TMS320C62x/C67x

Programmer's Guide

1998

Digital Signal Processing Solutions

Printed in U.S.A., February 1998
SDS

SPRU198B

Programmer's Guide

TMS320C62x/C67x

1998

TMS320C62x/C67x Programmer's Guide

Literature Number: SPRU198B
February 1998

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

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

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1998, Texas Instruments Incorporated

Preface

Read This First

About This Manual

This manual is a reference for programming TMS320C6x digital signal processor (DSP) devices.

Before you use this book, you should install your code generation and debugging tools.

This book is organized in four major parts:

- **Part I: Introduction** includes a brief description of the 'C6x architecture and code development flow. It also includes a tutorial that introduces you to the tools you will use in each phase of development and an optimization checklist to help you achieve optimal performance from your code.
- **Part II: C Code** includes C code examples and discusses optimization methods for the code. This information can help you choose the most appropriate optimization techniques for your code.
- **Part III: Assembly Code** describes the structure of assembly code. It provides examples and discusses optimizations for assembly code. It also includes a chapter on interrupt subroutines.
- **Part IV: Appendix** provides extensive code examples from the GSM EFR vocoder.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x 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.

TMS320C6x Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C6x generation of devices.

TMS320C6x Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6x C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6x generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C6x C Source Debugger User's Guide (literature number SPRU188) tells you how to invoke the 'C6x simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

TMS320C62x/C67x CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62x/C67x CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320 DSP Designer's Notebook: Volume 1 (literature number SPRT125) presents solutions to common design problems using 'C2x, 'C3x, 'C4x, 'C5x, and other TI DSPs.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6201 and provides pinouts, electrical specifications, and timings for the device.

Trademarks

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

VelociTI is a trademark of Texas Instruments Incorporated.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32
Email: epic@ti.com		
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

1 Introduction **1-1**

Introduces some features of the 'C6x microprocessor and discusses the basic process for creating code.

1.1 TMS320C6x Architecture 1-2
1.2 TMS320C6x Pipeline 1-2
1.3 Code Development Flow to Increase Performance 1-3

2 Code Development Flow Tutorial **2-1**

Uses example code to walk you through the code development flow for the TMS320C6x.

2.1 Before You Begin 2-2
2.2 Introduction to the Example Code 2-3
2.3 Lesson 1: Compiling, Assembling, and Linking the Example Code 2-5
2.4 Lesson 2: Profiling the Example Code 2-8
2.5 Lesson 3: Phase 1 of the Code Development Flow 2-14
2.6 Lesson 4: Phase 2 of the Code Development Flow 2-17
2.7 Lesson 5: Phase 3 of the Code Development Flow 2-25
2.8 Summary 2-31

3 TMS320C6x Optimization Checklist **3-1**

Provides a code development flow and checklist for optimizing loops.

4 Optimizing C Code **4-1**

Explains how to maximize C performance by using compiler options, intrinsics, and code transformations.

4.1 Writing C Code 4-2
4.1.1 Tips on Data Types 4-2
4.1.2 Analyzing C Code Performance 4-2
4.2 Compiling C Code 4-4
4.2.1 Compiler Options 4-4
4.2.2 Memory Dependencies 4-5
4.3 Refining C Code 4-9
4.3.1 Using Intrinsics 4-9
4.3.2 Using Word Access for Short Data 4-14
4.3.3 Software Pipelining 4-20

5 Structure of Assembly Code 5-1

Describes the structure of the assembly code, including labels, conditions, instructions, functional units, operands, and comments.

5.1 Labels 5-2
5.2 Parallel Bars 5-2
5.3 Conditions 5-3
5.4 Instructions 5-4
5.5 Functional Units 5-6
5.6 Operands 5-8
5.7 Comments 5-9

6 Optimizing Assembly Code via Linear Assembly 6-1

Describes methods that help you develop more efficient assembly language programs.

6.1 Assembly Code 6-2
6.2 Writing Parallel Code 6-4
6.2.1 Dot Product C Code 6-4
6.2.2 Translating C Code to Linear Assembly 6-5
6.2.3 Linear Assembly Resource Allocation 6-6
6.2.4 Drawing a Dependency Graph 6-6
6.2.5 Nonparallel Versus Parallel Assembly Code 6-10
6.2.6 Comparing Performance 6-14
6.3 Using Word Access for Short Data and Doubleword Access for Floating-Point Data 6-15
6.3.1 Unrolled Dot Product C Code 6-15
6.3.2 Translating C Code to Linear Assembly 6-16
6.3.3 Drawing a Dependency Graph 6-18
6.3.4 Linear Assembly Resource Allocation 6-19
6.3.5 Final Assembly 6-22
6.3.6 Comparing Performance 6-24
6.4 Software Pipelining 6-25
6.4.1 Modulo Iteration Interval Scheduling 6-28
6.4.2 Using the Assembly Optimizer to Create Optimized Loops 6-35
6.4.3 Final Assembly 6-36
6.4.4 Comparing Performance 6-53
6.5 Modulo Scheduling of Multicycle Loops 6-54
6.5.1 Weighted Vector Sum C Code 6-54
6.5.2 Translating C Code to Linear Assembly 6-54
6.5.3 Determining the Minimum Iteration Interval 6-55
6.5.4 Drawing a Dependency Graph 6-57
6.5.5 Linear Assembly Resource Allocation 6-58
6.5.6 Modulo Iteration Interval Scheduling 6-58
6.5.7 Using the Assembly Optimizer for the Weighted Vector Sum 6-69
6.5.8 Final Assembly 6-70

6.6 Loop Carry Paths 6-73
6.6.1 IIR Filter C Code 6-73
6.6.2 Translating C Code to Linear Assembly (Inner Loop) 6-74
6.6.3 Drawing a Dependency Graph 6-75
6.6.4 Determining the Minimum Iteration Interval 6-76
6.6.5 Linear Assembly Resource Allocation 6-78
6.6.6 Modulo Iteration Interval Scheduling 6-79
6.6.7 Using the Assembly Optimizer for the IIR Filter 6-80
6.6.8 Final Assembly 6-81
6.7 If-Then-Else Statements in a Loop 6-82
6.7.1 If-Then-Else C Code 6-82
6.7.2 Translating C Code to Linear Assembly 6-83
6.7.3 Drawing a Dependency Graph 6-84
6.7.4 Determining the Minimum Iteration Interval 6-85
6.7.5 Linear Assembly Resource Allocation 6-86
6.7.6 Final Assembly 6-87
6.7.7 Comparing Performance 6-88
6.8 Loop Unrolling 6-90
6.8.1 Unrolled If-Then-Else C Code 6-90
6.8.2 Translating C Code to Linear Assembly 6-91
6.8.3 Drawing a Dependency Graph 6-92
6.8.4 Determining the Minimum Iteration Interval 6-93
6.8.5 Linear Assembly Resource Allocation 6-93
6.8.6 Final Assembly 6-95
6.8.7 Comparing Performance 6-96
6.9 Live-Too-Long Issues 6-97
6.9.1 C Code With Live-Too-Long Problem 6-97
6.9.2 Translating C Code to Linear Assembly 6-98
6.9.3 Drawing a Dependency Graph 6-98
6.9.4 Determining the Minimum Iteration Interval 6-100
6.9.5 Linear Assembly Resource Allocation 6-102
6.9.6 Final Assembly With Move Instructions 6-104
6.10 Redundant Load Elimination 6-106
6.10.1 FIR Filter C Code 6-106
6.10.2 Translating C Code to Linear Assembly 6-108
6.10.3 Drawing a Dependency Graph 6-109
6.10.4 Determining the Minimum Iteration Interval 6-110
6.10.5 Linear Assembly Resource Allocation 6-110
6.10.6 Final Assembly 6-111
6.11 Memory Banks 6-114
6.11.1 FIR Filter Inner Loop 6-116
6.11.2 Unrolled FIR Filter C Code 6-118
6.11.3 Translating C Code to Linear Assembly 6-119
6.11.4 Drawing a Dependency Graph 6-120

6.11.5 Linear Assembly for Unrolled FIR Inner Loop With .mptr Directive 6-121
6.11.6 Linear Assembly Resource Allocation 6-123
6.11.7 Determining the Minimum Iteration Interval 6-124
6.11.8 Final Assembly 6-124
6.11.9 Comparing Performance 6-124
6.12 Software Pipelining the Outer Loop 6-127
6.12.1 Unrolled FIR Filter C Code 6-127
6.12.2 Making the Outer Loop Parallel With the Inner Loop Epilog and Prolog . . . 6-128
6.12.3 Final Assembly 6-128
6.12.4 Comparing Performance 6-131
6.13 Outer Loop Conditionally Executed With Inner Loop 6-132
6.13.1 Unrolled FIR Filter C Code 6-132
6.13.2 Translating C Code to Linear Assembly (Inner Loop) 6-133
6.13.3 Translating C Code to Linear Assembly (Outer Loop) 6-134
6.13.4 Unrolled FIR Filter C Code 6-134
6.13.5 Translating C Code to Linear Assembly (Inner Loop) 6-136
6.13.6 Translating C Code to Linear Assembly (Inner Loop and Outer Loop) 6-138
6.13.7 Determining the Minimum Iteration Interval 6-142
6.13.8 Final Assembly 6-142
6.13.9 Comparing Performance 6-145

7 Interrupts 7-1

Describes interrupts from a software programming point of view.

7.1 Overview of Interrupts 7-2
7.2 Single Assignment vs. Multiple Assignment 7-3
7.3 Interruptible Loops 7-5
7.4 Interruptible Code Generation 7-6
7.4.1 Level 0 – Specified Code is Guaranteed to Not Be Interrupted 7-6
7.4.2 Level 1 – Specified Code Interruptible at All Times 7-7
7.4.3 Level 2 – Specified Code Interruptible Within Threshold Cycles 7-7
7.5 Interrupt Subroutines 7-8
7.5.1 ISR with the C Compiler 7-8
7.5.2 ISR with Hand-Coded Assembly 7-9
7.5.3 Nested Interrupts 7-9

A Applications Programming A-1

Provides extensive code examples from the GSM EFR vocoder.

A.1 Summary of Major Programming Methods A-2
A.2 Implementation of the GSM EFR Vocoder A-3
A.2.1 Implementation of the Multiply-Accumulate Loop A-4
A.2.2 Implementation of the Windowing and Scaling Part of autocorr.c A-7
A.2.3 Implementation of cor_h A-20
A.2.4 Implementation of the rrv Computation in search_10i40 . . . . A-27
A.2.5 Implementation of the Index Search in search_10i40 . . . . A-38

A.2.6 Implementation of the FIR Filter, residu.c, in GSM EFR Vocoder A-51
A.2.7 Implementation of the Lag Search in the lag_max ( ) Routine A-56

Figures

4–1 Dependency Graph for Vector Sum #1 4-6
4–2 Dependency Graph for Vector Sum #2 4-7
4–3 Software-Pipelined Loop 4-20
5–1 Labels in Assembly Code 5-2
5–2 Parallel Bars in Assembly Code 5-2
5–3 Conditions in Assembly Code 5-3
5–4 Instructions in Assembly Code 5-4
5–5 TMS320C6x Functional Units 5-6
5–6 Units in the Assembly Code 5-7
5–7 Operands in the Assembly Code 5-8
5–8 Operands in Instructions 5-8
5–9 Comments in Assembly Code 5-9
6–1 Dependency Graph of Fixed-Point Dot Product 6-7
6–2 Dependency Graph of Floating-Point Dot Product 6-8
6–3 Dependency Graph of Fixed-Point Dot Product with Parallel Assembly 6-11
6–4 Dependency Graph of Floating-Point Dot Product with Parallel Assembly 6-13
6–5 Dependency Graph of Fixed-Point Dot Product With LDW 6-18
6–6 Dependency Graph of Floating-Point Dot Product With LDDW 6-19
6–7 Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units) 6-20
6–8 Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units) 6-21
6–9 Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units) 6-26
6–10 Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units) 6-27
6–11 Dependency Graph of Weighted Vector Sum 6-57
6–12 Dependency Graph of Weighted Vector Sum (Showing Resource Conflict) 6-61
6–13 Dependency Graph of Weighted Vector Sum (With Resource Conflict Resolved) 6-64
6–14 Dependency Graph of Weighted Vector Sum (Scheduling ci +1) 6-66
6–15 Dependency Graph of IIR Filter 6-75
6–16 Dependency Graph of IIR Filter (With Smaller Loop Carry) 6-77
6–17 Dependency Graph of If-Then-Else Code 6-84
6–18 Dependency Graph of If-Then-Else Code (Unrolled) 6-92
6–19 Dependency Graph of Live-Too-Long Code 6-99
6–20 Dependency Graph of Live-Too-Long Code (Split-Join Path Resolved) 6-102

6–21 Dependency Graph of FIR Filter (With Redundant Load Elimination) 6-109
6–22 4-Bank Interleaved Memory 6-114
6–23 4-Bank Interleaved Memory With Two Memory Blocks 6-115
6–24 Dependency Graph of FIR Filter (With Even and Odd Elements of Each Array on Same Loop Cycle) 6-117
6–25 Dependency Graph of FIR Filter (With No Memory Hits) 6-120
A–1 Flow Diagram for the Windowing and Scaling Part of autocorr.c A-9
A–2 Flow Diagram for autocorr.c With Loop Unrolling A-12
A–3 Flow Diagram for autocorr.c With Rearranged C Code A-13

Tables

2–1 Using the C_OPTIONS Environment Variable . . . 2-7
2–2 Cycle Counts . . . 2-11
2–3 Revised Cycle Counts for vec_mpy( ) . . . 2-22
2–4 Revised Cycle Counts for iir( ) . . . 2-23
2–5 Revised Cycle Counts . . . 2-24
2–6 Revised Cycle Counts for iir( ) . . . 2-29
2–7 Revised Cycle Counts . . . 2-30
3–1 Code Development Steps . . . 3-2
3–2 TMS320C6x Optimization Checklist . . . 3-4
4–1 Subset of Compiler Options . . . 4-4
4–2 TMS320C6x C Compiler Intrinsics . . . 4-10
5–1 Selected TMS320C6x Directives . . . 5-4
5–2 Selected TMS320C6x Instruction Mnemonics . . . 5-5
5–3 Functional Units and Descriptions . . . 5-7
6–1 Comparison of Nonparallel and Parallel Assembly Code for Fixed-Point Dot Product . . . 6-14
6–2 Comparison of Nonparallel and Parallel Assembly Code for Floating-Point Dot Product . . . 6-14
6–3 Comparison of Fixed-Point Dot Product Code With Use of LDW . . . 6-24
6–4 Comparison of Floating-Point Dot Product Code With Use of LDDW . . . 6-24
6–5 Modulo Iteration Interval Scheduling Table for Fixed-Point Dot Product (Before Software Pipelining) . . . 6-28
6–6 Modulo Iteration Interval Scheduling Table for Floating-Point Dot Product (Before Software Pipelining) . . . 6-29
6–7 Modulo Iteration Interval Table for Fixed-Point Dot Product (After Software Pipelining) . . . 6-31
6–8 Modulo Iteration Interval Table for Floating-Point Dot Product (After Software Pipelining) . . . 6-32
6–9 Software Pipeline Accumulation Staggered Results Due to Three-Cycle Delay . . . 6-34
6–10 Comparison of Fixed-Point Dot Product Code Examples . . . 6-53
6–11 Comparison of Floating-Point Dot Product Code Examples . . . 6-53
6–12 Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop) . . . 6-60
6–13 Modulo Iteration Interval Table for Weighted Vector Sum With SHR Instructions . . . 6-62
6–14 Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop) . . . 6-65
6–15 Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop) . . . 6-68
6–16 Resource Table for IIR Filter . . . 6-76
6–17 Modulo Iteration Interval Table for IIR (4-Cycle Loop) . . . 6-79
6–18 Resource Table for If-Then-Else Code . . . 6-85

6–19 Comparison of If-Then-Else Code Examples 6-89
6–20 Resource Table for Unrolled If-Then-Else Code 6-93
6–21 Comparison of If-Then-Else Code Examples 6-96
6–22 Resource Table for Live-Too-Long Code 6-100
6–23 Resource Table for FIR Filter Code 6-110
6–24 Resource Table for FIR Filter Code 6-124
6–25 Comparison of FIR Filter Code 6-124
6–26 Comparison of FIR Filter Code 6-131
6–27 Resource Table for FIR Filter Code 6-142
6–28 Comparison of FIR Filter Code 6-145

Examples

2–1 The Code Example—demo1.c 2-3
2–2 The Multiply Accumulate Function—mac1.c 2-3
2–3 The Vector Multiply Function—vec_mpy1.c 2-4
2–4 The Biquad Filter—iir1.c 2-4
2–5 Including the clock( ) Function in demo1.c (count.c) 2-12
2–6 Inner Loop Kernel of mac1.asm 2-14
2–7 Inner Loop Kernel of vec_mpy1.asm 2-15
2–8 Inner Loop Kernel of iir1.asm 2-16
2–9 The Vector Multiply Function—vec_mpy1.c 2-17
2–10 Inner Loop Kernel of vec_mpy1.asm 2-17
2–11 The Revised Vector Multiply Function—vec_mpy2.c 2-18
2–12 The Biquad Filter—iir1.c 2-19
2–13 The Revised Biquad Filter—iir2.c 2-20
2–14 The Revised Example—demo2.c 2-21
2–15 Inner Loop Kernel of vec_mpy2.asm 2-22
2–16 Inner Loop Kernel of iir2.asm 2-23
2–17 The Revised Biquad Filter—iir2.c 2-26
2–18 The Biquad Filter, Revised and Assembly-Optimized—iir3.sa 2-27
2–19 The Revised Example—demo3.c 2-28
2–20 Inner Loop Kernel of iir3.asm 2-29
3–1 Compiler and/or Assembly Optimizer Feedback 3-3
4–1 Basic Vector Sum 4-5
4–2 Vector Sum With const Keywords 4-7
4–3 Compiler Output for Vector Sum Code 4-8
4–4 Saturated Add Without Intrinsics 4-9
4–5 Saturated Add With Intrinsics 4-10
4–6 Vector Sum With const Keywords, _nassert, Word Reads 4-14
4–7 Vector Sum With const Keywords, _nassert, Word Reads (Generic Version) 4-15
4–8 Dot Product Using Intrinsics 4-16
4–9 FIR Filter—Original Form 4-16
4–10 FIR Filter—Optimized Form 4-17
4–11 Basic Float Dot Product 4-18
4–12 Float Dot Product Using Intrinsics 4-18
4–13 Float Dot Product With Peak Performance 4-19
4–14 Trip Counters 4-21
4–15 Vector Sum With const Keywords and _nassert 4-22

4–16 Vector Sum With Three Memory Operations 4-23
4–17 Word-Aligned Vector Sum 4-23
4–18 Vector Sum Using const Keywords, _nassert, Word Reads, and Loop Unrolling 4-24
4–19 FIR_Type2—Original Form 4-25
4–20 FIR_Type2—Inner Loop Completely Unrolled 4-26
6–1 Fixed-Point Dot Product C Code 6-4
6–2 Floating-Point Dot Product C Code 6-4
6–3 List of Assembly Instructions for Fixed-Point Dot Product 6-5
6–4 List of Assembly Instructions for Floating-Point Dot Product 6-5
6–5 Nonparallel Assembly Code for Fixed-Point Dot Product 6-10
6–6 Parallel Assembly Code for Fixed-Point Dot Product 6-11
6–7 Nonparallel Assembly Code for Floating-Point Dot Product 6-12
6–8 Parallel Assembly Code for Floating-Point Dot Product 6-13
6–9 Fixed-Point Dot Product C Code (Unrolled) 6-15
6–10 Floating-Point Dot Product C Code (Unrolled) 6-16
6–11 Linear Assembly for Fixed-Point Dot Product Inner Loop with LDW 6-16
6–12 Linear Assembly for Floating-Point Dot Product Inner Loop with LDDW 6-17
6–13 Linear Assembly for Fixed-Point Dot Product Inner Loop With LDW (With Allocated Resources) 6-20
6–14 Linear Assembly for Floating-Point Dot Product Inner Loop With LDDW (With Allocated Resources) 6-21
6–15 Assembly Code for Fixed-Point Dot Product With LDW (Before Software Pipelining) 6-22
6–16 Assembly Code for Floating-Point Dot Product With LDDW (Before Software Pipelining) 6-23
6–17 Linear Assembly for Fixed-Point Dot Product Inner Loop (With Conditional SUB Instruction) 6-26
6–18 Linear Assembly for Floating-Point Dot Product Inner Loop (With Conditional SUB Instruction) 6-27
6–19 Pseudo-Code for Single-Cycle Accumulator With ADDSP 6-33
6–20 Linear Assembly for Full Fixed-Point Dot Product 6-35
6–21 Linear Assembly for Full Floating-Point Dot Product 6-36
6–22 Assembly Code for Fixed-Point Dot Product (Software Pipelined) 6-38
6–23 Assembly Code for Floating-Point Dot Product (Software Pipelined) 6-39
6–24 Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads) 6-42
6–25 Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads) 6-44
6–26 Assembly Code for Fixed-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog) 6-48
6–27 Assembly Code for Floating-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog) 6-49
6–28 Assembly Code for Fixed-Point Dot Product (Software Pipelined With Smallest Code Size) 6-51
6–29 Assembly Code for Floating-Point Dot Product (Software Pipelined With Smallest Code Size) 6-52

6–30 Weighted Vector Sum C Code 6-54
6–31 Linear Assembly for Weighted Vector Sum Inner Loop 6-54
6–32 Weighted Vector Sum C Code (Unrolled) 6-55
6–33 Linear Assembly for Weighted Vector Sum Using LDW 6-56
6–34 Linear Assembly for Weighted Vector Sum With Resources Allocated 6-58
6–35 Linear Assembly for Weighted Vector Sum 6-69
6–36 Assembly Code for Weighted Vector Sum 6-71
6–37 IIR Filter C Code 6-73
6–38 Linear Assembly for IIR Inner Loop 6-74
6–39 Linear Assembly for IIR Inner Loop With Reduced Loop Carry Path 6-78
6–40 Linear Assembly for IIR Inner Loop (With Allocated Resources) 6-78
6–41 Linear Assembly for IIR Filter 6-80
6–42 Assembly Code for IIR Filter 6-81
6–43 If-Then-Else C Code 6-82
6–44 Linear Assembly for If-Then-Else Inner Loop 6-83
6–45 Linear Assembly for Full If-Then-Else Code 6-86
6–46 Assembly Code for If-Then-Else 6-87
6–47 Assembly Code for If-Then-Else With Loop Count Greater Than 3 6-88
6–48 If-Then-Else C Code (Unrolled) 6-90
6–49 Linear Assembly for Unrolled If-Then-Else Inner Loop 6-91
6–50 Linear Assembly for Full Unrolled If-Then-Else Code 6-94
6–51 Assembly Code for Unrolled If-Then-Else 6-95
6–52 Live-Too-Long C Code 6-97
6–53 Linear Assembly for Live-Too-Long Inner Loop 6-98
6–54 Linear Assembly for Full Live-Too-Long Code 6-103
6–55 Assembly Code for Live-Too-Long With Move Instructions 6-104
6–56 FIR Filter C Code 6-106
6–57 FIR Filter C Code With Redundant Load Elimination 6-107
6–58 Linear Assembly for FIR Inner Loop 6-108
6–59 Linear Assembly for Full FIR Code 6-110
6–60 Final Assembly Code for FIR Filter With Redundant Load Elimination 6-112
6–61 Final Assembly Code for Inner Loop of FIR Filter 6-116
6–62 FIR Filter C Code (Unrolled) 6-118
6–63 Linear Assembly for Unrolled FIR Inner Loop 6-119
6–64 Linear Assembly for Full Unrolled FIR Filter 6-121
6–65 Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits 6-125
6–66 Unrolled FIR Filter C Code 6-127
6–67 Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits With Outer Loop Software-Pipelined 6-129
6–68 Unrolled FIR Filter C Code 6-132
6–69 Linear Assembly for Unrolled FIR Inner Loop 6-133
6–70 Linear Assembly for FIR Outer Loop 6-134
6–71 Unrolled FIR Filter C Code 6-135

6–72 Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop 6-137
6–73 Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (With Functional Units) 6-139
6–74 Final Assembly Code for FIR Filter 6-143
7–1 Code With Multiple Assignment of A1 7-3
7–2 Code Using Single Assignment 7-4
7–3 Hand-Coded Assembly ISR 7-9
7–4 Hand-Coded Assembly ISR Allowing Nesting of Interrupts 7-10
A–1 C Code for the Typical MAC Loop A-4
A–2 Linear Assembly for the MAC Loop A-4
A–3 C Code for MAC Loop With Loop Unrolling A-5
A–4 C Code for Energy Computation MAC Loop A-5
A–5 Linear Assembly for Energy Computation MAC Loop A-5
A–6 Assembly Code for the Energy Computation MAC Loop A-6
A–7 C Code for the Windowing and Scaling Part of autocorr.c A-8
A–8 Linear Assembly for One Iteration of autocorr.c (Loop 1) A-9
A–9 Linear Assembly for Loop 1 of autocorr.c (Using LDW) A-10
A–10 Linear Assembly for Loop 2 of autocorr.c (No Loop Unrolling) A-10
A–11 Linear Assembly for Loop 2 of autocorr.c (With Loop Unrolling) A-11
A–12 Linear Assembly for Loop 3 of autocorr.c A-11
A–13 Linear Assembly for Loop I of autocorr.c (Modified) A-14
A–14 Linear Assembly for Loop II of autocorr.c (Modified) A-15
A–15 Implemented C Code for autocorr.c A-16
A–16 Assembly Code for Windowing and Scaling Part of autocorr.c A-17
A–17 C Code for cor_h A-20
A–18 Linear Assembly for cor_h (One Inner Loop Iteration) A-21
A–19 C Code for cor_h (With Inner Loop Unrolling) A-22
A–20 Linear Assembly for cor_h (With Inner Loop Unrolling) A-23
A–21 Assembly Code for cor_h With Reduced Code Size A-24
A–22 C Code for the rrv Computation in search_10i40 . . . . . A-27
A–23 Linear Assembly for the rrv Computation in Search_10i40 (One Loop Iteration) A-28
A–24 C Code for the rrv Computation in search_10i40 (Unrolled Loop) A-30
A–25 Linear Assembly for rrv Computation in search_10i40 (One Loop Iteration) A-31
A–26 Assembly Code for the rrv Computation in search_10i40 . . . . . A-33
A–27 C Code for the Index Search for search_10i40 . . . . . A-38
A–28 Linear Assembly for the Index Search for search_10i40 (Inner Loop) A-41
A–29 Modified C Code for the Index Search A-42
A–30 Assembly Code for the search_10i40 Index Search A-44
A–31 C Code for residu.c A-51
A–32 C Code for residu.c After Rearrangement Using Intrinsics A-52
A–33 Implemented C Code for residu.c A-53
A–34 Assembly Code for residu.c A-54
A–35 C Code for the Lag Search in lag_max() A-57

A–36 C Code for the Lag Search in lag_max ( ) (Comparison Order Changed) A-58
A–37 C Code for the Lag Search in lag_max() With Outer Loop Unrolling A-59
A–38 Linear Assembly for the Lag Search in lag_max() Inner Loop A-59
A–39 C Code for the Lag Search in lag_max() With Inner and Outer Loops Unrolled A-60
A–40 Linear Assembly for the Lag Search in lag_max() Inner Loop A-61
A–41 Assembly Code for the Lag Search in lag_max() A-62

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Chapter 1

Introduction

This chapter introduces some features of the ’C6x microprocessor and discusses the basic process for creating code. Any reference to ’C6x pertains to both the ’C62x (fixed-point) and the ’C67x (floating-point) devices. All techniques are applicable to both devices, even though most of the examples shown are fixed-point specific.

Topic **Page**

1.1 TMS320C6x Architecture 1-2

1.2 TMS320C6x Pipeline 1-2

1.3 Code Development Flow to Increase Performance 1-3

1.1 TMS320C6x Architecture

The ’C62x is a fixed-point digital signal processor (DSP) and is the first DSP to use the VelociTI™ architecture. VelociTI is a high-performance, advanced very-long-instruction-word (VLIW) architecture, making it an excellent choice for multichannel, multifunction, and performance-driven applications.

The ’C67x is a floating-point DSP with the same features. It is the second DSP to use the VelociTI™ architecture.

The ’C6x DSPs are based on the ’C6x CPU, which consists of:

- Program fetch unit
- Instruction dispatch unit
- Instruction decode unit
- Two data paths, each with four functional units
- Thirty-two 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic

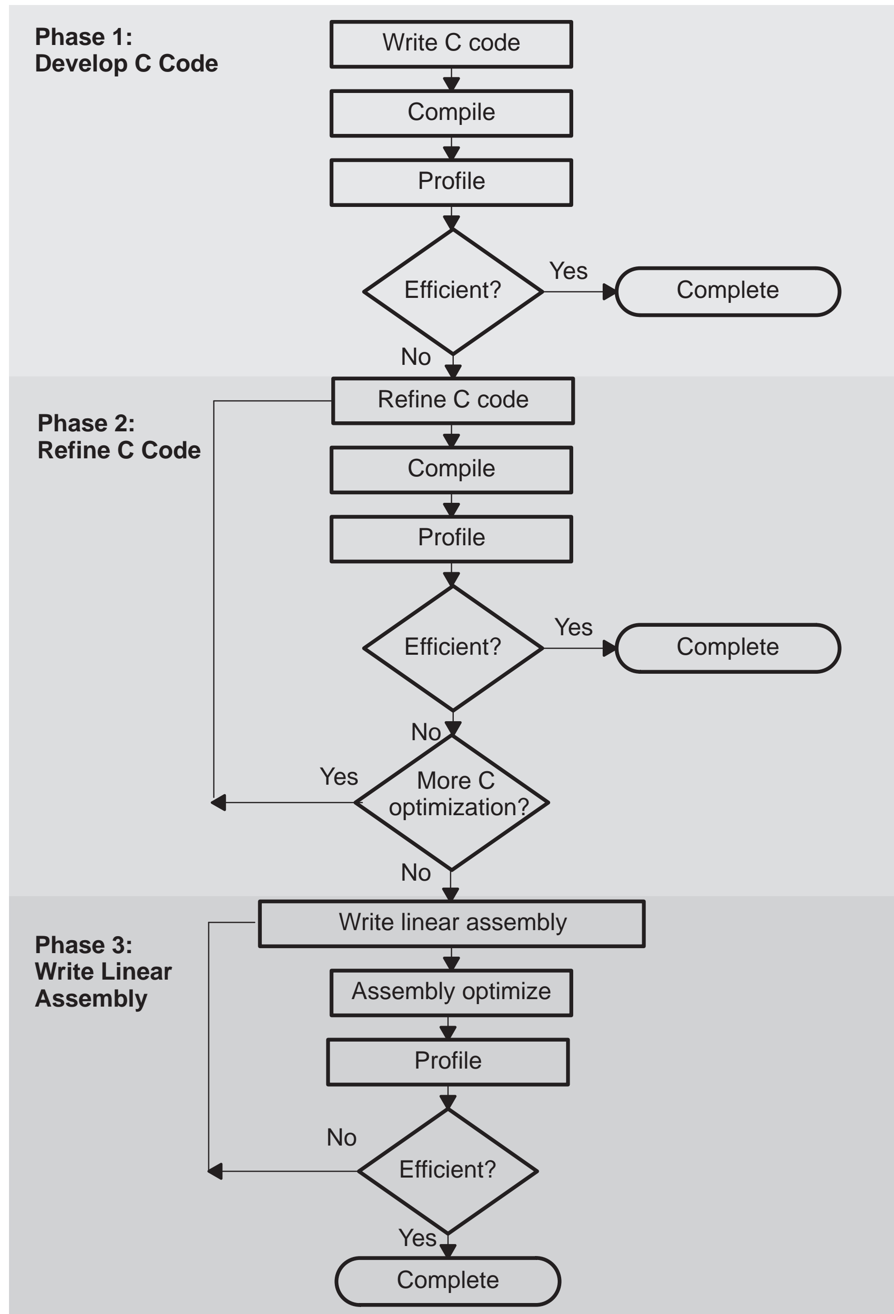
1.2 TMS320C6x Pipeline

The ’C6x pipeline has several features that provide optimum performance, low cost, and simple programming.

- Increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operations.
- Pipeline control is simplified by eliminating pipeline locks.
- The pipeline can dispatch eight parallel instructions every cycle.
- Parallel instructions proceed simultaneously through the same pipeline phases.

1.3 Code Development Flow to Increase Performance

You can achieve the best performance from your ’C6x code if you follow this flow when you are writing and debugging your code:



The following lists the phases in the 3-step software development flow shown on page 1-3, and the goal for each phase:

Phase	Goal
1	You can develop your C code for phase 1 without any knowledge of the ’C6x. Use the ’C6x profiling tools that are described in the *TMS320C6x C Source Debugger User’s Guide* to identify any inefficient areas that you might have in your C code. To improve the performance of your code, proceed to phase 2.
2	Use the intrinsics, shell options, and techniques that are described in Chapter 4 of this book to improve your C code. Use the ’C6x profiling tools to check its performance. If your code is still not as efficient as you would like it to be, proceed to phase 3.
3	Extract the time-critical areas from your C code and rewrite the code in linear assembly. You can use the assembly optimizer to optimize this code.

Chapter 2

Code Development Flow Tutorial

Part I

This chapter walks you through the code development flow that was introduced in Chapter 1. It uses step-by-step instructions and code examples to show you how to use the software development tools in each phase of development.

Before you start this tutorial, you should install the code generation tools and the C source debugger. If you do not have a Texas Instruments C source debugger, use your own debugger to check your results.

The sample code that is used in this tutorial is included on the code generation tools CD-ROM. When you install your code generation tools, the example code is installed in the c6xtools directory. Use the code in that directory to go through the examples in this chapter.

The examples in this chapter were run on the most recent version of the software development tools that were available as of the publication of this book. Because the tools are being continuously improved, you may get different results if you are using a more recent version of the tools.

Topic **Page**

	Topic	Page
2.1	**Before You Begin**	**2-2**
2.2	**Introduction to the Example Code**	**2-3**
2.3	**Lesson 1: Compiling, Assembling, and Linking the Example Code**	**2-5**
2.4	**Lesson 2: Profiling the Example Code**	**2-8**
2.5	**Lesson 3: Phase 1 of the Code Development Flow**	**2-14**
2.6	**Lesson 4: Phase 2 of the Code Development Flow**	**2-17**
2.7	**Lesson 5: Phase 3 of the Code Development Flow**	**2-25**
2.8	**Summary**	**2-31**

2.1 Before You Begin

This tutorial contains three basic types of information:

Primary tasks

Primary tasks identify the main lessons in the tutorial; they are boxed so that you can find them easily. A primary task looks like this:

> On a command line, enter:
>
> `load6x count.out`

Important information

In addition to primary actions, important information ensures that the tutorial works correctly. Important information is marked like this:

Important! If you are using SunOS, be sure you reinitialize your shell before continuing with this tutorial.

Optional tasks

Optional tasks allow you to learn more about the 'C6x tools; however, you do not need to perform the optional tasks to complete the tutorial successfully. Optional tasks are marked like this:

Try This: The stand-alone simulator (load6x) is another tool that you can use to find out what the cycle count for each function is.

This tutorial is divided into lessons. Each lesson builds on the previous lesson. To get the most benefit from the tutorial, you should start at the beginning and work your way through each lesson in order to the end.

2.2 Introduction to the Example Code

The C code example that you will use to start this tutorial is demo1.c, which is shown in Example 2–1. This example calls three functions: mac1( ), vec_mpy1( ), and iir1( ).

Example 2–1. The Code Example — demo1.c

```
main(int argc, char *argv[])

{
    const short coefs[150];
    short optr[150];
    short state[2];
    const short a[150];
    const short b[150];
    int c = 0;
    int dotp[1] = {0};
    int sum= 0;
    short y[150];
    short scalar = 3345;
    const short x[150];

    sum = mac1(a, b, c, dotp);
    vec_mpy1(y, x, scalar);
    iir1(coefs, x, optr, state);
}
```


The mac1( ) function, a multiply accumulate and squaring accumulate example, is shown in Example 2–2. It is performing a dot product of vector a with vector b and is also squaring and summing vector b.

Example 2–2. The Multiply Accumulate Function—mac1.c

```
int mac1(const short *a, const short *b, int sqr, int *sum)
{
    int i;
    int dotp = *sum;

    for (i = 0; i < 150; i++)
    {
        dotp += b[i] * a[i];
        sqr  += b[i] * b[i];
    }

    *sum = dotp;
    return sqr;
}
```

The vec_mpy( ) function shown in Example 2–3 is a vector multiply, which is a scalar multiply followed by a right shift. The result is stored to a second vector.

Example 2–3. The Vector Multiply Function—vec_mpy1.c

```
void vec_mpy1(short y[], const short x[], short scalar)
{
    int i;

    for (i = 0; i < 150; i++)
        y[i] += ((scalar * x[i]) >> 15);
}
```

The third function, iir1( ), is a typical infinite impulse response (IIR) biquad filter. The code for this function is shown in Example 2–4.

Example 2–4. The Biquad Filter—iir1.c

```
void iir1(const short *coefs, const short *input,
          short *optr, short *state)
{
    short x;
    short t;
    int n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
            t = x + ((coefs[2] * state[0] +
                      coefs[3] * state[1]) >> 15);

            x = t + ((coefs[0] * state[0] +
                      coefs[1] * state[1]) >> 15);

            state[1] = state[0];
            state[0] = t;
            coefs  += 4;  /* point to next filter coefs  */
            state  += 2;  /* point to next filter states */
        }

    *optr++ = x;
}
```

2.3 Lesson 1: Compiling, Assembling, and Linking the Example Code

The first step is to compile, assemble, and link the code.

Compiling for the ’C62x:
On a command line, enter the following on a single line:

```
cl6x -g -o -k -mg demo1.c mac1.c vec_mpy1.c iir1.c
-z lnk.cmd -l rts6201.lib -o demo1.out
```

Compiling for the ’C67x:
On a command line, enter the following on a single line:

```
cl6x -g -o -k -mg -mv6700 demo1.c mac1.c vec_mpy1.c
iir1.c -z lnk.cmd -l rts6701.lib -o demo1.out
```

You should not receive any errors, and the file, demo1.out, should be created. If you receive an error message, look up that error message in the appropriate user’s guide.

Here is a description of what you told the shell program (cl6x) to do:

cl6x	Run the compiler and the assembler.
–g	Generate symbolic debugging directives that are used by the debugger.
–o	Invoke the optimizer at the default level (–o is the same as –o2). Not all optimizations work well with debugging because the optimizer’s rearrangement of code can make it difficult for you to correlate source code with object code. Using the –g option with the –o option allows for the maximum amount of optimization that is compatible with debugging.
–k	Keep the assembly output files. Notice that you now have the following .asm files in your current directory: demo1.asm, mac1.asm, vec_mpy1.asm, and iir1.asm. When the –k option is **not** used, the shell program deletes the assembly output files after assembly is complete.
–mg	Turn on the maximum amount of optimization that is compatible with profiling. The –mg option allows you to profile optimized code.
–mv6700	Compiler is invoked to target ’C67x devices. If this switch is not used, the compiler defaults to the ’C62x device. This code will run on a ’C67x device, but it will run slower if using floating-point instructions since the code will have been compiled for the ’C62x device.

–z	Invoke the linker. The addition of this option to the cl6x command line means that the code is compiled, assembled, and linked in one step.
lnk.cmd	Use lnk.cmd as the linker command file. Linker command files allow you to put linking information into a file, which is useful when you invoke the linker often with the same information. Linker command files are also useful because they allow you to use the MEMORY directive, which defines the target memory configuration, and the SECTIONS directive, which controls how sections are built and allocated.
–l rts6201.lib	Include the runtime-support library for the 'C62x device, rts6201.lib, which is included on your CD-ROM. The runtime-support functions in rts6201.lib were compiled for little-endian mode. For big-endian mode, use the runtime support functions in rts6201e.lib.
–l rts6701.lib	Include the runtime-support library for the 'C67x device, rts6701.lib, which is included on your CD-ROM. The runtime-support functions in rts6701.lib were compiled for little-endian mode. For big-endian mode, use the runtime support functions in rts6701e.lib.
–o demo1.out	Name the output file demo1.out. (The default is a.out.) Because this option comes after the –z option, it is considered a linker option and is interpreted differently than the –o option that you entered before –z.

Try This: The options above are used throughout the rest of this tutorial. They are fairly common and might be ones that you want to use repeatedly. To avoid having to retype them each time you run the code development tools, you can use the C_OPTIONS environment variable. The shell program uses the default options and/or input filenames that you name with the C_OPTIONS environment variable every time you run the shell.

Use the commands in Table 2–1 to set up the C_OPTIONS environment variable with the options used on page 2-5.

Table 2–1. Using the C_OPTIONS Environment Variable

Your Setup	What to Change	Command
Windows NT™	System applet	SET C_OPTION=–g –o –k –mg –z lnk.cmd –l rts6201.lib
Windows™ 95	autoexec.bat	SET C_OPTION=–g –o –k –mg –z lnk.cmd –l rts6201.lib
C shell	.cshrc	setenv C_OPTION ”–g –o –k –mg –z lnk.cmd –l rts6201.lib”
Bourne or Korn shell	.profile	setenv C_OPTION ”–g –o –k –mg –z lnk.cmd –l rts6201.lib”

Notice that the –o demo1.out linker option was not included. If it were included, running the second tutorial example, demo2.c, would result in an output file named demo1.out instead of a more logical name such as demo2.out.

Files must be explicitly called on command and not as an environment variable. To compile all of the C files in a directory, use the cl6x command with the appropriate options and use *.c where the files are normally indicated. For example:

```
cl6x –g –mg *.c –z lnk.cmd –l rts6201.lib –o demo1.out
```

Important! If you are using SunOS, be sure you reinitialize your shell before continuing with this tutorial:

- For C shells, enter the following on a command line:

```
source ~/.cshrc
```

- For Bourne or Korn shells, enter the following on a command line:

```
source ~/.profile
```

2.4 Lesson 2: Profiling the Example Code

Now, use the profiler to look at the output of demo1. In this lesson, you will use the profiler to see the total execution time in number of cycles of each C function in demo1.out.

To start the profiler and load demo1.out, follow these steps:

1) Double-click the icon for the debugger.
2) From the Profile menu, select Profile Mode.

   The debugger switches to profiling mode and displays only the Command, Disassembly, File, and Profile windows.
3) From the File menu, select Load Program.

   This displays the Load Program File dialog box.
4) Double-click the demo1.out file. To do so, you might need to change the working directory.

   This loads demo1.out into the profiler. Because the File window is reserved for C programs, it disappears.

To select the areas of demo1 that you want profiled, follow these steps:

1) From the Profile menu, select Select Areas.

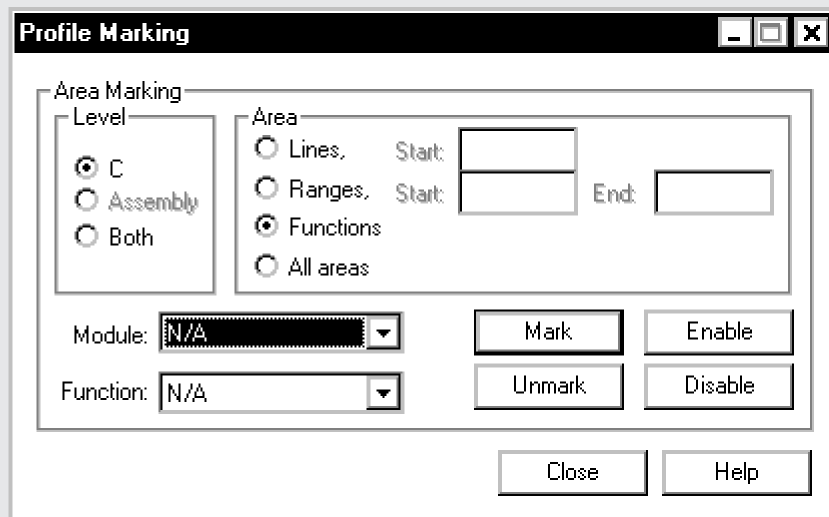
   This displays the Profile Marking dialog box.

2) In the Level box, select C.

3) In the Area box, select Functions.

   This indicates that the C functions in demo1.out will be your profile areas.

4) Click Mark.



5) Click Close.

   The Profile window is updated to include a line for each C function in demo1.

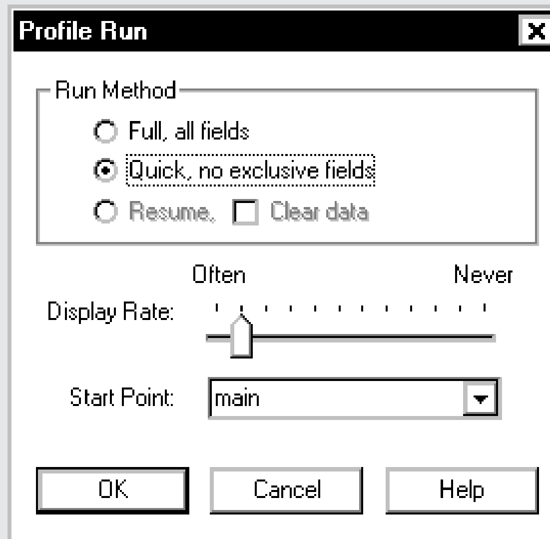
To start the profiling session, follow these steps:

1) Click the run icon on the toolbar:

This displays the Profile Run dialog box.

2) In the Run Method box, select Quick, no exclusive fields. This will show you the total execution time (cycle count) of a profile area, including the execution time of any subroutines called within the functions.

3) If main( ) is not already selected as your starting point, choose it from the list of starting points.



4) Click OK.

The Run Method dialog box closes and the status bar reads *Target: Profiling* to indicate that the profiling session has started.

The program restarts and runs to main( ) without profiling. Profiling begins when main( ) is reached and continues until the exit point of main( ) is reached. When profiling is complete, the status bar reads *Target: Halted* and your Profile window looks like this:

Profile

Type	Area Name	Count	Inclusive	Incl-Max
C Function	iir1()	1	270	270
C Function	mac1()	1	167	167
C Function	main()	1	831	831
C Function	vec_mpy1()	1	316	316

The Inclusive column indicates the cycle counts for each function, including any function that it calls. Because these functions do not call any other functions, the inclusive cycle counts are the same as the exclusive cycle counts. Notice that the cycle count for the mac1( ) function is 167, and that the cycle counts for the vec_mpy1( ) and iir1( ) functions are much higher—316 and 270, respectively.

To interpret the cycle counts in the Profile window, you need to understand how they are calculated. Here is the formula for calculating cycle counts:

$$\text{Execute packets} \times \text{loop iterations in C code} + \text{constant}$$

An execute packet is a group of parallel instructions. You can have up to eight instructions executing in parallel; therefore, each execute packet can contain up to eight instructions. An example of execute packets is shown in Example 2–7 on page 2-15.

Table 2–2 shows how the cycle counts were calculated for each function.

Table 2–2. Cycle Counts

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
mac1( )	1	150	17	$1 \times 150 + 17 = 167$
vec_mpy1( )	2	150	16	$2 \times 150 + 16 = 316$
iir1( )	5	50	20	$5 \times 50 + 20 = 270$

Try This: The stand-alone simulator (load6x) is another tool that you can use to find out what the cycle count for each function is. To get cycle count information for each function with the stand-alone simulator, embed the clock( ) function in your C code. Example 2–5 shows how to rewrite demo1.c to include the clock( ) function.

Example 2–5. Including the clock( ) Function in demo1.c (count.c)

```
#include <stdio.h>
#include <time.h>

main(int argc, char *argv[])

{
    const short coefs[150];
  short optr[150];
  short state[2];
  const short a[150];
  const short b[150];
  int c = 0;
  int dotp[1] = {0};
  int sum= 0;
  short y[150];
  short scalar = 3345;
  const short x[150];
  clock_t start, stop, overhead;

  start    = clock();
  stop     = clock();
  overhead = stop - start;

  start = clock();
  sum = mac1(a, b, c, dotp);
  stop = clock();
  printf(”mac1 cycles: %d\n”, stop - start - overhead);

  start = clock();
  vec_mpy1(y, x, scalar);
  stop = clock();
 printf(”vec_mpy1 cycles: %d\n”, stop - start - overhead);

  start = clock();
  iir1(coefs, x, optr, state);
  stop = clock();
  printf(”iir1 cycles: %d\n”, stop - start - overhead);
}
```

Note:

When using this method, remember to calculate the overhead and include the appropriate header files.

Now, compile, assemble, and link count.c.

> If you did not set up your C_OPTIONS environment variable as described on page 2-6, enter the following on a command line:
>
> ```
> cl6x -g -o -k -mg count.c mac1.c vec_mpy1.c iir1.c
> -z lnk.cmd -l rts6201.lib -o count.out
> ```
>
> OR
>
> If you set up your C_OPTIONS environment variable as described on page 2-6, enter the following on a command line:
>
> ```
> cl6x -z -o count.out
> ```
>
> Although the –z option is already specified in the C_OPTIONS environment variable, you need to specify it on the command line to indicate that this occurrence of –o is a linker option.

Use load6x to see the output of the printf statements that were embedded in the C code.

> On a command line, enter:
>
> ```
> load6x count.out
> ```

You should see the following output:

```
TMS320C6x C I/O COFF Loader      Version 1.01
Copyright (c) 1989–1997 Texas Instruments Incorporated
Interrupt to abort . . .
mac1 cycles: 175
vec_mpy1 cycles: 324
iir1 cycles: 278
NORMAL COMPLETION: 20949 cycles
```

Notice that these cycle counts are higher than the cycle counts that you saw with the profiler. For example, mac1 is listed here as having 175 cycles; however, it was listed in the Profiler window as having 167 cycles. You will see some extra cycles when you use load6x because you still have overhead for each function call. When you use the profiler, the cycles needed for calling the functions are not included in the profile display.

The *Using the Stand–Alone Simulator* chapter in the *TMS320C6x Optimizing C Compiler User's Guide* discusses load6x in more detail.

2.5 Lesson 3: Phase 1 of the Code Development Flow

Looking at the functions in demo1 one at a time, you can determine whether or not they need to be improved and, if they do need to be improved, how they can be improved. Start by looking at the first function, mac1( ).

Example 2–6 shows the assembly output of the function's inner loop kernel. The loop kernel is the area of the loop with the most parallelism. Only the inner loop is shown, because this is the area that can be improved with software pipelining. Notice that there are eight instructions executing in parallel (as indicated by the seven sets of parallel bars). This is the maximum number of instructions that the 'C6x can execute in parallel, so this code does not need to be improved.

Part I

Example 2–6. Inner Loop Kernel of mac1.asm

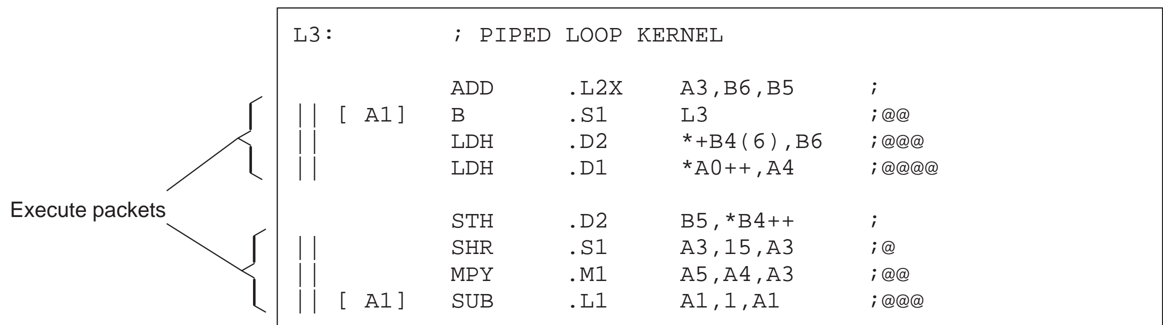
```
L3:         ; PIPED LOOP KERNEL

            ADD     .L2     B4,B7,B7      ;
||          ADD     .L1     A5,A3,A3      ;
||          MPY     .M2X    A4,B5,B4      ;@@
||          MPY     .M1     A4,A4,A5      ;@@
||  [ B0]   B       .S1     L3            ;@@@@@
||  [ B0]   SUB     .S2     B0,1,B0       ;@@@@@@
||          LDH     .D1     *A0++,A4      ;@@@@@@@
||          LDH     .D2     *B6++,B5      ;@@@@@@@
```

The @ characters specify the iteration of the loop that an instruction is on in the software pipeline; these symbols are automatically created by the code generation tools. The first iteration does not have an @ character; one @ character represents the second iteration; two @ characters represents the third iteration, and so on.

Because the mac1( ) function does not need to be improved, it does not need to go beyond phase 1 of the code development flow.

Look at Example 2–7, which shows the assembly output of the innermost loop for the vec_mpy1( ) function. Recall from page 2-11 that the vec_mpy1( ) function took 316 cycles to execute. This code is not as parallel as the mac1( ) function. The assembly output for the vec_mpy1( ) function shows two execute packets. Each execute packet has four parallel instructions. This loop can be improved.

Example 2–7. Inner Loop Kernel of vec_mpy1.asm




```
L3:         ; PIPED LOOP KERNEL

            ADD     .L2X    A3,B6,B5     ;
|| [ A1]    B       .S1     L3           ;@@
||          LDH     .D2     *+B4(6),B6   ;@@@
||          LDH     .D1     *A0++,A4     ;@@@@

            STH     .D2     B5,*B4++     ;
||          SHR     .S1     A3,15,A3     ;@
||          MPY     .M1     A5,A4,A3     ;@@
|| [ A1]    SUB     .L1     A1,1,A1      ;@@@
```

Example 2–8 shows the assembly output of the innermost loop for the iir( ) function. Recall from page 2-11 that the iir1( ) function took 270 cycles to execute. As you can see, some execute packets have five parallel instructions, while others have as few as four parallel instructions, which indicates that the code can probably be improved.

Example 2–8. Inner Loop Kernel of iir1.asm

```
L3:         ; PIPED LOOP KERNEL

            SHR     .S2     B4,15,B4      ;
||          SHR     .S1     A3,15,A5      ;
||          MPY     .M2X    B6,A5,B6      ;@
||          LDH     .D1     *+A6(16),A4   ;@@
||          LDH     .D2     *+B7(10),B6   ;@@

            ADD     .L1     A0,A5,A0      ;
||          MPY     .M1X    B6,A3,A3      ;@
||          MPY     .M2X    B5,A4,B5      ;@
||          LDH     .D1     *+A6(22),A3   ;@@
||          LDH     .D2     *+B7(8),B5    ;@@

            EXT     .S1     A0,16,16,A0   ;
||          STH     .D2     B5,*+B7(6)    ;@
||          MPY     .M1X    B5,A3,A4      ;@
||          LDH     .D1     *+A6(20),A3   ;@@

            ADD     .S1     8,A6,A6       ;
||          STH     .D2     A0,*B7++(4)   ;
||          ADD     .L1X    A0,B4,A0      ;
|| [ B0]    SUB     .L2     B0,1,B0       ;@
||          ADD     .S2     B6,B5,B4      ;@

            EXT     .S1     A0,16,16,A0   ;
|| [ B0]    B       .S2     L3            ;@
||          ADD     .L1     A3,A4,A3      ;@
||          LDH     .D1     *+A6(18),A5   ;@@@
```

To improve the vec_mpy( ) and iir( ) functions, start by seeing how you can refine and improve your C code. This is what is referred to as phase 2 of the code development flow, and this is what the next lesson is about.

2.6 Lesson 4: Phase 2 of the Code Development Flow

For your convenience, the vec_mpy1( ) function is duplicated here as Example 2–9 (the C version) and Example 2–10 (the assembly output of the inner loop). This is the same code that you saw in Example 2–3 and Example 2–7.

Example 2–9. The Vector Multiply Function—vec_mpy1.c

```
void vec_mpy1(short y[], const short x[], short scalar)
{
    int         i;

    for (i = 0; i < 150; i++)
        y[i] += ((scalar * x[i]) >> 15);
}
```

Example 2–9 uses short data types. Because short data types are 16 bits, they translate into halfword instructions, such as LDH and STH (see Example 2–10).

The loop in Example 2–10 uses two LDH instructions and an STH instruction to load x[i] and y[i] and store back to y[i]. Because only two memory operations can occur per cycle, the fastest that this loop can execute is one y[i] result every two cycles. The performance of this loop is limited by the number of D units.

Example 2–10. Inner Loop Kernel of vec_mpy1.asm

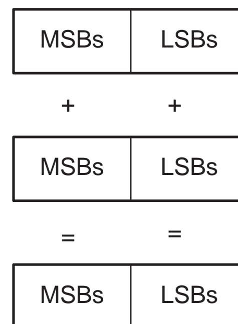
```
L3:         ; PIPED LOOP KERNEL

            ADD      .L2X     A3,B6,B5      ;
|| [ A1]    B        .S1      L3            ;@@
||          LDH      .D2      *+B4(6),B6    ;@@@
||          LDH      .D1      *A0++,A4      ;@@@@

            STH      .D2      B5,*B4++      ;
||          SHR      .S1      A3,15,A3      ;@
||          MPY      .M1      A5,A4,A3      ;@@
|| [ A1]    SUB      .L1      A1,1,A1       ;@@@
```

Because x is an array, x[i] and x[i + 1] are next to each other in memory. This means that instead of using halfword instructions (LDH and STH) to load and store each element in the array, you can use word instructions (LDW and STW) to load and store two elements at a time, as long as the data is aligned on a word boundary. In other words, all word accesses should have the 2 LSBs of the address set to 0. Two elements at a time, x[i] and x[i + 1], fit into one 32-bit register.

To achieve this in C, declare x[ ] as an integer instead of as a short data type. Also, you need to use some intrinsics.

Now that you have determined that you can load x[i] and x[i + 1] into the same register, you need to figure out how to do it. You can do this by using the _mpy and _mpylh intrinsics. Intrinsics are like built-in C functions that correspond to 'C6x assembly language instructions. The _mpy intrinsic multiplies the 16 LSBs of one operand by the 16 LSBs of another and returns the result. The _mpylh intrinsic multiplies the 16 LSBs of the first operand by the 16 MSBs of the second and returns the result.

You can then use the _add2 intrinsic to add the 16 MSBs of the first operand to the 16 MSBs of the second operand. At the same time, the _add2 intrinsic also adds the 16 LSBs of the first operand to the 16 LSBs of the second operand. The result of both additions is stored in a 32-bit operand.



Example 2–11 shows how to rewrite the vec_mpy( ) function to include the _mpy and _mpylh intrinsics:

Example 2–11. The Revised Vector Multiply Function—vec_mpy2.c

```
void vec_mpy2(int y[], const int x[], short scalar)
{
    int          i, val;
    unsigned int temph, templ;

    for (i = 0; i < 75; i++)
    {
     val = x[i];
     templ   = (_mpy  (scalar, val) >> 15) & 0x0000ffff;
     temph   = (_mpylh(scalar, val) << 1 ) & 0xffff0000;
     y[i] = _add2(y[i], temph | templ);
    }
}
```

Now, look at the iir1( ) function. Example 2–12 shows the same code that you saw in Example 2–4.

Example 2–12. The Biquad Filter—iir1.c

```
void iir1(const short *coefs, const short *input,
          short *optr, short *state)
{
    short                x;
    short                t;
    int                  n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
            t = x + ((coefs[2] * state[0] +
                      coefs[3] * state[1]) >> 15);

            x = t + ((coefs[0] * state[0] +
                      coefs[1] * state[1]) >> 15);

            state[1] = state[0];
            state[0] = t;
            coefs  += 4;  /* point to next filter coefs  */
            state  += 2;  /* point to next filter states */
        }

    *optr++ = x;
}
```

You can improve the iir( ) function by using the same methods that you used to improve the vec_mpy( ) function. Example 2–13 shows how to rewrite the iir( ) function:

Example 2–13. The Revised Biquad Filter—iir2.c

```
void iir2(const int *coefs, const short *input,
          short *optr, short *state)
{
    short                x;
    short                t;
    int                  n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
        t= x+((_mpy(coefs[1],state[0]) +
             _mpyhl(coefs[1],state[1])) >> 15);

        x= t+((_mpy(coefs[0],state[0]) +
             _mpyhl(coefs[0],state[1])) >> 15);

        state[1] = state[0];
        state[0] = t;

        coefs += 2;
        state += 2;
    }

     *optr++ = x;
}
```

Using demo2.c, shown in Example 2–14, run the revised functions through the compiler, assembler, and linker.

Example 2–14. The Revised Example—demo2.c

```
main(int argc, char *argv[])

{
    const short coefs[100];
    short optr[100];
    short state[2];
    const short a[100];
    const short b[100];
    int c = 0;
    int dotp[1] = {0};
    int sum= 0;
    short y[100];
    short scalar = 3345;
    const short x[100];

    sum = mac1(a, b, c, dotp);
    vec_mpy2(y, x, scalar);
    iir2(coefs, x, optr, state);
}
```


If you did not set up your C_OPTIONS environment variable as described on page 2-6, enter the following on a command line:

```
cl6x -g -o -k -mg demo2.c mac1.c vec_mpy2.c iir2.c
-z lnk.cmd -l rts6201.lib -o demo2.out
```

OR

If you set up your C_OPTIONS environment variable as described on page 2-6, enter the following on a command line:

```
cl6x -z -o demo2.out
```

Although the –z option is already specified in the C_OPTIONS environment variable, you need to specify it on the command line to indicate that this occurrence of –o is a linker option.

The inner loop of the vec_mpy2( ) function translates into the assembly output shown in Example 2–15.

Example 2–15. Inner Loop Kernel of vec_mpy2.asm

```
L3:           ; PIPED LOOP KERNEL

              OR       .L2X    B5,A8,B7      ;@
||            SHL      .S1     A6,1,A4       ;@@
|| [ A1]      B        .S2     L3            ;@@
||            AND      .L1     A5,A4,A6      ;@@
||            LDW      .D2     *+B4(12),B5   ;@@@
||            MPYLH    .M1     A0,A9,A6      ;@@@
||            LDW      .D1     *A3++,A9      ;@@@@@@

              STW      .D2     B6,*B4++      ;
||            ADD2     .S2     B5,B7,B6      ;@
||            AND      .L1     A7,A4,A8      ;@@
||            MV       .L2X    A6,B5         ;@@
|| [ A1]      SUB      .D1     A1,1,A1       ;@@@
||            SHR      .S1     A8,15,A4      ;@@@
||            MPY      .M1     A0,A9,A8      ;@@@@
```

As you can see, the code for the vec_mpy2( ) function is improved over the original vec_mpy( ) code. Two LDW instructions are loading four elements (x[i], x[i+1], y[i], and y[i+1]), and one STW instruction is storing two elements: x[i] and y[i+1]. With the revised code, two y[i] results are stored every two cycles. Recall that only one y[i] result was stored every two cycles in Example 2–10.

Table 2–3 shows how the vec_mpy( ) function has improved as it moves from phase 1 to phase 2.

Table 2–3. Revised Cycle Counts for vec_mpy( )

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
vec_mpy1( )	2	150	16	$2 \times 150 + 16 = 316$
vec_mpy2( )	2	75	22	$2 \times 75 + 22 = 172$

Now, look at the inner loop of the third function, iir( ). Example 2–16 shows the assembly output of the innermost loop for the revised iir( ) function:

Example 2–16. Inner Loop Kernel of iir2.asm

```
L3:         ; PIPED LOOP KERNEL

            ADD     .L2     B7,B8,B7      ;
||          ADD     .L1     A0,A3,A0      ;
||          MV      .S2     B6,B9         ;@
||          STH     .D1     A5,*+A4(6)    ;@
||          LDW     .D2     *B5++(8),B8   ;@@

            SHR     .S2     B7,15,B7      ;
||          EXT     .S1     A0,16,16,A0   ;
|| [ B0]    SUB     .L2     B0,1,B0       ;@
||          MPY     .M2X    B8,A5,B8      ;@
||          ADD     .L1X    B6,A3,A3      ;@
||          LDH     .D2     *+B4(14),B6   ;@@@

            ADD     .L1X    A0,B7,A6      ;
||          MPYHL   .M2     B8,B9,B7      ;@
||          SHR     .S1     A3,15,A3      ;@
|| [ B0]    B       .S2     L3            ;@
||          LDW     .D2     *+B5(4),B7    ;@@@
||          LDH     .D1     *+A4(12),A5   ;@@@

            ADD     .L2     4,B4,B4       ;
||          STH     .D1     A0,*A4++(4)   ;
||          EXT     .S1     A6,16,16,A0   ;
||          MPYHL   .M2     B7,B6,B6      ;@@
||          MPY     .M1X    B7,A5,A3      ;@@
```

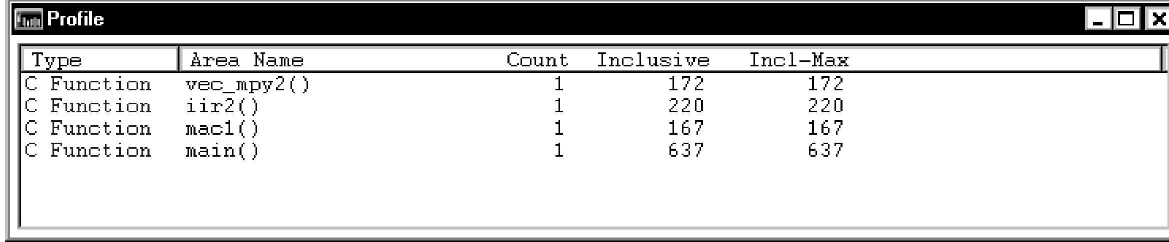
Table 2–4 shows how the iir( ) function has improved. Now, the code has only four execute packets; however, each packet has only five or six parallel instructions, which could be probably improved.

Table 2–4. Revised Cycle Counts for iir( )

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
iir1( )	5	50	20	$5 \times 50 + 20 = 270$
iir2( )	4	50	20	$4 \times 50 + 20 = 220$

Use the profiler to view the cycle counts of the revised functions.

Your profile window should look like this:



Profile

Type	Area Name	Count	Inclusive	Incl-Max
C Function	vec_mpy2()	1	172	172
C Function	iir2()	1	220	220
C Function	mac1()	1	167	167
C Function	main()	1	637	637

Notice that the cycle count for the second function, the vector multiply, is down from 316 to 172. The IIR filter has improved also: it is down from 270 to 220. However, the cycle count for the IIR filter is still too high. Naturally, the cycle count for main( ) has decreased also. It is down from 831 to 637.

Table 2–5. Revised Cycle Counts

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
mac1( )†	1	150	17	1 × 150 + 17 = 167
vec_mpy2( )	2	75	22	2 × 75 + 22 = 172
iir2( )	4	50	20	4 × 50 + 20 = 220

† The cycle count for the mac1( ) function has not changed.

You have done everything you can to refine the C code in the iir( ) function. To improve your code at this point, you need to use the assembly optimizer. This leads you to phase 3 of the code development flow.

2.7 Lesson 5: Phase 3 of the Code Development Flow

To further improve the iir( ) function, you will need to rewrite it in linear assembly. Linear assembly is the input for the assembly optimizer.

Linear assembly is similar to regular ’C6x assembly code in that you use ’C6x instructions to write your code. With linear assembly, however, you do not need to specify all of the information that you need to specify in regular ’C6x assembly code. With linear assembly code, you have the option of specifying the information or letting the assembly optimizer specify it for you. Here is the information that you do *not* need to specify in linear assembly code:

- Parallel instructions
- Pipeline latency
- Register usage
- Which functional unit is being used

If you choose not to specify these things, the assembly optimizer determines the information that you do not include, based on the information that it has about your code. As with other code generation tools, you might need to modify your linear assembly code until you are satisfied with its performance. When you do this, you will probably want to add more detail to your linear assembly. For example, you might want to specify which functional unit should be used.

Before you use the assembly optimizer, you need to know the following things about how it works:

- A linear assembly file must be specified with a **.sa** extension.
- Linear assembly code should include the **.cproc** and **.endproc** directives. The .cproc and .endproc directives delimit a section of your code that you want the assembly optimizer to optimize. Use .cproc at the beginning of the section and .endproc at the end of the section. In this way, you can set off sections of your assembly code that you want to be optimized, like procedures or functions.
- Linear assembly code may include a **.reg** directive. The .reg directive allows you to use descriptive names for values that will be stored in registers. When you use .reg, the assembly optimizer chooses a register whose use agrees with the functional units chosen for the instructions that operate on the value.
- Linear assembly code may include a **.trip** directive. The .trip directive specifies the value of the trip count. The trip count indicates how many times a loop will iterate.

Now that you have some information about the fundamentals of linear assembly code, look at the revised C code for the biquad filter again. Example 2–17 shows the same code that you saw in Example 2–13 on page 2-20.

Example 2–17. The Revised Biquad Filter—iir2.c

```
void iir2(const int *coefs, const short *input,
          short *optr, short *state)
{
    short                x;
    short                t;
    int                  n;

    x = input[0];

    for (n = 0; n < 50; n++)
    {
        t= x+((_mpy(coefs[1],state[0]) +
             _mpyhl(coefs[1],state[1])) >> 15);

        x= t+((_mpy(coefs[0],state[0]) +
             _mpyhl(coefs[0],state[1])) >> 15);

        state[1] = state[0];
        state[0] = t;

        coefs += 2;
        state += 2;
    }

     *optr++ = x;
}
```

Example 2–18 shows how to rewrite the iir( ) function in linear assembly.

Example 2–18. The Biquad Filter, Revised and Assembly-Optimized—iir3.sa

```
        .def    _iir3

_iir3   .cproc  cptr0,sptr0

        .reg cptr1, s01, s10, s23, c10, c32, s10_s, s10_t
        .reg p0, p1, p2, p3, s23_s, s1, t, x, mask, sptr1, s10p, ctr

        MV      .2     cptr0,cptr1
        MV      .1     sptr0,sptr1

        MVK      50,ctr                 ; setup loop counter

LOOP:   .trip 50

        LDW     .D1T1   *cptr0++[2],c32 ; coefAddr[3] & CoefAddr[2]
        LDW     .D2T2   *cptr1++[2],c10 ; CoefAddr[1] & CoefAddr[0]
        LDW     .D1T2   *sptr0,s10      ; StateAddr[1] & StateAddr[0]
        MV      .2      s10,s10p        ; save StateAddr[1] & StateAddr[0]

        MPY     .M1     c32,s10,p2      ; CoefAddr[2] * StateAddr[0]
        MPYH    .M1     c32,s10,p3      ; CoefAddr[3] * StateAddr[1]
        ADD     .1      p2,p3,s23       ; CA[2] * SA[0] + CA[3] * SA[1]
        SHR     .1      s23,15,s23_s    ; (CA[2] * SA[0] + CA[3] * SA[1]) >> 15
        ADD     .2      s23_s,x,t       ; t = x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
        AND     .2      t,mask,t        ; clear upper 16 bits

        MPY     .M2     c10,s10,p0      ; CoefAddr[0] * StateAddr[0]
        MPYH    .M2     c10,s10,p1      ; CoefAddr[1] * StateAddr[1]
        ADD     .2      p0,p1,s10_t     ; CA[0] * SA[0] + CA[1] * SA[1]
        SHR     .2      s10_t,15,s10_s  ; (CA[0] * SA[0] + CA[1] * SA[1]) >> 15
        ADD     .2      s10_s,t,x       ; x = t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)

        SHL     .2      s10p,16,s1      ; StateAddr[1] = StateAddr[0]
        OR      .2      t,s1,s01        ; StateAddr[0] = t
        STW     .D1     s01,*sptr1++    ; store StateAddr[1] & StateAddr[0]

  [ctr] ADD     .S1     -1,ctr,ctr      ; dec outer lp cntr
  [ctr] B       .S1     LOOP            ; Branch outer loop

        .endproc
```

Using demo3.c, shown in Example 2–19, run the revised functions through the code generation tools.

Example 2–19. The Revised Example—demo3.c

```
main(int argc, char *argv[])

{
    const short coefs[150];
    short optr[150];
    short state[2];
    const short a[150];
    const short b[150];
    int c = 0;
    int dotp[1] = {0};
    int sum = 0;
    short y[150];
    short scalar = 3345;
    const short x[150];

    sum = mac1(a, b, c, dotp);
    vec_mpy2(y, x, scalar);
    iir3(coefs, x, optr, state);
}
```

Use the shell program (cl6x) to compile, assemble, and link. Be sure you use the –mg option. The –mg option ensures that the optimizations that are used are compatible with profiling.

On a command line, enter:

```
cl6x -g -o -k -mg demo3.c mac1.c vec_mpy2.c iir3.sa
-z lnk.cmd -l rts6201.lib -o demo3.out
```

Notice that you used the shell program to compile a linear assembly file and a C file at the same time. Also notice that (except for the –mg option) you used the same options that you used in the first part of this tutorial. The assembly optimizer has a small set of some unique options, but many of the options that you will use are shell options that apply to either linear assembly files or C files.

Example 2–20. Inner Loop Kernel of iir3.asm

```
L3:      ; PIPED LOOP KERNEL

         AND     .L2     B3,B7,B0      ; clear upper 16 bits
||       ADD     .S2     B0,B8,B8      ;@ CA[0] * SA[0] + CA[1] * SA[1]
|| [ A1] B       .S1     L3            ;@ Branch outer loop
||       ADD     .L1     A4,A5,A4      ;@ CA[2] * SA[0] + CA[3] * SA[1]
||       MPYH    .M2     B2,B1,B8      ;@@ CoefAddr[1] * StateAddr[1]
||       MPY     .M1X    A0,B1,A4      ;@@ CoefAddr[2] * StateAddr[0]
||       LDW     .D2     *B6++(8),B2   ;@@@@ CoefAddr[1] & CoefAddr[0]
||       LDW     .D1     *A3++(8),A0   ;@@@@ coefAddr[3] & CoefAddr[2]

         ADD     .D2     B4,B0,B9      ; x = t+((CA[0]*SA[0]+CA[1]*SA[1])>>15)
||       OR      .L2     B0,B9,B0      ; StateAddr[0] = t
||       SHR     .S2     B8,0xf,B4     ;@ (CA[0] * SA[0] + CA[1] * SA[1]) >> 15
||       SHR     .S1     A4,0xf,A5     ;@ (CA[2] * SA[0] + CA[3] * SA[1]) >> 15
||       MPY     .M2     B2,B1,B0      ;@@ CoefAddr[0] * StateAddr[0]
||       MPYH    .M1X    A0,B1,A5      ;@@ CoefAddr[3] * StateAddr[1]
||       LDW     .D1     *A6++,B1      ;@@@@ StateAddr[1] & StateAddr[0]

         STW     .D1     B0,*A7++      ; store StateAddr[1] & StateAddr[0]
||       SHL     .S2     B5,0x10,B9    ;@ StateAddr[1] = StateAddr[0]
||       ADD     .L2X    B9,A5,B3      ;@ t = x+((CA[2]*SA[0]+CA[3]*SA[1])>>15)
|| [ A1] ADD     .S1     0xffffffff,A1,A1 ;@@ dec outer lp cntr
||       MV      .D2     B1,B5         ;@@ save StateAddr[1] & StateAddr[0]
```

Table 2–6 shows how the iir( ) function has improved as it has moved through the three phases of code development.

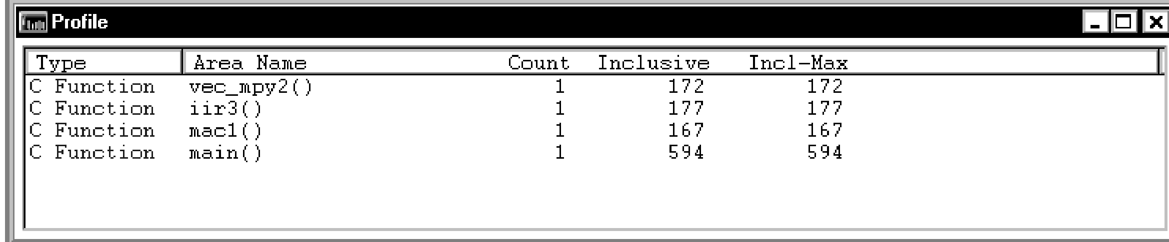
Table 2–6. Revised Cycle Counts for iir( )

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
iir1( )	6	50	20	6 × 50 + 20 = 270
iir2( )	4	50	20	4 × 50 + 20 = 220
iir3( )	3	50	27	3 × 50 + 27 = 177

Part I

Use the profiler to view the cycle counts of the revised functions.

Your profile window should look like this:



Notice that the cycle count for the IIR filter has improved: it is down from 220 to 177. Naturally, the cycle count for main( ) has decreased also. It is down from 637 to 594.

Table 2–7. Revised Cycle Counts

Function	Execute Packets	Loop Iterations	Constant	Cycle Count
mac1( )†	1	150	17	$1 \times 150 + 17 = 167$
vec_mpy2( )†	2	75	22	$2 \times 75 + 22 = 172$
iir3( )	3	50	27	$3 \times 50 + 27 = 177$

† The cycle count for the mac1( ) function and the vec_mpy( ) function have not changed.

The *Using the Assembly Optimizer* chapter in the *TMS320C6x Optimizing C Compiler User's Guide* discusses the assembly optimizer in more detail.

2.8 Summary

Congratulations! In this tutorial, you learned the following things:

- What the three phases of code development are, how to determine which phases are appropriate for improving different parts of your code, and how to write your code for each phase.
- What a linear assembly file is and some fundamental information on how to write one.
- How to use the code generation tools to compile, assemble, and link your C and linear assembly files.
- How to use the profiler to analyze your results and determine whether or not you need to continue refining your code.

Chapter 3

TMS320C6x Optimization Checklist

Because most of the millions of instructions per second (MIPS) in DSP applications occur in tight loops, it is important for the 'C6x code generation tools to make maximal use of all the hardware resources in important loops. Fortunately, loops inherently have more parallelism than non-looping code because there are multiple iterations of the same code executing with limited dependencies between each iteration. Through a technique called software pipelining, the 'C6x code generation tools use the multiple resources of the VelociTI architecture efficiently and obtain very high performance.

This chapter shows the code development flow recommended to achieve the highest performance on loops and provides a checklist that can be used to optimize loops with references to more detailed documentation.

Table 3–1 describes the steps recommended for developing code to achieve the highest performance on loops.

Table 3–1. Code Development Steps

Step	Description
1	Compile and profile native C code ❑ Validates original C code ❑ Determines which loops are most important in terms of MIPS requirements
2	Add const declarations and loop count information ❑ Reduces potential pointer aliasing problems ❑ Allows loops with indeterminate iteration counts to execute epilogs
3	Optimize C code using intrinsics and other methods ❑ Facilitates use of certain ’C6x instructions not easily represented in C ❑ Optimizes data flow bandwidth
4a	Write linear assembly ❑ Allows control in determining exact ’C6x instructions to be used ❑ Provides flexibility of hand-coded assembly without worry of pipelining, parallelism, or register allocation ❑ Can pass memory bank information to the tools
4b	Add partitioning information to the linear assembly ❑ Can improve partitioning of loops when necessary ❑ Can avoid bottlenecks of certain hardware resources

When you achieve the desired performance in your code, there is no need to move to the next step. Each of the steps in the development involve passing more information to the ’C6x tools. Even at the final step, development time is greatly reduced from that of hand-coding, and the performance approaches the best that can be achieved by hand.

Internal benchmarking efforts at Texas Instruments have shown that most loops achieve maximal throughput after steps 1 and 2. For loops that do not, the C compiler offers a rich set of optimizations that can fine tune all from the high level C language. For the few loops that need even further optimizations, the assembly optimizer gives the programmer more flexibility than C can offer, works within the framework of C, and is much like programming in higher level C. For more information on the assembly optimizer, see the *TMS320C6x Optimizing C Compiler User’s Guide* and Chapter 6, *Optimizing Assembly Code*

via Linear Assembly, in this book. For example, linear assembly files point to the demo directory included with the 'C6x tools.

In order to aid the development process, a feedback option (–mw) is included in the code generation tools. Example 3–1 shows output from the compiler and/or assembly optimizer of a particular loop. See the *TMS320C6x Optimizing C Compiler User's Guide* for more information about the –mw option.

Example 3–1. Compiler and/or Assembly Optimizer Feedback

```
;*====================================================*
;* SOFTWARE PIPELINE INFORMATION
;*
;*       Loop label     :  LOOP
;*       Loop Carried Dependency Bound  :  3
;*       Unpartitioned Resource Bound   :  3
;*       Partitioned Resource Bound(*)  :  4
;*       Resource Partition:
;*                               A–side    B–side
;*       .L units                  0          0
;*       .S units                  2          2
;*       .D units                  2          2
;*       .M units                  2          2
;*       .X cross paths            4*         3
;*       .T address paths          2          2
;*       Long read paths           1          0
;*       Long write paths          0          0
;*       Logical ops (.LS)         4          1
;*       Additional ops (.LSD)     2          1
;*       Bound (.L .S .LS)         3          2
;*       Bound (.L .S .D .LS .LSD) 4*         2
;*
;* Searching for software pipeline schedule
;*  at   ii = 4  Failed register allocation
;*       ii = 5  Schedule found with 4
;*               iterations in parallel
;*      Done
;*===================================================*
```

This feedback is important in determining which optimizations might be useful for further improved performance. The following checklist is provided as a quick reference to techniques that can be used to optimize loops and refers to specific sections within this book for more detail.

Table 3–2. TMS320C6x Optimization Checklist

Feedback	Solution	For more information, refer to …	Page #
Loop carried dependency bound is much larger than unpartitioned resource bound	C Code ✓ Use –pm program level optimization to reduce memory pointer aliasing.	*Performing Program-Level Optimization (–pm Option)*	4-8
	✓ Add const declarations to all pointers passed to a function that are read only.	*The const Keyword*	4-6
	✓ Use –mt option to assume no memory pointer aliasing.	*Memory Dependencies*	4-5
	Linear assembly ✓ Make sure instructions accessing memory at the beginning of the loop do not use the same pointer variables as instructions accessing memory at the end of the loop.	*Loop Carry Paths*	6-73
Partitioned resource bound is higher than unpartitioned resource bound	✓ Write code in linear assembly with partitioning/functional unit information.	*Linear Assembly Resource Allocation*	6-19
Too many instructions, or inefficient instructions were generated by the compiler	✓ Use intrinsics in C code to select more efficient ’C6x instructions.	*Using Intrinsics*	4-9
	✓ Write code in linear assembly to pick exact ’C6x instruction to be executed.	*TMS320C6x Optimizing C Compiler User’s Guide*	
Failed to software pipeline due to register live-too-long	✓ Write linear assembly and insert MV instructions to split register lifetimes that are live-too-long.	*Split-Join-Path Problems*	6-101
Failed to software pipeline due to register allocation	✓ Try splitting the loop into two separate loops. ✓ If multiple conditionals are used in the loop, allocation of the condition registers could be the reason for the failure. Try writing linear assembly and partition all instructions, writing to condition registers evenly between the A and B sides of the machine. If there are an uneven number, put more on the B side, since there are 3 condition registers on the B side and only 2 on the A side.		

Table 3–2. TMS320C6x Optimization Checklist (Continued)

Feedback	Solution	For more information, refer to …	Page #
T address paths are resource bound	C code		
	✓ Use word access for short arrays; declare int* and use mpy intrinsics to multiply upper and lower halves of registers.	*Using Word Access for Short Data* in *Part II*	4-14
	✓ Try to employ redundant load elimination technique if possible.	*Redundant Load Elimination*	6-106
	Linear assembly		
	✓ Use LDW/STW instructions for accesses to memory.	*Using Word Access for Short Data* in *Part III*	6-15
There are memory bank conflicts (specified in the memory analysis window of simulator)	✓ Write linear assembly and use the .mptr directive.	*Loop Unrolling*	6-90
Larger outer loop overhead in nested loop	✓ Unroll the inner loop.	*Loop Unrolling* in *Part II* and *Part III*	4-23, 6-90
	✓ Make one loop with the outer loop instructions conditional on an inner loop counter.	*Outer Loop Conditionally Executed With Inner Loop*	6-132
Uneven resources (for example, 3 multiplies per loop iteration)	✓ Unroll the loop to make an even number of resources.	*Loop Unrolling* in *Part III*	6-90
Two loops are generated, one not software pipelined	✓ Use the _nassert statement to specify loop count information.	*Communicating Trip-Count Information to the Compiler*	4-22
Two loops are generated, one not software pipelined	✓ Use the .trip directive to specify loop count information.	*Lesson 5: Phase 3 of the Code Development Flow*	2-25
Loop will not software pipeline for other reasons	✓ Make sure there are no function calls, branches to other code, or conditional break statements in the loop.	*What Disqualifies a Loop from Being Software-Pipelined*	4-26
	✓ Try making the loop counter down-counting and declare it an int in C.	*Tips on Data Types* and *Trip Count Issues*	4-2, 4-21
	✓ Remove any modifications to the loop counter inside the loop.	*What Disqualifies a Loop from Being Software-Pipelined*	4-26

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Chapter 4

Optimizing C Code

You can maximize C performance by using compiler options, intrinsics, and code transformations. This chapter discusses the following topics:

- The compiler and its options
- Intrinsics
- Software pipelining
- Loop unrolling

Part II

Topic **Page**

4.1 Writing C Code **4-2**

4.2 Compiling C Code **4-4**

4.3 Refining C Code **4-9**

4.1 Writing C Code

This chapter shows you how to analyze and tailor your code to be sure you are getting the best performance from the 'C6x architecture.

4.1.1 Tips on Data Types

Give careful consideration to the data type size when writing your code. The 'C6x compiler defines a size for each data type (signed and unsigned):

- char 8 bits
- short 16 bits
- int 32 bits
- long 40 bits
- float 32 bits
- double 64 bits

Based on the size of each data type, follow these guidelines when writing C code:

- Avoid code that assumes that int and long types are the same size, because the 'C6x compiler uses long values for 40-bit operations.
- Use the short data type for fixed-point multiplication inputs whenever possible because this data type provides the most efficient use of the 16-bit multiplier in the 'C6x.
- Use int or unsigned int types for loop counters, rather than short or unsigned short data types, to avoid unnecessary sign-extension instructions.
- When using floating-point instructions on a floating-point device such as the 'C67x, use the –mv6700 compiler switch so the code generated will use the device's floating-point hardware instead of performing the task with fixed point hardware.

4.1.2 Analyzing C Code Performance

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

- One of the preliminary measures of code is the time it takes the code to run. Use the clock( ) and printf( ) functions in C to time and display the performance of specific code regions. You can use the stand-alone simulator (load6x) to run the code for this purpose.
- Use the profile mode in the debugger, as explained in the *TMS320C6x C Source Debugger User's Guide*, to collect execution statistics about specific areas in your code.

- Use breakpoints, the clk register, and the RUNB command in the debugger, as described in the *TMS320C6x C Source Debugger User's Guide*, to track the number of CPU clock cycles consumed by a particular section of code.
- The critical performance areas in your code are most often loops. The easiest way to optimize a loop is by extracting it into a separate file that can be rewritten, recompiled, and run stand-alone.

As you use the techniques described in this chapter to optimize your C code, you can then evaluate the performance results by running the code and looking at the instructions generated by the compiler.

4.2 Compiling C Code

The ’C6x compiler offers high-level language support by transforming your C code into more efficient assembly language source code. The compiler tools include a shell program (cl6x), which you use to compile, assembly optimize, assemble, and link programs in a single step. To invoke the compiler shell, enter:

cl6x [*options*] [*filenames*] [**–z** [*linker options*] [*object files*]]

For a complete description of the C compiler and the options discussed in this chapter, see the *TMS320C6x Optimizing C Compiler User’s Guide*.

4.2.1 Compiler Options

Options control the operation of the compiler. Table 4–1 defines the options discussed in this chapter.

Table 4–1. Subset of Compiler Options

Option	Description
–o†	Enables software pipelining and other optimizations in the compiler
–pm‡	Enables program-level optimization
–mt	Enables the compiler to use assumptions that allow it to be more aggressive with certain optimizations
–mg	Allows you to profile optimized code
–ms	Ensures that redundant loops are not generated, thereby reducing code size
–k	Keeps the assembly file so that you can inspect it
–mu	Disables software pipelining
–mh <*n*>	Allows speculative execution

† Although –o3 is preferable, at a minimum use the –o option.
‡ Use the –pm option for as much of your program as possible.

4.2.2 Memory Dependencies

To maximize the efficiency of your code, the ’C6x compiler schedules as many instructions as possible in parallel. To schedule instructions in parallel, the compiler must determine the relationships, or dependencies, between instructions. Dependency means that one instruction must occur before another. Because only independent instructions can execute in parallel, dependencies inhibit parallelism.

- If the compiler cannot determine that two instructions are independent (for example, *b* does not depend on *a*), it assumes a dependency and schedules the two instructions sequentially.
- If the compiler can determine that two instructions are independent of one another, it can schedule them in parallel.

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

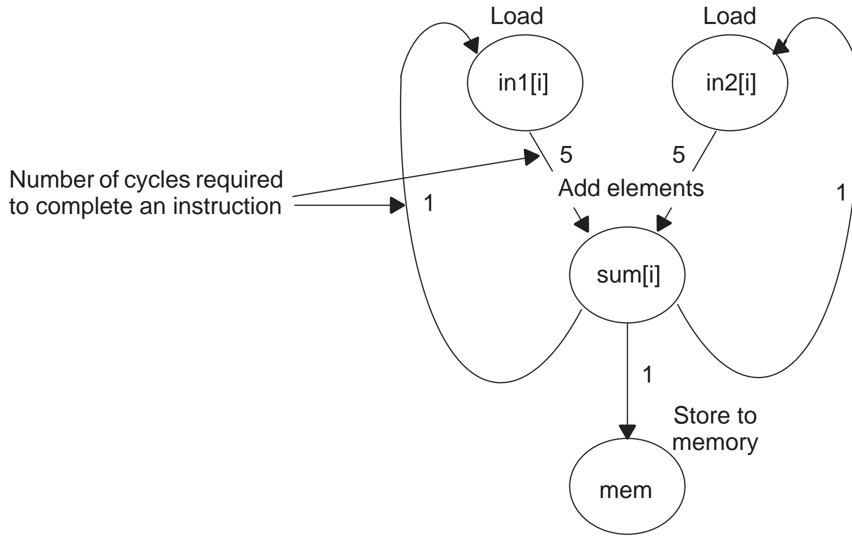
- Use the const keyword to indicate which objects are not changed by a function.
- Use the –pm (program-level optimization) option, which gives the compiler global access to the whole program or module and allows it to be more aggressive in ruling out dependencies.
- Use the –mt option, which allows the compiler to use assumptions that allow it to eliminate dependencies.

To illustrate the concept of memory dependencies, it is helpful to look at the algorithm code in a dependency graph. Example 4–1 shows the C code for a basic vector sum. Figure 4–1 shows the dependency graph for this basic vector sum. (For more information, see section 6.2.4, *Drawing a Dependency Graph,* on page 6-6.)

Example 4–1. Basic Vector Sum

```
void vecsum(short *sum, short *in1, short *in2, unsigned int N)
{
  int i;

  for (i = 0; i < N; i++)
    sum[i] = in1[i] + in2[i];

}
```

Figure 4–1. Dependency Graph for Vector Sum #1



The dependency graph in Figure 4–1 shows that:

- ❑ The paths from sum[i] back to 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);
```

4.2.2.1 The const Keyword

In Figure 4–1, the reads from in1 and in2 finish before the write to sum within a single iteration. However, the 'C6x compiler uses software pipelining to execute multiple iterations in parallel and, therefore, must determine memory dependencies that exist across loop iterations.

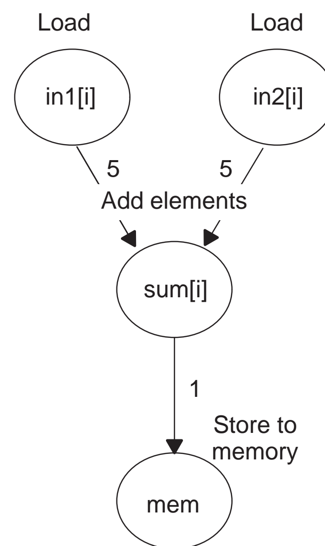
To help the compiler, you can qualify an object with the const keyword, which indicates that a variable or the memory referenced by a variable will not be changed, but will remain constant. It is good coding practice to use the const keyword wherever you can, because it is a simple way to increase the performance and robustness of your code.

Example 4–2 shows the vecsum( ) example rewritten with the const keyword to indicate that the write to sum never changes the memory referenced by in1 and in2. Figure 4–2 shows the revised dependency graph for the code in the inner loop.

Example 4–2. Vector Sum With const Keywords

```
void vecsum2(short *sum, const short *in1, const short *in2, unsigned int N)
{
  int i;

  for (i = 0; i < N; i++)
    sum[i] = in1[i] + in2[i];
}
```

Figure 4–2. Dependency Graph for Vector Sum #2



Part II

Example 4–3 shows the output of the compiler for the vector sum in Example 4–2. The compiler finds better schedules when dependency paths are eliminated between instructions. For this loop, the compiler found a software pipeline with a 2-cycle kernel, compared with seven cycles for the previous loop. (The kernel is the body of a pipelined loop where all instructions execute in parallel.)

Example 4–3. Compiler Output for Vector Sum Code

```
L14:                  ; PIPE LOOP KERNEL

       ADD    .L1X   B4,A0,A5
||[B0] B      .S2    L14
||     LDH    .D1    *A3++,A0

       STH    .D1    A5,*A4++
||[B0] SUB    .L2    B0,1,B0
||     LDH    .D2    *B5++,B4
```

For basic information on assembly code, see Chapter 4, *Structure of Assembly Code*.

4.2.2.2 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 called a module. The module moves to the optimization and code generation passes of the compiler. Because the compiler has access to the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- 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 function.

4.2.2.3 The –mt Option

Another way to eliminate memory dependencies is to use the –mt option, which allows the compiler to use assumptions that can eliminate memory dependency paths. For example, if you use the –mt option when compiling the code in Example 4–1, the compiler uses the assumption that that in1 and in2 do not alias memory pointed to by sum and, therefore, eliminates memory dependencies among the instructions that access those variables.

Part II

4.3 Refining C Code

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

- Using intrinsics to replace complicated C code
- Using word access to operate on 16-bit data stored in the high and low parts of a 32-bit register
- Software pipelining the instructions manually
- Using double access to operate on 32-bit data stored in a 64-bit register pair (’C67x only)

4.3.1 Using Intrinsics

The ’C6x compiler provides intrinsics, special functions that map directly to inlined ’C62x/’C67x instructions, to optimize your C code quickly. All instructions that are not easily expressed in C code are supported as intrinsics. Intrinsics are specified with a leading underscore (_) and are accessed by calling them as you call a function.

For example, saturated addition can be expressed in C code only by writing a multicycle function, such as the one in Example 4–4.

Example 4–4. Saturated Add Without Intrinsics

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

  result = a + b;

  if (((a ^ b) & 0x80000000) == 0)
  {
    if ((result ^ a) & 0x80000000)
    {
      result = (a < 0) ? 0x80000000 : 0x7fffffff;
    }
  }
  return (result);
}
```

This complicated code can be replaced by the _sadd( ) intrinsic, which results in a single ’C6x instruction (see Example 4–5).

Example 4–5. Saturated Add With Intrinsics

```
result = _sadd(a,b);
```

Table 4–2 lists the ’C6x intrinsics. For more information on using intrinsics, see the *TMS320C6x Optimizing C Compiler User’s Guide*.

Table 4–2. TMS320C6x C Compiler Intrinsics

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_abs(**int *src2***);** int**_labs(**long *src2***);**	**ABS**	Returns the saturated absolute value of src2.	
int **_add2(**int *src1*, int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.	
uint **_clr(**uint *src2,* uint *csta*, uint *cstb***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.	
unsigned **_clrr(**uint *src1*, int *src2***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of the source register.	
int**_dpint(**double**)**;	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register.	’C67x
int **_ext(**uint *src2,* uint *csta*, int *cstb***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	
int **_extr(**int *src2*, int *src1***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	

Note: Instructions not specified with a device apply to all ’C6x devices.

Table 4–2. TMS320C6x C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
uint **_extu(**uint *src2,* uint *csta*, uint *cstb***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_extur(**uint *src2*, int *src1***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_ftoi(**float**);**		Reinterprets the bits in the float as an unsigned integer. (Ex: _ftoi(1.0) == 1065353216U)	’C67x
uint **_hi(**double**);**		Returns the high 32 bits of a double as an integer.	’C67x
double **_itod(**uint, uint**);**		Creates a new double register pair from two unsigned integers.	’C67x
float **_itof(**uint**);**		Reinterprets the bits in the unsigned integer as a float. (Ex: _itof(0x3f800000) == 1.0)	’C67x
uint **_lmbd(**uint s*rc1,* uint *src2***);**	**LMBD**	Searches for a leftmost 1 or 0 of *src2* determined by the LSB of *src1*. Returns the number of bits up to the bit change.	
uint **_lo(**double**);**		Returns the low (even) register of a double register pair as an integer.	’C67x
int **_mpy(**int *src1,* int *src2***);** int **_mpyus(**uint *src1,* int *src2***);** int **_mpysu(**int *src1,* uint *src2***);** uint **_mpyu(**uint *src1,* uint *src2***);**	**MPY** **MPYUS** **MPYSU** **MPYU**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyh(**int *src1,* int *src2***);** int **_mpyhus(**uint *src1,* int *src2***);** int **_mpyhsu(**int *src1,* uint *src2***);** uint **_mpyhu(**uint *src1,* uint *src2***);**	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	

Note: Instructions not specified with a device apply to all ’C6x devices.

Table 4–2. TMS320C6x C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_mpyhl(**int *src1,* int *src2***);** int **_mpyhuls(**uint *src1,* int *src2***);** int **_mpyhslu(**int *src1,* uint *src2***);** uint **_mpyhlu(**uint *src1,* uint *src2***);**	**MPYHL** **MPYHULS** **MPYHSLU** **MPYHLU**	Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpylh(**int *src1,* int *src2***);** int **_mpyluhs(**uint *src1,* int *src2***);** int **_mpylshu(**int *src1,* uint *src2***);** uint **_mpylhu(**uint *src1,* uint *src2***);**	**MPYLH** **MPYLUHS** **MPYLSHU** **MPYLHU**	Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
void **_nassert(**int**);**		Generates no code. Tells the optimizer that the expression declared with the assert function is true; this gives a hint to the optimizer as to what optimizations might be valid.	
uint **_norm(**int *src2***);** uint **_lnorm(**long *src2***);**	**NORM**	Returns the number of bits up to the first nonredundant sign bit of src2.	
double **_rcpdp(**double**)**;	**RCPDP**	Computes the approximate 64-bit double reciprocal.	’C67x
float **_rcpsp(**float**)**;	**RCPSP**	Computes the approximate 64-bit double reciprocal.	’C67x
double **_rsqrdp(**double *src***)**;	**RSQRDP**	Computes the approximate 64-bit double reciprocal square root.	’C67x
float **_rsqrsp(**float *src***)**;	**RSQRSP**	Computes the approximate 32-bit float reciprocal square root.	’C67x
int **_sadd(**int *src1,* int *src2***);** long **_lsadd(**int *src1,* long *src2***):**	**SADD**	Adds src1 to src2 and saturates the result. Returns the result.	
int **_sat(**long *src2***);**	**SAT**	Converts a 40-bit value to an 32-bit value and saturates if necessary.	
uint **_set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csta and cstb, respectively.	

Note: Instructions not specified with a device apply to all ’C6x devices.

Table 4–2. TMS320C6x C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
unsigned **_setr(**unsigned, int**);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of the source register.	
int **_smpy(**int *src1*, int *sr2***);** int **_smpyh(**int *src1*, int *sr2***);** int **_smpyhl(**int *src1*, int *sr2***);** int **_smpylh(**int *src1*, int *sr2***);**	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	Multiplies src1 by src2, left shifts the result by one, and returns the result. If the result is 0x8000 0000, saturates the result to 0x7FFF FFFF.	
int **_spint(**float**)**;	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register.	’C67x
uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.	
int **_ssub(**int *src1,* int *src2***);** long **_lssub(**int *src1,* long *src2***):**	**SSUB**	Subtracts src2 from src1, saturates the result size, and returns the result.	
uint **_subc(**uint *src1,* uint *src2***);**	**SUBC**	Conditional subtract divide step.	
int **_sub2(**int *src1*, int *src2***);**	**SUB2**	Subtracts the upper and lower halves of *src2* from the upper and lower halves of *src1*, and returns the result. Any borrowing from the lower half subtract does not affect the upper half subtract.	

Note: Instructions not specified with a device apply to all ’C6x devices.

4.3.2 Using Word Access for Short Data

The 'C6x has instructions with corresponding intrinsics, such as _add2( ), _mpyhl( ), _mpylh( ), that operate on 16-bit data stored in the high and low parts of a 32-bit register. When operating on a stream of short data, you can use word (int) accesses to read two short values at a time, and then use 'C6x intrinsics to operate on the data. For example, rewriting the vecsum( ) function to use word accesses (as in Example 4–6) doubles the performance of the loop. See section 6.3, *Loading Two Data Values with LDW*, on page 6-15 for more information.

Example 4–6. Vector Sum With const Keywords, _nassert, Word Reads

```
void vecsum4(short *sum, const short *in1, const short *in2, unsigned int N)
{
  int i;

   const int *i_in1 = (const int *)in1;
   const int *i_in2 = (const int *)in2;
         int *i_sum = (int *)sum;

  _nassert(N >= 20);

  for (i = 0; i < (N/2); i++)
    i_sum[i] = _add2(i_in1[i], i_in2[i]);
}
```

Note:

The _nassert intrinsic tells the optimizer that the code that follows meets the condition specified.

This transformation assumes that the pointers sum, in1, and in2 can be cast to int *, which means that they must point to word-aligned data. By default, the compiler aligns all short arrays on word boundaries; however, a call like the following creates an illegal memory access:

```
short a[51], b[50], c[50]; vecsum4(&a[1], b, c, 50);
```

Another consideration is that the loop must now run for an even number of iterations. You can ensure that this happens by padding the short arrays so that the loop always operates on an even number of elements.

If a vecsum( ) function is needed to handle short-aligned data and odd-numbered loop counters, then you must add code within the function to check for these cases. Knowing what type of data is passed to a function can improve performance considerably. It may be useful to write different functions that can handle different types of data. If your short-data operations always operate on even-numbered word-aligned arrays, then the performance of your application can be improved. However, Example 4–7 provides a generic vecsum( ) function that handles all types of data.

Example 4–7. Vector Sum With const Keywords, _nassert, Word Reads (Generic Version)

```
void vecsum5(short *sum, const short *in1, const short *in2, unsigned int N)
{
  int i;

  _nassert(N >= 20);

  if (((int)sum | (int)in2 | (int)in1) & 0x2)
  {
    for (i = 0; i < N; i++)
      sum[i] = in1[i] + in2[i];
  }
  else
  {
     const int *i_in1 = (const int *)in1;
     const int *i_in2 = (const int *)in2;
           int *i_sum = (int *)sum;

    for (i = 0; i < (N/2); i++)
      i_sum[i] = _add2(i_in1[i], i_in2[i]);

    if (N & 0x1) sum[i] = in1[i] + in2[i];
  }
}
```

4.3.2.1 Using Word Access in Dot Product

Other intrinsics that are useful for reading short data as words are the multiply intrinsics. Example 4–8 is a dot product example that reads word-aligned short data and uses the _mpy( ) and _mpyh( ) intrinsics. The _mpyh( ) intrinsic uses the ’C6x instruction MPYH, which multiplies the high 16 bits of two registers, giving a 32-bit result.

This example also uses two sum variables (sum1 and sum2). Using only one sum variable inhibits parallelism by creating a dependency between the write from the first sum calculation and the read in the second sum calculation. Within a small loop body, avoid writing to the same variable, because it inhibits parallelism and creates dependencies.

Example 4–8. Dot Product Using Intrinsics

```
int dotprod(const short *a, const short *b, unsigned int N)
{
  int i, sum1 = 0, sum2 = 0;

  const int *i_a = (const int *)a;
  const int *i_b = (const int *)b;

  for (i = 0; i < (N >> 1); i++)
  {
    sum1 = sum1 + _mpy (i_a[i], i_b[i]);
    sum2 = sum2 + _mpyh(i_a[i], i_b[i]);
  }

  return sum1 + sum2;
}
```


4.3.2.2 Using Word Access in FIR Filter

Example 4–9 shows an FIR filter that can be optimized with word reads of short data and multiply intrinsics.

Example 4–9. FIR Filter—Original Form

```
void fir1(const short x[], const short h[], short y[], int n, int m, int s)
{
  int i, j;
  long y0;
  long round = 1L << (s - 1);

  for (j = 0; j < m; j++)
  {
    y0 = round;

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

    y[j] = (int) (y0 >> s);
  }
}
```

Example 4–10 shows an optimized version of Example 4–9. The optimized version passes an int array instead of casting the short arrays to int arrays and, therefore, helps ensure that data passed to the function is word-aligned. Assuming that a prototype is used, each invocation of the function ensures that the input arrays are word-aligned by forcing you to insert a cast or by using int arrays that contain short data.

Example 4–10. FIR Filter—Optimized Form

```
void fir2(const int x[], const int h[], short y[], int n, int m, int s)
{
  int i, j;
  long y0, y1;
  long round = 1L << (s - 1);

  _nassert(m >= 16);
  _nassert(n >= 16);

  for (j = 0; j < (m >> 1); j++)
  {
    y0 = y1 = round;

    for (i = 0; i < (n >> 1); i++)
    {
      y0 += _mpy (x[i + j],   h[i]);
      y0 += _mpyh (x[i + j],   h[i]);
      y1 += _mpyhl(x[i + j],   h[i]);
      y1 += _mpylh(x[i + j + 1], h[i]);
    }

    *y++ = (int)(y0 >> s);
    *y++ = (int)(y1 >> s);
  }
}
```

4.3.2.3 Using Double Word Access for Word Data (’C67x Specific)

The ’C67x architecture has a load double word (LDDW) instruction, which can read 64 bits of data into a register pair. Just like using word acesses to read 2 short data items, double word acesses can be used to read 2 word data items (or 4 short data items). When operating on a stream of float data, you can use double accesses to read 2 float values at a time, and then use intrinsics to operate on the data.

The basic float dot product is shown in Example 4–11. Since the float addition (ADDSP) instruction takes 4 cycles to complete, the minimum kernel size for this loop is 4 cycles. For this version of the loop, a result is completed every 4 cycles.

Example 4–11. Basic Float Dot Product

```
float dotp1(const float a[], const float b[])
{
    int i;
    float sum = 0;

    for (i=0; i<512; i++)
        sum += a[i] * b[i];

    return sum;
}
```

In Example 4–12, the dot product example is rewritten to use double word loads and instrincs are used to extract the high and low 32-bit values contained in the 64-bit double. The _hi() and _lo() instrinsics return integer values, the _itof() intrinsic subverts the C typing system by interpreting an integer value as a float value. In this version of the loop, 2 float results are computed every 4 cycles.

Example 4–12. Float Dot Product Using Intrinsics

```
float dotp2(const double a[], const double b[])
{
    int i;
    float sum0 = 0;
    float sum1 = 0;

    for (i=0; i<512; i+=2)
    {
        sum0 += _itof(_hi(a[i]))   * _itof(_hi(b[i]));
        sum1 += _itof(_lo(a[i+1])) * _itof(_lo(b[i+2]));
    }

    return sum0 + sum1;
}
```

In Example 4–13, the dot product example is unrolled to maximize performance. The preprocessor is used to define convenient macros FHI() and FLO() for accessing the high and low 32-bit values in a double word. In this version of the loop, 8 float values are computed every 4 cycles.

Example 4–13. Float Dot Product With Peak Performance

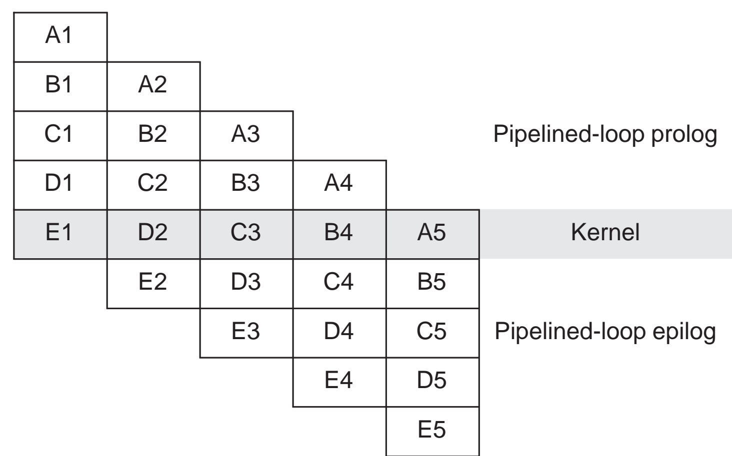
```
#define FHI(a) _itof(_hi(a))
#define FLO(a) _itof(_lo(a))

float dotp3(const double a[], const double b[])
{
    int i;
    float sum0 = 0;
    float sum1 = 0;
    float sum2 = 0;
    float sum3 = 0;
    float sum4 = 0;
    float sum5 = 0;
    float sum6 = 0;
    float sum7 = 0;

    for (i=0; i<512; i+=4)
    {
        sum0 += FHI(a[i])   * FHI(b[i]);
        sum1 += FLO(a[i])   * FLO(b[i]);
        sum2 += FHI(a[i+1]) * FHI(b[i+1]);
        sum3 += FLO(a[i+1]) * FLO(b[i+1]);
        sum4 += FHI(a[i+2]) * FHI(b[i+2]);
        sum5 += FLO(a[i+2]) * FLO(b[i+2]);
        sum6 += FHI(a[i+3]) * FHI(b[i+3]);
        sum7 += FLO(a[i+3]) * FLO(b[i+3]);
    }

    sum0 += sum1;
    sum2 += sum3;
    sum4 += sum5;
    sum6 += sum7;
    sum0 += sum2;
    sum4 += sum6;

    return sum0 + sum4;
}
```

4.3.3 Software Pipelining

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. When you use the –o2 and –o3 compiler options, the compiler attempts to software pipeline your code with information that it gathers from your program.

Figure 4–3 illustrates a software-pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop kernel. In the loop kernel, all five stages execute in parallel. The area immediately before the kernel is known as the pipelined-loop prolog, and the area immediately following the kernel is known as the pipelined-loop epilog.

Figure 4–3. Software-Pipelined Loop



Because loops present critical performance areas in your code, consider the following areas to improve the performance of your C code:

- ❑ Trip count
- ❑ Redundant loops
- ❑ Loop unrolling
- ❑ Speculative execution

4.3.3.1 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 a limit equal to the trip count, the loop terminates. The structure of a software pipeline requires the execution of a minimum number of loop iterations (a minimum trip count) in order to fill, or prime, the pipeline.

Loops that are eligible for software pipelining have loop trip counters that count down. In most cases, the compiler can transform the loop to use a trip counter that counts down even if the original code was not written that way.

For example, the optimizer transforms the loop in Example 4–14(a) to something like the code in Example 4–14(b).

Example 4–14. Trip Counters

(a) Original code

```
for (i = 0; i < N; i++) /* i = trip counter, N = trip count */
```

(b) Optimized code

```
for (i = N; i != 0; i--) /* Downcounting trip counter */
```

The minimum trip count for a software pipelined loop is determined by the minimum number of times the loop will execute.

If the compiler knows the trip count, it can generate faster and more compact code. If the compiler cannot determine that a loop always executes for the minimum trip count, it generates a redundant unpipelined loop. The redundant unpipelined loop is executed only when the runtime trip count is less than the minimum trip count; otherwise, the software-pipelined version of the loop is executed.

4.3.3.2 Eliminating Redundant Loops

In Example 4–2 on page 4-7, the compiler cannot determine if the loop always executes more than the minimum trip count. Therefore, it generates two versions of the loop:

- An unpipelined version that executes if N is less than the minimum trip count
- A software-pipelined version that executes if N is equal to or greater than the minimum trip count

To indicate to the compiler that you do not want two versions of the loop, you can use the –ms option so that the compiler generates only the software-pipelined code and never generates a redundant loop; however, loops with an unknown trip count are not software pipelined.

4.3.3.3 Communicating Trip-Count Information to the Compiler

When invoking the compiler, use the following options to communicate trip-count information to the compiler:

- Use the –o3 and –pm compiler options to allow the optimizer to access the whole program or large parts of it and to characterize the behavior of loop trip counts.
- Use the _nassert intrinsic to help reduce code size by preventing the generation of a redundant loop or by allowing the compiler (with or without the –ms option) to software pipeline innermost loops.

  Example 4–15 shows the vector sum code with an _nassert intrinsic that asserts that N is always at least 10.

Example 4–15. Vector Sum With const Keywords and _nassert

```
void vecsum3(short *sum, const short *in1, const short *in2, unsigned int N)
{
    int i;

    _nassert(N >= 10);

    for (i = 0; i < N; i++)
        sum[i] = in1[i] + in2[i];
}
```

See the *TMS320C6x Optimizing C Compiler User's Guide* for a complete discussion of the –ms, –o3, and –pm options and the _nassert intrinsic.

4.3.3.4 Loop Unrolling

Another technique that improves performance is unrolling the loop; that is, expanding small loops so that each iteration of the loop appears in your code. This optimization increases the number of instructions available to execute in parallel. You can use loop unrolling when the operations in a single iteration do not use all of the resources of the 'C6x architecture.

In Example 4–16, the loop produces a new sum[i] every two cycles. Three memory operations are performed: a load for both in1[i] and in2[i] and a store for sum[i]. Because only two memory operations can execute per cycle, two cycles are necessary to perform three memory operations.

Example 4–16. Vector Sum With Three Memory Operations

```
void vecsum2(short *sum, const short *in1, const short *in2, unsigned int N)
{
  int i;

  for (i = 0; i < N; i++)
    sum[i] = in1[i] + in2[i];
}
```

The performance of a software pipeline is limited by the number of resources that can execute in parallel. In its word-aligned form (Example 4–17), the vector sum loop delivers two results every two cycles because the two loads and the store are all operating on two 16-bit values at a time.

Example 4–17. Word-Aligned Vector Sum

```
void vecsum4(short *sum, const short *in1, const short *in2, unsigned int N)
{
  int i;

   const int *i_in1 = (const int *)in1;
   const int *i_in2 = (const int *)in2;
         int *i_sum = (int *)sum;

  _nassert(N >= 20);

  for (i = 0; i < (N/2); i++)
    i_sum[i] = _add2(i_in1[i], i_in2[i]);
}
```

If you unroll the loop once, the loop then performs six memory operations per iteration, which means the unrolled vector sum loop can deliver four results every three cycles (that is, 1.33 results per cycle). Example 4–18 shows four results for each iteration of the loop: sum[i] and sum[i+sz] each store an int value that represents two 16-bit values.

Example 4–18 is not simple loop unrolling where the loop body is simply replicated. The additional instructions use memory pointers that are offset to point midway into the input arrays and the assumptions that the additional arrays are a multiple of four shorts in size.

Example 4–18. Vector Sum Using const Keywords, _nassert, Word Reads, and Loop Unrolling


```
void vecsum6(int *sum, const int *in1, const int *in2, unsigned int N)
{
  int i;
  int sz = N >> 2;

  _nassert(N >= 20);

  for (i = 0; i < sz; i++)
  {
     sum[i]  = _add2(in1[i],  in2[i]);
     sum[i+sz] = _add2(in1[i+sz], in2[i+sz]);
  }
}
```

Software pipelining is performed by the compiler only on inner loops; therefore, you can increase performance by creating larger inner loops. One method for creating large inner loops is to completely unroll inner loops that execute for a small number of cycles.

In Example 4–19, the compiler pipelines the inner loop with a kernel size of one cycle; therefore, the inner loop completes a result every cycle. However, the overhead of filling and draining the software pipeline can be significant, and other outer-loop code is not software pipelined.

Example 4–19. FIR_Type2—Original Form

```
void fir2(const short input[], const short coefs[], short out[])
{
  int i, j;
  int sum = 0;

  for (i = 0; i < 40; i++)
  {
    for (j = 0; j < 16; j++)
      sum += coefs[j] * input[i + 15 - j];

    out[i] = (sum >> 15);
  }
}
```

For loops with a simple loop structure, the compiler uses a heuristic to determine if it should unroll the loop. Because unrolling can increase code size, in some cases the compiler does not unroll the loop. If you have identified this loop as being critical to your application, then unroll the inner loop in C code, as in Example 4–20.

Example 4–20. FIR_Type2—Inner Loop Completely Unrolled

```
void fir2_u(const short input[], const short coefs[], short out[])
{
  int i, j;
  int sum;

  for (i = 0; i < 40; i++)
  {
    sum = coefs[0] * input[i + 15];
    sum += coefs[1] * input[i + 14];
    sum += coefs[2] * input[i + 13];
    sum += coefs[3] * input[i + 12];
    sum += coefs[4] * input[i + 11];
    sum += coefs[5] * input[i + 10];
    sum += coefs[6] * input[i + 9];
    sum += coefs[7] * input[i + 8];
    sum += coefs[8] * input[i + 7];
    sum += coefs[9] * input[i + 6];
    sum += coefs[10] * input[i + 5];
    sum += coefs[11] * input[i + 4];
    sum += coefs[12] * input[i + 3];
    sum += coefs[13] * input[i + 2];
    sum += coefs[14] * input[i + 1];
    sum += coefs[15] * input[i + 0];

    out[i] = (sum >> 15);
  }
}
```

Now the outer loop is software-pipelined, and the overhead of draining and filling the software pipeline occurs only once per invocation of the function rather than for each iteration of the outer loop.

4.3.3.5 Speculative Execution (–mh option)

The –mh option eliminates the epilog for a software pipelined loop, which can result in significant code size savings. Software pipelined loop epilogs can often be eliminated if load instructions can be speculatively executed. An instruction is speculatively executed if it is executed before it is known whether the result of the instruction is needed. Allowing speculative execution of load instructions may result in a read past the beginning or end of a buffer. For a complete discussion on the –mh option see the *TMS320C6x Optimizing C Compiler User's Guide*.

4.3.3.6 What Disqualifies a Loop from Being Software-Pipelined

In a sequence of nested loops, the innermost loop is the only one that can be software-pipelined. The following restrictions apply to the software pipelining of loops:

- Although a software-pipelined loop can contain intrinsics, it cannot contain function calls.
- You must not have a conditional break (early exit) in the loop.
- The loop cannot have an incrementing loop counter. One reason that you run the optimizer with the –o2 or –o3 option is to convert as many loops as possible into downcounting loops.
- If the trip counter is modified within the body of the loop, it typically cannot be converted into a downcounting loop. For example, the following code is not software-pipelined:

```
for (i = 0; i < n; i++)
{
 ...
 i += x;
}
```

- A conditionally incremented loop control variable is not software-pipelined. For example, the following code is not software-pipelined:

```
for (i = 0; i < x; i++)
{
  ...
  if (b > a)
       i += 2
}
```

- If the code size is too large and requires more than the 32 registers in the 'C6x, it is not software-pipelined.
- If a register value is live too long, the code is not software-pipelined. See section 6.5.6.2, *Live Too Long*, on page 6-63 and section 6.9, *Live-Too-Long Issues*, on page 6-97 for examples of code that is live too long.
- If the loop has complex condition code within the body that requires more than the five 'C6x condition registers, the loop is not software pipelined.

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Chapter 5

Structure of Assembly Code

An assembly language program must be an ASCII text file. Any line of assembly code can include up to seven items:

- Label
- Parallel bars
- Conditions
- Instruction
- Functional unit
- Operands
- Comment

Topic **Page**

5.1 Labels 5-2
5.2 Parallel Bars 5-2
5.3 Conditions 5-3
5.4 Instructions 5-4
5.5 Functional Units 5-6
5.6 Operands 5-8
5.7 Comments 5-9

Part III

5.1 Labels

A label identifies a line of code or a variable and represents a memory address that contains either an instruction or data.

Figure 5–1 shows the position of the label in a line of assembly code. The colon following the label is optional.

Figure 5–1. Labels in Assembly Code

label:	parallel bars	[condition]	instruction	unit	operands	; comments

Labels must meet the following conditions:

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

5.2 Parallel Bars

An instruction that executes in parallel with the previous instruction signifies this with parallel bars (||). This field is left blank for an instruction that does not execute in parallel with the previous instruction.

Figure 5–2. Parallel Bars in Assembly Code

label:	**parallel bars**	[condition]	instruction	unit	operands	; comments

5.3 Conditions

Five registers in the ’C6x are available for conditions: A1, A2, B0, B1, and B2. Figure 5–3 shows the position of a condition in a line of assembly code.

Figure 5–3. Conditions in Assembly Code

label:	parallel bars	**[condition]**	instruction	unit	operands	; comments

All ’C6x instructions are conditional:

- If no condition is specified, the instruction is always performed.
- If a condition is specified and that condition is true, the instruction executes. For example:

With this condition ...	The instruction executes if ...
`[A1]`	A1 ! = 0
`[!A1]`	A1 = 0

- If a condition is specified and that condition is false, the instruction does not execute.

With this condition ...	The instruction does not execute if ...
`[A1]`	A1 = 0
`[!A1]`	A1! = 0

Part III

5.4 Instructions

Assembly code instructions are either directives or mnemonics:

- *Assembler directives* are commands for the assembler (asm6x) that control the assembly process or define the data structures (constants and variables) in the assembly language program. All assembler directives begin with a period, as shown in the partial list in Table 5–1.
- *Processor mnemonics* are the actual microprocessor instructions that execute at runtime and perform the operations in the program. Table 5–2 summarizes the 'C6x mnemonics. Processor mnemonics must begin in column 2 or greater.

Figure 5–4 shows the position of the instruction in a line of assembly code.

Figure 5–4. Instructions in Assembly Code

label:	parallel bars	[condition]	**instruction**	unit	operands	; comments

Table 5–1. Selected TMS320C6x Directives

Directives	Description
.sect "*name*"	Creates section of information (data or code)
.double *value*	Reserve two consecutive 32 bits (64 bits) in memory and fill with double-precision (64-bit) IEEE floating-point representation of specified value
.float *value*	Reserve 32 bits in memory and fill with single-precision (32-bit) IEEE floating-point representation of specified value
.int *value* .long *value* .word *value*	Reserve 32 bits in memory and fill with specified value
.short *value* .half *value*	Reserve 16 bits in memory and fill with specified value
.byte *value*	Reserve 8 bits in memory and fill with specified value

See the *TMS320C6x Assembly Language Tools User's Guide* for a complete list of directives.

Part III

Table 5–2. Selected TMS320C6x Instruction Mnemonics

Arithmetic	Multiply	Load/Store	Program Control	Bit Management	Logical	Pseudo/Other
ABS	MPY	LD	B	CLR	AND	IDLE
ADD	MPYDP†	LDDW†	B IRP	EXT	CMPEQ	MV
ADDA	MPYH	MVK	B NRP	LMBD	CMPEQDP†	MVC
ADDK	MPYHL	MVKH		NORM	CMPEQSP†	NOP
ADDP†	MPYI†	ST		SET	CMPGT	ZERO
ADDSP†	MPYID†				CMPGTDP†	NEG
ADD2	MPYLH				CMPGTSP†	NOT
DPINT†	MPYSP†				CMPLT	
DPSP†	SMPY				CMPLTDP†	
DPTRUNC†					CMPLTSP†	
INTDP†					OR	
INTSP†					SHL	
RCPDP†					SHR	
RCPSP†					SSHL	
RSQRDP†					XOR	
RSQRSP†						
SADD						
SAT						
SPDP†						
SPINT†						
SPTRUNC†						
SSUB						
SUB						
SUBA						
SUBC						
SUBDP†						
SUBSP†						
SUB2						

† 'C67x instruction mnemonics only

See the *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for a complete list of instructions.

5.5 Functional Units

The ’C6x CPU contains eight functional units, which are shown in Figure 5–5 and described in Table 5–3.

Figure 5–5. TMS320C6x Functional Units

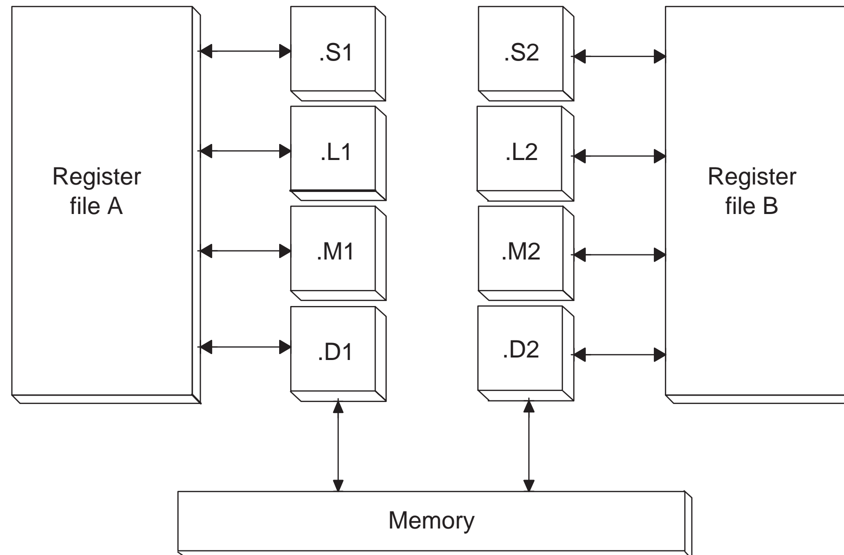


Table 5–3. Functional Units and Descriptions

Functional Unit	Description
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations Left most 1, 0, bit counting for 32 bits Normalization count for 32 and 40 bits 32 bit logical operations 32/64-bit IEEE floating-point arithmetic† Floating-point/fixed-point conversions†
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40 bit shifts and 32-bit bit-field operations 32 bit logical operations Branching Constant generation Register transfers to/from the control register file 32/64-bit IEEE floating-point compare operations† 32/64-bit IEEE floating-point reciprocal and square root reciprocal approximation†
.M unit (.M1, .M2)	16 $\times$16 bit multiplies 32 $\times$ 32-bit multiplies† Single-precision (32-bit) floating-point IEEE multiplies† Double-precision (64-bit) floating-point IEEE multiplies†
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation

† ’C67x floating-point devices only

Figure 5–6 shows the position of the unit in a line of assembly code.

Figure 5–6. Units in the Assembly Code

label:	parallel bars	[condition]	instruction	**unit**	operands	; comments

Specifying the functional unit in the assembly code is optional. The functional unit can be used to document which resource(s) each instruction uses.

5.6 Operands

The ’C6x architecture requires that memory reads and writes move data between memory and a register. Figure 5–7 shows the position of the operands in a line of assembly code.

Figure 5–7. Operands in the Assembly Code

label:	parallel bars	[condition]	instruction	unit	**operands**	; comments

Instructions have the following requirements for operands in the assembly code:

- ❑ All instructions require a destination operand.
- ❑ Most instructions require one or two source operands.
- ❑ The destination operand must be in the same register file as one source operand.
- ❑ One source operand from each register file per execute packet can come from the register file opposite that of the other source operand.

  When an operand comes from the other register file, the unit includes an X, as shown in Figure 5–8, indicating that the instruction is using one of the cross paths. (See the *TMS320C6x CPU and Instruction Set Reference Guide* for more information on cross paths.)

Figure 5–8. Operands in Instructions

```
ADD     .L1     A0,A1,A3
ADD     .L1X    A0,B1,A3
```

All registers except B1 are on the same side of the CPU.

The ’C6x instructions use three types of operands to access data:

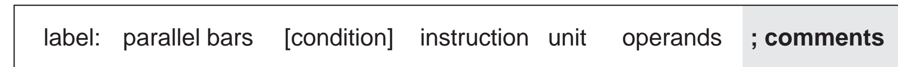
- ❑ Register operands indicate a register that contains the data.
- ❑ Constant operands specify the data within the assembly code.
- ❑ Pointer operands contain addresses of data values.

  Only the load and store instructions require and use pointer operands to move data values between memory and a register.

5.7 Comments

As with all programming languages, comments provide code documentation. Figure 5–9 shows the position of the comment in a line of assembly code.

Figure 5–9. Comments in Assembly Code



The following are guidelines for using comments in assembly code:

- A comment may begin in any column when preceded by a semicolon (;).
- A comment must begin in first column when preceded by an asterisk (*).
- Comments are not required but are recommended.

Chapter 6

Optimizing Assembly Code via Linear Assembly

This chapter describes methods that help you develop more efficient assembly language programs, understand the code produced by the assembly optimizer, and perform manual optimization.

This chapter encompasses phase 3 of the code development flow. After you have developed and optimized your C code using the 'C6x compiler, extract the inefficient areas from your C code and rewrite them in linear assembly (assembly code that has not been register-allocated and is unscheduled).

The assembly code shown in this chapter has been hand-optimized in order to direct your attention to particular coding issues. The actual output from the assembly optimizer may look different, depending on the version you are using.

Topic **Page**

6.1 Assembly Code 6-2

6.2 Writing Parallel Code 6-4

6.3 Using Word Access for Short Data and Doubleword Access for Floating-Point Data 6-15

6.4 Software Pipelining 6-25

6.5 Modulo Scheduling of Multicycle Loops 6-54

6.6 Loop Carry Paths 6-74

6.7 If-Then-Else Statements in a Loop 6-83

6.8 Loop Unrolling 6-91

6.9 Live-Too-Long Issues 6-98

6.10 Redundant Load Elimination 6-107

6.11 Memory Banks 6-115

6.12 Software Pipelining the Outer Loop 6-128

6.13 Outer Loop Conditionally Executed With Inner Loop 6-133

6.1 Assembly Code

The source that you write for the assembly optimizer is similar to assembly source code; however, linear assembly does not include information about parallel instructions, instruction latencies, or register usage. The assembly optimizer takes care of the difficulties of streamlining your code by:

- Finding instructions that can be executed in parallel
- Handling pipeline latencies during software pipelining
- Assigning register usage
- Defining which unit to use

Although you have the option with the ’C6x to specify the functional unit or register used, this may restrict the compiler’s ability to fully optimize your code. See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information.

This chapter takes you through the optimization process manually to show you how the assembly optimizer works and to help you understand when you might want to perform some of the optimizations manually. Each section introduces optimization techniques in increasing complexity:

- Section 6.2 and section 6.3 begin with a dot product algorithm to show you how to translate the C code to assembly code and then how to optimize the linear assembly code with several simple techniques.
- Section 6.4 and section 6.5 introduce techniques for the more complex algorithms associated with software pipelining, such as modulo iteration interval scheduling for both single-cycle loops and multicycle loops.
- Section 6.6 uses an IIR filter algorithm to discuss the problems with loop carry paths.
- Section 6.7 and section 6.8 discuss the problems encountered with if-then-else statements in a loop and how loop unrolling can be used to resolve them.
- Section 6.9 introduces live-too-long issues in your code.
- Section 6.10 uses a simple FIR filter algorithm to discuss redundant load elimination.
- Section 6.11 discusses the same FIR filter in terms of the interleaved memory bank scheme used by ’C6x devices.
- Section 6.12 and section 6.13 show you how to execute the outer loop of the FIR filter conditionally and in parallel with the inner loop.

Each example discusses the:

- Algorithm in C code
- Translation of the C code to linear assembly
- Dependency graph to describe the flow of data in the algorithm
- Allocation of resources (functional units, registers, and cross paths) in linear assembly

Note:

There are three types of code for the ’C6x: C code (which is input for the C compiler), linear assembly code (which is input for the assembly optimizer), and assembly code (which is input for the assembler).

In the next three sections, we use the dot product to demonstrate how to use various programming techniques to optimize both performance and code size. Most of the examples provided in this book use fixed-point arithmetic; however, the next three sections give both fixed-point and floating-point examples of the dot product to show that the same optimization techniques apply to both fixed- and floating-point programs.

6.2 Writing Parallel Code

One way to optimize linear assembly code is to reduce the number of execution cycles in a loop. You can do this by rewriting linear assembly instructions so that the final assembly instructions execute in parallel.

6.2.1 Dot Product C Code

The dot product is a sum in which each element in array *a* is multiplied by the corresponding element in array *b*. Each of these products is then accumulated into *sum*. The C code in Example 6–1 is a fixed-point dot product algorithm. The C code in Example 6–2 is a floating-point dot product algorithm.

Example 6–1. Fixed-Point Dot Product C Code

```
int dotp(short a[], short b[])
{
   int sum, i;
   sum = 0;

   for(i=0; i<100; i++)
      sum += a[i] * b[i];

   return(sum);
}
```

Example 6–2. Floating-Point Dot Product C Code

```
float dotp(float a[], float b[])
{
   int i;
   float sum;
   sum = 0;

   for(i=0; i<100; i++)
      sum += a[i] * b[i];

   return(sum);
}
```

6.2.2 Translating C Code to Linear Assembly

The first step in optimizing your code is to translate the C code to linear assembly.

6.2.2.1 Fixed-Point Dot Product

Example 6–3 shows the linear assembly instructions used for the inner loop of the fixed-point dot product C code.

Example 6–3. List of Assembly Instructions for Fixed-Point Dot Product

```
        LDH     .D1     *A4++,A2        ; load ai from memory
        LDH     .D1     *A3++,A5        ; load bi from memory
        MPY     .M1     A2,A5,A6        ; ai * bi
        ADD     .L1     A6,A7,A7        ; sum += (ai * bi)
        SUB     .S1     A1,1,A1         ; decrement loop counter
   [A1] B       .S2     LOOP            ; branch to loop
```

The load halfword (LDH) instructions increment through the *a* and *b* arrays. Each LDH does a postincrement on the pointer. Each iteration of these instructions sets the pointer to the next halfword (16 bits) in the array. The ADD instruction accumulates the total of the results from the multiply (MPY) instruction. The subtract (SUB) instruction decrements the loop counter.

An additional instruction is included to execute the branch back to the top of the loop. The branch (B) instruction is conditional on the loop counter, A1, and executes only until A1 is 0.

6.2.2.2 Floating-Point Dot Product

Example 6–4 shows the linear assembly instructions used for the inner loop of the floating-point dot product C code.

Example 6–4. List of Assembly Instructions for Floating-Point Dot Product

```
        LDW     .D1     *A4++,A2        ; load ai from memory
        LDW     .D2     *A3++,A5        ; load bi from memory
        MPYSP†  .M1     A2,A5,A6        ; ai * bi
        ADDSP†  .L1     A6,A7,A7        ; sum += (ai * bi)
        SUB     .S1     A1,1,A1         ; decrement loop counter
   [A1] B       .S2     LOOP            ; branch to loop
```

† ADDSP and MPYSP are ’C67x (floating-point) instructions only.

The load word (LDW) instructions increment through the *a* and *b* arrays. Each LDW does a postincrement on the pointer. Each iteration of these instructions sets the pointer to the next word (32 bits) in the array. The ADDSP instruction

accumulates the total of the results from the multiply (MPYSP) instruction. The subtract (SUB) instruction decrements the loop counter.

An additional instruction is included to execute the branch back to the top of the loop. The branch (B) instruction is conditional on the loop counter, A1, and executes only until A1 is 0.

6.2.3 Linear Assembly Resource Allocation

The following rules affect the assignment of functional units for Example 6–3 and Example 6–4 (shown in the third column of each example):

- Load (LDH and LDW) instructions must use a .D unit.
- Multiply (MPY and MPYSP) instructions must use a .M unit.
- Add (ADD and ADDSP) instructions use a .L unit.
- Subtract (SUB) instructions use a .S unit.
- Branch (B) instructions must use a .S unit.

Note:

The ADD and SUB can be on the .S, .L, or .D units; however, for Example 6–3 and Example 6–4, they are assigned as listed above.

The ADDSP instruction in Example 6–4 must use a .L unit.

6.2.4 Drawing a Dependency Graph

Dependency graphs can help analyze loops by showing the flow of instructions and data in an algorithm. These graphs also show how instructions depend on one another. The following terms are used in defining a dependency graph.

- A *node* is a point on a dependency graph with one or more data paths flowing in and/or out.
- The *path* shows the flow of data between nodes. The numbers beside each path represent the number of cycles required to complete the instruction.
- An instruction that writes to a variable is referred to as a parent instruction and defines a *parent node*.
- An instruction that reads a variable written by a parent instruction is referred to as its child and defines a *child node*.

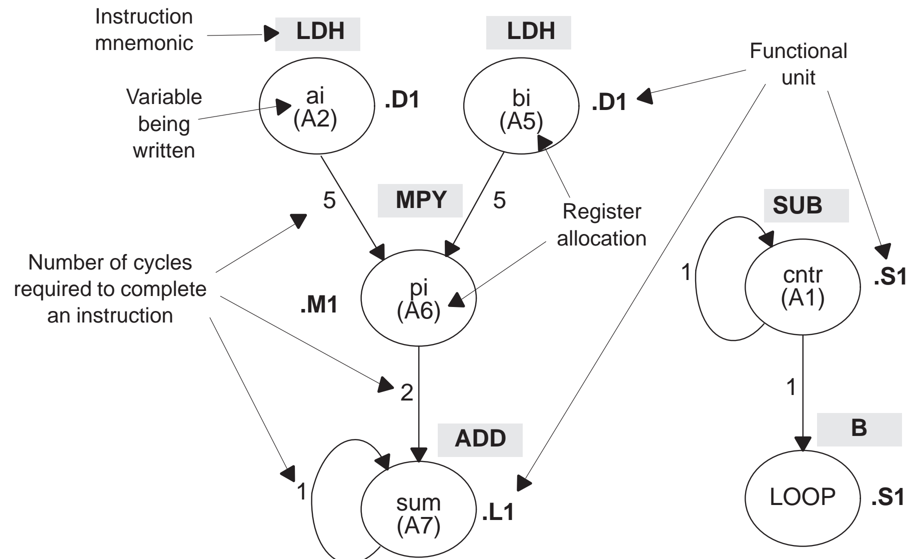
Use the following steps to draw a dependency graph:

1) Define the nodes based on the variables accessed by the instructions.
2) Define the data paths that show the flow of data between nodes.
3) Add the instructions and the latencies.
4) Add the functional units.

6.2.4.1 Fixed-Point Dot Product

Figure 6–1 shows the dependency graph for the fixed-point dot product assembly instructions shown in Example 6–3 and their corresponding register allocations.

Figure 6–1. Dependency Graph of Fixed-Point Dot Product



- ❑ The two LDH instructions, which write the values of ai and bi, are parents of the MPY instruction. It takes five cycles for the parent (LDH) instruction to complete. Therefore, if LDH is scheduled on cycle i, then its child (MPY) cannot be scheduled until cycle i + 5.

- ❑ The MPY instruction, which writes the product pi, is the parent of the ADD instruction. The MPY instruction takes two cycles to complete.

- ❑ The ADD instruction adds pi (the result of the MPY) to sum. The output of the ADD instruction feeds back to become an input on the next iteration and, thus, creates a *loop carry* path. (See section 6.6 on page 6-74 for more information on loop carry paths.)

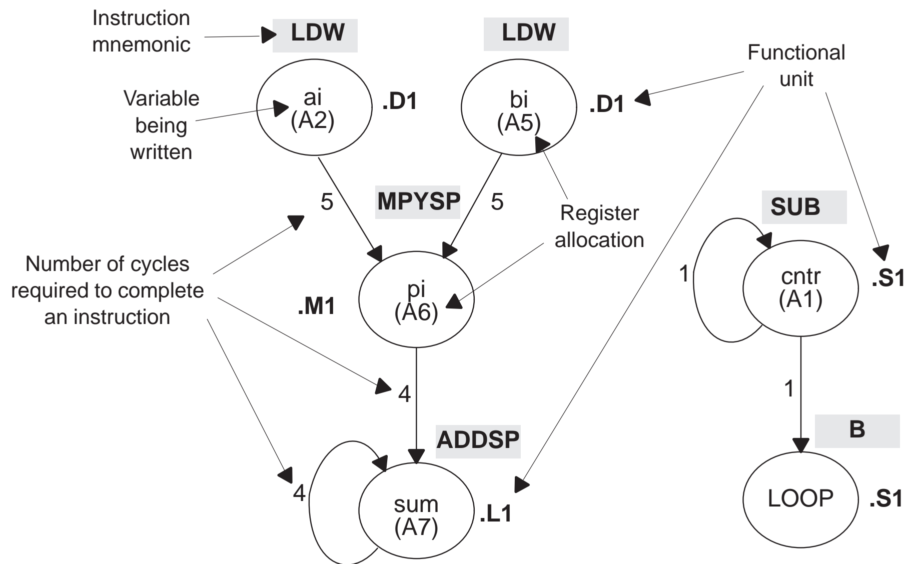
The dependency graph for this dot product algorithm has two separate parts because the decrement of the loop counter and the branch do not read or write any variables from the other part.

- The SUB instruction writes to the loop counter, cntr. The output of the SUB instruction feeds back and creates a loop carry path.
- The branch (B) instruction is a child of the loop counter.

6.2.4.2 Floating-Point Dot Product

Similarly, Figure 6–2 shows the dependency graph for the floating-point dot product assembly instructions shown in Example 6–4 and their corresponding register allocations.

Figure 6–2. Dependency Graph of Floating-Point Dot Product



- The two LDW instructions, which write the values of ai and bi, are parents of the MPYSP instruction. It takes five cycles for the parent (LDW) instruction to complete. Therefore, if LDW is scheduled on cycle i, then its child (MPYSP) cannot be scheduled until cycle i + 5.
- The MPYSP instruction, which writes the product pi, is the parent of the ADDSP instruction. The MPYSP instruction takes four cycles to complete.
- The ADDSP instruction adds pi (the result of the MPYSP) to sum. The output of the ADDSP instruction feeds back to become an input on the next iteration and, thus, creates a *loop carry* path. (See section 6.6 on page 6-74 for more information on loop carry paths.)

The dependency graph for this dot product algorithm has two separate parts because the decrement of the loop counter and the branch do not read or write any variables from the other part.

- The SUB instruction writes to the loop counter, cntr. The output of the SUB instruction feeds back and creates a loop carry path.
- The branch (B) instruction is a child of the loop counter.

6.2.5 Nonparallel Versus Parallel Assembly Code

Nonparallel assembly code is performed serially, that is, one instruction following another in sequence. This section explains how to rewrite the instructions so that they execute in parallel.

6.2.5.1 Fixed-Point Dot Product

Example 6–5 shows the nonparallel assembly code for the fixed-point dot product loop. The MVK instruction initializes the loop counter to 100. The ZERO instruction clears the accumulator. The NOP instructions allow for the delay slots of the LDH, MPY, and B instructions.

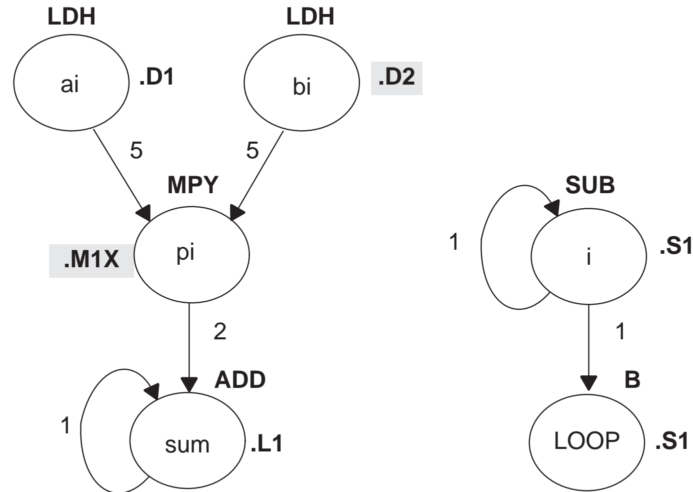
Executing this dot product code serially requires 16 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator; 100 iterations require 1602 cycles.

Example 6–5. Nonparallel Assembly Code for Fixed-Point Dot Product

```
       MVK    .S1    100, A1       ; set up loop counter
       ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDH    .D1    *A4++,A2      ; load ai from memory
       LDH    .D1    *A3++,A5      ; load bi from memory
       NOP    4                    ; delay slots for LDH
       MPY    .M1    A2,A5,A6      ; ai * bi
       NOP                         ; delay slot for MPY
       ADD    .L1    A6,A7,A7      ; sum += (ai * bi)
       SUB    .S1    A1,1,A1       ; decrement loop counter
  [A1] B      .S2    LOOP          ; branch to loop
       NOP    5                    ; delay slots for branch
;  Branch occurs here
```

Assigning the same functional unit to both LDH instructions slows performance of this loop. Therefore, reassign the functional units to execute the code in parallel, as shown in the dependency graph in Figure 6–3. The parallel assembly code is shown in Example 6–6.

Figure 6–3. Dependency Graph of Fixed-Point Dot Product with Parallel Assembly



Example 6–6. Parallel Assembly Code for Fixed-Point Dot Product

```
       MVK    .S1    100, A1       ; set up loop counter
||     ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDH    .D1    *A4++,A2      ; load ai from memory
||     LDH    .D2    *B4++,B2      ; load bi from memory
       SUB    .S1    A1,1,A1       ; decrement loop counter
  [A1] B      .S2    LOOP          ; branch to loop
       NOP    2                    ; delay slots for LDH
       MPY    .M1X   A2,B2,A6      ; ai * bi
       NOP                         ; delay slots for MPY
       ADD    .L1    A6,A7,A7      ; sum += (ai * bi)
;  Branch occurs here
```

Because the loads of ai and bi do not depend on one another, both LDH instructions can execute in parallel as long as they do not share the same resources. To schedule the load instructions in parallel, allocate the functional units as follows:

- ai and the pointer to ai to a functional unit on the A side, .D1
- bi and the pointer to bi to a functional unit on the B side, .D2

Because the MPY instruction now has one source operand from A and one from B, MPY uses the 1X cross path.

Rearranging the order of the instructions also improves the performance of the code. The SUB instruction can take the place of one of the NOP delay slots for the LDH instructions. Moving the B instruction after the SUB removes the need for the NOP 5 used at the end of the code in Example 6–5.

The branch now occurs immediately after the ADD instruction so that the MPY and ADD execute in parallel with the five delay slots required by the branch instruction.

6.2.5.2 Floating-Point Dot Product

Similarly, Example 6–7 shows the nonparallel assembly code for the floating-point dot product loop. The MVK instruction initializes the loop counter to 100. The ZERO instruction clears the accumulator. The NOP instructions allow for the delay slots of the LDW, ADDSP, MPYSP, and B instructions.

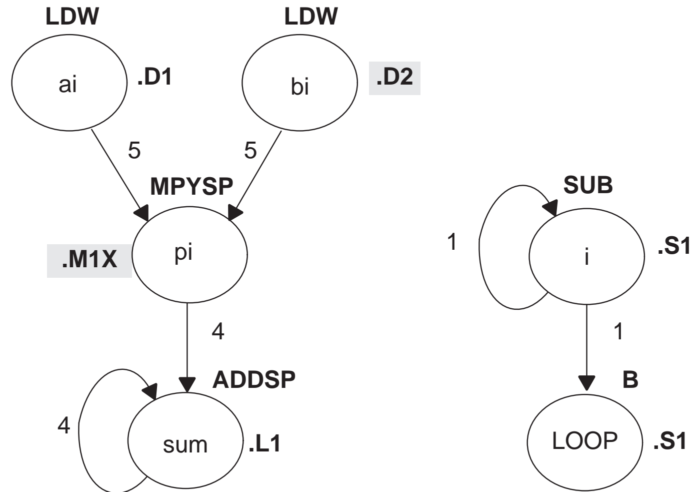
Executing this dot product code serially requires 21 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator; 100 iterations require 2102 cycles.

Example 6–7. Nonparallel Assembly Code for Floating-Point Dot Product

```
       MVK    .S1    100, A1       ; set up loop counter
       ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDW    .D1    *A4++,A2      ; load ai from memory
       LDW    .D1    *A3++,A5      ; load bi from memory
       NOP    4                    ; delay slots for LDW
       MPYSP  .M1    A2,A5,A6      ; ai * bi
       NOP    3                    ; delay slots for MPYSP
       ADDSP  .L1    A6,A7,A7      ; sum += (ai * bi)
       NOP    3                    ; delay slots for ADDSP
       SUB    .S1    A1,1,A1       ; decrement loop counter
  [A1] B      .S2    LOOP          ; branch to loop
       NOP    5                    ; delay slots for branch
;  Branch occurs here
```

Assigning the same functional unit to both LDW instructions slows performance of this loop. Therefore, reassign the functional units to execute the code in parallel, as shown in the dependency graph in Figure 6–4. The parallel assembly code is shown in Example 6–8.

Figure 6–4. Dependency Graph of Floating-Point Dot Product with Parallel Assembly



Example 6–8. Parallel Assembly Code for Floating-Point Dot Product

```
       MVK    .S1    100, A1       ; set up loop counter
||     ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDW    .D1    *A4++,A2      ; load ai from memory
||     LDW    .D2    *B4++,B2      ; load bi from memory
       SUB    .S1    A1,1,A1       ; decrement loop counter
       NOP    2                    ; delay slots for LDW
  [A1] B      .S2    LOOP          ; branch to loop
       MPYSP  .M1X   A2,B2,A6      ; ai * bi
       NOP    3                    ; delay slots for MPYSP
       ADDSP  .L1    A6,A7,A7      ; sum += (ai * bi)
;  Branch occurs here
```


Because the loads of ai and bi do not depend on one another, both LDW instructions can execute in parallel as long as they do not share the same resources. To schedule the load instructions in parallel, allocate the functional units as follows:

- ai and the pointer to ai to a functional unit on the A side, .D1
- bi and the pointer to bi to a functional unit on the B side, .D2

Because the MPYSP instruction now has one source operand from A and one from B, MPYSP uses the 1X cross path.

Rearranging the order of the instructions also improves the performance of the code. The SUB instruction replaces one of the NOP delay slots for the LDW instructions. Moving the B instruction after the SUB removes the need for the NOP 5 used at the end of the code in Example 6–7 on page 6-12.

The branch now occurs immediately after the ADDSP instruction so that the MPYSP and ADDSP execute in parallel with the five delay slots required by the branch instruction.

Since the ADDSP finishes execution before the result is needed, the NOP 3 for delay slots is removed, further reducing cycle count.

6.2.6 Comparing Performance

Executing the fixed-point dot product code in Example 6–6 requires eight cycles for each iteration plus one cycle to set up the loop counter and initialize the accumulator; 100 iterations require 801 cycles.

Table 6–1 compares the performance of the nonparallel code with the parallel code for the fixed-point example.

Table 6–1. Comparison of Nonparallel and Parallel Assembly Code for Fixed-Point Dot Product

Code Example		100 Iterations	Cycle Count
Example 6–5	Fixed-point dot product nonparallel assembly	$2 + 100 \times 16$	1602
Example 6–6	Fixed-point dot product parallel assembly	$1 + 100 \times 8$	801

Executing the floating-point dot product code in Example 6–8 requires ten cycles for each iteration plus one cycle to set up the loop counter and initialize the accumulator; 100 iterations require 1001 cycles.

Table 6–2 compares the performance of the nonparallel code with the parallel code for the floating-point example.

Table 6–2. Comparison of Nonparallel and Parallel Assembly Code for Floating-Point Dot Product

Code Example		100 Iterations	Cycle Count
Example 6–7	Floating-point dot product nonparallel assembly	$2 + 100 \times 21$	2102
Example 6–8	Floating-point dot product parallel assembly	$1 + 100 \times 10$	1001

6.3 Using Word Access for Short Data and Doubleword Access for Floating-Point Data

The parallel code for the fixed-point example in section 6.2 uses an LDH instruction to read a[i]. Because a[i] and a[i+1] are next to each other in memory, you can optimize the code further by using the load word (LDW) instruction to read a[i] and a[i+1] at the same time and load both into a single 32-bit register. (The data must be word-aligned in memory.)

In the floating-point example, the parallel code uses an LDW instruction to read a[i]. Because a[i] and a[i+1] are next to each other in memory, you can optimize the code further by using the load doubleword (LDDW) instruction to read a[i] and a[i+1] at the same time and load both into a register pair. (The data must be doubleword-aligned in memory.) See the *TMS320C62x/C67x CPU and Instruction Set User's Guide* for more specific information on the LDDW instruction.

Note:

The load doubleword (LDDW) instruction is only available on the 'C67x (floating-point) device.

6.3.1 Unrolled Dot Product C Code

The fixed-point C code in Example 6–9 has the effect of unrolling the loop by accumulating the even elements, a[i] and b[i], into sum0 and the odd elements, a[i+1] and b[i+1], into sum1. After the loop, sum0 and sum1 are added to produce the final sum. The same is true for the floating-point C code in Example 6–10. (For another example of loop unrolling, see section 6.8 on page 6-91.)

Example 6–9. Fixed-Point Dot Product C Code (Unrolled)

```
int dotp(short a[], short b[] )
{
  int sum0, sum1, sum, i;

  sum0 = 0;
  sum1 = 0;
  for(i=0; i<100; i+=2){
       sum0 += a[i] * b[i];
       sum1 += a[i + 1] * b[i + 1];
       }
  sum = sum0 + sum1;
  return(sum);
}
```

Part III

Example 6–10. Floating-Point Dot Product C Code (Unrolled)

```
float dotp(float a[], float b[])
{
  int i;
  float sum0, sum1, sum;
  sum0 = 0;
  sum1 = 0;
  for(i=0; i<100; i+=2){
       sum0 += a[i] * b[i];
       sum1 += a[i + 1] * b[i + 1];
       }
  sum = sum0 + sum1;
  return(sum);
}
```

6.3.2 Translating C Code to Linear Assembly

The first step in optimizing your code is to translate the C code to linear assembly.

6.3.2.1 Fixed-Point Dot Product

Example 6–11 shows the list of ’C6x instructions that execute the unrolled fixed-point dot product loop. Symbolic variable names are used instead of actual registers. Using symbolic names for data and pointers makes code easier to write and allows the optimizer to allocate registers. However, you must use the .reg assembly optimizer directive. See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information on writing linear assembly code.

Example 6–11. Linear Assembly for Fixed-Point Dot Product Inner Loop with LDW

```
        LDW      *a++,ai_i1       ; load ai & a1 from memory
        LDW      *b++,bi_i1       ; load bi & b1 from memory
        MPY      ai_i1,bi_i1,pi   ; ai * bi
        MPYH     ai_i1,bi_i1,pi1 ; ai+1 * bi+1
        ADD      pi,sum0,sum0     ; sum0 += (ai * bi)
        ADD      pi1,sum1,sum1    ; sum1 += (ai+1 * bi+1)
 [cntr] SUB      cntr,1,cntr      ; decrement loop counter
 [cntr] B        LOOP             ; branch to loop
```

The two load word (LDW) instructions load a[i], a[i+1], b[i], and b[i+1] on each iteration.

Two MPY instructions are now necessary to multiply the second set of array elements:

- The first MPY instruction multiplies the 16 least significant bits (LSBs) in each source register: a[i] $\times$ b[i].
- The MPYH instruction multiplies the 16 most significant bits (MSBs) of each source register: a[i+1] $\times$ b [i+1].

The two ADD instructions accumulate the sums of the even and odd elements: sum0 and sum1.

Note:

This is true only when the ’C6x is in little-endian mode. In big-endian mode, MPY operates on a[i+1] and b[i+1] and MPYH operates on a[i] and b[i]. See the *TMS320C62x/C67x Peripherals Reference Guide* for more information.

6.3.2.2 Floating-Point Dot Product

Example 6–12 shows the list of ’C6x instructions that execute the unrolled floating-point dot product loop. Symbolic variable names are used instead of actual registers. Using symbolic names for data and pointers makes code easier to write and allows the optimizer to allocate registers. However, you must use the .reg assembly optimizer directive. See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information on writing linear assembly code.

Example 6–12. Linear Assembly for Floating-Point Dot Product Inner Loop with LDDW

```
        LDDW     *a++,ai1:ai0    ; load a[i+0] & a[i+1] from memory
        LDDW     *b++,bi1:bi0    ; load b[i+0] & b[i+1] from memory
        MPYSP    ai0,bi0,pi0     ; a[i+0] * b[i+0]
        MPYSP    ai1,bi1,pi1     ; a[i+1] * b[i+1]
        ADDSP    pi0,sum0,sum0   ; sum0 += (a[i+0] * b[i+0])
        ADDSP    pi1,sum1,sum1   ; sum1 += (a[i+1] * b[i+1])
 [cntr] SUB      cntr,1,cntr     ; decrement loop counter
 [cntr] B        LOOP            ; branch to loop
```

The two load doubleword (LDDW) instructions load a[i], a[i+1], b[i], and b[i+1] on each iteration.

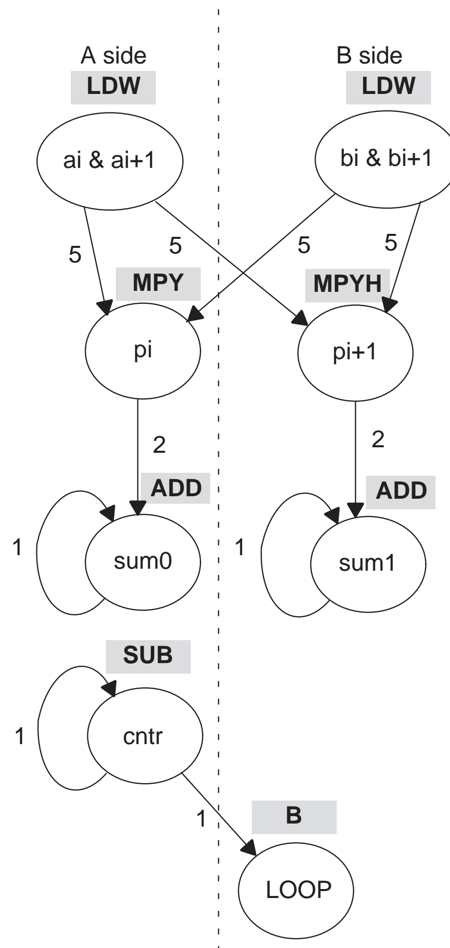
Two MPYSP instructions are now necessary to multiply the second set of array elements.

The two ADDSP instructions accumulate the sums of the even and odd elements: sum0 and sum1.

6.3.3 Drawing a Dependency Graph

The dependency graph in Figure 6–5 for the fixed-point dot product shows that the LDW instructions are parents of the MPY instructions and the MPY instructions are parents of the ADD instructions. To split the graph between the A and B register files, place an equal number of LDWs, MPYs, and ADDs on each side. To keep both sides even, place the remaining two instructions, B and SUB, on opposite sides.

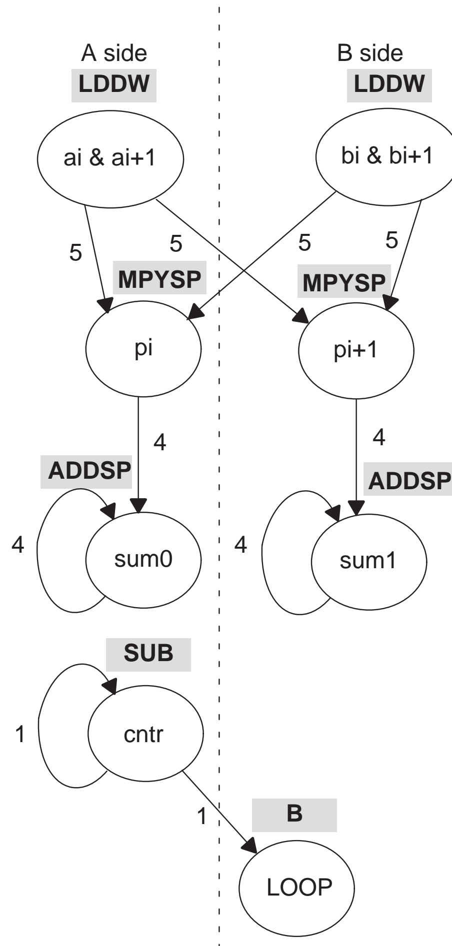
Figure 6–5. Dependency Graph of Fixed-Point Dot Product With LDW



Similarly, the dependency graph in Figure 6–6 for the floating-point dot product shows that the LDDW instructions are parents of the MPYSP instructions and the MPYSP instructions are parents of the ADDSP instructions. To split the graph between the A and B register files, place an equal number of

LDDWs, MPYSPs, and ADDSPs on each side. To keep both sides even, place the remaining two instructions, B and SUB, on opposite sides.

Figure 6–6. Dependency Graph of Floating-Point Dot Product With LDDW

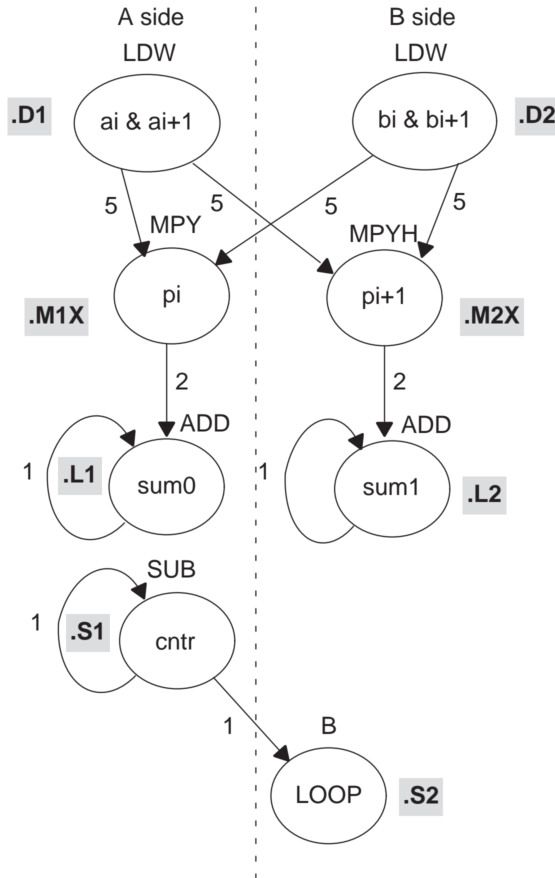


6.3.4 Linear Assembly Resource Allocation

After splitting the dependency graph for both the fixed-point and floating-point dot products, you can assign functional units and registers, as shown in the dependency graphs in Figure 6–7 and Figure 6–8 and in the instructions in Example 6–13 and Example 6–14. The .M1X and .M2X represent a path in the dependency graph crossing from one side to the other.

Part III

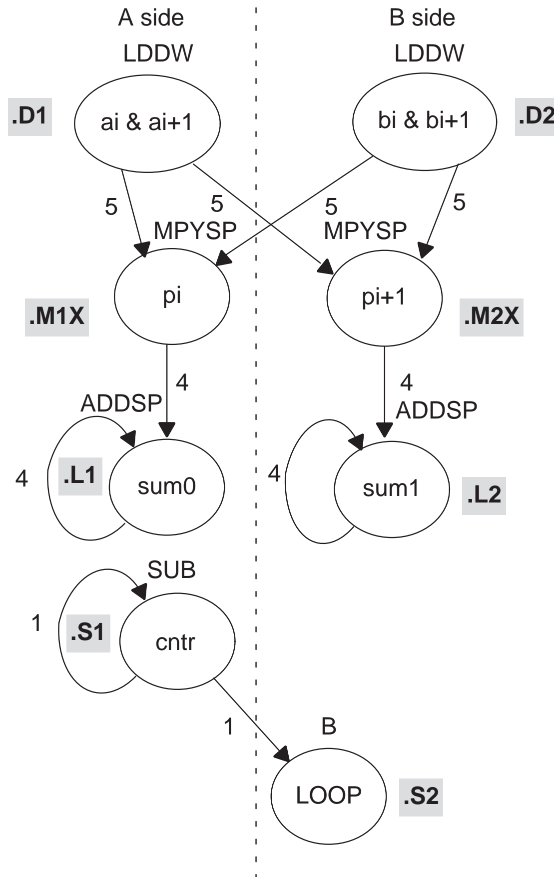
Figure 6–7. Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)



Example 6–13. Linear Assembly for Fixed-Point Dot Product Inner Loop With LDW (With Allocated Resources)

```
        LDW    .D1    *A4++,A2      ; load ai and ai+1 from memory
        LDW    .D2    *B4++,B2      ; load bi and bi+1 from memory
        MPY    .M1X   A2,B2,A6      ; ai * bi
        MPYH   .M2X   A2,B2,B6      ; ai+1 * bi+1
        ADD    .L1    A6,A7,A7      ; sum0 += (ai * bi)
        ADD    .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
        SUB    .S1    A1,1,A1       ; decrement loop counter
   [A1] B      .S2    LOOP          ; branch to loop
```

Figure 6–8. Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)



Example 6–14. Linear Assembly for Floating-Point Dot Product Inner Loop With LDDW (With Allocated Resources)

```
      LDDW   .D1    *A4++,A3:A2   ; load ai and ai+1 from memory
      LDDW   .D2    *B4++,B3:B2   ; load bi and bi+1 from memory
      MPYSP  .M1X   A2,B2,A6      ; ai * bi
      MPYSP  .M2X   A3,B3,B6      ; ai+1 * bi+1
      ADDSP  .L1    A6,A7,A7      ; sum0 += (ai * bi)
      ADDSP  .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
      SUB    .S1    A1,1,A1       ; decrement loop counter
 [A1] B      .S2    LOOP          ; branch to loop
```

Part III

6.3.5 Final Assembly

Example 6–15 shows the final assembly code for the unrolled loop of the fixed-point dot product and Example 6–16 shows the final assembly code for the unrolled loop of the floating-point dot product.

6.3.5.1 Fixed-Point Dot Product

Example 6–15 uses LDW instructions instead of LDH instructions.

Example 6–15. Assembly Code for Fixed-Point Dot Product With LDW (Before Software Pipelining)

```
       MVK    .S1    50,A1         ; set up loop counter
||     ZERO   .L1    A7            ; zero out sum0 accumulator
||     ZERO   .L2    B7            ; zero out sum1 accumulator

LOOP:
       LDW    .D1    *A4++,A2      ; load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2      ; load bi & bi+1 from memory

       SUB    .S1    A1,1,A1       ; decrement loop counter

[A1]   B      .S1    LOOP          ; branch to loop

       NOP    2

       MPY    .M1X   A2,B2,A6      ; ai * bi
||     MPYH   .M2X   A2,B2,B6      ; ai+1 * bi+1

       NOP

       ADD    .L1    A6,A7,A7      ; sum0+= (ai * bi)
||     ADD    .L2    B6,B7,B7      ; sum1+= (ai+1 * bi+1)
       ; Branch occurs here

       ADD    .L1X   A7,B7,A4      ; sum = sum0 + sum1
```

The code in Example 6–15 includes the following optimizations:

- The setup code for the loop is included to initialize the array pointers and the loop counter and to clear the accumulators. The setup code assumes that A4 and B4 have been initialized to point to arrays *a* and *b*, respectively.
- The MVK instruction initializes the loop counter.
- The two ZERO instructions, which execute in parallel, initialize the even and odd accumulators (sum0 and sum1) to 0.
- The third ADD instruction adds the even and odd accumulators.

6.3.5.2 Floating-Point Dot Product

Example 6–16 uses LDDW instructions instead of LDW instructions.

Example 6–16. Assembly Code for Floating-Point Dot Product With LDDW (Before Software Pipelining)

```
       MVK    .S1    50,A1          ; set up loop counter
||     ZERO   .L1    A7             ; zero out sum0 accumulator
||     ZERO   .L2    B7             ; zero out sum1 accumulator

LOOP:
       LDDW   .D1    *A4++,A2       ; load ai & ai+1 from memory
||     LDDW   .D2    *B4++,B2       ; load bi & bi+1 from memory

       SUB    .S1    A1,1,A1        ; decrement loop counter

       NOP    2

[A1]   B      .S1    LOOP           ; branch to loop

       MPYSP  .M1X   A2,B2,A6       ; ai * bi
||     MPYSP  .M2X   A3,B3,B6       ; ai+1 * bi+1

       NOP    3

       ADDSP  .L1    A6,A7,A7       ; sum0 += (ai * bi)
||     ADDSP  .L2    B6,B7,B7       ; sum1 += (ai+1 * bi+1)
       ; Branch occurs here

       NOP    3

       ADDSP  .L1X   A7,B7,A4       ; sum = sum0 + sum1

       NOP    3
```

The code in Example 6–16 includes the following optimizations:

- The setup code for the loop is included to initialize the array pointers and the loop counter and to clear the accumulators. The setup code assumes that A4 and B4 have been initialized to point to arrays *a* and *b*, respectively.
- The MVK instruction initializes the loop counter.
- The two ZERO instructions, which execute in parallel, initialize the even and odd accumulators (sum0 and sum1) to 0.
- The third ADDSP instruction adds the even and odd accumulators.

6.3.6 Comparing Performance

Executing the fixed-point dot product with the optimizations in Example 6–15 requires only 50 iterations, because you operate in parallel on both the even and odd array elements. With the setup code and the final ADD instruction, 100 iterations of this loop require a total of 402 cycles (1 + 8 × 50 + 1).

Table 6–3 compares the performance of the different versions of the fixed-point dot product code discussed so far.

Table 6–3. Comparison of Fixed-Point Dot Product Code With Use of LDW

Code Example		100 Iterations	Cycle Count
Example 6–5	Fixed-point dot product nonparallel assembly	2 + 100 × 16	1602
Example 6–6	Fixed-point dot product parallel assembly	1 + 100 × 8	801
Example 6–15	Fixed-point dot product parallel assembly with LDW	1 + (50× 8)+ 1	402

Executing the floating-point dot product with the optimizations in Example 6–16 requires only 50 iterations, because you operate in parallel on both the even and odd array elements. With the setup code and the final ADDSP instruction, 100 iterations of this loop require a total of 508 cycles (1 + 10 × 50 + 7).

Table 6–4 compares the performance of the different versions of the floating-point dot product code discussed so far.

Table 6–4. Comparison of Floating-Point Dot Product Code With Use of LDDW

Code Example		100 Iterations	Cycle Count
Example 6–7	Floating-point dot product nonparallel assembly	2 + 100 × 21	2102
Example 6–8	Floating-point dot product parallel assembly	1 + 100 × 10	1001
Example 6–16	Floating-point dot product parallel assembly with LDDW	1 + (50× 10)+ 7	508

6.4 Software Pipelining

This section describes the process for improving the performance of the assembly code in the previous section through *software pipelining*.

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations execute in parallel. The parallel resources on the ’C6x make it possible to initiate a new loop iteration before previous iterations finish. The goal of software pipelining is to start a new loop iteration as soon as possible.

The modulo iteration interval scheduling table is introduced in this section as an aid to creating software-pipelined loops.

The fixed-point dot product code in Example 6–15 needs eight cycles for each iteration of the loop: five cycles for the LDWs, two cycles for the MPYs, and one cycle for the ADDs.

Figure 6–9 shows the dependency graph for the fixed-point dot product instructions. Example 6–17 shows the same dot product assembly code in Example 6–13 on page 6-20, except that the SUB instruction is now conditional on the loop counter (A1).

Note:

Making the SUB instruction conditional on A1 ensures that A1 stops decrementing when it reaches 0. Otherwise, as the loop executes five more times, the loop counter becomes a negative number. When A1 is negative, it is nonzero and, therefore, causes the condition on the branch to be true again. If the SUB instruction were not conditional on A1, you would have an infinite loop.

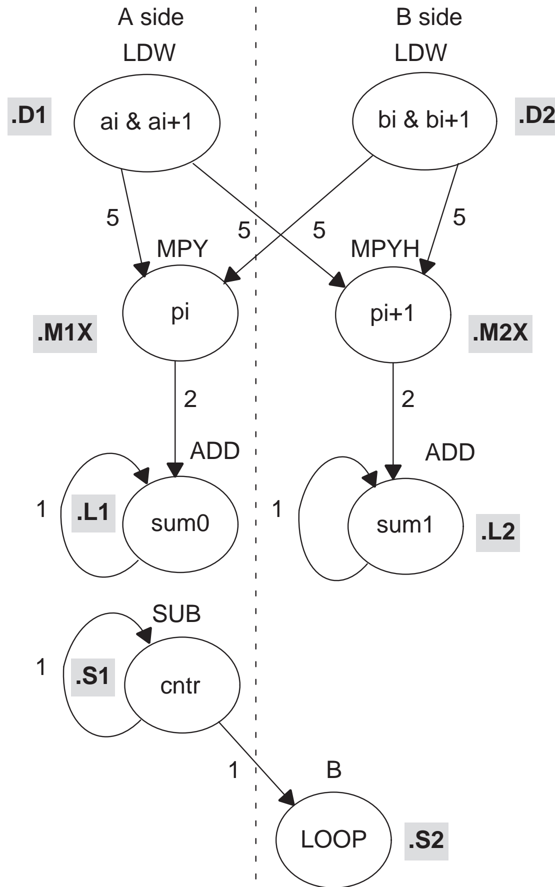
The floating-point dot product code in Example 6–16 needs ten cycles for each iteration of the loop: five cycles for the LDDWs, four cycles for the MPYSPs, and one cycle for the ADDSPs.

Figure 6–10 shows the dependency graph for the floating-point dot product instructions. Example 6–18 shows the same dot product assembly code in Example 6–14 on page 6-21, except that the SUB instruction is now conditional on the loop counter (A1).

Note:

The ADDSP has 3 delay slots associated with it. The extra delay slots are taken up by the LDDW, SUB, and NOP when executing the next cycle of the loop. Thus an NOP 3 is not required inside the loop but is required outside the loop prior to adding sum0 and sum1 together.

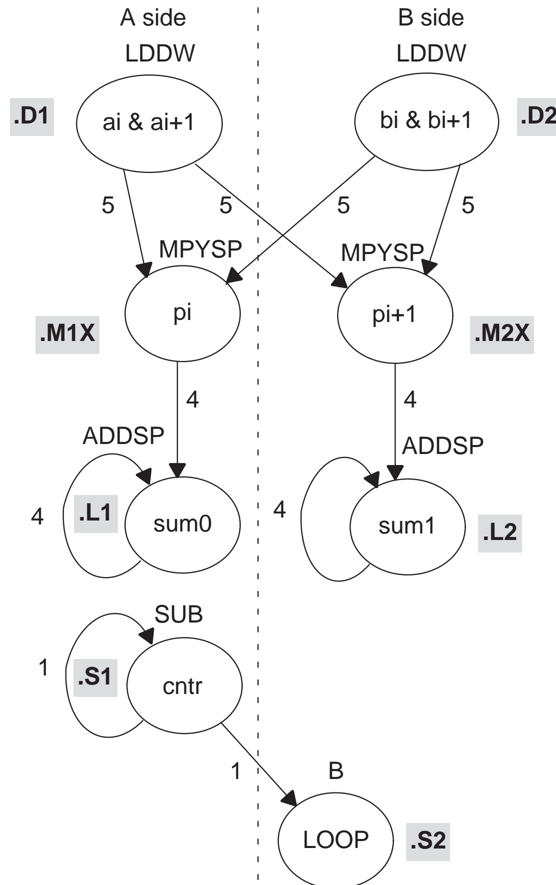
Figure 6–9. Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)



Example 6–17. Linear Assembly for Fixed-Point Dot Product Inner Loop (With Conditional SUB Instruction)

```
      LDW    .D1    *A4++,A2      ; load ai and ai+1 from memory
      LDW    .D2    *B4++,B2      ; load bi and bi+1 from memory
      MPY    .M1X   A2,B2,A6      ; ai * bi
      MPYH   .M2X   A2,B2,B6      ; ai+1 * bi+1
      ADD    .L1    A6,A7,A7      ; sum0 += (ai * bi)
      ADD    .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
 [A1] SUB    .S1    A1,1,A1       ; decrement loop counter
 [A1] B      .S2    LOOP          ; branch to top of loop
```

Figure 6–10. Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)



Example 6–18. Linear Assembly for Floating-Point Dot Product Inner Loop (With Conditional SUB Instruction)

```
      LDDW   .D1    *A4++,A2      ; load ai and ai+1 from memory
      LDDW   .D2    *B4++,B2      ; load bi and bi+1 from memory
      MPYSP  .M1X   A2,B2,A6      ; ai * bi
      MPYSP  .M2X   A2,B2,B6      ; ai+1 * bi+1
      ADDSP  .L1    A6,A7,A7      ; sum0 += (ai * bi)
      ADDSP  .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
 [A1] SUB    .S1    A1,1,A1       ; decrement loop counter
 [A1] B      .S2    LOOP          ; branch to top of loop
```

6.4.1 Modulo Iteration Interval Scheduling

Another way to represent the performance of the code is by looking at it in a modulo iteration interval scheduling table. This table shows how a software-pipelined loop executes and tracks the available resources on a cycle-by-cycle basis to ensure that no resource is used twice on any given cycle. The *iteration interval* of a loop is the number of cycles between the initiations of successive iterations of that loop.

6.4.1.1 Fixed-Point Example

The fixed-point code in Example 6–15 needs eight cycles for each iteration of the loop, so the iteration interval is eight.

Table 6–5 shows a modulo iteration interval scheduling table for the fixed-point dot product loop before software pipelining (Example 6–15). Each row represents a functional unit. There is a column for each cycle in the loop showing the instruction that is executing on a particular cycle:

- ❑ LDWs on the .D units are issued on cycles 0, 8, 16, 24, etc.
- ❑ MPY and MPYH on the .M units are issued on cycles 5, 13, 21, 29, etc.
- ❑ ADDs on the .L units are issued on cycles 7, 15, 23, 31, etc.
- ❑ SUB on the .S1 unit is issued on cycles 1, 9, 17, 25, etc.
- ❑ B on the .S2 unit is issued on cycles 2, 10, 18, 24, etc.

Table 6–5. Modulo Iteration Interval Scheduling Table for Fixed-Point Dot Product (Before Software Pipelining)

Unit / Cycle	0, 8, ...	1, 9, ...	2, 10, ...	3, 11, ...	4, 12, ...	5, 13, ...	6, 14, ...	7, 15, ...
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

In this example, each unit is used only once every eight cycles.

6.4.1.2 Floating-Point Example

The floating-point code in Example 6–16 needs ten cycles for each iteration of the loop, so the iteration interval is ten.

Table 6–6 shows a modulo iteration interval scheduling table for the floating-point dot product loop before software pipelining (Example 6–16). Each row represents a functional unit. There is a column for each cycle in the loop showing the instruction that is executing on a particular cycle:

- LDDWs on the .D units are issued on cycles 0, 10, 20, 30, etc.
- MPYSPs and on the .M units are issued on cycles 5, 15, 25, 35, etc.
- ADDSPs on the .L units are issued on cycles 9, 19, 29, 39, etc.
- SUB on the .S1 unit is issued on cycles 3, 13, 23, 33, etc.
- B on the .S2 unit is issued on cycles 4, 14, 24, 34, etc.

Table 6–6. Modulo Iteration Interval Scheduling Table for Floating-Point Dot Product (Before Software Pipelining)

Unit / Cycle	0, 10, ...	1, 11, ...	2, 12, ...	3, 13, ...	4, 14, ...	5, 15, ...	6, 16, ...	7, 17, ...	8, 18, ...	9, 19, ...
.D1	LDDW									
.D2	LDDW									
.M1						MPYSP				
.M2						MPYSP				
.L1										ADDSP
.L2										ADDSP
.S1				SUB						
.S2					B					

In this example, each unit is used only once every ten cycles.

6.4.1.3 Determining the Minimum Iteration Interval

Software pipelining increases performance by using the resources more efficiently. However, to create a fully pipelined schedule, it is helpful to first determine the *minimum iteration interval*.

The minimum iteration interval of a loop is the minimum number of cycles you must wait between each initiation of successive iterations of that loop. The smaller the iteration interval, the fewer cycles it takes to execute a loop.

Resources and data dependency constraints determine the minimum iteration interval. The most-used resource constrains the minimum iteration interval. For example, if four instructions in a loop all use the .S1 unit, the minimum iteration interval is at least 4. Four instructions using the same resource cannot execute in parallel and, therefore, require at least four separate cycles to execute each instruction.

With the SUB and branch instructions on opposite sides of the dependency graph in Figure 6–9 and Figure 6–10, all eight instructions use a different functional unit and no two instructions use the same cross paths (1X and 2X). Because no two instructions use the same resource, the minimum iteration interval based on resources is 1.

Note:

In this particular example, there are no data dependencies to affect the minimum iteration interval. However, future examples may demonstrate this constraint.

6.4.1.4 Creating a Fully Pipelined Schedule

Having determined that the minimum iteration interval is 1, you can initiate a new iteration every cycle. You can schedule LDW (or LDDW) and MPY (or MPYSP) instructions on every cycle.

Fixed-Point Example

Table 6–7 shows a fully pipelined schedule for the fixed-point dot product example.

Table 6–7. Modulo Iteration Interval Table for Fixed-Point Dot Product (After Software Pipelining)

Loop Prolog (cycles 0–6)

Unit / Cycle	0	1	2	3	4	5	6	7, 8, 9...
.D1	LDW	* LDW	** LDW	*** LDW	**** LDW	***** LDW	****** LDW	******* **LDW**
.D2	LDW	* LDW	** LDW	*** LDW	**** LDW	***** LDW	****** LDW	******* **LDW**
.M1						MPY	* MPY	** **MPY**
.M2						MPYH	* MPYH	** **MPYH**
.L1								**ADD**
.L2								**ADD**
.S1		SUB	* SUB	** SUB	*** SUB	**** SUB	***** SUB	****** **SUB**
.S2			B	* B	** B	*** B	**** B	***** **B**

Note: The asterisks indicate the iteration of the loop; shading indicates the single-cycle loop.

The rightmost column in Table 6–7 is a single-cycle loop that contains the entire loop. Cycles 0–6 are loop setup code, or loop prolog.

Asterisks define which iteration of the loop the instruction is executing each cycle. For example, the rightmost column shows that on any given cycle inside the loop:

- The ADD instructions are adding data for iteration n.
- The MPY instructions are multiplying data for iteration n + 2 (**).
- The LDW instructions are loading data for iteration n + 7 (*******).
- The SUB instruction is executing for iteration n + 6 (******).
- The B instruction is executing for iteration n + 5 (*****).

In this case, multiple iterations of the loop execute in parallel in a software pipeline that is eight iterations deep, with iterations n through n + 7 executing in parallel. Fixed-point software pipelines are rarely deeper than the one created by this single-cycle loop. As loop sizes grow, the number of iterations that can execute in parallel tends to become fewer.

Floating-Point Example

Table 6–8 shows a fully pipelined schedule for the floating-point dot product example.

Table 6–8. Modulo Iteration Interval Table for Floating-Point Dot Product (After Software Pipelining)

	Loop Prolog									
Unit / Cycle	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9, 10, 11...**
.D1	LDDW	* LDDW	** LDDW	*** LDDW	**** LDDW	***** LDDW	****** LDDW	******* LDDW	******** LDDW	********* **LDDW**
.D2	LDDW	* LDDW	** LDDW	*** LDDW	**** LDDW	***** LDDW	****** LDDW	******* LDDW	******** LDDW	********* **LDDW**
.M1						MPYSP	* MPYSP	** MPYSP	*** MPYSP	**** **MPYSP**
.M2						MPYSP	* MPYSP	** MPYSP	*** MPYSP	**** **MPYSP**
.L1										**ADDSP**
.L2										**ADDSP**
.S1				SUB	* SUB	** SUB	*** SUB	**** SUB	***** SUB	****** **SUB**
.S2					B	* B	** B	*** B	**** B	***** **B**

Note: The asterisks indicate the iteration of the loop; shading indicates the single-cycle loop.

The rightmost column in Table 6–8 is a single-cycle loop that contains the entire loop. Cycles 0–8 are loop setup code, or loop prolog.

Asterisks define which iteration of the loop the instruction is executing each cycle. For example, the rightmost column shows that on any given cycle inside the loop:

- The ADDSP instructions are adding data for iteration n.
- The MPYSP instructions are multiplying data for iteration n + 4 (****).
- The LDDW instructions are loading data for iteration n + 9 (*********).
- The SUB instruction is executing for iteration n + 6 (******).
- The B instruction is executing for iteration n + 5 (*****).

Note:

Since the ADDSP instruction has three delay slots associated with it, the results of adding are staggered by four. That is, the first result from the ADDSP is added to the fifth result, which is then added to the ninth, and so on. The second result is added to the sixth, which is then added to the 10th. This is shown in Table 6–9.

In this case, multiple iterations of the loop execute in parallel in a software pipeline that is ten iterations deep, with iterations n through n + 9 executing in parallel. Floating-point software pipelines are rarely deeper than the one created by this single-cycle loop. As loop sizes grow, the number of iterations that can execute in parallel tends to become fewer.

6.4.1.5 Staggered Accumulation With a Multicycle Instruction

When accumulating results with an instruction that is multicycle (that is, has delay slots other than 0), you must either unroll the loop or stagger the results. When unrolling the loop, multiple accumulators collect the results so that one result has finished executing and has been written into the accumulator before adding the next result of the accumulator. If you do not unroll the loop, then the accumulator will contain staggered results.

Staggered results occur when you attempt to accumulate successive results while in the delay slots of previous execution. This can be achieved without error if you are aware of what is in the accumulator, what will be added to that accumulator, and when the results will be written on a given cycle (such as the pseudo-code shown in Example 6–19).

Example 6–19. Pseudo-Code for Single-Cycle Accumulator With ADDSP

```
LOOP:       ADDSP    x,sum,sum
  ||        LDW      *xptr++,x
  ||[cond]  B        cond
  ||[cond]  SUB      cond,1,cond
```

Table 6–9 shows the results of the loop kernel for a single-cycle accumulator using a multicycle add instruction; in this case, the ADDSP, which has three delay slots (a 4-cycle instruction).

Table 6–9. Software Pipeline Accumulation Staggered Results Due to Three-Cycle Delay

Cycle #	Pseudoinstruction	Current value of pseudoregister sum	Written expected result
0	ADDSP x(0), sum, sum	0	; cycle 4 sum = x(0)
1	ADDSP x(1), sum, sum	0	; cycle 5 sum = x(1)
2	ADDSP x(2), sum, sum	0	; cycle 6 sum = x(2)
3	ADDSP x(3), sum, sum	0	; cycle 7 sum = x(3)
4	ADDSP x(4), sum, sum	x(0)	; cycle 8 sum = x(0) + x(4)
5	ADDSP x(5), sum, sum	x(1)	; cycle 9 sum = x(1) + x(5)
6	ADDSP x(6), sum, sum	x(6)	; cycle 10 sum = x(2) + x(6)
7	ADDSP x(7), sum, sum	x(7)	; cycle 11 sum = x(3) + x(7)
8	ADDSP x(8), sum, sum	x(0) + x(4)	; cycle 12 sum = x(0) + x(8)
	• • •		
i + j†	ADDSP x(i+j), sum, sum	x(j) + x(j+4) + x(j+8) … x(i–4+j)	; cycle i + j + 4 sum = x(j) + x(j+4) + x(j+8) … x(i–4+j) + x(i+j)
	• • •		

† where i is a multiple of 4

Part III

The first value of the array x, x(0) is added to the accumulator (sum) on cycle 0, but the result is not ready until cycle 4. This means that on cycle 1 when x(1) is added to the accumulator (sum), sum has no value in it from x(0). Thus, when this result is ready on cycle 5, sum will have the value x(1) in it, instead of the value x(0) + x(1). When you reach cycle 4, sum will have the value x(0) in it and the value x(4) will be added to that, causing sum = x(0) + x(4) on cycle 8. This is continuously repeated, resulting in four separate accumulations (using the register “sum”).

The current value in the accumulator “sum” depends on which iteration is being done. After the completion of the loop, the last four sums should be written into separate registers and then added together to give the final result. This is shown in Example 6–23 on page 6-39.

6.4.2 Using the Assembly Optimizer to Create Optimized Loops

Example 6–20 shows the linear assembly code for the full fixed-point dot product loop. Example 6–21 shows the linear assembly code for the full floating-point dot product loop. You can use this code as input to the assembly optimizer tool to create software-pipelined loops automatically. See the *TMS320C6x Optimizing C Compiler User's Guide* for more information on the assembly optimizer.

Example 6–20. Linear Assembly for Full Fixed-Point Dot Product

```
         .global _dotp

_dotp:  .cproc   a, b

         .reg     sum, sum0, sum1, cntr
         .reg     ai_i1, bi_i1, pi, pi1

         MVK      50,cntr          ; cntr = 100/2
         ZERO     sum0             ; multiply result = 0
         ZERO     sum1             ; multiply result = 0

LOOP:    .trip 50
         LDW      *a++,ai_i1       ; load ai & ai+1 from memory
         LDW      *b++,bi_i1       ; load bi & bi+1 from memory
         MPY      ai_i1,bi_i1,pi   ; ai * bi
         MPYH     ai_i1,bi_i1,pi1 ; ai+1 * bi+1
         ADD      pi,sum0,sum0     ; sum0 += (ai * bi)
         ADD      pi1,sum1,sum1    ; sum1 += (ai+1 * bi+1)
 [cntr]  SUB      cntr,1,cntr      ; decrement loop counter
 [cntr]  B        LOOP             ; branch to loop

         ADD      sum0,sum1,sum    ; compute final result

         .return sum

         .endproc
```

Resources such as functional units and 1X and 2X cross paths do not have to be specified because these can be allocated automatically by the assembly optimizer.

Example 6–21. Linear Assembly for Full Floating-Point Dot Product

```
        .global _dotp

_dotp:  .cproc   a, b

        .reg     sum, sum0, sum1, a, b
        .reg     ai:ai1, bi:bi1, pi, pi1

        MVK      50,cntr          ; cntr = 100/2
        ZERO     sum0             ; multiply result = 0
        ZERO     sum1             ; multiply result = 0

LOOP:   .trip 50
        LDDW     *a++,ai:ai1      ; load ai & ai+1 from memory
        LDDW     *b++,bi:bi1      ; load bi & bi+1 from memory
        MPYSP    a0,b0,pi         ; ai * bi
        MPYSP    a1,b1,pi1        ; ai+1 * bi+1
        ADDSP    pi,sum0,sum0     ; sum0 += (ai * bi)
        ADDSP    pi1,sum1,sum1    ; sum1 += (ai+1 * bi+1)
 [cntr] SUB      cntr,1,cntr      ; decrement loop counter
 [cntr] B        LOOP             ; branch to loop

        ADDSP    sum,sum1,sum0    ; compute final result

        .return sum

        .endproc
```


6.4.3 Final Assembly

Example 6–22 shows the assembly code for the fixed-point software-pipelined dot product in Table 6–7 on page 6-31. Example 6–23 shows the assembly code for the floating-point software-pipelined dot product in Table 6–8 on page 6-32. The accumulators are initialized to 0 and the loop counter is set up in the first execute packet in parallel with the first load instructions. The asterisks in the comments correspond with those in Table 6–7 and Table 6–8, respectively.

Note:

All instructions executing in parallel constitute an execute packet. An execute packet can contain up to eight instructions.

See the *TMS320C62x/C67x CPU and Instruction Set Reference Guide* for more information about pipeline operation.

6.4.3.1 Fixed-Point Example

Multiple branch instructions are in the pipe. The first branch in the fixed-point dot product is issued on cycle 2 but does not actually branch until the end of cycle 7 (after five delay slots). The branch target is the execute packet defined by the label LOOP. On cycle 7, the first branch returns to the same execute packet, resulting in a single-cycle loop. On every cycle after cycle 7, a branch executes back to LOOP until the loop counter finally decrements to 0. Once the loop counter is 0, five more branches execute because they are already in the pipe.

Executing the dot product code with the software pipelining as shown in Example 6–22 requires a total of 58 cycles (7 + 50 + 1), which is a significant improvement over the 402 cycles required by the code in Example 6–15.

Note:

The code created by the assembly optimizer will not completely match the final assembly code shown in this and future sections because different versions of the tool will produce slightly different code. However, the inner loop performance (number of cycles per iteration) should be similar.

Example 6–22. Assembly Code for Fixed-Point Dot Product (Software Pipelined)

```
        LDW    .D1   *A4++,A2      ; load ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ; load bi & bi+1 from memory
||      MVK    .S1   50,A1         ; set up loop counter
||      ZERO   .L1   A7            ; zero out sum0 accumulator
||      ZERO   .L2   B7            ; zero out sum1 accumulator

   [A1] SUB    .S1   A1,1,A1       ; decrement loop counter
||      LDW    .D1   *A4++,A2      ;* load ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;* load bi & bi+1 from memory

   [A1] SUB    .S1   A1,1,A1       ;* decrement loop counter
|| [A1] B      .S2   LOOP          ; branch to loop
||      LDW    .D1   *A4++,A2      ;** load ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;** load bi & bi+1 from memory

   [A1] SUB    .S1   A1,1,A1       ;** decrement loop counter
|| [A1] B      .S2   LOOP          ;* branch to loop
||      LDW    .D1   *A4++,A2      ;*** load ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;*** load bi & bi+1 from memory

   [A1] SUB    .S1   A1,1,A1       ;*** decrement loop counter
|| [A1] B      .S2   LOOP          ;** branch to loop
||      LDW    .D1   *A4++,A2      ;**** load ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;**** load bi & bi+1 from memory

        MPY    .M1X  A2,B2,A6      ; ai * bi
||      MPYH   .M2X  A2,B2,B6      ; ai+1 * bi+1
|| [A1] SUB    .S1   A1,1,A1       ;**** decrement loop counter
|| [A1] B      .S2   LOOP          ;*** branch to loop
||      LDW    .D1   *A4++,A2      ;***** ld ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;***** ld bi & bi+1 from memory

        MPY    .M1X  A2,B2,A6      ;* ai * bi
||      MPYH   .M2X  A2,B2,B6      ;* ai+1 * bi+1
|| [A1] SUB    .S1   A1,1,A1       ;***** decrement loop counter
|| [A1] B      .S2   LOOP          ;**** branch to loop
||      LDW    .D1   *A4++,A2      ;****** ld ai & ai+1 from memory
||      LDW    .D2   *B4++,B2      ;****** ld bi & bi+1 from memory

LOOP:
        ADD    .L1   A6,A7,A7      ; sum0 += (ai * bi)
||      ADD    .L2   B6,B7,B7      ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X  A2,B2,A6      ;** ai * bi
||      MPYH   .M2X  A2,B2,B6      ;** ai+1 * bi+1
|| [A1] SUB    .S1   A1,1,A1       ;****** decrement loop counter
|| [A1] B      .S2   LOOP          ;***** branch to loop
||      LDW    .D1   *A4++,A2      ;******* ld ai & ai+1 fm memory
||      LDW    .D2   *B4++,B2      ;******* ld bi & bi+1 fm memory
        ; Branch occurs here

        ADD    .L1X  A7,B7,A4      ; sum = sum0 + sum1
```

Part III

6.4.3.2 Floating-Point Example

The first branch in the floating-point dot product is issued on cycle 4 but does not actually branch until the end of cycle 9 (after five delay slots). The branch target is the execute packet defined by the label LOOP. On cycle 9, the first branch returns to the same execute packet, resulting in a single-cycle loop. On every cycle after cycle 9, a branch executes back to LOOP until the loop counter finally decrements to 0. Once the loop counter is 0, five more branches execute because they are already in the pipe.

Executing the floating-point dot product code with the software pipelining as shown in Example 6–23 requires a total of 74 cycles (9 + 50 + 15), which is a significant improvement over the 508 cycles required by the code in Example 6–16.

Example 6–23. Assembly Code for Floating-Point Dot Product (Software Pipelined)

```
        MVK     .S1    50,A1          ; set up loop counter
||      ZERO    .L1    A8             ; sum0 = 0
||      ZERO    .L2    B8             ; sum1 = 0
||      LDDW    .D1    A4++,A7:A6     ; load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ; load bi & bi + 1 from memory

        LDDW    .D1    A4++,A7:A6     ;* load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;* load bi & bi + 1 from memory

        LDDW    .D1    A4++,A7:A6     ;** load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;** load bi & bi + 1 from memory

        LDDW    .D1    A4++,A7:A6     ;*** load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;*** load bi & bi + 1 from memory
||[A1]  SUB     .S1    A1,1,A1        ; decrement loop counter

        LDDW    .D1    A4++,A7:A6     ;**** load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;**** load bi & bi + 1 from memory
||[A1]  B       .S2    LOOP           ; branch to loop
||[A1]  SUB     .S1    A1,1,A1        ;* decrement loop counter

        LDDW    .D1    A4++,A7:A6     ;***** load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;***** load bi & bi + 1 from memory
||      MPYSP   .M1X   A6,B6,A5       ; pi = a0  b0
||      MPYSP   .M2X   A7,B7,B5       ; pi1 = a1  b1
||[A1]  B       .S2    LOOP           ;* branch to loop
||[A1]  SUB     .S1    A1,1,A1        ;** decrement loop counter

        LDDW    .D1    A4++,A7:A6     ;****** load ai & ai + 1 from memory
||      LDDW    .D2    B4++,B7:B6     ;****** load bi & bi + 1 from memory
||      MPYSP   .M1X   A6,B6,A5       ;* pi = a0  b0
||      MPYSP   .M2X   A7,B7,B5       ;* pi1 = a1  b1
||[A1]  B       .S2    LOOP           ;** branch to loop
||[A1]  SUB     .S1    A1,1,A1        ;*** decrement loop counter
```

Example 6–23. Assembly Code for Floating-Point Dot Product (Software Pipelined) (Continued)

```
       LDDW      .D1    A4++,A7:A6       ;******* load ai & ai + 1 from memory
||     LDDW      .D2    B4++,B7:B6       ;******* load bi & bi + 1 from memory
||     MPYSP     .M1X   A6,B6,A5         ;** pi = a0  b0
||     MPYSP     .M2X   A7,B7,B5         ;** pi1 = a1  b1
||[A1] B         .S2    LOOP             ;*** branch to loop
||[A1] SUB       .S1    A1,1,A1          ;**** decrement loop counter

       LDDW      .D1    A4++,A7:A6       ;******** load ai & ai + 1 from memory
||     LDDW      .D2    B4++,B7:B6       ;******** load bi & bi + 1 from memory
||     MPYSP     .M1X   A6,B6,A5         ;*** pi = a0  b0
||     MPYSP     .M2X   A7,B7,B5         ;*** pi1 = a1  b1
||[A1] B         .S2    LOOP             ;**** branch to loop
||[A1] SUB       .S1    A1,1,A1          ;***** decrement loop counter

LOOP:
       LDDW      .D1    A4++,A7:A6       ;********* load ai & ai + 1 from memory
||     LDDW      .D2    B4++,B7:B6       ;********* load bi & bi + 1 from memory
||     MPYSP     .M1X   A6,B6,A5         ;**** pi = a0  b0
||     MPYSP     .M2X   A7,B7,B5         ;**** pi1 = a1  b1
||     ADDSP     .L1    A5,A8,A8         ; sum0 += (ai  bi)
||     ADDSP     .L2    B5,B8,B8         sum1 += (ai+1  bi+1)
||[A1] B         .S2    LOOP             ;***** branch to loop
||[A1] SUB       .S1    A1,1,A1          ;****** decrement loop counter
; Branch occurs here

       ADDSP     .L1X   A8,B8,A0         ; sum(0) = sum0(0) + sum1(0)

       ADDSP     .L2X   A8,B8,B0         ; sum(1) = sum0(1) + sum1(1)

       ADDSP     .L1X   A8,B8,A0         ; sum(2) = sum0(2) + sum1(2)

       ADDSP     .L2X   A8,B8,B0         ; sum(3) = sum0(3) + sum1(3)

       NOP                               ; wait for B0

       ADDSP     .L1X   A0,B0,A5         ; sum(01) = sum(0) + sum(1)

       NOP                               ; wait for next B0

       ADDSP     .L2X   A0,B0,B5         ; sum(23) = sum(2) + sum(3)

       NOP              3

       ADDSP     .L1X   A5,B5,A4         ; sum = sum(01) + sum(23)

       NOP              3                ;
```

Part III

6.4.3.3 Removing Extraneous Instructions

The code in Example 6–22 and Example 6–23 executes extra iterations of some of the instructions in the loop. The following operations occur in parallel on the last cycle of the loop in Example 6–22:

- Iteration 50 of the ADD instructions
- Iteration 52 of the MPY and MPYH instructions
- Iteration 57 of the LDW instructions

The following operations occur in parallel on the last cycle of the loop in Example 6–23:

- Iteration 50 of the ADDSP instructions
- Iteration 54 of the MPYSP instructions
- Iteration 59 of the LDDW instructions

In most cases, extra iterations are not a problem; however, when extraneous LDWs and LDDWs access unmapped memory, you can get unpredictable results. If the extraneous instructions present a potential problem, remove the extraneous load and multiply instructions by adding an epilog like that included in the second part of Example 6–24 on page 6-43 and Example 6–25 on page 6-44.

Fixed-Point Example

To eliminate LDWs in the fixed-point dot product from iterations 51 through 57, run the loop seven fewer times. This brings the loop counter to 43 (50 – 7), which means you still must execute seven more cycles of ADD instructions and five more cycles of MPY instructions. Five pairs of MPYs and seven pairs of ADDs are now outside the loop. The LDWs, MPYs, and ADDs all execute exactly 50 times. (The shaded areas of Example 6–24 indicate the changes in this code.)

Executing the dot product code in Example 6–24 with no extraneous LDWs still requires a total of 58 cycles (7 + 43 + 7 + 1), but the code size is now larger.

Floating-Point Example

To eliminate LDDWs in the floating-point dot product from iterations 51 through 59, run the loop nine fewer times. This brings the loop counter to 41 (50 – 9), which means you still must execute nine more cycles of ADDSP instructions and five more cycles of MPYSP instructions. Five pairs of MPYSPs and nine pairs of ADDSPs are now outside the loop. The LDDWs, MPYSPs, and

ADDSPs all execute exactly 50 times. (The shaded areas of Example 6–25 indicate the changes in this code.)

Executing the dot product code in Example 6–25 with no extraneous LDDWs still requires a total of 74 cycles (9 + 41 + 9 + 15), but the code size is now larger.

Example 6–24. Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads)

```
       LDW    .D1    *A4++,A2     ; load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ; load bi & bi+1 from memory
||     MVK    .S1    43,A1        ; set up loop counter
||     ZERO   .L1    A7           ; zero out sum0 accumulator
||     ZERO   .L2    B7           ; zero out sum1 accumulator

  [A1] SUB    .S1    A1,1,A1      ; decrement loop counter
||     LDW    .D1    *A4++,A2     ;* load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;* load bi & bi+1 from memory

  [A1] SUB    .S1    A1,1,A1      ;* decrement loop counter
||[A1] B      .S2    LOOP         ; branch to loop
||     LDW    .D1    *A4++,A2     ;** load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;** load bi & bi+1 from memory

  [A1] SUB    .S1    A1,1,A1      ;** decrement loop counter
||[A1] B      .S2    LOOP         ;* branch to loop
||     LDW    .D1    *A4++,A2     ;*** load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;*** load bi & bi+1 from memory

 [A1]  SUB    .S1    A1,1,A1      ;*** decrement loop counter
||[A1] B      .S2    LOOP         ;** branch to loop
||     LDW    .D1    *A4++,A2     ;**** load ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;**** load bi & bi+1 from memory

       MPY    .M1X   A2,B2,A6     ; ai * bi
||     MPYH   .M2X   A2,B2,B6     ; ai+1 * bi+1
||[A1] SUB    .S1    A1,1,A1      ;**** decrement loop counter
||[A1] B      .S2    LOOP         ;*** branch to loop
||     LDW    .D1    *A4++,A2     ;***** ld ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;***** ld bi & bi+1 from memory

       MPY    .M1X   A2,B2,A6     ;* ai * bi
||     MPYH   .M2X   A2,B2,B6     ;* ai+1 * bi+1
||[A1] SUB    .S1    A1,1,A1      ;***** decrement loop counter
||[A1] B      .S2    LOOP         ;**** branch to loop
||     LDW    .D1    *A4++,A2     ;****** ld ai & ai+1 from memory
||     LDW    .D2    *B4++,B2     ;****** ld bi & bi+1 from memory
```

Example 6–24. Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads) (Continued)

```
LOOP:
        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1
||[A1]  SUB    .S1    A1,1,A1      ;****** decrement loop counter
||[A1]  B      .S2    LOOP         ;***** branch to loop
||      LDW    .D1    *A4++,A2     ;******* ld ai & ai+1 fm memory
||      LDW    .D2    *B4++,B2     ;******* ld bi & bi+1 fm memory
        ; Branch occurs here
                                                                    ADDs   MPYs

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (1)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi                             (1)
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (2)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi                             (2)
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (3)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi                             (3)
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (4)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi                             (4)
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (5)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)
||      MPY    .M1X   A2,B2,A6     ;** ai * bi                             (5)
||      MPYH   .M2X   A2,B2,B6     ;** ai+1 * bi+1

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (6)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)

        ADD    .L1    A6,A7,A7     ; sum0 += (ai * bi)              (7)
||      ADD    .L2    B6,B7,B7     ; sum1 += (ai+1 * bi+1)

        ADD    .L1X   A7,B7,A4     ; sum = sum0 + sum1
```

Part III

Example 6–25. Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads)

```
        MVK      .S1    41,A1         ; set up loop counter
||      ZERO     .L1    A8            ; sum0 = 0
||      ZERO     .L2    B8            ; sum1 = 0
||      LDDW     .D1    A4++,A7:A6    ; load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ; load bi & bi + 1 from memory

        LDDW     .D1    A4++,A7:A6    ;* load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;* load bi & bi + 1 from memory

        LDDW     .D1    A4++,A7:A6    ;** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;** load bi & bi + 1 from memory

        LDDW     .D1    A4++,A7:A6    ;*** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;*** load bi & bi + 1 from memory
||[A1]  SUB      .S1    A1,1,A1       ; decrement loop counter

        LDDW     .D1    A4++,A7:A6    ;**** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;**** load bi & bi + 1 from memory
||[A1]  B        .S2    LOOP          ; branch to loop
||[A1]  SUB      .S1    A1,1,A1       ;* decrement loop counter

        LDDW     .D1    A4++,A7:A6    ;***** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;***** load bi & bi + 1 from memory
||      MPYSP    .M1X   A6,B6,A5      ; pi = a0  b0
||      MPYSP    .M2X   A7,B7,B5      ; pi1 = a1  b1
||[A1]  B        .S2    LOOP          ;* branch to loop
||[A1]  SUB      .S1    A1,1,A1       ;** decrement loop counter

        LDDW     .D1    A4++,A7:A6    ;****** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;****** load bi & bi + 1 from memory
||      MPYSP    .M1X   A6,B6,A5      ;* pi = a0  b0
||      MPYSP    .M2X   A7,B7,B5      ;* pi1 = a1  b1
||[A1]  B        .S2    LOOP          ;** branch to loop
||[A1]  SUB      .S1    A1,1,A1       ;*** decrement loop counter
        LDDW     .D1    A4++,A7:A6    ;******* load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;******* load bi & bi + 1 from memory
||      MPYSP    .M1X   A6,B6,A5      ;** pi = a0  b0
||      MPYSP    .M2X   A7,B7,B5      ;** pi1 = a1  b1
||[A1]  B        .S2    LOOP          ;*** branch to loop
||[A1]  SUB      .S1    A1,1,A1       ;**** decrement loop counter


        LDDW     .D1    A4++,A7:A6    ;******** load ai & ai + 1 from memory
||      LDDW     .D2    B4++,B7:B6    ;******** load bi & bi + 1 from memory
||      MPYSP    .M1X   A6,B6,A5      ;*** pi = a0  b0
||      MPYSP    .M2X   A7,B7,B5      ;*** pi1 = a1  b1
||[A1]  B        .S2    LOOP          ;**** branch to loop
||[A1]  SUB      .S1    A1,1,A1       ;***** decrement loop counter
```

Example 6–25. Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads) (Continued

```
LOOP:
        LDDW     .D1   A4++,A7:A6    ;********* load ai & ai + 1 from memory
||      LDDW     .D2   B4++,B7:B6    ;********* load bi & bi + 1 from memory
||      MPYSP    .M1X  A6,B6,A5      ;**** pi = a0  b0
||      MPYSP    .M2X  A7,B7,B5      ;**** pi1 = a1  b1
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)
||[A1] B         .S2   LOOP          ;***** branch to loop
||[A1] SUB       .S1   A1,1,A1       ;****** decrement loop counter
; Branch occurs here
                                                              ADDSPs   MPYSPs
        MPYSP    .M1X  A6,B6,A5      ; pi = a0  b0
||      MPYSP    .M2X  A7,B7,B5      ; pi1 = a1  b1                      (1)
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (1)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        MPYSP    .M1X  A6,B6,A5      ; pi = a0  b0
||      MPYSP    .M2X  A7,B7,B5      ; pi1 = a1  b1                      (2)
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (2)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        MPYSP    .M1X  A6,B6,A5      ; pi = a0  b0
||      MPYSP    .M2X  A7,B7,B5      ; pi1 = a1  b1                      (3)
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (3)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        MPYSP    .M1X  A6,B6,A5      ; pi = a0  b0                       (4)
||      MPYSP    .M2X  A7,B7,B5      ; pi1 = a1  b1
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)      (4)

        MPYSP    .M1X  A6,B6,A5      ; pi = a0  b0                       (5)
||      MPYSP    .M2X  A7,B7,B5      ; pi1 = a1  b1
||      ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (5)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (6)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (7)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)

        ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)      (8)

        ADDSP    .L1   A5,A8,A8      ; sum0 += (ai  bi)          (9)
||      ADDSP    .L2   B5,B8,B8      ; sum1 += (ai+1  bi+1)
```

Example 6–25. Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads) (Continued)

```
       ADDSP     .L1X   A8,B8,A0      ; sum(0) = sum0(0) + sum1(0)

       ADDSP     .L2X   A8,B8,B0      ; sum(1) = sum0(1) + sum1(1)

       ADDSP     .L1X   A8,B8,A0      ; sum(2) = sum0(2) + sum1(2)

       ADDSP     .L2X   A8,B8,B0      ; sum(3) = sum0(3) + sum1(3)

       NOP                            ; wait for B0

       ADDSP     .L1X   A0,B0,A5      ; sum(01) = sum(0) + sum(1)

       NOP                            ; wait for next B0

       ADDSP     .L2X   A0,B0,B5      ; sum(23) = sum(2) + sum(3)

       NOP              3

       ADDSP     .L1X   A5,B5,A4      ; sum = sum(01) + sum(23)

       NOP              3             ;
```

Part III

6.4.3.4 Priming the Loop

Although Example 6–24 and Example 6–25 execute as fast as possible, the code size can be smaller without significantly sacrificing performance. To help reduce code size, you can use a technique called *priming the loop*. Assuming that you can handle extraneous loads, start with Example 6–22 or Example 6–23, which do not have epilogs and, therefore, contain fewer instructions. (This technique can be used equally well with Example 6–24 or Example 6–25.)

Fixed-Point Example

To eliminate the prolog of the fixed-point dot product and, therefore, the extra LDW and MPY instructions, begin execution at the loop body (at the LOOP label). Eliminating the prolog means that:

- Two LDWs, two MPYs, and two ADDs occur in the first execution cycle of the loop.
- Because the first LDWs require five cycles to write results into a register, the MPYs do not multiply valid data until after the loop executes five times. The ADDs have no valid data until after seven cycles (five cycles for the first LDWs and two more cycles for the first valid MPYs).

Example 6–26 shows the loop without the prolog but with four new instructions that zero the inputs to the MPY and ADD instructions. Making the MPYs and ADDs use 0s before valid data is available ensures that the final accumulator values are unaffected. (The loop counter is initialized to 57 to accommodate the seven extra cycles needed to prime the loop.)

Because the first LDWs are not issued until after seven cycles, the code in Example 6–26 requires a total of 65 cycles (7 + 57+ 1). Therefore, you are reducing the code size with a slight loss in performance.

Example 6–26. Assembly Code for Fixed-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog)

```
       MVK    .S1    57,A1         ; set up loop counter

  [A1] SUB    .S1    A1,1,A1       ; decrement loop counter
||     ZERO   .L1    A7            ; zero out sum0 accumulator
||     ZERO   .L2    B7            ; zero out sum1 accumulator

  [A1] SUB    .S1    A1,1,A1       ;* decrement loop counter
||[A1] B      .S2    LOOP          ; branch to loop
||     ZERO   .L1    A6            ; zero out add input
||     ZERO   .L2    B6            ; zero out add input

  [A1] SUB    .S1    A1,1,A1       ;** decrement loop counter
||[A1] B      .S2    LOOP          ;* branch to loop
||     ZERO   .L1    A2            ; zero out mpy input
||     ZERO   .L2    B2            ; zero out mpy input

  [A1] SUB    .S1    A1,1,A1       ;*** decrement loop counter
||[A1] B      .S2    LOOP          ;** branch to loop

  [A1] SUB    .S1    A1,1,A1       ;**** decrement loop counter
||[A1] B      .S2    LOOP          ;*** branch to loop

  [A1] SUB    .S1    A1,1,A1       ;***** decrement loop counter
||[A1] B      .S2    LOOP          ;**** branch to loop

LOOP:
       ADD    .L1    A6,A7,A7      ; sum0 += (ai * bi)
||     ADD    .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
||     MPY    .M1X   A2,B2,A6      ;** ai * bi
||     MPYH   .M2X   A2,B2,B6      ;** ai+1 * bi+1
||[A1] SUB    .S1    A1,1,A1       ;****** decrement loop counter
||[A1] B      .S2    LOOP          ;***** branch to loop
||     LDW    .D1    *A4++,A2      ;******* ld ai & ai+1 fm memory
||     LDW    .D2    *B4++,B2      ;******* ld bi & bi+1 fm memory
       ; Branch occurs here

         ADD      .L1X     A7,B7,A4      ; sum = sum0 + sum1
```

Floating-Point Example

To eliminate the prolog of the floating-point dot product and, therefore, the extra LDDW and MPYSP instructions, begin execution at the loop body (at the LOOP label). Eliminating the prolog means that:

- Two LDDWs, two MPYSPs, and two ADDSPs occur in the first execution cycle of the loop.
- Because the first LDDWs require five cycles to write results into a register, the MPYSPs do not multiply valid data until after the loop executes five times. The ADDSPs have no valid data until after nine cycles (five cycles for the first LDDWs and four more cycles for the first valid MPYSPs).

Example 6–27 shows the loop without the prolog but with four new instructions that zero the inputs to the MPYSP and ADDSP instructions. Making the MPYSPs and ADDSPs use 0s before valid data is available ensures that the final accumulator values are unaffected. (The loop counter is initialized to 59 to accommodate the nine extra cycles needed to prime the loop.)

Because the first LDDWs are not issued until after nine cycles, the code in Example 6–27 requires a total of 81 cycles (7 + 59+ 15). Therefore, you are reducing the code size with a slight loss in performance.

Example 6–27. Assembly Code for Floating-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog)

```
        MVK     .S1     59,A1        ; set up loop counter

        ZERO    .L1     A7           ; zero out mpysp input
||      ZERO    .L2     B7           ; zero out mpysp input
||[A1]  SUB     .S1     A1,1,A1      ; decrement loop counter

  [A1]  B       .S2     LOOP         ; branch to loop
||[A1]  SUB     .S1     A1,1,A1      ;* decrement loop counter
||      ZERO    .L1     A8           ; zero out sum0 accumulator
||      ZERO    .L2     B8           ; zero out sum0 accumulator

  [A1]  B       .S2     LOOP         ;* branch to loop
||[A1]  SUB     .S1     A1,1,A1      ;** decrement loop counter
||      ZERO    .L1     A5           ; zero out addsp input
||      ZERO    .L2     B5           ; zero out addsp input

  [A1]  B       .S2     LOOP         ;** branch to loop
||[A1]  SUB     .S1     A1,1,A1      ;*** decrement loop counter
||      ZERO    .L1     A6           ; zero out mpysp input
||      ZERO    .L2     B6           ; zero out mpysp input
```

Example 6–27. Assembly Code for Floating-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog) (Continued)

```
  [A1]   B        .S2      LOOP          ;*** branch to loop
||[A1]   SUB      .S1      A1,1,A1       ;**** decrement loop counter

  [A1]   B        .S2      LOOP          ;**** branch to loop
||[A1]   SUB      .S1      A1,1,A1       ;***** decrement loop counter

LOOP:
         LDDW     .D1      A4++,A7:A6    ;********* load ai & ai + 1 from memory
||       LDDW     .D2      B4++,B7:B6    ;********* load bi & bi + 1 from memory
||       MPYSP    .M1X     A6,B6,A5      ;**** pi = a0  b0
||       MPYSP    .M2X     A7,B7,B5      ;**** pi1 = a1  b1
||       ADDSP    .L1      A5,A8,A8      ; sum0 += (ai  bi)
||       ADDSP    .L2      B5,B8,B8      ; sum1 += (ai+1  bi+1)
||[A1]   B        .S2      LOOP          ;***** branch to loop
||[A1]   SUB      .S1      A1,1,A1       ;****** decrement loop counter
; Branch occurs here

         ADDSP    .L1X     A8,B8,A0      ; sum(0) = sum0(0) + sum1(0)

         ADDSP    .L2X     A8,B8,B0      ; sum(1) = sum0(1) + sum1(1)

         ADDSP    .L1X     A8,B8,A0      ; sum(2) = sum0(2) + sum1(2)

         ADDSP    .L2X     A8,B8,B0      ; sum(3) = sum0(3) + sum1(3)

         NOP                             ; wait for B0

         ADDSP    .L1X     A0,B0,A5      ; sum(01) = sum(0) + sum(1)

         NOP                             ; wait for next B0

         ADDSP    .L2X     A0,B0,B5      ; sum(23) = sum(2) + sum(3)

         NOP               3

         ADDSP    .L1X     A5,B5,A4      ; sum = sum(01) + sum(23)

         NOP               3             ;
```

6.4.3.5 Removing Extra SUB Instructions

To reduce code size further, you can remove extra SUB instructions. If you know that the loop count is at least 6, you can eliminate the extra SUB instructions as shown in Example 6–28 and Example 6–29. The first five branch instructions are made unconditional, because they always execute. (If you do not know that the loop count is at least 6, you must keep the SUB instructions that decrement before each conditional branch as in Example 6–26 and Example 6–27.) Based on the elimination of six SUB instructions, the loop counter is now 51 (57 – 6) for the fixed-point dot product and 53 (59 – 6) for the floating-point dot product. This code shows some improvement over Example 6–26 and Example 6–27. The loop in Example 6–28 requires 63 cycles (5 + 57 + 1) and the loop in Example 6–27 requires 79 cycles (5 + 59 + 15).

Example 6–28. Assembly Code for Fixed-Point Dot Product (Software Pipelined With Smallest Code Size)

```
       B      .S2    LOOP          ; branch to loop
||     MVK    .S1    51,A1         ; set up loop counter

       B      .S2    LOOP          ;* branch to loop

       B      .S2    LOOP          ;** branch to loop
||     ZERO   .L1    A7            ; zero out sum0 accumulator
||     ZERO   .L2    B7            ; zero out sum1 accumulator

       B      .S2    LOOP          ;*** branch to loop
||     ZERO   .L1    A6            ; zero out add input
||     ZERO   .L2    B6            ; zero out add input

       B      .S2    LOOP          ;**** branch to loop
||     ZERO   .L1    A2            ; zero out mpy input
||     ZERO   .L2    B2            ; zero out mpy input

LOOP:
       ADD    .L1    A6,A7,A7      ; sum0 += (ai * bi)
||     ADD    .L2    B6,B7,B7      ; sum1 += (ai+1 * bi+1)
||     MPY    .M1X   A2,B2,A6      ;** ai * bi
||     MPYH   .M2X   A2,B2,B6      ;** ai+1 * bi+1
||[A1] SUB    .S1    A1,1,A1       ;****** decrement loop counter
||[A1] B      .S2    LOOP          ;***** branch to loop
||     LDW    .D1    *A4++,A2      ;******* ld ai & ai+1 fm memory
||     LDW    .D2    *B4++,B2      ;******* ld bi & bi+1 fm memory
       ; Branch occurs here

       ADD    .L1X   A7,B7,A4      ; sum = sum0 + sum1
```

Part III

Example 6–29. Assembly Code for Floating-Point Dot Product (Software Pipelined With Smallest Code Size)

```
        B      .S2    LOOP             ; branch to loop
||      MVK    .S1    53,A1            ; set up loop counter

        B      .S2    LOOP             ;* branch to loop
||      ZERO   .L1    A7               ; zero out mpysp input
||      ZERO   .L2    B7               ; zero out mpysp input

        B      .S2    LOOP             ;** branch to loop
||      ZERO   .L1    A8               ; zero out sum0 accumulator
||      ZERO   .L2    B8               ; zero out sum0 accumulator

        B      .S2    LOOP             ;*** branch to loop
||      ZERO   .L1    A5               ; zero out addsp input
||      ZERO   .L2    B5               ; zero out addsp input

        B      .S2    LOOP             ;**** branch to loop
||      ZERO   .L1    A6               ; zero out mpysp input
||      ZERO   .L2    B6               ; zero out mpysp input

LOOP:
        LDDW   .D1    A4++,A7:A6       ;********* load ai & ai + 1 from memory
||      LDDW   .D2    B4++,B7:B6       ;********* load bi & bi + 1 from memory
||      MPYSP  .M1X   A6,B6,A5         ;**** pi = a0  b0
||      MPYSP  .M2X   A7,B7,B5         ;**** pi1 = a1  b1
||      ADDSP  .L1    A5,A8,A8         ; sum0 += (ai  bi)
||      ADDSP  .L2    B5,B8,B8         ; sum1 += (ai+1  bi+1)
||[A1]  B      .S2    LOOP             ;***** branch to loop
||[A1]  SUB    .S1    A1,1,A1          ;****** decrement loop counter
; Branch occurs here

        ADDSP  .L1X   A8,B8,A0         ; sum(0) = sum0(0) + sum1(0)

        ADDSP  .L2X   A8,B8,B0         ; sum(1) = sum0(1) + sum1(1)

        ADDSP  .L1X   A8,B8,A0         ; sum(2) = sum0(2) + sum1(2)

        ADDSP  .L2X   A8,B8,B0         ; sum(3) = sum0(3) + sum1(3)

        NOP                            ; wait for B0

        ADDSP  .L1X   A0,B0,A5         ; sum(01) = sum(0) + sum(1)

        NOP                            ; wait for next B0

        ADDSP  .L2X    A0,B0,B5        ; sum(23) = sum(2) + sum(3)

        NOP           3

        ADDSP  .L1X   A5,B5,A4         ; sum = sum(01) + sum(23)

        NOP           3                ;
```

6.4.4 Comparing Performance

Table 3–2 compares the performance of all versions of the fixed-point dot product code. Table 6–11 compares the performance of all versions of the floating-point dot product code.

Table 6–10. Comparison of Fixed-Point Dot Product Code Examples

Code Example		100 Iterations	Cycle Count
Example 6–5	Fixed-point dot product linear assembly	2 + 100 × 16	1602
Example 6–6	Fixed-point dot product parallel assembly	1 + 100 × 8	801
Example 6–15	Fixed-point dot product parallel assembly with LDW	1 + (50 × 8) + 1	402
Example 6–22	Fixed-point software-pipelined dot product	7 + 50 + 1	58
Example 6–24	Fixed-point software-pipelined dot product with no extraneous loads	7 + 43 + 7 + 1	58
Example 6–26	Fixed-point software-pipelined dot product with no prolog or epilog	7 + 57 + 1	65
Example 6–28	Fixed-point software-pipelined dot product with smallest code size	5 + 57 + 1	63

Table 6–11. Comparison of Floating-Point Dot Product Code Examples

Code Example		100 Iterations	Cycle Count
Example 6–7	Floating-point dot product nonparallel assembly	2 + 100 × 21	2102
Example 6–8	Floating-point dot product parallel assembly	1 + 100 × 10	1001
Example 6–16	Floating-point dot product parallel assembly with LDDW	1 + (50 × 10) + 7	508
Example 6–23	Floating-point software-pipelined dot product	9 + 50 + 15	74
Example 6–25	Floating-point software-pipelined dot product with no extraneous loads	9 + 41 + 9 + 15	74
Example 6–27	Floating-point software-pipelined dot product with no prolog or epilog	7 + 59 + 15	81
Example 6–29	Floating-point software-pipelined dot product with smallest code size	5 + 59 + 15	79

6.5 Modulo Scheduling of Multicycle Loops

Section 6.4 demonstrated the modulo-scheduling technique for the dot product code. In that example of a single-cycle loop, none of the instructions used the same resources. Multicycle loops can present resource conflicts which affect modulo scheduling. This section describes techniques to deal with this issue.

6.5.1 Weighted Vector Sum C Code

Example 6–30 shows the C code for a weighted vector sum.

Example 6–30. Weighted Vector Sum C Code

```
void w_vec(short a[],short b[],short c[],short m)
{
  int i;

  for (i=0; i<100; i++) {
       c[i] = ((m * a[i]) >> 15) + b[i];
       }
}
```

6.5.2 Translating C Code to Linear Assembly

Example 6–31 shows the linear assembly that executes the weighted vector sum in Example 6–30. This linear assembly does not have functional units assigned. The dependency graph will help in those decisions. However, before looking at the dependency graph, the code can be optimized further.

Example 6–31. Linear Assembly for Weighted Vector Sum Inner Loop

```
      LDH    *aptr++,ai          ; ai
      LDH    *bptr++,bi          ; bi
      MPY    m,ai,pi             ; m * ai
      SHR    pi,15,pi_scaled     ; (m * ai) >> 15
      ADD    pi_scaled,bi,ci     ; ci = (m * ai) >> 15 + bi
      STH    ci,*cptr++          ; store ci
 [cntr]SUB   cntr,1,cntr         ; decrement loop counter
 [cntr]B     LOOP                ; branch to loop
```

6.5.3 Determining the Minimum Iteration Interval

Example 6–31 includes three memory operations in the inner loop (two LDHs and the STH) that must each use a .D unit. Only two .D units are available on any single cycle; therefore, this loop requires at least two cycles. Because no other resource is used more than twice, the minimum iteration interval for this loop is 2.

Memory operations determine the minimum iteration interval in this example. Therefore, before scheduling this assembly code, unroll the loop and perform LDWs to help improve the performance.

6.5.3.1 Unrolling the Weighted Vector Sum C Code

Example 6–32 shows the C code for an unrolled version of the weighted vector sum.

Example 6–32. Weighted Vector Sum C Code (Unrolled)

```
void w_vec(short a[],short b[],short c[],short m)
{
   int i;

   for (i=0; i<100; i+=2) {
        c[i] = ((m * a[i]) >> 15) + b[i];
        c[i+1] = ((m * a[i+1]) >> 15) + b[i+1];
        }
}
```

6.5.3.2 Translating Unrolled Inner Loop to Linear Assembly

Example 6–33 shows the linear assembly that calculates c[i] and c[i+1] for the weighted vector sum in Example 6–32.

- The two store pointers (*ciptr and *ci+1ptr) are separated so that one (*ciptr) increments by 2 through the odd elements of the array and the other (*ci+1ptr) increments through the even elements.
- AND and SHR separate bi and bi+1 into two separate registers.
- This code assumes that mask is preloaded with 0x0000FFFF to clear the upper 16 bits. The shift right of 16 places bi+1 into the 16 LSBs.

Example 6–33. Linear Assembly for Weighted Vector Sum Using LDW

```
       LDW    *aptr++,ai_i+1           ; ai & ai+1
       LDW    *bptr++,bi_i+1           ; bi & bi+1
       MPY    m,ai_i+1,pi              ; m * ai
       MPYHL  m,ai_i+1,pi+1            ; m * ai+1
       SHR    pi,15,pi_scaled          ; (m * ai) >> 15
       SHR    pi+1,15,pi+1_scaled      ; (m * ai+1) >> 15
       AND    bi_i+1,mask,bi           ; bi
       SHR    bi_i+1,16,bi+1           ; bi+1
       ADD    pi_scaled,bi,ci          ; ci = (m * ai) >> 15 + bi
       ADD    pi+1_scaled,bi+1,ci+1    ; ci+1 = (m * ai+1) >> 15 + bi+1
       STH    ci,*ciptr++[2]           ; store ci
       STH    ci+1,*ci+1ptr++[2]       ; store ci+1
 [cntr]SUB    cntr,1,cntr              ; decrement loop counter
 [cntr]B      LOOP                     ; branch to loop
```


6.5.3.3 Determining a New Minimum Iteration Interval

Use the following considerations to determine the minimum iteration interval for the assembly instructions in Example 6–33:

- Four memory operations (two LDWs and two STHs) must each use a .D unit. With two .D units available, this loop still requires only two cycles.
- Four instructions must use the .S units (three SHRs and one branch). With two .S units available, the minimum iteration interval is still 2.
- The two MPYs do not increase the minimum iteration interval.
- Because the remaining four instructions (two ADDs, AND, and SUB) can all use a .L unit, the minimum iteration interval for this loop is the same as in Example 6–31.

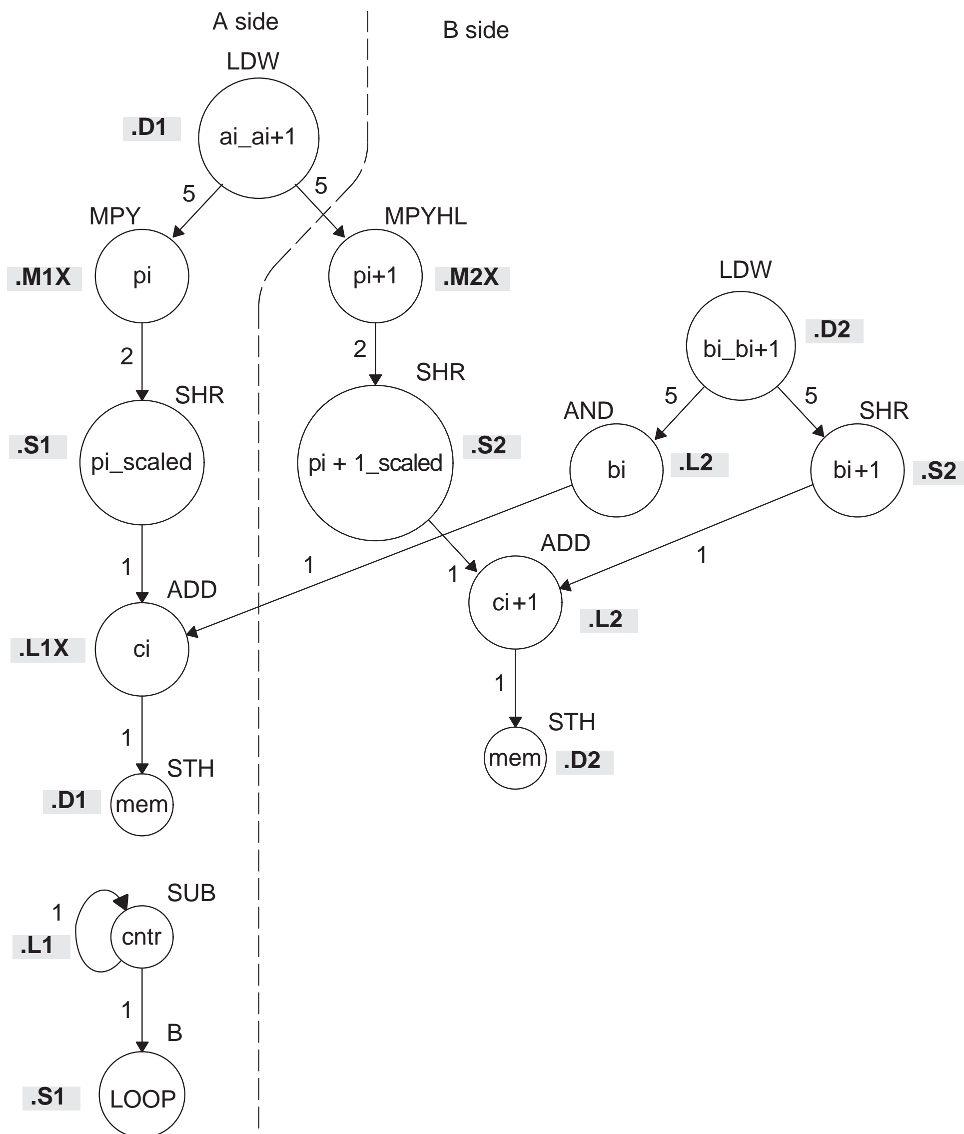
By using LDWs instead of LDHs, the program can do twice as much work in the same number of cycles.

6.5.4 Drawing a Dependency Graph

To achieve a minimum iteration interval of 2, you must put an equal number of operations per unit on each side of the dependency graph. Three operations in one unit on a side would result in an minimum iteration interval of 3.

Figure 6–11 shows the dependency graph divided evenly with a minimum iteration interval of 2.

Figure 6–11. Dependency Graph of Weighted Vector Sum



6.5.5 Linear Assembly Resource Allocation

Using the dependency graph, you can allocate functional units and registers as shown in Example 6–34. This code is based on the following assumptions:

- The pointers are initialized outside the loop.
- m resides in B6, which causes both .M units to use a cross path.
- The mask in the AND instruction resides in B10.

Example 6–34. Linear Assembly for Weighted Vector Sum With Resources Allocated

```
      LDW    .D1    *A4++,A2      ; ai & ai+1
      LDW    .D2    *B4++,B2      ; bi & bi+1
      MPY    .M1X   A2,B6,A5      ; pi = m * ai
      MPYHL  .M2X   A2,B6,B5      ; pi+1 = m * ai+1
      SHR    .S1    A5,15,A7      ; pi_scaled = (m * ai) >> 15
      SHR    .S2    B5,15,B7      ; pi+1_scaled = (m * ai+1) >> 15
      AND    .L2    B2,B10,B8     ; bi
      SHR    .S2    B2,16,B1      ; bi+1
      ADD    .L1X   A7,B8,A9      ; ci = (m * ai) >> 15 + bi
      ADD    .L2    B7,B1,B9      ; ci+1 = (m * ai+1) >> 15 + bi+1
      STH    .D1    A9,*A6++[2]   ; store ci
      STH    .D2    B9,*B0++[2]   ; store ci+1
 [A1] SUB    .L1    A1,1,A1       ; decrement loop counter
 [A1] B      .S1    LOOP          ; branch to loop
```


6.5.6 Modulo Iteration Interval Scheduling

Table 6–12 provides a method to keep track of resources that are a modulo iteration interval away from each other. In the single-cycle dot product example, every instruction executed every cycle and, therefore, required only one set of resources. Table 6–12 includes two groups of resources, which are necessary because you are scheduling a two-cycle loop.

- Instructions that execute on cycle k also execute on cycle k + 2, k + 4, etc. Instructions scheduled on these even cycles cannot use the same resources.
- Instructions that execute on cycle k + 1 also execute on cycle k + 3, k + 5, etc. Instructions scheduled on these odd cycles cannot use the same resources.
- Because two instructions (MPY and ADD) use the 1X path but do not use the same functional unit, Table 6–12 includes two rows (1X and 2X) that help you keep track of the cross path resources.

Only seven instructions have been scheduled in this table.

- The two LDWs use the .D units on the even cycles.
- The MPY and MPYH are scheduled on cycle 5 because the LDW has four delay slots. The MPY instructions appear in two rows because they use the .M and cross path resources on cycles 5, 7, 9, etc.
- The two SHR instructions are scheduled two cycles after the MPY to allow for the MPY's single delay slot.
- The AND is scheduled on cycle 5, four delay slots after the LDW.

Table 6–12. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

Unit/Cycle	0	2	4	6	8	10
.D1	**LDW ai_i+1**	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	**LDW bi_i+1**	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1						
.L2						
.S1						
.S2						
1X						
2X						
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11**
.D1						
.D2						
.M1			**MPY pi**	* MPY pi	** MPY pi	*** MPY pi
.M2			**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1			**AND bi**	* AND bi	** AND bi	*** AND bi
.L2						
.S1				**SHR pi_s**	* SHR pi_s	** SHR pi_s
.S2				**SHR pi+1_s**	* SHR pi+1_s	** SHR pi+1_s
1X			**MPY pi**	* MPY pi	** MPY pi	*** MPY pi
2X			**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

Note: The asterisks indicate the iteration of the loop; shaded cells indicate cycle 0.

6.5.6.1 Resource Conflicts

Resources from one instruction cannot conflict with resources from any other instruction scheduled modulo iteration intervals away. In other words, for a 2-cycle loop, instructions scheduled on cycle n cannot use the same resources as instructions scheduled on cycles n + 2, n + 4, n + 6, etc. Table 6–13 shows the addition of the SHR bi+1 instruction. This must avoid a conflict of resources in cycles 5 and 7, which are one iteration interval away from each other.

Even though LDW bi_i+1 (.D2, cycle 0) finishes on cycle 5, its child, SHR bi+1, cannot be scheduled on .S2 until cycle 6 because of a resource conflict with SHR pi+1_scaled, which is on .S2 in cycle 7.

Figure 6–12. Dependency Graph of Weighted Vector Sum (Showing Resource Conflict)

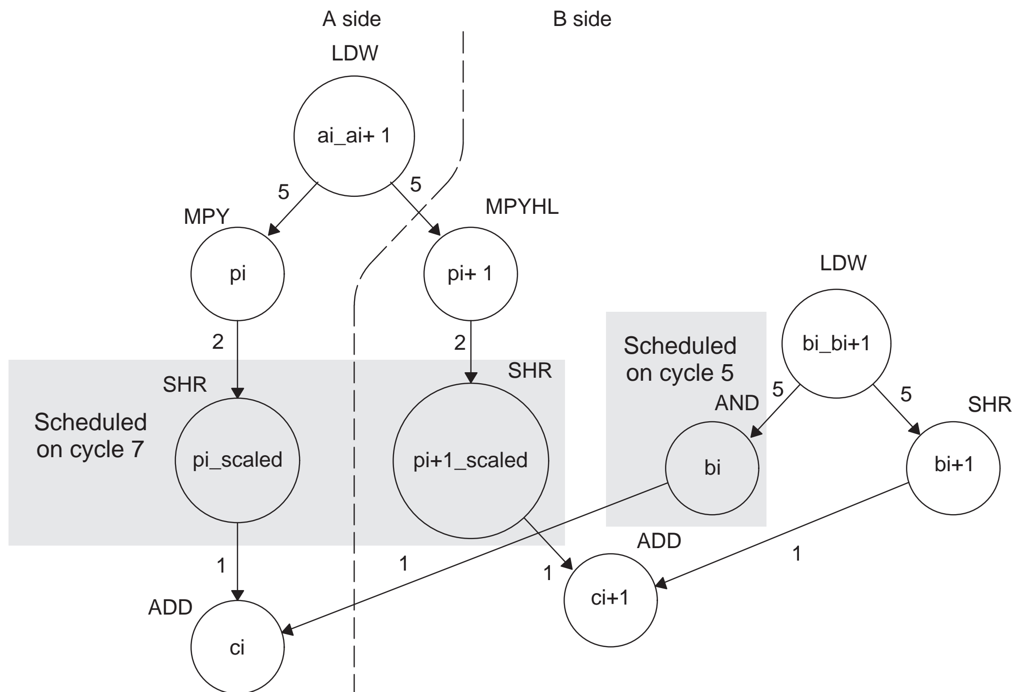


Table 6–13. Modulo Iteration Interval Table for Weighted Vector Sum With SHR Instructions

Unit / Cycle	0	2	4	6	8	10, 12, 14, ...
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	LDW bi_i+1	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1						
.L2						
.S1						
.S2				SHR bi+1	* SHR bi+1	** SHR bi+1
1X						
2X						
Unit / Cycle	**1**	**3**	**5**	**7**	**9**	**11, 13, 15, ...**
.D1						
.D2						
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1			AND bi	* AND bi	** AND bi	*** AND bi
.L2						
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2				SHR pi+1_s	* SHR pi+1_s	** SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–12.

Unit / Cycle	1	3	5	7	9	11, 13, 15, ...
2X			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–12.

Part III

6.5.6.2 Live Too Long

Scheduling SHR bi+1 on cycle 6 now creates a problem with scheduling the ADD ci instruction. The parents of ADD ci (AND bi and SHR pi_scaled) are scheduled on cycles 5 and 7, respectively. Because the SHR pi_scaled is scheduled on cycle 7, the earliest you can schedule ADD ci is cycle 8.

However, in cycle 7, AND bi * writes bi for the next iteration of the loop, which creates a scheduling problem with the ADD ci instruction. If you schedule ADD ci on cycle 8, the ADD instruction reads the parent value of bi for the next iteration, which is incorrect. The ADD ci demonstrates a live-too-long problem.

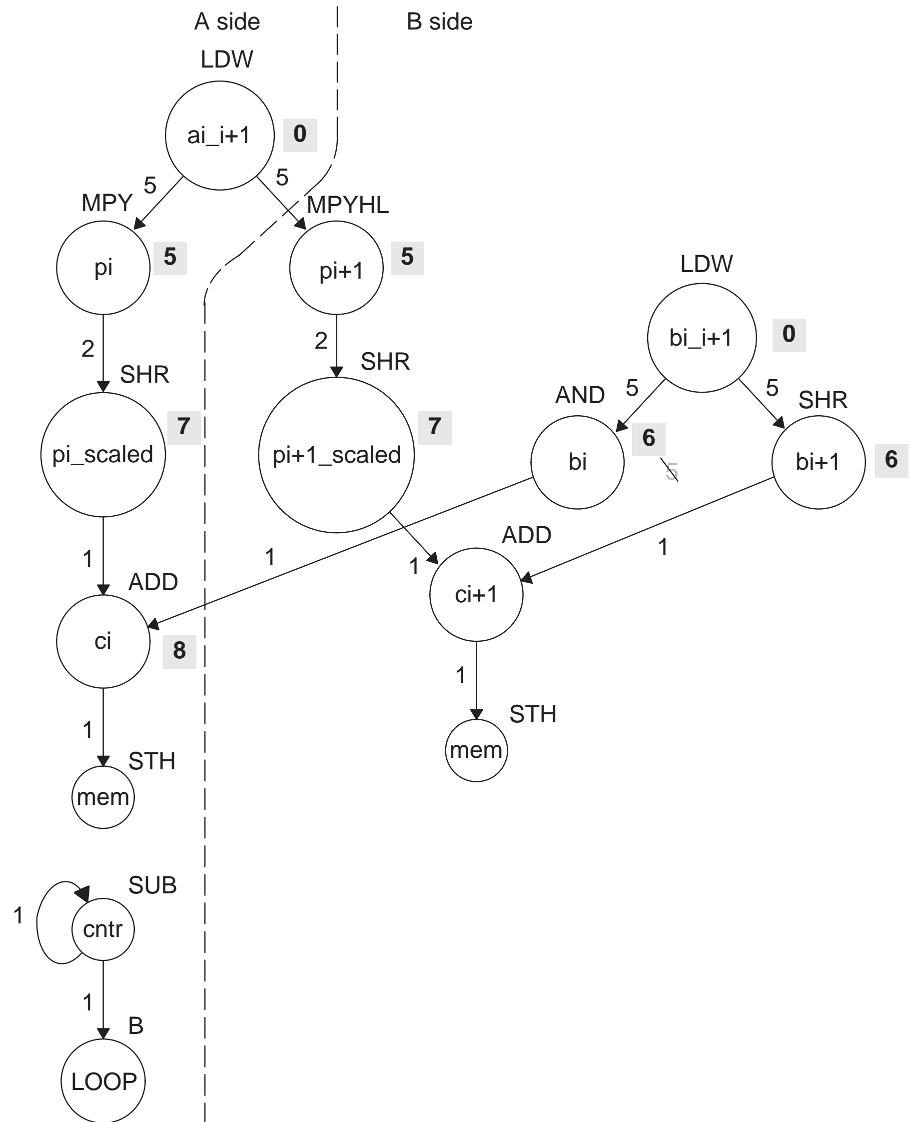
No value can be live in a register for more than the number of cycles in the loop. Otherwise, iteration n + 1 writes into the register before iteration n has read that register. Therefore, in a 2-cycle loop, a value is written to a register at the end of cycle n, then all children of that value must read the register before the end of cycle n + 2.

6.5.6.3 Solving the Live-Too-Long Problem

The live-too-long problem in Table 6–13 means that the bi value would have to be live from cycles 6–8, or 3 cycles. *No loop variable can live longer than the iteration interval,* because a child would then read the parent value for the next iteration.

To solve this problem move AND bi to cycle 6 so that you can schedule ADD ci to read the correct value on cycle 8, as shown in Figure 6–13 and Table 6–14.

Figure 6–13. Dependency Graph of Weighted Vector Sum (With Resource Conflict Resolved)



Note: Shaded numbers indicate the cycle in which the instruction is first scheduled.

Table 6–14. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

Unit/Cycle	0	2	4	6	8	10
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	LDW bi_i+1	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1					**ADD ci**	* ADD ci
.L2				**AND bi**	* AND bi	** AND bi
.S1						
.S2				SHR bi+1	* SHR bi+1	** SHR bi+1
1X						
2X						
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11**
.D1						
.D2						
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1						
.L2						
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2				SHR pi+1_s	* SHR pi+1_s	** SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi
2X			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

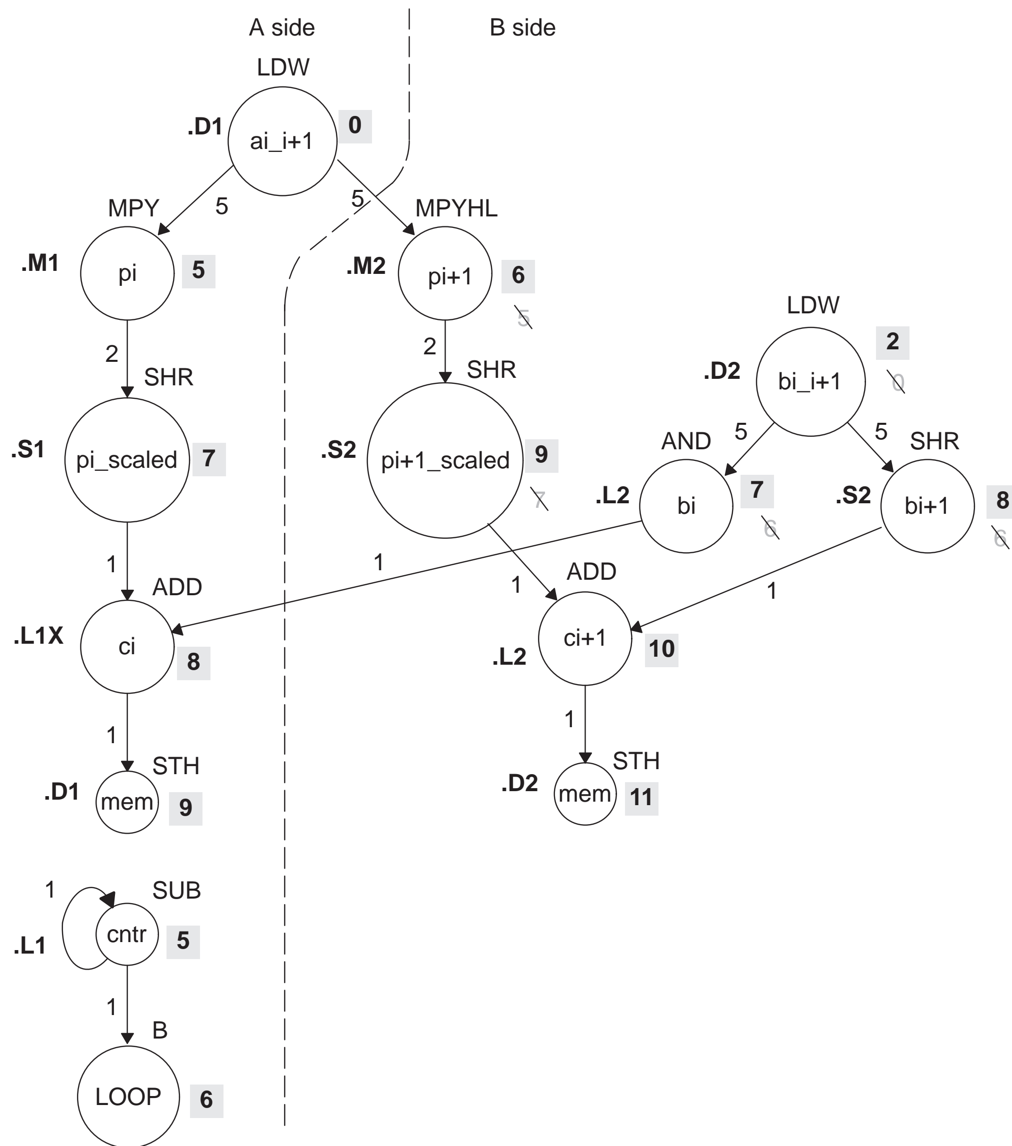
Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–13.

Part III

6.5.6.4 Scheduling the Remaining Instructions

Figure 6–14 shows the dependency graph with additional scheduling changes. The final version of the loop, with all instructions scheduled correctly, is shown in Table 6–15.

Figure 6–14. Dependency Graph of Weighted Vector Sum (Scheduling ci +1)



Note: Shaded numbers indicate the cycle in which the instruction is first scheduled.

Table 6–15 shows the following additions:

- B LOOP (.S1, cycle 6)
- SUB cntr (.L1, cycle 5)
- ADD ci+1 (.L2, cycle 10)
- STH ci (cycle 9)
- STH ci+1 (cycle 11)

To avoid resource conflicts and live-too-long problems, Table 6–15 also includes the following additional changes:

- LDW bi_i+1 (.D2) moved from cycle 0 to cycle 2.
- AND bi (.L2) moved from cycle 6 to cycle 7.
- SHR pi+1_scaled (.S2) moved from cycle 7 to cycle 9.
- MPYHL pi+1 moved from cycle 5 to cycle 6.
- SHR bi+1 moved from cycle 6 to 8.

From the table, you can see that this loop is pipelined six iterations deep, because iterations n and n + 5 execute in parallel.

Table 6–15. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

Unit/Cycle	0	2	4	6	8	10, 12, 14, ...
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2		**LDW bi_i+1**	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1
.M1						
.M2				**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1
.L1					**ADD ci**	* ADD ci
.L2						**ADD ci+1**
.S1				**B LOOP**	* B LOOP	** B LOOP
.S2					**SHR bi+1**	* SHR bi+1
1X					**ADD ci**	* ADD ci
2X				**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11, 13, 15, ...**
.D1					**STH ci**	* STH ci
.D2						**STH ci+1**
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2						
.L1			**SUB cntr**	* SUB cntr	** SUB cntr	*** SUB cntr
.L2				**AND bi**	* AND bi	** AND bi
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2					**SHR pi+1_s**	* SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi
2X						

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–14.

6.5.7 Using the Assembly Optimizer for the Weighted Vector Sum

Example 6–35 shows the linear assembly code to perform the weighted vector sum. You can use this code as input to the assembly optimizer to create a software-pipelined loop instead of scheduling this by hand.

Example 6–35. Linear Assembly for Weighted Vector Sum

```
         .global _w_vec

_w_vec: .cproc    a, b, c, m

         .reg     ai_i1, bi_i1, pi, pi1, pi_i1, pi_s, pi1_s
         .reg     mask, bi, bi1, ci, ci1, c1, cntr

         MVK      –1,mask                ; set to all 1s to create 0xFFFFFFFF
         MVKH     0,mask                 ; clear upper 16 bits to create 0xFFFF
         MVK      50,cntr                ; cntr = 100/2
         ADD      2,c,c1                 ; point to c[1]

LOOP:    .trip 50
         LDW      .D1      *a++,ai_i1    ; ai & ai+1
         LDW      .D2      *b++,bi_i1    ; bi & bi+1
         MPY      .M1X     ai_i1,m,pi    ; m * ai
         MPYHL    .M2X     ai_i1,m,pi1   ; m * ai+1
         SHR      .S1      pi,15,pi_s    ; (m * ai) >> 15
         SHR      .S2      pi1,15,pi1_s  ; (m * ai+1) >> 15
         AND      .L2      bi_i1,mask,bi ; bi
         SHR      .S2      bi_i1,16,bi1  ; bi+1
         ADD      .L1X     pi_s,bi,ci    ; ci = (m * ai) >> 15 + bi
         ADD      .L2      pi1_s,bi1,ci1 ; ci+1 = (m * ai+1) >> 15 + bi+1
         STH      .D1      ci,*c++[2]    ; store ci
         STH      .D2      ci1,*c1++[2]  ; store ci+1
[cntr]   SUB      .L1      cntr,1,cntr   ; decrement loop counter
[cntr]   B        .S1      LOOP          ; branch to loop

         .endproc
```

6.5.8 Final Assembly

Example 6–36 shows the final assembly code for the weighted vector sum. The following optimizations are included:

- While iteration n of instruction STH ci+1 is executing, iteration n + 1 of STH ci is executing. To prevent the STH ci instruction from executing iteration 51 while STH ci + 1 executes iteration 50, execute the loop only 49 times and schedule the final executions of ADD ci+1 and STH ci+1 after exiting the loop.
- The mask for the AND instruction is created with MVK and MVKH in parallel with the loop prolog.
- The pointer to the odd elements in array c is also set up in parallel with the loop prolog.

Example 6–36. Assembly Code for Weighted Vector Sum

```
        LDW    .D1    *A4++,A2      ; ai & ai+1

        ADD    .L2X   A6,2,B0       ; set pointer to ci+1

        LDW    .D2    *B4++,B2      ; bi & bi+1
||      LDW    .D1    *A4++,A2      ;* ai & ai+1

        MVK    .S2    –1,B10        ; set to all 1s (0xFFFFFFFF)

        LDW    .D2    *B4++,B2      ;* bi & bi+1
||      LDW    .D1    *A4++,A2      ;** ai & ai+1
||      MVK    .S1    49,A1         ; set up loop counter
||      MVKH   .S2    0,B10         ; clr upper 16 bits (0x0000FFFF)

        MPY    .M1X   A2,B6,A5      ; m * ai
||[A1]  SUB    .L1    A1,1,A1       ; decrement loop counter

        MPYHL  .M2X   A2,B6,B5      ; m * ai+1
||[A1]  B      .S1    LOOP          ; branch to loop
||      LDW    .D2    *B4++,B2      ;** bi & bi+1
||      LDW    .D1    *A4++,A2      ;*** ai & ai+1

        SHR    .S1    A5,15,A7      ; (m * ai) >> 15
||      AND    .L2    B2,B10,B8     ; bi
||      MPY    .M1X   A2,B6,A5      ;* m * ai
||[A1]  SUB    .L1    A1,1,A1       ;* decrement loop counter

        SHR    .S2    B2,16,B1      ; bi+1
||      ADD    .L1X   A7,B8,A9      ; ci = (m * ai) >> 15 + bi
||      MPYHL  .M2X   A2,B6,B5      ;* m * ai+1
||[A1]  B      .S1    LOOP          ;* branch to loop
||      LDW    .D2    *B4++,B2      ;*** bi & bi+1
||      LDW    .D1    *A4++,A2      ;**** ai & ai+1

        SHR    .S2    B5,15,B7      ; (m * ai+1) >> 15
||      STH    .D1    A9,*A6++[2]   ; store ci
||      SHR    .S1    A5,15,A7      ;* (m * ai) >> 15
||      AND    .L2    B2,B10,B8     ;* bi
||[A1]  SUB    .L1    A1,1,A1       ;** decrement loop counter
||      MPY    .M1X   A2,B6,A5      ;** m * ai

LOOP:
        ADD    .L2    B7,B1,B9      ; ci+1 = (m * ai+1) >> 15 + bi+1
||      SHR    .S2    B2,16,B1      ;* bi+1
||      ADD    .L1X   A7,B8,A9      ;* ci = (m * ai) >> 15 + bi
||      MPYHL  .M2X   A2,B6,B5      ;** m * ai+1
||[A1]  B      .S1    LOOP          ;** branch to loop
||      LDW    .D2    *B4++,B2      ;**** bi & bi+1
||      LDW    .D1    *A4++,A2      ;***** ai & ai+1
```

Example 6–36. Assembly Code for Weighted Vector Sum (Continued)

```
       STH    .D2    B9,*B0++[2]   ; store ci+1
||     SHR    .S2    B5,15,B7      ;* (m * ai+1) >> 15
||     STH    .D1    A9,*A6++[2]   ;* store ci
||     SHR    .S1    A5,15,A7      ;** (m * ai) >> 15
||     AND    .L2    B2,B10,B8     ;** bi
||[A1] SUB    .L1    A1,1,A1       ;*** decrement loop counter
||     MPY    .M1X   A2,B6,A5      ;*** m * ai
       ; Branch occurs here

       ADD    .L2    B7,B1,B9      ; ci+1 = (m * ai+1) >> 15 + bi+1

       STH    .D2    B9,*B0        ; store ci+1
```

Part III

6.6 Loop Carry Paths

Loop carry paths occur when one iteration of a loop writes a value that must be read by a future iteration. A loop carry path can affect the performance of a software-pipelined loop that executes multiple iterations in parallel. Sometimes loop carry paths (instead of resources) determine the minimum iteration interval.

IIR filter code contains a loop carry path; output samples are used as input to the computation of the next output sample.

6.6.1 IIR Filter C Code

Example 6–37 shows C code for a simple IIR filter. In this example, y[i] is an input to the calculation of y[i+1]. Before y[i] can be read for the next iteration, y[i+1] must be computed from the previous iteration.

Example 6–37. IIR Filter C Code

```
void iir(short x[],short y[],short c1, short c2, short c3)
{
   int i;

   for (i=0; i<100; i++) {
         y[i+1] = (c1*x[i] + c2*x[i+1] + c3*y[i]) >> 15;
         }
}
```

Part III

6.6.2 Translating C Code to Linear Assembly (Inner Loop)

Example 6–38 shows the 'C6x instructions that execute the inner loop of the IIR filter C code. In this example:

- xptr is not postincremented after loading xi+1, because xi of the next iteration is actually xi+1 of the current iteration. Thus, the pointer points to the same address when loading both xi+1 for one iteration and xi for the next iteration.
- yptr is also not postincremented after storing yi+1, because yi of the next iteration is yi+1 for the current iteration.

Example 6–38. Linear Assembly for IIR Inner Loop

```
       LDH    *xptr++,xi    ; xi+1
       MPY    c1,xi,p0      ; c1 * xi
       LDH    *xptr,xi+1    ; xi+1
       MPY    c2,xi+1,p1    ; c2 * xi+1
       ADD    p0,p1,s0      ; c1 * xi + c2 * xi+1
       LDH    *yptr++,yi    ; yi
       MPY    c3,yi,p2      ; c3 * yi
       ADD    s0,p2,s1      ; c1 * xi + c2 * xi+1 + c3 * yi
       SHR    s1,15,yi+1    ; yi+1
       STH    yi+1,*yptr    ; store yi+1
 [cntr]SUB    cntr,1,cntr   ; decrement loop counter
 [cntr]B      LOOP          ; branch to loop
```

6.6.3 Drawing a Dependency Graph

Figure 6–15 shows the dependency graph for the IIR filter. A loop carry path exists from the store of yi+1 to the load of yi. The path between the STH and the LDH is one cycle because the load and store instructions use the same memory pipeline. Therefore, if a store is issued to a particular address on cycle n and a load from that same address is issued on the next cycle, the load reads the value that was written by the store instruction.

Figure 6–15. Dependency Graph of IIR Filter

Note: The shaded numbers show the loop carry path: 5 + 2 + 1 + 1 + 1 = 10.

6.6.4 Determining the Minimum Iteration Interval

To determine the minimum iteration interval, you must consider both resources and data dependency constraints. Based on resources in Table 6–16, the minimum iteration interval is 2.

Note:

There are six non-.M units available: three on the A side (.S1, .D1, .L1) and three on the B side (.S2, .D2, .L2). Therefore, to determine resource constraints, divide the total number of non-.M units used on each side by 3 (3 is the total number of non-.M units available on each side).

Based on non-.M unit resources in Table 6–16, the minimum iteration interval for the IIR filter is 2 because the total non-.M units on the A side is 5 (5 ÷ 3 is greater than 1 so you round up to the next whole number). The B side uses only three non-.M units, so this does not affect the minimum iteration interval, and no other unit is used more than twice.

Table 6–16. Resource Table for IIR Filter

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	2 MPYs	2
.S1	B	1
.D1	2 LDHs	2
.L1,.S1, or .D1	ADD & SUB	2
Total non-.M units		5

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	MPY	1
.S2	SHR	1
.D2	STH	1
.L2 or .S2, .D2	ADD	1
Total non-.M units		3

However, the IIR has a data dependency constraint defined by its loop carry path. Figure 6–15 shows that if you schedule LDH yi on cycle 0:

- The earliest you can schedule MPY p2 is on cycle 5.
- The earliest you can schedule ADD s1 is on cycle 7.
- SHR yi+1 must be on cycle 8 and STH on cycle 9.
- Because the LDH must wait for the STH to be issued, the earliest the the second iteration can begin is cycle 10.

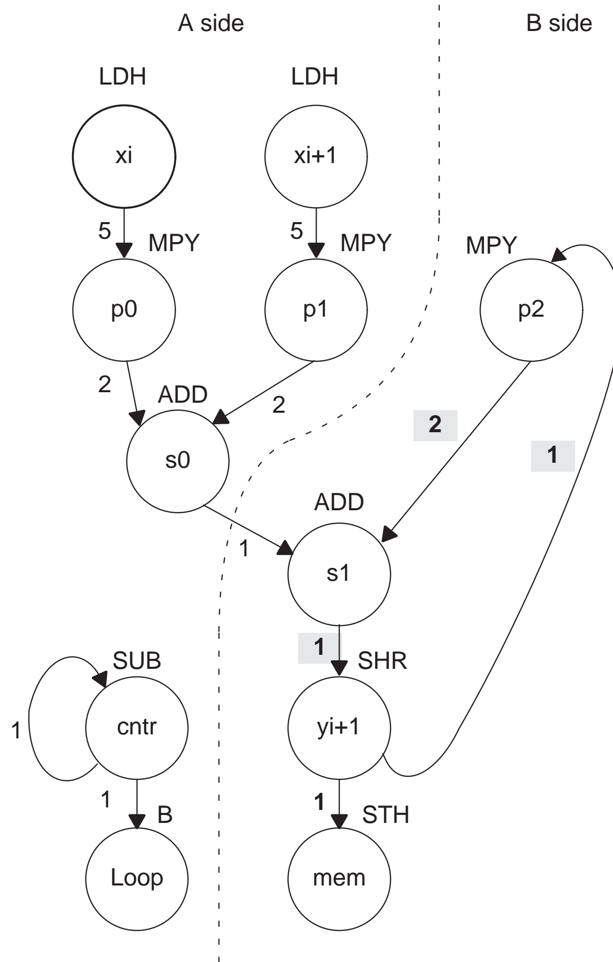
To determine the minimum loop carry path, add all of the numbers along the loop paths in the dependency graph. This means that this loop carry path is 10 (5 + 2 + 1 + 1 + 1).

Although the minimum iteration interval is the greater of the resource limits and data dependency constraints, an interval of 10 seems slow. Figure 6–16 shows how to improve the performance.

6.6.4.1 Drawing a New Dependency Graph

Figure 6–16 shows a new graph with a loop carry path of 4 (2 +1 + 1). because the MPY p2 instruction can read yi+1 while it is still in a register, you can reduce the loop carry path by six cycles. LDH yi is no longer in the graph. Instead, you can issue LDH y[0] once outside the loop. In every iteration after that, the y+1 values written by the SHR instruction are valid y inputs to the MPY instruction.

Figure 6–16. Dependency Graph of IIR Filter (With Smaller Loop Carry)



Note: The shaded numbers show the loop carry path: 2 + 1 + 1 = 4.

6.6.4.2 New 'C6x Instructions (Inner Loop)

Example 6–39 shows the new linear assembly from the graph in Figure 6–16, where LDH yi was removed. The one variable y that is read and written is yi for the MPY p2 instruction and yi+1 for the SHR and STH instructions.

Example 6–39. Linear Assembly for IIR Inner Loop With Reduced Loop Carry Path

```
      LDH    *xptr++,xi    ; xi+1
      MPY    c1,xi,p0      ; c1 * xi
      LDH    *xptr,xi+1    ; xi+1
      MPY    c2,xi+1,p1    ; c2 * xi+1
      ADD    p0,p1,s0      ; c1 * xi + c2 * xi+1
      MPY    c3,y,p2       ; c3 * yi
      ADD    s0,p2,s1      ; c1 * xi + c2 * xi+1 + c3 * yi
      SHR    s1,15,y       ; yi+1
      STH    y,*yptr++     ; store yi+1
[cntr]SUB    cntr,1,cntr   ; decrement loop counter
[cntr]B      LOOP          ; branch to loop
```

6.6.5 Linear Assembly Resource Allocation

Example 6–40 shows the same linear assembly instructions as those in Example 6–39 with the functional units and registers assigned.

Example 6–40. Linear Assembly for IIR Inner Loop (With Allocated Resources)

```
      LDH    .D1    *A4++,A2      ; xi+1
      MPY    .M1    A6,A2,A5      ; c1 * xi
      LDH    .D1    *A4,A3        ; xi+1
      MPY    .M1X   B6,A3,A7      ; c2 * xi+1
      ADD    .L1    A5,A7,A9      ; c1 * xi + c2 * xi+1
      MPY    .M2X   A8,B2,B3      ; c3 * yi
      ADD    .L2X   B3,A9,B5      ; c1 * xi + c2 * xi+1 + c3 * yi
      SHR    .S2    B5,15,B2      ; yi+1
      STH    .D2    B2,*B4++      ; store yi+1
 [A1] SUB    .L1    A1,1,A1       ; decrement loop counter
 [A1] B      .S1    LOOP          ; branch to loop
```

6.6.6 Modulo Iteration Interval Scheduling

Table 6–17 shows the modulo iteration interval table for the IIR filter. The SHR instruction on cycle 10 finishes in time for the MPY p2 instruction from the next iteration to read its result on cycle 11.

Table 6–17. Modulo Iteration Interval Table for IIR (4-Cycle Loop)

Unit/Cycle	0	4	8, 12, 16, ...	Unit/Cycle	1	5	9, 13, 17, ...
.D1	LDH xi	* LDH xi	** LDH xi	.D1	LDH xi+1	* LDH xi+1	** LDH ci+1
.D2			ADD s0	.D2			
.M1				.M1		MPY p0	* MPY p0
.M2				.M2			
.L1				.L1		SUB cntr	* SUB cntr
.L2				.L2			ADD s1
.S1				.S1			
.S2				.S2			
1X				1X			
2X				2X			ADD s1
Unit/Cycle	**2**	**6**	**10, 14, 18, ...**	**Unit/Cycle**	**3**	**7**	**11, 15, 19, ...**
.D1				.D1			
.D2				.D2			STH yi+1
.M1		MPY p1	* MPY p1	.M1			
.M2				.M2		MPY p2	* **MPY p2**
.L1				.L1			
.L2				.L2			
.S1		B LOOP	* B LOOP	.S1			
.S2			**SHR yi+1**	.S2			
1X		MPY p1	* MPY p1	1X			
2X				2X		MPY p2	* MPY p2

Note: The asterisks indicate the iteration of the loop.

6.6.7 Using the Assembly Optimizer for the IIR Filter

Example 6–41 shows the linear assembly code to perform the IIR filter. Once again, you can use this code as input to the assembly optimizer to create a software-pipelined loop instead of scheduling this by hand.

Example 6–41. Linear Assembly for IIR Filter

```
       .global _iir

_iir:  .cproc  x, y, c1, c2, c3

       .reg     xi, xi1, yi1
       .reg     p0, p1, p2, s0, s1, cntr

       MVK      100,cntr                 ; cntr = 100

       LDH      .D2  *y++,yi1            ; yi+1

LOOP:  .trip 100
       LDH      .D1  *x++,xi             ; xi
       MPY      .M1  c1,xi,p0            ; c1 * xi
       LDH      .D1  *x,xi1              ; xi+1
       MPY      .M1X c2,xi1,p1           ; c2 * xi+1
       ADD      .L1  p0,p1,s0            ; c1 * xi + c2 * xi+1
       MPY      .M2X c3,yi1,p2           ; c3 * yi
       ADD      .L2X s0,p2,s1            ; c1 * xi + c2 * xi+1 + c3 * yi
       SHR      .S2  s1,15,yi1           ; yi+1
       STH      .D2  yi1,*y++            ; store yi+1
[cntr] SUB      .L1  cntr,1,cntr         ; decrement loop counter
[cntr] B        .S1  LOOP                ; branch to loop

       .endproc
```

6.6.8 Final Assembly

Example 6–42 shows the final assembly for the IIR filter. With one load of y[0] outside the loop, no other loads from the y array are needed. Example 6–42 requires 408 cycles: $(4 \times 100) + 8$.

Example 6–42. Assembly Code for IIR Filter

```
       LDH    .D1    *A4++,A2      ; xi

       LDH    .D1    *A4,A3        ; xi+1

       LDH    .D2    *B4++,B2      ; load y[0] outside of loop

       MVK    .S1    100,A1        ; set up loop counter

       LDH    .D1    *A4++,A2      ;* xi

  [A1] SUB    .L1    A1,1,A1       ; decrement loop counter
||     MPY    .M1    A6,A2,A5      ; c1 * xi
||     LDH    .D1    *A4,A3        ;* xi+1

       MPY    .M1X   B6,A3,A7      ; c2 * xi+1
||[A1] B      .S1    LOOP          ; branch to loop

       MPY    .M2X   A8,B2,B3      ; c3 * yi

LOOP:
       ADD    .L1    A5,A7,A9      ; c1 * xi + c2 * xi+1
||     LDH    .D1    *A4++,A2      ;** xi

       ADD    .L2X   B3,A9,B5      ; c1 * xi + c2 * xi+1 + c3 * yi
||[A1] SUB    .L1    A1,1,A1       ;* decrement loop counter
||     MPY    .M1    A6,A2,A5      ;* c1 * xi
||     LDH    .D1    *A4,A3        ;** xi+1

       SHR    .S2    B5,15,B2      ; yi+1
||     MPY    .M1X   B6,A3,A7      ;* c2 * xi+1
||[A1] B      .S1    LOOP          ;* branch to loop

       STH    .D2    B2,*B4++      ; store yi+1
||     MPY    .M2X   A8,B2,B3      ;* c3 * yi
       ; Branch occurs here
```

6.7 If-Then-Else Statements in a Loop

If-then-else statements in C cause certain instructions to execute when the if condition is true and other instructions to execute when it is false. One way to accomplish this in linear assembly code is with conditional instructions. because all ’C6x instructions can be conditional on one of five general-purpose registers, conditional instructions can handle both the true and false cases of the if-then-else C statement.

6.7.1 If-Then-Else C Code

Example 6–43 contains a loop with an if-then-else statement. You either add a[i] to sum or subtract a[i] from sum.

Example 6–43. If-Then-Else C Code

```
int if_then(short a[], int codeword, int mask, short theta)
{
  int i,sum, cond;

  sum = 0;
  for (i = 0; i < 32; i++){
       cond = codeword & mask;
       if (theta  ==  !(!(cond)))
            sum += a[i];
       else
            sum -= a[i];
       mask = mask << 1;
       }
  return(sum);
}
```


Branching is one way to execute the if-then-else statement: branch to the ADD when the if statement is true and branch to the SUB when the if statement is false. However, because each branch has five delay slots, this method requires additional cycles. Furthermore, branching within the loop makes software pipelining almost impossible.

Using conditional instructions, on the other hand, eliminates the need to branch to the appropriate piece of code after checking whether the condition is true or false. Simply program both the ADD and SUB as usual, but make them conditional on the zero and nonzero values of a condition register. This method also allows you to software pipeline the loop and achieve much better performance than you would with branching.

6.7.2 Translating C Code to Linear Assembly

Example 6–44 shows the linear assembly instructions needed to execute inner loop of the C code in Example 6–43.

Example 6–44. Linear Assembly for If-Then-Else Inner Loop

```
      AND    codeword,mask,cond  ; cond = codeword & mask
[cond]MVK    1,cond              ; !(!(cond))
      CMPEQ  theta,cond,if       ; (theta == !(!(cond)))
      LDH    *aptr++,ai          ; a[i]
[if]  ADD    sum,ai,sum          ; sum += a[i]
[!if] SUB    sum,ai,sum          ; sum -= a[i]
      SHL    mask,1,mask         ; mask = mask << 1;

[cntr]ADD    -1,cntr,cntr        ; decrement counter
[cntr]B      LOOP                ; for LOOP
```

CMPEQ is used to create IF. The ADD is conditional when IF is nonzero (corresponds to then); the SUB is conditional when IF is 0 (corresponds to else).

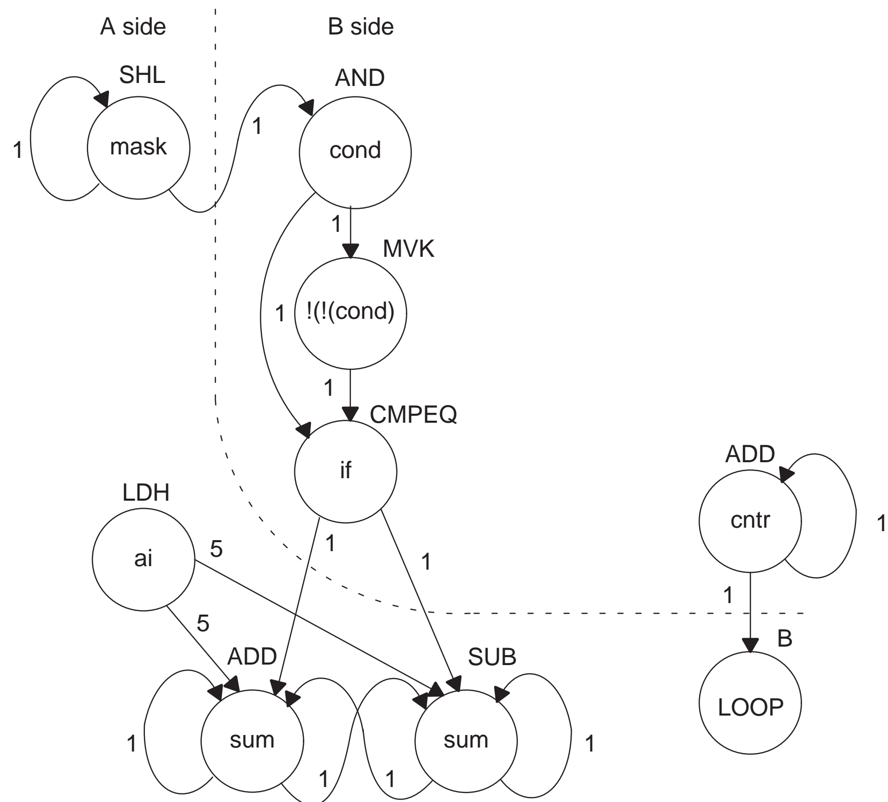
A conditional MVK performs the !(!(cond)) C statement. If the result of the bitwise AND is nonzero, a 1 is written into cond; if the result of the AND is 0, cond remains at 0.

6.7.3 Drawing a Dependency Graph

Figure 6–17 shows the dependency graph for the if-then-else C code. This graph illustrates the following arrangement:

- Two nodes on the graph contain sum: one for the ADD and one for the SUB. Because some iterations are performing an ADD and others are performing a SUB, each of these nodes is a possible input to the next iteration of either node.
- The LDH ai instruction is a parent of both ADD sum and SUB sum, because both instructions read ai.
- CMPEQ if is also a parent to ADD sum and SUB sum, because both read IF for the conditional execution.
- The result of SHL mask is read on the next iteration by the AND cond instruction.

Figure 6–17. Dependency Graph of If-Then-Else Code



6.7.4 Determining the Minimum Iteration Interval

With nine instructions, the minimum iteration interval is at least 2, because a maximum of eight instructions can be in parallel. Based on the way the dependency graph in Figure 6–17 is split, five instructions are on the A side and four are on the B side. Because none of the instructions are MPYs, all instructions must go on the .S, .D, or .L units, which means you have a total of six resources.

- LDH must be on a .D unit.
- SHL, B, and MVK must be on a .S unit.
- The ADDs and SUB can be on the .S, .L, or .D units.
- The AND can be on a .S or .L unit.

From Table 6–18, you can see that no one resource is used more than two times, so the minimum iteration interval is still 2.

Table 6–18. Resource Table for If-Then-Else Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1		0
.S1	SHL & B	2
.D1	LDH	1
.L1, .S1, or .D1	ADD & SUB	2
Total non-.M units		5

(b) B side

Unit(s)	Instructions	Total/Unit
.M2		0
.S2	MVK	1
.L2	CMPEQ	1
.L2 or .S2	AND	1
.L2, .S2, or .D2	ADD	1
Total non-.M units		4

The minimum iteration interval is also affected by the total number of instructions. Because three units can perform nonmultiply operations on a given side, a total of five instructions can be performed with a minimum iteration interval of 2. Because only four instructions are on the B side, the minimum iteration interval is still 2.

Part III

6.7.5 Linear Assembly Resource Allocation

Now that the graph is split and you know the minimum iteration interval, you can allocate functional units and registers to the instructions. You must ensure that no resource is used more than twice.

Example 6–45 shows the linear assembly with the functional units and registers that are used in the inner loop.

Example 6–45. Linear Assembly for Full If-Then-Else Code

```
         .global _if_then

_if_then: .cproc  a, cword, mask, theta

         .reg     cond, if, ai, sum, cntr

         MVK      32,cntr                 ; cntr = 32
         ZERO     sum                     ; sum = 0

LOOP:    .trip 32
         AND      .S2X     cword,mask,cond ; cond = codeword & mask
 [cond]  MVK      .S2      1,cond          ; !(!(cond))
         CMPEQ    .L2      theta,cond,if   ; (theta == !(!(cond)))
         LDH      .D1      *a++,ai         ; a[i]
   [if]  ADD      .L1      sum,ai,sum      ; sum += a[i]
  [!if]  SUB      .D1      sum,ai,sum      ; sum -= a[i]
         SHL      .S1      mask,1,mask     ; mask = mask << 1;
 [cntr]  ADD      .L2      -1,cntr,cntr    ; decrement counter
 [cntr]  B        .S1      LOOP            ; for LOOP

         .return  sum

         .endproc
```

Part III

6.7.6 Final Assembly

Example 6–46 shows the final assembly code after software pipelining. The performance of this loop is 70 cycles ($2 \times 32 + 6$).

Example 6–46. Assembly Code for If-Then-Else

```
       MVK    .S2    32,B0         ; set up loop counter

  [B0] ADD    .L2    -1,B0,B0      ; decrement counter

  [B0] ADD    .L2    -1,B0,B0      ; decrement counter
||[B0] B      .S1    LOOP          ; for LOOP
||     LDH    .D1    *A4++,A5      ; a[i]

       SHL    .S1    A6,1,A6       ; mask = mask << 1;
||     AND    .S2X   B4,A6,B2      ; cond = codeword & mask

  [B2] MVK    .S2    1,B2          ; !(!(cond))
||[B0] ADD    .L2    -1,B0,B0      ; decrement counter
||[B0] B      .S1    LOOP          ;* for LOOP
||     LDH    .D1    *A4++,A5      ;* a[i]

       CMPEQ  .L2    B6,B2,B1      ; (theta == !(!(cond)))
||     SHL    .S1    A6,1,A6       ;* mask = mask << 1;
||     AND    .S2X   B4,A6,B2      ;* cond = codeword & mask
||     ZERO   .L1    A7            ; zero out accumulator

LOOP:
  [B0] ADD    .L2    -1,B0,B0      ; decrement counter
||[B2] MVK    .S2    1,B2          ;* !(!(cond))
||[B0] B      .S1    LOOP          ;** for LOOP
||     LDH    .D1    *A4++,A5      ;** a[i]

  [B1] ADD    .L1    A7,A5,A7      ; sum += a[i]
||[!B1]SUB    .D1    A7,A5,A7      ; sum -= a[i]
||     CMPEQ  .L2    B6,B2,B1      ;* (theta == !(!(cond)))
||     SHL    .S1    A6,1,A6       ;** mask = mask << 1;
||     AND    .S2X   B4,A6,B2      ;** cond = codeword & mask
       ; Branch occurs here
```

Part III

6.7.7 Comparing Performance

You can improve the performance of the code in Example 6–46 if you know that the loop count is at least 3. If the loop count is at least 3, remove the decrement counter instructions outside the loop and put the MVK (for setting up the loop counter) in parallel with the first branch. These two changes save two cycles at the beginning of the loop prolog.

The first two branches are now unconditional, because the loop count is at least 3 and you know that the first two branches must execute. To account for the removal of the three decrement-loop-counter instructions, set the loop counter to 3 fewer than the actual number of times you want the loop to execute: in this case, 29 (32 – 3).

Example 6–47. Assembly Code for If-Then-Else With Loop Count Greater Than 3

```
        B       .S1    LOOP          ; for LOOP
||      LDH     .D1    *A4++,A5      ; a[i]
||      MVK     .S2    29,B0         ; set up loop counter

        SHL     .S1    A6,1,A6       ; mask = mask << 1;
||      AND     .S2X   B4,A6,B2      ; cond = codeword & mask

  [B2]  MVK     .S2    1,B2          ; !(!(cond))
||      B       .S1    LOOP          ;* for LOOP
||      LDH     .D1    *A4++,A5      ;* a[i]

        CMPEQ   .L2    B6,B2,B1      ; (theta == !(!(cond)))
||      SHL     .S1    A6,1,A6       ;* mask = mask << 1;
||      AND     .S2X   B4,A6,B2      ;* cond = codeword & mask
||      ZERO    .L1    A7            ; zero out accumulator

LOOP:
  [B0]  ADD     .L2    -1,B0,B0      ; decrement counter
||[B2]  MVK     .S2    1,B2          ;* !(!(cond))
||[B0]  B       .S1    LOOP          ;** for LOOP
||      LDH     .D1    *A4++,A5      ;** a[i]

  [B1]  ADD     .L1    A7,A5,A7      ; sum += a[i]
||[!B1]SUB      .D1    A7,A5,A7      ; sum -= a[i]
||      CMPEQ   .L2    B6,B2,B1      ;* (theta == !(!(cond)))
||      SHL     .S1    A6,1,A6       ;** mask = mask << 1;
||      AND     .S2X   B4,A6,B2      ;** cond = codeword & mask
        ; Branch occurs here
```

Example 6–47 shows the improved loop with a cycle count of 68 ($2 \times 32 + 4$). Table 6–19 compares the performance of Example 6–46 and Example 6–47.

Table 6–19. Comparison of If-Then-Else Code Examples

Code Example		Cycles	Cycle Count
Example 6–46	If-then-else assembly code	$(2 \times 32) + 6$	70
Example 6–47	If-then-else assembly code with loop count greater than 3	$(2 \times 32) + 4$	68

6.8 Loop Unrolling

Even though the performance of the previous example is good, it can be improved. When resources are not fully used, you can improve performance by unrolling the loop. In Example 6–48, only nine instructions execute every two cycles. If you unroll the loop and analyze the new minimum iteration interval, you have room to add instructions. A minimum iteration interval of 3 provides a 25% improvement in throughput: three cycles to do two iterations, rather than the four cycles required in Example 6–47.

6.8.1 Unrolled If-Then-Else C Code

Example 6–48 shows the unrolled version of the if-then-else C code in Example 6–43 on page 6-83.

Example 6–48. If-Then-Else C Code (Unrolled)

```
int unrolled_if_then(short a[], int codeword, int mask, short theta)
{
  int i,sum, cond;

  sum = 0;
  for (i = 0; i < 32; i+=2){
       cond = codeword & mask;
       if (theta  ==  !(!(cond)))
           sum += a[i];
       else
           sum -= a[i];
       mask = mask << 1;

       cond = codeword & mask;
       if (theta  ==  !(!(cond)))
           sum += a[i+1];
       else
           sum -= a[i+1];
       mask = mask << 1;
       }
  return(sum);
}
```

Part III

6.8.2 Translating C Code to Linear Assembly

Example 6–49 shows the unrolled inner loop with 16 instructions and the possibility of achieving a loop with a minimum iteration interval of 3.

Example 6–49. Linear Assembly for Unrolled If-Then-Else Inner Loop

```
         AND        codeword,maski,condi     ; condi = codeword & maski
  [condi]MVK        1,condi                  ; !(!(condi))
         CMPEQ      theta,condi,ifi          ; (theta == !(!(condi)))
         LDH        *aptr++,ai               ; a[i]
  [ifi]  ADD        sumi,ai,sumi             ; sum += a[i]
  [!ifi] SUB        sumi,ai,sumi             ; sum -= a[i]
         SHL        maski,1,maski+1          ; maski+1 = maski << 1;

         AND        codeword,maski+1,condi+1 ; condi+1 = codeword & maski+1
[condi+1]MVK        1,condi+1                ; !(!(condi+1))
         CMPEQ      theta,condi+1,ifi+1      ; (theta == !(!(condi+1)))
         LDH        *aptr++,ai+1             ; a[i+!]
 [ifi+1] ADD        sumi+1,ai+1,sumi+1       ; sum += a[i+1]
[!ifi+1] SUB        sumi+1,ai+1,sumi+1       ; sum -= a[i+1]
         SHL        maski+1,1,maski          ; maski = maski+1 << 1;

 [cntr]  ADD        -1,cntr,cntr             ; decrement counter
 [cntr]  B          LOOP                     ; for LOOP
```

Part III

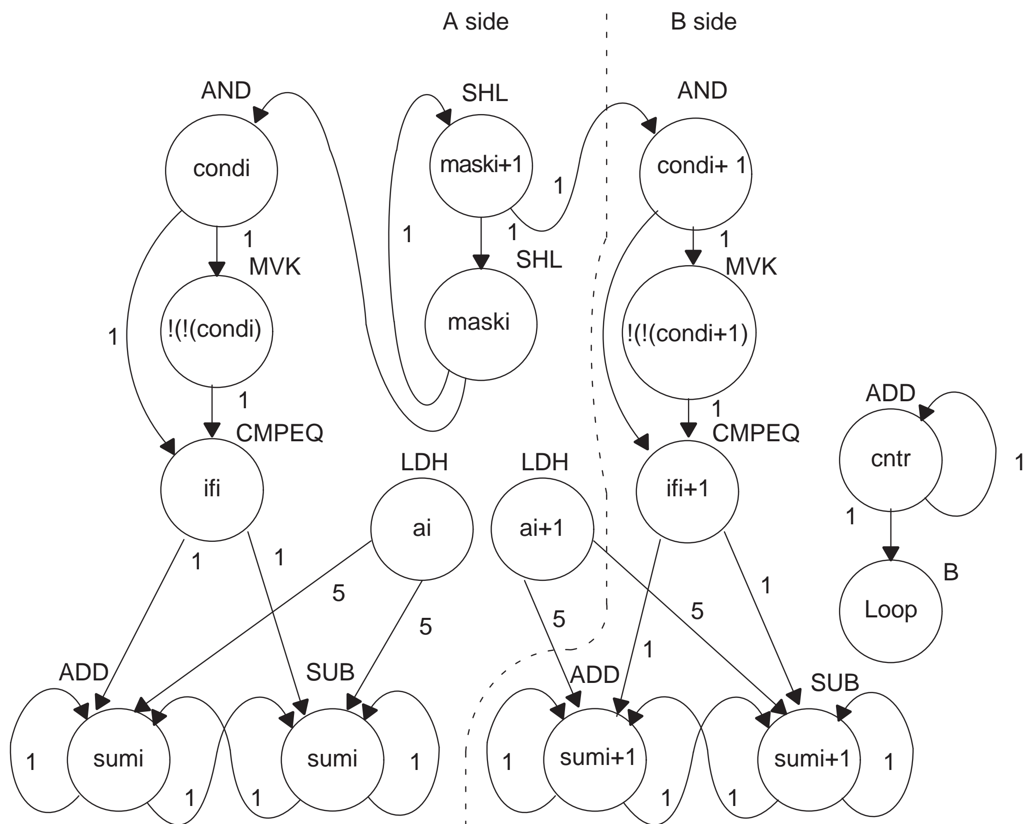
6.8.3 Drawing a Dependency Graph

Although there are numerous ways to split the dependency graph, the main goal is to achieve a minimum iteration interval of 3 and meet these conditions:

- You cannot have more than nine non-.M instructions on either side.
- Only three non-.M instructions can execute per cycle.

Figure 6–18 shows the dependency graph for the unrolled if-then-else code. Nine instructions are on the A side, and seven instructions are on the B side.

Figure 6–18. Dependency Graph of If-Then-Else Code (Unrolled)



6.8.4 Determining the Minimum Iteration Interval

With 16 instructions, the minimum iteration interval is at least 3 because a maximum of six instructions can be in parallel with the following allocation possibilities:

- LDH must be on a .D unit.
- SHL, B, and MVK must be on a .S unit.
- The ADDs and SUB can be on a .S, .L, or .D unit.
- The AND can be on a .S or .L unit.

From Table 6–20, you can see that no one resource is used more than three times so that the minimum iteration interval is still 3.

Checking the total number of non-.M instructions on each side shows that a total of nine instructions can be performed with the minimum iteration interval of 3. because only seven non-.M instructions are on the B side, the minimum iteration interval is still 3.

Table 6–20. Resource Table for Unrolled If-Then-Else Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1		0
.S1	MVK and 2 SHLs	3
.D1	2 LDHs	2
.L1	CMPEQ	1
.L1 or .S1	AND	1
.L1, .S1, or .D1	ADD and SUB	2
Total non-.M units		9

(b) B side

Unit(s)	Instructions	Total/Unit
.M2		0
.S2	MVK and B	2
.L2	CMPEQ	1
.L2 pr.S2	AND	1
.L2 ,.S2, or .D2	SUB and 2 ADDs	3
Total non-.M units		7

6.8.5 Linear Assembly Resource Allocation

Now that the graph is split and you know the minimum iteration interval, you can allocate functional units and registers to the instructions. You must ensure no resource is used more than three times.

Example 6–50 shows the linear assembly code with the functional units and registers.

Example 6–50. Linear Assembly for Full Unrolled If-Then-Else Code

```
         .global    _unrolled_if_then

_unrolled_if_then:  .cproc   a, cword, mask, theta

         .reg   cword, mask, theta, ifi, ifi1, a, ai, ai1, cntr
         .reg   cdi, cdi1, sumi, sumi1, sum

         MV     A4,a                ; C callable register for 1st operand
         MV     B4,cword            ; C callable register for 2nd operand
         MV     A6,mask             ; C callable register for 3rd operand
         MV     B6,theta            ; C callable register for 4th operand
         MVK    16,cntr             ; cntr = 32/2
         ZERO   sumi                ; sumi = 0
         ZERO   sumi1               ; sumi+1 = 0

LOOP:    .trip 32
         AND    .L1X cword,mask,cdi  ; cdi = codeword & maski
  [cdi]  MVK    .S1  1,cdi           ; !(!(cdi))
         CMPEQ  .L1X theta,cdi,ifi   ; (theta == !(!(cdi)))
         LDH    .D1  *a++,ai         ; a[i]
  [ifi]  ADD    .L1  sumi,ai,sumi    ; sum += a[i]
 [!ifi]  SUB    .D1  sumi,ai,sumi    ; sum -= a[i]
         SHL    .S1  mask,1,mask     ; maski+1 = maski << 1;

         AND    .L2X cword,mask,cdi1 ; cdi+1 = codeword & maski+1
 [cdi1]  MVK    .S2  1,cdi1          ; !(!(cdi+1))
         CMPEQ  .L2  theta,cdi1,ifi1 ; (theta == !(!(cdi+1)))
         LDH    .D1  *a++,ai1        ; a[i+1]
 [ifi1]  ADD    .L2  sumi1,ai1,sumi1 ; sum += a[i+1]
[!ifi1]  SUB    .D2  sumi1,ai1,sumi1 ; sum -= a[i+1]
         SHL    .S1  mask,1,mask     ; maski = maski+1 << 1;

 [cntr]  ADD    .D2  -1,cntr,cntr    ; decrement counter
 [cntr]  B      .S2  LOOP            ; for LOOP

         ADD    sumi,sumi1,sum      ; Add sumi and sumi+1 for ret value

         .return sum

         .endproc
```

Part III

6.8.6 Final Assembly

Example 6–51 shows the final assembly code after software pipelining. The cycle count of this loop is now 53: (3×16) + 5.

Example 6–51. Assembly Code for Unrolled If-Then-Else

```
       MVK    .S2    16,B0         ; set up loop counter

       LDH    .D1    *A4++,A5      ; a[i]
||[B0] ADD    .D2    -1,B0,B0      ; decrement counter

       LDH    .D1    *A4++,B5      ; a[i+1]
||[B0] B      .S2    LOOP          ; for LOOP
||[B0] ADD    .D2    -1,B0,B0      ; decrement counter
||     SHL    .S1    A6,1,A6       ; maski+1 = maski << 1;
||     AND    .L1X   B4,A6,A2      ; condi = codeword & maski

  [A2] MVK    .S1    1,A2          ; !(!(condi))
||     AND    .L2X   B4,A6,B2      ; condi+1 = codeword & maski+1
||     ZERO   .L1    A7            ; zero accumulator

  [B2] MVK    .S2    1,B2          ; !(!(condi+1))
||     CMPEQ  .L1X   B6,A2,A1      ; (theta == !(!(condi)))
||     SHL    .S1    A6,1,A6       ; maski = maski+1 << 1;
||     LDH    .D1    *A4++,A5      ;* a[i]
||     ZERO   .L2    B7            ; zero accumulator

LOOP:
       CMPEQ  .L2    B6,B2,B1      ; (theta == !(!(condi+1)))
||[B0] ADD    .D2    -1,B0,B0      ; decrement counter
||     LDH    .D1    *A4++,B5      ;* a[i+1]
||[B0] B      .S2    LOOP          ;* for LOOP
||     SHL    .S1    A6,1,A6       ;* maski+1 = maski << 1;
||     AND    .L1X   B4,A6,A2      ;* condi = codeword & maski

  [A1] ADD    .L1    A7,A5,A7      ; sum += a[i]
||[!A1]SUB    .D1    A7,A5,A7      ; sum -= a[i]
||[A2] MVK    .S1    1,A2          ;* !(!(condi))
||     AND    .L2X   B4,A6,B2      ;* condi+1 = codeword & maski+1

  [B1] ADD    .L2    B7,B5,B7      ; sum += a[i+1]
||[!B1]SUB    .D2    B7,B5,B7      ; sum -= a[i+1]
||[B2] MVK    .S2    1,B2          ;* !(!(condi+1))
||     CMPEQ  .L1X   B6,A2,A1      ;* (theta == !(!(condi)))
||     SHL    .S1    A6,1,A6       ;* maski = maski+1 << 1;
||     LDH    .D1    *A4++,A5      ;** a[i]
       ; Branch occurs here

       ADD    .L1X   A7,B7,A4      ; move to return register
```

Part III

6.8.7 Comparing Performance

Table 6–21 compares the performance of all versions of the if-then-else code examples.

Table 6–21. Comparison of If-Then-Else Code Examples

Code Example		Cycles	Cycle Count
Example 6–46	If-then-else assembly code	(2 × 32) + 6	70
Example 6–47	If-then-else assembly code with loop count greater than 3	(2 × 32) + 4	68
Example 6–51	Unrolled if-then-else assembly code	(3 × 16) + 5	53

6.9 Live-Too-Long Issues

When the result of a parent instruction is live longer than the minimum iteration interval of a loop, you have a live-too-long problem. Because each instruction executes every iteration interval cycle, the next iteration of that parent overwrites the register with a new value before the child can read it. Section 6.5.6.1, *Resource Conflicts,* on page 6-61 showed how to solve this problem simply by moving the parent to a later cycle. This is not always a valid solution.

6.9.1 C Code With Live-Too-Long Problem

Example 6–52 shows C code with a live-too-long problem that cannot be solved by rescheduling the parent instruction. Although it is not obvious from the C code, the dependency graph in Figure 6–19 on page 6-100 shows a *split-join* path that causes this live-too-long problem.

Example 6–52. Live-Too-Long C Code

```
int live_long(short a[],short b[],short c, short d, short e)
{
  int i,sum0,sum1,sum,a0,a2,a3,b0,b2,b3;
  short a1,b1;

  sum0 = 0;
  sum1 = 0;
  for(i=0; i<100; i++){
       a0 = a[i] * c;
       a1 = a0 >> 15;
       a2 = a1 * d;
       a3 = a2 + a0;
       sum0 += a3;
       b0 = b[i] * c;
       b1 = b0 >> 15;
       b2 = b1 * e;
       b3 = b2 + b0;
       sum1 += b3;
       }
  sum = sum0 + sum1;
  return(sum);
}
```

6.9.2 Translating C Code to Linear Assembly

Example 6–53 shows the assembly instructions that execute the inner loop in Example 6–52.

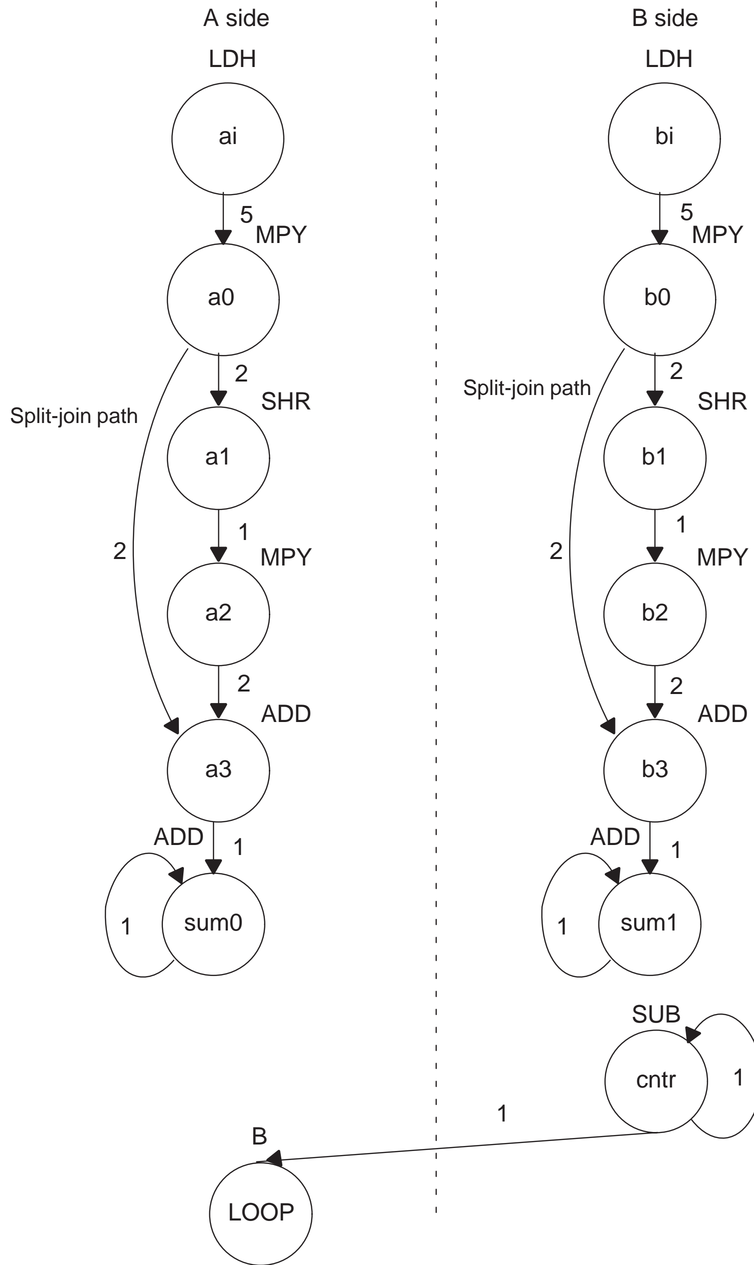
Example 6–53. Linear Assembly for Live-Too-Long Inner Loop

```
      LDH    *aptr++,ai    ; load ai from memory
      LDH    *bptr++,bi    ; load bi from memory
      MPY    ai,c,a0       ; a0 = ai * c
      SHR    a0,15,a1      ; a1 = a0 >> 15
      MPY    a1,d,a2       ; a2 = a1 * d
      ADD    a2,a0,a3      ; a3 = a2 + a0
      ADD    sum0,a3,sum0  ; sum0 += a3
      MPY    bi,c,b0       ; b0 = bi * c
      SHR    b0,15,b1      ; b1 = b0 >> 15
      MPY    b1,e,b2       ; b2 = b1 * e
      ADD    b2,b0,b3      ; b3 = b2 + b0
      ADD    sum1,b3,sum1  ; sum1 += b3

 [cntr]SUB   cntr,1,cntr   ; decrement loop counter
 [cntr]B     LOOP          ; branch to loop
```

6.9.3 Drawing a Dependency Graph

Figure 6–19 shows the dependency graph for the live-too-long code. This algorithm includes three separate and independent graphs. Two of the independent graphs have split-join paths: from a0 to a3 and from b0 to b3.

Figure 6–19. Dependency Graph of Live-Too-Long Code



Part III

6.9.4 Determining the Minimum Iteration Interval

Table 6–22 shows the functional unit resources for the loop. Based on the resource usage, the minimum iteration interval is 2 for the following reasons:

- No specific resource is used more than twice, implying a minimum iteration interval of 2.
- A total of five non-.M units on each side also implies a minimum iteration interval of 2, because three non-.M units can be used on a side during each cycle.

Table 6–22. Resource Table for Live-Too-Long Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	MPY	1
.S1	B and SHR	2
.D1	LDH	1
.L1, .S1, or .D1	2 ADDs	2
Total non-.M units		5

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	MPY	1
.S2	SHR	1
.D2	LDH	1
.L2, .S2, or .D2	2 ADDs and SUB	3
Total non-.M units		5

However, the minimum iteration interval is determined by both resources and data dependency. A loop carry path determined the minimum iteration interval of the IIR filter in section 6.6, *Loop Carry Paths*, on page 6-74. In this example, a live-too-long problem determines the minimum iteration interval.

6.9.4.1 Split-Join-Path Problems

In Figure 6–19, the two split-join paths from a0 to a3 and from b0 to b3 create the live-too-long problem. Because the ADD a3 instruction cannot be scheduled until the SHR a1 and MPY a2 instructions finish, a0 must be live for at least four cycles. For example:

- If MPY a0 is scheduled on cycle 5, then the earliest SHR a1 can be scheduled is cycle 7.
- The earliest MPY a2 can be scheduled is cycle 8.
- The earliest ADD a3 can be scheduled is cycle 10.

Because a0 is written at the end of cycle 6, it must be live from cycle 7 to cycle 10, or four cycles. No value can be live longer than the minimum iteration interval, because the next iteration of the loop will overwrite that value before the current iteration can read the value. Therefore, if the value has to be live for four cycles, the minimum iteration interval must be at least 4. A minimum iteration interval of 4 means that the loop executes at half the performance that it could based on available resources.

6.9.4.2 Unrolling the Loop

One way to solve this problem is to unroll the loop, so that you are doing twice as much work in each iteration. After unrolling, the minimum iteration interval is 4, based on both the resources and the data dependencies of the split-join path. Although unrolling the loop allows you to achieve the highest possible loop throughput, unrolling the loop does increase the code size.

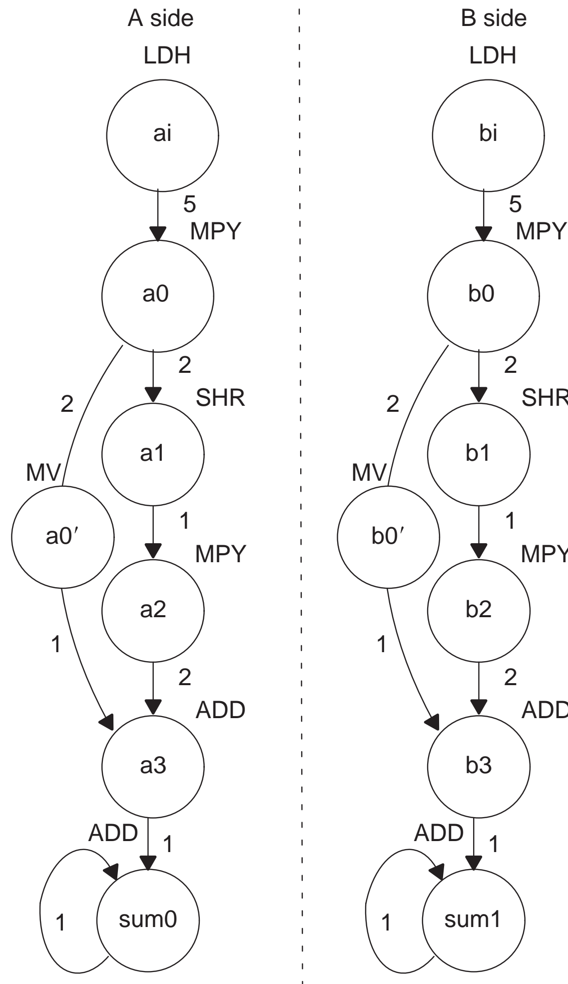
6.9.4.3 Inserting Moves

Another solution to the live-too-long problem is to break up the lifetime of a0 and b0 by inserting move (MV) instructions. The MV instruction breaks up the left path of the split-join path into two smaller pieces.

6.9.4.4 Drawing a New Dependency Graph

Figure 6–20 shows the new dependency graph with the MV instructions. Now the left paths of the split-join paths are broken into two pieces. Each value, a0 and a0', can be live for minimum iteration interval cycles. If MPY a0 is scheduled on cycle 5 and ADD a3 is scheduled on cycle 10, you can achieve a minimum iteration interval of 2 by scheduling MV a0' on cycle 8. Then a0 is live on cycles 7 and 8, and a0' is live on cycles 9 and 10. Because no values are live more than two cycles, the minimum iteration interval for this graph is 2.

Figure 6–20. Dependency Graph of Live-Too-Long Code (Split-Join Path Resolved)



6.9.5 Linear Assembly Resource Allocation

Example 6–54 shows the linear assembly code with the functional units assigned. The choice of units for the ADDs and SUB is flexible and represents one of a number of possibilities. One goal is to ensure that no functional unit is used more than the minimum iteration interval, or two times.

The two 2X paths and one 1X path are required because the values c, d, and e reside on the side opposite from the instruction that is reading them. If these values had created a bottleneck of resources and caused the minimum iteration interval to increase, c, d, and e could have been loaded into the opposite register file outside the loop to eliminate the cross path.

Example 6–54. Linear Assembly for Full Live-Too-Long Code

```
       .global _live_long

_live_long:   .cproc   a, b, c, d, e

       .reg   ai, bi, sum0, sum1, sum
       .reg   a0p, a_0, a_1, a_2, a_3, b_0, b0p, b_1, b_2, b_3, cntr

       MVK    100,cntr                    ; cntr = 100
       ZERO   sum0                        ; sum0 = 0
       ZERO   sum1                        ; sum1 = 0

LOOP:  .trip 100
       LDH    .D1    *a++,ai              ; load ai from memory
       LDH    .D2    *b++,bi              ; load bi from memory
       MPY    .M1    ai,c,a_0             ; a0 = ai * c
       SHR    .S1    a_0,15,a_1           ; a1 = a0 >> 15
       MPY    .M1X   a_1,d,a_2            ; a2 = a1 * d
       MV     .D1    a_0,a0p              ; save a0 across iterations
       ADD    .L1    a_2,a0p,a_3          ; a3 = a2 + a0
       ADD    .L1    sum0,a_3,sum0        ; sum0 += a3
       MPY    .M2X   bi,c,b_0             ; b0 = bi * ci
       SHR    .S2    b_0,15,b_1           ; b1 = b0 >> 15
       MPY    .M2X   b_1,e,b_2            ; b2 = b1 * e
       MV     .D2    b_0,b0p              ; save b0 across iterations
       ADD    .L2    b_2,b0p,b_3          ; b3 = b2 + b0
       ADD    .L2    sum1,b_3,sum1        ; sum1 += b3

[cntr] SUB    .S2    cntr,1,cntr          ; decrement loop counter
[cntr] B      .S1    LOOP                 ; branch to loop

       ADD    sum0,sum1,sum               ; Add sumi and sumi+1 for ret value

       .return sum

       .endproc
```

6.9.6 Final Assembly With Move Instructions

Example 6–55 shows the final assembly code after software pipelining. The performance of this loop is 212 cycles (2 × 100 + 11 + 1).

Example 6–55. Assembly Code for Live-Too-Long With Move Instructions

```
        LDH    .D1    *A4++,A0       ; load ai from memory
||      LDH    .D2    *B4++,B0       ; load bi from memory

        MVK    .S2    100,B2         ; set up loop counter

        LDH    .D1    *A4++,A0       ;* load ai from memory
||      LDH    .D2    *B4++,B0       ;* load bi from memory

        ZERO   .S1    A1             ; zero out accumulator
||      ZERO   .S2    B1             ; zero out accumulator

        LDH    .D1    *A4++,A0       ;** load ai from memory
||      LDH    .D2    *B4++,B0       ;** load bi from memory

   [B2] SUB    .S2    B2,1,B2        ; decrement loop counter

        MPY    .M1    A0,A6,A3       ; a0 = ai * c
||      MPY    .M2X   B0,A6,B10      ; b0 = bi * c
||      LDH    .D1    *A4++,A0       ;*** load ai from memory
||      LDH    .D2    *B4++,B0       ;*** load bi from memory

   [B2] SUB    .S2    B2,1,B2        ; decrement loop counter
|| [B2] B      .S1    LOOP           ; branch to loop

        SHR    .S1    A3,15,A5       ; a1 = a0 >> 15
||      SHR    .S2    B10,15,B5      ; b1 = b0 >> 15
||      MPY    .M1    A0,A6,A3       ;* a0 = ai * c
||      MPY    .M2X   B0,A6,B10      ;* b0 = bi * c
||      LDH    .D1    *A4++,A0       ;**** load ai from memory
||      LDH    .D2    *B4++,B0       ;**** load bi from memory

        MPY    .M1X   A5,B6,A7       ; a2 = a1 * d
||      MV     .D1    A3,A2          ; save a0 across iterations
||      MPY    .M2X   B5,A8,B7       ; b2 = b1 * e
||      MV     .D2    B10,B8         ; save b0 across iterations
|| [B2] SUB    .S2    B2,1,B2        ;* decrement loop counter
|| [B2] B      .S1    LOOP           ;* branch to loop

        SHR    .S1    A3,15,A5       ;* a1 = a0 >> 15
||      SHR    .S2    B10,15,B5      ;* b1 = b0 >> 15
||      MPY    .M1    A0,A6,A3       ;** a0 = ai * c
||      MPY    .M2X   B0,A6,B10      ;** b0 = bi * c
||      LDH    .D1    *A4++,A0       ;***** load ai from memory
||      LDH    .D2    *B4++,B0       ;***** load bi from memory
```

Part III

Example 6–55. Assembly Code for Live-Too-Long With Move Instructions (Continued)

```
LOOP:
       ADD    .L1    A7,A2,A9      ;* a3 = a2 + a0
||     ADD    .L2    B7,B8,B9      ;* b3 = b2 + b0
||     MPY    .M1X   A5,B6,A7      ;* a2 = a1 * d
||     MV     .D1    A3,A2         ;* save a0 across iterations
||     MPY    .M2X   B5,A8,B7      ;* b2 = b1 * e
||     MV     .D2    B10,B8        ;* save b0 across iterations
||[B2] SUB    .S2    B2,1,B2       ;** decrement loop counter
||[B2] B      .S1    LOOP          ;** branch to loop

       ADD    .L1    A1,A9,A1      ; sum0 += a3
||     ADD    .L2    B1,B9,B1      ; sum1 += b3
||     SHR    .S1    A3,15,A5      ;** a1 = a0 >> 15
||     SHR    .S2    B10,15,B5     ;** b1 = b0 >> 15
||     MPY    .M1    A0,A6,A3      ;*** a0 = ai * c
||     MPY    .M2X   B0,A6,B10     ;*** b0 = bi * c
||     LDH    .D1    *A4++,A0      ;****** load ai from memory
||     LDH    .D2    *B4++,B0      ;****** load bi from memory
       ; Branch occurs here

       ADD    .L1X   A1,B1,A4      ; sum = sum0 + sum1
```

6.10 Redundant Load Elimination

Filter algorithms typically read the same value from memory multiple times and are, therefore, prime candidates for optimization by eliminating redundant load instructions. Rather than perform a load operation each time a particular value is read, you can keep the value in a register and read the register multiple times.

6.10.1 FIR Filter C Code

Example 6–56 shows C code for a simple FIR filter. There are two memory reads (x[i+j] and h[i]) for each multiply. Because the 'C6x can perform only two LDHs per cycle, it seems, at first glance, that only one multiply-accumulate per cycle is possible.

Example 6–56. FIR Filter C Code

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum;

         for (j = 0; j < 100; j++) {
                  sum = 0;
                  for (i = 0; i < 32; i++)
                           sum += x[i+j] * h[i];
                  y[j] = sum >> 15;
         }
}
```

One way to optimize this situation is to perform LDWs instead of LDHs to read two data values at a time. Although using LDW works for the h array, the x array presents a different problem because the 'C6x does not allow you to load values across a word boundary.

For example, on the first outer loop (j = 0), you can read the x-array elements (0 and 1, 2 and 3, etc.) as long as elements 0 and 1 are aligned on a 4-byte word boundary. However, the second outer loop (j = 1) requires reading x-array elements 1 through 32. The LDW operation must load elements that are not word-aligned (1 and 2, 3 and 4, etc.).

6.10.1.1 Redundant Loads

In order to achieve two multiply-accumulates per cycle, you must reduce the number of LDHs. Because successive outer loops read all the same h-array values and almost all of the same x-array values, you can eliminate the redundant loads by unrolling the inner and outer loops.

For example, x[1] is needed for the first outer loop (x[j+1] with j = 0) and for the second outer loop (x[j] with j = 1). You can use a single LDH instruction to load this value.

6.10.1.2 New FIR Filter C Code

Example 6–57 shows that after eliminating redundant loads, there are four memory-read operations for every four multiply-accumulate operations. Now the memory accesses no longer limit the performance.

Example 6–57. FIR Filter C Code With Redundant Load Elimination

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum0, sum1;
         short x0,x1,h0,h1;

         for (j = 0; j < 100; j+=2) {
                  sum0 = 0;
                  sum1 = 0;
                  x0 = x[j];
                  for (i = 0; i < 32; i+=2){
                           x1 = x[j+i+1];
                           h0 = h[i];
                           sum0 += x0 * h0;
                           sum1 += x1 * h0;
                           x0 = x[j+i+2];
                           h1 = h[i+1];
                           sum0 += x1 * h1;
                           sum1 += x0 * h1;
                           }
                  y[j] = sum0 >> 15;
                  y[j+1] = sum1 >> 15;
         }
}
```

6.10.2 Translating C Code to Linear Assembly

Example 6–58 shows the linear assembly that perform the inner loop of the FIR filter C code.

Element x0 is read by the MPY p00 before it is loaded by the LDH x0 instruction; x[j] (the first x0) is loaded outside the loop, but successive even elements are loaded inside the loop.

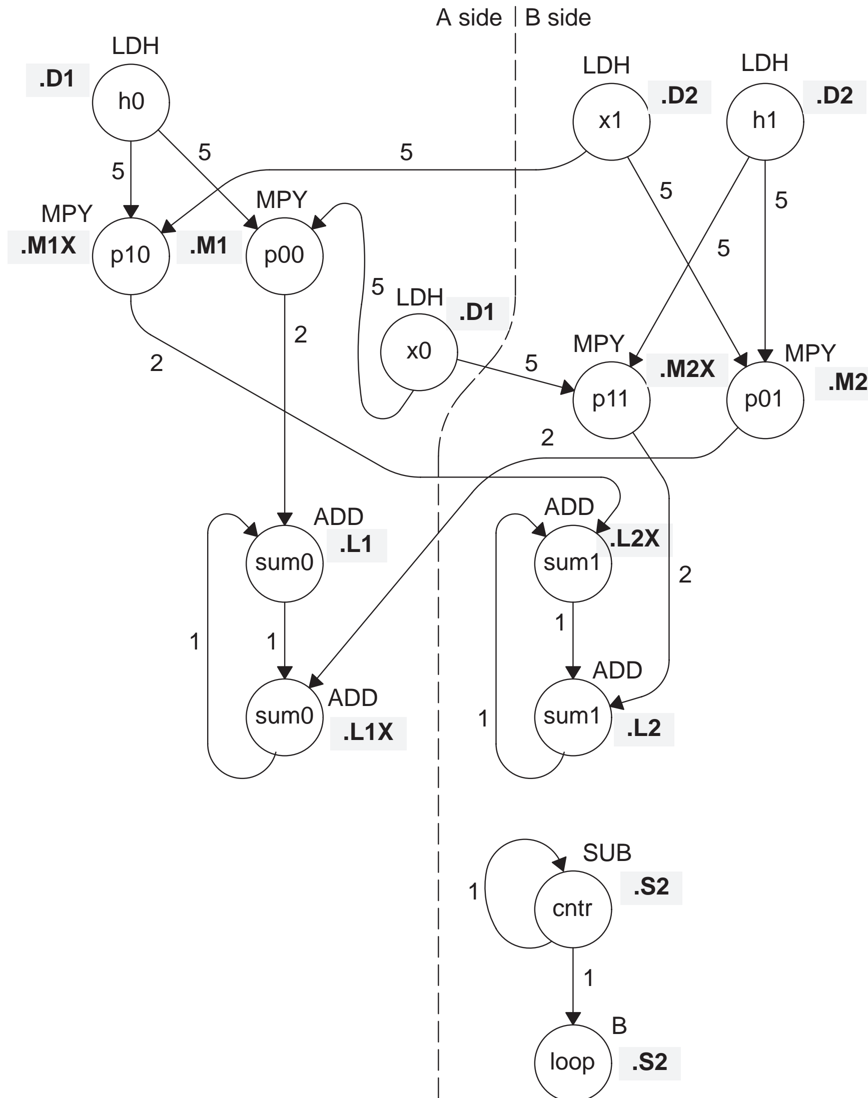
Example 6–58. Linear Assembly for FIR Inner Loop

```
        LDH     .D2     *x_1++[2],x1        ; x1 = x[j+i+1]
        LDH     .D1     *h++[2],h0          ; h0 = h[i]
        MPY     .M1     x0,h0,p00           ; x0 * h0
        MPY     .M1X    x1,h0,p10           ; x1 * h0
        ADD     .L1     p00,sum0,sum0       ; sum0 += x0 * h0
        ADD     .L2X    p10,sum1,sum1       ; sum1 += x1 * h0

        LDH     .D1     *x++[2],x0          ; x0 = x[j+i+2]
        LDH     .D2     *h_1++[2],h1        ; h1 = h[i+1]
        MPY     .M2     x1,h1,p01           ; x1 * h1
        MPY     .M2X    x0,h1,p11           ; x0 * h1
        ADD     .L1X    p01,sum0,sum0       ; sum0 += x1 * h1
        ADD     .L2     p11,sum1,sum1       ; sum1 += x0 * h1

[ctr]   SUB     .S2     ctr,1,ctr           ; decrement loop counter
[ctr]   B       .S2     LOOP                ; branch to loop
```

6.10.3 Drawing a Dependency Graph

Figure 6–21 shows the dependency graph of the FIR filter with redundant load elimination.

Figure 6–21. Dependency Graph of FIR Filter (With Redundant Load Elimination)



6.10.4 Determining the Minimum Iteration Interval

Table 6–23 shows that the minimum iteration interval is 2. An iteration interval of 2 means that two multiply-accumulates are executing per cycle.

Table 6–23. Resource Table for FIR Filter Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	2 MPYs	2
.S1		0
.D1	2 LDHs	2
.L1, .S1, or .D1	2 ADDs	2
Total non-.M units		4
1X paths		2

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	2 MPYs	2
.S2	B	1
.D2	2 LDHs	2
.L2, .S2, .D2	2 ADDs and SUB	3
Total non-.M units		6
2X paths		2

6.10.5 Linear Assembly Resource Allocation

Example 6–59 shows the linear assembly with functional units and registers assigned.

Example 6–59. Linear Assembly for Full FIR Code

```
         .global _fir

_fir:    .cproc   x, h, y

         .reg     x_1, h_1, sum0, sum1, ctr, octr
         .reg     p00, p01, p10, p11, x0, x1, h0, h1, rstx, rsth

         ADD      h,2,h_1                 ; set up pointer to h[1]
         MVK      50,octr                 ; outer loop ctr = 100/2
         MVK      64,rstx                 ; used to rst x pointer each outer loop
         MVK      64,rsth                 ; used to rst h pointer each outer loop
OUTLOOP:
         ADD      x,2,x_1                 ; set up pointer to x[j+1]
         SUB      h_1,2,h                 ; set up pointer to h[0]
         MVK      16,ctr                  ; inner loop ctr = 32/2
         ZERO     sum0                    ; sum0 = 0
         ZERO     sum1                    ; sum1 = 0
 [octr]  SUB      octr,1,octr             ; decrement outer loop counter

         LDH      .D1      *x++[2],x0     ; x0 = x[j]
```

Example 6–59. Linear Assembly for Full FIR Code (Continued)

```
LOOP:    .trip 16

         LDH      .D2      *x_1++[2],x1       ; x1 = x[j+i+1]
         LDH      .D1      *h++[2],h0         ; h0 = h[i]
         MPY      .M1      x0,h0,p00          ; x0 * h0
         MPY      .M1X     x1,h0,p10          ; x1 * h0
         ADD      .L1      p00,sum0,sum0      ; sum0 += x0 * h0
         ADD      .L2X     p10,sum1,sum1      ; sum1 += x1 * h0

         LDH      .D1      *x++[2],x0         ; x0 = x[j+i+2]
         LDH      .D2      *h_1++[2],h1       ; h1 = h[i+1]
         MPY      .M2      x1,h1,p01          ; x1 * h1
         MPY      .M2X     x0,h1,p11          ; x0 * h1
         ADD      .L1X     p01,sum0,sum0      ; sum0 += x1 * h1
         ADD      .L2      p11,sum1,sum1      ; sum1 += x0 * h1

 [ctr]   SUB      .S2      ctr,1,ctr          ; decrement loop counter
 [ctr]   B        .S2      LOOP               ; branch to loop
         SHR      sum0,15,sum0                ; sum0 >> 15
         SHR      sum1,15,sum1                ; sum1 >> 15
         STH      sum0,*y++                   ; y[j] = sum0 >> 15
         STH      sum1,*y++                   ; y[j+1] = sum1 >> 15
         SUB      x,rstx,x                    ; reset x pointer to x[j]
         SUB      h_1,rsth,h_1                ; reset h pointer to h[0]
 [octr]  B        OUTLOOP                     ; branch to outer loop

         .endproc
```


6.10.6 Final Assembly

Example 6–60 shows the final assembly for the FIR filter without redundant load instructions. At the end of the inner loop is a branch to OUTLOOP that executes the next outer loop. The outer loop counter is 50 because iterations j and j + 1 execute each time the inner loop is run. The inner loop counter is 16 because iterations i and i + 1 execute each inner loop iteration.

The cycle count for this nested loop is 2352 cycles: 50 (16 $\times$ 2 + 9 + 6) + 2. Fifteen cycles are overhead for each outer loop:

- ❑ Nine cycles execute the inner loop prolog.
- ❑ Six cycles execute the branch to the outer loop.

See section 6.12, *Software Pipelining the Outer Loop*, on page 6-128 for information on how to reduce this overhead.

Example 6–60. Final Assembly Code for FIR Filter With Redundant Load Elimination

```
        MVK     .S1     50,A2           ; set up outer loop counter

        MVK     .S1     80,A3           ; used to rst x ptr outer loop
||      MVK     .S2     82,B6           ; used to rst h ptr outer loop

OUTLOOP:
        LDH     .D1     *A4++[2],A0     ; x0 = x[j]                        1
||      ADD     .L2X    A4,2,B5         ; set up pointer to x[j+1]
||      ADD     .D2     B4,2,B4         ; set up pointer to h[1]
||      ADD     .L1X    B4,0,A5         ; set up pointer to h[0]
||      MVK     .S2     16,B2           ; set up inner loop counter
||[A2]  SUB     .S1     A2,1,A2         ; decrement outer loop counter

        LDH     .D1     *A5++[2],A1     ; h0 = h[i]                        2
||      LDH     .D2     *B5++[2],B1     ; x1 = x[j+i+1]
||      ZERO    .L1     A9              ; zero out sum0
||      ZERO    .L2     B9              ; zero out sum1

        LDH     .D2     *B4++[2],B0     ; h1 = h[i+1]                      3
||      LDH     .D1     *A4++[2],A0     ; x0 = x[j+i+2]

        LDH     .D1     *A5++[2],A1     ;* h0 = h[i]                       4
||      LDH     .D2     *B5++[2],B1     ;* x1 = x[j+i+1]

  [B2]  SUB     .S2     B2,1,B2         ; decrement inner loop counter     5
||      LDH     .D2     *B4++[2],B0     ;* h1 = h[i+1]
||      LDH     .D1     *A4++[2],A0     ;* x0 = x[j+i+2]

  [B2]  B       .S2     LOOP            ; branch to inner loop             6
||      LDH     .D1     *A5++[2],A1     ;** h0 = h[i]
||      LDH     .D2     *B5++[2],B1     ;** x1 = x[j+i+1]

        MPY     .M1     A0,A1,A7        ; x0 * h0                          7
||[B2]  SUB     .S2     B2,1,B2         ;* decrement inner loop counter
||      LDH     .D2     *B4++[2],B0     ;** h1 = h[i+1]
||      LDH     .D1     *A4++[2],A0     ;** x0 = x[j+i+2]

        MPY     .M2     B1,B0,B7        ; x1 * h1                          8
||      MPY     .M1X    B1,A1,A8        ; x1 * h0
||[B2]  B       .S2     LOOP            ;* branch to inner loop
||      LDH     .D1     *A5++[2],A1     ;*** h0 = h[i]
||      LDH     .D2     *B5++[2],B1     ;*** x1 = x[j+i+1]

        MPY     .M2X    A0,B0,B8        ; x0 * h1                          9
||      MPY     .M1     A0,A1,A7        ;* x0 * h0
||[B2]  SUB     .S2     B2,1,B2         ;** decrement inner loop counter
||      LDH     .D2     *B4++[2],B0     ;*** h1 = h[i+1]
||      LDH     .D1     *A4++[2],A0     ;*** x0 = x[j+i+2]
```

Part III

Example 6–60 Final Assembly Code for FIR Filter With Redundant Load Elimination (Continued)

```
LOOP:
       ADD    .L2X   A8,B9,B9      ; sum1 += x1 * h0
||     ADD    .L1    A7,A9,A9      ; sum0 += x0 * h0
||     MPY    .M2    B1,B0,B7      ;* x1 * h1
||     MPY    .M1X   B1,A1,A8      ;* x1 * h0
||[B2] B      .S2    LOOP          ;** branch to inner loop
||     LDH    .D1    *A5++[2],A1   ;**** h0 = h[i]
||     LDH    .D2    *B5++[2],B1   ;**** x1 = x[j+i+1]

       ADD    .L1X   B7,A9,A9      ; sum0 += x1 * h1
||     ADD    .L2    B8,B9,B9      ; sum1 += x0 * h1
||     MPY    .M2X   A0,B0,B8      ;* x0 * h1
||     MPY    .M1    A0,A1,A7      ;** x0 * h0
||[B2] SUB    .S2    B2,1,B2              ;*** decrement inner loop cntr
||     LDH    .D2    *B4++[2],B0   ;**** h1 = h[i+1]
||     LDH    .D1    *A4++[2],A0   ;**** x0 = x[j+i+2]
       ; inner loop branch occurs here

  [A2] B      .S1    OUTLOOP              ; branch to outer loop          (1)
||     SUB    .L1    A4,A3,A4      ; reset x pointer to x[j]
||     SUB    .L2    B4,B6,B4      ; reset h pointer to h[0]

       SHR    .S1    A9,15,A9      ; sum0 >> 15                           (2)
||     SHR    .S2    B9,15,B9      ; sum1 >> 15

       STH    .D1    A9,*A6++      ; y[j] = sum0 >> 15                    (3)

       STH    .D1    B9,*A6++      ; y[j+1] = sum1 >> 15                  (4)

       NOP    2                    ; branch delay slots                   (5)
       ; outer loop branch occurs here                                    (6)
```

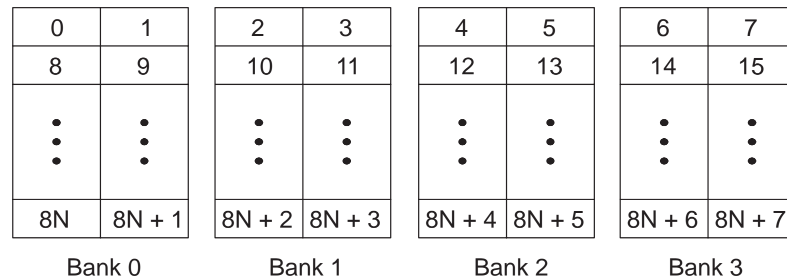
6.11 Memory Banks

The internal memory of the ’C6x family varies from device to device. See the *TMS320C62x/C67x Peripherals Reference Guide* to determine the memory blocks in your particular device. This section discusses how to write code to avoid memory bank conflicts.

Most ’C6x devices use an interleaved memory bank scheme, as shown in Figure 6–22. Each number in the boxes represents a byte address. A load byte (LDB) instruction from address 0 loads byte 0 in bank 0. A load halfword (LDH) from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. An LDW from address 0 loads bytes 0 through 3 in banks 0 and 1.

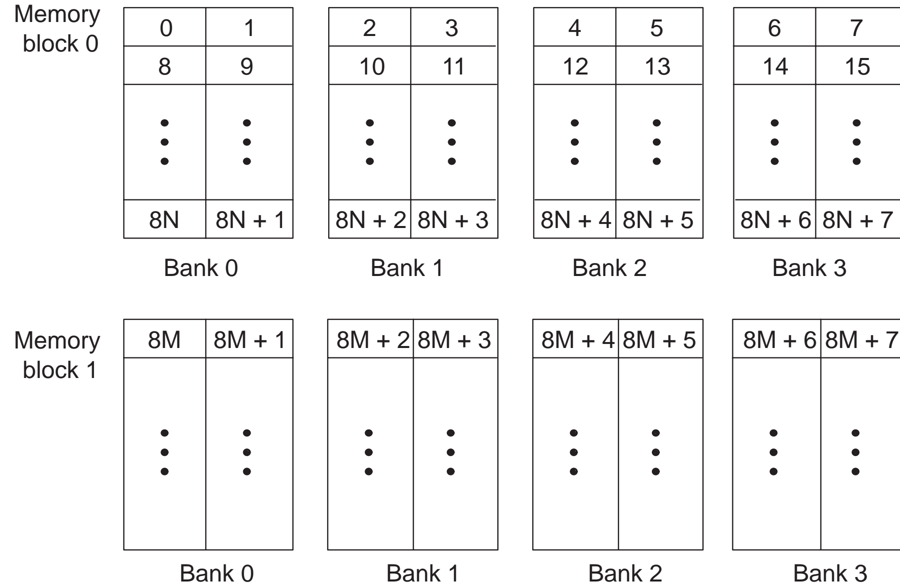
Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle, while the second value is read from memory. Two memory operations per cycle are allowed without any stall, as long as they do not access the same bank.

Figure 6–22. 4-Bank Interleaved Memory



For devices that have more than one memory block (see Figure 6–23), an access to bank 0 in one block does not interfere with an access to bank 0 in another memory block, and no pipeline stall occurs.

Figure 6–23. 4-Bank Interleaved Memory With Two Memory Blocks



If each array in a loop resides in a separate memory block, the 2-cycle loop in Example 6–57 on page 6-108 is sufficient. This section describes a solution when two arrays must reside in the same memory block.

6.11.1 FIR Filter Inner Loop

Example 6–61 shows the inner loop from the final assembly in Example 6–60. The LDHs from the h array are in parallel with LDHs from the x array. If x[1] is on an even halfword (bank 0) and h[0] is on an odd halfword (bank 1), Example 6–61 has no memory conflicts. However, if both x[1] and h[0] are on an even halfword in memory (bank 0) and they are in the same memory block, every cycle incurs a memory pipeline stall and the loop runs at half the speed.

Example 6–61. Final Assembly Code for Inner Loop of FIR Filter

```
LOOP:
        ADD     .L2X    A8,B9,B9        ; sum1 += x1 * h0
||      ADD     .L1     A7,A9,A9        ; sum0 += x0 * h0
||      MPY     .M2     B1,B0,B7        ;* x1 * h1
||      MPY     .M1X    B1,A1,A8        ;* x1 * h0
||[B2]  B       .S2     LOOP            ;** branch to inner loop
||      LDH     .D1     *A5++[2],A1     ;**** h0 = h[i]
||      LDH     .D2     *B5++[2],B1     ;**** x1 = x[j+i+1]

        ADD     .L1X    B7,A9,A9        ; sum0 += x1 * h1
||      ADD     .L2     B8,B9,B9        ; sum1 += x0 * h1
||      MPY     .M2X    A0,B0,B8        ;* x0 * h1
||      MPY     .M1     A0,A1,A7        ;** x0 * h0
||[B2]  SUB     .S2     B2,1,B2         ;*** decrement inner loop cntr
||      LDH     .D2     *B4++[2],B0     ;**** h1 = h[i+1]
||      LDH     .D1     *A4++[2],A0     ;**** x0 = x[j+i+2]
```

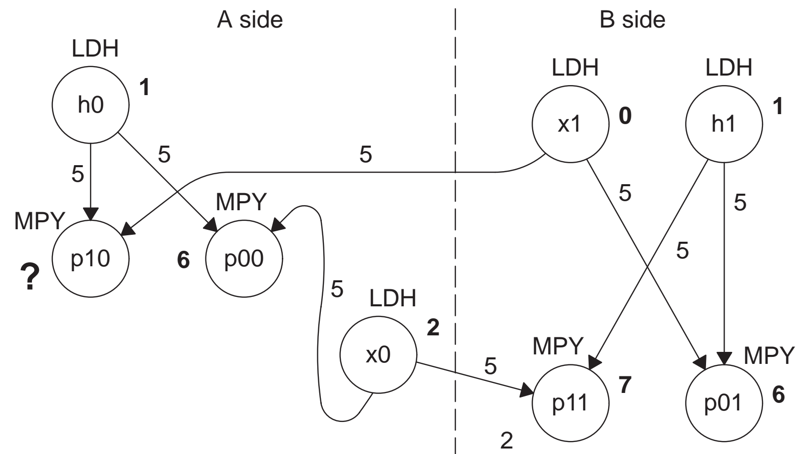
It is not always possible to fully control how arrays are aligned, especially if one of the arrays is passed into a function as a pointer and that pointer has different alignments each time the function is called. One solution to this problem is to write an FIR filter that avoids memory hits, regardless of the x and h array alignments.

If accesses to the even and odd elements of an array (h or x) are scheduled on the same cycle, the accesses are always on adjacent memory banks. Thus, to write an FIR filter that never has memory hits, even and odd elements of the same array must be scheduled on the same loop cycle.

In the case of the FIR filter, scheduling the even and odd elements of the same array on the same loop cycle cannot be done in a 2-cycle loop, as shown in Figure 6–24. In this example, a valid 2-cycle software-pipelined loop without memory constraints is ruled by the following constraints:

- LDH h0 and LDH h1 are on the same loop cycle.
- LDH x0 and LDH x1 are on the same loop cycle.
- MPY p00 must be scheduled three or four cycles after LDH x0, because it must read x0 from the previous iteration of LDH x0.
- All MPYs must be five or six cycles after their LDH parents.
- No MPYs on the same side (A or B) can be on the same loop cycle.

Figure 6–24. Dependency Graph of FIR Filter (With Even and Odd Elements of Each Array on Same Loop Cycle)



Note: Numbers in bold represent the cycle the instruction is scheduled on.

The scenario in Figure 6–24 *almost* works. All nodes satisfy the above constraints except MPY p10. Because one parent is on cycle 1 (LDH h0) and another on cycle 0 (LDH x1), the only cycle for MPY p10 is cycle 6. However, another MPY on the A side is also scheduled on cycle 6 (MPY p00). Other combinations of cycles for this graph produce similar results.

6.11.2 Unrolled FIR Filter C Code

The main limitation in solving the problem in Figure 6–24 is in scheduling a 2-cycle loop, which means that no value can be live more than two cycles. Increasing the iteration interval to 3 decreases performance. A better solution is to unroll the inner loop one more time and produce a 4-cycle loop.

Example 6–62 shows the FIR filter C code after unrolling the inner loop one more time. This solution adds to the flexibility of scheduling and allows you to write FIR filter code that never has memory hits, regardless of array alignment and memory block.

Example 6–62. FIR Filter C Code (Unrolled)

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum0, sum1;
         short x0,x1,x2,x3,h0,h1,h2,h3;

         for (j = 0; j < 100; j+=2) {
                  sum0 = 0;
                  sum1 = 0;
                  x0 = x[j];
                  for (i = 0; i < 32; i+=4){
                           x1 = x[j+i+1];
                           h0 = h[i];
                           sum0 += x0 * h0;
                           sum1 += x1 * h0;
                           x2 = x[j+i+2];
                           h1 = h[i+1];
                           sum0 += x1 * h1;
                           sum1 += x2 * h1;
                           x3 = x[j+i+3];
                           h2 = h[i+2];
                           sum0 += x2 * h2;
                           sum1 += x3 * h2;
                           x0 = x[j+i+4];
                           h3 = h[i+3];
                           sum0 += x3 * h3;
                           sum1 += x0 * h3;
                           }
                  y[j] = sum0 >> 15;
                  y[j+1] = sum1 >> 15;
         }
}
```

Part III

6.11.3 Translating C Code to Linear Assembly

Example 6–63 shows the linear assembly for the unrolled inner loop of the FIR filter C code.

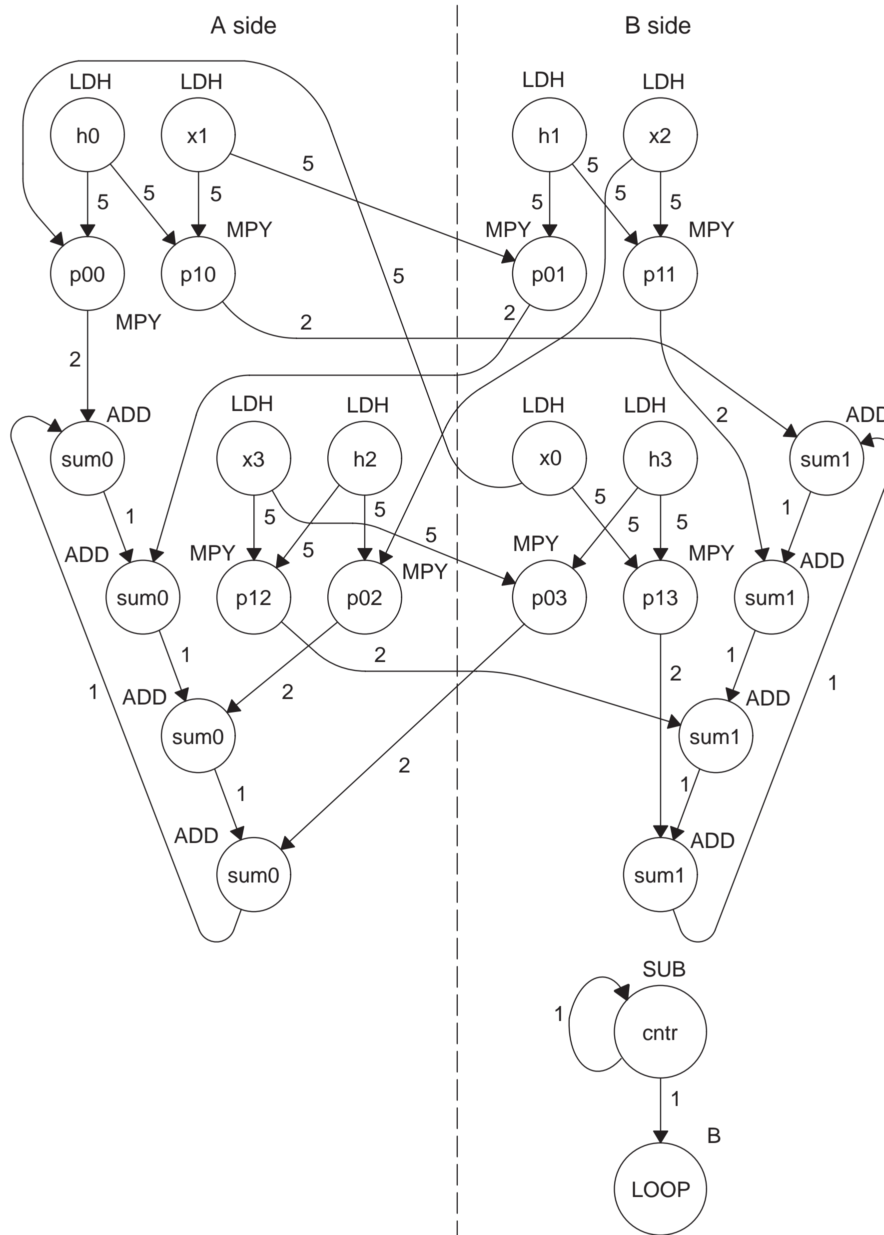
Example 6–63. Linear Assembly for Unrolled FIR Inner Loop

```
        LDH      *x++,x1          ; x1 = x[j+i+1]
        LDH      *h++,h0          ; h0 = h[i]
        MPY      x0,h0,p00        ; x0 * h0
        MPY      x1,h0,p10        ; x1 * h0
        ADD      p00,sum0,sum0    ; sum0 += x0 * h0
        ADD      p10,sum1,sum1    ; sum1 += x1 * h0

        LDH      *x++,x2          ; x2 = x[j+i+2]
        LDH      *h++,h1          ; h1 = h[i+1]
        MPY      x1,h1,p01        ; x1 * h1
        MPY      x2,h1,p11        ; x2 * h1
        ADD      p01,sum0,sum0    ; sum0 += x1 * h1
        ADD      p11,sum1,sum1    ; sum1 += x2 * h1

        LDH      *x++,x3          ; x3 = x[j+i+3]
        LDH      *h++,h2          ; h2 = h[i+2]
        MPY      x2,h2,p02        ; x2 * h2
        MPY      x3,h2,p12        ; x3 * h2
        ADD      p02,sum0,sum0    ; sum0 += x2 * h2
        ADD      p12,sum1,sum1    ; sum1 += x3 * h2

        LDH      *x++,x0          ; x0 = x[j+i+4]
        LDH      *h++,h3          ; h3 = h[i+3]
        MPY      x3,h3,p03        ; x3 * h3
        MPY      x0,h3,p13        ; x0 * h3
        ADD      p03,sum0,sum0    ; sum0 += x3 * h3
        ADD      p13,sum1,sum1    ; sum1 += x0 * h3

 [cntr] SUB      cntr,1,cntr      ; decrement loop counter
 [cntr] B        LOOP             ; branch to loop
```

6.11.4 Drawing a Dependency Graph

Figure 6–25 shows the dependency graph of the FIR filter with no memory hits.

Figure 6–25. Dependency Graph of FIR Filter (With No Memory Hits)



Part III

6.11.5 Linear Assembly for Unrolled FIR Inner Loop With .mptr Directive

Example 6–64 shows the unrolled FIR inner loop with the .mptr directive. The .mptr directive allows the assembly optimizer to automatically determine if two memory operations have a bank conflict by associating memory access information with a specific pointer register.

If the assembly optimizer determines that two memory operations have a bank conflict, then it will not schedule them in parallel. The .mptr directive tells the assembly optimizer that when the specified register is used as a memory pointer in a load or store instruction, it is initialized to point at a base location + <offset>, and is incremented a number of times each time through the loop.

Without the .mptr directives, the loads of x1 and h0 are scheduled in parallel, and the loads of x2 and h1 are scheduled in parallel. This results in a 50% chance of a memory conflict on every cycle.

Example 6–64. Linear Assembly for Full Unrolled FIR Filter

```
         .global _fir

_fir:    .cproc   x, h, y

         .reg     x_1, h_1, sum0, sum1, ctr, octr
         .reg     p00, p01, p02, p03, p10, p11, p12, p13
         .reg     x0, x1, x2, x3, h0, h1, h2, h3, rstx, rsth

         ADD      h,2,h_1           ; set up pointer to h[1]
         MVK      50,octr           ; outer loop ctr = 100/2
         MVK      64,rstx           ; used to rst x pointer each outer loop
         MVK      64,rsth           ; used to rst h pointer each outer loop
OUTLOOP:
         ADD      x,2,x_1           ; set up pointer to x[j+1]
         SUB      h_1,2,h           ; set up pointer to h[0]
         MVK      8,ctr             ; inner loop ctr = 32/2
         ZERO     sum0              ; sum0 = 0
         ZERO     sum1              ; sum1 = 0
  [octr] SUB      octr,1,octr       ; decrement outer loop counter

         .mptr    x,   x+0
         .mptr    x_1, x+2
         .mptr    h,   h+0
         .mptr    h_1, h+2

         LDH      .D2      *x++[2],x0        ; x0 = x[j]
```

Example 6–64. Linear Assembly for Full Unrolled FIR Filter (Continued)

```
LOOP:   .trip 8

        LDH     .D1     *x_1++[2],x1      ; x1 = x[j+i+1]
        LDH     .D1     *h++[2],h0        ; h0 = h[i]
        MPY     .M1X    x0,h0,p00         ; x0 * h0
        MPY     .M1     x1,h0,p10         ; x1 * h0
        ADD     .L1     p00,sum0,sum0     ; sum0 += x0 * h0
        ADD     .L2X    p10,sum1,sum1     ; sum1 += x1 * h0

        LDH     .D2     *x++[2],x2        ; x2 = x[j+i+2]
        LDH     .D2     *h_1++[2],h1      ; h1 = h[i+1]
        MPY     .M2X    x1,h1,p01         ; x1 * h1
        MPY     .M2     x2,h1,p11         ; x2 * h1
        ADD     .L1X    p01,sum0,sum0     ; sum0 += x1 * h1
        ADD     .L2     p11,sum1,sum1     ; sum1 += x2 * h1

        LDH     .D1     *x_1++[2],x3      ; x3 = x[j+i+3]
        LDH     .D1     *h++[2],h2        ; h2 = h[i+2]
        MPY     .M1X    x2,h2,p02         ; x2 * h2
        MPY     .M1     x3,h2,p12         ; x3 * h2
        ADD     .L1     p02,sum0,sum0     ; sum0 += x2 * h2
        ADD     .L2X    p12,sum1,sum1     ; sum1 += x3 * h2

        LDH     .D2     *x++[2],x0        ; x0 = x[j+i+4]
        LDH     .D2     *h_1++[2],h3      ; h3 = h[i+3]
        MPY     .M2X    x3,h3,p03         ; x3 * h3
        MPY     .M2     x0,h3,p13         ; x0 * h3
        ADD     .L1X    p03,sum0,sum0     ; sum0 += x3 * h3
        ADD     .L2     p13,sum1,sum1     ; sum1 += x0 * h3


[ctr]   SUB     .S2     ctr,1,ctr         ; decrement loop counter
[ctr]   B       .S2     LOOP              ; branch to loop

        SHR     sum0,15,sum0              ; sum0 >> 15
        SHR     sum1,15,sum1              ; sum1 >> 15
        STH     sum0,*y++                 ; y[j] = sum0 >> 15
        STH     sum1,*y++                 ; y[j+1] = sum1 >> 15
        SUB     x,rstx,x                  ; reset x pointer to x[j]
        SUB     h_1,rsth,h_1              ; reset h pointer to h[0]
 [octr] B       OUTLOOP                   ; branch to outer loop

        .endproc
```

Part III

6.11.6 Linear Assembly Resource Allocation

As the number of instructions in a loop increases, assigning a specific register to every value in the loop becomes increasingly difficult. If 33 instructions in a loop each write a value, they cannot each write to a unique register because the ’C6x has only 32 registers. As a result, values that are not live on the same cycles in the loop must share registers.

For example, in a 4-cycle loop:

- If a value is written at the end of cycle 0 and read on cycle 2 of the loop, it is live for two cycles (cycles 1 and 2 of the loop).
- If another value is written at the end of cycle 2 and read on cycle 0 (the next iteration) of the loop, it is also live for two cycles (cycles 3 and 0 of the loop).

Because both of these values are not live on the same cycles, they can occupy the same register. Only after scheduling these instructions and their children do you know that they can occupy the same register.

Register allocation is not complicated but can be tedious when done by hand. Each value has to be analyzed for its lifetime and then appropriately combined with other values not live on the same cycles in the loop. The assembly optimizer handles this automatically after it software pipelines the loop. See the *TMS320C6x Optimizing C Compiler User’s Guide* for more information.

6.11.7 Determining the Minimum Iteration Interval

Based on Table 6–24, the minimum iteration interval for the FIR filter with no memory hits should be 4. An iteration interval of 4 means that two multiply/accumulates still execute per cycle.

Table 6–24. Resource Table for FIR Filter Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	4 MPYs	4
.S1		0
.D1	4 LDHs	4
.L1, .S1, or .D1	4 ADDs	4
Total non-.M units		8
1X paths		4

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	4 MPYs	4
.S2	B	1
.D2	4 LDHs	4
.L2, .S2, or .D2	4 ADDs and SUB	5
Total non-.M units		10
2X paths		4

6.11.8 Final Assembly

Example 6–65 shows the final assembly to the FIR filter with redundant load elimination and no memory hits. At the end of the inner loop, there is a branch to OUTLOOP to execute the next outer loop. The outer loop counter is set to 50 because iterations j and j+1 are executing each time the inner loop is run. The inner loop counter is set to 8 because iterations i, i + 1, i + 2, and i + 3 are executing each inner loop iteration.

Part III

6.11.9 Comparing Performance

The cycle count for this nested loop is 2402 cycles. There is a rather large outer-loop overhead for executing the branch to the outer loop (6 cycles) and the inner loop prolog (10 cycles). Section 6.12 addresses how to reduce this overhead by software pipelining the outer loop.

Table 6–25. Comparison of FIR Filter Code

Code Example		Cycles	Cycle Count
Example 6–60	FIR with redundant load elimination	50 (16 × 2 + 9 + 6) + 2	2352
Example 6–65	FIR with redundant load elimination and no memory hits	50 (8 × 4 + 10 + 6) + 2	2402

Example 6–65. Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits

```
        MVK     .S1     50,A2           ; set up outer loop counter

        MVK     .S1     62,A3           ; used to rst x pointer outloop
||      MVK     .S2     64,B10          ; used to rst h pointer outloop

OUTLOOP:
        LDH     .D1     *A4++,B5 ; x0 = x[j]
||      ADD     .L2X    A4,4,B1         ; set up pointer to x[j+2]
||      ADD     .L1X    B4,2,A8         ; set up pointer to h[1]
||      MVK     .S2     8,B2            ; set up inner loop counter
||[A2]  SUB     .S1     A2,1,A2         ; decrement outer loop counter

        LDH     .D2     *B1++[2],B0     ; x2 = x[j+i+2]
||      LDH     .D1     *A4++[2],A0     ; x1 = x[j+i+1]
||      ZERO    .L1     A9              ; zero out sum0
||      ZERO    .L2     B9              ; zero out sum1

        LDH     .D1     *A8++[2],B6     ; h1 = h[i+1]
||      LDH     .D2     *B4++[2],A1     ; h0 = h[i]

        LDH     .D1     *A4++[2],A5     ; x3 = x[j+i+3]
||      LDH     .D2     *B1++[2],B5     ; x0 = x[j+i+4]

        LDH     .D2     *B4++[2],A7     ; h2 = h[i+2]
||      LDH     .D1     *A8++[2],B8     ; h3 = h[i+3]
||[B2]  SUB     .S2     B2,1,B2         ; decrement loop counter

        LDH     .D2     *B1++[2],B0     ;* x2 = x[j+i+2]
||      LDH     .D1     *A4++[2],A0     ;* x1 = x[j+i+1]

        LDH     .D1     *A8++[2],B6     ;* h1 = h[i+1]
||      LDH     .D2     *B4++[2],A1     ;* h0 = h[i]

        MPY     .M1X    B5,A1,A0        ; x0 * h0
||      MPY     .M2X    A0,B6,B6        ; x1 * h1
||      LDH     .D1     *A4++[2],A5     ;* x3 = x[j+i+3]
||      LDH     .D2     *B1++[2],B5     ;* x0 = x[j+i+4]

  [B2]  B       .S1     LOOP            ; branch to loop
||      MPY     .M2     B0,B6,B7        ; x2 * h1
||      MPY     .M1     A0,A1,A1        ; x1 * h0
||      LDH     .D2     *B4++[2],A7     ;* h2 = h[i+2]
||      LDH     .D1     *A8++[2],B8     ;* h3 = h[i+3]
||[B2]  SUB     .S2     B2,1,B2         ;* decrement loop counter

        ADD     .L1     A0,A9,A9        ; sum0 += x0 * h0
||      MPY     .M2X    A5,B8,B8        ; x3 * h3
||      MPY     .M1X    B0,A7,A5        ; x2 * h2
||      LDH     .D2     *B1++[2],B0     ;** x2 = x[j+i+2]
||      LDH     .D1     *A4++[2],A0     ;** x1 = x[j+i+1]
```

Part III

Example 6–65. Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits (Continued)

```
LOOP:
         ADD      .L2X     A1,B9,B9          ; sum1 += x1 * h0
||       ADD      .L1X     B6,A9,A9          ; sum0 += x1 * h1
||       MPY      .M2      B5,B8,B7          ; x0 * h3
||       MPY      .M1      A5,A7,A7          ; x3 * h2
||[B2]   LDH      .D1      *A8++[2],B6       ;** h1 = h[i+1]
||[B2]   LDH      .D2      *B4++[2],A1       ;** h0 = h[i]

         ADD      .L2      B7,B9,B9          ; sum1 += x2 * h1
||       ADD      .L1      A5,A9,A9          ; sum0 += x2 * h2
||       MPY      .M1X     B5,A1,A0          ;* x0 * h0
||       MPY      .M2X     A0,B6,B6          ;* x1 * h1
||[B2]   LDH      .D1      *A4++[2],A5       ;** x3 = x[j+i+3]
||[B2]   LDH      .D2      *B1++[2],B5       ;** x0 = x[j+i+4]

         ADD      .L2X     A7,B9,B9          ; sum1 += x3 * h2
||       ADD      .L1X     B8,A9,A9          ; sum0 += x3 * h3
||[B2]   B        .S1      LOOP              ;* branch to loop
||       MPY      .M2      B0,B6,B7          ;* x2 * h1
||       MPY      .M1      A0,A1,A1          ;* x1 * h0
||[B2]   LDH      .D2      *B4++[2],A7       ;** h2 = h[i+2]
||[B2]   LDH      .D1      *A8++[2],B8       ;** h3 = h[i+3]
||[B2]   SUB      .S2      B2,1,B2           ;** decrement loop counter

         ADD      .L2      B7,B9,B9          ; sum1 += x0 * h3
||       ADD      .L1      A0,A9,A9          ;* sum0 += x0 * h0
||       MPY      .M2X     A5,B8,B8          ;* x3 * h3
||       MPY      .M1X     B0,A7,A5          ;* x2 * h2
||[B2]   LDH      .D2      *B1++[2],B0       ;*** x2 = x[j+i+2]
||[B2]   LDH      .D1      *A4++[2],A0       ;*** x1 = x[j+i+1]
         ; inner loop branch occurs here

  [A2]   B        .S2      OUTLOOP           ; branch to outer loop
||       SUB      .L1      A4,A3,A4          ; reset x pointer to x[j]
||       SUB      .L2      B4,B10,B4         ; reset h pointer to h[0]
||       SUB      .S1      A9,A0,A9          ; sum0 -= x0*h0 (eliminate add)

         SHR      .S1      A9,15,A9          ; sum0 >> 15
||       SHR      .S2      B9,15,B9          ; sum1 >> 15

         STH      .D1      A9,*A6++          ; y[j] = sum0 >> 15

         STH      .D1      B9,*A6++          ; y[j+1] = sum1 >> 15

         NOP      2                          ; branch delay slots
         ; outer loop branch occurs here
```

6.12 Software Pipelining the Outer Loop

In previous examples, software pipelining has always affected the inner loop. However, software pipelining works equally well with the outer loop in a nested loop.

6.12.1 Unrolled FIR Filter C Code

Example 6–66 shows the FIR filter C code after unrolling the inner loop (identical to Example 6–62 on page 6-119).

Example 6–66. Unrolled FIR Filter C Code

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum0, sum1;
         short x0,x1,x2,x3,h0,h1,h2,h3;

         for (j = 0; j < 100; j+=2) {
                  sum0 = 0;
                  sum1 = 0;
                  x0 = x[j];
                  for (i = 0; i < 32; i+=4){
                           x1 = x[j+i+1];
                           h0 = h[i];
                           sum0 += x0 * h0;
                           sum1 += x1 * h0;
                           x2 = x[j+i+2];
                           h1 = h[i+1];
                           sum0 += x1 * h1;
                           sum1 += x2 * h1;
                           x3 = x[j+i+3];
                           h2 = h[i+2];
                           sum0 += x2 * h2;
                           sum1 += x3 * h2;
                           x0 = x[j+i+4];
                           h3 = h[i+3];
                           sum0 += x3 * h3;
                           sum1 += x0 * h3;
                           }
                  y[j] = sum0 >> 15;
                  y[j+1] = sum1 >> 15;
         }
}
```

Part III

6.12.2 Making the Outer Loop Parallel With the Inner Loop Epilog and Prolog

The final assembly code for the FIR filter with redundant load elimination and no memory hits (shown in Example 6–65 on page 6-126) contained 16 cycles of overhead to call the inner loop every time: ten cycles for the loop prolog and six cycles for the outer loop instructions and branching to the outer loop.

Most of this overhead can be reduced as follows:

- Put the outer loop and branch instructions in parallel with the prolog.
- Create an epilog to the inner loop.
- Put some outer loop instructions in parallel with the inner-loop epilog.

6.12.3 Final Assembly

Example 6–67 shows the final assembly for the FIR filter with a software-pipelined outer loop. Below the inner loop (starting on page 6-131), each instruction is marked in the comments with an e, p, or o for instructions relating to epilog, prolog, or outer loop, respectively.

The inner loop is now only run seven times, because the eighth iteration is done in the epilog in parallel with the prolog of the next inner loop and the outer loop instructions.

Part III

Example 6–67. Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits With Outer Loop Software-Pipelined

```
         MVK      .S1      50,A2            ; set up outer loop counter

         STW      .D2      B11,*B15--       ; push register
||       MVK      .S1      74,A3            ; used to rst x ptr outer loop
||       MVK      .S2      72,B10           ; used to rst h ptr outer loop
||       ADD      .L2X     A6,2,B11         ; set up pointer to y[1]

         LDH      .D1      *A4++,B8         ; x0 = x[j]
||       ADD      .L2X     A4,4,B1          ; set up pointer to x[j+2]
||       ADD      .L1X     B4,2,A8          ; set up pointer to h[1]
||       MVK      .S2      8,B2             ; set up inner loop counter
||[A2]   SUB      .S1      A2,1,A2          ; decrement outer loop counter

         LDH      .D2      *B1++[2],B0      ; x2 = x[j+i+2]
||       LDH      .D1      *A4++[2],A0      ; x1 = x[j+i+1]
||       ZERO     .L1      A9               ; zero out sum0
||       ZERO     .L2      B9               ; zero out sum1

         LDH      .D1      *A8++[2],B6      ; h1 = h[i+1]
||       LDH      .D2      *B4++[2],A1      ; h0 = h[i]

         LDH      .D1      *A4++[2],A5      ; x3 = x[j+i+3]
||       LDH      .D2      *B1++[2],B5      ; x0 = x[j+i+4]

OUTLOOP:
         LDH      .D2      *B4++[2],A7      ; h2 = h[i+2]
||       LDH      .D1      *A8++[2],B8      ; h3 = h[i+3]
||[B2]   SUB      .S2      B2,2,B2          ; decrement loop counter

         LDH      .D2      *B1++[2],B0      ;* x2 = x[j+i+2]
||       LDH      .D1      *A4++[2],A0      ;* x1 = x[j+i+1]

         LDH      .D1      *A8++[2],B6      ;* h1 = h[i+1]
||       LDH      .D2      *B4++[2],A1      ;* h0 = h[i]

         MPY      .M1X     B8,A1,A0         ; x0 * h0
||       MPY      .M2X     A0,B6,B6         ; x1 * h1
||       LDH      .D1      *A4++[2],A5      ;* x3 = x[j+i+3]
||       LDH      .D2      *B1++[2],B5      ;* x0 = x[j+i+4]

  [B2]   B        .S1      LOOP             ; branch to loop
||       MPY      .M2      B0,B6,B7         ; x2 * h1
||       MPY      .M1      A0,A1,A1         ; x1 * h0
||       LDH      .D2      *B4++[2],A7      ;* h2 = h[i+2]
||       LDH      .D1      *A8++[2],B8      ;* h3 = h[i+3]
||[B2]   SUB      .S2      B2,1,B2          ;* decrement loop counter
```

Example 6–67. Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits With Outer Loop Software-Pipelined (Continued)

```
        ADD     .L1     A0,A9,A9        ; sum0 += x0 * h0
||      MPY     .M2X    A5,B8,B8        ; x3 * h3
||      MPY     .M1X    B0,A7,A5        ; x2 * h2
||      LDH     .D2     *B1++[2],B0     ;** x2 = x[j+i+2]
||      LDH     .D1     *A4++[2],A0     ;** x1 = x[j+i+1]

LOOP:
        ADD     .L2X    A1,B9,B9        ; sum1 += x1 * h0
||      ADD     .L1X    B6,A9,A9        ; sum0 += x1 * h1
||      MPY     .M2     B5,B8,B7        ; x0 * h3
||      MPY     .M1     A5,A7,A7        ; x3 * h2
||      LDH     .D1     *A8++[2],B6     ;** h1 = h[i+1]
||      LDH     .D2     *B4++[2],A1     ;** h0 = h[i]

        ADD     .L2     B7,B9,B9        ; sum1 += x2 * h1
||      ADD     .L1     A5,A9,A9        ; sum0 += x2 * h2
||      MPY     .M1X    B5,A1,A0        ;* x0 * h0
||      MPY     .M2X    A0,B6,B6        ;* x1 * h1
||      LDH     .D1     *A4++[2],A5     ;** x3 = x[j+i+3]
||      LDH     .D2     *B1++[2],B5     ;** x0 = x[j+i+4]

        ADD     .L2X    A7,B9,B9        ; sum1 += x3 * h2
||      ADD     .L1X    B8,A9,A9        ; sum0 += x3 * h3
||[B2]  B       .S1     LOOP            ;* branch to loop
||      MPY     .M2     B0,B6,B7        ;* x2 * h1
||      MPY     .M1     A0,A1,A1        ;* x1 * h0
||      LDH     .D2     *B4++[2],A7     ;** h2 = h[i+2]
||      LDH     .D1     *A8++[2],B8     ;** h3 = h[i+3]
||[B2]  SUB     .S2     B2,1,B2         ;** decrement loop counter

        ADD     .L2     B7,B9,B9        ; sum1 += x0 * h3
||      ADD     .L1     A0,A9,A9        ;* sum0 += x0 * h0
||      MPY     .M2X    A5,B8,B8        ;* x3 * h3
||      MPY     .M1X    B0,A7,A5        ;* x2 * h2
||      LDH     .D2     *B1++[2],B0     ;*** x2 = x[j+i+2]
||      LDH     .D1     *A4++[2],A0     ;*** x1 = x[j+i+1]
        ; inner loop branch occurs here

        ADD     .L2X    A1,B9,B9        ;e sum1 += x1 * h0
||      ADD     .L1X    B6,A9,A9        ;e sum0 += x1 * h1
||      MPY     .M2     B5,B8,B7        ;e x0 * h3
||      MPY     .M1     A5,A7,A7        ;e x3 * h2
||      SUB     .D1     A4,A3,A4        ;o reset x pointer to x[j]
||      SUB     .D2     B4,B10,B4       ;o reset h pointer to h[0]
||[A2]  B       .S1     OUTLOOP         ;o branch to outer loop
```

Part III

Example 6–67. Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits With Outer Loop Software-Pipelined (Continued)

```
         ADD     .D2     B7,B9,B9         ;e sum1 += x2 * h1
||       ADD     .L1     A5,A9,A9         ;e sum0 += x2 * h2
||       LDH     .D1     *A4++,B8         ;p x0 = x[j]
||       ADD     .L2X    A4,4,B1          ;o set up pointer to x[j+2]
||       ADD     .S1X    B4,2,A8          ;o set up pointer to h[1]
||       MVK     .S2     8,B2             ;o set up inner loop counter

         ADD     .L2X    A7,B9,B9         ;e sum1 += x3 * h2
||       ADD     .L1X    B8,A9,A9         ;e sum0 += x3 * h3
||       LDH     .D2     *B1++[2],B0      ;p x2 = x[j+i+2]
||       LDH     .D1     *A4++[2],A0      ;p x1 = x[j+i+1]
||[A2]   SUB     .S1     A2,1,A2          ;o decrement outer loop counter

         ADD     .L2     B7,B9,B9         ;e sum1 += x0 * h3
||       SHR     .S1     A9,15,A9         ;e sum0 >> 15
||       LDH     .D1     *A8++[2],B6      ;p h1 = h[i+1]
||       LDH     .D2     *B4++[2],A1      ;p h0 = h[i]

         SHR     .S2     B9,15,B9         ;e sum1 >> 15
||       LDH     .D1     *A4++[2],A5      ;p x3 = x[j+i+3]
||       LDH     .D2     *B1++[2],B5      ;p x0 = x[j+i+4]

         STH     .D1     A9,*A6++[2]      ;e y[j] = sum0 >> 15
||       STH     .D2     B9,*B11++[2]     ;e y[j+1] = sum1 >> 15
||       ZERO    .S1     A9               ;o zero out sum0
||       ZERO    .S2     B9               ;o zero out sum1
         ; outer loop branch occurs here
```


6.12.4 Comparing Performance

The improved cycle count for this loop is 2006 cycles: 50 ((7 × 4) + 6 + 6) + 6. The outer-loop overhead for this loop has been reduced from 16 to 8 (6 + 6 – 4); the –4 represents one iteration less for the inner-loop iteration (seven instead of eight).

Table 6–26. Comparison of FIR Filter Code

Code Example		Cycles	Cycle Count
Example 6–60	FIR with redundant load elimination	50 (16 × 2 + 9 + 6) + 2	2352
Example 6–65	FIR with redundant load elimination and no memory hits	50 (8 × 4 + 10 + 6) + 2	2402
Example 6–67	FIR with redundant load elimination and no memory hits with outer loop software-pipelined	50 (7 × 4 + 6 + 6) + 6	2006

6.13 Outer Loop Conditionally Executed With Inner Loop

Software pipelining the outer loop improved the outer loop overhead in the previous example from 16 cycles to 8 cycles. Executing the outer loop conditionally and in parallel with the inner loop eliminates the overhead entirely.

6.13.1 Unrolled FIR Filter C Code

Example 6–68 shows the same unrolled FIR filter C code that used in the previous example.

Example 6–68. Unrolled FIR Filter C Code

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum0, sum1;
         short x0,x1,x2,x3,h0,h1,h2,h3;

         for (j = 0; j < 100; j+=2) {
                  sum0 = 0;
                  sum1 = 0;
                  x0 = x[j];
                  for (i = 0; i < 32; i+=4){
                           x1 = x[j+i+1];
                           h0 = h[i];
                           sum0 += x0 * h0;
                           sum1 += x1 * h0;
                           x2 = x[j+i+2];
                           h1 = h[i+1];
                           sum0 += x1 * h1;
                           sum1 += x2 * h1;
                           x3 = x[j+i+3];
                           h2 = h[i+2];
                           sum0 += x2 * h2;
                           sum1 += x3 * h2;
                           x0 = x[j+i+4];
                           h3 = h[i+3];
                           sum0 += x3 * h3;
                           sum1 += x0 * h3;
                           }
                  y[j] = sum0 >> 15;
                  y[j+1] = sum1 >> 15;
         }
}
```

6.13.2 Translating C Code to Linear Assembly (Inner Loop)

Example 6–69 shows a list of linear assembly for the inner loop of the FIR filter C code (identical to Example 6–63 on page 6-120).

Example 6–69. Linear Assembly for Unrolled FIR Inner Loop

```
        LDH      *x++,x1           ; x1 = x[j+i+1]
        LDH      *h++,h0           ; h0 = h[i]
        MPY      x0,h0,p00         ; x0 * h0
        MPY      x1,h0,p10         ; x1 * h0
        ADD      p00,sum0,sum0     ; sum0 += x0 * h0
        ADD      p10,sum1,sum1     ; sum1 += x1 * h0

        LDH      *x++,x2           ; x2 = x[j+i+2]
        LDH      *h++,h1           ; h1 = h[i+1]
        MPY      x1,h1,p01         ; x1 * h1
        MPY      x2,h1,p11         ; x2 * h1
        ADD      p01,sum0,sum0     ; sum0 += x1 * h1
        ADD      p11,sum1,sum1     ; sum1 += x2 * h1

        LDH      *x++,x3           ; x3 = x[j+i+3]
        LDH      *h++,h2           ; h2 = h[i+2]
        MPY      x2,h2,p02         ; x2 * h2
        MPY      x3,h2,p12         ; x3 * h2
        ADD      p02,sum0,sum0     ; sum0 += x2 * h2
        ADD      p12,sum1,sum1     ; sum1 += x3 * h2

        LDH      *x++,x0           ; x0 = x[j+i+4]
        LDH      *h++,h3           ; h3 = h[i+3]
        MPY      x3,h3,p03         ; x3 * h3
        MPY      x0,h3,p13         ; x0 * h3
        ADD      p03,sum0,sum0     ; sum0 += x3 * h3
        ADD      p13,sum1,sum1     ; sum1 += x0 * h3

[cntr] SUB       cntr,1,cntr       ; decrement loop counter
[cntr] B         LOOP              ; branch to loop
```

Part III

6.13.3 Translating C Code to Linear Assembly (Outer Loop)

Example 6–70 shows the instructions that execute all of the outer loop functions. All of these instructions are conditional on inner loop counters. Two different counters are needed, because they must decrement to 0 on different iterations.

- The resetting of the x and h pointers is conditional on the pointer reset counter, prc.
- The shifting and storing of the even and odd y elements are conditional on the store counter, sctr.

When these counters are 0, all of the instructions that are conditional on that value execute.

- The MVK instruction resets the pointers to 8 because after every eight iterations of the loop, a new inner loop is completed (8 × 4 elements are processed).
- The pointer reset counter becomes 0 first to reset the load pointers, then the store counter becomes 0 to shift and store the result.

Example 6–70. Linear Assembly for FIR Outer Loop

```
 [sctr]  SUB      sctr,1,sctr               ; dec store lp cntr
[!sctr]  SHR      sum07,15,y0               ; (sum0 >> 15)
[!sctr]  SHR      sum17,15,y1               ; (sum1 >> 15)
[!sctr]  STH      y0,*y++[2]                ; y[j] = (sum0 >> 15)
[!sctr]  STH      y1,*y_1++[2]              ; y[j+1] = (sum1 >> 15)
[!sctr]  MVK      4,sctr                    ; reset store lp cntr
 [pctr]  SUB      pctr,1,pctr               ; dec pointer reset lp cntr
[!pctr]  SUB      x,rstx2,x                 ; reset x ptr
[!pctr]  SUB      x_1,rstx1,x_1             ; reset x_1 ptr
[!pctr]  SUB      h,rsth1,h                 ; reset h ptr
[!pctr]  SUB      h_1,rsth2,h_1             ; reset h_1 ptr
[!pctr]  MVK      4,pctr                    ; reset pointer reset lp cntr
```

6.13.4 Unrolled FIR Filter C Code

The total number of instructions to execute both the inner and outer loops is 38 (26 for the inner loop and 12 for the outer loop). A 4-cycle loop is no longer possible. To avoid slowing down the throughput of the inner loop to reduce the outer-loop overhead, you must unroll the FIR filter again.

Example 6–71 shows the C code for the FIR filter, which operates on eight elements every inner loop. Two outer loops are also being processed together, as in Example 6–68 on page 6-133.

Example 6–71. Unrolled FIR Filter C Code

```
void fir(short x[], short h[], short y[])
{
         int i, j, sum0, sum1;
         short x0,x1,x2,x3,x4,x5,x6,x7,h0,h1,h2,h3,h4,h5,h6,h7;

         for (j = 0; j < 100; j+=2) {
                  sum0 = 0;
                  sum1 = 0;
                  x0 = x[j];
                  for (i = 0; i < 32; i+=8){
                           x1 = x[j+i+1];
                           h0 = h[i];
                           sum0 += x0 * h0;
                           sum1 += x1 * h0;
                           x2 = x[j+i+2];
                           h1 = h[i+1];
                           sum0 += x1 * h1;
                           sum1 += x2 * h1;
                           x3 = x[j+i+3];
                           h2 = h[i+2];
                           sum0 += x2 * h2;
                           sum1 += x3 * h2;
                           x4 = x[j+i+4];
                           h3 = h[i+3];
                           sum0 += x3 * h3;
                           sum1 += x4 * h3;
                           x5 = x[j+i+5];
                           h4 = h[i+4];
                           sum0 += x4 * h4;
                           sum1 += x5 * h4;
                           x6 = x[j+i+6];
                           h5 = h[i+5];
                           sum0 += x5 * h5;
                           sum1 += x6 * h5;
                           x7 = x[j+i+7];
                           h6 = h[i+6];
                           sum0 += x6 * h6;
                           sum1 += x7 * h6;
                           x0 = x[j+i+8];
                           h7 = h[i+7];
                           sum0 += x7 * h7;
                           sum1 += x0 * h7;
                           }
                  y[j] = sum0 >> 15;
                  y[j+1] = sum1 >> 15;
         }
}
```

6.13.5 Translating C Code to Linear Assembly (Inner Loop)

Example 6–72 shows the instructions that perform the inner and outer loops of the FIR filter. These instructions reflect the following modifications:

- LDWs are used instead of LDHs to reduce the number of loads in the loop.
- The reset pointer instructions immediately follow the LDW instructions.
- The first ADD instructions for sum0 and sum1 are conditional on the same value as the store counter, because when sctr is 0, the end of one inner loop has been reached and the first ADD, which adds the previous sum07 to p00, must not be executed.
- The first ADD for sum0 writes to the same register as the first MPY p00. The second ADD reads p00 and p01. At the beginning of each inner loop, the first ADD is not performed, so the second ADD correctly reads the results of the first two MPYs (p01 and p00) and adds them together. For other iterations of the inner loop, the first ADD executes, and the second ADD sums the second MPY result (p01) with the running accumulator. The same is true for the first and second ADDs of sum1.

Example 6–72. Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop

```
        LDW     *h++[2],h01       ; h[i+0] & h[i+1]
        LDW     *h_1++[2],h23     ; h[i+2] & h[i+3]
        LDW     *h++[2],h45       ; h[i+4] & h[i+5]
        LDW     .*h_1++[2],h67    ; h[i+6] & h[i+7]

        LDW     *x++[2],x01       ; x[j+i+0] & x[j+i+1]
        LDW     *x_1++[2],x23     ; x[j+i+2] & x[j+i+3]
        LDW     *x++[2],x45       ; x[j+i+4] & x[j+i+5]
        LDW     *x_1++[2],x67     ; x[j+i+6] & x[j+i+7]
        LDH     *x,x8             ; x[j+i+8]

 [sctr] SUB     sctr,1,sctr       ; dec store lp cntr
[!sctr] SHR     sum07,15,y0       ; (sum0 >> 15)
[!sctr] SHR     sum17,15,y1       ; (sum1 >> 15)
[!sctr] STH     y0,*y++[2]        ; y[j] = (sum0 >> 15)
[!sctr] STH     y1,*y_1++[2]      ; y[j+1] = (sum1 >> 15)

        MV      x01,x01b          ; move to other reg file
        MPYLH   h01,x01b,p10      ; p10 = h[i+0]*x[j+i+1]
[sctr]  ADD     p10,sum17,p10     ; sum1(p10) = p10 + sum1

        MPYHL   h01,x23,p11       ; p11 = h[i+1]*x[j+i+2]
        ADD     p11,p10,sum11     ; sum1 += p11

        MPYLH   h23,x23,p12       ; p12 = h[i+2]*x[j+i+3]
        ADD     p12,sum11,sum12   ; sum1 +=  p12

        MPYHL   h23,x45,p13       ; p13 = h[i+3]*x[j+i+4]
        ADD     p13,sum12,sum13   ; sum1 += p13

        MPYLH   h45,x45,p14       ; p14 = h[i+4]*x[j+i+5]
        ADD     p14,sum13,sum14   ; sum1 += p14

        MPYHL   h45,x67,p15       ; p15 = h[i+5]*x[j+i+6]
        ADD     p15,sum14,sum15   ; sum1 += p15

        MPYLH   h67,x67,p16       ; p16 = h[i+6]*x[j+i+7]
        ADD     p16,sum15,sum16   ; sum1 += p16

        MPYHL   h67,x8,p17        ; p17 = h[i+7]*x[j+i+8]
        ADD     p17,sum16,sum17   ; sum1 += p17

        MPY     h01,x01,p00       ; p00 = h[i+0]*x[j+i+0]
[sctr]  ADD     p00,sum07,p00     ; sum0(p00) = p00 + sum0

        MPYH    h01,x01,p01       ; p01 = h[i+1]*x[j+i+1]
        ADD     p01,p00,sum01     ; sum0 += p01
```

Example 6–72. Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (Continued)

```
        MPY     h23,x23,p02         ; p02 = h[i+2]*x[j+i+2]
        ADD     p02,sum01,sum02     ; sum0 += p02

        MPYH    h23,x23,p03         ; p03 = h[i+3]*x[j+i+3]
        ADD     p03,sum02,sum03     ; sum0 += p03

        MPY     h45,x45,p04         ; p04 = h[i+4]*x[j+i+4]
        ADD     p04,sum03,sum04     ; sum0 += p04

        MPYH    h45,x45,p05         ; p05 = h[i+5]*x[j+i+5]
        ADD     p05,sum04,sum05     ; sum0 += p05

        MPY     h67,x67,p06         ; p06 = h[i+6]*x[j+i+6]
        ADD     p06,sum05,sum06     ; sum0 += p06

        MPYH    h67,x67,p07         ; p07 = h[i+7]*x[j+i+7]
        ADD     p07,sum06,sum07     ; sum0 += p07

[!sctr] MVK     4,sctr              ; reset store lp cntr

 [pctr] SUB     pctr,1,pctr         ; dec pointer reset lp cntr
[!pctr] SUB     x,rstx2,x           ; reset x ptr
[!pctr] SUB     x_1,rstx1,x_1       ; reset x_1 ptr
[!pctr] SUB     h,rsth1,h           ; reset h ptr
[!pctr] SUB     h_1,rsth2,h_1       ; reset h_1 ptr
[!pctr] MVK     4,pctr              ; reset pointer reset lp cntr

 [octr] SUB     octr,1,octr         ; dec outer lp cntr
 [octr] B       LOOP                ; Branch outer loop
```


6.13.6 Translating C Code to Linear Assembly (Inner Loop and Outer Loop)

Example 6–73 shows the linear assembly with functional units assigned. (As in Example 6–64 on page 6-122, symbolic names now have an A or B in front of them to signify the register file where they reside.) Although this allocation is one of many possibilities, one goal is to keep the 1X and 2X paths to a minimum. Even with this goal, you have five 2X paths and seven 1X paths.

One requirement that was assumed when the functional units were chosen was that all the sum0 values reside on the same side (A in this case) and all the sum1 values reside on the other side (B). Because you are scheduling eight accumulates for both sum0 and sum1 in an 8-cycle loop, each ADD must be scheduled immediately following the previous ADD. Therefore, it is undesirable for any sum0 ADDs to use the same functional units as sum1 ADDs.

One MV instruction was added to get x01 on the B side for the MPYLH p10 instruction.

Example 6–73. Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (With Functional Units)

```
         .global _fir

_fir:    .cproc   x, h, y

         .reg     x_1, h_1, y_1, octr, pctr, sctr
         .reg     sum01, sum02, sum03, sum04, sum05, sum06, sum07
         .reg     sum11, sum12, sum13, sum14, sum15, sum16, sum17
         .reg     p00, p01, p02, p03, p04, p05, p06, p07
         .reg     p10, p11, p12, p13, p14, p15, p16, p17
         .reg     x01b, x01, x23, x45, x67, x8, h01, h23, h45, h67
         .reg     y0, y1, rstx1, rstx2, rsth1, rsth2

         ADD      x,4,x_1               ; point to x[2]
         ADD      h,4,h_1               ; point to h[2]
         ADD      y,2,y_1               ; point to y[1]
         MVK      60,rstx1              ; used to rst x pointer each outer loop
         MVK      60,rstx2              ; used to rst x pointer each outer loop
         MVK      64,rsth1              ; used to rst h pointer each outer loop
         MVK      64,rsth2              ; used to rst h pointer each outer loop
         MVK      201,octr              ; loop ctr = 201 = (100/2) * (32/8) + 1
         MVK      4,pctr                ; pointer reset lp cntr = 32/8
         MVK      5,sctr                ; reset store lp cntr = 32/8 + 1
         ZERO     sum07                 ; sum07 = 0
         ZERO     sum17                 ; sum17 = 0

         .mptr    x,   x+0
         .mptr    x_1, x+4
         .mptr    h,   h+0
         .mptr    h_1, h+4

LOOP:    .trip 8


         LDW      .D1T1    *h++[2],h01   ; h[i+0] & h[i+1]
         LDW      .D2T2    *h_1++[2],h23; h[i+2] & h[i+3]
         LDW      .D1T1    *h++[2],h45   ; h[i+4] & h[i+5]
         LDW      .D2T2    *h_1++[2],h67; h[i+6] & h[i+7]

         LDW      .D2T1    *x++[2],x01   ; x[j+i+0] & x[j+i+1]
         LDW      .D1T2    *x_1++[2],x23; x[j+i+2] & x[j+i+3]
         LDW      .D2T1    *x++[2],x45   ; x[j+i+4] & x[j+i+5]
         LDW      .D1T2    *x_1++[2],x67; x[j+i+6] & x[j+i+7]
         LDH      .D2T1    *x,x8         ; x[j+i+8]

 [sctr]  SUB      .S1      sctr,1,sctr   ; dec store lp cntr
[!sctr]  SHR      .S1      sum07,15,y0   ; (sum0 >> 15)
[!sctr]  SHR      .S2      sum17,15,y1   ; (sum1 >> 15)
[!sctr]  STH      .D1      y0,*y++[2]    ; y[j] = (sum0 >> 15)
[!sctr]  STH      .D2      y1,*y_1++[2] ; y[j+1] = (sum1 >> 15)
```

Example 6–73. Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (With Functional Units) (Continued)

```
        MV       .L2X     x01,x01b          ; move to other reg file
        MPYLH    .M2X     h01,x01b,p10      ; p10 = h[i+0]*x[j+i+1]

[sctr]  ADD      .L2      p10,sum17,p10     ; sum1(p10) = p10 + sum1

        MPYHL    .M1X     h01,x23,p11       ; p11 = h[i+1]*x[j+i+2]
        ADD      .L2X     p11,p10,sum11     ; sum1 += p11

        MPYLH    .M2      h23,x23,p12       ; p12 = h[i+2]*x[j+i+3]
        ADD      .L2      p12,sum11,sum12   ; sum1 +=  p12

        MPYHL    .M1X     h23,x45,p13       ; p13 = h[i+3]*x[j+i+4]
        ADD      .L2X     p13,sum12,sum13   ; sum1 += p13

        MPYLH    .M1      h45,x45,p14       ; p14 = h[i+4]*x[j+i+5]
        ADD      .L2X     p14,sum13,sum14   ; sum1 += p14

        MPYHL    .M2X     h45,x67,p15       ; p15 = h[i+5]*x[j+i+6]
        ADD      .S2      p15,sum14,sum15   ; sum1 += p15

        MPYLH    .M2      h67,x67,p16       ; p16 = h[i+6]*x[j+i+7]
        ADD      .L2      p16,sum15,sum16   ; sum1 += p16

        MPYHL    .M1X     h67,x8,p17        ; p17 = h[i+7]*x[j+i+8]
        ADD      .L2X     p17,sum16,sum17   ; sum1 += p17

        MPY      .M1      h01,x01,p00       ; p00 = h[i+0]*x[j+i+0]
[sctr]  ADD      .L1      p00,sum07,p00     ; sum0(p00) = p00 + sum0

        MPYH     .M1      h01,x01,p01       ; p01 = h[i+1]*x[j+i+1]
        ADD      .L1      p01,p00,sum01     ; sum0 += p01

        MPY      .M2      h23,x23,p02       ; p02 = h[i+2]*x[j+i+2]
        ADD      .L1X     p02,sum01,sum02   ; sum0 += p02

        MPYH     .M2      h23,x23,p03       ; p03 = h[i+3]*x[j+i+3]
        ADD      .L1X     p03,sum02,sum03   ; sum0 += p03

        MPY      .M1      h45,x45,p04       ; p04 = h[i+4]*x[j+i+4]
        ADD      .L1      p04,sum03,sum04   ; sum0 += p04

        MPYH     .M1      h45,x45,p05       ; p05 = h[i+5]*x[j+i+5]
        ADD      .L1      p05,sum04,sum05   ; sum0 += p05
```

Example 6–73. Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (With Functional Units)(Continued)

```
        MPY     .M2     h67,x67,p06         ; p06 = h[i+6]*x[j+i+6]
        ADD     .L1X    p06,sum05,sum06     ; sum0 += p06

        MPYH    .M2     h67,x67,p07         ; p07 = h[i+7]*x[j+i+7]
        ADD     .L1X    p07,sum06,sum07     ; sum0 += p07

[!sctr] MVK     .S1     4,sctr              ; reset store lp cntr

 [pctr] SUB     .S1     pctr,1,pctr         ; dec pointer reset lp cntr
[!pctr] SUB     .S2     x,rstx2,x           ; reset x ptr
[!pctr] SUB     .S1     x_1,rstx1,x_1       ; reset x_1 ptr
[!pctr] SUB     .S1     h,rsth1,h           ; reset h ptr
[!pctr] SUB     .S2     h_1,rsth2,h_1       ; reset h_1 ptr
[!pctr] MVK     .S1     4,pctr              ; reset pointer reset lp cntr

 [octr] SUB     .S2     octr,1,octr         ; dec outer lp cntr
 [octr] B       .S2     LOOP                ; Branch outer loop

.endproc
```

Part III

6.13.7 Determining the Minimum Iteration Interval

Based on Table 6–27, the minimum iteration interval is 8. An iteration interval of 8 means that two multiply-accumulates per cycle are still executing.

Table 6–27. Resource Table for FIR Filter Code

(a) A side | *(b) B side*

Unit(s)	Total/Unit	Unit(s)	Total/Unit
.M1	8	.M2	8
.S1	7	.S2	6
.D1	5	.D2	6
.L1	8	.L2	8
Total non-.M units	20	Total non-.M units	20
1X paths	7	2X paths	5

6.13.8 Final Assembly

Example 6–74 shows the final assembly for the FIR filter with the outer loop conditionally executing in parallel with the inner loop.

Example 6–74. Final Assembly Code for FIR Filter

```
        MV      .L1X    B4,A0           ; point to h[0] & h[1]
||      ADD     .D2     B4,4,B2         ; point to h[2] & h[3]
||      MV      .L2X    A4,B1           ; point to x[j] & x[j+1]
||      ADD     .D1     A4,4,A4         ; point to x[j+2] & x[j+3]
||      MVK     .S2     200,B0          ; set lp ctr ((32/8)*(100/2))

        LDW     .D1     *A4++[2],B9     ; x[j+i+2] & x[j+i+3]
||      LDW     .D2     *B1++[2],A10    ; x[j+i+0] & x[j+i+1]
||      MVK     .S1     4,A1            ; set pointer reset lp cntr

        LDW     .D2     *B2++[2],B7     ; h[i+2] & h[i+3]
||      LDW     .D1     *A0++[2],A8     ; h[i+0] & h[i+1]
||      MVK     .S1     60,A3           ; used to reset x ptr (16*4–4)
||      MVK     .S2     60,B14          ; used to reset x ptr (16*4–4)

        LDW     .D2     *B1++[2],A11    ; x[j+i+4] & x[j+i+5]
||      LDW     .D1     *A4++[2],B10    ; x[j+i+6] & x[j+i+7]
||[A1]  SUB     .L1     A1,1,A1         ; dec pointer reset lp cntr
||      MVK     .S1     64,A5           ; used to reset h ptr (16*4)
||      MVK     .S2     64,B5           ; used to reset h ptr (16*4)
||      ADD     .L2X    A6,2,B6         ; point to y[j+1]

        LDW     .D1     *A0++[2],A9     ; h[i+4] & h[i+5]
||      LDW     .D2     *B2++[2],B8     ; h[i+6] & h[i+7]
||[!A1] SUB     .S1     A4,A3,A4        ; reset x ptr

  [!A1] SUB     .S2     B1,B14,B1       ; reset x ptr
||[!A1] SUB     .S1     A0,A5,A0        ; reset h ptr
||      LDH     .D2     *B1,A8          ; x[j+i+8]

        ADD     .S2X    A10,0,B8        ; move to other reg file
||      MVK     .S1     5,A2            ; set store lp cntr

        MPYLH   .M2X    A8,B8,B4        ; p10 = h[i+0]*x[j+i+1]
||[!A1] SUB     .S2     B2,B5,B2        ; reset h ptr
||      MPYHL   .M1X    A8,B9,A14       ; p11 = h[i+1]*x[j+i+2]

        MPY     .M1     A8,A10,A7       ; p00 = h[i+0]*x[j+i+0]
||      MPYLH   .M2     B7,B9,B13       ; p12 = h[i+2]*x[j+i+3]
||[A2]  SUB     .S1     A2,1,A2         ; dec store lp cntr
||      ZERO    .L2     B11             ; zero out initial accumulator

  [!A2] SHR     .S2     B11,15,B11      ; (Bsum1 >> 15)
||      MPY     .M2     B7,B9,B9        ; p02 = h[i+2]*x[j+i+2]
||      MPYH    .M1     A8,A10,A10      ; p01 = h[i+1]*x[j+i+1]
||[A2]  ADD     .L2     B4,B11,B4       ; sum1(p10) = p10 + sum1
||      LDW     .D1     *A4++[2],B9     ;* x[j+i+2] & x[j+i+3]
||      LDW     .D2     *B1++[2],A10    ;* x[j+i+0] & x[j+i+1]
||      ZERO    .L1     A10             ; zero out initial accumulator
```

Part III

Example 6–74. Final Assembly Code for FIR Filter (Continued)

```
LOOP:
  [!A2] SHR      .S1      A10,15,A12        ; (Asum0 >> 15)
||[B0]  SUB      .S2      B0,1,B0           ; dec outer lp cntr
||      MPYH     .M2      B7,B9,B13         ; p03 = h[i+3]*x[j+i+3]
||[A2]  ADD      .L1      A7,A10,A7         ; sum0(p00) = p00 + sum0
||      MPYHL    .M1X     B7,A11,A10        ; p13 = h[i+3]*x[j+i+4]
||      ADD      .L2X     A14,B4,B7         ; sum1 += p11
||      LDW      .D2      *B2++[2],B7       ;* h[i+2] & h[i+3]
||      LDW      .D1      *A0++[2],A8       ;* h[i+0] & h[i+1]

        ADD      .L1      A10,A7,A13        ; sum0 += p01
||      MPYHL    .M2X     A9,B10,B12        ; p15 = h[i+5]*x[j+i+6]
||      MPYLH    .M1      A9,A11,A10        ; p14 = h[i+4]*x[j+i+5]
||      ADD      .L2      B13,B7,B7         ; sum1 +=  p12
||      LDW      .D2      *B1++[2],A11      ;* x[j+i+4] & x[j+i+5]
||      LDW      .D1      *A4++[2],B10      ;* x[j+i+6] & x[j+i+7]
||[A1]  SUB      .S1      A1,1,A1           ;* dec pointer reset lp cntr

  [B0]  B        .S2      LOOP              ; Branch outer loop
||      MPY      .M1      A9,A11,A11        ; p04 = h[i+4]*x[j+i+4]
||      ADD      .L1X     B9,A13,A13        ; sum0 += p02
||      MPYLH    .M2      B8,B10,B13        ; p16 = h[i+6]*x[j+i+7]
||      ADD      .L2X     A10,B7,B7         ; sum1 += p13
||      LDW      .D1      *A0++[2],A9       ;* h[i+4] & h[i+5]
||      LDW      .D2      *B2++[2],B8       ;* h[i+6] & h[i+7]
||[!A1] SUB      .S1      A4,A3,A4          ;* reset x ptr

        MPY      .M2      B8,B10,B11        ; p06 = h[i+6]*x[j+i+6]
||      MPYH     .M1      A9,A11,A11        ; p05 = h[i+5]*x[j+i+5]
||      ADD      .L1X     B13,A13,A9        ; sum0 += p03
||      ADD      .L2X     A10,B7,B7         ; sum1 += p14
||[!A1] SUB      .S2      B1,B14,B1         ;* reset x ptr
||[!A1] SUB      .S1      A0,A5,A0          ;* reset h ptr
||      LDH      .D2      *B1,A8            ;* x[j+i+8]

  [!A2] MVK      .S1      4,A2              ; reset store lp cntr
||      MPYH     .M2      B8,B10,B13        ; p07 = h[i+7]*x[j+i+7]
||      ADD      .L1      A11,A9,A9         ; sum0 += p04
||      MPYHL    .M1X     B8,A8,A9          ; p17 = h[i+7]*x[j+i+8]
||      ADD      .S2      B12,B7,B10        ; sum1 += p15
||[!A2] STH      .D2      B11,*B6++[2]      ; y[j+1] = (Bsum1 >> 15)
||[!A2] STH      .D1      A12,*A6++[2]      ; y[j] = (Asum0 >> 15)
||      ADD      .L2X     A10,0,B8          ;* move to other reg file

        ADD      .L1      A11,A9,A12        ; sum0 += p05
||      ADD      .L2      B13,B10,B8        ; sum1 += p16
||      MPYLH    .M2X     A8,B8,B4          ;* p10 = h[i+0]*x[j+i+1]
||[!A1] MVK      .S1      4,A1              ;* reset pointer reset lp cntr
||[!A1] SUB      .S2      B2,B5,B2          ;* reset h ptr
||      MPYHL    .M1X     A8,B9,A14         ;* p11 = h[i+1]*x[j+i+2]
```

Example 6–74. Final Assembly Code for FIR Filter (Continued)

```
        ADD      .L2X     A9,B8,B11          ; sum1 += p17
||      ADD      .L1X     B11,A12,A12        ; sum0 += p06
||      MPY      .M1      A8,A10,A7          ;* p00 = h[i+0]*x[j+i+0]
||      MPYLH    .M2      B7,B9,B13          ;* p12 = h[i+2]*x[j+i+3]
||[A2]  SUB      .S1      A2,1,A2            ;* dec store lp cntr

        ADD      .L1X     B13,A12,A10        ; sum0 += p07
||[!A2] SHR      .S2      B11,15,B11         ;* (Bsum1 >> 15)
||      MPY      .M2      B7,B9,B9           ;* p02 = h[i+2]*x[j+i+2]
||      MPYH     .M1      A8,A10,A10         ;* p01 = h[i+1]*x[j+i+1]
||[A2]  ADD      .L2      B4,B11,B4          ;* sum1(p10) = p10 + sum1
||      LDW      .D1      *A4++[2],B9        ;** x[j+i+2] & x[j+i+3]
||      LDW      .D2      *B1++[2],A10       ;** x[j+i+0] & x[j+i+1]
        ;Branch occurs here

  [!A2] SHR      .S1      A10,15,A12         ; (Asum0 >> 15)

  [!A2] STH      .D2      B11,*B6++[2]       ; y[j+1] = (Bsum1 >> 15)
||[!A2] STH      .D1      A12,*A6++[2]       ; y[j] = (Asum0 >> 15)
```

6.13.9 Comparing Performance

The cycle count of this code is 1612: 50 (8 × 4 + 0) + 12. The overhead due to the outer loop has been completely eliminated.

Table 6–28. Comparison of FIR Filter Code

Code Example		Cycles	Cycle Count
Example 6–57	FIR with redundant load elimination	50 (16 × 2 + 9 + 6) + 2	2352
Example 6–65	FIR with redundant load elimination and no memory hits	50 (8 × 4 + 10 + 6) + 2	2402
Example 6–67	FIR with redundant load elimination and no memory hits with outer loop software-pipelined	50 (7 × 4 + 6 + 6) + 6	2006
Example 6–70	FIR with redundant load elimination and no memory hits with outer loop conditionally executed with inner loop	50 (8 × 4 + 0) + 12	1612

Part III

Chapter 7

Interrupts

This chapter describes interrupts from a software-programming point of view. A description of single and multiple register assignment is included, followed by code generation of interruptible code and finally, descriptions of interrupt subroutines.

Topic **Page**

7.1 Overview of Interrupts 7-2

7.2 Single Assignment vs. Multiple Assignment 7-3

7.3 Interruptible Loops 7-5

7.4 Interruptible Code Generation 7-6

7.5 Interrupt Subroutines 7-8

Part III

7.1 Overview of Interrupts

An interrupt is an event that stops the current process in the CPU so that the CPU can attend to the task needing completion because of another event. These events are external to the core CPU but may originate on-chip or off-chip. Examples of on-chip interrupt sources include timers, serial ports, DMAs and external memory stalls. Examples of off-chip interrupt sources include analog-to-digital converters, host controllers and other peripheral devices.

Typically, DSPs compute different algorithms very quickly within an asynchronous system environment. Asynchronous systems must be able to control the DSP based on events outside of the DSP core. Because certain events can have higher priority than algorithms already executing on the DSP, it is sometimes necessary to change, or interrupt, the task currently executing on the DSP.

The ’C6x provides hardware interrupts that allow this to occur automatically. Once an interrupt is taken, an interrupt subroutine performs certain tasks or actions, as required by the event. Servicing an interrupt involves switching contexts while saving all state of the machine. Thus, upon return from the interrupt, operation of the interrupted algorithm is resumed as if there had been no interrupt. Saving state involves saving various registers upon entry to the interrupt subroutine and then restoring them to their original state upon exit.

This chapter focuses on the software issues associated with interrupts. The hardware description of interrupt operation is fully described in the *TMS320C6x CPU and Instruction Set Reference Guide*.

In order to understand the software issues of interrupts, we must talk about two types of code: the code that is interrupted and the interrupt subroutine, which performs the tasks required by the interrupt. The following sections provide information on:

- Single and multiple assignment of registers
- Loop interruptibility
- How to use the ’C6x code generation tools to satisfy different requirements
- Interrupt subroutines

7.2 Single Assignment vs. Multiple Assignment

Register allocation on the 'C6x can be classified as either single assignment or multiple assignment. Single assignment code is interruptible; multiple assignment is not interruptible. This section discusses the differences between each and explains why only single assignment is interruptible.

Example 7–1 shows multiple assignment code. The term multiple assignment means that a particular register has been assigned with more than one value (in this case 2 values). On cycle 4, at the beginning of the ADD instruction, register A1 is assigned to two different values. One value, written by the SUB instruction on cycle 1, already resides in the register. The second value is called an *in-flight* value and is assigned by the LDW instruction on cycle 2. Because the LDW instruction does not actually write a value into register A1 until the end of cycle 6, the assignment is considered in-flight.

In-flight operations cause code to be uninterruptible due to unpredictability. Take, for example, the case where an interrupt is taken on cycle 3. At this point, all instructions which have begun execution are allowed to complete and no new instructions execute. So, 3 cycles after the interrupt is taken on cycle 3, the LDW instruction writes to A1. After the interrupt service routine has been processed, program execution continues on cycle 4 with the ADD instruction. In this case, the ADD reads register A1 and will be reading the result of the LDW, whereas normally the result of the SUB should be read. This unpredictability means that in order to ensure correct operation, multiple assignment code should not be interrupted and is thus, considered uninterruptible.

Example 7–1. Code With Multiple Assignment of A1

```
cycle
   1      SUB    .S1    A4,A5,A1  ; writes to A1 in single cycle
   2      LDW    .D1    *A0,A1    ; writes to A1 after 4 delay slots
   3      NOP
   4      ADD    .L1    A1,A2,A3  ; uses old A1 (result of SUB)
   5–6    NOP           2
   7      MPY    .M1    A1,A4,A5  ; uses new A1 (result of LDW)
```

Example 7–2 shows the same code with a new register allocation to produce single assignment code. Now the LDW assigns a value to register A6 instead of A1. Now, regardless of whether an interrupt is taken or not, A1 maintains the value written by the SUB instruction because LDW now writes to A6. Because there are no in-flight registers that are read before an in-flight instruction completes, this code is interruptible.

Example 7–2. Code Using Single Assignment

```
cycle
   1   SUB    .S1    A4,A5,A1  ; writes to A1 in single cycle
   2   LDW    .D1    *A0,A6    ; writes to A1 after 4 delay slots
   3   NOP
   4   ADD    .L1    A1,A2,A3  ; uses old A1 (result of SUB)
   5–6 NOP           2
   7   MPY    .M1    A6,A4,A5  ; uses new A1 (result of LDW)
```

Both examples involve exactly the same schedule of instructions. The only difference is the register allocation. The single assignment register allocation, as shown in Example 7–2, can result in higher register pressure (Example 7–2 uses one more register than Example 7–1).

The next section describes how to generate interruptible and non-interruptible code with the ’C6x code generation tools.

7.3 Interruptible Loops

Even if code employs single assignment, it may not be interruptible in a loop. Because the delay slots of all branch operations are protected from interrupts in hardware, all interrupts remain pending as long as the CPU has a pending branch. Since the branch instruction on the ’C6x has 5 delay slots, loops smaller than 6 cycles always have a pending branch. For this reason, all loops smaller than 6 cycles are uninterruptible.

There are two options for making a loop with an iteration interval less than 6 interruptible.

1) Simply slow down the loop and force an iteration interval of 6 cycles. This is not always desirable since there will be a performance degradation.

2) Unroll the loop until an iteration interval of 6 or greater is achieved. This ensures at least the same performance level and in some cases can improve performance (see section 6.8 *Loop Unrolling*). The disadvantage is that code size increases.

The next section describes how to automatically generate these different options with the ’C6x code generation tools.

Part III

7.4 Interruptible Code Generation

The ’C6x code generation tools provide a large degree of flexibility for interruptibility. Various combinations of single and multiple assignment code can be generated automatically to provide the best tradeoff in interruptibility and performance for each part of an application. In most cases, code performance is not affected by interruptibility, but there are some exceptions:

- Software pipelined loops that have high register pressure can fail to register allocate at a given iteration interval when single assignment is required, but might otherwise succeed to allocate if multiple assignment were allowed. This can result in a larger iteration interval for single assignment software pipelined loops and thus lower performance. To determine if this is the problem for looped code, use the –mw feedback option.
- Because loops with minimum iteration intervals less than 6 are not interruptible, higher iteration intervals might be used which results in lower performance. Unrolling the loop, however, prevents this reduction in performance (see section 7.2)
- Higher register pressure in single assignment can cause data spilling to memory in both looped code and non-looped code when there are not enough registers to store all temporary values. This reduces performance but occurs rarely and only in extreme cases.

The tools provide 3 levels of control to the user. These levels are described in the following sections. For a full discussion of interruptible code generation, see the *TMS320C6x Optimizing C Compiler User’s Guide*.

7.4.1 Level 0 – Specified Code is Guaranteed to Not Be Interrupted

The compiler does not disable interrupts. Thus, it is up to the system developer to guarantee that no interrupts occur. This level has the advantage that the compiler is allowed to use multiple assignment code and generate the minimum iteration intervals for software pipelined loops.

The command line option –mi can be used for an entire module and the following pragma can be used to force this level on a particular function:

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, uint_max);
```

7.4.2 Level 1 – Specified Code Interruptible at All Times

The compiler will not disable interrupts. Thus, the compiler will employ single assignment everywhere and will never produce a loop of less than 6 cycles. The command line option –mi1 can be used for an entire module and the following pragma can be used to force this level on a particular function:

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, 1);
```

7.4.3 Level 2 – Specified Code Interruptible Within Threshold Cycles

The compiler will disable interrupts around loops if the specified threshold number is not exceeded. In other words, the user can specify a threshold, or maximum interrupt delay, that allows the compiler to use multiple assignment in loops that do not exceed this threshold. The code outside of loops can have interrupts disabled and also use multiple assignment as long as the threshold of uninterruptible cycles is not exceeded. If the compiler cannot determine the loop count of a loop, then it assumes the threshold is exceeded and will generate an interruptible loop.

The command line option –mi (threshold) can be used for an entire module and the following pragma can be used to specify a threshold for a particular function.

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);
```

7.5 Interrupt Subroutines

The interrupt subroutine (ISR) is simply the routine, or function, that is called by an interrupt. The 'C6x provides hardware to automatically branch to this routine when an interrupt is received based on an interrupt service table. (See the *Interrupt Service Table* in the *TMS320C6x CPU and Instruction Set Reference Guide.*) Once the branch is complete, execution begins at the first execute packet of the ISR.

Certain state must be saved upon entry to an ISR in order to ensure program accuracy upon return from the interrupt. For this reason, all registers that are used by the ISR must be saved to memory, preferably a stack pointed to by a general purpose register acting as a stack pointer. Then, upon return, all values must be restored. This is all handled automatically by the C compiler, but must be done manually when writing hand-coded assembly.

7.5.1 ISR with the C Compiler

The C compiler automatically generates ISRs with the keyword *interrupt*. The interrupt function must be declared with no arguments and should return void. For example:

```
interrupt void int_handler()
 {
 unsigned int flags;
 ...
 }
```

Alternatively, you can use the interrupt pragma to define a function to be an ISR:

```
#pragma INTERRUPT(func);
```

The result either case is that the C compiler automatically creates a function that obeys all the requirements for an ISR. These are different from the calling convention of a normal C function in the following ways:

- All general purpose registers used by the subroutine must be saved to the stack. If another function is called from the ISR, then all the registers (A0–A15, B0–B15) are saved to the stack.
- A B IRP instruction is used to return from the interrupt subroutine instead of the B B3 instruction used for standard C functions
- A function cannot return a value and thus, must be declared void.

See the section on *Register Conventions* in the *TMS320C6x Optimizing C Compiler User's Guide* for more information on standard function calling conventions.

7.5.2 ISR with Hand-Coded Assembly

When writing an ISR by hand, it is necessary to handle the same tasks the C compiler does. So, the following steps must be taken:

- All registers used must be saved to the stack before modification. For this reason, it is preferable to maintain one general purpose register to be used as a stack pointer in your application. (The C compiler uses B15.)
- If another C routine is called from the ISR (with an assembly branch instruction to the _c_func_name label) then all registers must be saved to the stack on entry.
- A B IRP instruction must be used to return from the routine. If this is the NMI ISR, a B NRP must be used instead.
- An NOP 4 is required after the last LDW in this case to ensure that B0 is restored before returning from the interrupt.

Example 7–3. Hand-Coded Assembly ISR

```
* Assume Register B0–B4 & A0 are the only registers used by the
* ISR and no other functions are called
   STW  B0,*B15––      ; store B0 to stack
   STW  A0,*B15––      ; store A0 to stack
   STW  B1,*B15––      ; store B1 to stack
   STW  B2,*B15––      ; store B2 to stack
   STW  B3,*B15––      ; store B3 to stack
   STW  B4,*B15––      ; store B4 to stack
* Beginning of ISR code
    ...
* End of ISR code

   LDW  *++B15,B4      ; restore B4
   LDW  *++B15,B3      ; restore B3
   LDW  *++B15,B2      ; restore B2
   LDW  *++B15,B1      ; restore B1
   LDW  *++B15,A0      ; restore A0
|| B     IRP           ; return from interrupt
   LDW  *++B15,B0      ; restore B0
   NOP  4              ; allow all multi–cycle instructions
                       ; to complete before branch is taken
```

7.5.3 Nested Interrupts

Sometimes it is desirable to allow higher priority interrupts to interrupt lower priority ISRs. To allow nested interrupts to occur, you must first save the IRP, IER, and CSR to a register which is not being used or to or some other memory location (usually the stack). Once these have been saved, you can reenable

the appropriate interrupts. This involves resetting the GIE bit and then doing any necessary modifications to the IER, providing only certain interrupts are allowed to interrupt the particular ISR. On return from the ISR, the original values of the IRP, IER, and CSR must be restored.

Example 7–4. Hand-Coded Assembly ISR Allowing Nesting of Interrupts

```
* Assume Register B0–B4 & A0 are the only registers used by the
* ISR and no other functions are called
   STW    B0,*B15--     ; store B0 to stack
|| MVC    IRP, B0       ; save IRP
   STW    A0,*B15--     ; store A0 to stack
|| MVC    IER, B1       ; save IER
|| MVK    mask,A0       ; setup a new IER (if desirable)
   STW    B1,*B15--     ; store B1 to stack
|| MVC    A0, IER       ; setup a new IER (if desirable)
   STW    B2,*B15--     ; store B2 to stack
|| MVC    CSR,A0        ; read current CSR
   STW    B3,*B15--     ; store B3 to stack
|| OR     1,A0,A0       ; set GIE bit field in CSR
   STW    B4,*B15--     ; store B4 to stack
|| MVC    A0,CSR        ; write new CSR with GIE enabled
   STW    B0,*B15--     ; store B0 to stack (contains IRP)
   STW    B1,*B15--     ; store B1 to stack (contains IER)
   STW    A0,*B15--     ; store A0 to stack (original CSR)
* Beginning of ISR code
   ...
* End of ISR code

   LDW    *++B15,A0     ; restore A0 (original CSR)
   LDW    *++B15,B1     ; restore B1 (contains IER)
   LDW    *++B15,B0     ; restore B0 (contains IRP)
   LDW    *++B15,B4     ; restore B4
   LDW    *++B15,B3     ; restore B3
|| MVC    A0,CSR        ; restore original CSR
   LDW    *++B15,B2     ; restore B2
|| MVC    B0,IRP        ; restore original IRP
   LDW    *++B15,B1     ; restore B1
|| MVC    B1,IER        ; restore original IER
   LDW    *++B15,A0     ; restore A0
|| B      IRP           ; return from interrupt
   LDW    *++B15,B0     ; restore B0
   NOP    4             ; allow all multi–cycle instructions
                        ; to complete before branch is taken
```

Part III

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Appendix A

Applications Programming

This appendix provides extensive code examples from the Global Systems for Mobile Communications (GSM) enhanced full-rate (EFR) vocoder. The assembly code examples in this appendix represent hand-optimized code; the code produced by the assembly optimizer will vary, depending on the version used.

Topic **Page**

A.1 Summary of Major Programming Methods **A-2**

A.2 Implementation of GSM EFR Vocoder **A-3**

A.1 Summary of Major Programming Methods

The key to implementing applications on the 'C6x is to take advantage of the processor's full speed. The main technique for achieving this goal involves unrolling software loops to reach the limits of the functional units while meeting the data dependency constraints.

In addition to loop unrolling, the following methods are helpful for improving performance:

- Rearranging the C code

  If you are implementing a system based on an existing C code, rearranging the tasks in the C code is a useful method to gain better performance.

- Avoiding memory bank hits

  Memory bank hits, especially those in the inner loop in a nested loop application, hurt the performance dramatically and must be avoided. Most of the memory bank hits, however, can be eliminated by allocating the relevant arrays properly. Some situations, like accessing a word and a half-word in the same cycle, can also create the chance of a memory bank hit and should also be avoided.

If the system implementation is quite complicated, the program-memory size becomes an issue. To achieve a good balance between program-memory size and speed, you can implement the less critical portions with highly-compact assembly code that sacrifices performance.

A.2 Implementation of the GSM EFR Vocoder

This section presents the implementation of some representative pieces of code for the Global Systems for Mobile Communications (GSM) enhanced full-rate (EFR) vocoder. These include the:

- Multiply-accumulate loop
- Windowing and scaling part of autocorr.c
- cor_h
- rrv computation in search_10i40
- Index search in search_10i40
- FIR filter (residu.c)
- Lag search in the lag_max ( ) routine

Note:

European Telecommunications Standards Institute (ETSI) has the copyright to all the C code used in this section.

The following global constants/symbols are defined in the EFR vocoder:

```
#define Word16    short
#define Word32    int
#define MAX_32    0x7fffffffL
#define MIN_32    0x80000000L
#define MAX_16    0x7fff
#define MIN_16    0x8000
```

Part IV

A.2.1 Implementation of the Multiply-Accumulate Loop

First, examine the most popular loop used in almost every fixed-point vocoder, the multiply-accumulate (MAC) loop, shown in Example A–1.

Example A–1. C Code for the Typical MAC Loop

```
input:
         Word16  N;  (typical value of N is an even integer,
                     greater than or equal to 20)
         Word16  *x, *y;

result:
         Word32 sum;

C Code
------------------------------------------------------------------------------
         sum=0;
         for(i=0;i<N;i++) sum=L_mac(sum,x[i],y[i]);
------------------------------------------------------------------------------
where    L_mac(a,b,c) = _sadd(a,_smpy(b,c))
```

Example A–2 shows a list of symbolic instructions for each iteration of the loop.

Example A–2. Linear Assembly for the MAC Loop

```
LOOP:
       LDH    .D     *xptr++, xi       ;   load x[i]
       LDH    .D     *yptr++, yi       ;   load y[i]
       SMPY   .M     xi,yi,tmp         ;   smpy(x[i],y[i])
       SADD   .L     sum,tmp,sum       ;   sum=sadd(sum,smpy(x[i],y[i])
[cntr] SUB    .ALU   cntr,1,cntr       ;   decrement the loop counter
[cntr] B      .S     LOOP              ;   branch to the loop
```

In Example A–2, xptr is the pointer for the x array and yptr is the pointer for the y array. Because there are eight functional units, these instructions can easily fit into one execution packet.

In general, unrolling the loop once as in the code in Example A–3 does not give the same result as the code shown in Example A–1, because of the ordering dependence of the saturated addition.

Example A–3. C Code for MAC Loop With Loop Unrolling

```
Word32 sum_e, sum_o;
       sum_e=0;
       sum_o=0;
       for(i=0;i<N;i+=2) {
           sum_e=L_mac(sum_e,x[i],y[i]);
           sum_o=L_mac(sum_o,x[i+1],y[i+1]);
       }
       sum=L_add(sum_o,sum_e);
-------------------------------------------------------------------------------
where  L_add(a,b)=_sadd(a,b)
```

However, both approaches lead to the same result if x[i] = y[i] for every i, Because _smpy (x[i], x[i]) is always greater than or equal to 0. This special MAC loop is used to compute the energy of a particular signal segment. In this case, take the approach shown in Example A–3, because it doubles the performance of the code shown in Example A–2. Example A–4 shows the C code for this special MAC loop. Example A–5 lists the symbolic instructions for this loop.

Example A–4. C Code for Energy Computation MAC Loop

```
    sum=0;
    for(i=0;i<N;i++)
        sum = L_mac(sum,x[i],x[i]);

or
    sum_e=0;
    sum_o=0;
    for(i=0;i<N;i+=2) {
         sum_e=L_mac(sum_e,x[i],x[i]);
         sum_o=L_mac(sum_o,x[i+1],x[i+1]);
    }
    sum=L_add(sum_o,sum_e);
```

Example A–5. Linear Assembly for Energy Computation MAC Loop

```
LOOP:
       LDH    .D  *xptre++, xi            ; load x[i]
       SMPY   .M  xi,xi,tmp_e             ; smpy(x[i],x[i])
       SADD   .L  sum_e,tmp_e,sum_e       ; sum_e=sadd(sum_e,smpy(x[i],x[i])
       LDH    .D  *xptro++, xi+1          ; load x[i+1]
       SMPY   .M  xi+1,xi+1,tmp_o         ; smpy(x[i+1],x[i+1])
       SADD   .L  sum_o,tmp_o,sum_o       ; sum_o=sadd(sum_o,smpy(x[i+1],x[i+1])
[cntr] SUB    .S  cntr,2,cntr             ; decrement the loop counter
[cntr] B      .S  LOOP                    ; branch to the loop

       SADD   .L  sum_e, sum_o, sum       ; sum=sadd(sum_o+sum_e)
```

In Example A–5, xptre and xptro are the pointers for the x array and, at the beginning, point to x[0] and x[1], respectively. The eight instructions in the loop fit perfectly into one execution packet. This approach computes two MACs in one cycle. It doubles the performance of the code shown in Example A–2 for the general MAC loop.

The final assembly code is shown in Example A–6.

Example A–6. Assembly Code for the Energy Computation MAC Loop

```
******************************************************************************
**       Texas Instruments, Inc                                             **
**                                                                          **
**       MAC Loop -- Energy Computation                                     **
**                                                                          **
**       Compute two samples a time                                         **
**                                                                          **
**       Total cycles = (N/2+2)                                             **
**                                                                          **
**       Register Usage:        A               B                           **
**                              4               5                           **
**                                                                          **
**       Notice that x[0] and x[1] will not be available till LOOP          **
**       is executed once. Therefore, sum_e and sum_o should be 0s          **
**       for the first three iterations. This is why A5, B5, A6,            **
**       and B6 should be set to 0s in the prolog.                          **
******************************************************************************

                       ; A4 -- &x[0]
                       ; B4 -- N
                       ; A6 -- sum

         ADD    .L2X   A4,2,B4          ; &x[1]
||       SUB    .D2    B4,6,B1          ; loop counter
||       B      .S2    LOOP             ; branch to the loop
||       MVK    .S1    0,A6             ; initialize sum_e

         LDH    .D1    *A4++[2],A5      ; load x[0]
||       LDH    .D2    *B4++[2],B5      ; load x[1]
||       B      .S2    LOOP             ; branch to the loop
||       MV     .L2X   A6,B6            ; initialize sum_o

         LDH    .D1    *A4++[2],A5      ; load x[2]
||       LDH    .D2    *B4++[2],B5      ; load x[3]
||       B      .S1    LOOP             ; branch to the loop
||       MV     .L1    A6,A5            ; take care the initial three iterations
||       MV     .L2    B6,B5            ; take care the initial three iterations

         LDH    .D1    *A4++[2],A5      ; load x[4]
||       LDH    .D2    *B4++[2],B5      ; load x[5]
||       B      .S1    LOOP
```

Part IV

Example A–6. Assembly Code for the Energy Computation MAC Loop (Continued)

```
       LDH    .D1    *A4++[2],A5      ;   load x[6]
||     LDH    .D2    *B4++[2],B5      ;   load x[7]


LOOP:
       SMPY   .M1    A5,A5,A7         ;   smpy(x[i],x[i])
||     SMPY   .M2    B5,B5,B7         ;   smpy(x[i+1],x[i+1])
||     SADD   .L1    A7,A6,A6         ;   sum_e=sadd(sum_e,smpy(x[i],x[i]))
||     SADD   .L2    B7,B6,B6         ;   sum_o=sadd(sum_o,smpy(x[i+1],x[i+1]))
||     LDH    .D1    *A4++[2],A5      ;   load x[i]
||     LDH    .D2    *B4++[2],B5      ;   load x[i+1]
||[B1] B      .S1    LOOP             ;   branch to the loop
||[B1] SUB    .S2    B1,2,B1          ;   decrement loop counter

       SADD   .L1X   A6,B6,A6         ;   final result, sum = sum_e + sum_o
```

A.2.2 Implementation of the Windowing and Scaling Part of autocorr.c

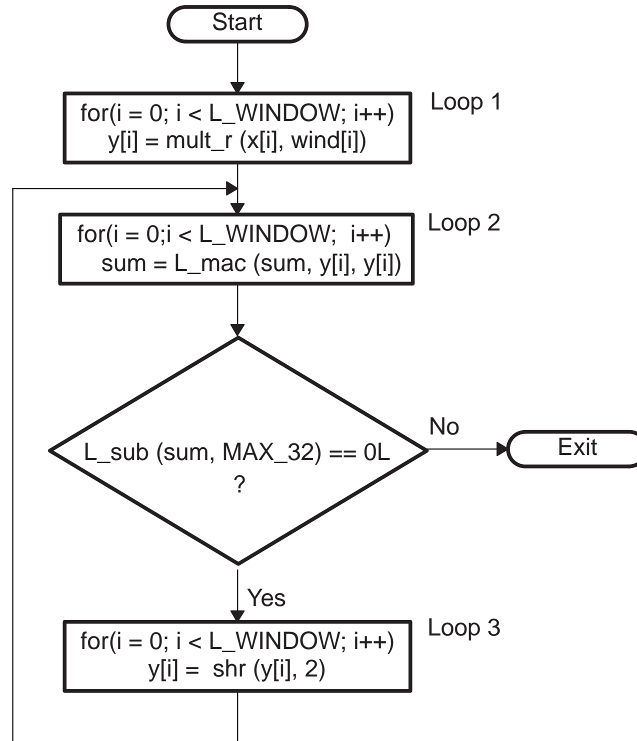
The autocorr.c routine is one of the most computationally intensive modules in the EFR vocoder. The part used in Example A–7 is used for windowing speech samples and for scaling down the windowed sample sequence if the input level is too high. Figure A–1 shows the flow diagram for this code.

Example A–7. C Code for the Windowing and Scaling Part of autocorr.c

```
#define L_WINDOW       240

input:
        Word16 x[L_WINDOW], wind[L_WINDOW];

local variables/arrays:
        Word16 i;
        Word16 y[L_WINDOW];
        Word32 sum;
        Word16 overfl, overfl_shft;

Original C code:
----------------------------------------------------------

        /* Windowing of signal */

for (i = 0; i < L_WINDOW; i++)
{
        y[i] = mult_r (x[i], wind[i]);
}

/* Compute r[0] and test for overflow */

overfl_shft = 0;

do
{
        overfl = 0;
        sum = 0L;

        for (i = 0; i < L_WINDOW; i++)
        {
                        sum = L_mac (sum, y[i], y[i]);
        }

        /* If overflow divide y[] by 4 */

        if (L_sub (sum, MAX_32) == 0L)
        {
                overfl_shft = add (overfl_shft, 4);
                overfl = 1;            /* Set the overflow flag */

                for (i = 0; i < L_WINDOW; i++)
                {
                        y[i] = shr (y[i], 2);
                }
         }
}
while (overfl != 0);
-------------------------------------------------------
Where   mult_r(a,b) = _sadd(_smpy(a,b),0x8000L)>>16
        L_mac(a,b,c)= _sadd(a,_smpy(b,c))
        L_sub(a,b) = _ssub(a,b)
        add(a,b) = ((_sadd((a)<<16,((b)<<16)))>>16)
        shr(a,b) = ((b)<0 ? (_sshl((a),(-b+16))>>16):((a)>>(b)))
```

Part IV

Figure A–1. Flow Diagram for the Windowing and Scaling Part of autocorr.c



A.2.2.1 Unrolling the Loop

Try the loop unrolling technique for each loop.

Example A–8 shows the list of symbolic instructions needed to execute one iteration of loop 1. You can use any arithmetic logic unit (ALU) for the loop-counter update.

Example A–8. Linear Assembly for One Iteration of autocorr.c (Loop 1)

```
LOOP1:
       LDH    .D     *windptr++,windi           ;load wind[i]
       LDH    .D     *xptr++,xi                 ;load x[i]
       SMPY   .M     windi,xi,windxi0           ;smpy(x[i],wind[i])
       SADD   .L     windxi0,0x8000L,wind1xi1   ;sadd(smpy(x[i],wind[i]),0x8000L)
       SHR    .S     windxi1,16,yi              ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
       STH    .D     yi,*yptr++                 ;store y[i]
[cntr] SUB    .ALU   cntr,1,cntr                ;decrement loop counter
[cntr] B      .S     LOOP1                      ;branch to loop
```

In Example A–8, windptr, xptr, and yptr are the pointers of wind, x, and y.

The .D unit is used most often (three times). With properly partitioned resources, this is a 2-cycle loop.

If you unroll the loop once and load both x and wind in words (in GSM EFR, both x and wind can be loaded in words if they are map-aligned with the word boundary), you can compute two y values with two cycles. The following is the new list of the instructions in one loop iteration.

Example A–9. Linear Assembly for Loop 1 of autocorr.c (Using LDW)

```
LOOP1:
        LDW    .D  *windptr++,windi_windi+1           ;load wind[i] and wind[i+1]
        LDW    .D  *xptr++,xi_xi+1                    ;load x[i] and x[i+1]
        SMPY   .M  windi_windi+1,xi_xi+1,windxi0      ;smpy(x[i],wind[i])
        SMPYH  .M  windi_windi+1,xi_xi+1,windxi0+1    ;smpy(x[i+1],wind[i+1])
        SADD   .L  windxi0,0x8000L,windxi1            ;sadd(smpy(x[i],wind[i]),0x8000L)
        SADD   .L  windxi0+1,0x8000L,windxi1+1        ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
        SHR    .S  windxi1,16,yi                      ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
        SHR    .S  windxi1+1,16,yi+1                  ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
        STH    .D  yi,*yptre++[2]                     ;store y[i]
        STH    .D  yi+1,*yptro++[2]                   ;store y[i+1]
[cntr]  SUB    .S  cntr,2,cntr                        ;decrement loop counter
[cntr]  B      .S  LOOP1                              ;branch to loop
```

In Example A–9, yptre and yptro are the pointers for the y array and, at the beginning, point to y[0] and y[1], respectively.

Note:

Loop 2 is a special MAC loop, as described in section A.2.1 on page A-4. It can be implemented either as shown in Example A–10 without loop unrolling or as in Example A–11 with loop unrolling for one iteration.

Example A–10. Linear Assembly for Loop 2 of autocorr.c (No Loop Unrolling)

```
LOOP2:
       LDH    .D     *yptr++,yi           ;load y[i]
       SMPY   .M     yi,yi,yyi            ;smpy(y[i],y[i])
       SADD   .L     sum,yyi,sum          ;sadd(sum,smpy(y[i],y[i]))
[cntr] SUB    .S     cntr,1,cntr          ;decrement loop counter
[cntr] B      .S     LOOP2                ;branch to loop
```

Example A–11. Linear Assembly for Loop 2 of autocorr.c (With Loop Unrolling)

```
LOOP2:
       LDH    .D     *yptre++,yi          ;load y[i]
       LDH    .D     *yptro++,yi+1        ;load y[i+1]
       SMPY   .M     yi,yi,yyi            ;smpy(y[i],y[i])
       SMPY   .M     yi+1,yi+1,yyi+1      ;smpy(y[i+1],y[i+1])
       SADD   .L     sum_e,yyi,sum_e      ;sadd(sum_e,smpy(y[i],y[i])
       SADD   .L     sum_o,yyi+1,sum_o    ;sadd(sum_o,smpy(y[i+1],y[i+1]))
[cntr] SUB    .S     cntr,2,cntr          ;decrement loop counter
[cntr] B      .S     LOOP2                ;branch to loop

       SADD   .L     sum_e,sum_o,sum      ;sum=sum_o+sum_e
```

Later, you will see that both approaches are used in this application.

Loop 3 is a single-cycle loop and you cannot speed it up by simply unrolling the loop. The instructions for each iteration are shown in Example A–12.

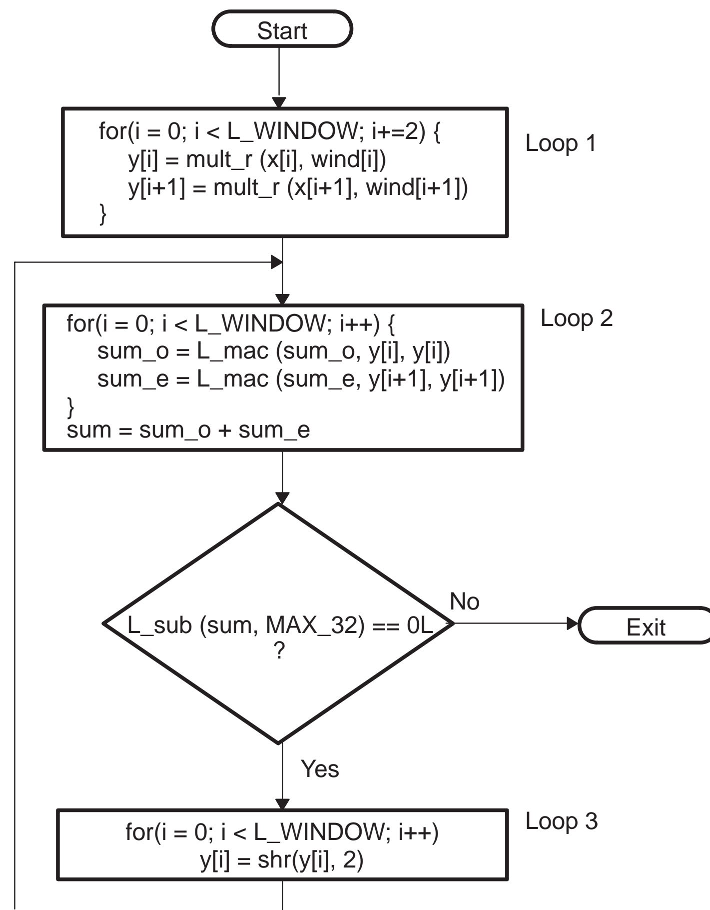
Example A–12. Linear Assembly for Loop 3 of autocorr.c

```
LOOP3:
       LDH    .D     *yptrl++,yi          ;load y[i]
       SHR    .S     yi,2,yi0             ;shr(y[i],2)
       STH    .D     yi0,*yptrs++         ;store y[i]=shr(y[i],2)
[cntr] SUB    .L     cntr,1,cntr          ;decrement loop counter
[cntr] B      .S     LOOP3                ;branch to loop
```

In Example A–12, yptrl is the pointer for loading the y array and yptrs is the pointer for storing the y array.

The new flow diagram is shown in Figure A–2.

Figure A–2. Flow Diagram for autocorr.c With Loop Unrolling



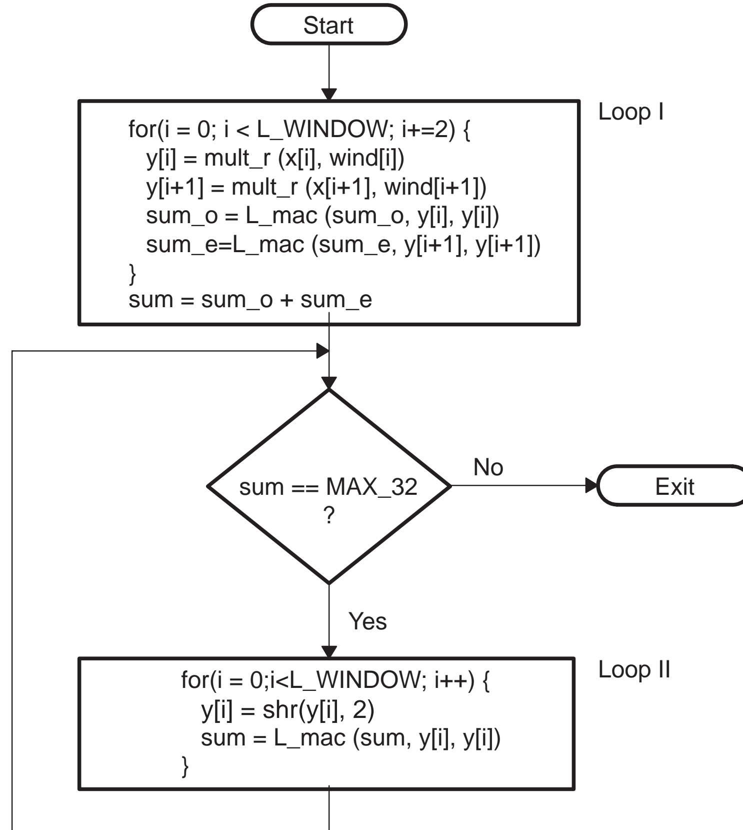
A.2.2.2 Rearrange the C Code

The first execution of loop 2 can be combined with loop 1 to form a new loop I and its subsequent executions can be combined with loop 3 to form a new loop II.

Another small change is the implementation of if L_sub(sum, MAX_32) == 0L as if sum == MAX_32.

The new flow diagram with rearranged C code is shown in Figure A–3.

Figure A–3. Flow Diagram for autocorr.c With Rearranged C Code



You can implement loop I as one of the two approaches as shown in Example A–13.

Example A–13. Linear Assembly for Loop I of autocorr.c (Modified)

```
LOOPI:
        LDW    .D  *windptr++,windi_windi+1          ;load wind[i] and wind[i+1]
        LDW    .D  *xptr++,xi_xi+1                   ;load x[i] and x[i+1]
        SMPY   .M  windi_windi+1,xi_xi+1,windxi0     ;smpy(x[i],wind[i])
        SMPYH  .M  windi_windi+1,xi_xi+1,windxi0+1   ;smpy(x[i+1],wind[i+1])
        SADD   .L  windxi0,0x8000L,windxi1           ;sadd(smpy(x[i],wind[i]),0x8000L)
        SADD   .L  windxi0+1,0x8000L,windxi1+1       ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
        SHR    .S  windxi1,16,yi                     ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
        SHR    .S  windxi1+1,16,yi+1                 ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
        SMPY   .M  yi,yi,yyi                         ;smpy(y[i],y[i])
        SMPY   .M  yi+1,yi+1,yyi+1                   ;smpy(y[i+1],y[i+1])
        SADD   .L  sum_e,yyi,sum_e                   ;sum_e=sadd(sum_e,smpy(y[i],y[i]))
        SADD   .L  sum_o,yyi+1,sum_o                 ;sum_o=sadd(sum_o,smpy(y[i+1],y[i+1])
        STD    .D  yi,*yptre++[2]                    ;store y[i]
        STD    .D  yi+1,*yptro++[2]                  ;store y[i+1]
[cntr]  SUB    .S  cntr,2,cntr                       ;decrement loop counter
[cntr]  B      .S  LOOPI                             ;branch to loop
```

or as

```
LOOPI:
        LDW    .D  *windptr++,windi_windi+1          ;load wind[i] and wind[i+1]
        LDW    .D  *xptr++,xi_xi+1                   ;load x[i] and x[i+1]
        SMPY   .M  windi_windi+1,xi_xi+1,windxi0     ;smpy(x[i],wind[i])
        SMPYH  .M  windi_windi+1,xi_xi+1,windxi0+1   ;smpy(x[i+1],wind[i+1])
        SADD   .L  windxi0,0x8000L,windxi1           ;sadd(smpy(x[i],wind[i]),0x8000L)
        SADD   .L  windxi0+1,0x8000L,windxi1+1       ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
        SHR    .S  windxi1,16,yi                     ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
        SHR    .S  windxi1+1,16,yi+1                 ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
        SMPYH  .M  windxi1,windxi1,yyi               ;smpy(y[i],y[i])
        SMPYH  .M  windxi+1,windxi+1,yyi+1           ;smpy(y[i+1],y[i+1])
        SADD   .L  sum_e,yyi,sum_e                   ;sum_e=sadd(sum_e,smpy(y[i],y[i]))
        SADD   .L  sum_o,yyi+1,sum_o                 ;sum_o=sadd(sum_o,smpy(y[i+1],y[i+1])
        STD    .D  yi,*yptre++[2]                    ;store y[i]
        STD    .D  yi+1,*yptro++[2]                  ;store y[i+1]
[cntr]  SUB    .S  cntr,2,cntr                       ;decrement loop counter
[cntr]  B      .S  LOOPI                             ;branch to loop
```

The only difference between these two implementations is how to compute yyi and yyi + 1. Using yyi as an example, the former approach computes yyi following the order of the original C code:

```
yyi = _smpy(_sadd(_smpy(a,b),0x8000L)>>16,
      _sadd(_smpy(a,b),0x8000L)>>16),
```

The latter computes yyi in a slightly different way as:

```
yyi = _smpyh(_sadd(_smpy(a,b),0x8000L),
      _sadd(_smpy(a,b),0x8000L)).
```

This provides the flexibility to better pack the instructions and reduces cycle count.

Loop I is a two-cycle loop. Loop II is still a single-cycle loop. Its instructions are shown in Example A–14.

Example A–14. Linear Assembly for Loop II of autocorr.c (Modified)

```
LOOPII:
       LDH    .D     *yptrl++,yi   ;load y[i]
       SHR    .S     yi,2,yi0      ;shr(y[i],2)
       SMPY   .M     yi0,yi0,yyi   ;smpy(shr(y[i],2),shr(y[i],2))
       SADD   .L     sum,yyi,sum   ;sum=sadd(sum,smpy(shr(y[i],2),shr(y[i],2)))
       STH    .D     yi0,*yptrs++ ;store y[i]=shr(y[i],2)
[cntr] SUB    .L     cntr,1,cntr   ;decrement loop counter
[cntr] B      .S     LOOPII        ;branch to loop
```

A.2.2.3 Memory Bank Hits

To schedule loop I as a 2-cycle loop:

- x[i] + x[i + 1] << 16 and wind[i] + wind[i + 1] << 16 must be loaded in the same cycle.
- y[i] and y[i+1] must be stored in the other cycle.

To avoid a memory bank hit:

- Allocate x and wind in different memory spaces, if possible. For instance, allocate wind[i] in data ROM and x in data RAM.
- If no data ROM is available, allocate x and wind so they are offset from each other by one word.

There is no memory bank problem when storing y[i] and y[i + 1].

No memory bank hits occur in loop II, because the distance between the load and store is always six halfwords.

The modified C code of this part of autocorr.c is shown in Example A–15.

Example A–15. Implemented C Code for autocorr.c

```
Word16 i;
Word16 y[L_WINDOW];
Word32 sum, sum_e, sum_o;
Word16 overfl, overfl_shft;

/* Windowing of signal */

sum_e=sum_o=0L;
for (i = 0; i < L_WINDOW; i+=2)
{
       y[i] = mult_r (x[i], wind[i]);
       y[i+1] = mult_r(x[i+1], window[i+1]);
       sum_e = L_mac(sum_e, y[i], y[i]);
       sum_o = L_mac(sum_o, y[i+1], y[i+1]);
}
sum=sum_e+sum_o;

/* Compute r[0] and test for overflow */

overfl_shft = 0;

do
{
       overfl = 0;
       /* If overflow divide y[] by 4 */

       if (sum == MAX_32)
       {
              overfl_shft = add (overfl_shft, 4);
              overfl = 1;                  /* Set the overflow flag */

              sum=0L;
              for (i = 0; i < L_WINDOW; i++)
              {
                     y[i] = shr (y[i], 2);
                     sum = L_mac(sum, y[i], y[i]);
              }
       }
}
while (overfl != 0);
```


A.2.2.4 Code-Size Reduction

Finally, consider the code-size reduction. In Figure A–3 on page A-13, loop I is always executed and loop II is executed only for high-input levels. This means that cycle count is the most important factor for loop I, while code size is more critical for loop II.

A.2.2.5 Final Assembly Code

The final assembly code is presented in Example A–16.

Example A–16. Assembly Code for Windowing and Scaling Part of autocorr.c

```
************************************************************************************
**     Texas Instruments, Inc                                                     **
**                                                                                **
**     Implementation of The Windowing and Scaling Part of autocorr.c             **
**     In EFR                                                                     **
**                                                                                **
**     Compute two samples a time                                                 **
**                                                                                **
**     Total cycles = 257 (No Scaling)                                            **
**                  = 519 (One Scaling)                                           **
**                                                                                **
**     Register Usage:         A               B                                  **
**                             11              9                                  **
**                                                                                **
************************************************************************************
       ; B4 -- &x[0]
       ; A4 -- &window[0]
       ; A6 -- &y[0]
       ; B8 -- L_WINDOW
       ; A0 -- sum and sum_e
       ; B0 -- sum_o
       ; B15 -- stack pointer


       ; notice that we use the latter approach in Example A–13

       LDW    .D2    *B4++,B5      ; load x[0] & x[1]
||     LDW    .D1    *A4++,A5      ; load wind[0] & wind[1]
||     MVK    .S1    480,A6        ; reserve space for y[i]
||     SUB    .S2    B8,6,B1       ; LOOP I counter

       SUB    .L1X   B15,A6,A6     ; &y[0]
||     MVK    .S2    1,B7

       LDW    .D2    *B4++,B5      ; load x[2] & x[3]
||     LDW    .D1    *A4++,A5      ; load wind[2] & wind[3]

       SHL    .S2    B7,15,B7      ; 32768 or 0x8000L for rounding
||     MVK    .S1    -1,A10
||     ADD    .L2X   A6,2,B6       ; &y[1]
||     MV     .L1    A6,A3         ; &y[0]

       LDW    .D2    *B4++,B5      ; load x[4] & x[5]
||     LDW    .D1    *A4++,A5      ; load wind[4] & wind[5]
||     MVKLH  .S1    32767,A10     ; 7fffffff = MAX_32
||     MV     .L1X   B7,A7         ; 32768
```

Part IV

Example A–16. Assembly Code for Windowing and Scaling Part of autocorr.c (Continued)

```
       SMPYH  .M2X   B5,A5,B2       ; smpy(x[1],wind[1])
||     SMPY   .M1X   B5,A5,A2       ; smpy(x[0],wind[0])
||     B      .S2    LOOPI

       LDW    .D2    *B4++,B5       ; load x[6] & x[7]
||     LDW    .D1    *A4++,A5       ; load wind[6] & wind[7]
||     MVK    .S1    0,A0           ; sum_o = 0
||     MVK    .S2    0,B0           ; sum_e = 0

       SMPYH  .M2X   B5,A5,B2       ; smpy(x[3],wind[3])
||     SMPY   .M1X   B5,A5,A2       ; smpy(x[2],wind[2])
||     SADD   .L1    A2,A7,A2       ; sadd(smpy(x[1],wind[1]),0x8000L)
||     SADD   .L2    B2,B7,B2       ; sadd(smpy(x[0],wind[0]),0x8000L)
||     B      .S1    LOOPI

       LDW    .D2    *B4++,B5       ; load x[8] & x[9]
||     LDW    .D1    *A4++,A5       ; load wind[8] & wind[9]
||     SHR    .S1    A2,16,A9       ; y[1]=sadd(smpy(x[1],wind[1]),0x8000L)>>16
||     SHR    .S2    B2,16,B9       ; y[0]=sadd(smpy(x[0],wind[0])+0x8000L)>>16
||     SMPYH  .M1    A2,A2,A11      ; smpy(y[0],y[0])
||     SMPYH  .M2    B2,B2,B11      ; smpy(y[1],y[1])

LOOPI:

       STH    .D1    A9,*A6++[2]    ; store y[1]
||     STH    .D2    B9,*B6++[2]    ; store y[0]
||     SADD   .L1    A2,A7,A2       ; sadd(smpy(x[3],wind[3]),0x8000L)
||     SADD   .L2    B2,B7,B2       ; sadd(smpy(x[2],wind[2]),0x8000L)
||     SMPYH  .M2X   B5,A5,B2       ; smpy(x[5],wind[5])
||     SMPY   .M1X   B5,A5,A2       ; smpy(x[4],wind[4])
||[B1] SUB    .S2    B1,2,B1        ; decrement the loop counter
||[B1] B      .S1    LOOPI

       SADD   .L1    A0,A11,A0      ; sum_e += smpy(y[0],y[0])
||     SADD   .L2    B0,B11,B0      ; sum_o += smpy(y[1],y[1])
||     SMPYH  .M1    A2,A2,A11      ; smpy(y[2],y[2])
||     SMPYH  .M2    B2,B2,B11      ; smpy(y[3],y[3])
||     SHR    .S1    A2,16,A9       ; y[3]=sadd(smpy(x[3],wind[3]),0x8000L)>>16
||     SHR    .S2    B2,16,B9       ; y[2]=sadd(smpy(x[2],wind[2]),0x8000L)>>16
||     LDW    .D2    *B4++,B5       ; load x[10] & x[11]
||     LDW    .D1    *A4++,A5       ; load wind[10] & wind[11]

       SADD   .L1X   A0,B0,A0       ; sum = sum_e + sum_o
||     MPY    .M2    B0,0,B0        ; overfl_shift = 0
                                    ; LOOP I completed
```

Part IV

Example A–16. Assembly Code for Windowing and Scaling Part of autocorr.c (Continued)

```
LTEST:

       CMPEQ  .L1    A0,A10,A1     ; if (sum == MAX_32)

 [!A1] B      .S1    FINISH        ; No, exit
||[A1] LDH    .D1    *A3,B5        ; load y[0]
||[A1] ADD    .L2X   A3,2,B9       ; &y[1]
||[A1] ADD    .D2    B0,4,B0       ; add (overfl_shift,4)

  [A1] LDH    .D2    *B9++,B5      ; load y[1]
||[A1] SUB    .S2    B8,7,B1       ; counter for LOOPII
||[A1] B      .S1    LOOP II
||[A1] MV     .L1    A3,A9         ; &y[0]
  [A1] LDH    .D2    *B9++,B5      ; load y[2]
||[A1] MVK    .S1    0,A0          ; sum = 0
||[A1] B      .S2    LOOPII

  [A1] LDH    .D2    *B9++,B5      ; load y[3]
||[A1] B      .S1    LOOPII
||[A1] MV     .L1    A0,A2         ; to take care of the initial condition

  [A1] LDH    .D2    *B9++,B5      ; load y[4]
||[A1] B      .S1    LOOPII

  [A1] LDH    .D2    *B9++,B5      ; load y[5]
||[A1] SHR    .S1X   B5,2,A5       ; y[0]=shr(y[0],2)
||[A1] B      .S2    LOOPII

LOOPII:

       LDH    .D2    *B9++,B5      ; load y[6]
||     SHR    .S1X   B5,2,A5       ; y[1] = shr(y[1],2)
||[B1] B      .S2    LOOPII        ; branch
||     STH    .D1    A5,*A9++      ; store y[0]
||[B1] ADD    .L2    B1,-1,B1      ; decrement LOOPII counter
||     SMPY   .M1    A5,A5,A2      ; smpy(y[0],y[0])
||     SADD   .L1    A2,A0,A0      ; sum +=smpy(y[i],y[i])

       STH    .D1    A5,*A9++      ; store y[n-1]
||     SMPY   .M1    A5,A5,A2      ; smpy(y[n-1],y[n-1])
||     SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-3],y[n-3])
||     B      .S2    LTEST         ; branch back to LTEST

       SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-2],y[n-2])

       SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-1],y[n-1])

       NOP    3                    ; save the code size

FINISH:
```

If code size is not an issue, you can eliminate the last three NOPs by expanding the epilog of loop II. This saves three cycle counts every time loop II executes; however, code size increases by two fetch packets (2 × 32 = 64 bytes).

A.2.3 Implementation of cor_h

The cor_h routine is the second most computationally intensive routine called to compute the matrix of autocorrelation, rr. The core part of cor_h is presented in Example A–17.

Example A–17. C Code for cor_h

```
#define L_CODE  40

input:
       Word16 sign[L_CODE], h[L_CODE];


output:
       Word16 rr[L_CODE][L_CODE];



local variables/arrays:
       Word16 h2[L_CODE]; /* function of h, the impulse response of weighted
                              synthesis filter */
       Word16 dec, j, i, k;
       Word32 s;



Original C code
------------------------------------------------------
       for (dec=1; dec<L_CODE; dec++)
       {
              s = 0;
              j = L_CODE-1;
              i = sub(j, dec);
              for (k=0; k<(L_CODE-dec); k++, i--, j--)
              {
                     s = L_mac(s, h2[k], h2[k+dec]);
                     rr[j][i] = mult(round(s), mult(sign[i],sign[j]));
                     rr[i][j] = rr[j][i];
              }
       }
-------------------------------------------------------
where  sub(a,b) = _ssub(a<<16, b<<16)>>16
       L_mac(a,b,c) = _sadd(a,_smpy(b,c))
       mult(a,b) = _smpy(a,b)>>16
and    round(a) = _sadd(a,0x8000L)>>16
```

The instructions to execute one iteration of the inner loop are listed in Example A–18.

Example A–18. Linear Assembly for cor_h (One Inner Loop Iteration)

```
INNERLOOP:
  LDH       .D      *h2ptr++,h2k          ;load h2[k]
  LDH       .D      *h2decptr++,h2deck    ;load h2[k+dec]
  SMPY      .M      h2k,h2deck,h2kk       ;smpy(h2[k],h2[k+dec])
  SADD      .L      s,h2kk,s              ;sadd(s,smpy(h2[k],h2[k+dec])
  SADD      .L      s,0x8000L,sround      ;round(s)<<16
  LDH       .D      *signiptr––,signi     ;load sign[i]
  LDH       .D      *signjptr––,signj     ;load sign[j]
  SMPY      .M      signi,signj,signij    ;smpy(sign[i],sign[j])=mult(sign[i],sign[j])<<16
  SMPYH     .M      signij,sround,rrji0   ;L_mult(round(s),mult(sign[i],sign[j]))
  SHR       .S      rrji0,16,rrji         ;rr[j][i]
  STH       .D      rrji,*rrjiptr––[41]   ;store rr[j][i]
  STH       .D      rrji,*rrijptr––[41]   ;store rr[i][j]
[icntr] SUB.ALU     icntr,1,icntr         ;decrement inner loop counter
[icntr] B   .S      INNERLOOP             ;branch to inner loop
```

In Example A–18, h2ptr and h2decptr are the pointers for h2, pointing to h2[k] and h2[k+dec]. The pointers for sign, signiptr and signjptr, point to sign[i] and sign[j]. The pointers for rr, rrjiptr and rrijptr, point to rr[j][i] and rr[i][j], respectively.

Notice that each element rr[j][i] is implemented as:

rr[j][i] = (_smpyh (_sadd (s, 0x8000L), _smpy (sign[i], sign[j]))) >> 16

The .D unit is used most often (six times in the inner loop). Ideally, these instructions can be arranged in three cycles. However, memory bank hits occur with any combination of the load and/or store instructions.

Next, consider unrolling the inner loop once. The C code is shown in Example A–19.

Example A–19. C Code for cor_h (With Inner Loop Unrolling)

```
for (dec=1; dec<L_CODE; dec++)
{
       s = 0;
       j = L_CODE-1;
       i = sub(j, dec);
       for (k=0; k<(L_CODE-dec); k+=2, i-=2, j-=2)
       {
              s = L_mac(s, h2[k], h2[k+dec]);
              rr[j][i] = mult(round(s), mult(sign[i],sign[j]));
              rr[i][j] = rr[j][i];
              s = L_mac(s, h2[k+1], h2[k+1+dec]);
              rr[j-1][i-1] = mult(round(s), mult(sign[i-1],sign[j-1]));
              rr[i-1][j-1] = rr[j-1][i-1];
       }
       if((dec&1)!=0) {
              s = L_mac(s,h2[L_CODE-dec-1],h2[L_CODE-1]);
              rr[dec][0] = mult(round(s),mult(sign[0],sign[dec]));
              rr[0][dec] = rr[dec][0];
       }
}
```

Eight values must be loaded and four values must be stored in every iteration; however, h2[k] and h2[k + 1] can be loaded in a word. The same is true for sign[j] and sign[j – 1]. A total of six loads are required. The inner loop instructions are shown in Example A–20.

Part IV

Example A–20. Linear Assembly for cor_h (With Inner Loop Unrolling)

```
INNER LOOP:
  LDW     .D   *h2ptr++,h2k_h2k+1          ;load h2[k] and h2[k+1]
  LDH     .D   *h2decptr++,h2deck          ;load h2[k+dec]
  SMPY    .M   h2k_h2k+1,h2deck,h2kk0      ;smpy(h2[k],h2[k+dec])
  SADD    .L   s,h2kk0,s                   ;sadd(s,smpy(h2[k],h2[k+dec])
  SADD    .L   s,0x8000L,sround            ;round(s)<<16
  LDH     .D   *signiptr––,signi           ;load sign[i]
  LDW     .D   *signjptr––,signj_signj–1   ;load sign[j] and sign[j–1]
  SMPYLH  .M   signi,signj_signj–1,signij0 ;smpy(sign[i],sign[j])
  SMPYH   .M   signij0,sround,rrji0        ;L_mult(round(s),mult(sign[i],sign[j]))
  SHR     .S   rrji0,16,rrji               ;rr[j][i]
  STH     .D   rrji,*rrjiptr––[82]         ;store rr[j][i]
  STH     .D   rrji,*rrijptr––[82]         ;store rr[i][j]
  LDH     .D   *h2decptr++,h2deck+1        ;load h2[k+1+dec]
  SMPYHL  .M   h2k_h2k+1,h2deck+1,h2kk1    ;smpy(h2[k+1],h2[k+1+dec])
  SADD    .L   s,h2kk1,s                   ;sadd(s,smpy(h2[k+1],h2[k+1+dec])
  SADD    .L   s,0x8000L,sround            ;round(s)<<16
  LDH     .D   *signiptr––,signi–1         ;load sign[i–1]
  SMPY    .M   signi–1,signj_signj–1,signij1;smpy(sign[i–1],sign[j–1])
  SMPYH   .M   signij1,sround,rrji1        ;L_mult(round(s),mult(sign[i–1],sign[j–1]))
  SHR     .S   rrji1,16,rrj1i1             ;rr[j–1][i–1]
  STH     .D   rrj1i1,*rrj1i1ptr––[82]     ;store rr[j–1][i–1]
  STH     .D   rrj1i1,*rri1j1ptr––[82]     ;store rr[i–1][j–1]
[icntr]SUB.ALUicntr,2,icntr                ;decrement inner loop counter
[icntr]B  .S   INNERLOOP                   ;branch to loop
```

To avoid memory bank hits:

- Load words (h2[k], h2[k+1]) and (sign[i–1], sign[i]) together and allocate h2 and sign so that they are aligned with each other.
- Store rr[j][i] and rr[j–1][i–1] together and rr[i][j] and rr[i–1][j–1] together.

There are five load/store pairs, so each iteration requires only five cycles. You gain speed by eliminating both the memory bank hits, as well as by reducing the cycles required to complete each rr.

The final assembly code with reduced code size is shown in Example A–21. Here, the primitive technique introduced in section 6.4.3.4, *Priming the Loop*, on page 6-47 is used to reduce the code size for both the prolog and epilog of the inner loop.

Part IV

Example A–21. Assembly Code for cor_h With Reduced Code Size

```
*********************************************************************************
**      Texas Instruments, Inc                                                 **
**                                                                             **
**      Implementation of cor_h in EFR                                         **
**                                                                             **
**      Compute four rrs at a time                                             **
**                                                                             **
**      Total cycles = 2533                                                    **
**                                                                             **
**      Register Usage:             A              B                           **
**                                                                             **
**                                 16             15                           **
**                                                                             **
*********************************************************************************

                                   ; A4 --- L_CODE
                                   ; B4 --- &h2[0]
                                   ; A6 --- &sign[0]
                                   ; B6 --- &rr[0][0]


       SUB     .L1     A4,1,A13      ; used to obtain &rr[i][j] and &rr[i-1][j-1]
||     ADDK    .S1     76,A6         ; &sign[L_CODE-2]
||     ADDK    .S2     3360,B6       ; &rr[L_CODE-1][L_CODE-2]+[82]=&rr[j][i]+[82]
||     SUB     .D1     A4,1,A2       ; outer loop counter

       MVK     .S2     0,B2          ; not doing the initial store
||     ADD     .L2     B4,2,B13      ; &h2[k+dec]
||     MVK     .S1     2,A11         ; used to increase/decrease the pointers
                                     ; for h2 and sign


OUTERLOOP:

       LDW     .D1     *A6,A10       ; load sign[j-1] & sign[j]
||     LDW     .D2     *B4,B12       ; load h2[k] & h2[k+1]
||     ADD     .L1X    B13,2,A3      ; &h2[k+dec+1]
||     SUB     .S1     A6,A11,A4     ; &sign[i-1]
||[A2] ADD     .L2X    A2,2,B0       ; define the inner loop counter
||     MPY     .M1     A13,A11,A3
||     MPY     .M2     B11,0,B11     ; initialize s

       LDH     .D2     *B13++[2],A7  ; load h2[k+dec]
||     LDH     .D1     *A3,B7        ; load h2[k+dec+1]
||     ADD     .L2X    A4,2,B9       ; &sign[i]
||     MV      .S2     B6,B14        ; &rr[j][i]+[82]
||     SUB     .L1     A6,4,A8       ; &sign[j-3]
||[B2] ADDK    .S1     -164,A14      ; from &rr[dec][0]+[82] to &rr[dec][0]
```

Part IV

Example A–21. Assembly Code for cor_h With Reduced Code Size (Continued)

```
        LDH     .D2   *B9,A0         ; load sign[i]
||      LDH     .D1   *A4--[2],B5    ; load sign[i-1]
||      ADD     .L2   B4,4,B8        ; &h2[k+2]
||      ADD     .L1   A11,2,A11      ; update A11
||      ADDK    .S2   -82,B14        ; &rr[j][i]
||      MVK     .S1   3,A1           ; determine when the stores in the inner loop
                                     ; actually starts

 [B2]   STH     .D1   A12,*A14       ; store rr[dec][0]
||[B2]  ADDK    .S1   -164,A9        ; from &rr[0][dec]+[82] tp &rr[0][dec]
||      MV      .L1X  B6,A14         ; &rr[j][i]+[82]
||      SHR     .S2   B0,1,B0        ; inner loop counter
||      SUB     .L2X  B14,A3,B3      ; &rr[i-1][j-1]
||      SUB     .D2   B6,2,B6        ; &rr[j][i-1], for the next
                                     ; outer loop iteration
 [B0]   B       .S2   INNERLOOP
||[A2]  SUB     .S1   A2,1,A2        ; decrement outer loop counter
||[A2]  AND     .L2X  A2,1,B2        ; decide if the last store is needed
||[B2]  STH     .D1   A12,*A9        ; store rr[0][dec]
||      SUB     .L1   A14,A3,A9      ; &rr[i][j]+[82]
||[B0]  MV      .D2   B0,B1          ; counter for branching to outer loop

INNERLOOP:

        SHR     .S2   B9,16,B10      ; ## obtain rr[j-1][i-1]
||      SMPYH   .M1   A3,A0,A3       ; # smpyh(sadd(s,0x8000L),smpy(sign[i],sign[j]))
||      SADD    .L2X  B11,A15,B9     ; # sadd(s,0x8000L)

||      LDW     .D1   *A8--,A10      ; *load sign[j] & sign[j-1]
||      LDW     .D2   *B8++,B12      ; *load h2[k] & h2[k+1]
||[!B1] B       .S1   OUTERLOOP      ; outer LOOP
||      ADD     .L1X  B13,2,A3       ; &h2[k+dec+1]

        LDH     .D1   *A3,B7         ; *h2[k+dec+1]
||      LDH     .D2   *B13++[2],A7   ; *h2[k+dec]
||      SMPYH   .M2   B9,B5,B9       ; # smpyh(sadd(s,0x8000L),smpy(sign[i-1],sign[j-1]))
||      SMPY    .M1X  A7,B12,A7      ; smpy(h2[k],h2[k+dec]
||      SUB     .S2   B1,1,B1        ; decrement the counter for branching to the outer loop
||[A1]  SUB     .L1   A1,1,A1        ; decrement the inner loop
||      ADD     .L2X  A4,2,B9        ; &sign[i]

        LDH     .D2   *B9,A0         ; *load sign[i]
||      LDH     .D1   *A4--[2],B5    ; *load sign[i-1]
||      SMPYHL  .M1   A10,A0,A0      ; smpy(sign[j],sign[i])
||      SMPYLH  .M2   B7,B12,B7      ; smpy(h2[k+1],h2[k+1+dec])
||[!A1] ADDK    .S1   -164,A14       ; ## from &rr[j][i]+[82] to &rr[j][i]
||[!A1] ADDK    .S2   -164,B14       ; ## from &rr[j-1][i-1]+[82] to &rr[j-1][i-1]
```

Part IV

Example A–21. Assembly Code for cor_h With Reduced Code Size (Continued)

```
 [!A1]   STH    .D1    A12,*A14      ; ## store rr[j][i]
||[!A1]  STH    .D2    B10,*B14      ; ## store rr[j-1][i-1]
||       SADD   .L1X   B11,A7,A5     ; s = sadd(s,smpy(h2[k],h2[k+dec])
||       SMPY   .M2X   A10,B5,B5     ; smpy(sign[i-1],sign[j-1]
||[B0]   SUB    .L2    B0,1,B0       ; decrement inner loop counter
||[!A1]  ADDK   .S1    -164,A9       ; ## from &rr[i][j]+[82] to rr[i][j]
||[!A1]  ADDK   .S2    -164,B3       ; ## from &rr[i-1][j-1]+[82] to &rr[i-1][j-1]

[!A1]    STH    .D1    A12,*A9       ; ## store rr[i][j]
||[!A1]  STH    .D2    B10,*B3       ; ## store rr[j-1][i-1]
||       SHR    .S1    A3,16,A12     ; # obtain rr[j-1][i-1]
||       SADD   .L1    A5,A15,A3     ; sadd(s,0x8000L)
||       SADD   .L2X   A5,B7,B11     ; s = sadd(s,smpy(h2[k+1],h2[k+dec+1]
||[B0]   B      .S2    INNERLOOP     ; end of INNERLOOP

         ADD    .L2X   B4,A11,B13    ; &h2[k+dec]
||[!A2]  B      .S2    FINISH        ; exit

FINISH:

         NOP    5
```

The value of s is represented by both B11 and A5 to avoid two .L1 or two .L2 units occurring in the same execute packet. Due to the dependence on s, as well as the removal of memory bank hits, it takes 20 cycles for each iteration of the modified C code. The pound sign (#) in the comments indicates that, each time the outer loop enters the inner loop, this instruction is not executed (or that the result of this instruction is not useful) until the number of iterations denoted by # has occurred.

The code size is 11 fetch packets (352 bytes). Without applying the primitive technique, the code size will be at least four fetch packets more than the code shown in Example A–21.

You can squeeze the instruction

```
ADD    .L2X      B4,A11,B13   ; &h2[k+dec]
```

into the inner loop to save about 1.5% of the cycle counts, with an increase in program memory of one fetch packet.

A.2.4 Implementation of the rrv Computation in search_10i40

Example A–22 shows the implementation of the rrv computation in search_10i40.

Example A–22. C Code for the rrv Computation in search_10i40

```
#define L_CODE 40
#define STEP 5
#define _1_16 (Word16)(32768L/16)
#define _1_8  (Word16)(32768L/8)

input:
       Word16 rr[L_CODE][L_CODE], ipos[L_CODE];

local variables/arrays:
       Word16 rrv[L_CODE];
       Word16 i0,i1,i2,i3,i4,i5,i6,i7,i8,i9; /* defined on [0,L_CODE-1] */
       Word32 s;
```

(The values of i0, i1, i2, i3, i4, i5, i6, and i7 were obtained before entering this loop.)

```
Original C code
------------------------------------------------------
       for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
       {
              s = L_mult (rr[i9][i9], _1_16);
              s = L_mac (s, rr[i0][i9], _1_8);
              s = L_mac (s, rr[i1][i9], _1_8);
              s = L_mac (s, rr[i2][i9], _1_8);
              s = L_mac (s, rr[i3][i9], _1_8);
              s = L_mac (s, rr[i4][i9], _1_8);
              s = L_mac (s, rr[i5][i9], _1_8);
              s = L_mac (s, rr[i6][i9], _1_8);
              s = L_mac (s, rr[i7][i9], _1_8);
              rrv[i9] = round (s);
       }
-------------------------------------------------------
       where  L_mult(a,b) = _smpy(a,b)
              L_mac(a,b,c) = _sadd(a,_smpy(b,c))
       and    round(a) = _sadd(a,0x8000L)>>16
```

The instructions for one loop iteration are shown in Example A–23.

Example A–23. Linear Assembly for the rrv Computation in Search_10i40 (One Loop Iteration)

```
LOOP:
       LDH    .D     *rr9ptr++[205],rr99 ;load rr[i9][i9]
       SMPY   .M     rr99,_1_16,s        ;s=L_mult(rr[i9][i9],_1_16)
       LDH    .D     *rr0ptr++[5],rr09   ;load rr[i0][i9]
       SMPY   .M     rr09,_1_8,s0        ;L_mult(rr[i0][i9],_1_8)
       SADD   .L     s,s0,s              ;s=L_mac(s,rr[i0][i9],_1_8)
       LDH    .D     *rr1ptr++[5],rr19   ;load rr[i1][i9]
       SMPY   .M     rr19,_1_8,s1        ;L_mult(rr[i1][i9],_1_8)
       SADD   .L     s,s1,s              ;s=L_mac(s,rr[i1][i9],_1_8)
       LDH    .D     *rr2ptr++[5],rr29   ;load rr[i2][i9]
       SMPY   .M     rr29,_1_8,s2        ;L_mult(rr[i2][i9],_1_8)
       SADD   .L     s,s2,s              ;s=L_mac(s,rr[i2][i9],_1_8)
       LDH    .D     *rr3ptr++[5],rr39   ;load rr[i3][i9]
       SMPY   .M     rr39,_1_8,s3        ;L_mult(rr[i3][i9],_1_8)
       SADD   .L     s,s3,s              ;s=L_mac(s,rr[i3][i9],_1_8)
       LDH    .D     *rr4ptr++[5],rr49   ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,s4        ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     s,s4,s              ;s=L_mac(s,rr[i4][i9],_1_8)
       LDH    .D     *rr5ptr++[5],rr59   ;load rr[i5][i9]
       SMPY   .M     rr59,_1_8,s5        ;L_mult(rr[i5][i9],_1_8)
       SADD   .L     s,s5,s              ;s=L_mac(s,rr[i5][i9],_1_8)
       LDH    .D     *rr6ptr++[5],rr69   ;load rr[i6][i9]
       SMPY   .M     rr69,_1_8,s6        ;L_mult(rr[i6][i9],_1_8)
       SADD   .L     s,s6,s              ;s=L_mac(s,rr[i6][i9],_1_8)
       LDH    .D     *rr7ptr++[5],rr79   ;load rr[i7][i9]
       SMPY   .M     rr79,_1_8,s7        ;L_mult(rr[i7][i9],_1_8)
       SADD   .L     s,s7,s              ;s=L_mac(s,rr[i7][i9],_1_8)
       SADD   .L     s,0x8000L,sround    ;round(s)
       SHR    .S     sround,16,rrv9      ;rrv[i9]
       STH    .D     rrv9,*rrv9ptr++[5]  ;store rrv[i9]
[icntr]SUB    .ALU   icntr,1,icntr       ;decrement inner loop counter
[icntr]B      .S     LOOP                ;branch to loop
```

Part IV

The following table shows the pointers In Example A–23 and the arrays they point to.

Pointer	for array
rr9ptr	rr[i9][i9]
rr0ptr	rr[i0][i9]
rr1ptr	rr[i1][i9]
rr2ptr	rr[i2][i9]
rr3ptr	rr[i3][i9]
rr4ptr	rr[i4][i9]
rr5ptr	rr[i5][i9]
rr6ptr	rr[i6][i9]
rr7ptr	rr[i7][i9]
rrv9ptr	rrv[i9]

The .D unit is used the most (ten times per iteration). Although these instructions can be arranged in five cycles, any combination of the load hits the same memory bank, Because any two values loaded are exactly 40 halfwords apart. It still takes ten cycles for one rrv.

Next, consider unrolling the inner loop once. The C code is shown in Example A–24.

Example A–24. C Code for the rrv Computation in search_10i40 (Unrolled Loop)

```
for (i9 = ipos[9]; i9 < L_CODE; i9 += 2*STEP)
{
     s = L_mult (rr[i9][i9], _1_16);
     S = L_mult (rr[i9+5][i9+5], _1_16);
     s = L_mac (s, rr[i0][i9], _1_8);
     S = L_mac (S, rr[i0][i9+5], _1_8);
     s = L_mac (s, rr[i1][i9], _1_8);
     S = L_mac (S, rr[i1][i9+5], _1_8);
     s = L_mac (s, rr[i2][i9], _1_8);
     S = L_mac (S, rr[i2][i9+5], _1_8);
     s = L_mac (s, rr[i3][i9], _1_8);
     S = L_mac (S, rr[i3][i9+5], _1_8);
     s = L_mac (s, rr[i4][i9], _1_8);
     S = L_mac (S, rr[i4][i9+5], _1_8);
     s = L_mac (s, rr[i5][i9], _1_8);
     S = L_mac (S, rr[i5][i9+5], _1_8);
     s = L_mac (s, rr[i6][i9], _1_8);
     S = L_mac (S, rr[i6][i9+5], _1_8);
     s = L_mac (s, rr[i7][i9], _1_8);
     S = L_mac (S, rr[i7][i9+5], _1_8);
     rrv[i9] = round (s);
     rrv[i9+5] = round (S);
}
```

Example A–25 shows the instructions for each iteration.

Example A–25. Linear Assembly for rrv Computation in search_10i40 (One Loop Iteration)

```
LOOP:
       LDH    .D     *rr9ptr++[410],rr99    ;load rr[i9][i9]
       SMPY   .M     rr99,_1_16,s           ;s=L_mult(rr[i9][i9],_1_16)
       LDH    .D     *rr95ptr++[410],rr995  ;load rr[i9+5][i9+5]
       SMPY   .M     rr995,_1_16,S          ;S=L_mult(rr[i9+5][i9+5],_1_16)
       LDH    .D     *rr0ptr++[10],rr09     ;load rr[i0][i9]
       SMPY   .M     rr09,_1_8,s0           ;L_mult(rr[i0][i9],_1_8)
       SADD   .L     s,s0,s                 ;s=L_mac(s,rr[i0][i9],_1_8)
       LDH    .D     *rr05ptr++[10],rr095   ;load rr[i0][i9+5]
       SMPY   .M     rr095,_1_8,S0          ;L_mult(rr[i0][i9+5],_1_8)
       SADD   .L     S,S0,S                 ;S=L_mac(S,rr[i0][i9+5],_1_8)
       LDH    .D     *rr1ptr++[10],rr19     ;load rr[i1][i9]
       SMPY   .M     rr19,_1_8,s1           ;L_mult(rr[i1][i9],_1_8)
       SADD   .L     s,s1,s                 ;s=L_mac(s,rr[i1][i9],_1_8)
       LDH    .D     *rr15ptr++[10],rr195   ;load rr[i1][i9+5]
       SMPY   .M     rr195,_1_8,S1          ;L_mult(rr[i1][i9+5],_1_8)
       SADD   .L     S,S1,S                 ;S=L_mac(S,rr[i1][i9+5],_1_8)
       LDH    .D     *rr2ptr++[10],rr29     ;load rr[i2][i9]
       SMPY   .M     rr29,_1_8,s2           ;L_mult(rr[i2][i9],_1_8)
       SADD   .L     s,s2,s                 ;s=L_mac(s,rr[i2][i9],_1_8)
       LDH    .D     *rr2ptr++[10],rr295    ;load rr[i2][i9+5]
       SMPY   .M     rr295,_1_8,S2          ;L_mult(rr[i2][i9+5],_1_8)
       SADD   .L     S,S2,S                 ;S=L_mac(S,rr[i2][i9+5],_1_8)
       LDH    .D     *rr3ptr++[10],rr39     ;load rr[i3][i9]
       SMPY   .M     rr39,_1_8,s3           ;L_mult(rr[i3][i9],_1_8)
       SADD   .L     s,s3,s                 ;s=L_mac(s,rr[i3][i9],_1_8)
       LDH    .D     *rr3ptr++[10],rr395    ;load rr[i3][i9+5]
       SMPY   .M     rr395,_1_8,S3          ;L_mult(rr[i3][i9+5],_1_8)
       SADD   .L     S,S3,S                 ;S=L_mac(S,rr[i3][i9+5],_1_8)
       LDH    .D     *rr4ptr++[10],rr49     ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,s4           ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     s,s4,s                 ;s=L_mac(s,rr[i4][i9],_1_8)
       LDH    .D     *rr4ptr++[10],rr49     ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,S4           ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     S,S4,S                 ;S=L_mac(S,rr[i4][i9],_1_8)
       LDH    .D     *rr5ptr++[10],rr59     ;load rr[i5][i9]
       SMPY   .M     rr59,_1_8,s5           ;L_mult(rr[i5][i9],_1_8)
       SADD   .L     s,s5,s                 ;s=L_mac(s,rr[i5][i9],_1_8)
       LDH    .D     *rr5ptr++[10],rr595    ;load rr[i5][i9+5]
       SMPY   .M     rr595,_1_8,S5          ;L_mult(rr[i5][i9+5],_1_8)
       SADD   .L     S,S5,S                 ;S=L_mac(S,rr[i5][i9+5],_1_8)
       LDH    .D     *rr6ptr++[10],rr69     ;load rr[i6][i9]
       SMPY   .M     rr69,_1_8,s6           ;L_mult(rr[i6][i9],_1_8)
```

Part IV

Example A–25. Linear Assembly for rrv Computation in search_10i40 (One Loop Iteration) (Continued)

```
        SADD   .L    s,s6,s                 ;s=L_mac(s,rr[i6][i9],_1_8)
        LDH    .D    *rr6ptr++[10],rr695    ;load rr[i6][i9+5]
        SMPY   .M    rr695,_1_8,S6          ;L_mult(rr[i6][i9+5],_1_8)
        SADD   .L    S,S6,S                 ;S=L_mac(S,rr[i6][i9+5],_1_8)
        LDH    .D    *rr7ptr++[10],rr79     ;load rr[i7][i9]
        SMPY   .M    rr79,_1_8,s7           ;L_mult(rr[i7][i9],_1_8)
        SADD   .L    s,s7,s                 ;s=L_mac(s,rr[i7][i9],_1_8)
        LDH    .D    *rr7ptr++[10],rr795    ;load rr[i7][i9+5]
        SMPY   .M    rr795,_1_8,S7          ;L_mult(rr[i7][i9+5],_1_8)
        SADD   .L    S,S7,S                 ;S=L_mac(S,rr[i7][i9+5],_1_8)
        SADD   .L    s,0x8000L,sround       ;round(s)
        SHR    .S    sround,16,rrv9         ;rrv[i9]
        STH    .D    rrv9,*rrv9ptr++[10]    ;store rrv[i9]
        SADD   .L    S,0x8000L,Sround       ;round(S)
        SHR    .S    Sround,16,rrv95        ;rrv[i9+5]
        STH    .D    rrv95,*rrv95ptr++[10]  ;store rrv[i9+5]
 [icntr]SUB    .ALU  icntr,2,icntr          ;decrement inner loop counter
 [icntr]B      .S    INNERLOOP              ;branch to loop
```

The following table shows the pointers In Example A–25 and the arrays they point to.

Pointer	for array
rr9ptr and rr95ptr	rr[i9][i9] and rr[i9+5][i9+5]
rrxptr and rrx5ptr	rr[ix][i9] and rr[ix][i9+5] (where x = 0, 1, ..., 7)
rrv9ptr and rrv95ptr	rrv[i9] and rrv[i9+5]

Again, the .D unit is used the most (twenty times per iteration).

None of the pairs of rr[ix][i9], rr[iy][i9+5] hit the same memory bank (where ix, iy = i0, i1, ..., i7). The same is true for pairs rrv[i9], rrv[i9+5], as well as for rr[i9][i9] and rr[i9+5][i9+5]. For ease of understanding:

- ❑ Load rr[ix][i9], rr[ix][i9+5] together.
- ❑ Load rr[i9][i9], rr[i9+5][i9+5] together.
- ❑ Store rrv[i9], rrv[i9+5] together.

In this way, each iteration takes ten cycles without any memory bank hits. You double the speed by unrolling the loop once.

The final assembly code is shown in Example A–26.

Example A–26. Assembly Code for the rrv Computation in search_10i40

```
*********************************************************************************
**     Texas Instruments, Inc                                                  **
**                                                                             **
**     Implementation of the rrv Computation in search_10i40 in EFR            **
**                                                                             **
**     Compute two rrvs a time                                                 **
**                                                                             **
**     Total cycles = 55                                                       **
**                                                                             **
**     Register Usage:              A              B                           **
**                                                                             **
**                                  16             14                          **
**                                                                             **
*********************************************************************************
                           ; B4 --- i0
                           ; B5 --- i1
                           ; B6 --- i2
                           ; B7 --- i3
                           ; A8 --- i4
                           ; B9 --- i5
                           ; A10 -- i6
                           ; A11 -- i7
                           ; B3 --- i9
                           ; A15 --- &rr[0][0]
                           ; A0 --- &rrv[0]
                           ; B14 --- stack pointer

       MVK     .S1     410,A2              ; offset of rr[i9][i9]
||     MVK     .S2     410,B2              ; offset of rr[i9+5][i9+5]

       MVK     .S2     82,B0

       MPYU    .M2     B3,B0,B3            ; [i9][i9]
||     SHL     .S1X    B3,1,A13
||     SUB     .L2     B0,2,B0             ; 80
||     ADD     .S2X    A15,B2,B13          ; &rr[5][5]

       MPYU    .M2     B4,B0,B4            ; [i0][0]
||     ADD     .L2X    A15,10,B15          ; &rr[0][5]
||     MVK     .S1     80,A1

       MPYU    .M2     B5,B0,B5            ; [i1][0]
||     ADD     .L1X    B3,A15,A3           ; &rr[i9][i9]
||     ADD     .L2     B3,B13,B3           ; &rr[i9+5][i9+5]
||     ADD     .S1     A15,A13,A15         ; &rr[0][i9]
||     ADD     .S2X    B15,A13,B15         ; &rr[0][i9+5]
||     MPYU    .M1     A10,A1,A10          ; [i6][0]
```

Example A–26. Assembly Code for the rrv Computation in search_10i40 (Continued)

```
        MPYU   .M2    B6,B0,B6            ; [i2][0]
||      MPYU   .M1    A11,A1,A11          ; [i7][0]
||      ADD    .L1X   B4,A15,A4           ; &rr[i0][i9]
||      ADD    .L2    B4,B15,B4           ; &rr[i0][i9+5]
||      LDH    .D1    *A3++[A2],A13       ; load rr[i9][i9]
||      LDH    .D2    *B3++[B2],B13       ; load rr[i9+5][i9+5]
||      ADD    .S1    A0,A13,A0           ; &rrv[i9]

        MPYU   .M2    B7,B0,B7            ; [i3][0]
||      MPYU   .M1    A8,A1,A8            ; [i4][0]
||      ADD    .L1X   B5,A15,A5           ; &rr[i1][i9]
||      ADD    .L2    B5,B15,B5           ; &rr[i1][i9+5]
||      ADD    .S1    A10,A15,A10         ; &rr[i6][9]
||      ADD    .S2X   A10,B15,B10         ; &rr[i6][i9+5]
||      LDH    .D1    *A4++[10],A13       ; load rr[i0][i9]
||      LDH    .D2    *B4++[10],B13       ; load rr[i0][i9+5]

        MPYU   .M2    B9,B0,B9            ; [i9][0]
||      ADD    .L1X   B6,A15,A6           ; &rr[i2][i9]
||      ADD    .L2    B6,B15,B6           ; &rr[i2][i9+5]
||      ADD    .S1    A11,A15,A11         ; &rr[i7][i9]
||      ADD    .S2X   A11,B15,B11         ; &rr[i7][i9+5]
||      LDH    .D1    *A5++[10],A13       ; load rr[i1][i9]
||      LDH    .D2    *B5++[10],B13       ; load rr[i1][i9+5]

        ADD    .L1X   B7,A15,A7           ; &rr[i3][i9]
||      ADD    .L2    B7,B15,B7           ; &rr[i3][i9+5]
||      ADD    .S1    A8,A15,A8           ; &rr[i4][i9]
||      ADD    .S2X   A8,B15,B8           ; &rr[i4][i9+5]
||      LDH    .D1    *A6++[10],A13       ; load rr[i2][i9]
||      LDH    .D2    *B6++[10],B13       ; load rr[i2][i9+5]

        ADD    .L1X   B9,A15,A9           ; &rr[i5][i9]
||      ADD    .L2    B9,B15,B9           ; &rr[i5][i9+5]
||      LDH    .D1    *A7++[10],A13       ; load rr[i3][i9]
||      LDH    .D2    *B7,B13             ; load rr[i3][i9+5]
||      MVK    .S2    2048,B7             ; _1_16

        LDH    .D1    *A8++[10],A13       ; load rr[i4][i9]
||      LDH    .D2    *B8++[10],B13       ; load rr[i4][i9+5]
||      SMPY   .M1X   A13,B7,A12          ; s=smpy(rr[i9][i9],_1_16)
||      SMPY   .M2    B13,B7,B12          ; S=smpy(rr[i9+5][i9+5],_1_16)
||      SHL    .S2    B7,1,B7             ; _1_8
||      ADD    .L2X   A0,10,B0            ; &rrv[i9+5]
```

Part IV

Example A–26. Assembly Code for the rrv Computation in search_10i40 (Continued)

```
       LDH    .D1    *A9++[10],A13        ; load rr[i5][i9]
||     LDH    .D2    *B9++[10],B13        ; load rr[i5][i9+5]
||     SMPY   .M1X   A13,B7,A15           ; s0=smpy(rr[i0][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S0=smpy(rr[i0][i9+5],_1_8)

       LDH    .D1    *A10++[10],A13       ; load rr[i6][i9]
||     LDH    .D2    *B10++[10],B13       ; load rr[i6][i9+5]
||     SMPY   .M1X   A13,B7,A15           ; s1=smpy(rr[i1][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S1=smpy(rr[i1][i9+5],_1_8)

       LDH    .D1    *A11++[10],A13       ; load rr[i7][i9]
||     LDH    .D2    *B11++[10],B13       ; load rr[i7][i9+5]
||     SMPY   .M1X   A13,B7,A15           ; s2=smpy(rr[i2][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S2=smpy(rr[i2][i9+5],_1_8)
||     SADD   .L1    A12,A15,A12          ; s=sadd(s,s0)
||     SADD   .L2    B12,B15,B12          ; S=sadd(S,S0)
||     MVK    .S1    3,A1                 ; loop counter

       SMPY   .M1X   A13,B7,A15           ; s3=smpy(rr[i3][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S3=smpy(rr[i3][i9+5],_1_8)
||     SADD   .L1    A12,A15,A12          ; s=sadd(s,s1)
||     SADD   .L2    B12,B15,B12          ; S=sadd(S,S1)
||     MVK    .S1    32767,A14

LOOP:
       SADD   .L1    A12,A15,A12          ; s=sadd(s,s2)
||     SADD   .L2    B12,B15,B12          ; S=sadd(S,S2)
||     SMPY   .M1X   A13,B7,A15           ; s4=smpy(rr[i4][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S4=smpy(rr[i4][i9+5],_1_8)
||     LDH    .D1    *A3++[A2],A13        ;* load rr[i9][i9]
||     LDH    .D2    *B3++[B2],B13        ;* load rr[i9+5][i9+5]
||     ADD    .S1    A14,1,A14            ; 32768 for rounding

       SMPY   .M1X   A13,B7,A15           ; s5=smpy(rr[i5][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S5=smpy(rr[i5][i9+5],_1_8)
||     SADD   .L1    A12,A15,A12          ; s=sadd(s,s3)
||     SADD   .L2    B12,B15,B12          ; S=sadd(S,S3)
||     LDH    .D1    *A4++[10],A13        ;* load rr[i0][i9]
||     LDH    .D2    *B4,B13              ;* load rr[i0][i9+5]

       SMPY   .M1X   A13,B7,A15           ; s6=smpy(rr[i6][i9],_1_8)
||     SMPY   .M2    B13,B7,B15           ; S6=smpy(rr[i6][i9+5],_1_8)
||     SADD   .L1    A12,A15,A12          ; s=sadd(s,s4)
||     SADD   .L2    B12,B15,B12          ; S=sadd(S,S4)
||     LDH    .D1    *A5++[10],A13        ;* load rr[i1][i9]
||     LDH    .D2    *B5++[10],B13        ;* load rr[i1][i9+5]
```

Example A–26. Assembly Code for the rrv Computation in search_10i40 (Continued)

```
        SMPY    .M1X    A13,B7,A15          ; s7=smpy(rr[i7][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ; S7=smpy(rr[i7][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12         ; s=sadd(s,s5)
||      SADD    .L2     B12,B15,B12         ; S=sadd(S,S5)
||      LDH     .D1     *A6++[10],A13       ;* load rr[i2][i9]
||      LDH     .D2     *B6++[10],B13       ;* load rr[i2][i9+5]
||      ADD     .S2X    A7,10,B7            ; &rr[i3][i9+5]

        SADD    .L1     A12,A15,A12         ; s=sadd(s,s6)
||      SADD    .L2     B12,B15,B12         ; S=sadd(S,S6)
||      LDH     .D1     *A7++[10],A13       ;* load rr[i3][i9]
||      LDH     .D2     *B7,B13             ;* load rr[i3][i9+5]
||      MVK     .S2     2048,B7             ; _1_16
||[A1]  B       .S1     LOOP                ; branch to the loop

        SADD    .L1     A12,A15,A12         ; s=sadd(s,s7)
||      SADD    .L2     B12,B15,B12         ; S=sadd(S,S7)
||      LDH     .D1     *A8++[10],A13       ;* load rr[i4][i9]
||      LDH     .D2     *B8++[10],B13       ;* load rr[i4][i9+5]
||      SMPY    .M1X    A13,B7,A12          ;* s=smpy(rr[i9][i9],_1_16)
||      SMPY    .M2     B13,B7,B12          ;* S=smpy(rr[i9+5][i9+5],_1_16)
||      SHL     .S2     B7,1,B7             ; _1_8
||[A1]  SUB     .S1     A1,1,A1             ; decrement loop counter

        SADD    .L1     A12,A14,A14         ; round(s)
||      SADD    .L2X    B12,A14,B4          ; round(S)
||      LDH     .D1     *A9++[10],A13       ;* load rr[i5][i9]
||      LDH     .D2     *B9++[10],B13       ;* load rr[i5][i9+5]
||      SMPY    .M1X    A13,B7,A15          ;* s0=smpy(rr[i0][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ;* S0=smpy(rr[i0][i9+5],_1_8)

        SHR     .S1     A14,16,A14          ; rrv[i9]
||      SHR     .S2     B4,16,B4            ; rrv[i9+5]
||      SMPY    .M1X    A13,B7,A15          ;* s1=smpy(rr[i1][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ;* S1=smpy(rr[i1][i9+5],_1_8)
||      LDH     .D1     *A10++[10],A13      ;* load rr[i6][i9]
||      LDH     .D2     *B10++[10],B13      ;* load rr[i6][i9+5]

        SMPY    .M1X    A13,B7,A15          ;* s2=smpy(rr[i2][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ;* S2=smpy(rr[i2][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12         ;* s=sadd(s,s0)
||      SADD    .L2     B12,B15,B12         ;* S=sadd(S,S0)
||      LDH     .D1     *A11++[10],A13      ;* load rr[i7][i9]
||      LDH     .D2     *B11++[10],B13      ;* load rr[i7][i9+5]
```

Part IV

Example A–26. Assembly Code for the rrv Computation in search_10i40 (Continued)

```
        STH     .D1     A14,*A0++[10]       ; store rrv[i9]
||      STH     .D2     B4,*B0++[10]        ; store rrv[i9+5]
||      SMPY    .M1X    A13,B7,A15          ;* s3=smpy(rr[i3][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ;* S3=smpy(rr[i3][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12         ;* s=sadd(s,s1)
||      SADD    .L2     B12,B15,B12         ;* S=sadd(S,S1)
||      ADD     .S2X    A4,10,B4            ;* &rr[i0][i9+5]
||      MVK     .S1     32767,A14           ; end of LOOP
```

Because of the shortage of registers:

- B7 serves as _1_16, _1_8 and as the pointer for rr[i3][i9+5].
- B4 serves both the value of rrv[i9+5] and the pointer to rr[i0][i9+5].
- A14 represents 0x8000L as well as rrv[i9].

The last iteration of the loop can be expanded as the epilog of the loop to overlap with the prolog of the code that follows this part of the code.

A.2.5 Implementation of the Index Search in search_10i40

The index search in search_10i40 is the core of search_10i40. The C code is shown in Example A–27.

Example A–27. C Code for the Index Search for search_10i40

```
#define  L_CODE     40
#define  STEP  5
#define  _1_16 (Word16)(32768L/16)
#define  _1_8  (Word16)(32768L/8)

input:
       Word16 rr[L_CODE][L_CODE], ipos[L_CODE], dn[L_CODE];

local variables/arrays:
       Word16 rrv[L_CODE];
       Word16 i0,i1,i2,i3,i4,i5,i6,i7,i8,i9; /* defined on [0,L_CODE-1] */
       Word16 ia,ib;
       Word16 ps,ps0,ps1,ps2,sq,sq2;
       Word16 alp,alp_16;
       Word32 s,alp0,alp1,alp2;
```

(The values of i0, i1, i2, i3, i4, i5, i6, i7 , ps0, and alp0 have been obtained before entering this loop.)

```
Original C code
-------------------------------------------------------
       sq = -1;
       alp = 1;
       ps = 0;
       ia = ipos[8];
       ib = ipos[9];

       /* initialize 10 indices for i8 loop (see i2-i3 loop) */
       for (i8 = ipos[8]; i8 < L_CODE; i8 += STEP)
       {
           ps1 = add (ps0, dn[i8]);

           alp1 = L_mac (alp0, rr[i8][i8], _1_128);
           alp1 = L_mac (alp1, rr[i0][i8], _1_64);
           alp1 = L_mac (alp1, rr[i1][i8], _1_64);
           alp1 = L_mac (alp1, rr[i2][i8], _1_64);
           alp1 = L_mac (alp1, rr[i3][i8], _1_64);
           alp1 = L_mac (alp1, rr[i4][i8], _1_64);
           alp1 = L_mac (alp1, rr[i5][i8], _1_64);
           alp1 = L_mac (alp1, rr[i6][i8], _1_64);
           alp1 = L_mac (alp1, rr[i7][i8], _1_64);
```

Part IV

Example A–27. C Code for the Index Search for search_10i40 (Continued)

```
            /* initialize 3 indices for i9 inner loop (see i2–i3 loop) */

           for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
           {
              ps2 = add (ps1, dn[i9]);

              alp2 = L_mac (alp1, rrv[i9], _1_8);
              alp2 = L_mac (alp2, rr[i8][i9], _1_64);

              sq2 = mult (ps2, ps2);

              alp_16 = round (alp2);

              s = L_msu (L_mult (alp, sq2), sq, alp_16);

              if (s > 0) {
                   sq = sq2;
                   ps = ps2;
                   alp = alp_16;
                   ia = i8;
                   ib = i9;
              }
           }
       }
-------------------------------------------------------
  where add(a,b) = _sadd(a<<16,b<<16)>>16
        L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        mult(a,b) = _smpy(a<<16,b<<16)>>16
        L_mult(a,b)=_smpy(a,b)
        round(a) = _sadd(a,0x8000L)>>16
  and   L_msu(a,b,c)=_ssub(a,_smpy(b,c))
```

This is a typical example of the performance being limited by data dependency constraints. In this case, the dependency is between the values of alp and sq.

A.2.5.1 Rearranging the C Code

To avoid the unnecessary shift, ps, ps1, ps2, alp, alp_16, sq, and sq2 are implemented as int (Word32) variables. The calculations are implemented as:

Original	Implemented as
`ps1 = add (ps0, dn[i8]);`	`ps1 = _sadd(ps0, dn[i8]<<16);`
`ps2 = add (ps1, dn[i9]);`	`ps2 = _sadd(ps1, dn[i9]<<16);`
`sq2 = mult (ps2, ps2);`	`sq2 = _smpyh(ps2,ps2);`
`alp_16 = round(alp2);`	`alp_16 = _sadd(alp2,0x8000L);`

There is no need to compute s explicitly. Instead of implementing the following sequence:

```
s = L_msu (L_mult (alp, sq2), sq, alp_16);

if (s > 0)
{
   sq = sq2;
   ps = ps2;
   alp = alp_16;
   ia = i8;
   ib = i9;
   }
```

you can do this sequence to fulfill the same task:

```
if(_smpyh(alp,sq2) > _smpyh(sq,alp_16)) {
  sq = sq2;
  ps = ps2;
  alp = alp_16;
  ia = i8;
  ib = i9;
}
```

A.2.5.2 Performance Analysis

The instructions to execute one iteration of the inner loop are shown in Example A–28.

Example A–28. Linear Assembly for the Index Search for search_10i40 (Inner Loop)

```
INNERLOOP:
        LDH   .D     *dn9ptr++[5],dn9      ; load dn[i9]
        SHL   .S     dn9,16,dn9h           ; dn[i9] << 16
        SADD  .L     ps1,dn9h,ps2          ; ps2 = sadd(ps1, dn[i9] << 16)
        SMPYH .M     ps2,ps2,sq2           ; sq2 = smpyh(ps2,ps2)
        LDH   .D     *rrvptr++[5],rrv      ; load rrv[i9]
        SMPY  .M     rrv,_1_8,tmp1         ; smpy(rrv[i9], _1_8)
        SADD  .L     alp1,temp1,alp2       ; alp2=sadd(alp1,smpy(rrv[i9],_1_8))
        LDH   .D     *rr89prt++,rr89       ; load rr[i8][i9]
        SMPY  .M     rr89,_1_64,tmp2       ; smpy(rr[i8][i9],_1_64)
        SADD  .L     alp2,tmp2,alp2        ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
        SADD  .L     alp2,0x8000L,alp_16   ; alp_16=sadd(alp2,0x8000L)
        SMPYH .M     alp,sq2,tmp3          ; smpyh(alp,sq2)
        SMPYH .M     sq,alp_16,tmp4        ; smpyh(sq,alp_16)
        CMPGT .L     tmp3,tmp4,cndr        ; if(smpyh(alp,sq2) > smpyh(sq,alp_16))
 [cndr] MV     .ALU   sq2,sq
 [cndr] MV     .ALU   ps2,ps
 [cndr] MV     .ALU   alp_16,alp
 [cndr] MV     .ALU   i8,ia
 [cndr] MV     .ALU   i9,ib
 [icntr]SUB    .ALU   icntr,1,icntr
 [icntr]B      .S     INNERLOOP            ;branch to the loop
```

Because both sq and alp are carried over and required from one iteration to the next, their values should be put in registers to allow speedy retrieval. At least four cycles are required to compute new sq and alp values, and the requirement on the functional units does not exceed four execution packets. Therefore, the inner loop can be effected in four cycles per iteration.

For the outer loop, any pair of rr[ix][i8], rr[iy][i8] (where ix, iy = i0, i1, ..., i7) will definitely hit the memory bank if they are read together. Therefore, they should be loaded in one cycle each.

A.2.5.3 Partitioning the Registers

The total number of registers required for this code, including the registers for the pointer of the arrays, loop counters, intermediate results, etc., exceeds the number of registers available. To partition the registers without losing speed, the strategies are:

- For the inner loop, store the results of ps, ia, and ib, whose values are not used in this code.
- For the outer loop, store the pointers of arrays starting at rr[i5][i8], rr[i6][i8], and rr[i7][i8], whose values are needed last in the outer loop.

Assume that before entering this code, the following values are known: &dn[0], &ipos[0], &rr[0][0], &rrv[0][0], i0, i1, i2, i3, i4, i5, i6, i7, ps0, and alp0. Assume that the short (Word16) integers are stored in the stack in the order i0, i1, i2, i3, i4, i5, i6, i7, ia, and ib, and that a pointer &local_16[0], pointing to i0, is also known. The int integers and the pointers of the rr arrays are stored in the stack in the following order: ps0, ps, alp0, alp1, &rr[i5][i8], &rr[i6][i8], and &rr[i7][i8]. The pointer, &local_32[0], pointing to ps0, is known as well.

The C code is shown in Example A–29.

Example A–29. Modified C Code for the Index Search

```
sq = -1;
alp = 1;
local_32[1] = 0;
local_16[8] = ipos[8];
local_16[9] = ipos[9];

/* initialize 10 indices for i8 loop (see i2-i3 loop) */

for (i8 = ipos[8]; i8 < L_CODE; i8 += STEP) {
  ps1 = _sadd (local_32[0], dn[i8]<<16);

  local_32[3] = _sadd(local_32[2], _smpy(rr[i8][i8], _1_128));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i0][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i1][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i2][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i3][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i4][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i5][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i6][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i7][i8], _1_64));

/* initialize 3 indices for i9 inner loop (see i2-i3 loop) */

   for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP) {

     ps2 = _sadd(ps1, dn[i9]<<16);

     alp2 = _sadd(local_32[3], _smpy(rrv[i9], _1_8));
     alp2 = _sadd(alp2, _smpy(rr[i8][i9], _1_64));

     sq2 = _smpyh(ps2, ps2);

     alp_16 = _sadd(alp2,0x8000L);
```

Part IV

Example A–29. Modified C Code for the Index Search (Continued)

```
    if (_smpyh(alp,sq2) > _smpyh(sq,alp_16)) {

       sq = sq2;
       local_32[1]= ps2;
       alp = alp_16;
       local_16[8] = i8;
       local_16[9] = i9;
     }
   }
}
```

A.2.5.4 Final Assembly Code

The final code consists of the following steps:

Step 1: Load i0, i1, ... i9, alp0, and ps0; and initialize sq, ia, and ib. Part of the code overlaps that of the last iteration of the code in section A.2.4 on page A-27.

Step 2: Obtain the pointer for the arrays started at rr[i0][i8], rr[i1][i8], ... rr[i7][i8], rr[i8][i9], rrv[i9], dn[i8], and dn[i9].

Step 3: Load rr[i0][i8], rr[i1][i8], ... rr[i7][i8] and dn[i8], compute the new ps1 and alp1, update the pointers, and store pointers &rr[i5][i8], &rr[i6][i8], and &rr[i7][i8].

Step 4: Load rr[i8][i9], rrv[i9], and dn[i9]. Compute alp2, ps2, alp_16, sq2 and perform a comparison. Update the parameters ia, ib, alp, sq, and ps based on the comparison result. Repeat this step eight times.

Step 5: Reload the values of ps0 and alp0, and &rr[i5][i8], &rr[i6][i8], and &rr[i7][i8]. Verify that step 3 has been repeated eight times. If not, go to step 3. If so, exit.

To avoid memory bank hits, arrays rr and rrv must not be aligned on the same word or half-word boundary. The same applies to arrays rr and dn. As you can see in the final assembly code shown in Example A–30, there are several places that LDH (or STH) and LDW (or STW) occur in the same execution packet. They belong to one of the two categories; that is, always loading values from or storing values to the same memory locations, as in iterations like this:

```
       LDW    .D1    *+A6[3],A11  ; load alp1
||[B2] STH    .D2    B13,*+B6[9]  ; store ib=i9
```

The following instructions are used in the inner loop in different memory locations such as the outer loop:

```
   [B2]  STW   .D1   B11,*+A6[1]  ; store ps
||       LDH   .D2   *B10++[5],A5 ; load rr[i5][i8]
```

In the former case, memory bank hits can be completely eliminated by allocating the corresponding arrays in memory properly. Memory bank hits occur in every other iteration in the latter case, however. Although, in general, you should avoid writing such code, in this case, the performance of the prolog of the outer loop after the first iteration is limited by the .D unit. You still save some cycle counts in this example.

To improve the performance, the last two iterations of the inner loop overlap part of the prolog of the outer loop.

Example A–30. Assembly Code for the search_10i40 Index Search

```
**********************************************************************************
**     Texas Instruments, Inc                                                   **
**                                                                              **
**     Implementation of The Index Search in search_10i40 in EFR                **
**                                                                              **
**     Total cycles = 400 (among the 400 cycles, 10 cycles are caused           **
**                     by memory bank hits)                                     **
**                                                                              **
**     Register Usage:      A              B                                    **
**                                                                              **
**                          15             15                                   **
**                                                                              **
**********************************************************************************

                            ; A13 --- &ipos[0] and alp
                            ; B6 --- &local_16[0]
                            ; A6 --- stack pointer, point to &local_32[0]
                            ; B8 --- &rr[0][0]
                            ; A4 --- &rrv[0]
                            ; B14 --- &dn[0]
                            ; B1 --- reserved for the counter of the
                            ;        outmost loop in search_10i40


       LDH    .D1    *+A13[8],A7               ; load i8 = ipos[8]

       LDH    .D1    *+A13[9],B13              ; load i9 = ipos[9]
||     LDH    .D2    *B6,A13                   ; load i0
||     MV     .S1X   B6,A5                     ; &local_v16[0]
```

Part IV

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
        LDH     .D2     *+B6[2],B9                      ; load i2
||      LDH     .D1     *+A5[1],A14                     ; load i1
        MVK     .S1     0,A8                            ; could insert two .D
                                                        ; units here for the store
                                                        ; of rrv[i9+30] and rrv[i9+35]
                                                        ; in the code which this piece
                                                        ; immediately follows

        LDH     .D1     *+A5[4],A15                     ; load i4
||      LDH     .D2     *+B6[3],B10                     ; load i3
||      MVK     .S1     80,A0
||      MVK     .S2     80,B0

        STW     .D1     A8,*+A6[1]                      ; ps=0
||      LDH     .D2     *+B6[5],B11                     ; load i5
||      SHL     .S2X    A7,1,B10                        ; [0][i8]
||      MPYU    .M1     A7,A0,A12                       ; [i8][0]

        STH     .D1     A7,*+A5[8]                      ; store ia=i8
||      STH     .D2     B13,*+B6[9]                     ; store ib=i9
||      ADD     .L2     B8,B10,B2                       ; &rr[0][i8]
||      MPYU    .M2X    A13,B0,B3                       ; [i0][0]

        LDW     .D1     *A6,B15                         ; load ps0
||      LDH     .D2     *+B6[6],A1                      ; load i6
||      ADD     .S1X    A12,B2,A12                      ; &rr[i8][i8]
||      ADD     .S2     B14,B10,B7                      ; &dn[i8]
||      ADD     .L2X    B8,A12,B8                       ; &rr[i8][0]
||      MPYU    .M1     A14,A0,A14                      ; [i1][0]
||      MPYU    .M2     B9,B0,B9                        ; [i2][0]

        LDW     .D1     *+A6[2],A11                     ; load alp0
||      LDH     .D2     *+B6[7],B5                      ; load i7
||      ADD     .S2     B13,B13,B12                     ; [0][i9]
||      ADD     .L2     B3,B2,B3                        ; &rr[i0][i8]

        LDH     .D1     *A12,A5                         ; load rr[i8][i8]
||      LDH     .D2     *B7++[5],B12                    ; load dn[i8]
||      ADD     .S2     B14,B12,B14                     ; &dn[i9]
||      ADD     .L1X    A14,B2,A14                      ; &rr[i1][i8]

        LDH     .D2     *B3++[5],A5                     ; load rr[i0][i8]
||      ADD     .L2     B9,B2,B9                        ; &rr[i2][i8]
||      MPYU    .M1X    B10,A0,A9                       ; [i3][0]

        LDH     .D1     *A14++[5],A5                    ; load rr[i1][i8]
||      ADD     .L1X    A4,B12,A4                       ; &rrv[i9]
||      MPYU    .M1     A15,A0,A15                      ; [i4][0]
||      MPYU    .M2     B11,B0,B11                      ; [i5][0]
```

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
        LDH     .D2     *B9++[5],A5           ; load rr[i2][i8]
||      MVK     .S1     256,A0                ; A0=_1_128
||      ADD     L1X     A9,B2,A9              ; &rr[i3][i8]
||      MPYU    .M1     A1,A0,A1              ; [i6][0]

        LDH     .D1     *A9++[5],B12          ; load rr[i3][i8]
||      ADD     .D2     B11,B2,B10            ; &rr[i5][i8]
||      MVK     .S1     7,A2                  ; outer loop counter
||      MVK     .S      512,B0                ; B0=_1_64
||      ADD     .L1X    A15,B2,A15            ; &rr[i4][i8]
||      ADD     .L2     B8,B12,B4             ; &rr[i8][i9]
||      MPYU    .M2     B5,B0,B5              ; [i7][0]

        LDH     .D1     *A15++[5],A5          ; load rr[i4][i8]
||      SHL     .S1     A0,1,A0               ; _1_64

||      SHL     .S2     B12,16,B11            ; dn[i8] << 16
||      ADD     .L1X    A1,B2,A1              ; &rr[i6][i8]
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i8][i8],_1_128)

        LDH     .D2     *B10++[5],A5          ; load rr[i5][i8]
||      MVK     .S1     -1,A3                 ; sq=-1
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i0][i8],_1_64)

        LDH     .D1     *A1++[5],B12          ; load rr[i6][i8]
||      ADD     .D2     B5,B2,B11             ; &rr[i7][i8]
||      SHL     .S1     A0,7,A13              ; alp=0x10000
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp0,smpy(rr[i8][i8],_1_128))
||      SADD    .L2     B15,B11,B15           ; ps1
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i1][i8],_1_64)

        LDH     .D2     *B11++[5],A5          ; load rr[i7][i8]
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i0][i8],_1_64))
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i2][i8],_1_64)

OUTERLOOP:

        LDH     .D1     *A4++[5],A5           ; load rrv[i9]
||      LDH     .D2     *B4++[5],B12          ; load rr[i8][i9]
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i1][i8],_1_64))
||      SUB     .L2     B13,5,B13
||      SMPY    .M1X    B12,A0,A8             ; smpy(rr[i3][i8],_1_64)

        LDH     .D2     *B14++[5],B12         ; load dn[i9]
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i2][i8],_1_64))
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i4][i8],_1_64)
```

Part IV

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
        STW     .D1     B10,*+A6[4]         ; store &rr[i5][i8+5]
||      SADD    .L1     A11,A8,A11          ; alp1=sadd(alp1,smpy(rr[i3][i8],_1_64))
||      SMPY    .M1     A5,A0,A8            ; smpy(rr[i5][i8],_1_64)

        STW     .D1     A1,*+A6[5]          ; store &rr[i6][i8+5]
||      SHL     .S1     A0,6,A10            ; 0x8000L
||      SADD    .L1     A11,A8,A11          ; alp1=sadd(alp1,smpy(rr[i4][i8],_1_64))
||      SMPY    .M1X    B12,A0,A8           ; smpy(rr[i6][i8],_1_64)

        LDH     .D1     *A4++[5],A5         ;* load rrv[i9]
||      LDH     .D2     *B4++[5],B12        ;* load rr[i8][i9]
||      SHL     .S1     A0,3,A0             ; A0=_1_8
||      SADD    .L1     A11,A8,A11          ; alp1=sadd(alp1,smpy(rr[i5][i8],_1_64))
||      SMPY    .M1     A5,A0,A8            ; smpy(rr[i7][i8],_1_64)


        LDH     .D2     *B14++[5],B12       ;* load dn[i9]
||      SADD    .L1     A11,A8,A11          ; alp1=sadd(alp1,smyp(rr[i6][i8],_1_64))
||      SMPY    .M1     A5,A0,A5            ; smpy(rrv[i9],_1_8)
||      SMPY    .M2     B12,B0,B12          ; smpy(rr[i8][i9],_1_64)


        STW     .D      B11,*+A6[6]         ; store &rr[i7][i8+5]
||      SHL     .S2     B12,16,B1           ; dn[i9] << 16
||      SADD    .L1     A11,A8,A11          ; done alp1=sadd(alp1,smpy(rr[i7][i8],_1_64))


        STW     .D1     A11,*+A6[3]         ; store alp1
||      SADD    .L1     A11,A5,A5           ; alp2=sadd(alp1,smpy(rrv[i9],_1_8))
||      SADD    .L2     B11,B15,B5          ; ps2=sadd(ps1,dn[i9]<<16)


        LDH     .D1     *A4++[5],A5         ;** load rrv[i9]
||      LDH     .D2     *B4++[5],B12        ;** load rr[i8][i9]
||      B       .S2     INNERLOOP           ; branch to the innerloop
||      SADD    .L1X    A5,B12,A1           ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||      SMPYH   .M2     B5,B5,B8            ; sq2=smpyh(ps2,ps2)


        LDH     .D2     *B14++[5],B12       ;** load dn[i9]
||      MVK     .S1     4,A1                ; innerloop counter
||      MVK     .S2     0,B2
||      SADD    .L1     A1,A10,A8           ; alp_16 = sacc(alp2, 0x8000L)
||      SMPY    .M1     A5,A0,A5            ;* smpy(rrv[i9],_1_8)
||      SMPY    .M2     B12,B0,B12          ;* smpy(rr[i8][i9],_1_64)
```

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
INNERLOOP:

        LDW     .D1     *+A6[3],A11         ; load alp1
||[B2]  STH     .D2     B13,*+B6[9]         ; store ib=i9
||      SHL     .S2     B12,16,B10          ;* dn[i9]<<16
||      ADD     .L2     B13,5,B13           ; i9=i9+STEP
||      SMPYH   .M1     A8,A3,A11           ; smpyh(alp_16,sq)
||      SMPYH   .M2X    B8,A13,B10          ; smpyh(alp,sq2)


 [B2]   STW     .D1     B11,*+A6[1]         ; store ps
||[B2]  STH     .D2     A7,*+B6[8]          ; store ia = i8
||      MV      .S2     B5,B11              ;
||      SADD    .L1     A11,A5,A5           ;*alp2=sadd(alp1,smpy(rrv[i9],_1_8))
||      SADD    .L2     B10,B15,B5          ;* ps2=sadd(ps1,dn[i9]<<16)


        LDH     .D1     *A4++[5],A5         ;*** load rrv[i9+10]
||      LDH     .D2     *B4++[5],B12        ;*** load rr[i8][i9+10]
||[A1]  SUB     .S1     A1,1,A1             ; decrement innerloop counter
||[A1]  B       .S2     INNERLOOP           ; branch to INNERLOOP
||      SADD    .L1X    A5,B12,A11          ;*alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||      CMPGT   .L2X    B10,A11,B2          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)
||      SMPYH   .M2     B5,B5,B8            ;* sq2=smpyh(ps2,ps2)


  [B2]  MV      .D1     A8,A13              ; alp=alp_16
||      LDH     .D2     *B14++[5],B12       ;*** load dn[i9+10]
||[B2]  MV      .S1X    B8,A3               ; sq=sq2
||      SADD    .L1     A11,A10,A8          ;* alp_16=sadd(alp2, 0x8000L)
||      SMPY    .M1     A5,A0,A5            ;*** A0 = _1_8
||      SMPY    .M2     B12,B0,B12          ;*** B0 = _1_64
                                            ; end of innerloop


 [B2]  STW    .D1    B11,*+A6[1]            ; store ps
||[B2] STH    .D2    A7,*+B6[8]             ; store ia = i8
||     SHL    .S2    B12,16,B10             ; dn[i9]<<16
||     MV     .L2    B5,B11                 ; ps2
||     SMPYH  .M1    A8,A3,A11              ; smpyh(alp_16,sq)
||     SMPYH  .M2X   B8,A13,B10             ; smpyh(alp,sq2)


       LDW    .D1    *+A6[2],A11            ; load alp0
||[B2] STH    .D2    B13,*+B6[9]            ; store ib=i9
||     MV     .S2X   A6,B2                  ; stack pointer
||     SADD   .L1    A11,A5,A5              ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||     SADD   .L2    B10,B15,B5             ; ps2=sadd(ps1,dn[i9]<<16)
```

Part IV

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
        LDW     .D1     *+A6[5],A1          ; &rr[i6][i8]
||      LDW     .D2     *B2,B15             ; load ps0
||      MVK     .S1     205,A0
||      SADD    .L1X    A5,B12,A11          ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||      CMPGT   .L2X    B10,A11,B2          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)
||      SMPYH   .M2     B5,B5,B8            ; sq2=smpyh(ps2,ps2)


        LDH     .D1     *++A12[A0],A5       ; load rr[i8][i8]
||      LDH     .D2     *B7++[5],B12        ; load dn[i8]
||[B2]  MV      .S1     A8,A13              ; alp=alp_16
||      ADDK    .S2     -90,B14             ; &dn[i9]
||[B2]  MV      .L1X    B8,A3               ; sq=sq2


        LDW     .D1     *+A6[4],B10         ; &rr[i5][i8]
||      LDH     .D2     *B3++[5],A5         ; load rr[i0][i8]
||      ADDK    .S1     -90,A4              ; &rrv[i9]
||      SADD    .L1     A11,A10,A8          ; alp_16=sadd(alp2, 0x8000L)
||      ADD     .L2     B13,5,B13


        LDH     .D1     *A14++[5],A5        ; load rr[i1][i8]
||[B2]  STH     .D2     B13,*+B6[9]         ; store ib=i9
||      SMPYH   .M1     A8,A3,A10           ; smpyh(alp_16,sq)
||      SMPYH   .M2X    B8,A13,B10          ; smpyh(alp,sq2)


        MVK     .S1     256,A0              ; _1_128
||      LDW     .D1     *+A6[6],B11         ; &rr[i7][i8]
||      LDH     .D2     *B9++[5],A5         ; load rr[i2][i8]
||[A2]  B       .S2     OUTERLOOP           ; branch to OUTERLOOP


        LDH     .D1     *A9++[5],B12        ; load rr[i3][i8]
||[B2]  STH     .D2     A7,*+B6[8]          ; store ia = i8
||      ADD     .S2     B13,5,B13           ; update i9
||      CMPGT   .L2X    B10,A10,B0          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)


        LDH     .D1     *A15++[5],A5        ; load rr[i4][i8]
||[B0]  STH     .D2     B13,*+B6[9]         ; store ib=i9
||      SHL     .S1     A0,1,A0             ; _1_64
||      ADDK    .S2     -35,B13             ; update i9
||[B0]  MV      .L1     A8,A13              ; alp=alp_16
||      SMPY    .M1     A5,A0,A             ; smpy(rr[i8][i8],_1_128)
```

Part IV

Example A–30. Assembly Code for the search_10i40 Index Search (Continued)

```
 [B2]  STW    .D1B11,*+A6[1]            ; store ps
||     LDH    .D2    *B10++[5],A5       ; load rr[i5][i8]
||[B0] MV     .S1X   B8,A3              ; sq=sq2
||     SHL    .S2    B12,16,B11         ; dn[i8] << 16
||[A2] SUB    .L1    A2,1,A2            ; decrement OUTERLOOP counter
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i0][i8],_1_64)


       LDH    .D1    *A1++[5],B12       ; load rr[i6][i8]
||[B0] STH    .D2    A7,*+B6[8]         ; store ia = i8
||     ADDK   .S2    310,B4             ; &rr[i8][i9]
||     SADD   .L1    A11,A8,A11         ; alp1=sadd(alp0,smpy(rr[i8][i8],_1_128))
||     SADD   .L2    B15,B11,B15        ; ps1 = sadd(ps0,dn[i8]<<16)
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i1][i8],_1_64)


 [B0]  STW    .D1    B5,*+A6[1]         ; store ps
||     LDH    .D2    *B11++[5],A5       ; load rr[i7][i8]
||     ADD    .S1    A7,5,A7            ; update i8
||     SADD   .L1    A11,A8,A11         ; alp1=sadd(alp1,smpy(rr[i0][i8],_1_64))
||     MV     .L2X   A0,B0              ; _1_64
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i2][i8],_1_64)
```

A.2.6 Implementation of the FIR Filter, residu.c, in GSM EFR Vocoder

Example A–31 shows the C code for the FIR filter, residu.c, in the GSM EFR vocoder.

Example A–31. C Code for residu.c

```
 #define Word16  short #define Word32  int
  Original C code
------------------------------------------------------
/* m = LPC order == 10 */ #define m 10

void Residu (
    Word16 a[], /* (i)     : prediction coefficients                      */
    Word16 x[], /* (i)     : speech signal                                */
    Word16 y[], /* (o)     : residual signal                              */
    Word16 lg   /* (i)     : size of filtering                            */
)
{
    Word16 i, j;
    Word32 s;


    for (i = 0; i < lg; i++)
    {
        s = L_mult (x[i], a[0]);
        for (j = 1; j <= m; j++)
            s = L_mac (s, a[j], x[i - j]);
        s = L_shl (s, 3);
        y[i] = round (s);
    }
    return;
}
-------------------------------------------------------
  where L_mult(a,b) = _smpy(a,b)
        L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        L_shl(a,b) = (b>0) ? _sshl(a,b) : a >> (-b)
        round(a) = _sadd(a,0x8000L)>>16
   and  lg = 40.
```

A.2.6.1 Rearranging the C Code

L_shl (s, 3) can be implemented simply as _sshl (s,3). Because array *a* has dimension m + 1 = 11 and the inner loop is always executed 10 times per outer loop iteration, you can completely unroll the inner loop to gain speed by representing array *a* with registers. Because *a* is a short integer array, it requires six registers at most for full representation. You can assign one register only for a[0] for the following reasons:

- a[0] is always a constant, 4096
- _shr (0x8000L, 3) = 4096

You can change the order of rounding and left shift to save one register. (Otherwise, you need another register for 0x8000L.) The C code, after complete inner loop unrolling, is shown in Example A–32.

Example A–32. C Code for residu.c After Rearrangement Using Intrinsics

```
for (i = 0; i < lg; i++)
{
    s = _smpy(x[i], a[0]);
    s = _sadd(s,_smpy(a[1], x[i-1]));
    s = _sadd(s,_smpy(a[2], x[i-2]));
    s = _sadd(s,_smpy(a[3], x[i-3]));
    s = _sadd(s,_smpy(a[4], x[i-4]));
    s = _sadd(s,_smpy(a[5], x[i-5]));
    s = _sadd(s,_smpy(a[6], x[i-6]));
    s = _sadd(s,_smpy(a[7], x[i-7]));
    s = _sadd(s,_smpy(a[8], x[i-8]));
    s = _sadd(s,_smpy(a[9], x[i-9]));
    s = _sadd(s,_smpy(a[10], x[i-10]));
    s = _sadd(s, a[0]);
    s = _sshl(s,3);
    y[i] = _shr(s,16);
}
```

A.2.6.2 Performance Analysis

The performance is limited by the .L unit for _sadd because this unit is used at least 11 times per iteration. In other words, it takes at least six cycles per iteration. You may choose to unroll the loop once to compute two y values per iteration for the following reasons:

- To satisfy the ordering property of _sadd
- To maximize speed: eleven cycles are required to compute two y values, while six cycles are needed for one y

The C code is is shown in Example A–33.

Example A–33. Implemented C Code for residu.c

```
for (i = 0; i < lg; i+=2)
{
    s0 = _smpy(x[i], a[0]);
    s1 = _smpy(x[i+1], a[0]);
    s0 = _sadd(s0,_smpy(a[1], x[i-1]));
    s1 = _sadd(s1,_smpy(a[1], x[i]));
    s0 = _sadd(s0,_smpy(a[2], x[i-2]));
    s1 = _sadd(s1,_smpy(a[2], x[i-1]));
    s0 = _sadd(s0,_smpy(a[3], x[i-3]));
    s1 = _sadd(s1,_smpy(a[3], x[i-2]));
    s0 = _sadd(s0,_smpy(a[4], x[i-4]));
    s1 = _sadd(s1,_smpy(a[4], x[i-3]));
    s0 = _sadd(s0,_smpy(a[5], x[i-5]));
    s1 = _sadd(s1,_smpy(a[5], x[i-4]));
    s0 = _sadd(s0,_smpy(a[6], x[i-6]));
    s1 = _sadd(s1,_smpy(a[6], x[i-5]));
    s0 = _sadd(s0,_smpy(a[7], x[i-7]));
    s1 = _sadd(s1,_smpy(a[7], x[i-6]));
    s0 = _sadd(s0,_smpy(a[8], x[i-8]));
    s1 = _sadd(s1,_smpy(a[8], x[i-7]));
    s0 = _sadd(s0,_smpy(a[9], x[i-9]));
    s1 = _sadd(s1,_smpy(a[9], x[i-8]));
    s0 = _sadd(s0,_smpy(a[10], x[i-10]));
    s1 = _sadd(s1,_smpy(a[10], x[i-9]));
    s0 = _sadd(s0, a[0]);
    s1 = _sadd(s1, a[0]);
    s0 = _sshl(s0,3);
    s1 = _sshl(s1,3);
    y[i] = _shr(s0,16);
    y[i+1] = _shr(s1,16);
}
```

A.2.6.3 Final Assembly Code for residu.c

The final assembly code is shown in Example A–34.

Example A–34. Assembly Code for residu.c

```
*********************************************************************************
**                                                                             **
**      Implementation of residu.c EFR                                         **
**                                                                             **
**      Compute two ys at a time                                               **
**                                                                             **
**      Total cycles = (lg/2+1)*11+6                                           **
**                   = 237  (for lg = 40)                                      **
**      Register Usage:             A                B                         **
**                                  9                10                        **
**                                                                             **
*********************************************************************************
                                   ; A4 --- &a[0]
                                   ; B4 --- &x[0]
                                   ; A6 --- &y[0]
                                   ; B6 --- lg

        LDH     .D2     *B4++,B0          ; load a[0] = 4096

        LDW     .D1     *A4--,A3          ; load x[0] & x[1]
||      LDW     .D2     *B4++,B4          ; load a[1] & a[2]

        LDW     .D1     *A4--,A1          ; load x[-2] & x[-1]
||      LDW     .D2     *B4++,B1          ; load a[3] & a[4]

        LDW     .D2     *B4++,B5          ; load a[5] & a[6]

        LDW     .D2     *B4++,B6          ; load a[7] & a[8]
||      LDW     .D1     *A4--,A3          ; load x[-4] & x[-3]

        LDW     .D2     *B4++,B7          ; load a[9] & a[10]
||      MVK     .S1     1,A2              ; to take care of the first execution
||      MV      .L1X    B0,A0             ; a[0] = 4096
||      MV      .S2     B6,B2             ; loop counter, L_SUBFR/2

LOOP:
        SMPY    .M1     A3,A0,A8          ; smpy(x[0],a[0])
||      SMPYHL  .M2X    A3,B0,B8          ; smpy(x[1],a[0])
||      LDW     .D1     *A4--,A1          ; load x[-6] & x[-5]
||[!A2] SADD    .L1     A8,A9,A9          ; s0 = sadd(s0, smpy(x[-9],a[9]))
||[!A2] SADD    .L2     B8,B9,B9          ; s1 = sadd(s1, smpy(x[-8],a[9]))

        SMPYHL  .M1X    A1,B4,A8          ; smpy(x[-1],a[1])
||      SMPY    .M2X    A3,B4,B8          ; smpy(x[0],a[1])
||[!A2] SADD    .L1     A8,A9,A9          ; s0 = sadd(s0, smpy(x[-10],a[10]))
||[!A2] SADD    .L2     B8,B9,B9          ; s1 = sadd(s1, smpy(x[-9],a[10]))
```

Part IV

Example A–34. Assembly Code for residu.c (Continued)

```
        SMPYLH  .M1X    A1,B4,A8        ; smpy(x[-2],a[2])
||      SMPYH   .M2X    A1,B4,B8        ; smpy(x[-1],a[2])
||      ADD     .S1     A8,0,A9         ; s0=smpy(x[0],a[0])
||      ADD     S2      B8,0,B9         ; s1=smpy(x[1],a[0])
||      LDW     .D1     *A4--,A3        ; load x[-8] & x[-7]
||[!A2] SADD    L1      A9,A0,A9        ; s0 = sadd(s0, 4096)
||[!A2] SADD    .L2     B9,B0,B9        ; s1 = sadd(s1, 4096)

        SMPYHL  .M1X    A3,B1,A8        ; smpy(x[-3],a[3])
||      SMPY    .M2X    A1,B1,B8        ; smpy(x[-2],a[3])
||      SADD    .L1     A8,A9,A9        ; s0 = sadd(s0, smpy(x[-1],a[1]))

||      SADD    .L2     B8,B9,B9        ; s1 = sadd(s1, smpy(x[0],a[1]))
||[!A2] SSHL    .S1     A9,3,A7         ; s0 = L_shl(s0,3)
||[!A2] SSHL    .S2     B9,3,B1         ; s1 = L_shl(s1,3)

        SMPYLH  .M1X    A3,B1,A8        ; smpy(x[-4],a[4])
||      SMPYH   .M2X    A3,B1,B8        ; smpy(x[-3],a[4])
||      SADD    .L1     A8,A9,A9        ; s0 = sadd(s0, smpy(x[-2],a[2]))
||      SADD    .L2     B8,B9,B9        ; s1 = sadd(s1, smpy(x[-1],a[2]))
||      LDW     .D1     *A4++[6],A1     ; load x[-10] & x[-9] and update the
                                          pointer
||[!A2] SHR     .S1     A7,16,A7        ; y[0] = shr(s0, 16)
||[!A2] SHR     .S2     B10,16,B10      ; y[1] = shr(s1, 16)
                                        ; to the new &x[0]

        SMPYHL  .M1X    A1,B5,A8        ; smpy(x[-5],a[5])
||      SMPY    .M2X    A3,B5,B8        ; smpy(x[-4],a[5])
||      SADD    .L1     A8,A9,A9        ; s0 = sadd(s0, smpy(x[-3],a[3]))
||      SADD    .L2     B8,B9,B         ; s1 = sadd(s1, smpy(x[-2],a[3]))
||[!A2] STH     .D1     A7,*A6++        ; store y[0]
|| [B2] SUB     .S2     B2,2,B          ; decrement loop counter
|| [B2] B       S1      LOOP            ; branch to the loop

        SMPYLH  .M1X    A1,B5,A8        ; smpy(x[-6],a[6])
||      SMPYH   .M2X    A1,B5,B8        ; smpy(x[-5],a[6])
||      SADD    .L1     A8,A9,A9        ; s0 = sadd(s0, smpy(x[-4],a[4]))
||      SADD    .L2     B8,B9,B9        ; s1 = sadd(s1, smpy(x[-3],a[4]))
||      LDW     .D1     *A4--,A3        ;* load x[0] & x[1] for the next iteration

        SMPYHL  .M1X    A3,B6,A8        ; smpy(x[-7],a[7])
||      SMPY    .M2X    A1,B6,B8        ; smpy(x[-6],a[7])
||      SADD    .L1     A8,A9,A9        ; s0 = sadd(s0, smpy(x[-5],a[5]))
||      SADD    .L2     B8,B9,B9        ; s1 = sadd(s1, smpy(x[-4],a[5]))
||      LDW     .D1     *A4--,A1        ;* load x[-1] & x[-2]
```

Example A–34. Assembly Code for residu.c (Continued)

```
        SMPYLH  .M1X    A3,B6,A8         ; smpy(x[-8],a[8])
||      SMPYH   .M2X    A3,B6,B8         ; smpy(x[-7],a[8])
||      SADD    .L1     A8,A9,A9         ; s0 = sadd(s0, smpy(x[-6],a[6]))
||      SADD    .L2     B8,B9,B9         ; s1 = sadd(s1, smpy(x[-5],a[6]))
||[!A2] STH     .D1     B10,*A6++        ; store y[1]

        SMPYHL  .M1X    A1,B7,A8         ; smpy(x[-9],a[9])
||      SMPY    .M2X    A3,B7,B8         ; smpy(x[-8],a[9])
||      SADD    .L1     A8,A9,A9         ; s0 = sadd(s0, smpy(x[-7],a[7]))
||      SADD    .L2     B8,B9,B9         ; s1 = sadd(s1, smpy(x[-6],a[7]))
||[A2]  SUB     .S2     A2,1,A2
||      LDW     .D1     *A4--,A3         ;* load x[-3] & x[-4]


        SMPYLH  .M1X    A1,B7,A8         ; smpy(x[-10],a[10])
||      SMPYH   .M2X    A1,B7,B8         ; smpy(x[-9],a[10])
||      SADD    .L1     A8,A9,A9         ; s0 = sadd(s0, smpy(x[-8],a[8]))
||      SADD    .L2     B8,B9,B9         ; s1 = sadd(s1, smpy(x[-7],a[8]))
```

There is no memory bank hit within the loop. To avoid a memory bank hit within the prolog of the loop, arrays *a* and *x* must be allocated so that a[1] and x[0] are offset from each other by one word. Some of the instructions in the loop cannot be executed in the first iteration. Register A2 indicates which instructions these are.

A.2.7 Implementation of the Lag Search in the lag_max ( ) Routine

The lag_max ( ) routine performs an open-loop pitch (or lag) search and computes the normalized correlation for the selected lag. This section illustrates the implementation of the lag search. The lag search C code is shown in Example A–35.

Part IV

Example A–35. C Code for the Lag Search in lag_max()

```
#define Word16  short
#define Word32  int
#define MIN_32  0x80000000L
#define PIT_MAX 143
#define L_FRAME 160


input:
        Word16 scal_sig[PIT_MAX+L_FRAME]; (pointed at scal_sig[PIT_MAX] when passed)
        Word16 scal_fac; (not used in this part of the code)
        Word16 L_frame, lag_min, lag_max;


local variables:
        Word16 i, j, *p, *p1, p_max;
        Word32 t0, max;


return:
        Word16 p_max;


Original C code
------------------------------------------------------
    max = MIN_32;


    for (i = lag_max; i >= lag_min; i--)
    {
        p = scal_sig;
        p1 = &scal_sig[-i];
        t0 = 0;


        for (j = 0; j < L_frame; j++, p++, p1++)
        {
            t0 = L_mac (t0, *p, *p1);
        }
        if (L_sub (t0, max) >= 0)
        {
            max = t0;
            p_max = i;
        }
    }
-------------------------------------------------------
where   L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        L_sub(a,b) = _ssub(a,b)
        L_frame = L_FRAME/2 = 80
   and the search range (lag_min, lag_max) is (18,35), (36,71), or (72,143).
```

Part IV

A.2.7.1 Rearranging The C Code and Unrolling The Loops

This algorithm is preferable to smaller lag candidates, because it performs a comparison with if(L_sub (t0,max) >= 0) and the search starts from lag_max. Because there is not a single instruction for the >= (or <=) comparison, you can change the search order to start from lag_min to compare with if (t0 > max); p_max is initialized to lag_min. The C code is modified as shown in Example A–36.

Example A–36. C Code for the Lag Search in lag_max ( ) (Comparison Order Changed)

```
max = MIN_32;
p_max = lag_min;
for (i = lag_min; i < lag_max; i++)
{
  p  = scal_sig;
  p1 = &scal_sig[-i];
  t0 = 0;

  for (j=0; j<L_frame; j++, *p++, *p1++) {
    t0 = L_mac(t0, *p, *p1);
  }
  if (t0 > max)
  {
    max = t0;
    p_max = i;
  }
}
```

Next, look at the inner loop, a general MAC loop. Because *p does not always equal *p1, it does not fall into the special case described in section A.2.1, *Implementation of the Multiply-Accumulate Loop,* beginning on page A-4. Therefore, the performance cannot be improved by simply unrolling the inner loop.

Now consider unrolling the outer loop once. The C code with outer loop unrolling is shown in Example A–37. Because the number of lags that needs to be searched within each search range is always even, such unrolling does not create an additional case to handle.

Example A–37. C Code for the Lag Search in lag_max() With Outer Loop Unrolling

```
Word32  t1;


  max = MIN_32;
  p_max = lag_min;
  for (i = lag_min; i < lag_max; i+=2)
  {
    p = scal_sig;
    p1 = scal_sig[-i];
    t0 = 0;
    t1 = 0;
    for (j=0; j<L_frame; j++, p++, p1++) {
      t1=_sadd(t1,_smpy(*p,*-p1)); (or t1=_sadd(t1,_smpy(scal_sig[j],scal_sig[-i-1+j]))
      t0=_sadd(t0,_smpy(*p,*p1));  (or t0=_sadd(t0,_smpy(scal_sig[j],scal_sig[-i+j]))
    }
    if (t0 > max)
    {
      max = t0;
      p_max = i;
    }
    if( t1 > max)
    {
      max = t1;
      p_max = i+1;
    }
  }


with intrinsics substitutes.
```

The smaller lag is always compared first in the order of the comparisons.

The instructions required for one iteration of the inner loop are shown in Example A–38.

Example A–38. Linear Assembly for the Lag Search in lag_max() Inner Loop

```
INNERLOOP:
        LDH    .D    *p++, sigj             ; load scal_sig[j]
        LDH    .D    *-p1, scalij1          ; load scal_sig[-i-1+j]
        SMPY   .M    sigj,scalij1,tmp1      ; smpy(scal_sig[j],scal_sig[-i-1+j])
        SADD   .L    t1,tmp1,t1             ; t1=sadd(t1,smpy(scal_sig[j],scal_sig[-i-1+j])
        LDH    .D    *p1++,scalij           ; load scal_sig[-i+j]
        SMPY   .M    sigj,scalij,tmp0       ; smpy(scal_sig[j],scal_sig[-i+j])
        SADD   .L    t0,tmp0,t0             ; t0=sadd(t0,smpy(scal_sig[j],scal_sig[-i+j])
[icntr] SUB    .S    icntr,1,icntr          ; decrement inner loop counter
[icntr] B      .S    INNERLOOP              ; branch to inner loop
```

The .D unit is used the most (three times). Therefore, the inner loop takes two cycles.

Now unroll the inner loop once. The first iteration of t1 and the last iteration of t0 perform outside the inner loop. This avoids memory bank hits. The C code with the inner and outer loops unrolled is shown in Example A–39.

Example A–39. C Code for the Lag Search in lag_max() With Inner and Outer Loops Unrolled

```
Word32  t1;

  max = MIN_32;
  p_max = lag_min;
  for (i = lag_min; i < lag_max; i+=2)
  {
      p = scal_sig;
      p1 = scal_sig[-i];
      t0 = 0;
      t1=_sadd(t1,_smpy(*p,*-p1)); (or t1=_sadd(t1,_smpy(scal_sig[j],scal_sig[-i-1+j]))
  for (j=0; j<(L_frame-1); j+=2, p+=2, p1+=2) {
      t0=_sadd(t0,_smpy(*p,*p1)); (or t0=_sadd(t0,_smpy(scal_sig[j],scal_sig[-i+j]))
      t1=_sadd(t1,_smpy(*+p,*p1)); (or t1=_sadd(t1,_smpy(scal_sig[j+1],scal_sig[-i+j]))
      t0=_sadd(t0,_smpy(*+p,*+p1)); (or t0=_sadd(t0,_smpy(scal_sig[j+1],scal_sig[-i+j+1]))
      t1=_sadd(t1,_smpy(*+p[2],*+p1)); (or t1=_sadd(t1,_smpy(scal_sig[j+2],scal_sig[-i+j+1]))
  }
   t0=_sadd(t0,_smpy(scal_sig[L_frame-1],scal_sig[-i+L_frame-1]));
   if (t0 > max)   {
     max = t0;
     p_max = i;
   }
   if( t1 > max)   {
     max = t1;
     p_max = i+1;
   }
 }
```

Although five values of scal_sig, scal_sig[j], scal_sig[j+1], scal_sig[j+2], scal_sig[–i+j], and scal_sig[–i+j+1], are required for each inner loop iteration, scal_sig[j] does not need to be loaded, because it was loaded in the previous iteration. This means only four loads are required per iteration. Example A–40 gives the instructions for the modified inner loop.

Part IV

Example A–40. Linear Assembly for the Lag Search in lag_max() Inner Loop

```
       LDH   .D   *p++, sigj              ; load scal_sig[j]
       LDH   .D   *–p1, scalij1           ; load scal_sig[–i–1+j]
       SMPY  .M   sigj, scalij1,t1        ; t1=smpy(scal_sig[j],scal_sig[–i–1+j])

INNERLOOP:
       LDH   .D   *p1++, scalij           ; load scal_sig[–i+j]
       SMPY  .M   sigj,scalij,tmp0        ; smpy(scal_sig[j],scal_sig[–i+j])
       SADD  .L   t0,tmp0,t0              ; t0=sadd(t0,smpy(scal_sig[j],scal_sig[–i+j])
       LDH   .D   *p++, sigj+1            ; load scal_sig[j+1]
       SMPY  .M   sigj+1,scalij,tmp1      ; smpy(scal_sig[j+1],scal_sig[–i+j])
       SADD  .L   t1,tmp1,t1              ; t1=sadd(t1,smpy(scal_sig[j+1],scal_sig[–i+j])
       LDH   .D   *p1++,scalij+1          ; load scal_sig[–i+j+1]
       SMPY  .M   sigj+1,scalij+1,tmp0    ; smpy(scal_sig[j+1],scal_sig[–i+j+1])
       SADD  .L   t0,tmp0,t0              ; t0=sadd(t0,smpy(scal_sig[j+1],scal_sig[–i+j+1])
       LDH   .D   *p++, sigj+2            ; load scal_sig[j+2], the scal_sig[j] for the
                                          ; next iteration
       SMPY  .M   sigj+2,scalij+1,tmp1    ; smpy(scal_sig[j+2],scal_sig[–i+j+1])
       SADD  .L   t1,tmp1,t1              ; t1=sadd(t1,smpy(scal_sig[j+2],scal_sig[–i+j+1])
[icntr] SUB   .S   icntr,2,icnt            ; decrement inner loop counter
[icntr] B     .S   INNERLOOP               ; branch to inner loop
```

The inner loop uses two cycles. You double the performance, therefore, by unrolling both the outer loop and inner loop if no memory bank hits occur.

A.2.7.2 Avoiding Memory Bank Hits

Load scal_sig[–i+j] and scal_sig[j+1] together and scal_sig[–i+j+1] and scal_sig[j+2] together to avoid memory bank hits. Memory bank hits can also be avoided by loading scal_sig[–i+j] and scal_sig[–i+j+1] together and scal_sig[j+1] and scal_sig[j+2] together.

A.2.7.3 Final Assembly Code for Lag Search

The final assembly code for the lag search segment is shown in Example A–41.

Example A–41. Assembly Code for the Lag Search in lag_max()

```
*****************************************************************************************
**                                                                                     **
**      Implementation of residu.c EFR                                                 **
**                                                                                     **
**      Compare two lags a time                                                        **
**                                                                                     **
**      Total cycles = 7+(L_frame+6)*(lag_max-lag_min+1)/2                             **
**                                                                                     **
**      Register Usage:         A               B                                      **
**                              10              9                                      **
**                                                                                     **
*****************************************************************************************
                        ; A4 --- &scal_sig
                        ; A6 --- lag_max
                        ; B6 --- lag_min

        SUBAH   .D1     A4,A6,A7        ; p1=&scal_sig[-LAG_MIN]
||      MVK     .S2     1,B2
||      SUB     .L1X    B6,A6,A1        ; the outer loop counter
||      MV      .L2X    A4,B7           ; p=&scal_sig[0]
||      MPY     .M2     B0,0,B0         ; initialize the comparison result
||      MPY     .M1     A2,0,A2         ; take care the initial iteration
||      MV      .S1     A6,A4           ; p_max = lag_min

        SHL     .S2     B2,31,B2        ; max=MIN_32=0x80000000L
||      LDH     .D1     *-A7[1],A5      ; scal_sig[-LAG_MIN-1]
||      LDH     .D2     *B7,B5          ; scal_sig[0]
||      ADD     .L1     A1,1,A1         ; make the counter to be an even number

OUTERLOOP:

        LDH     .D1     *A7,A5          ; scal_sig[-LAG_MIN]
||      LDH     .D2     *+B7[1],B6      ; scal_sig[1]
||[A2]  SADD    .L2     B10,B8,B10
||[A1]  MV      .S2     37,B1           ; inner loop counter
||      MPY     .M1     A3,0,A3
||      MPY     .M2     B8,0,B8
||      ADD     .S1     A7,2,A9         ; &scal_sig[-LAG_MIN+1]
||      SUB     .L1     A7,4,A7         ; update p1 = &scal_sig[-LAG_MIN-2]

        LDH     .D1     *A9++,A5        ; scal_sig[-LAG_MIN+1]
||      LDH     .D2     *+B7[2],B5      ; scal_sig[2]
||[B1]  B       .S      INNERLOOP       ; branch to the inner loop
||[A2]  CMPGT   .L2     B10,B2,B0       ; if(t0>max)

        LDH     .D1     *A9++,A5        ; scal_sig[-LAG_MIN+2]
||      LDH     .D2     *+B7[3],B6      ; scal_sig[3]
||[B0]  MV      .L2     B10,B2          ; max = t0
||      MPY     .M1X    B1,1,A2         ; counter to branch to the outerloop
```

Part IV

Example A–41. Assembly Code for the Lag Search in lag_max() (Continued)

```
       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+3]
||     LDH    .D2    *+B7[4],B5  ; scal_sig[4]
||[B1] B      .S2    INNERLOOP   ; branch to the inner loop
||[A2] CMPGT  .L2X   A0,B2,B0    ; if(t1>max)
||[B0] SUB    .L1    A6,2,A4     ; p_max = i
||     ADD    .S1    A6,2,A6     ; update i
||     MPY    .M1    A0,0,A0     ; initialize t1=0
||     MPY    .M2    B10,0,B10   ; initialize t0=0

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+4]
||     LDH    .D2    *+B7[5],B6  ; scal_sig[5]
||     SMPY   .M1X   A5,B5,A3    ; _smpy(scal_sig[-LAG_MIN-1], scal_sig[0])
||[B0] MV     .L2X   A0,B2       ; max = t1
||[B0] SUB    .L1    A6,3,A4     ; p_max = i+1
||[A1] SUB    .S1    A1,2,A1     ; update inner loop counter
||     ADD    .S2    B7,12,B9    ; &scal_sig[1]

 INNERLOOP:

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+5]
||     LDH    .D2    *B9++,B5    ; scal_sig[6]
||     SMPY   .M1X   A5,B6,A3    ; _smpy(scal_sig[-LAG_MIN], scal_sig[1])
||     SMPY   .M2X   A5,B5,B8    ; _smpy(scal_sig[-LAG_MIN], scal_sig[0])
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||[B1] B      .S1    INNERLOOP   ; branch to inner loop
||[B1] SUB    .S2    B1,1,B1     ; decrement inner loop counter

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+6]
||     LDH    .D2    *B9++,B6    ; scal_sig[7]
||     SMPY   .M1X   A5,B5,A3    ; _smpy(scal_sig[-LAG_MIN+1], scal_sig[2])
||     SMPY   .M2X   A5,B6,B8    ; _smpy(scal_sig[-LAG_MIN+1], scal_sig[1])
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||     SUB    .S1    A2,1,A2     ; decrement the counter to branch to the outer loop
||[!A2]B      .S2    OUTERLOOP   ; branch to the outer loop

       LDH    .D1    *-A7[1],A5  ; scal_sig[-LAG_MIN-3]
||     LDH    .D2    *B7,B5      ; scal_sig[0]
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||[!A1]B      .S1    FINISH      ; lag search is complete

FINISH:

       NOP    5
```

All the epilogs and prologs of the outer and inner loops are compressed to minimize the code size. A2 is both the indicator for avoiding comparisons during the initial iteration of the outer loop and the counter for branching to the outer loop during inner loop executions.

Part IV

Index

[ ] in assembly code 5-3

@ symbol in assembly output 2-14

|| (parallel bars) in assembly code 5-2

_ (underscore) in intrinsics 4-9

A

_add2 intrinsic 4-14
  tutorial 2-18

aliasing 4-6

allocating resources
  conflicts 6-61
  dot product 6-19
  if-then-else 6-86, 6-93
  IIR filter 6-78
  in writing parallel code 6-6
  live-too-long resolution 6-102
  weighted vector sum 6-58

AND instruction, mask for 6-70

arrays, controlling alignment 6-116

assembler directives 5-4

assembly code
  comments in 5-9
  conditions in 5-3
  directives in 5-4
  dot product, fixed-point
    nonparallel 6-10
    parallel 6-11
  final
    autocorr.c, windowing and scaling part A-17 to A-20
    dot product, fixed-point 6-22, 6-42, 6-48, 6-51
    dot product, floating-point 6-44, 6-49, 6-52
    FIR filter 6-116, 6-125, 6-129 to 6-132, 6-143 to 6-146
    FIR filter with redundant load elimination 6-112

assembly code (continued)
  final
    if-then-else 6-87, 6-88, 6-95
    IIR filter 6-81
    index search in search_10i40 A-43, A-44 to A-50
    live-too-long, with move instructions 6-104
    MAC loop for energy computation A-6
    residu.c A-54 to A-57
    rrv computation A-33 to A-37
    weighted vector sum 6-71
  functional units in 5-6
  instructions in 5-4
  labels in 5-2
  linear
    autocorr.c, one iteration of loop A-9
    dot product, fixed-point 6-5, 6-16, 6-20, 6-26, 6-35
    dot product, floating-point 6-17, 6-21, 6-27, 6-36
    FIR filter 6-108, 6-110, 6-119, 6-121
    FIR filter, outer loop 6-134
    FIR filter, outer loop conditionally executed with inner loop 6-137, 6-139
    FIR filter, unrolled 6-133
    if-then-else 6-83, 6-86, 6-91, 6-94
    IIR filter 6-74, 6-78
    index search in search_10i40 A-41
    lag search in lag_max( ) A-59
    live-too-long 6-98, 6-103
    MAC loop A-4
    rrv computation in search_10i40 A-28, A-31
    special MAC loop A-5
    weighted vector sum 6-54, 6-56, 6-58
  mnemonics in 5-4
  operands in 5-8
  optimizing (phase 3 of flow), description 6-2
  parallel bars in 5-2
  structure of 5-1 to 5-11
  writing parallel code 6-4

assembly optimizer
  for dot product 6-37
  tutorial 2-25, 2-28
  using to create optimized loops 6-35
autocorr.c, windowing and scaling part A-7

B

big-endian mode
  and MPY operation 6-17
  runtime support (rts6201e.lib) 2-6
biquad filter
  inner loop kernel
    assembly from C with intrinsics 2-23
    linear assembly 2-29
    original assembly code 2-16
  linear assembly 2-27
  original C code 2-4
  with word instructions and intrinsics 2-20
branch target, for software-pipelined dot product 6-37, 6-39
branching to create if-then-else 6-82
breakpoints 4-3

C

C code
  analyzing performance of 4-2
  autocorr.c A-8, A-16
  basic vector sum 4-5
  copyright for A-3
  cor_h A-20
  dot product 4-16
    fixed-point 6-4, 6-15
    floating-point 6-16
  FIR filter 4-16, 4-25, 6-106, 6-118
    inner loop completely unrolled 4-26
    optimized form 4-17
    unrolled 6-127, 6-132, 6-135
    with redundant load elimination 6-107
  if-then-else 6-82, 6-90
  IIR filter 6-73
  index search in search_10i40 A-38, A-40, A-42
  lag search in lag_max( ) A-57, A-58, A-59, A-60
  live-too-long 6-97
  MAC loop A-4, A-5
  rearranging A-2, A-12, A-51
  refining (phase 2 of flow), in flow diagram 1-3
  residu.c A-51, A-52, A-53
C code (continued)
  rrv computation in search_10i40 A-27, A-30
  saturated add 4-9
  trip counters 4-21
  vector sum
    with const keywords 4-7
    with const keywords, _nassert 4-22
    with const keywords, _nassert, word reads 4-14, 4-15
    with const keywords, _nassert, word reads, unrolled 4-24
    with three memory operations 4-23
    word-aligned 4-23
  weighted vector sum 6-54
    unrolled version 6-55
  writing 4-2
C_OPTIONS environment variable 2-6
’C6x mnemonics 5-5
char data type 4-2
child node 6-6
cl6x command 2-5, 4-4
clk register 4-3
clock ( ) function 2-12, 4-2
code development flow diagram 1-3
  phase 1: develop C code 1-3, 2-14 to 2-16
  phase 2: refine C code 1-3, 2-17 to 2-24
  phase 3: write linear assembly 1-3, 2-25 to 2-30
code development steps 3-2
code documentation 5-9
comments in assembly code 5-9
compiler options
  –ms 4-22
  –o2 4-27
  –o3 4-22, 4-27
  –pm 4-22
conditional break 4-27
conditional execution of outer loop with inner loop 6-134
conditional instructions to execute if-then-else 6-83
conditional SUB instruction 6-25
conditions in assembly code 5-3
const keyword 4-5, 4-6
  in vector sum 4-14
constant operands 5-8
cor_h, implementing A-20
.cproc directive 2-25
CPU elements 1-2

cycle count
for biquad filter 2-29
for functions in demo1.c 2-11
for multiply accumulate 2-11
for vector multiply 2-22
formula for calculating 2-11

D

data types 4-2
demo1.c example code 2-3
demo2.c example code 2-21
demo3.c example code 2-28
dependency graph
dot product, fixed-point 6-7
parallel execution 6-11
with LDW 6-18, 6-20, 6-26
dot product, floating-point, with LDW 6-19, 6-21, 6-27
drawing 6-6
steps in 6-7
FIR filter
with arrays aligned on same loop cycle 6-117
with no memory hits 6-120
with redundant load elimination 6-109
if-then-else 6-84, 6-92
IIR filter 6-75, 6-77
live-too-long code 6-99, 6-102
showing resource conflict 6-61
resolved 6-64
vector sum 4-6
weighted 6-57, 6-61, 6-64, 6-66
with const keywords 4-7
weighted vector sum 6-64
destination operand 5-8
dot product
C code 6-4
fixed-point 6-4
translated to linear assembly, fixed-point 6-5
with intrinsics 4-16
dependency graph of basic 6-7
fixed-point
assembly code with LDW before software pipelining 6-22
assembly code with no extraneous loads 6-42
assembly code with no prolog or epilog 6-48

dot product (continued)
fixed-point
assembly code with smallest code size 6-51
assembly code, fully pipelined 6-38
assembly code, nonparallel 6-10
C code with loop unrolling 6-15
dependency graph of parallel assembly code 6-11
dependency graph with LDW 6-20
fully pipelined 6-37
linear assembly for full code 6-35
linear assembly for inner loop with LDW 6-16
linear assembly for inner loop with LDW and allocated resources 6-20
linear assembly for inner loop with conditional SUB instruction 6-26
nonparallel assembly code 6-10
parallel assembly code 6-11
floating-point
assembly code with LDW before software pipelining 6-23
assembly code with no extraneous loads 6-44
assembly code with no prolog or epilog 6-49
assembly code with smallest code size 6-52
assembly code, fully pipelined 6-39
C code with loop unrolling 6-16
linear assembly for inner loop with LDW 6-17
linear assembly for inner loop with LDW and allocated resources 6-21
linear assembly for inner loop with conditional SUB instruction 6-27
fully pipelined 6-39
linear assembly for full code 6-36
word accesses in 4-15
double data type 4-2

E

.endproc directive 2-25
energy computation in MAC loop A-6 to A-8
enhanced full rate (EFR) A-3
epilog 4-20
execute packet 2-11, 2-15, 6-36
execution cycles, reducing number of 6-4
extraneous instructions, removing 6-41
SUB instruction 6-51

F

feedback, from compiler or assembly optimizer 3-3
File menu (debugger) 2-8
FIR filter
  C code 4-16, 6-106
    optimized form 4-17
    unrolled 6-132, 6-135
    with inner loop unrolled 6-127
    with redundant load elimination 6-107
  final assembly 6-143
    for inner loop 6-116
    with redundant load elimination 6-112
    with redundant load elimination, no memory hits 6-125
    with redundant load elimination, no memory hits, outer loop software-pipelined 6-129
  linear assembly
    for inner loop 6-108
    for outer loop 6-134
    for unrolled inner loop 6-119
    for unrolled inner loop with .mptr directive 6-121
    with inner loop unrolled 6-133
    with outer loop conditionally executed with inner loop 6-137, 6-139
  software pipelining the outer loop 6-127
  using word access in 4-16
  with inner loop unrolled 6-118
fixed-point, dot product
  linear assembly for inner loop with LDW 6-16
  linear assembly for inner loop with LDW and allocated resources 6-20
float data type 4-2
floating-point, dot product
  dependency graph with LDW 6-21
  linear assembly for inner loop with LDDW 6-17
  linear assembly for inner loop with LDDW with allocated resources 6-21
flow diagram
  autocorr.c A-9, A-12, A-13
  code development 1-3
functional units
  description 5-7
  in assembly code 5-7
  reassigning for parallel execution 6-10, 6-12
functions
  clock ( ) 4-2
  printf ( ) 4-2

G

–g option 2-5
global constants/symbols defined in EFR A-3
global systems for mobile communications (GSM) A-3

I

if-then-else
  branching versus conditional instructions 6-82
  C code 6-82, 6-90
  final assembly 6-87, 6-88, 6-95
  linear assembly 6-83, 6-86, 6-91, 6-94
IIR filter, C code 6-73
iir1.asm, inner loop kernel 2-16
iir1.c example code 2-4
in-flight value 7-3
index search in search_10i40 A-38
information elements in tutorial 2-2
inserting moves 6-101
instructions, placement in assembly code 5-4
int data type 4-2
interrupt subroutines 7-8 to 7-10
  hand-coded assembly allowing nested interrupts 7-10
  nested interrupts 7-9
  with hand-coded assembly 7-9
  with the C compiler 7-8
interruptible
  code generation 7-6 to 7-7
  loops 7-5
interrupts
  overview 7-2
  single assignment versus multiple assignment 7-3 to 7-4
intrinsics
  _add2 ( ) 4-14
  _mpy ( ) 4-15
  _mpyh ( ) 4-15
  _mpyhl ( ) 4-14
  _mpylh ( ) 4-14
  _nassert 4-22
  described 2-18, 4-9
  in residu.c A-51 to A-53
  in saturated add 4-9
  summary table 4-10 to 4-12
iteration interval, defined 6-28

K

–k compiler option 2-5, 4-4
kernel
  loop 2-14, 4-7, 4-20
  of iir1.asm code 2-16
  of iir2.asm code 2-23
  of iir3.asm code 2-29
  of mac1.asm code 2-14
  of vec_mpy1.asm code 2-15
  of vec_mpy2.asm code 2-22

L

–l linker option 2-6
labels in assembly code 5-2
lag search in lag_max ( ) A-56
linear, optimizing (phase 3 of flow), in flow diagram 1-3
linear assembly 2-25
  code
    autocorr.c, one iteration of loop A-9
    dot product, fixed-point 6-5
    dot product, fixed-point 6-10, 6-16, 6-20, 6-26, 6-35
    dot product, floating-point 6-17, 6-21, 6-27, 6-36
    FIR filter 6-108, 6-110, 6-119, 6-121
    FIR filter with outer loop conditionally executed with inner loop 6-137, 6-139
    FIR filter, outer loop 6-134
    FIR filter, unrolled 6-133
    if-then-else 6-86, 6-94
    index search in search_10i40 A-41
    lag search in lag_max( ) A-59
    live-too-long 6-103
    MAC loop A-4
    rrv computation in search_10i40 A-28, A-31
    special MAC loop A-5
    weighted vector sum 6-58
  resource allocation
    conflicts 6-61
    dot product 6-19
    if-then-else 6-86, 6-93
    IIR filter 6-78
    in writing parallel code 6-6
    live-too-long resolution 6-102
    weighted vector sum 6-58
linker command file 2-6
little-endian mode
  and MPY operation 6-17
  runtime support (rts6201.lib) 2-6
live-too-long
  code 6-63
    C code 6-97
    inserting move (MV) instructions 6-101
    unrolling the loop 6-101
  issues 6-97
    and software pipelining 4-27
    created by split-join paths 6-100
load
  doubleword (LDDW) instruction 6-15
  word (LDW) instruction 6-15
Load Program File dialog box (debugger) 2-8
load6x 2-12, 2-13
long data type 4-2
loop
  carry path, described 6-73
  control variable, conditionally incremented 4-27
  counter, handling odd-numbered 4-15
  interruptible 7-5
  iterations 4-21
  kernel 2-14
  unrolling
    as major programming method A-2
    dot product 6-15
    for simple loop structure 4-25
    for windowing and scaling in autocorr.c A-9
    if-then-else code 6-90
    in cor_h A-22
    in FIR filter 6-118, 6-121, 6-127, 6-132, 6-134
    in lag_max A-58
    in live-too-long solution 6-101
    in vector sum 4-23

M

mac1.asm kernel, inner loop 2-14
mac1.c example code 2-3
memory bank hits
  avoiding A-2
    cor_h A-23
  in windowing and scaling in autocorr.c A-15
memory bank scheme, interleaved 6-114 to 6-116
–mg compiler option 2-5

minimum iteration interval, determining 6-30
for FIR code 6-110, 6-124, 6-142
for if-then-else code 6-85, 6-93
for IIR code 6-76
for live-too-long code 6-100
for weighted vector sum 6-55, 6-56
mnemonic (instruction) 5-4
modulo iteration interval table
dot product, fixed-point
after software pipelining 6-31
before software pipelining 6-28
dot product, floating-point
after software pipelining 6-32
before software pipelining 6-29
IIR filter, 4-cycle loop 6-79
weighted vector sum
2-cycle loop 6-60, 6-65, 6-68
with SHR instructions 6-62
modulo-scheduling technique, multicycle loops 6-54
move (MV) instruction 6-101
_mpy intrinsic 4-15
tutorial 2-18
_mpyh ( ) intrinsic 4-15
_mpyhl intrinsic 4-14
_mpylh intrinsic 4-14
tutorial 2-18
multicycle instruction, staggered accumulation 6-33
multiple assignment, code example 7-3
multiply accumulate function
inner loop kernel of original assembly code 2-14
original C code 2-3
multiply-accumulate loop (MAC), implementation in vocoder application A-4
–mw compiler option 3-3

N

_nassert intrinsic 4-12, 4-14, 4-22
node 6-6

O

–o compiler option 2-5, 4-4, 4-20, 4-22, 4-27
–o linker option 2-6
operands
placement in assembly code 5-8
types of 5-8
optimization checklist 3-1 to 3-5
optimizing assembly code, introduction 6-2
optional tasks in tutorial 2-2
outer loop conditionally executed with inner loop 6-132
OUTLOOP 6-111, 6-124

P

parallel bars, in assembly code 5-2
parent instruction 6-6
parent node 6-6
path in dependency graph 6-6
performance analysis
index search in search_10i40 A-40
of C code 4-2
of dot product examples 6-14, 6-24, 6-53
of FIR filter code 6-124, 6-131, 6-145
of if-then-else code 6-89, 6-96
residu.c A-52
pipeline in ’C6x 1-2
–pm compiler option 4-4, 4-5, 4-8, 4-22
pointer operands 5-8
preparation for tutorial 2-1
primary tasks in tutorial 2-2
priming the loop, described 6-47
priming the pipeline 4-21
printf ( ) function 4-2
processor mnemonics 5-5
Profile
Marking dialog box 2-9
menu (debugger) 2-8
Run dialog box 2-10
profiling 2-8 to 2-13
program-level optimization 4-5
programming methods, summary of A-2
prolog 4-20, 6-47, 6-49
pseudo-code, for single-cycle accumulator with ADDSP 6-33

R

redundant
- load elimination 6-106
- loops 4-22

.reg directive 2-25, 6-16, 6-17

register
- allocation 6-123
- operands 5-8
- partitioning A-41

residu.c (FIR filter in EFR) A-51

resource
- conflicts
  - *described 6-61*
  - *live-too-long issues 6-63, 6-97*
- table
  - *FIR filter code 6-110, 6-124, 6-142*
  - *if-then-else code 6-85, 6-93*
  - *IIR filter code 6-76*
  - *live-too-long code 6-100*

routines
- autocorr.c A-7
- cor_h A-20
- lag_max ( ) A-56

rrv computation in search_10i40 A-27

rts6201.lib file 2-6

rts6201e.lib file 2-6

RUNB debugger command 4-3

S

.sa extension 2-25

_sadd intrinsic 4-9, 4-12

scheduling table. *See* modulo iteration interval table

shell program (cl6x) 2-5, 4-4

short
- arrays 4-14
- data type 4-2, 4-14

single assignment, code example 7-4

software pipeline 4-20, 4-24
- accumulation, staggered results due to 3-cycle delay 6-34
- described 6-25
- when not used 4-26

software-pipelined schedule, creating 6-30

source operands 5-8

split-join path 6-97, 6-98, 6-100

stand-alone simulator (load6x) 2-12, 4-2

SunOS shell initialization 2-7

symbolic names, for data and pointers 6-16, 6-17

T

techniques
- for priming the loop 6-47
- for refining C code 4-9
- for removing extra instructions 6-41, 6-51
- using intrinsics 4-9
- word access for short data 4-14

TMS320C6x pipeline 1-2

translating C code to ’C6x instructions
- dot product
  - *fixed-point, unrolled 6-16*
  - *floating-point, unrolled 6-17*
- IIR filter 6-74
  - *with reduced loop carry path 6-78*
- weighted vector sum 6-54
  - *unrolled inner loop 6-56*

translating C code to linear assembly, dot product, fixed-point 6-5

trip count 2-25, 4-21
- communicating information to the compiler 4-22
- determining the minimum 4-21

trip counter
- converting to a downcounting loop 4-27
- defined 4-21

.trip directive 2-25

V

vec_mpy1.asm kernel, inner loop 2-15

vec_mpy1.c example code 2-4

vector multiply function
- C with word instructions and intrinsics 2-18
- inner loop kernel
  - *of assembly from C with intrinsics 2-22*
  - *of original assembly code 2-15*
- original C code 2-4
- tutorial C code example (vec_mpy1.c) 2-4

vector sum function
- *See also* weighted vector sum
- C code 4-5
  - *with const keyword 4-7*
  - *with const keywords and _nassert 4-22*
  - *with const keywords, _nassert, word reads 4-14*

vector sum function (continued)
C code
with const keywords, _nassert, word reads, and loop unrolling 4-24
with const keywords,_nassert, and word reads (generic) 4-15
with three memory operations 4-23
word-aligned 4-23
compiler output (original assembly code) 4-8
dependency graph 4-6, 4-7
handling odd-numbered loop counter with 4-15
handling short-aligned data with 4-15
rewriting to use word accesses 4-14

VelociTI 1-2

very long instruction word (VLIW) 1-2

vocoder
application A-1
implementing A-3

W

weighted vector sum
C code 6-54
unrolled version 6-55
final assembly 6-71
linear assembly 6-69
for inner loop 6-54
with resources allocated 6-58
translating C code to assembly instructions 6-56

windowing and scaling, autocorr.c A-7

word access
in dot product 4-15 to 4-16
in FIR filter 4-16
using for short data 4-14 to 4-19

Z

–z compiler option 2-6