is error amount
; Point to oldest data sample
   SUB   AC1, *ar_des+ << #16, AC2
   ||AMAR     *ar_data+

; Place updated mu_error term in AC1
   AMPYMR T_step, AC2, AC1
OuterLoop:
```

Note: The algebraic instructions code example for Delayed LMS Implementation of an Adaptive Filter is shown in Example B–36 on page B-38.

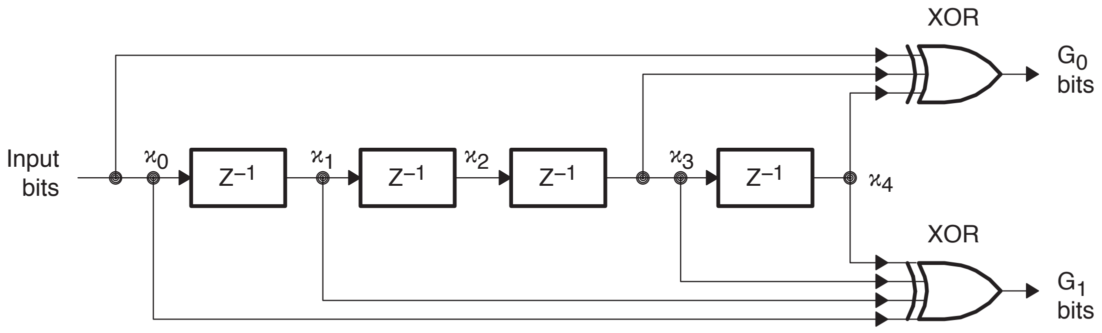
7.3 Convolutional Encoding (BFXPA, BFXTR)

The goal in every telecommunication system is to achieve maximum data transfer, using a minimum bandwidth, while maintaining an acceptable quality of transmission. Convolutional codes are a forward error control (FEC) technique in which extra binary digits are added to the original information binary digits prior to transmission to create a code structure which is resistant to errors that may occur within the channel. A decoder at the receiver exploits the code structure to correct any errors that may have occurred. The redundant bits are formed by XORing the current bit with time-delayed bits within the past K input sample history. This is effectively a 1-bit convolution sum; hence the term convolutional encoder. The coefficients of the convolution sum are described using polynomial notation. A convolutional code is defined by the following parameters:

n = Number of function generators
G0, G1, ... , Gn = Polynomials that define the convolutions of bit streams
K = Constraint length (number of delays plus 1)

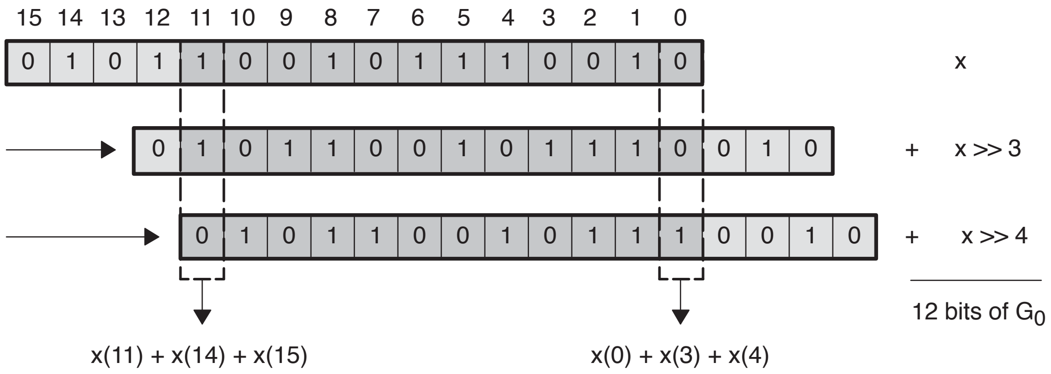
The rate of the convolutional encoder is defined as R = 1/n. Figure 7–3 gives an example of a convolutional encoder with K=5 and R = 1/2.

Figure 7–3. Example of a Convolutional Encoder



The C55x DSP creates the output streams (G0 and G1) by XORing the shifted input stream (see Figure 7–4).

Figure 7–4. Generation of an Output Stream G_0



Example 7–3 shows an implementation of the output bit streams for the convolutional encoder of Figure 7–3.

Example 7–3. Generation of Output Streams G_0 and G_1

```
MOV   #09H, AR3                    ; AC0_H
MOV   #in_bit_stream, AR1


; Load 32 bits into the accumulators
MOV   *AR1+, AC0
ADD   *AR1 << #16, AC0
MOV   AC0, AC1

; Generate G0
XOR   AC1 << #-1, AC0              ; A = A XOR B>>1
XOR   AC1 << #-3, AC0              ; A = A XOR B>>3
MOV   AC0, T0                      ; Save G0

; Generate G1
XOR   AC1 << #-1, AC0              ; A = A XOR B>>1
XOR   AC1 << #-3, AC0              ; A = A XOR B>>3
XOR   AC1 << #-4, AC0              ; A = A XOR B>>4 --> AC0_L = G1

MOV   T0, *AR3                     ; AC0_H = G0 ------> AC0 = G0G1
```

Note: The algebraic instructions code example for Generation of Output Streams G_0 and G_1 is shown in Example B–37 on page B-39.

7.3.1 Bit-Stream Multiplexing and Demultiplexing

After a bit stream is convolved by the various generating polynomials, the redundant bits are typically multiplexed back into a single higher-rate bit stream. The C55x DSP has dedicated instructions that allow the extraction or insertion of a group of bits anywhere within a 32-bit accumulator. These instructions can be used to greatly expedite the bit-stream multiplexing operation on convolutional encoders.

Figure 7–5 illustrates the concept of bit multiplexing.

Figure 7–5. Bit Stream Multiplexing Concept

G_0
G_1
Multiplexor
G_0G_1

The C55x DSP has a dedicated instruction to perform the multiplexing of the bit streams:

dst = field_expand(ACx,k16)

This instruction executes in 1 cycle according to the following algorithm:

1) Clear the destination register.

2) Reset to 0 the bit index pointing within the destination register: index_in_dst.

3) Reset to 0 the bit index pointing within the source accumulator: index_in_ACx.

4) Scan the bit field mask k16 from bit 0 to bit 15, testing each bit. For each tested mask bit:

   If the tested bit is 1:

   a) Copy the bit pointed to by index_in_ACx to the bit pointed to by index_in_dst.

   b) Increment index_in_ACx.

   c) Increment index_in_dst, and test the next mask bit.

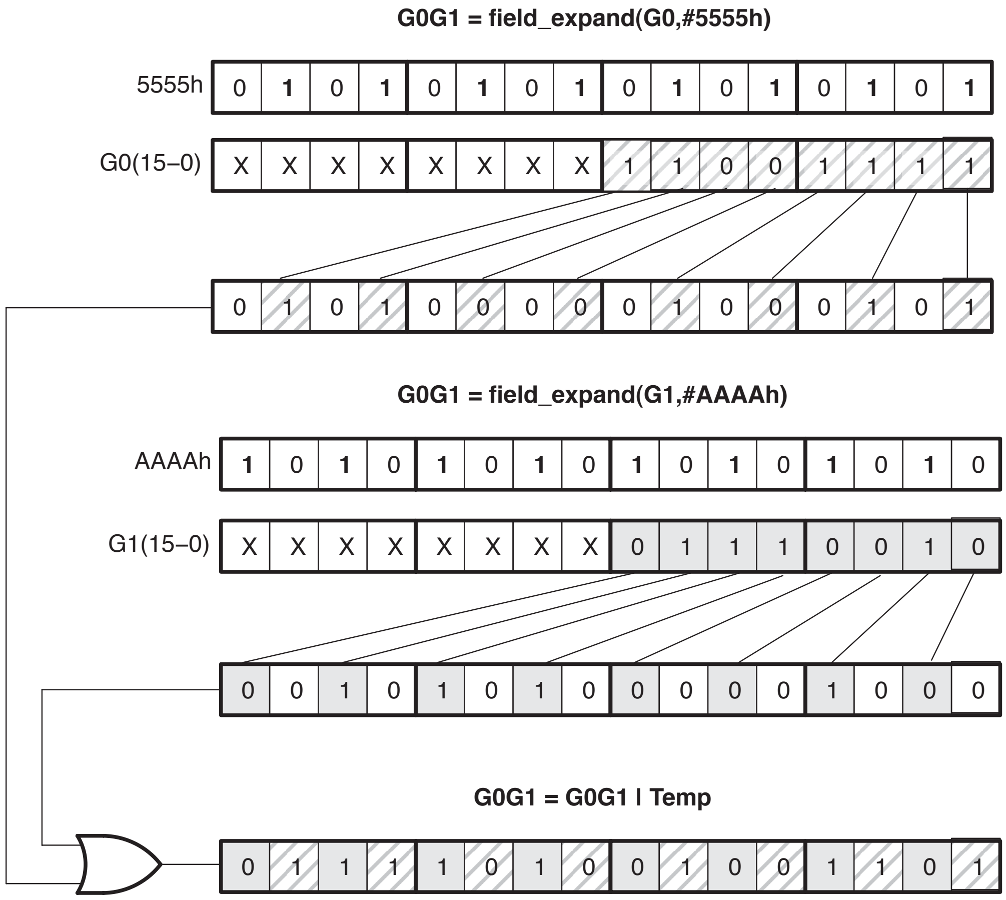
   If the tested bit is 0:

   Increment index_in_dst, and test the next mask bit.

Example 7–4 demonstrates the use of the field_expand() instruction.

Example 7–4. Multiplexing Two Bit Streams With the Field Expand Instruction

```
.asg   AC0, G0                       ; Assign G0 to register AC0.
.asg   AC1, G1                       ; Assign G1 to register AC1.
.asg   AC2, Temp                     ; Assign Temp to register AC2.
.asg   AC3, G0G1                     ; Assign G0G1 to register AC3.

G0 = *get_G0                         ; Load G0 stream.
G1 = *get_G1                         ; Load G1 stream.
G0G1 = field_expand(G0, #5555h)      ; Expand G0 stream.
Temp = G0G1                          ; Temporarily store expanded G0 stream.
G0G1 = field_expand(G1, #AAAAh)      ; Expand G1 stream
G0G1 = G0G1 | Temp                   ; Interleave expanded streams
```



At the receiver G0 and G1 must be extracted by de-interleaving the G0 G1 stream. The DSP has dedicated instructions to perform the de-interleaving of the bit streams:

dst = field_extract(ACx,k16)

This instruction executes in 1 cycle according to the following algorithm:

1) Clear the destination register.

2) Reset to 0 the bit index pointing within the destination register: index_in_dst.

3) Reset to 0 the bit index pointing within the source accumulator: index_in_ACx.

4) Scan the bit field mask k16 from bit 0 to bit 15, testing each bit. For each tested mask bit:

   If the tested bit is 1:

   a) Copy the bit pointed to by index_in_ACx to the bit pointed to by index_in_dst.

   b) Increment index_in_dst.

   c) Increment index_in_ACx, and test the next mask bit.

   If the tested bit is 0:

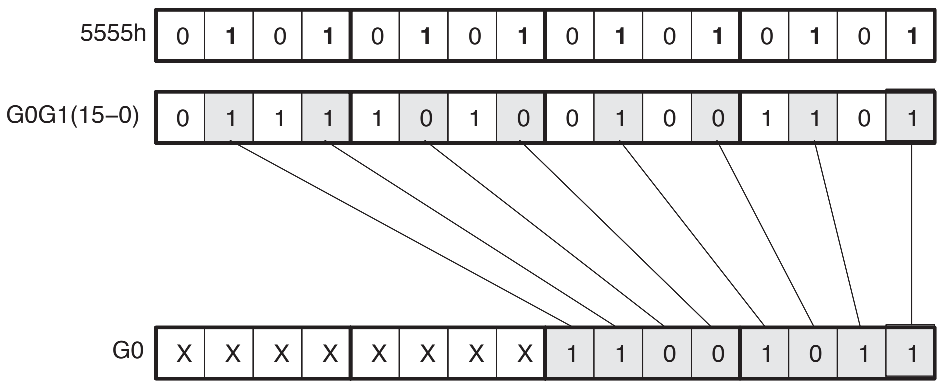
      Increment index_in_ACx, and test the next mask bit.

Example 7–5 demonstrates the use of the field_extract() instruction. The example shows how to de-multiplex the signal created in Example 7–4.

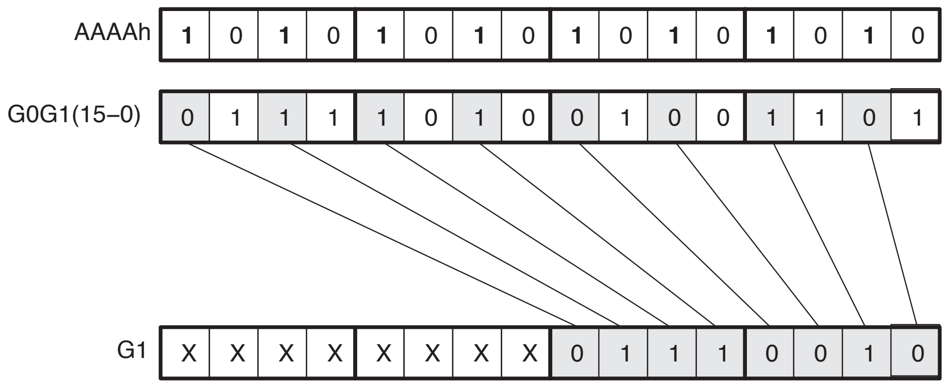
Example 7–5. Demultiplexing a Bit Stream With the Field Extract Instruction

```
    .asg   T2, G0                       ; Assign G0 to register T2.
    .asg   T3, G1                       ; Assign G1 to register T3.
    .asg   AC0, G0G1                    ; Assign G0G1 to register AC0.
    .asg   AR1, receive                 ; Assign receive to register AR1.

    G0G1 = *receive                     ; Get bit stream.
    G0 = field_extract(G0G1, #05555h)   ; Extract G0 from bit stream.
    G1 = field_extract(G0G1, #0AAAAh)   ; Extract G1 from bit stream.
```

G0 = field_extract(G0G1,#5555h)



G1 = field_extract(G0G1,#AAAAh)



7.4 Viterbi Algorithm for Channel Decoding (ADDSUB, SUBADD, MAXDIFF)

The Viterbi algorithm is widely used in communications systems for decoding information that has been convolutionally encoded. The most computationally intensive part of the routine is comprised of many add-compare-select (ACS) iterations. Minimizing the time for each ACS calculation is important. For a given system, the number of ACS calculations depends on the constraint length K and is equal to $2^{(K-2)}$. Thus, as K increases, the number of ACS calculations increases exponentially. The C55x DSP can perform the ACS operation in 1 cycle, due to dedicated instructions that support the Viterbi algorithm.

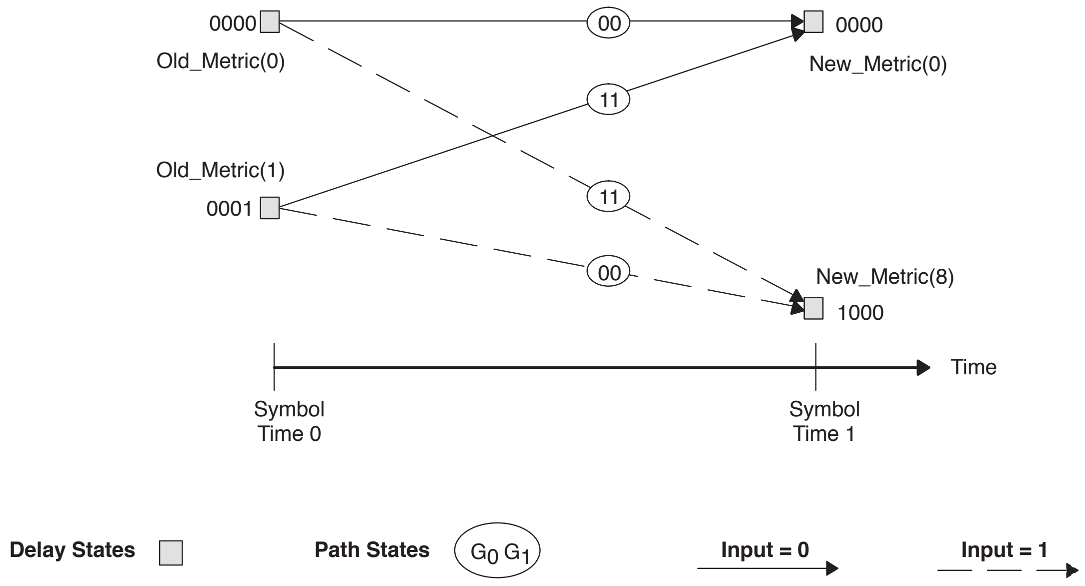
The convolutional encoder depicted in Figure 7–3 (page 7-10) is used in the global system for mobile communications (GSM) and is described by the following polynomials (K=5):

$$G_0(x) = 1 + x^3 + x^4 \qquad\qquad G_1(x) = 1 + x + x^3 + x^4$$

The convolutionally encoded outputs are dependent on past data inputs. Moreover, the contents of the encoder can be viewed as a finite state machine. A trellis diagram can represent the allowable state transitions, along with their corresponding path states. Decoding the data involves finding the optimal path through the trellis, by iteratively selecting possible paths to each delay state, for a given number of symbol time intervals. Two path metrics are calculated by adding a local distance to two old metrics. A comparison is made and a new path metric is selected from the two.

In the case of the GSM encoder, there are 16 possible states for every symbol time interval. For rate 1/n systems, there is some inherent symmetry in the trellis structure, which simplifies the calculations. The path states leading to a delay state are complementary. That is, if one path has G0G1 = 00, the other path has G0G1 = 11. This symmetry is based on the encoder polynomials and is true for most systems. Two starting and ending complementary states can be paired together, including all the paths between them, to form a butterfly structure (see Figure 7–6). Hence, only one local distance is needed for each butterfly; it is added and subtracted for each new state. Additionally, the old metric values are the same for both updates, so address manipulation is minimized.

Figure 7–6. Butterfly Structure for K = 5, Rate 1/2 GSM Convolutional Encoder



The following equation defines a local distance for the rate 1/2 GSM system:

$$LD = SD_0 \, G_0(j) + SD_1 \, G_1(j)$$

where

SD_X = Soft-decision input into the decoder
$G_X(j)$ = Expected encoder output for the symbol interval j

Usually, the Gx(j)s are coded as signed antipodal numbers, meaning that “0” corresponds to +1 and “1” corresponds to –1. This coding reduces the local distance calculation to simple addition and subtraction.

As shown in Example 7–6, the DSP can calculate a butterfly quickly by using its accumulators in a dual 16-bit computation mode. To determine the new path metric j, two possible path metrics, 2j and 2j+1, are calculated in parallel with local distances (LD and –LD) using the add-subtract (ADDSUB) instruction and an accumulator. To determine the new path metric ($j+2^{(K-2)}$), the subtract-add (SUBADD) instruction is also used, using the old path metrics plus local distances stored in a separate accumulator. The MAXDIFF instruction is then used on both accumulators to determine the new path metrics. The MAXDIFF instruction compares the upper and lower 16-bit values for two given accumulators, and stores the larger values in a third accumulator.

Example 7–6 shows two macros for Viterbi butterfly calculations. The add-subtract (ADDSUB) and subtract-add (SUBADD) computations in the two macros are performed in alternating order, which is based on the expected encoder state. A local distance is stored in the register T3 beforehand. The MAXDIFF instruction performs the add-compare-select function in 1 cycle. The updated path metrics are saved to memory by the next two lines of code.

Two 16-bit transition registers (TRN0 and TRN1) are updated with every comparison done by the MAXDIFF instruction, so that the selected path metric can be tracked. TRN0 tracks the results from the high part data path, and TRN1 tracks the low part data path. These bits are later used in traceback, to determine the original uncoded data. Using separate transition registers allows for storing the selection bits linearly, which simplifies traceback. In contrast, the TMS320C54x™ (C54x™) DSP has only one transition register, storing the selection bits as 0, 8, 1, 9, etc. As a result, on the C54x DSP, additional lines of code are needed to process these bits during traceback.

You can make the Viterbi butterfly calculations faster by implementing user-defined instruction parallelism (see section 4.2, page 4-16) and software pipelining. Example 7–7 (page 7-20) shows the inner loop of a Viterbi butterfly algorithm. The algorithm places some instructions in parallel (||) in the CPU, and the algorithm implements software pipelining by saving previous results at the same time it performs new calculations. Other operations, such as loading the appropriate local distances, are coded with the butterfly algorithm.

Example 7–6. Viterbi Butterflies for Channel Coding

```
BFLY_DIR_MNEM .MACRO
;new_metric(j)&(j+2^(K-2))
                                          ;
   ADDSUB T3, *AR5+, AC0                  ; AC0(39-16) = Old_Met(2*j)+LD
                                          ; AC0(15-0) = Old_met(2*j+1)-LD

   SUBADD T3, *AR5+, AC1                  ; AC1(39-16) = Old_Met(2*j)-LD
                                          ; AC1(15-0) = Old_met(2*j+1)+LD

   MAXDIFF AC0, AC1, AC2, AC3             ; Compare AC0 and AC1
   MOV AC2, *AR3+, *AR4+                  ; Store the lower maxima (with AR3)
                                          ; and upper maxima (with AR4)
   .ENDM

BFLY_REV_MNEM        .MACRO
;new_metric(j)&(j+2^(K-2))
   SUBADD T3, *AR5+, AC0                  ; AC0(39-16) = Old_Met(2*j)-LD
                                          ; AC0(15-0) = Old_met(2*j+1)+LD

   ADDSUB T3, *AR5+, AC1                  ; AC1(39-16) = Old_Met(2*j)+LD
                                          ; AC1(15-0) = Old_met(2*j+1)-LD

   MAXDIFF AC0, AC1, AC2, AC3             ; Compare AC0 and AC1
   MOV AC2, *AR3+, *AR4+                  ; Store the lower maxima (with AR3)
                                          ; and upper maxima (with AR4)
   .ENDM
```

Note: The algebraic instructions code example for Viterbi Butterflies for Channel Coding is shown in Example B–38 on page B-39.

Example 7–7. Viterbi Butterflies Using Instruction Parallelism

```
   RPTBLOCAL end
butterfly:
      ADDSUB T3, *AR0+, AC0                ; AC0(39–16) = Old_Met(2*j)+LD
                                           ; AC0(15–0) = Old_met(2*j+1)–LD
      || MOV *AR5+, AR7

      SUBADD T3, *AR0+, AC1                ; AC1(39–16) = Old_Met(2*j)–LD
                                           ; AC1(15–0) = Old_met(2*j+1)+LD
      || MOV *AR6, T3                      ; Load new local distance

      MOV AC2, *AR2+, *AR2(T0)             ; Store lower and upper maxima
                                           ; from previous MAXDIFF operation
      || MAXDIFF AC0, AC1, AC2, AC3        ; Compare AC0 and AC1

      ADDSUB T3, *AR0+, AC0                ; AC0(39–16) = Old_Met(2*j)+LD
                                           ; AC0(15–0) = Old_met(2*j+1)–LD
      || MOV *AR5+, AR6

      SUBADD T3, *AR0+, AC1                ; AC1(39–16) = Old_Met(2*j)–LD
                                           ; AC1(15–0) = Old_met(2*j+1)+LD
      || MOV *AR7, T3                      ; Load new local distance

end   MOV AC2, *AR2(T0), *AR2+             ; Store lower and upper maxima
                                           ; from previous MAXDIFF operation
      || MAXDIFF AC0, AC1, AC2, AC3        ; Compare AC0 and AC1
```

Note: The algebraic instructions code example for Viterbi Butterflies Using Instruction Parallelism is shown in Example B–39 on page B-40.

Chapter 8

TI C55x DSPLIB

The TI C55x DSPLIB is an optimized DSP function library for C programmers on TMS320C55x (C55x) 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 C55x Code Composer Studio product under the c:\ti\C5500\dsplib\55x_src directory.**

Full documentation on C55x DSPLIB can be found in the *TMS320C55x DSP Library Programmer's Reference* (SPRU422).

Topic **Page**

8.1 Features and Benefits

- ❑ Hand-coded assembly optimized routines
- ❑ C-callable routines fully compatible with the C55x DSP compiler
- ❑ Fractional Q15-format operands supported
- ❑ Complete set of examples on usage provided
- ❑ Benchmarks (cycles and code size) provided
- ❑ Tested against Matlab™ scripts

8.2 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.

8.3 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.

8.4 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, *55xdsp.lib*.

❑ Use a correct linker command file describing the memory configuration available in your C55x 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.

8.5 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 C55x DSP C compiler calling conventions. Refer to the *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281), 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.

8.6 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.
- *55x.cmd*: an example of a linker command you can use for this function.

8.7 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 *TMS320C55x DSP Library Programmer's Reference* (SPRU422).

Appendix A

Special D-Unit Instructions

This appendix is a list of D-unit instructions where A-unit registers are read ("pre-fetched") in the Read phase of the execution pipeline.

Instruction	Description
ABS_RR	dst = lsrcl
ADD_RM	dst = src + Smem
ADD_RM_IS	ACy = ACx + (Smem << Tx)
ADD_RR	dst = dst + src
ADD_RR_IS	ACy = ACy + (ACx << Tx)
ADD_RWK	dst = src + K16
ADD_SUB_RM_H	HI(ACx) = Smem + Tx , LO(ACx) = Smem – Tx
ADSC_RM_4	ACy = ads2c(Smem,ACx,Tx,TC1,TC2)
AND_RBK	dst = src & k8
AND_RM	dst = src & Smem
AND_RR	dst = dst & src
AND_RWK	dst = src & k16
CMPL_RR	dst = ~src
CMPR_RR_10	TCx = uns(src RELOP dst) {==,<,>=,!=}
CMPR_RR_11	TCx = TCy & uns(src RELOP dst) {==,<,>=,!=}
CMPR_RR_12	TCx = !TCy & uns(src RELOP dst) {==,<,>=,!=}
CMPR_RR_13	TCx = TCy l uns(src RELOP dst) {==,<,>=,!=}
CMPR_RR_14	TCx = !TCy l uns(src RELOP dst) {==,<,>=,!=}
DADD_RLM_D	HI(ACx) = HI(Lmem) + Tx , LO(ACx) = LO(Lmem) + Tx
DADS_RLM	HI(ACx) = HI(Lmem) + Tx , LO(ACx) = LO(Lmem) – Tx

Instruction	Description
DRSUB_RLM_D	HI(ACx) = HI(Lmem) – Tx , LO(ACx) = LO(Lmem) – Tx
DSAD_RLM	HI(ACx) = HI(Lmem) – Tx , LO(ACx) = LO(Lmem) + Tx
DSUB_RLM_D	HI(ACx) = Tx – HI(Lmem) , LO(ACx) = Tx – LO(Lmem)
LD_MAC_R	ACx = rnd(ACx + (Tx * Xmem)) ,ACy = Ymem << #16[,T3 = Xmem]
LD_MAS_R	ACx = rnd(ACx – (Tx * Xmem)) ,ACy = Ymem << #16[,T3 = Xmem]
LD_RM_IS	ACx = rnd(Smem << Tx)
MAC_R_RBK	ACy = rnd(ACx + (Tx * K8))
MAC_R_RM	ACy = rnd(ACx + (Tx * Smem)) [,T3 = Smem]
MAC_R_RR	ACy = rnd(ACy + (ACx * Tx))
MAC_R_RWK_S	ACy = rnd(ACx + (Tx * K16))
MAS_R_RM	ACy = rnd(ACx – (Tx * Smem)) [,T3 = Smem]
MAS_R_RR	ACy = rnd(ACy – (ACx * Tx))
MAX_RR	dst = max(src,dst)
MIN_RR	dst = min(src,dst)
MPOLY_R_RR	ACy = rnd((ACy * Tx) + ACx)
MPYU_R_RM	ACy = rnd(uns(Tx * Smem)) [,T3 = Smem]
MPY_R_RR_DR	ACy = rnd(ACx * Tx)
MV_RR	dst = src
MV_R_ACH	HI(ACx) = DAx
MV_XRR	dst = src
NEG_RR	dst = –src
OR_RBK	dst = src l k8
OR_RM	dst = src l Smem
OR_RR	dst = dst l src
OR_RWK	dst = src l k16
ROL_RR_1	dst = {TC2,Carry} \\ src \\ {TC2,Carry}

Instruction	Description
ROR_RR_1	dst = {TC2,Carry} // src // {TC2,Carry}
RSUB_RM	dst = Smem – src
SFTA_RR_IS	ACy = ACx << Tx
SFTA_RR_ISC	ACy = ACx <<C Tx
SFTL_RR_IS	ACy = ACx <<< Tx
STH_RS_RM_ASM	Smem = HI(saturate(uns(rnd(ACx << Tx))))
STH_R_RM_ASM	Smem = HI(rnd(ACx << Tx))
ST_ADD	ACy = ACx + (Xmem << #16) ,Ymem = HI(ACy << T2)
ST_LD	ACy = Xmem << #16 ,Ymem = HI(ACx << T2)
ST_MAC_R	ACy = rnd(ACy + (Tx * Xmem)) ,Ymem = HI(ACx << T2)[,T3 = Xmem]
ST_MAS_R	ACy = rnd(ACy – (Tx * Xmem)) ,Ymem = HI(ACx << T2)[,T3 = Xmem]
ST_MPY_R	ACy = rnd(Tx * Xmem) ,Ymem = HI(ACx << T2)[,T3 = Xmem]
ST_RM_ASM	Smem = LO(ACx << Tx)
ST_SUB	ACy = (Xmem << #16) – ACx ,Ymem = HI(ACy << T2)
SUB_ADD_RM_H	HI(ACx) = Smem – Tx , LO(ACx) = Smem + Tx
SUB_RM	dst = src – Smem
SUB_RM_IS	ACy = ACx – (Smem << Tx)
SUB_RR	dst = dst – src
SUB_RR_IS	ACy = ACy – (ACx << Tx)
SUB_RWK	dst = src – K16
XOR_RBK	dst = src ^ k8
XOR_RM	dst = src ^ Smem
XOR_RR	dst = dst ^ src
XOR_RWK	dst = src ^ k16

Appendix B

Algebraic Instructions Code Examples

This appendix shows the algebraic instructions code examples that correspond to the mnemonic instructions code examples shown in Chapters 2 through 7.

Example B–1. Partial Assembly Code of test.asm (Step 1)

```
* Step 1: Section allocation
* ------
       .def x,y,init
x      .usect “vars”,4         ; reserve 4 uninitalized locations for var x
y      .usect “vars”,1         ; reserve 1 uninitialized location for result y

       .sect “table”           ; create initialized section “table” to
init   .int 1,2,3,4            ; contain initialization values for x

       .text                   ; create code section (default is .text)
       .def start              ; make the start label global
start                          ; define label to the start of the code
```

Example B–2. Partial Assembly Code of test.asm (Step 2)

```
* Step 2: Processor mode initialization
* ------
   bit(ST1, #ST1_C54CM) = #0  ; set processor to C55x native mode instead of
                              C54x compatibility mode (reset value)
   bit(ST2, #ST2_AR0LC) = #0  ; set AR0 register in linear mode (reset value)
   bit(ST2, #ST2_AR6LC) = #0  ; set AR6 register in linear mode (reset value)
```

Example B–3. Partial Assembly Code of test.asm (Part3)

```
* Step 3a: Copy initialization values to vector x using indirect addressing
* -------
copy
   XAR0 = #x                 ; XAR0 pointing to startn of x array
   XAR6 = #init              ; XAR6 pointing to start of init array

   *AR0+ = *AR6+             ; copy from source ”init” to destination ”x”
   *AR0+ = *AR6+
   *AR0+ = *AR6+
   *AR0  = *AR6

* Step 3b: Add values of vector x elements using direct addressing
* -------
add
   XDP = #x                  ; XDP pointing to variable x
   .dp  x                    ; and the assembler is notified

   AC0 = @x
   AC0 += @(x+3)
   AC0 += @(x+1)
   AC0 += @(x+2)

* Step 3c. Write the result to y using absolute addressing
* -------
   *(#y) = AC0

end
   nop
   goto  end
```

Example B–4. Assembly Code Generated With –o3 and –pm Options

```
_sum:
;** Parameter deleted n == 9u
        T0 = #0  ; |3|
        repeat(#9)
            T0 = T0 + *AR0+

        return    ; |11|
_main:
        SP = SP + #–1
        XAR0 = #_a ; |9|
        call #_sum ; |9|
                                        ; call occurs [#_sum]   ; |9|
        *(#_sum1) = T0 ; |9|
        XAR0 = #_b ; |10|
        call #_sum ; |10|
                                        ; call occurs [#_sum]   ; |10|
        *(#_sum2) = T0 ; |10|
        S = SP + #1P
        return
                                        ; return occurs
```

Example B–5. Assembly Generated Using `–o3`, `–pm`, and `–oi50`

```
_sum:
        T0 = #0 ; |3|
        repeat(#9)
            T0 = T0 + *AR0+
        return    ; |11|
_main:
        XAR3 = #_a ; |9|
        repeat(#9)
||      AR1 = #0 ; |3|
            AR1 = AR1 + *AR3+

        *(#_sum1) = AR1 ; |11|
        AR1 = #0   ; |3|
        XAR3 = #_b ; |10|
        repeat(#9)
            AR1 = AR1 + *AR3+

        *(#_sum2) = AR1 ; |11|
        return
```

Example B–6. Assembly Code for `localrepeat` Generated by the Compiler

```
_vecsum:
        AR3 = T0 – #1
        BRC0 = AR3
        localrepeat {
            AC0 = (*AR0+ << #16) + (*AR1+ << #16) ; |7|
            *AR2+ = HI(AC0) ; |7|
        }                                ; loop ends   ; |8|
L2:
        return
```

Example B–7. Inefficient Loop Code for Variable and Constraints (Assembly)

```
_sum:
        AR1 = #0  ; |3|
        if (T0 <= #0) goto L2 ; |6|
                                        ; branch occurs  ; |6|
        AR2 = T0 - #1
        CSR = AR2
        repeat(CSR)
            AR1 = AR1 + *AR0+

        T0 = AR1  ; |11|
        return    ; |11|
```

Example B–8. Assembly Code Generated With the `MUST_ITERATE` Pragma

```
_sum:
        AR2 = T0 - #1
        CSR = AR2
        AR1 = #0  ; |3|
        repeat(CSR)
            AR1 = AR1 + *AR0+

        T0 = AR1  ; |12|
        return    ; |12|
```

Example B–9. Generated Assembly for FIR Filter Showing Dual-MAC

```
_fir:
        AR3 = T0 + #1
        AR3 = AR3 >> #1
        AR3 = AR3 - #1
        BRC0 = AR3
        push(T3,T2)
        T3 = #0    ; |6|
||      XCDP = XAR0
        SP = SP + #-1
        localrepeat {

            T2 = T1 - #1
            XAR3 = XAR1
            CSR = T2
            AR3 = AR3 + T3
            XAR4 = XAR3
            AR4 = AR4 + #1
            AC0 = #0  ; |8|
            repeat(CSR)
||          AC1 = AC0 ; |8|

                AC0 = AC0 + (*AR4+ * coef(*CDP+)), AC1 = AC1 + (*AR3+ *
coef(*CDP+))

            XAR0 = XCDP
            T3 = T3 + #2
            AR0 = AR0 - T1
||          *AR2(short(#1)) = HI(AC0)
            AR2 = AR2 + #2
||          *AR2 = HI(AC1)
            XCDP = XAR0
        }

        SP = SP + #1
        T3,T2 = pop()
        return
```

Example B–10. Inefficient Assembly Code Generated by C Version of Saturated Addition

```
_sadd:
        AR1 = T1  ; |5|
        T1 = T1 ^ T0 ; |9|
        TC1 = bit(T1,@#15) ; |9|
        AR1 = AR1 + T0
        if (TC1) goto L2 ; |9|
                                ; branch occurs  ; |9|
        AR2 = T0  ; |9|
        AR2 = AR2 ^ AR1 ; |9|
        TC1 = bit(AR2,@#15) ; |9|
        if (!TC1) goto L2 ; |9|
                                ; branch occurs  ; |9|
        if (T0 < #0) goto L1 ; |22|
                                ; branch occurs  ; |22|
        T0 = #32767 ; |22|
        goto L3   ; |22|
                                ; branch occurs  ; |22|
L1:
        AR1 = #-32768 ; |22|
L2:
        T0 = AR1  ; |25|
L3:
        return    ; |25|
                                ; return occurs  ; |25|
```

Example B–11. Assembly Code Generated When Using Compiler Intrinsic for Saturated Add

```
_sadd:
        bit(ST3, #ST3_SATA) = #1
        T0 = T0 + T1 ; |3|
        bit(ST3, #ST3_SATA) = #0
        return    ; |3|
                                        ; return occurs  ; |3|
```

Example B–12. Assembly Output for Circular Addressing C Code

```
_circ:
        AC0 = #0  ; |7|
        if (T1 <= #0) goto L2 ; |9|
                                        ; branch occurs  ; |9|
        AR3 = T1 - #1
        BRC0 = AR3
        AR2 = #0  ; |6|
        localrepeat {
                                            ; loop starts
L1:
            AC0 = AC0 + (*AR1+ * *AR0+) ; |11|
||          AR2 = AR2 + #1
            TC1 = (AR2 < T0) ; |12|
            if (!TC1) execute (D_Unit) ||
                AR1 = AR1 - T0
            if (!TC1) execute (D_Unit) ||
                AR2 = AR2 - T0
        }                               ; loop ends   ; |13|
L2:
        return    ; |14|
                                        ; return occurs  ; |14|
```

Example B–13. Assembly Output for Circular Addressing Using Modulo

```
_circ:
        push(T3,T2)
        SP = SP + #-7
        dbl(*SP(#0)) = XAR1
||      AC0 = #0  ; |4|
        dbl(*SP(#2)) = AC0 ; |4|
||      T2 = T0   ; |2|
        if (T1 <= #0) goto L2 ; |6|
                                ; branch occurs  ; |6|
        T0 = #0   ; |3|
        T3 = T1
||      dbl(*SP(#4)) = XAR0
L1:
        XAR3 = dbl(*SP(#0))
        T1 = *AR3(T0) ; |8|
        XAR3 = dbl(*SP(#4))
        AC0 = dbl(*SP(#2)) ; |8|
        T0 = T0 + #1
        AC0 = AC0 + (T1 * *AR3+) ; |8|
        dbl(*SP(#2)) = AC0 ; |8|
        dbl(*SP(#4)) = XAR3
        call #I$$MOD ; |9|
||      T1 = T2   ; |9|
                                ; call occurs [#I$$MOD] ; |9|
        T3 = T3 - #1
        if (T3 != #0) goto L1 ; |10|
                                ; branch occurs  ; |10|
L2:
        AC0 = dbl(*SP(#2))
        SP = SP + #7 ; |11|
        T3,T2 = pop()
        return    ; |11|
                                ; return occurs  ; |11|
```

Example B–14. Complex Vector Multiplication Code

```
N      .set   3                       ; Length of each complex vector

       .data
A      .int   1,2,3,4,5,6             ; Complex input vector #1
B      .int   7,8,9,10,11,12          ; Complex input vector #2

;Results are: 0xfff7, 0x0016, 0xfff3, 0x0042, 0xffef, 0x007e

       .bss C, 2*N, ,1                ; Results vector, long–word aligned

       .text
       bit(ST2,#ST2_ARMS) = #0        ; Clear ARMS bit (select DSP mode)
       .arms_off                      ; Tell assembler ARMS = 0

cplxmul:
       XAR0 = #A                      ; Pointer to A vector
       XCDP = #B                      ; Pointer to B vector
       XAR1 = #C                      ; Pointer to C vector
       BRC0 = #(N–1)                  ; Load loop counter
       T0 = #1                        ; Pointer offset
       T1 = #2                        ; Pointer increment

       localrepeat {                  ; Start the loop

       AC0 = *AR0 * coef(*CDP+),
       AC1 = *AR0(T0) * coef(*CDP+)

       AC0 = AC0 – (*AR0(T0) * coef(*CDP+)),
       AC1 = AC1 + (*(AR0+T1) * coef(*CDP+))

       *AR1+ = pair(LO(AC0))          ; Store complex result
       }                              ; End of loop
```

Example B–15. Block FIR Filter Code (Not Optimized)

```
N_TAPS    .set   4                              ; Number of filter taps
N_DATA    .set   11                             ; Number of input values

          .data
COEFFS    .int   1,2,3,4                        ; Coefficients
IN_DATA   .int   1,2,3,4,5,6,7,8,9,10,11        ; Input vector

;Results are:    0x0014, 0x001E, 0x0028, 0x0032,
;                0x003C, 0x0046, 0x0050, 0x005A

          .bss  OUT_DATA, N_DATA - N_TAPS + 1  ; Output vector

          .text
          bit(ST2,#ST2_ARMS) = #0       ; Clear ARMS bit (select DSP mode)
          .arms_off                     ; Tell assembler ARMS = 0

bfir:
       XCDP = #COEFFS                           ; Pointer to coefficient array
       XAR0 = #(IN_DATA + N_TAPS - 1)           ; Pointer to input vector
       XAR1 = #(IN_DATA + N_TAPS)               ; 2nd pointer to input vector
       XAR2 = #OUT_DATA                         ; Pointer to output vector
       BRC0 = #((N_DATA - N_TAPS + 1)/2 - 1)    ; Load outer loop counter
       CSR = #(N_TAPS - 1)                      ; Load inner loop counter

       localrepeat {                            ; Start the outer loop

       AC0 = #0                                 ; Clear AC0
       AC1 = #0                                 ; Clear AC1

       repeat(CSR)                              ; Start the inner loop
       AC0 = AC0 + ( *AR0- * coef(*CDP+) ),     ; All taps
       AC1 = AC1 + ( *AR1- * coef(*CDP+) )

       *AR2+ = AC0                              ; Write 1st result
       *AR2+ = AC1                              ; Write 2nd result

       CDP = #COEFFS                            ; Rewind coefficient pointer
       AR0 = AR0 + #(N_TAPS + 2)                ; Adjust 1st input vector
                                                ;   pointer
       AR1 = AR1 + #(N_TAPS + 2)                ; Adjust 2nd input vector
                                                ;   pointer

       }                                        ; End of outer loop
```

Example B–16. Block FIR Filter Code (Optimized)

```
N_TAPS    .set   4                            ; Number of filter taps
N_DATA    .set   11                           ; Number of input values

          .data
COEFFS    .int   1,2,3,4                      ; Coefficients
IN_DATA   .int   1,2,3,4,5,6,7,8,9,10,11      ; Input vector

;Results are:    0x0014, 0x001E, 0x0028, 0x0032,
;                0x003C, 0x0046, 0x0050, 0x005A

          .bss   OUT_DATA, N_DATA – N_TAPS + 1, ,1
                                              ; Output vector, long-word
                                              ;   aligned

          .text
          bit(ST2,#ST2_ARMS) = #0       ; Clear ARMS bit (select DSP mode)
          .arms_off                     ; Tell assembler ARMS = 0

bfir:
          XCDP = #COEFFS                      ; Pointer to coefficient array
          XAR0 = #(IN_DATA + N_TAPS – 1)      ; Pointer to input vector
          XAR1 = #(IN_DATA + N_TAPS)          ; 2nd pointer to input vector
          XAR2 = #OUT_DATA                    ; Pointer to output vector
          BRC0 = #((N_DATA – N_TAPS + 1)/2 – 1)
                                              ; Load outer loop counter
          CSR = #(N_TAPS – 3)                 ; Load inner loop counter
          T0 = #(–(N_TAPS – 1))               ; CDP rewind increment

          T1 = #(N_TAPS + 1)                  ; ARx rewind increment
          ||localrepeat {                     ; Start the outer loop

          AC0 = *AR0– * coef(*CDP+),          ; 1st tap
          AC1 = *AR1– * coef(*CDP+)

          repeat(CSR)                         ; Start the inner loop
          AC0 = AC0 + ( *AR0– * coef(*CDP+) ), ; Inner taps
          AC1 = AC1 + ( *AR1– * coef(*CDP+) )

          AC0 = AC0 + ( *(AR0+T1) * coef(*(CDP+T0)) ),   ; Last tap
          AC1 = AC1 + ( *(AR1+T1) * coef(*(CDP+T0)) )

          *AR2+ = pair(LO(AC0))               ; Store both results

          }                                   ; End of outer loop
```

Example B–17. A-Unit Code With No User-Defined Parallelism

```
; Variables
          .data

COEFF1    .word  0x0123               ; First set of coefficients
          .word  0x1234
          .word  0x2345
          .word  0x3456
          .word  0x4567

COEFF2    .word  0x7654               ; Second set of coefficients
          .word  0x6543
          .word  0x5432
          .word  0x4321
          .word  0x3210

HST_FLAG  .set   0x2000               ; Host flag address
HST_DATA  .set   0x2001               ; Host data address

CHANGE    .set   0x0000               ; "Change coefficients" command from host
READY     .set   0x0000               ; "READY" Flag from Host
BUSY      .set   0x1111               ; "BUSY" Flag set by DSP

          .global   start_a1

          .text

start_a1:
          AR0 = #HST_FLAG             ; AR0 points to Host Flag
          AR2 = #HST_DATA             ; AR2 points to Host Data
          AR1 = #COEFF1               ; AR1 points to COEFF1 buffer initially
          AR3 = #COEFF2               ; AR3 points to COEFF2 buffer initially
          CSR = #4                    ; Set CSR = 4 for repeat in COMPUTE
          BIT(ST1, #ST1_FRCT) = #1    ; Set fractional mode bit
          BIT(ST1, #ST1_SXMD) = #1    ; Set sign-extension mode bit
```

Example B–17. A-Unit Code With No User-Defined Parallelism (Continued)

```
LOOP:
          T0 = *AR0                      ; T0 = Host Flag
          if (T0 == #READY) GOTO PROCESS ; If Host Flag is ”READY”, continue
          GOTO LOOP                      ; process – else poll Host Flag again

PROCESS:
          T0 = *AR2                  ; T0 = Host Data

          if (T0 == #CHANGE) EXECUTE(AD_UNIT)
                                     ; The choice of either set of
                                     ; coefficients is based on the value
                                     ; of T0. COMPUTE uses AR3 for
                                     ; computation, so we need to
                                     ; load AR3 correctly here.

          SWAP(AR1, AR3)             ; Host message was ”CHANGE”, so we
                                     ; need to swap the two coefficient
                                     ; pointers.

          CALL COMPUTE               ; Compute subroutine

          *AR2 = AR4                 ; Write result to Host Data
          *AR0 = #BUSY               ; Set Host Flag to Busy
          GOTO LOOP                  ; Infinite loop continues
END

COMPUTE:
          AC1 = #0                   ; Initialize AC1 to 0
          REPEAT(CSR)                ; CSR has a value of 4
           AC1 = AC1 + (*AR2 * *AR3+) ; This MAC operation is performed
                                      ; 5 times
          AR4 = AC1                  ; Result is in AR4

          RETURN

HALT:
          GOTO HALT
```

Example B–18. A-Unit Code in Example B–17 Modified to Take Advantage of Parallelism

```
; Variables
          .data

COEFF1    .word  0x0123               ; First set of coefficients
          .word  0x1234
          .word  0x2345
          .word  0x3456
          .word  0x4567

COEFF2    .word  0x7654               ; Second set of coefficients
          .word  0x6543
          .word  0x5432
          .word  0x4321
          .word  0x3210

HST_FLAG  .set   0x2000               ; Host flag address
HST_DATA  .set   0x2001               ; Host data address

CHANGE    .set   0x0000               ; “Change coefficients” command from host
READY     .set   0x0000               ; “READY” Flag from Host
BUSY      .set   0x1111               ; “BUSY” Flag set by DSP

          .global   start_a2

          .text

start_a2:

          AR0 = #HST_FLAG             ; AR0 points to Host Flag
          AR2 = #HST_DATA             ; AR2 points to Host Data
          AR1 = #COEFF1               ; AR1 points to COEFF1 buffer initially
          AR3 = #COEFF2               ; AR3 points to COEFF2 buffer initially

          CSR = #4                    ; Set CSR = 4 for repeat in COMPUTE
          || BIT(ST1, #ST1_FRCT) = #1 ; Set fractional mode bit

          BIT(ST1, #ST1_SXMD) = #1    ; Set sign-extension mode bit

LOOP:
          T0 = *AR0                      ; T0 = Host Flag
          if (T0 == #READY) GOTO PROCESS ; If Host Flag is “READY”, continue
          GOTO LOOP                      ; process – else poll Host Flag again
```

Example B–18. A-Unit Code in Example B–17 Modified to Take Advantage of Parallelism (Continued)

```
PROCESS:
          T0 = *AR2                     ; T0 = Host Data

          if (T0 == #CHANGE) EXECUTE(AD_UNIT)
                                        ; The choice of either set of
                                        ; coefficients is based on the value
                                        ; of T0. COMPUTE uses AR3 for
                                        ; computation, so we need to
                                        ; load AR3 correctly here.


          || SWAP(AR1, AR3)             ; Host message was ”CHANGE”, so we
                                        ; need to swap the two coefficient
                                        ; pointers.

          CALL COMPUTE                  ; Compute subroutine


          *AR2 = AR4                    ; Write result to Host Data
          || *AR0 = #BUSY               ; Set Host Flag to Busy

          GOTO LOOP                     ; Infinite loop continues

END

COMPUTE:
          AC1 = #0                      ; Initialize AC1 to 0
          || REPEAT(CSR)                ; CSR has a value of 4
           AC1 = AC1 + (*AR2 * *AR3+)   ; This MAC operation is performed
                                         ; 5 times
          AR4 = AC1                     ; Result is in AR4
          || RETURN


HALT:
          GOTO HALT
```

Example B–19. P-Unit Code With No User-Defined Parallelism

```
       .CPL_off                    ; Tell assembler that CPL bit is 0
                                   ; (Direct addressing is done with DP)

; Variables
       .data

var1   .word  0x0004
var2   .word  0x0000

       .global   start_p1

       .text

start_p1:
       XDP = #var1
       AR3 = #var2

       BRC0 = #0007h               ; BRC0 loaded using KPB
       BRC1 = *AR3                 ; BRC1 loaded using DB

       AC2 = #0006h

       BLOCKREPEAT {
          AC1 = AC2
          AR1 = #8000h
          LOCALREPEAT {
             AC1 = AC1 - #1
             *AR1+ = AC1
          }
          AC2 = AC2 + #1
       }
       @(AC0_L) = BRC0             ; AC0_L loaded using EB
       @(AC1_L) = BRC1             ; AC1_L loaded using EB

       if (AC0 >= #0) goto start_p1:
       if (AC1 >= #0) goto start_p1:
end_p1
```

Example B–20. P-Unit Code in Example B–19 Modified to Take Advantage of Parallelism

```
       .CPL_off                     ; Tell assembler that CPL bit is 0
                                    ; (Direct addressing is done with DP)

; Variables
       .data

var1   .word  0x0004
var2   .word  0x0000

       .global   start_p2

       .text

start_p2:
       XDP = #var1
       AR3 = #var2

       BRC0 = #0007h                ; BRC0 loaded using KPB
       || BRC1 = *AR3               ; BRC1 loaded using DB

       AC2 = #0006h
       || BLOCKREPEAT {
          AC1 = AC2
          AR1 = #8000h
          || LOCALREPEAT {
             AC1 = AC1 - #1
             *AR1+ = AC1
          }
          AC2 = AC2 + #1
       }
       @(AC0_L) = BRC0              ; AC0_L loaded using EB
       @(AC1_L) = AR1               ; AC1_L loaded using EB


       if (AC0 >= #0) goto start_p2
       if (AC1 >= #0) goto start_p2
end_p2
```

Example B–21. D-Unit Code With No User-Defined Parallelism

```
; Variables
       .data

var1   .word  0x8000
var2   .word  0x0004

       .global   start_d1

       .text

start_d1:
       AR3 = #var1
       AR4 = #var2

       AC0 = #0004h              ; AC0 loaded using KDB
       AC2 = *AR3                ; AC2 loaded using DB

       T0 = #5A5Ah               ; T0 loaded with constant, 0x5A5A

       AC2 = AC2 + (AC0 * T0)    ; MAC
       AC1 = AC1 + (AC2 * T0)    ; MAC
       SWAP(AC0, AC2)            ; SWAP

       *AR3 = HI(AC1)            ; Store result in AC1
       *AR4 = HI(AC0)            ; Store result in AC0
end_d1
```

Example B–22. D-Unit Code in Example B–21 Modified to Take Advantage of Parallelism

```
; Variables
       .data

var1   .word  0x8000
var2   .word  0x0004

       .global   start_d2

       .text

start_d2:
       AR3 = #var1
       AR4 = #var2

       AC0 = #0004h                ; AC0 loaded using KDB
       || AC2 = *AR3               ; AC2 loaded using DB

       T0 = #5A5Ah                 ; T0 loaded with constant, 0x5A5A

       AC2 = AC2 + (AC0 * T0)      ; MAC
       AC1 = AC1 + (AC2 * T0)      ; MAC
       ||SWAP(AC0, AC2)            ; SWAP

       *AR3 = HI(AC1)              ; Store result in AC1
       || *AR4 = HI(AC0)           ; Store result in AC0
end_d2
```

Example B–23. Code That Uses Multiple CPU Units But No User-Defined Parallelism

```
    .CPL_ON                     ; Tell assembler that CPL bit is 1
                                ; (SP direct addressing like *SP(0) is enabled)

;   Register usage
;   -----------------------------------------------------------------------

    .asg   AR0, X_ptr              ; AR0 is pointer to input buffer – X_ptr
    .asg   AR1, H_ptr              ; AR1 is pointer to coefficients – H_ptr
    .asg   AR2, R_ptr              ; AR2 is pointer to result buffer – R_ptr
    .asg   AR3, DB_ptr             ; AR3 is pointer to delay buffer – DB_ptr

FRAME_SZ  .set 2

    .global _fir

    .text

;************************************************************************

_fir

;   Create local frame for temp values
;   -----------------------------------------------------------------------

    SP = SP – #FRAME_SZ

;   Turn on fractional mode
;   Turn on sign-extension mode
;   -----------------------------------------------------------------------


    BIT(ST1, #ST1_FRCT) = #1      ; Set fractional mode bit
    BIT(ST1, #ST1_SXMD) = #1      ; Set sign-extension mode bit


;   Set outer loop count by subtracting 1 from nx and storing into
;   block-repeat counter
;   -----------------------------------------------------------------------

    AC1 = T1 – #1                  ; AC1 = number of samples (nx) – 1
    *SP(0) = AC1                   ; Top of stack = nx – 1
    BRC0 = *SP(0)                  ; BRC0 = nx – 1 (outer loop counter)
```

Example B–23. Code That Uses Multiple CPU Units But No User-Defined Parallelism (Continued)

```
;   Store length of coefficient vector/delay buffer in BK register
;   ------------------------------------------------------------------

    BIT(ST2, #ST2_AR1LC) = #1     ; Enable AR1 circular configuration
    BSA01 = *(#0011h)             ; Set buffer (filter) start address
                                  ; AR1 used as filter pointer

    BIT(ST2, #ST2_AR3LC) = #1     ; Enable AR3 circular configuration
    *SP(1) = DB_ptr               ; Save pointer to delay buffer pointer
    AC1 = *DB_ptr                 ; AC1 = delay buffer pointer
    DB_ptr = AC1                  ; AR3 (DB_ptr) = delay buffer pointer
    BSA23 = *(#0013h)             ; Set buffer (delay buffer) start address
                                  ; AR3 used as filter pointer

    *SP(0) = T0                   ; Save filter length, nh – used as buffer
                                  ; size
    BK03 = *SP(0)                 ; Set circular buffer size – size passed
                                  ; in T0

    AC1 = T0 – #3                 ; AC1 = nh – 3
    *SP(0) = AC1
    CSR = *SP(0)                  ; Set inner loop count to nh – 3

    H_ptr = #0                    ; Initialize index of filter to 0
    DB_ptr = #0                   ; Initialize index of delay buffer to 0

;   Begin outer loop on nx samples
;   ------------------------------------------------------------------

    BLOCKREPEAT {

;   Move next input sample into delay buffer
;   ------------------------------------------------------------------

    *DB_ptr = *X_ptr+

;   Sum h * x for next y value
;   ------------------------------------------------------------------

    AC0 = *H_ptr+ * *DB_ptr+

    REPEAT (CSR)
       AC0 = AC0 + (*H_ptr+ * *DB_ptr+)

    AC0 = rnd(AC0 + (*H_ptr+ * *DB_ptr))    ; Round result
```

Example B–23. Code That Uses Multiple CPU Units But No User-Defined Parallelism (Continued)

```
;   Store result
;   -------------------------------------------------------------------

    *R_ptr+ = HI(AC0)
    }


;   Clear FRCT bit to restore normal C operating environment
;   Return overflow condition of AC0 (shown in ACOV0) in T0
;   Restore stack to previous value, FRAME, etc..
;   Update current index of delay buffer pointer
;   -------------------------------------------------------------------

END_FUNCTION:

    AR0 = *SP(1)                  ; AR0 = pointer to delay buffer pointer
    SP = SP + #FRAME_SZ           ; Remove local stack frame
    *AR0 = DB_ptr                 ; Update delay buffer pointer with current
                                  ; index

    BIT(ST1, #ST1_FRCT) = #0      ; Clear fractional mode bit

    T0 = #0                       ; Make T0 = 0 for no overflow (return value)
    if(overflow(AC0)) execute(AD_unit)
    T0 = #1                       ; Make T0 = 1 for overflow (return value)

    RETURN
;**********************************************************************
```

Example B–24. Code in Example B–23 Modified to Take Advantage of Parallelism

```
    .CPL_ON                     ; Tell assembler that CPL bit is 1
                                ; (SP direct addressing like *SP(0) is enabled)

;   Register usage
;   -----------------------------------------------------------------

    .asg   AR0, X_ptr              ; AR0 is pointer to input buffer – X_ptr
    .asg   AR1, H_ptr              ; AR1 is pointer to coefficients – H_ptr
    .asg   AR2, R_ptr              ; AR2 is pointer to result buffer – R_ptr
    .asg   AR3, DB_ptr             ; AR3 is pointer to delay buffer – DB_ptr

FRAME_SZ  .set 2

    .global _fir

    .text

;***************************************************************************

_fir

;   Create local frame for temp values
;   -----------------------------------------------------------------

    SP = SP – #FRAME_SZ            ; (Attempt to put this in parallel with
                                   ;   the following AC1 modification failed)

;   Set outer loop count by subtracting 1 from nx and storing into
;   block-repeat counter
;   Turn on fractional mode
;   Turn on sign-extension mode
;   -----------------------------------------------------------------

    AC1 = T1 – #1                  ; AC1 = number of samples (nx) – 1

    BIT(ST1, #ST1_FRCT) = #1       ; Set fractional mode bit
    || *SP(0) = AC1                ; Top of stack = nx – 1

    BRC0 = *SP(0)                  ; BRC0 = nx – 1 (outer loop counter)
    || BIT(ST1, #ST1_SXMD) = #1    ; Set sign-extension mode bit
```

Example B–24. Code in Example B–23 Modified to Take Advantage of Parallelism (Continued)

```
;   Store length of coefficient vector/delay buffer in BK register
;   -----------------------------------------------------------------

    BIT(ST2, #ST2_AR1LC) = #1     ; Enable AR1 circular configuration
    BSA01 = *(#0011h)             ; Set buffer (filter) start address
                                  ; AR1 used as filter pointer

    BIT(ST2, #ST2_AR3LC) = #1     ; Enable AR3 circular configuration
    || *SP(1) = DB_ptr            ; Save pointer to delay buffer pointer

    AC1 = *DB_ptr                 ; AC1 = delay buffer pointer
    DB_ptr = AC1                  ; AR3 (DB_ptr) = delay buffer pointer
    || *SP(0) = T0                ; Save filter length, nh – used as buffer
                                  ; size
    BSA23 = *(#0013h)             ; Set buffer (delay buffer) start address
                                  ; AR3 used as filter pointer

    BK03 = *SP(0)                 ; Set circular buffer size – size passed
                                  ; in T0

    AC1 = T0 – #3                 ; AC1 = nh – 3
    *SP(0) = AC1

    CSR = *SP(0)                  ; Set inner loop count to nh – 3
    || H_ptr = #0                 ; Initialize index of filter to 0

    DB_ptr = #0                   ; Initialize index of delay buffer to 0
                                  ; (in parallel with BLOCKREPEAT below)

;   Begin outer loop on nx samples
;   -----------------------------------------------------------------

    ||BLOCKREPEAT {

;   Move next input sample into delay buffer
;   -----------------------------------------------------------------

    *DB_ptr = *X_ptr+
```

Example B–24. Code in Example B–23 Modified to Take Advantage of Parallelism (Continued)

```
;   Sum h * x for next y value
;   ------------------------------------------------------------------

    AC0 = *H_ptr+ * *DB_ptr+
    || REPEAT (CSR)

    AC0 = AC0 + (*H_ptr+ * *DB_ptr+)

    AC0 = rnd(AC0 + (*H_ptr+ * *DB_ptr))    ; Round result

;   Store result
;   ------------------------------------------------------------------

    *R_ptr+ = HI(AC0)
     }


;   Clear FRCT bit to restore normal C operating environment
;   Return overflow condition of AC0 (shown in ACOV0) in T0
;   Restore stack to previous value, FRAME, etc..
;   Update current index of delay buffer pointer
;   ------------------------------------------------------------------

END_FUNCTION:

    AR0 = *SP(1)                ; AR0 = pointer to delay buffer pointer
    || SP = SP + #FRAME_SZ      ; Remove local stack frame

    *AR0 = DB_ptr               ; Update delay buffer pointer with current
                                ; index

    || BIT(ST1, #ST1_FRCT) = #0   ; Clear fractional mode bit

    T0 = #0                     ; Make T0 = 0 for no overflow (return value)
    || if(overflow(AC0)) execute(AD_unit)
    T0 = #1                     ; Make T0 = 1 for overflow (return value)

    || RETURN
;*********************************************************************
```

Example B–25. Nested Loops

```
   BRC0 = #(n0-1)
   BRC1 = #(n1-1)
;  ...
   localrepeat{            ; Level 0 looping (could instead be blockrepeat):
                           ; Loops n0 times
;      ...
      repeat(#(n2-1)
;      ...
      localrepeat{         ; Level 1 looping (could instead be blockrepeat):
                           ; Loops n1 times
;      ...
         repeat(#(n3-1))
;         ...
      }
;      ...
   }
```

Example B–26. Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code)

```
_cfft:

radix_2_stages:
; ...

outer_loop:
; ...
      BRC0 = T1
; ...
      BRC1 = T1
; ...
      AR4  = AR4 >> #1                   ; outer loop counter
      || if (AR5 == #0) goto no_scale    ; determine if scaling required
; ...

no_scale:

      localrepeat{

      AC0 = dbl(*AR3)                    ; Load ar,ai

      HI(AC2) = HI(*AR2) - HI(AC0),      ; tr = ar - br
      LO(AC2) = LO(*AR2) - LO(AC0)       ; ti = ai - bi

      localrepeat {
```

Note: This example shows portions of the file cfft.asm in the TI C55x DSPLIB (introduced in Chapter 8).

Example B–26. Branch-On-Auxiliary-Register-Not-Zero Construct (Shown in Complex FFT Loop Code) (Continued)

```
        HI(AC1) = HI(*AR2) + HI(AC0),           ; ar' = ar + br
        LO(AC1) = LO(*AR2) + LO(AC0)            ; ai' = ai + bi
        || dbl(*AR6)=AC2                        ; Store tr, ti

        AC2 = *AR6 * coef(*CDP+),               ; c*tr
        AC3 = *AR7 * coef(*CDP+)                ; c*ti

        dbl(*AR2+) = AC1                        ; Store ar, ai
        || AC0 = dbl(*AR3(T0))                  ; * load ar,ai

        AC3 = rnd(AC3 - (*AR6 * coef(*CDP-))),  ; bi' = c*ti - s*tr
        AC2 = rnd(AC2 + (*AR7 * coef(*CDP-)))   ; br' = c*tr + s*ti

        *AR3+  = pair(HI(AC2))                  ; Store br', bi'
        || HI(AC2) = HI(*AR2) - HI(AC0),        ; * tr = ar - br
        LO(AC2) = LO(*AR2) - LO(AC0)            ; * ti = ai - bi

        }

        HI(AC1) = HI(*AR2) + HI(AC0),           ; ar' = ar + br
        LO(AC1) = LO(*AR2) + LO(AC0)            ; ai' = ai + bi
        || dbl(*AR6)=AC2                        ; Store tr, ti

        AC2 = *AR6 * coef(*CDP+),               ; c*tr
        AC3 = *AR7 * coef(*CDP+)                ; c*ti

        dbl(*(AR2+T1)) = AC1                    ; Store ar, ai

        AC3 = rnd(AC3 - (*AR6 * coef(*CDP+))),  ; bi' = c*ti - s*tr
        AC2 = rnd(AC2 + (*AR7 * coef(*CDP+)))   ; br' = c*tr + s*ti

        *(AR3+T1) = pair(HI(AC2))               ; Store br', bi'

        }

        AR3 = AR3 << #1
        || CDP = #0                             ; rewind coefficient pointer
        T3 = T3 >> #1
        || if (AR4 != #0) goto outer_loop
```

Note: This example shows portions of the file cfft.asm in the TI C55x DSPLIB (introduced in Chapter 8).

Example B–27. 64-Bit Addition

```
;**********************************************************************
; 64–Bit Addition        Pointer assignments:
;
;     X3 X2 X1 X0        AR1 –> X3 (even address)
;   + Y3 Y2 Y1 Y0               X2
;   ––––––––––––––              X1
;     W3 W2 W1 W0               X0
;                        AR2 –> Y3 (even address)
;                               Y2
;                               Y1
;                               Y0
;                        AR3 –> W3 (even address)
;                               W2
;                               W1
;                               W0
;
;**********************************************************************

   AC0 = dbl(*AR1(#2))                    ; AC0 = X1 X0
   AC0 = AC0 + dbl(*AR2(#2))              ; AC0 = X1 X0 + Y1 Y0
   dbl(*AR3(#2)) = AC0                    ; Store W1 W0.
   AC0 = dbl (*AR1)                       ; AC0 = X3 X2
   AC0 = AC0 + uns(*AR2(#1))+ CARRY       ; AC0 = X3 X2 + 00 Y2 + CARRY
   AC0 = AC0 + (*AR2<< #16)               ; AC0 = X3 X2 + Y3 Y2 + CARRY
   dbl(*AR3) = AC0                        ; Store W3 W2.
```

Example B–28. 64-Bit Subtraction

```
;**********************************************************************
; 64–Bit Subtraction       Pointer assignments:
;
;     X3 X2 X1 X0          AR1 –> X3 (even address)
;   – Y3 Y2 Y1 Y0                 X2
;   ––––––––––––––                X1
;     W3 W2 W1 W0                 X0
;                          AR2 –> Y3 (even address)
;                                 Y2
;                                 Y1
;                                 Y0
;                          AR3 –> W3 (even address)
;                                 W2
;                                 W1
;                                 W0
;
;**********************************************************************

    AC0 = dbl(*AR1(#2))                    ; AC0 = X1 X0
    AC0 = AC0 – dbl(*AR2(#2))              ; AC0 = X1 X0 – Y1 Y0
    dbl(*AR3(#2)) = AC0                    ; Store W1 W0.
    AC0 = dbl (*AR1)                       ; AC0 = X3 X2
    AC0 = AC0 – uns(*AR2(#1)) – BORROW     ; AC0 = X3 X2 – 00 Y2 – BORROW
    AC0 = AC0 – (*AR2<< #16)               ; AC0 = X3 X2 – Y3 Y2 – BORROW
    dbl(*AR3) = AC0                        ; Store W3 W2.
```

Example B–29. 32-Bit Integer Multiplication

```
;*****************************************************************
;   This routine multiplies two 32-bit signed integers, giving a
;   64-bit result. The operands are fetched from data memory and the
;   result is written back to data memory.
;
;   Data Storage:                       Pointer Assignments:
;   X1 X0         32-bit operand        AR0 -> X1
;   Y1 Y0         32-bit operand               X0
;   W3 W2 W1 W0   64-bit product        AR1 -> Y1
;                                              Y0
;   Entry Conditions:                   AR2 -> W0
;   SXMD = 1 (sign extension on)               W1
;   SATD = 0 (no saturation)                   W2
;   FRCT = 0 (fractional mode off)             W3
;
;   RESTRICTION: The delay chain and input array must be
;   long-word aligned.
;****************************************************************

    mar(*AR0+)                              ; AR0 points to X0
    || mar(*AR1+)                           ; AR1 points to Y0
    AC0 = uns(*AR0-)*uns(*AR1)              ; ACO = X0*Y0
    *AR2+ = AC0                             ; Save W0
    AC0 = (AC0 >> #16) + ((*AR0+)*uns(*AR1-)) ; AC0 = X0*Y0>>16 + X1*Y0
    AC0 = AC0 + (uns(*AR0-)* (*AR1))        ; AC0 = X0*Y0>>16 + X1*Y0 + X0*Y1
    *AR2+ = AC0                             ; Save W1
    AC0 = (AC0 >> #16) + ((*AR0)*(*AR1))    ; AC0 = AC0>>16 + X1*Y1
    *AR2+ = AC0                             ; Save W2
    *AR2 = HI(AC0)                          ; Save W3
```

Example B–30. 32-Bit Fractional Multiplication

```
;**************************************************************************
;   This routine multiplies two Q31 signed integers, resulting in a
;   Q31 result. The operands are fetched from data memory and the
;   result is written back to data memory.
;
;   Data Storage:                        Pointer Assignments:
;   X1 X0     Q31 operand                AR0 -> X1
;   Y1 Y0     Q31 operand                       X0
;   W1 W0     Q31 product                AR1 -> Y1
;                                               Y0
;   Entry Conditions:                    AR2 -> W1 (even address)
;   SXMD = 1 (sign extension on)                W0
;   SATD = 0 (no saturation)
;   FRCT = 1 (shift result left by 1 bit)
;
;   RESTRICTION: W1 W0 is aligned such that W1 is at an even address.
;***************************************************************************
    mar(*AR0+)                           ; AR0 points to X0
    AC0 = uns(*AR0–)*(*AR1+)             ; AC0 = X0*Y1
    AC0 = AC0 + ((*AR0)* uns(*AR1–))     ; AC0 =X0*Y1 + X1*Y0
    AC0 = (AC0 >> #16) + ((*AR0)*(*AR1)) ; AC0 = AC0>>16 + X1*Y1
    dbl(*AR2) = AC0                      ; Save W1 W0
```

Example B–31. Unsigned, 16-Bit By 16-Bit Integer Division

```
;***************************************************************************
; Pointer assignments:                    ___________
;       AR0 -> Dividend          Divisor ) Dividend
;       AR1 -> Divisor
;       AR2 -> Quotient
;       AR3 -> Remainder
;
; Algorithm notes:
;       - Unsigned division, 16–bit dividend, 16–bit divisor
;       - Sign extension turned off. Dividend & divisor are positive numbers.
;       - After division, quotient in AC0(15–0), remainder in AC0(31–16)
;***************************************************************************

    bit(ST1,#ST1_SXMD) = #0    ; Clear SXMD (sign extension off)
    AC0 = *AR0                 ; Put Dividend into AC0
    repeat( #(16 – 1) )        ; Execute subc 16 times
       subc( *AR1, AC0, AC0 )  ; AR1 points to Divisor
    *AR2 = AC0                 ; Store Quotient
    *AR3 = HI(AC0)             ; Store Remainder
```

Example B–32. Unsigned, 32-Bit By 16-Bit Integer Division

```
;***************************************************************************
; Pointer assignments:                                     ___________
;      AR0 -> Dividend high half        Divisor ) Dividend
;             Dividend low half
;             ...
;      AR1 -> Divisor
;             ...
;      AR2 -> Quotient high half
;             Quotient low half
;             ...
;      AR3 -> Remainder
;
; Algorithm notes:
;      - Unsigned division, 32-bit dividend, 16-bit divisor
;      - Sign extension turned off. Dividend & divisor are positive numbers.
;      - Before 1st division: Put high half of dividend in AC0
;      - After 1st division:  High half of quotient in AC0(15-0)
;      - Before 2nd division: Put low part of dividend in AC0
;      - After 2nd division:  Low half of quotient in AC0(15-0) and
;                             Remainder in AC0(31-16)
;***************************************************************************

    bit(ST1,#ST1_SXMD) = #0       ; Clear SXMD (sign extension off)
    AC0 = *AR0+                   ; Put high half of Dividend in AC0
    || repeat( #(15 - 1) )        ; Execute subc 15 times
       subc( *AR1, AC0, AC0)      ; AR1 points to Divisor
    subc( *AR1, AC0, AC0)         ; Execute subc final time
    || AR4 = #8                   ; Load AR4 with AC0_L memory address
    *AR2+ = AC0                   ; Store high half of Quotient
    *AR4 = *AR0+                  ; Put low half of Dividend in AC0_L
    repeat( #(16 - 1) )           ; Execute subc 16 times
       subc( *AR1, AC0, AC0)
    *AR2+ = AC0                   ; Store low half of Quotient
    *AR3 = HI(AC0)                ; Store Remainder
    bit(ST1,#ST1_SXMD) = #1       ; Set SXMD (sign extension on)
```

Example B–33. Signed, 16-Bit By 16-Bit Integer Division

```
;*****************************************************************************
; Pointer assignments:                                ___________
;      AR0 -> Dividend        Divisor ) Dividend
;      AR1 -> Divisor
;      AR2 -> Quotient
;      AR3 -> Remainder
;
; Algorithm notes:
;      - Signed division, 16-bit dividend, 16-bit divisor
;      - Sign extension turned on. Dividend and divisor can be negative.
;      - Expected quotient sign saved in AC0 before division
;      - After division, quotient in AC1(15-0), remainder in AC1(31-16)
;*****************************************************************************

   bit(ST1,#ST1_SXMD) = #1       ; Set SXMD (sign extension on)
   AC0 = (*AR0) * (*AR1)         ; Sign of (Dividend x Divisor) should be
                                 ;   sign of Quotient
   AC1 = *AR1                    ; Put Divisor in AC1
   AC1 = |AC1|                   ; Find absolute value, |Divisor|
   *AR2 = AC1                    ; Store |Divisor| temporarily
   AC1 = *AR0                    ; Put Dividend in AC1
   AC1 = |AC1|                   ; Find absolute value, |Dividend|
   repeat( #(16 - 1) )           ; Execute subc 16 times
      subc( *AR2, AC1, AC1)      ; AR2 -> |Divisor|
   *AR3 = HI(AC1)                ; Save Remainder
   *AR2 = AC1                    ; Save Quotient
   AC1 = AC1 << #16              ; Shift quotient: Put MSB in sign position
   AC1 = - AC1                   ; Negate quotient
   if(AC0 < #0) execute (D_unit) ; If sign of Quotient should be negative,
   *AR2 = HI(AC1)                ;   replace Quotient with negative version
```

Example B–34. Signed, 32-Bit By 16-Bit Integer Division

```
;*****************************************************************************
;   Pointer assignments:    (Dividend and Quotient are long-word aligned)
;       AR0 -> Dividend high half (NumH) (even address)
;              Dividend low half (NumL)
;       AR1 -> Divisor (Den)
;       AR2 -> Quotient high half (QuotH) (even address)
;              Quotient low half (QuotL)
;       AR3 -> Remainder (Rem)
;
;   Algorithm notes:
;       - Signed division, 32-bit dividend, 16-bit divisor
;       - Sign extension turned on. Dividend and divisor can be negative.
;       - Expected quotient sign saved in AC0 before division
;       - Before 1st division: Put high half of dividend in AC1
;       - After 1st division:  High half of quotient in AC1(15-0)
;       - Before 2nd division: Put low part of dividend in AC1
;       - After 2nd division:  Low half of quotient in AC1(15-0) and
;                              Remainder in AC1(31-16)
;*****************************************************************************

   bit(ST1,#ST1_SXMD) = #1    ; Set SXMD (sign extension on)
   AC0 = (*AR0)* (*AR1)       ; Sign( NumH x Den ) is sign of actual result
   AC1 = *AR1                 ; AC1 = Den
   AC1 = |AC1|                ; AC1 = abs(Den)
   *AR3 = AC1                 ; Rem = abs(Den) temporarily
   AC1 = dbl(*AR0)            ; AC1 = NumH NumL
   AC1 = |AC1|                ; AC1 = abs(Num)
   dbl(*AR2) = AC1            ; QuotH = abs(NumH) temporarily
                              ; QuotL = abs(NumL) temporarily

   AC1 = *AR2                 ; AC1 = QuotH
   repeat( #(15 - 1) )        ; Execute subc 15 times
     subc( *AR3, AC1, AC1)
   subc( *AR3, AC1, AC1)      ; Execute subc final time
   ||AR4 = #11                ; Load AR4 with AC1_L memory address
   *AR2+ = AC1                ; Save QuotH
   *AR4 = *AR2                ; AC1_L = QuotH
   repeat( #(16 - 1) )        ; Execute subc 16 times
     subc( *AR3, AC1, AC1)
   *AR2- = AC1                ; Save QuotL
   *AR3 = HI(AC1)             ; Save Rem

   if (AC0 >= #0) goto skip   ; If actual result should be positive, goto skip.
   AC1  = dbl(*AR2)           ; Otherwise, negate Quotient.
   AC1 = - AC1
   dbl(*AR2) = AC1

skip:
   return
```

Example B–35. Off-Place Bit Reversing of a Vector Array (in Assembly)

```
;...
 bit (ST2, #ST2_ARMS) = #0 ; reset ARMS bit to allow bit-reverse addressing
 .arms_off                 ; notify the assembler of ARMS bit = 0
;...
off_place:
 localrepeat{
   AC0 = dbl(*AR0+)        ; AR0 points to input array
   dbl(*(AR1+T0B)) = AC0   ; AR1 points to output array
                           ; T0 = NX = number of complex elements in
                           ; array pointed to by AR0
   }
```

Note: This example shows portions of the file cbrev.asm in the TI C55x DSPLIB (introduced in Chapter 8).

Example B–36. Delayed LMS Implementation of an Adaptive Filter

```
;   ar_data: index in the delay buffer
;   ar_input: pointer to input vector
;   ar_coef: pointer to coefficient vector

StartSample:

; Clear AC0 for initial error term
    AC1 = #0
    || localrepeat {
    *ar_data+ = *ar_input+     ;copy input -> state(0)

; Place error term in T3
    T3 = HI(AC1)

; Place first update term in AC0
;...while clearing FIR value
    AC0 = T3 * *ar_data+
    || AC1 = #0

;AC0 = update coef
;AC1 = start of FIR output
    LMS(*ar_coef, *ar_data, AC0, AC1)
    || localrepeat {
    *ar_coef+ = HI(AC0)
    || AC0 = T3 * *ar_data+

;AC0 = update coef
;AC1 = update of FIR output
      LMS(*ar_coef, *ar_data, AC0, AC1)
     }

; Store Calculated Output
    *ar_coef+ = HI(AC0)
    || *ar_output+ = HI(rnd(AC1))

; AC2 is error amount
; Point to oldest data sample
    AC2 = (*ar_des+ << #16) - AC1
    || mar(*ar_data+)

; Place updated mu_error term in AC1
    AC1 = rnd(T_step*AC2)
    }
```

Note: This example shows portions of the file dlms.asm in the TI C55x DSPLIB (introduced in Chapter 8).

Example B–37. Generation of Output Streams G_0 and G_1

```
    AR3 = #09H                        ; AC0_H
    AR1 = #in_bit_stream


    ; Load 32 bits into the accumulators
    AC0 = *AR1+
    AC0 = AC0 + (*AR1 << #16)
    AC1 = AC0

    ; Generate G0
    AC0 = AC0 ^ (AC1 << #-1)          ; A = A XOR B>>1
    AC0 = AC0 ^ (AC1 << #-3)          ; A = A XOR B>>2
    T0 = AC0                          ; Save G0

    ; Generate G1
    AC0 = AC0 ^ (AC1 << #-1)          ; A = A XOR B>>1
    AC0 = AC0 ^ (AC1 << #-3)          ; A = A XOR B>>3
    AC0 = AC0 ^ (AC1 << #-4)          ; A = A XOR B>>4 --> AC0_L = G1

    *AR3 = T0                         ; AC0_H = G0 ------> AC0 = G0G1
```

Example B–38. Viterbi Butterflies for Channel Coding

```
BFLY_DIR_ALG .MACRO
;new_metric(j)&(j+2^(K-2))
   hi(AC0) = *AR5+ + T3,                ; AC0(39-16) = Old_Met(2*j)+LD
   lo(AC0) = *AR5+ - T3                 ; AC0(15-0) = Old_met(2*j+1)-LD

   hi(AC1) = *AR5+ - T3,                ; AC1(39-16) = Old_Met(2*j)-LD
   lo(AC1) = *AR5+ + T3                 ; AC1(15-0) = Old_met(2*j+1)+LD

   max_diff(AC0, AC1, AC2, AC3)         ; Compare AC0 and AC1
   *AR3+ = lo(AC2), *AR4+ = hi(AC2)     ; Store the lower maxima (with AR3)
                                        ; and upper maxima (with AR4)
   .ENDM

BFLY_REV_ALG        .MACRO
;new_metric(j)&(j+2^(K-2))
   hi(AC0) = *AR5+ - T3,                ; AC0(39-16) = Old_Met(2*j)-LD
   lo(AC0) = *AR5+ + T3                 ; AC0(15-0) = Old_met(2*j+1)+LD

   hi(AC1) = *AR5+ + T3,                ; AC1(39-16) = Old_Met(2*j)+LD
   lo(AC1) = *AR5+ - T3                 ; AC1(15-0) = Old_met(2*j+1)-LD

   max_diff(AC0, AC1, AC2, AC3)         ; Compare AC0 and AC1
   *AR3+ = lo(AC2), *AR4+ = hi(AC2)     ; Store the lower maxima (with AR3)
                                        ; and upper maxima (with AR4)
   .ENDM
```

Example B–39. Viterbi Butterflies Using Instruction Parallelism

```
    localrepeat {
butterfly:
    hi(AC0) = *AR0+ + T3,                ; AC0(39–16) = Old_Met(2*j)+LD
    lo(AC0) = *AR0+ – T3                 ; AC0(15–0) = Old_met(2*j+1)–LD
    || AR7 = *AR5+

    hi(AC1) = *AR0+ – T3,                ; AC1(39–16) = Old_Met(2*j)–LD
    lo(AC1) = *AR0+ + T3                 ; AC1(15–0) = Old_met(2*j+1)+LD
    || T3 = *AR6                         ; Load new local distance

    *AR2+ = lo(AC2), *AR2(T0) = hi(AC2)  ; Store lower and upper maxima
                                         ; from previous max_diff operation
    || max_diff( AC0, AC1, AC2, AC3)     ; Compare AC0 and AC1

    hi(AC0) = *AR0+ + T3,                ; AC0(39–16) = Old_Met(2*j)+LD
    lo(AC0) = *AR0+ – T3                 ; AC0(15–0) = Old_met(2*j+1)–LD
    || AR6 = *AR5+

    hi(AC1) = *AR0+ – T3,                ; AC1(39–16) = Old_Met(2*j)–LD
    lo(AC1) = *AR0+ + T3                 ; AC1(15–0) = Old_met(2*j+1)+LD
    || T3 = *AR7                         ; Load new local distance

    *AR2(T0) = lo(AC2), *AR2+ = hi(AC2)  ; Store lower and upper maxima
                                         ; from previous max_diff operation
    || max_diff( AC0, AC1, AC2, AC3)     ; Compare AC0 and AC1
    }
```

Index