

TMS320C62x/C67x Programmer's Guide

Literature Number: SPRU198C
May 1999



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.

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.

TMS320C6000 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 'C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the 'C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the 'C6000 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.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C6000 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.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced DMA (EDMA), expansion bus, 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		Email: dsph@ti.com
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	Email: dsph@ti.com
Technical Documentation Services, MS 702	
P.O. Box 1443	
Houston, Texas 77251-1443	

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

Contents

1	Introduction	1-1
	<i>Introduces some features of the 'C6x microprocessor and discusses the basic process for creating code.</i>	
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
	<i>Uses example code to walk you through the code development flow for the TMS320C6x.</i>	
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
	2.4.1 Using the Standalone Simulator for Profiling	2-8
	2.4.2 Using the Profiler in the Debugger	2-11
2.5	Lesson 3: Phase 1 of the Code Development Flow	2-15
2.6	Lesson 4: Phase 2 of the Code Development Flow	2-18
2.7	Lesson 5: Phase 3 of the Code Development Flow	2-26
2.8	Summary	2-32
3	TMS320C6x Optimization Checklist	3-1
	<i>Provides a code development flow and checklist for optimizing loops.</i>	
4	Optimizing C Code	4-1
	<i>Explains how to maximize C performance by using compiler options, intrinsics, and code transformations.</i>	
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	4-6
4.3	Refining C Code	4-13
	4.3.1 Using Intrinsics	4-13
	4.3.2 Using Word Access for Short Data	4-18
	4.3.3 Software Pipelining	4-32

5	Linking Issues	5-1
	<i>Explains linker messages and how to use RTS functions.</i>	
5.1	How to Use Linker Error Messages	5-2
5.1.1	Executable Flag	5-4
5.2	How to Save On-Chip Memory by Placing RTS Off-Chip	5-5
5.2.1	How to Compile	5-5
5.2.2	Must #include Header Files	5-6
5.2.3	RTS Data	5-6
5.2.4	How to Link	5-7
5.2.5	Example Compiler Invocation	5-9
5.2.6	Header File Details	5-9
5.2.7	Changing RTS Data to near	5-10
6	Structure of Assembly Code	6-1
	<i>Describes the structure of the assembly code, including labels, conditions, instructions, functional units, operands, and comments.</i>	
6.1	Labels	6-2
6.2	Parallel Bars	6-2
6.3	Conditions	6-3
6.4	Instructions	6-4
6.5	Functional Units	6-6
6.6	Operands	6-8
6.7	Comments	6-9
7	Optimizing Assembly Code via Linear Assembly	7-1
	<i>Describes methods that help you develop more efficient assembly language programs.</i>	
7.1	Assembly Code	7-2
7.2	Assembly Optimizer Options and Directives	7-4
7.2.1	The <code>-mt</code> Option and the <code>.no_mdep</code> Directive	7-4
7.2.2	The <code>.mdep</code> Directive	7-4
7.2.3	The <code>.mptr</code> Directive	7-5
7.2.4	The <code>.trip</code> Directive	7-8
7.3	Writing Parallel Code	7-9
7.3.1	Dot Product C Code	7-9
7.3.2	Translating C Code to Linear Assembly	7-10
7.3.3	Linear Assembly Resource Allocation	7-11
7.3.4	Drawing a Dependency Graph	7-11
7.3.5	Nonparallel Versus Parallel Assembly Code	7-15
7.3.6	Comparing Performance	7-19
7.4	Using Word Access for Short Data and Doubleword Access for Floating-Point Data	7-20
7.4.1	Unrolled Dot Product C Code	7-20
7.4.2	Translating C Code to Linear Assembly	7-21
7.4.3	Drawing a Dependency Graph	7-23

7.4.4	Linear Assembly Resource Allocation	7-24
7.4.5	Final Assembly	7-27
7.4.6	Comparing Performance	7-29
7.5	Software Pipelining	7-30
7.5.1	Modulo Iteration Interval Scheduling	7-33
7.5.2	Using the Assembly Optimizer to Create Optimized Loops	7-40
7.5.3	Final Assembly	7-41
7.5.4	Comparing Performance	7-58
7.6	Modulo Scheduling of Multicycle Loops	7-59
7.6.1	Weighted Vector Sum C Code	7-59
7.6.2	Translating C Code to Linear Assembly	7-59
7.6.3	Determining the Minimum Iteration Interval	7-60
7.6.4	Drawing a Dependency Graph	7-62
7.6.5	Linear Assembly Resource Allocation	7-63
7.6.6	Modulo Iteration Interval Scheduling	7-63
7.6.7	Using the Assembly Optimizer for the Weighted Vector Sum	7-74
7.6.8	Final Assembly	7-75
7.7	Loop Carry Paths	7-78
7.7.1	IIR Filter C Code	7-78
7.7.2	Translating C Code to Linear Assembly (Inner Loop)	7-79
7.7.3	Drawing a Dependency Graph	7-80
7.7.4	Determining the Minimum Iteration Interval	7-81
7.7.5	Linear Assembly Resource Allocation	7-83
7.7.6	Modulo Iteration Interval Scheduling	7-84
7.7.7	Using the Assembly Optimizer for the IIR Filter	7-85
7.7.8	Final Assembly	7-86
7.8	If-Then-Else Statements in a Loop	7-87
7.8.1	If-Then-Else C Code	7-87
7.8.2	Translating C Code to Linear Assembly	7-88
7.8.3	Drawing a Dependency Graph	7-89
7.8.4	Determining the Minimum Iteration Interval	7-90
7.8.5	Linear Assembly Resource Allocation	7-91
7.8.6	Final Assembly	7-92
7.8.7	Comparing Performance	7-93
7.9	Loop Unrolling	7-95
7.9.1	Unrolled If-Then-Else C Code	7-95
7.9.2	Translating C Code to Linear Assembly	7-96
7.9.3	Drawing a Dependency Graph	7-97
7.9.4	Determining the Minimum Iteration Interval	7-98
7.9.5	Linear Assembly Resource Allocation	7-98
7.9.6	Final Assembly	7-100
7.9.7	Comparing Performance	7-101

- 7.10 Live-Too-Long Issues 7-102
 - 7.10.1 C Code With Live-Too-Long Problem 7-102
 - 7.10.2 Translating C Code to Linear Assembly 7-103
 - 7.10.3 Drawing a Dependency Graph 7-103
 - 7.10.4 Determining the Minimum Iteration Interval 7-105
 - 7.10.5 Linear Assembly Resource Allocation 7-107
 - 7.10.6 Final Assembly With Move Instructions 7-109
- 7.11 Redundant Load Elimination 7-111
 - 7.11.1 FIR Filter C Code 7-111
 - 7.11.2 Translating C Code to Linear Assembly 7-113
 - 7.11.3 Drawing a Dependency Graph 7-114
 - 7.11.4 Determining the Minimum Iteration Interval 7-115
 - 7.11.5 Linear Assembly Resource Allocation 7-115
 - 7.11.6 Final Assembly 7-116
- 7.12 Memory Banks 7-119
 - 7.12.1 FIR Filter Inner Loop 7-121
 - 7.12.2 Unrolled FIR Filter C Code 7-123
 - 7.12.3 Translating C Code to Linear Assembly 7-124
 - 7.12.4 Drawing a Dependency Graph 7-125
 - 7.12.5 Linear Assembly for Unrolled FIR Inner Loop With `.mpro` Directive 7-126
 - 7.12.6 Linear Assembly Resource Allocation 7-128
 - 7.12.7 Determining the Minimum Iteration Interval 7-129
 - 7.12.8 Final Assembly 7-129
 - 7.12.9 Comparing Performance 7-129
- 7.13 Software Pipelining the Outer Loop 7-132
 - 7.13.1 Unrolled FIR Filter C Code 7-132
 - 7.13.2 Making the Outer Loop Parallel With the Inner Loop Epilog and Prolog ... 7-133
 - 7.13.3 Final Assembly 7-133
 - 7.13.4 Comparing Performance 7-136
- 7.14 Outer Loop Conditionally Executed With Inner Loop 7-137
 - 7.14.1 Unrolled FIR Filter C Code 7-137
 - 7.14.2 Translating C Code to Linear Assembly (Inner Loop) 7-138
 - 7.14.3 Translating C Code to Linear Assembly (Outer Loop) 7-139
 - 7.14.4 Unrolled FIR Filter C Code 7-139
 - 7.14.5 Translating C Code to Linear Assembly (Inner Loop) 7-141
 - 7.14.6 Translating C Code to Linear Assembly (Inner Loop and Outer Loop) ... 7-143
 - 7.14.7 Determining the Minimum Iteration Interval 7-147
 - 7.14.8 Final Assembly 7-147
 - 7.14.9 Comparing Performance 7-150

8	Interrupts	8-1
	<i>Describes interrupts from a software programming point of view.</i>	
8.1	Overview of Interrupts	8-2
8.2	Single Assignment vs. Multiple Assignment	8-3
8.3	Interruptible Loops	8-5
8.4	Interruptible Code Generation	8-6
8.4.1	Level 0 - Specified Code is Guaranteed to Not Be Interrupted	8-6
8.4.2	Level 1 – Specified Code Interruptible at All Times	8-7
8.4.3	Level 2 – Specified Code Interruptible Within Threshold Cycles	8-7
8.4.4	Getting the Most Performance Out of Interruptible Code	8-8
8.5	Interrupt Subroutines	8-11
8.5.1	ISR with the C Compiler	8-11
8.5.2	ISR with Hand-Coded Assembly	8-12
8.5.3	Nested Interrupts	8-12
A	Memory Alias Disambiguation	A-1
A.1	Overview	A-2
A.2	Background	A-3
A.2.1	Data Dependence Between Instructions	A-3
A.2.2	Dependence Graphs	A-4
A.2.3	Data Dependence in Loops	A-7
A.2.4	How Dependence Affects Instruction Scheduling	A-9
A.2.5	Memory Alias Disambiguation Defined	A-11
A.3	Tools Solution	A-12
A.3.1	Overview of the Assembly Optimizer Solution	A-12
A.3.2	Automatic Disambiguation	A-12
A.3.3	Default Presumption is Pessimistic	A-12
A.3.4	Change the Default Presumption to Optimistic	A-14
A.3.5	Using .mdep to Mark Aliases	A-14
A.4	Examples of Memory Alias Disambiguation	A-15
A.4.1	How .mdep Affects Instruction Scheduling	A-15
A.4.2	Handling Indexed Addressing Mode	A-21
A.4.3	Peripherals Access Example	A-24
A.5	C Compiler and Alias Disambiguation	A-26
A.6	Memory Alias Disambiguation versus Memory Bank Conflict Detection	A-27
A.7	Summary	A-28

Figures

4-1	Dependency Graph for Vector Sum #1	4-7
4-2	Dependency Graph for Vector Sum #2	4-9
4-3	Software-Pipelined Loop	4-32
6-1	Labels in Assembly Code	6-2
6-2	Parallel Bars in Assembly Code	6-2
6-3	Conditions in Assembly Code	6-3
6-4	Instructions in Assembly Code	6-4
6-5	TMS320C6x Functional Units	6-6
6-6	Units in the Assembly Code	6-7
6-7	Operands in the Assembly Code	6-8
6-8	Operands in Instructions	6-8
6-9	Comments in Assembly Code	6-9
7-1	Dependency Graph of Fixed-Point Dot Product	7-12
7-2	Dependency Graph of Floating-Point Dot Product	7-13
7-3	Dependency Graph of Fixed-Point Dot Product with Parallel Assembly	7-16
7-4	Dependency Graph of Floating-Point Dot Product with Parallel Assembly	7-18
7-5	Dependency Graph of Fixed-Point Dot Product With LDW	7-23
7-6	Dependency Graph of Floating-Point Dot Product With LDDW	7-24
7-7	Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)	7-25
7-8	Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)	7-26
7-9	Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)	7-31
7-10	Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)	7-32
7-11	Dependency Graph of Weighted Vector Sum	7-62
7-12	Dependency Graph of Weighted Vector Sum (Showing Resource Conflict)	7-66
7-13	Dependency Graph of Weighted Vector Sum (With Resource Conflict Resolved)	7-69
7-14	Dependency Graph of Weighted Vector Sum (Scheduling $ci + 1$)	7-71
7-15	Dependency Graph of IIR Filter	7-80
7-16	Dependency Graph of IIR Filter (With Smaller Loop Carry)	7-82
7-17	Dependency Graph of If-Then-Else Code	7-89
7-18	Dependency Graph of If-Then-Else Code (Unrolled)	7-97
7-19	Dependency Graph of Live-Too-Long Code	7-104
7-20	Dependency Graph of Live-Too-Long Code (Split-Join Path Resolved)	7-107
7-21	Dependency Graph of FIR Filter (With Redundant Load Elimination)	7-114

7-22	4-Bank Interleaved Memory	7-119
7-23	4-Bank Interleaved Memory With Two Memory Blocks	7-120
7-24	Dependency Graph of FIR Filter (With Even and Odd Elements of Each Array on Same Loop Cycle)	7-122
7-25	Dependency Graph of FIR Filter (With No Memory Hits)	7-125

Tables

2-1	Using the C_OPTIONS Environment Variable	2-7
2-2	Cycle Counts	2-14
2-3	Revised Cycle Counts for vec_mpy()	2-23
2-4	Revised Cycle Counts for iir()	2-24
2-5	Revised Cycle Counts	2-25
2-6	Revised Cycle Counts for iir()	2-30
2-7	Revised Cycle Counts	2-31
3-1	Code Development Steps	3-2
3-2	TMS320C6x Optimization Checklist	3-5
4-1	Subset of Compiler Options	4-4
4-2	TMS320C6x C Compiler Intrinsics	4-14
5-1	Definitions	5-5
5-2	Command Line Options for RTS Calls	5-5
5-3	How _FAR_RTS is Defined in Linkage.h With -mr	5-10
6-1	Selected TMS320C6x Directives	6-4
6-2	Selected TMS320C6x Instruction Mnemonics	6-5
6-3	Functional Units and Descriptions	6-7
7-1	Comparison of Nonparallel and Parallel Assembly Code for Fixed-Point Dot Product	7-19
7-2	Comparison of Nonparallel and Parallel Assembly Code for Floating-Point Dot Product	7-19
7-3	Comparison of Fixed-Point Dot Product Code With Use of LDW	7-29
7-4	Comparison of Floating-Point Dot Product Code With Use of LDDW	7-29
7-5	Modulo Iteration Interval Scheduling Table for Fixed-Point Dot Product (Before Software Pipelining)	7-33
7-6	Modulo Iteration Interval Scheduling Table for Floating-Point Dot Product (Before Software Pipelining)	7-34
7-7	Modulo Iteration Interval Table for Fixed-Point Dot Product (After Software Pipelining)	7-36
7-8	Modulo Iteration Interval Table for Floating-Point Dot Product (After Software Pipelining)	7-37
7-9	Software Pipeline Accumulation Staggered Results Due to Three-Cycle Delay	7-39
7-10	Comparison of Fixed-Point Dot Product Code Examples	7-58
7-11	Comparison of Floating-Point Dot Product Code Examples	7-58
7-12	Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)	7-65
7-13	Modulo Iteration Interval Table for Weighted Vector Sum With SHR Instructions	7-67
7-14	Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)	7-70
7-15	Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)	7-73

7-16	Resource Table for IIR Filter	7-81
7-17	Modulo Iteration Interval Table for IIR (4-Cycle Loop)	7-84
7-18	Resource Table for If-Then-Else Code	7-90
7-19	Comparison of If-Then-Else Code Examples	7-94
7-20	Resource Table for Unrolled If-Then-Else Code	7-98
7-21	Comparison of If-Then-Else Code Examples	7-101
7-22	Resource Table for Live-Too-Long Code	7-105
7-23	Resource Table for FIR Filter Code	7-115
7-24	Resource Table for FIR Filter Code	7-129
7-25	Comparison of FIR Filter Code	7-129
7-26	Comparison of FIR Filter Code	7-136
7-27	Resource Table for FIR Filter Code	7-147
7-28	Comparison of FIR Filter Code	7-150
A-1	Dependence Table	A-3

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)	2-10
2-6	Inner Loop Kernel of mac1.asm	2-15
2-7	Inner Loop Kernel of vec_mpy1.asm	2-16
2-8	Inner Loop Kernel of iir1.asm	2-17
2-9	The Vector Multiply Function—vec_mpy1.c	2-18
2-10	Inner Loop Kernel of vec_mpy1.asm	2-18
2-11	The Revised Vector Multiply Function—vec_mpy2.c	2-19
2-12	The Biquad Filter—iir1.c	2-20
2-13	The Revised Biquad Filter—iir2.c	2-21
2-14	The Revised Example—demo2.c	2-22
2-15	Inner Loop Kernel of vec_mpy2.asm	2-23
2-16	Inner Loop Kernel of iir2.asm	2-24
2-17	The Revised Biquad Filter—iir2.c	2-27
2-18	The Biquad Filter, Revised and Assembly-Optimized—iir3.sa	2-28
2-19	The Revised Example—demo3.c	2-29
2-20	Inner Loop Kernel of iir3.asm	2-30
3-1	Compiler and/or Assembly Optimizer Feedback	3-4
4-1	Basic Vector Sum	4-7
4-2	Vector Sum With const Keywords	4-9
4-3	Compiler Output for Vector Sum Code	4-10
4-4	Incorrect Use of the const Keyword	4-11
4-5	Saturated Add Without Intrinsics	4-13
4-6	Saturated Add With Intrinsics	4-14
4-7	Vector Sum With const Keywords, _nassert, Word Reads	4-18
4-8	Vector Sum With const Keywords, _nassert, Word Reads (Generic Version)	4-19
4-9	Dot Product Using Intrinsics	4-20
4-10	FIR Filter—Original Form	4-20
4-11	FIR Filter—Optimized Form	4-21
4-12	Basic Float Dot Product	4-22
4-13	Float Dot Product Using Intrinsics	4-22
4-14	Float Dot Product With Peak Performance	4-24
4-15	Using the Compiler to Generate a Dot Product With Word Accesses	4-25

4-16	Using the <code>_nassert()</code> Intrinsic to Generate Word Accesses for Vector Sum	4-27
4-17	Using <code>_nassert()</code> Intrinsic to Generate Word Accesses for FIR Filter	4-28
4-18	Automatic Use of Word Accesses Without the <code>_nassert</code> Intrinsic	4-30
4-19	Trip Counters	4-33
4-20	Vector Sum With Three Memory Operations	4-36
4-21	Word-Aligned Vector Sum	4-36
4-22	Vector Sum Using <code>const</code> Keywords, <code>_nassert</code> , Word Reads, and Loop Unrolling	4-37
4-23	FIR_Type2—Original Form	4-38
4-24	FIR_Type2—Inner Loop Completely Unrolled	4-39
4-25	Vector Sum	4-40
4-26	Use of If Statements in Float Collision Detection	4-42
7-1	Linear Assembly Block Copy	7-4
7-2	Block copy With <code>.mdep</code>	7-5
7-3	Linear Assembly Dot Product	7-5
7-4	Linear Assembly Dot Product With <code>.mptr</code>	7-7
7-5	Fixed-Point Dot Product C Code	7-9
7-6	Floating-Point Dot Product C Code	7-9
7-7	List of Assembly Instructions for Fixed-Point Dot Product	7-10
7-8	List of Assembly Instructions for Floating-Point Dot Product	7-10
7-9	Nonparallel Assembly Code for Fixed-Point Dot Product	7-15
7-10	Parallel Assembly Code for Fixed-Point Dot Product	7-16
7-11	Nonparallel Assembly Code for Floating-Point Dot Product	7-17
7-12	Parallel Assembly Code for Floating-Point Dot Product	7-18
7-13	Fixed-Point Dot Product C Code (Unrolled)	7-20
7-14	Floating-Point Dot Product C Code (Unrolled)	7-21
7-15	Linear Assembly for Fixed-Point Dot Product Inner Loop with LDW	7-21
7-16	Linear Assembly for Floating-Point Dot Product Inner Loop with LDDW	7-22
7-17	Linear Assembly for Fixed-Point Dot Product Inner Loop With LDW (With Allocated Resources)	7-25
7-18	Linear Assembly for Floating-Point Dot Product Inner Loop With LDDW (With Allocated Resources)	7-26
7-19	Assembly Code for Fixed-Point Dot Product With LDW (Before Software Pipelining)	7-27
7-20	Assembly Code for Floating-Point Dot Product With LDDW (Before Software Pipelining)	7-28
7-21	Linear Assembly for Fixed-Point Dot Product Inner Loop (With Conditional SUB Instruction)	7-31
7-22	Linear Assembly for Floating-Point Dot Product Inner Loop (With Conditional SUB Instruction)	7-32
7-23	Pseudo-Code for Single-Cycle Accumulator With ADDSP	7-38
7-24	Linear Assembly for Full Fixed-Point Dot Product	7-40
7-25	Linear Assembly for Full Floating-Point Dot Product	7-41
7-26	Assembly Code for Fixed-Point Dot Product (Software Pipelined)	7-43
7-27	Assembly Code for Floating-Point Dot Product (Software Pipelined)	7-44
7-28	Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads)	7-47

7-29	Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads)	7-49
7-30	Assembly Code for Fixed-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog)	7-53
7-31	Assembly Code for Floating-Point Dot Product (Software Pipelined With Removal of Prolog and Epilog)	7-54
7-32	Assembly Code for Fixed-Point Dot Product (Software Pipelined With Smallest Code Size)	7-56
7-33	Assembly Code for Floating-Point Dot Product (Software Pipelined With Smallest Code Size)	7-57
7-34	Weighted Vector Sum C Code	7-59
7-35	Linear Assembly for Weighted Vector Sum Inner Loop	7-59
7-36	Weighted Vector Sum C Code (Unrolled)	7-60
7-37	Linear Assembly for Weighted Vector Sum Using LDW	7-61
7-38	Linear Assembly for Weighted Vector Sum With Resources Allocated	7-63
7-39	Linear Assembly for Weighted Vector Sum	7-74
7-40	Assembly Code for Weighted Vector Sum	7-76
7-41	IIR Filter C Code	7-78
7-42	Linear Assembly for IIR Inner Loop	7-79
7-43	Linear Assembly for IIR Inner Loop With Reduced Loop Carry Path	7-83
7-44	Linear Assembly for IIR Inner Loop (With Allocated Resources)	7-83
7-45	Linear Assembly for IIR Filter	7-85
7-46	Assembly Code for IIR Filter	7-86
7-47	If-Then-Else C Code	7-87
7-48	Linear Assembly for If-Then-Else Inner Loop	7-88
7-49	Linear Assembly for Full If-Then-Else Code	7-91
7-50	Assembly Code for If-Then-Else	7-92
7-51	Assembly Code for If-Then-Else With Loop Count Greater Than 3	7-93
7-52	If-Then-Else C Code (Unrolled)	7-95
7-53	Linear Assembly for Unrolled If-Then-Else Inner Loop	7-96
7-54	Linear Assembly for Full Unrolled If-Then-Else Code	7-99
7-55	Assembly Code for Unrolled If-Then-Else	7-100
7-56	Live-Too-Long C Code	7-102
7-57	Linear Assembly for Live-Too-Long Inner Loop	7-103
7-58	Linear Assembly for Full Live-Too-Long Code	7-108
7-59	Assembly Code for Live-Too-Long With Move Instructions	7-109
7-60	FIR Filter C Code	7-111
7-61	FIR Filter C Code With Redundant Load Elimination	7-112
7-62	Linear Assembly for FIR Inner Loop	7-113
7-63	Linear Assembly for Full FIR Code	7-115
7-64	Final Assembly Code for FIR Filter With Redundant Load Elimination	7-117
7-65	Final Assembly Code for Inner Loop of FIR Filter	7-121
7-66	FIR Filter C Code (Unrolled)	7-123
7-67	Linear Assembly for Unrolled FIR Inner Loop	7-124
7-68	Linear Assembly for Full Unrolled FIR Filter	7-126

7-69	Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits	7-130
7-70	Unrolled FIR Filter C Code	7-132
7-71	Final Assembly Code for FIR Filter With Redundant Load Elimination and No Memory Hits With Outer Loop Software-Pipelined	7-134
7-72	Unrolled FIR Filter C Code	7-137
7-73	Linear Assembly for Unrolled FIR Inner Loop	7-138
7-74	Linear Assembly for FIR Outer Loop	7-139
7-75	Unrolled FIR Filter C Code	7-140
7-76	Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop	7-142
7-77	Linear Assembly for FIR With Outer Loop Conditionally Executed With Inner Loop (With Functional Units)	7-144
7-78	Final Assembly Code for FIR Filter	7-148
8-1	Code With Multiple Assignment of A1	8-3
8-2	Code Using Single Assignment	8-4
8-3	Dot Product With <code>_nassert</code> Guaranteeing Minimum Trip Count	8-8
8-4	Dot Product With <code>_nassert</code> Guaranteeing Trip Count Range	8-9
8-5	Dot Product With <code>_nassert</code> Guaranteeing Trip Count Range and Factor of 2	8-10
8-6	Dot Product With <code>_nassert</code> Guaranteeing Trip Count Range and Factor of 4	8-10
8-7	Hand-Coded Assembly ISR	8-12
8-8	Hand-Coded Assembly ISR Allowing Nesting of Interrupts	8-13

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

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

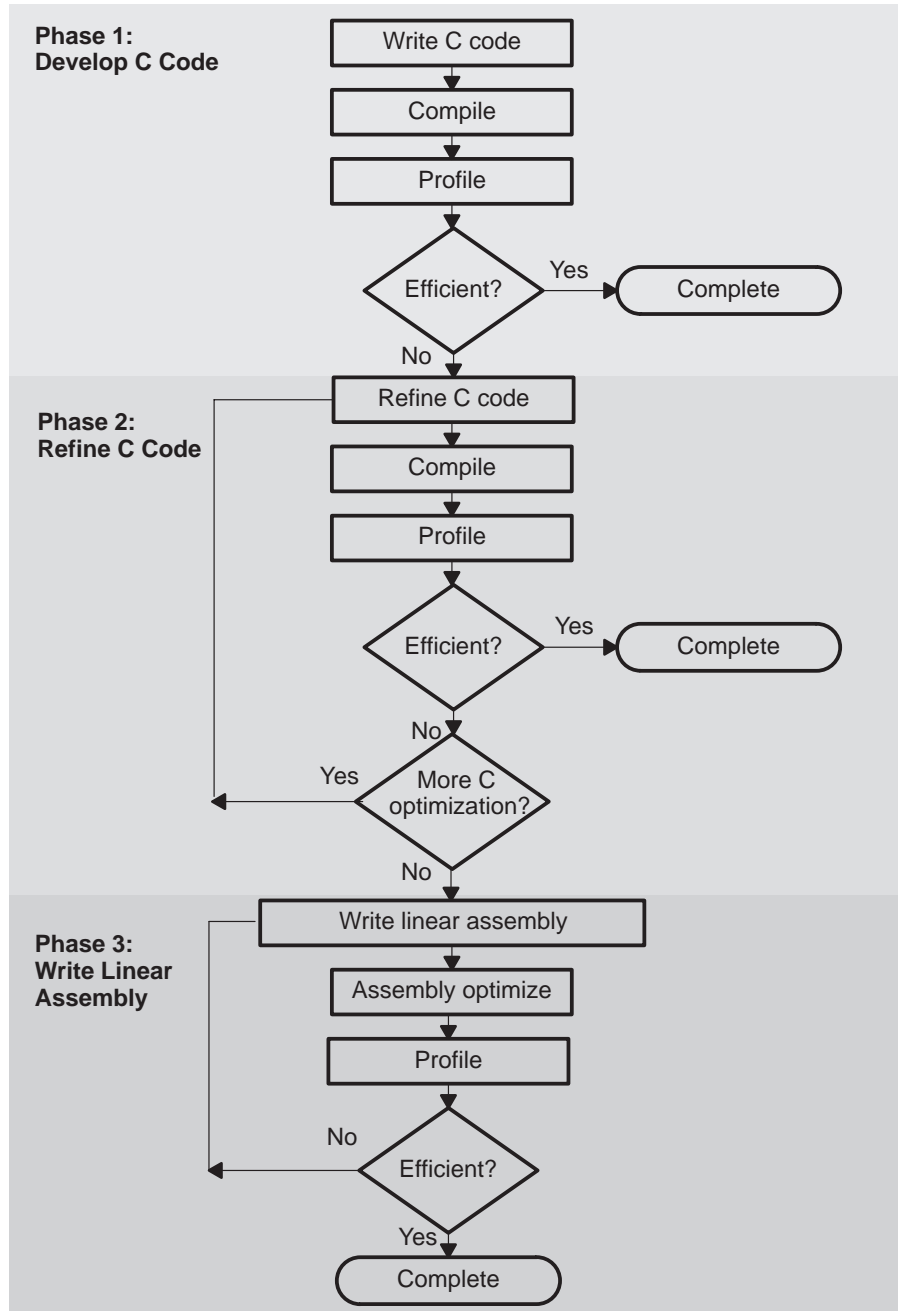
1.2 TMS320C6x Pipeline

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 <i>TMS320C6x C Source Debugger User's Guide</i> 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.

Code Development Flow Tutorial

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
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	2-15
2.6 Lesson 4: Phase 2 of the Code Development Flow	2-18
2.7 Lesson 5: Phase 3 of the Code Development Flow	2-26
2.8 Summary	2-32

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.

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

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_OPTION` environment variable. The shell program uses the default options and/or input filenames that you name with the `C_OPTION` environment variable every time you run the shell.

Use the commands in Table 2-1 to set up the `C_OPTION` 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

There are several different methods to profiling your code. For those who use Code Composer 4.02 or Code Composer Studio 1.00, you should follow the method described in section 2.4.1, Using the Standalone Simulator for Profiling. Others that have an older version of the TI debugger may follow either the method described in 2.4.1 or 2.4.2.

2.4.1 Using the Standalone Simulator for Profiling

There are two methods to using the standalone simulator (load6x) for profiling. If you are interested in just a profile of all of the functions in your application, there is an option in load6x. If you are interested in just profiling the cycle count of one or two functions or if you are interested in a region of code inside a particular function, you can use calls to the clock() function (supported by load6x) to time those particular functions or regions of code.

2.4.1.1 Using the -g Option to Profile on load6x

Invoking load6x with the -g option runs the standalone simulator in profiling mode. Source files must be compiled with the -mg profiling option for profiling to work on the standalone simulator. The profile results resemble the results given by the profiler in the TI simulator debugger. The profile results are stored in a file called by the same name as the .out file, but with the .vaa extension.

For example, to create a profile information file for the compiled and linked file named “example.out”, enter the following on your command line:

```
load6x -g example.out
```

Now, you can edit the file “example.vaa” to see the results of the profile session on the .out file.

For example, if you followed the command line to build demo1.out

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

Then run demo1.out on load6x with profiling enabled:

```
load6x -g demo1.out
```

A new file, `demo1.vaa`, should have been created in the same directory as the `demo1.out` file. Edit the `demo1.vaa` file with a text editor. You should see the following in the file:

```

Program Name:      demo1.out
Start Address:    00007980 main, at line 1, "demo1.c"
Stop Address:     00007860          exit
Run Cycles:       3339
Profile Cycles:   3339
BP Hits:          11
*****
Area Name          Count Inclusive Incl-Max Exclusive
Excl-Max
CF iir1()           1      236      236      236
236
CF vec_mpy1()       1      248      248      248
248
CF mac1()           1      168      168      168
168
CF main()           1      3333     3333      40
40

```

Count represents the number of times each function was called and entered. Inclusive represents the total cycle time spent inside that function including calls to other functions. Incl-Max (Inclusive Max) represents the longest time spent inside that function during one call. Exclusive and Excl-Max are the same as Inclusive and Incl-Max except that time spent in calls to other functions inside that function have been removed.

2.4.1.2 Using the `clock()` Function to Profile

To get cycle count information for a function or region of code with the standalone 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 = macl(a, b, c, dotp);
    stop = clock();
    printf("macl 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();
    iirl(coefs, x, optr, state);
    stop = clock();
    printf("iirl 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.

2.4.2 Using the Profiler in the Debugger

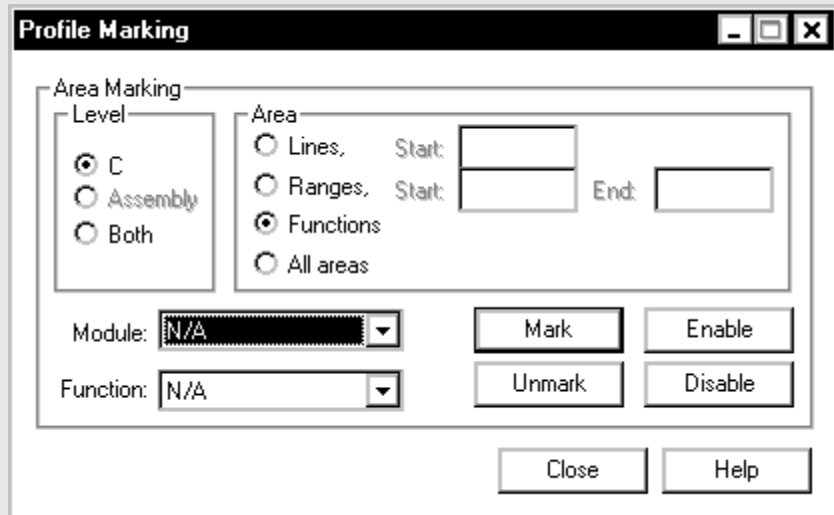
Now, use the profiler to look at the output of `demo1`. In this lesson, you will use the profiler to see the total execution time in number of cycles of each C function in `demo1.out`.

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

- 1) Double-click the icon for the debugger.
The debugger switches to profiling mode and displays only the Command, Disassembly, File, and Profile windows.
- 2) From the Profile menu, select Profile Mode.
This displays the Load Program File dialog box.
- 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.

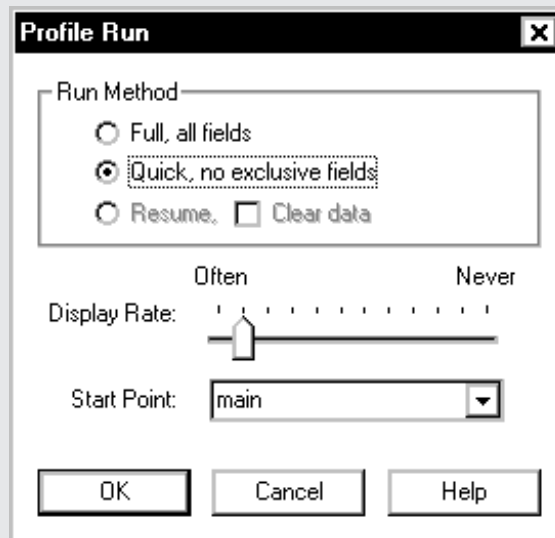
To start the profiling session, follow these steps:

- 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:

Part I

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 2-16.

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
<code>mac1()</code>	1	150	17	$1 \times 150 + 17 = 167$
<code>vec_mpy1()</code>	2	150	16	$2 \times 150 + 16 = 316$
<code>iir1()</code>	5	50	20	$5 \times 50 + 20 = 270$

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.

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      ;@@@@@@@

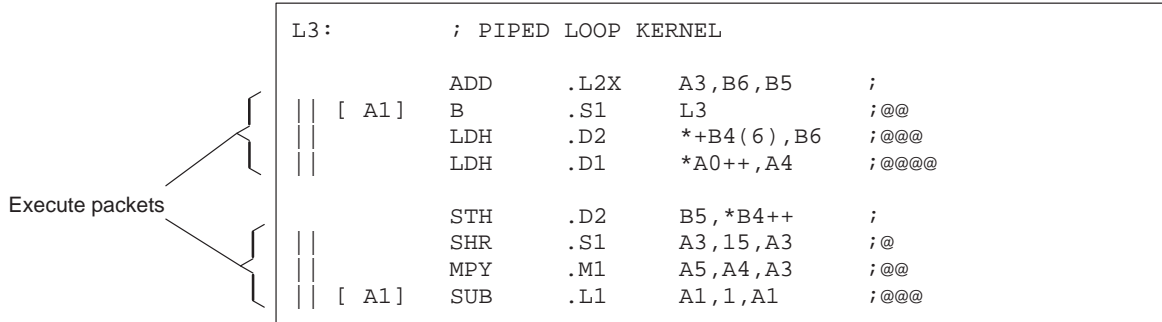
```

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 2-14 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`



Example 2–8 shows the assembly output of the innermost loop for the `iir()` function. Recall from page 2-14 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

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

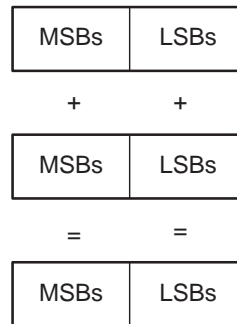
||          STH    .D2    B5,*B4++      ;
||          SHR    .S1    A3,15,A3      ;@
||          MPY    .M1    A5,A4,A3      ;@@
|| [ A1 ]   SUB    .L1    A1,1,A1       ;@@@
```

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]) +
                  _mpyh1(coefs[1], state[1])) >> 15);

        x = t + ((_mpy(coefs[0], state[0]) +
                  _mpyh1(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:

```
c16x -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:

```
c16x -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
<code>vec_mpy1()</code>	2	150	16	$2 \times 150 + 16 = 316$
<code>vec_mpy2()</code>	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
<code>iir1()</code>	5	50	20	$5 \times 50 + 20 = 270$
<code>iir2()</code>	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:

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
<code>mac1()</code> [†]	1	150	17	$1 \times 150 + 17 = 167$
<code>vec_mpy2()</code>	2	75	22	$2 \times 75 + 22 = 172$
<code>iir2()</code>	4	50	20	$4 \times 50 + 20 = 220$

[†] 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 2-21.

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]) +
                    _mpyh1(coefs[1],state[1])) >> 15);

        x = t + ((_mpy(coefs[0],state[0]) +
                    _mpyh1(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]
    
```

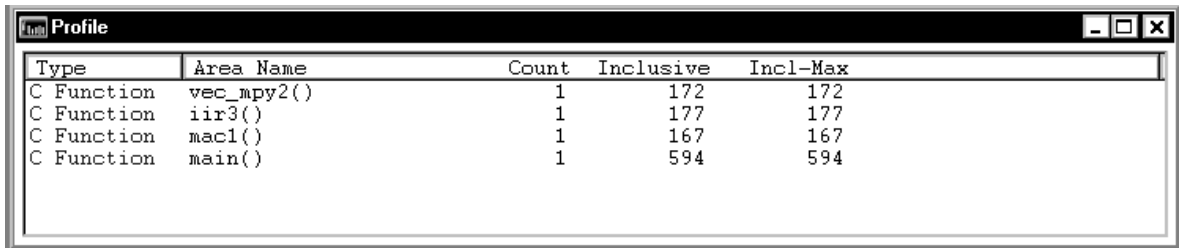
Table 2–6 shows how the iir() function has improved as it has moved through the three phases of code development.

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

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

Your profile window should look like this:



Type	Area Name	Count	Inclusive	Incl-Max
C Function	vec_mpy2()	1	172	172
C Function	iir3()	1	177	177
C Function	mac1()	1	167	167
C Function	main()	1	594	594

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.

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 <ul style="list-style-type: none"> <input type="checkbox"/> Validates original C code <input type="checkbox"/> Determines which loops are most important in terms of MIPS requirements
2	Add const declarations and loop count information <ul style="list-style-type: none"> <input type="checkbox"/> Reduces potential pointer aliasing problems <input type="checkbox"/> Allows loops with indeterminate iteration counts to execute epilogs <input type="checkbox"/> Uses <code>_nassert()</code> intrinsic to pass loop count information to the compiler
3	Optimize C code using other 'C6x intrinsics and other methods <ul style="list-style-type: none"> <input type="checkbox"/> Facilitates use of certain 'C6x instructions not easily represented in C <input type="checkbox"/> Optimizes data flow bandwidth (uses word access for short ('C62x) data and double word access for word ('C67x) data)
4a	Write linear assembly <ul style="list-style-type: none"> <input type="checkbox"/> Allows control in determining exact 'C6x instructions to be used <input type="checkbox"/> Provides flexibility of hand-coded assembly without worry of pipelining, parallelism, or register allocation <input type="checkbox"/> Can pass memory bank information to the tools <input type="checkbox"/> Uses <code>.trip</code> directive to convey loop count information
4b	Add partitioning information to the linear assembly <ul style="list-style-type: none"> <input type="checkbox"/> Can improve partitioning of loops when necessary <input type="checkbox"/> Can avoid bottlenecks of certain hardware resources
Code size considerations	
	<ul style="list-style-type: none"> <input type="checkbox"/> Can trade small performance degradation for smaller code on loops <input type="checkbox"/> Can significantly reduce code size for control code

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 7, *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
;*      Known Minimum Trip Count      : 128
;*      Known Maximum Trip Count      : 128
;*      Known Max Trip Count Factor   : 128
;*      Loop Carried Dependency Bound (^) : 2
;*      Unpartitioned Resource Bound   : 4
;*      Partitioned Resource Bound(*)   : 4
;*
;*              A-side      B-side
;*      .L units          1          0
;*      .S units          3          2
;*      .D units          1          5*
;*      .M units          3          2
;*      .X cross paths    4*         2
;*      .T address paths  3          3
;*      Long read paths   2          1
;*      Long write paths  0          0
;*      Logical ops (.LS)  1          0 (.L or .S unit)
;*      Addition ops (.LSD) 3          7 (.L or .S or .D unit)
;*      Bound (.L .S .LS)  3          1
;*      Bound (.L .S .D .LS .LSD) 3          5*
;*
;* Searching for software pipeline at...
;*      ii = 5 Schedule found with 3 iterations in parallel
;* Done
;*
;* Speculative Loop Threshold      : Unknown
;* Collapsed Epilog Stages        : 2
;*
;* Prolog not removed              : Cannot speculate or predicate instruction
;* Collapsed Prolog Stages        : 0
;*=====

```

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 <ul style="list-style-type: none"> ➤ Use -pm program level optimization to reduce memory pointer aliasing ➤ Add const declarations to all pointers passed to a function that are read only ➤ Use -mt option to assume no memory pointer aliasing Linear assembly <ul style="list-style-type: none"> ➤ Use the .mdep and .no_mdep assembly optimizer directives 	Performing Program–Level Optimization (–pm Option) The const Keyword Memory Dependencies Memory Alias Disambiguation Assembly Optimizer Options and Directives	4-11 4-8 4-6 Appendix A 7-4
Partitioned resource bound is higher than unpartitioned resource bound	<ul style="list-style-type: none"> ➤ Write code in linear assembly with partitioning/functional unit information 	Linear Assembly Resource Allocation	7-24
Too many instructions, or inefficient instructions were generated by the compiler	<ul style="list-style-type: none"> ➤ Use intrinsics in C code to select more efficient 'C6x instructions ➤ Write code in linear assembly to pick exact 'C6x instruction to be executed 	Using Intrinsics Optimizing Assembly Code via Linear Assembly	4-13 Chapter 7
Failed to software pipeline due to register live–too–long	<ul style="list-style-type: none"> ➤ Use the -mx option for both C code and linear assembly ➤ Write linear assembly and insert MV instructions to split register lifetimes that are live–too–long 	Software Pipelining Retry Split–Join–Path Problems	4-41 7-105
Failed to software pipeline due to register allocation (Cannot allocate machine registers)	<ul style="list-style-type: none"> ➤ Try splitting the loop into two separate loops Linear Assembly <ul style="list-style-type: none"> ➤ Repartition if too many instructions on one side ➤ Use symbolic register names instead of machine registers (A0–A15, B0–B15) 	Optimizing Assembly Code via Linear Assembly	Chapter 7

Table 3–2. TMS320C6x Optimization Checklist (Continued)

Feedback	Solution	For more information, refer to ...	Page #
Failed to software pipeline due to register allocation (Too many predicates live on one side)	<ul style="list-style-type: none"> ✔ Try splitting the loop into two separate loops ✔ If multiple conditionals are used in the loop, allocation of these conditionals is 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 is 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. 		
T address paths are resource bound	<p>C code</p> <ul style="list-style-type: none"> ✔ Use word accesses for short arrays; declare <code>int *</code> (or use <code>_nassert</code>) and use <code>mpy</code> intrinsics to multiply upper and lower halves of registers ✔ Try to employ redundant load elimination technique if possible <p>Linear Assembly</p> <ul style="list-style-type: none"> ✔ Use LDW/STW instructions for accesses to memory 	<p>Using Word Accesses for Short Data in Part II</p> <p>Redundant Load Elimination</p> <p>Using Word Access for Short Data in Part III</p>	<p>4-18</p> <p>7-111</p> <p>7-20</p>
There are memory bank conflicts (specified in the memory analysis window of simulator)	<ul style="list-style-type: none"> ✔ Write linear assembly and use the <code>.mptr</code> directive 	<p>The <code>.mptr</code> Directive</p> <p>Loop Unrolling</p>	<p>7-5</p> <p>7-95</p>
Larger outer loop overhead in nested loop	<ul style="list-style-type: none"> ✔ Unroll the inner loop ✔ Make one loop with the outer loop instructions conditional on an inner loop counter 	<p>Loop Unrolling in Part II And Part III</p> <p>Outer Loop Conditionally Executed With Inner Loop</p>	<p>4-36</p> <p>7-95</p> <p>7-137</p>
Uneven resources (for example, 3 multiplies per iteration)	<ul style="list-style-type: none"> ✔ Unroll the loop to make an even number of resources 	<p>Loop Unrolling in Part III</p>	<p>7-95</p>

Table 3–2. TMS320C6x Optimization Checklist (Continued)

Feedback	Solution	For more information, refer to ...	Page #
Two loops are generated, one not software pipelined	<p>C code</p> <ul style="list-style-type: none"> ✔ Use -pm program level optimization to gather more trip count information ✔ Use the <code>_nassert</code> intrinsic to specify loop count information <p>Linear Assembly</p> <ul style="list-style-type: none"> ✔ Use the <code>.trip</code> directive to specify loop count information 	<p>Performing Program Level Optimization (–pm Option)</p> <p>Communication Trip-Count Information to the Compiler</p> <p>The <code>.trip</code> Directive</p>	<p>4-11</p> <p>4-34</p> <p>7-8</p>
<p>Did not find schedule (Too many reads of one register)</p> <p>(Cycle Count too high. Not profitable)</p>	<ul style="list-style-type: none"> ✔ Split into multiple loops or reduce the complexity of the loop if possible <p>Linear Assembly</p> <ul style="list-style-type: none"> ✔ Unpartition/repartition the linear assembly source code ✔ Probably best modified by another technique (i.e. loop unrolling) ✔ Modify the register and/or partition constraints in linear assembly 	<p>Loop Unrolling in Part II And Part III</p>	<p>4-36</p> <p>7-95</p>
Address increment too large	<ul style="list-style-type: none"> ✔ Modify code so that the memory offsets are closer 		
Iterations in parallel > min. trip count	<ul style="list-style-type: none"> ✔ Use -pm program level optimization to gather more trip count information ✔ Add <code>_nassert</code> or <code>.trip</code> to provide more information on the minimum trip count ✔ Make sure that code size flag (–ms0) is not used in the compiler options 	<p>Performing Program Level Optimization (–pm Option)</p> <p>Communicating Trip Count Information to the Compiler</p> <p>The <code>.trip</code> Directive</p>	<p>4-11</p> <p>4-34</p> <p>7-8</p>
Iterations in parallel > max. trip count	<ul style="list-style-type: none"> ✔ Probably best optimized by another technique (i.e. unroll the loop completely) 	<p>Loop Unrolling in Part II And Part III</p>	<p>4-36</p> <p>7-95</p>

Table 3–2. TMS320C6x Optimization Checklist (Continued)

Feedback	Solution	For more information, refer to ...	Page #
Trip var. used in loop - Can't adjust trip count	<ul style="list-style-type: none"> ✔ Replicate the trip count variable and use the copy inside the loop so that the trip counter and the loop reference separate variables 	What Disqualifies a Loop From Being Software Pipelined	4-41
Loop will not software pipeline for other reasons	<ul style="list-style-type: none"> ✔ Make sure there are no function calls, branches to other code, or conditional break or continue statements in the loop. 	What Disqualifies a Loop from Being Software Pipelined	4-41
	<ul style="list-style-type: none"> ✔ Try making the loop counter down counting and declare it an int in C 	Tips on Data Types, Trip Count Issues	4-2, 4-33
	<ul style="list-style-type: none"> ✔ Refer to Section 4.3.3.7 What Disqualifies a Loop from Being Software–Pipelined for a full list of potential reasons 		

Part I

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

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

Topic	Page
4.1 Writing C Code	4-2
4.2 Compiling C Code	4-4
4.3 Refining C Code	4-13

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 (1 cycle for “short * short” versus 5 cycles for “int * int”).
- 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. For example, the RTS floating-point multiply will be used instead of the MPYSP instruction.

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. Remember to subtract out the overhead of calling the `clock()` function.

- ❑ Use the profile mode of the stand-alone simulator. This can be done by compiling your code with the `-mg` option and executing `load6x` with the `-g` option. The profile results will be stored in a file with the `.vaa` extension. Refer to the *TMS320C6x Optimizing C Compiler User's Guide* for more information.
- ❑ Enable the clock and use profile points and the `RUN` command in the Code Composer debugger to track the number of CPU clock cycles consumed by a particular section of code. Use “View Statistics” to view the number of cycles consumed.
- ❑ 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 with the stand-alone simulator (`load6x`).

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<n> [†]	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. When used on linear assembly files, it acts like a .no_mdep directive that has been defined for those linear assembly files.
-mg	Allows you to profile optimized code
-ms<n>	Allows you to reduce code size in loop code (-ms0) for a small performance degradation and reduce code size in control code (-ms2)
-k	Keeps the assembly file so that you can inspect it
-mu	Disables software pipelining (useful in helping to debug linear assembly source code)
-mh <n>	Allows speculative execution
-mi <n>	Describes the interrupt threshold to the compiler (See section 8.4.)
-ml <n>	Describes how to reach data and code (near/far)

[†] Although -o3 is preferable, at a minimum use the -o option.

[‡] Use the -pm option for as much of your program as possible.

Option	Description
<code>-mr <n></code>	Describes how to call RTS routines (near/far)
<code>-mx</code>	Enables software pipelined loop retry. This option tries multiple schedules on loops and selects the best schedule based on the trip count information known about the loop.

† 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, for example, a variable must be loaded from memory before it can be used. 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 accounting for any latencies needed to complete the first instruction.
- 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. Remember, using the `-mt` option on linear assembly code is equivalent to adding the `.no_mdep` directive to the linear assembly source file. Specific memory dependencies should be specified with the `.mdep` directive. For more information see section 4.4, *Assembly Optimizer Directives* in the *TMS320C6x Optimizing C Compiler User's Guide*.

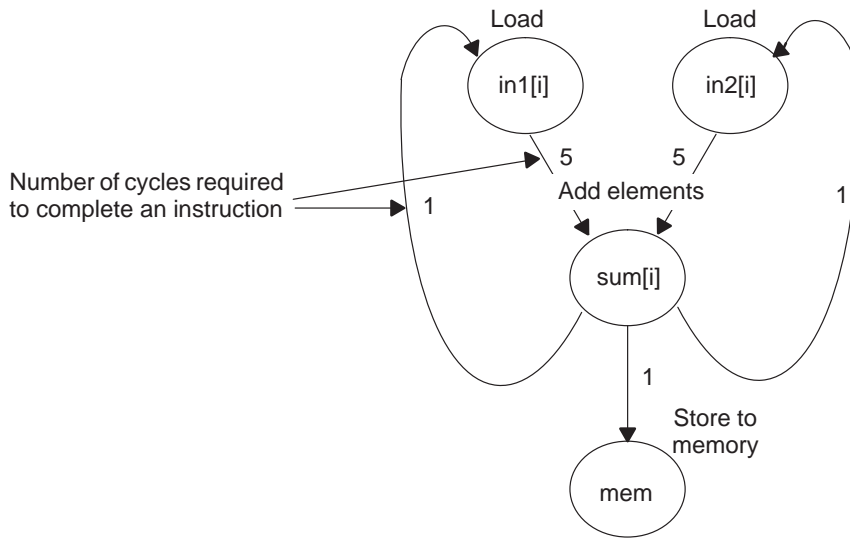
To illustrate the concept of memory dependencies, it is helpful to look at the algorithm code in a dependency graph. Example 4–1 shows the C code for a basic vector sum. Figure 4–1 shows the dependency graph for this basic vector sum. (For more information, see section 7.3.4, *Drawing a Dependency Graph*, on page 7-11.)

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.

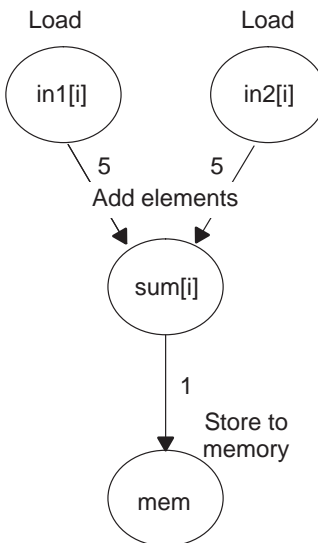
Example 4–2 shows the `vecsum()` example rewritten with the `const` keyword to indicate that the write to `sum` never changes the memory referenced by `in1` and `in2`. Figure 4–2 shows the revised dependency graph for the code in the inner loop.

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



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

```

L4:                ; PIPE LOOP KERNEL

        ADD        .L1X  B4,A0,A5
|| [B0] B         .S2    L4
|| [A1] LDH       .D1T1  *A3++,A0

        [A2] SUB   .S1    A2, 1, A2
|| [A1] SUB       .L1    A1, 1, A1
|| [!A2]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*.

The compiler has collapsed the prolog and epilog code for the loop into the kernel as conditional code. That is why the LDH and STH instructions are executed conditionally. For more information on understanding loop prologs, kernels, and epilogs, refer to Chapter 6.

Caution

Do not use the const keyword if two pointers point to the same object in memory and one of those pointers modifies memory.

Part II

Example 4–4. Incorrect Use of the const Keyword

```

void func (short *a, const short *b)    /*Bad!! */
{
    int i;
    for (i = 11; i < 44; i++) * (--a) = * (--b);
}
void main ()
{
    short array[] = { 1, 2, 3, 4, 5, 6, 7, 9, 9, 10,
                     11, 12, 13, 14, 15, 16, 17, 18,
                     19, 20, 21, 22, 23, 24, 25, 26,
                     27, 28, 29, 30, 31, 32, 33, 34,
                     35, 36, 37, 38, 39, 40, 41, 42,
                     43, 44};

    short *ptr1, *ptr2;

    ptr2 = array + 44;
    ptr1 = ptr2 - 11;

    func(ptr2, ptr1);                    /*Bad!! */
}

```

Do *not* use the `const` keyword with code such as listed in Example 4–4. By using the `const` keyword in Example 4–4, you are telling the compiler that it is legal to write to any location pointed to by `a` before reading the location pointed to by `b`. This is illegal because both `a` and `b` point to the same object—`array`.

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.

Also, using the `–pm` option can lead to better schedules for your loops. If the number of iterations of a loop is determined by a value passed into the function,

and the compiler can determine what that value is from the caller, then the compiler will have more information about the minimum trip count of the loop leading to a better resulting schedule.

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 `in1` and `in2` do not alias memory pointed to by `sum` and, therefore, eliminates memory dependencies among the instructions that access those variables.

You would get the same loop kernel listed in Example 4-3. If your code does not follow the assumptions generated by the `-mt` option, you can get incorrect results. For more information on the `-mt` option refer to section 3.6.2 in the *TMS320C6x Optimizing C Compiler User's Guide*.

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 4–5.

Example 4–5. 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 4–6).

Example 4–6. Saturated Add With Ininsics

```
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 <i>src2</i>); int _labs (long <i>src2</i>);	ABS	Returns the saturated absolute value of <i>src2</i> .	
int _add2 (int <i>src1</i> , int <i>src2</i>);	ADD2	Adds the upper and lower halves of <i>src1</i> to the upper and lower halves of <i>src2</i> and returns the result. Any overflow from the lower half add will not affect the upper half add.	
uint _clr (uint <i>src2</i> , uint <i>csta</i> , uint <i>cstb</i>);	CLR	Clears the specified field in <i>src2</i> . The beginning and ending bits of the field to be cleared are specified by <i>csta</i> and <i>cstb</i> , respectively.	
unsigned _clrr (uint <i>src1</i> , int <i>src2</i>);	CLR	Clears the specified field in <i>src2</i> . 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 <i>src2</i> , uint <i>csta</i> , int <i>cstb</i>);	EXT	Extracts the specified field in <i>src2</i> , sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; <i>csta</i> and <i>cstb</i> are the shift left and shift right amounts, respectively.	
int _extr (int <i>src2</i> , int <i>src1</i>);	EXT	Extracts the specified field in <i>src2</i> , sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; <i>csta</i> and <i>cstb</i> 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 Intrinsic (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
uint _extu (uint <i>src2</i> , uint <i>csta</i> , uint <i>cstb</i>);	EXTU	Extracts the specified field in <i>src2</i> , zero-extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right; <i>csta</i> and <i>cstb</i> are the shift left and shift right amounts, respectively.	
uint _extur (uint <i>src2</i> , int <i>src1</i>);	EXTU	Extracts the specified field in <i>src2</i> , zero-extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right; <i>csta</i> and <i>cstb</i> are the shift left and shift right amounts, respectively.	
uint _ftoi (float);		Reinterprets the bits in the float as an unsigned integer. (Ex: <code>_ftoi(1.0) == 1065353216U</code>)	'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: <code>_itof(0x3f800000) == 1.0</code>)	'C67x
uint _lmbd (uint <i>src1</i> , uint <i>src2</i>);	LMBD	Searches for a leftmost 1 or 0 of <i>src2</i> determined by the LSB of <i>src1</i> . 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 <i>src1</i> , int <i>src2</i>); int _mpyus (uint <i>src1</i> , int <i>src2</i>); int _mpysu (int <i>src1</i> , uint <i>src2</i>); uint _mpyuu (uint <i>src1</i> , uint <i>src2</i>);	MPY MPYUS MPYSU MPYU	Multiplies the 16 LSBs of <i>src1</i> by the 16 LSBs of <i>src2</i> and returns the result. Values can be signed or unsigned.	
int _mpyh (int <i>src1</i> , int <i>src2</i>); int _mpyhus (uint <i>src1</i> , int <i>src2</i>); int _mpyhsu (int <i>src1</i> , uint <i>src2</i>); uint _mpyhu (uint <i>src1</i> , uint <i>src2</i>);	MPYH MPYHUS MPYHSU MPYHU	Multiplies the 16 MSBs of <i>src1</i> by the 16 MSBs of <i>src2</i> 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 <code>_mpyhl</code> (int <i>src1</i> , int <i>src2</i>); int <code>_mpyhuls</code> (uint <i>src1</i> , int <i>src2</i>); int <code>_mpyhslu</code> (int <i>src1</i> , uint <i>src2</i>); uint <code>_mpyhlu</code> (uint <i>src1</i> , uint <i>src2</i>);	MPYHL MPYHULS MPYHSLU MPYHLU	Multiplies the 16 MSBs of <i>src1</i> by the 16 LSBs of <i>src2</i> and returns the result. Values can be signed or unsigned.	
int <code>_mpylh</code> (int <i>src1</i> , int <i>src2</i>); int <code>_mpyluhs</code> (uint <i>src1</i> , int <i>src2</i>); int <code>_mpylshu</code> (int <i>src1</i> , uint <i>src2</i>); uint <code>_mpylhu</code> (uint <i>src1</i> , uint <i>src2</i>);	MPYLH MPYLUHS MPYLSHU MPYLHU	Multiplies the 16 LSBs of <i>src1</i> by the 16 MSBs of <i>src2</i> and returns the result. Values can be signed or unsigned.	
void <code>_nassert</code> (int);		Generates no code. Tells the optimizer that the expression declared with the <code>assert</code> function is true. This gives a hint to the compiler as to what optimizations might be valid (trip count information for software pipelined loops and about using word-wide optimizations).	
uint <code>_norm</code> (int <i>src2</i>); uint <code>_lnorm</code> (long <i>src2</i>);	NORM	Returns the number of bits up to the first nonredundant sign bit of <i>src2</i> .	
double <code>_rcpdp</code> (double);	RCPDP	Computes the approximate 64-bit double reciprocal.	'C67x
float <code>_rcpdp</code> (float);	RCPSP	Computes the approximate 64-bit double reciprocal.	'C67x
double <code>_rsqrdp</code> (double <i>src</i>);	RSQRDP	Computes the approximate 64-bit double reciprocal square root.	'C67x
float <code>_rsqrsp</code> (float <i>src</i>);	RSQRSP	Computes the approximate 32-bit float reciprocal square root.	'C67x
int <code>_sadd</code> (int <i>src1</i> , int <i>src2</i>); long <code>_lsadd</code> (int <i>src1</i> , long <i>src2</i>);	SADD	Adds <i>src1</i> to <i>src2</i> and saturates the result. Returns the result.	
int <code>_sat</code> (long <i>src2</i>);	SAT	Converts a 40-bit value to an 32-bit value and saturates if necessary.	
uint <code>_set</code> (uint <i>src2</i> , uint <i>csta</i> , uint <i>cstb</i>);	SET	Sets the specified field in <i>src2</i> to all 1s and returns the <i>src2</i> value. The beginning and ending bits of the field to be set are specified by <i>csta</i> and <i>cstb</i> , respectively.	

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

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

C Compiler Intrinsic	Assembly Instruction	Description	Device
<code>unsigned _setr(unsigned, int);</code>	SET	Sets the specified field in <code>src2</code> to all 1s and returns the <code>src2</code> value. The beginning and ending bits of the field to be set are specified by the lower ten bits of the source register.	
<code>int _smpy(int src1, int src2);</code> <code>int _smpyh(int src1, int src2);</code> <code>int _smpyhl(int src1, int src2);</code> <code>int _smpylh(int src1, int src2);</code>	SMPY SMPYH SMPYHL SMPYLH	Multiplies <code>src1</code> by <code>src2</code> , left shifts the result by one, and returns the result. If the result is <code>0x80000000</code> , saturates the result to <code>0x7FFFFFFF</code> .	
<code>int _spint(float);</code>	SPINT	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register.	'C67x
<code>uint _sshl(uint src2, uint src1);</code>	SSHL	Shifts <code>src2</code> left by the contents of <code>src1</code> , saturates the result to 32 bits, and returns the result.	
<code>int _ssub(int src1, int src2);</code> <code>long _lssub(int src1, long src2);</code>	SSUB	Subtracts <code>src2</code> from <code>src1</code> , saturates the result size, and returns the result.	
<code>uint _subc(uint src1, uint src2);</code>	SUBC	Conditional subtract divide step.	
<code>int _sub2(int src1, int src2);</code>	SUB2	Subtracts the upper and lower halves of <code>src2</code> from the upper and lower halves of <code>src1</code> , 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 4–7) doubles the performance of the loop. See section 7.4, *Loading Two Data Values with LDW*, on page 7-20 for more information. This type of optimization is called SIMD (Single Instruction Multiple Data).

Example 4–7. 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 4–8 provides a generic `vecsum()` function that handles all types of alignments and array sizes.

Example 4–8. 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);
    /* test to see if sum, in2, and in1 are aligned to a word boundary */

    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 4–9 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 4–9. 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 4–10 shows an FIR filter that can be optimized with word reads of short data and multiply intrinsics.

Example 4–10. 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 4–11 shows an optimized version of Example 4–10. 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 4–11. 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 += _mpyh1(x[i + j],    h[i]);
            y1 += _mpyh1(x[i + j + 1], h[i]);
        }

        *y++ = (int)(y0 >> s);
        *y++ = (int)(y1 >> s);
    }
}
short x[SIZE_X], h[SIZE_H], y[SIZE_Y];
void main()
{
    fir1((int *)x, (int *)h, y, n, ,m, 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 accesses to read 2 short data items, double word accesses 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 4–12. 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 4–12. 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 4–13, the dot product example is rewritten to use double word loads and intrinsics are used to extract the high and low 32-bit values contained in the 64-bit double. The `_hi()` and `_lo()` intrinsics 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. Recall that earlier it was said arrays are aligned on double word boundaries by using either the `DATA_ALIGN` (for globally defined arrays) or `DATA_MEM_BANK` (for locally defined arrays) pragmas. Example 4–13 and Example 4–14 show these pragmas.

Example 4–13. 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<512/2; i++)
    {
        sum0 += _itof(_hi(a[i])) * _itof(_hi(b[i]));
        sum1 += _itof(_lo(a[i])) * _itof(_lo(b[i]));
    }

    return sum0 + sum1;
}
#pragma DATA_ALIGN(a, 8);
#pragma DATA_ALIGN(b, 8);
float ret_val, a[SIZE_A], b[SIZE_B];

void main()
{
    ret_val = dotp2((double *)a, (double *)b);
}
```

In Example 4–14, 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 4–14. 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;
    float sum8 = 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;
}

void main()
{
    /* Using 0 as the bank parameter for the DATA_MEM_BANK */
    /* pragma aligns variable to a double word boundary for */
    /* both C62xx and C67xx. */
    #pragma DATA_MEM_BANK(a, 0);
    #pragma DATA_MEM_BANK (b, 0);
    float ret_val, a[SIZE_A], b[SIZE_B];
    ret_val = dotp3((double *)a, (double *)b);
}
```

4.3.2.4 Using `_nassert()` and Word Accesses

It is possible for the compiler to automatically perform SIMD optimizations for some, but not all loops. By either using global arrays, or by using the `_nassert()` intrinsic to provide alignment information about your pointers, the compiler can transform your code to use word accesses and the 'C6x intrinsics.

Example 4–15 shows how the compiler can automatically do this optimizations.

Example 4–15. Using the Compiler to Generate a Dot Product With Word Accesses

```
int dotprod1(const short *a, const short *b, unsigned int N)
{
    int i, sum = 0;
    /* a and b are aligned to a word boundary */
    _nassert(((int)(a) & 0x3) == 0);
    _nassert(((int)(b) & 0x3) == 0);
    _nassert(N == 40);
    for (i = 0; i < N; i++)
        sum += a[i] * b[i];
    return sum;
}
```

Compile Example 4–15 with the following options: `-o -mw -k`. Open up the assembly file and look at the loop kernel. The results are the exact same as those produced by Example 4–9. The first 2 `_nassert()` intrinsics in Example 4–15 tell the compiler that the arrays pointed to by `a` and `b` are aligned to a word boundary, so it is safe for the compiler to use a LDW instruction to load two short values. The compiler generates the `_mpy()` and `_mpyh()` intrinsics internally as well as the two sums that were used in Example 4–9 (shown again below).

```
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;
}
```

You need some way to convey to the compiler that this loop will also execute an even number of times. The third intrinsic, `_nassert(N == 40)`, (refer to Section 4.3.3) conveys this information by telling the compiler that the loop will execute exactly 40 times - an even number. (For more information on the `_nassert` intrinsic, refer to section 4.3.3.3, Communicating Trip Count Information to the Compiler).

Example 4–16 and Example 4–17 show how to use the `_nassert()` intrinsic to get word accesses on the vector sum and the FIR filter.

Example 4–16. Using the `_nassert()` Intrinsic to Generate Word Accesses for Vector Sum

```
void vecsum(short *sum, const short *in1,
const short *in2, unsigned int N)
{
    int i;
    _nassert(((int)sum & 0x3) == 0);
    _nassert(((int)in1 & 0x3) == 0);
    _nassert(((int)in2 & 0x3) == 0);
    _nassert (N == 40);
    for (i = 0; i < N; i++)
        sum[i] = in1[i] + in2[i];
}
```

Example 4–17. Using `_nassert()` Intrinsic to Generate Word Accesses for FIR Filter

```

void fir (const short x[], const short h[], short y[]
          int n, int m, int s)
{
    int i, j;
    long y0;
    long round = 1L << (s - 1);
    _nassert(((int)x & 0x3) == 0);
    _nassert(((int)h & 0x3) == 0);
    _nassert(((int)y & 0x3) == 0);
    _nassert(n == 40);
    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);
    }
}

```

As you can see from Example 4–17, the optimization done by the compiler is not as optimal as the code produced in Example 4–11, but it is more optimal than the code in Example 4–10.

```

<compiler output from Example 4-17>
L3:      ; PIPED LOOP KERNEL
        [!B0]  ADD      .L1      A9,A7:A6,A7:A6
        ||      MPY      .M2X     A8,B9,B3
        ||      MPYHL    .M1X     B9,A0,A0
        || [ A1]  B        .S2      L3
        ||      LDH      .D2T2    *++B2(8),B9
        ||      LDH      .D1T1    *+A3(4),A8
        [!B0]  ADD      .L2      B9,B5:B4,B5:B4
        ||      MPY      .M1X     A0,B1,A9
        ||      LDW      .D2T2    *+B8(4),B9
        ||      LDH      .D1T1    *+A3(6),A0
        [ B0]  SUB      .S2      B0,1,B0
        || [!B0]  ADD      .L2      B3,B7:B6,B7:B6
        || [!B0]  ADD      .L1      A0,A5:A4,A5:A4
        ||      MPYHL    .M2      B1,B9,B9

```

```

|| [ A1]  SUB    .S1    A1,1,A1
||          LDW    .D2T2  *++B8(8),B1
||          LDH    .D1T1  *++A3(8),A0

```

<compiler output from Example 4-11>

```

L3:      ; PIPED LOOP KERNEL
          ADD    .L2    B3,B5:B4,B5:B4
||
||      ADD    .L1    A3,A5:A4,A5:A4
||
||      MV     .S2    B1,B2
||
||      MPY    .M2X   B1,A8,B3
||
||      MPYHL  .M1X   B1,A8,A3
|| [ A1]  B     .S1    L3
|| [ B0]  LDW    .D2T2  *B8,B1
||
|| [ B0]  SUB    .S2    B0,1,B0
||
||      ADD    .L1    A3,A7:A6,A7:A6
||
||      ADD    .L2    B3,B7:B6,B7:B6
||
||      MPYH   .M1X   B2,A8,A3
||
||      MPYHL  .M2X   A8,B9,B3
|| [ A1]  SUB    .S1    A1,1,A1
|| [ B0]  LDW    .D1T1  *A0++,A8
|| [ B0]  LDW    .D2T2  *++B8,B9

```

<compiler output from Example 4-10>

```

L4:      ; PIPED LOOP KERNEL
|| [ A2]  SUB    .S1    A2,1,A2
||
||      ADD    .L1    A5,A1:A0,A1:A0
||
||      MPY    .M1X   B5,A4,A5
|| [ B0]  B     .S2    L4
|| [ B0]  SUB    .L2    B0,1,B0
|| [ A2]  LDH    .D1T1  *A3++,A4
|| [ A2]  LDH    .D2T2  *B4++,B5

```

Note:

The `_nassert()` intrinsic may not solve all of your short to int or float-to-double accesses, but it can be a useful tool in achieving better performance without rewriting the C code. Floating point code will not improve with the `_nassert()` intrinsic to try and force double word accesses.

If your code operates on global arrays as in Example 4–18, and you build your application with the `-pm` and `-o3` options, the compiler will have enough information (trip counts and alignments of variables) to determine whether or not SIMD optimization is feasible.

Example 4–18. Automatic Use of Word Accesses Without the `_nassert` Intrinsic

```
<file1.c>
int dotp (short *a, short *b, int c)
{
    int sum = 0, i;
    for (i = 0; i < c; i++) sum += a[i] * b[i];
    return sum;
}
<file2.c>
#include <stdio.h>
short x[40] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
               21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
               31, 32, 33, 34, 35, 36, 37, 38, 39, 40 };
short y[40] = { 40, 39, 38, 37, 36, 35, 34, 33, 32, 31,
               30, 29, 28, 27, 26, 25, 24, 23, 22, 21,
               20, 19, 18, 17, 16, 15, 14, 13, 12, 11,
               10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
void main()
{
    int z;
    z = dotp(x, y, 40);
    printf("z = %d\n", z);
}
```

Compile file1.c and file2.c with:
`cl6x -pm -o3 -k -mw file1.c file2.c`

Edit the resulting assembly file (file1.asm). Notice that the dot product loop uses word accesses and the 'C6x intrinsics.

```

L2:      ; PIPED LOOP KERNEL
        [ !A1 ]  ADD      .L2      B6 ,B7 ,B7
        || [ !A1 ]  ADD      .L1      A6 ,A0 ,A0
        ||          MPY      .M2X     B5 ,A4 ,B6
        ||          MPYH     .M1X     B5 ,A4 ,A6
        || [ B0 ]   B        .S1      L2
        ||          LDW      .D1T1    *+A5(4) ,A4
        ||          LDW      .D2T2    *+B4(4) ,B6

        [ A1 ]   SUB      .S1      A1 ,1 ,A1
        || [ !A1 ]  ADD      .S2      B5 ,B8 ,B8
        || [ !A1 ]  ADD      .L1      A6 ,A3 ,A3
        ||          MPY      .M2X     B6 ,A4 ,B5
        ||          MPYH     .M1X     B6 ,A4 ,A6
        || [ B0 ]   SUB      .L2      B0 ,1 ,B0
        ||          LDW      .D1T1    *++A5(8) ,A4
        ||          LDW      .D2T2    *++B4(8) ,B5

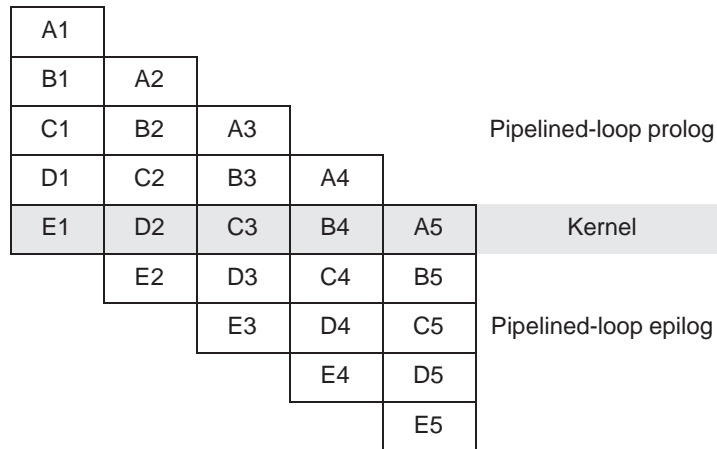
```

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 at levels `-o2` and `-o3` transforms the loop in Example 4–19(a) to something like the code in Example 4–19(b).

Example 4–19. 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 lower bound on the trip count (and in some cases, the upper bound), 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 4-9, 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 (in this case, the minimum trip count equals 2)
- 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 `-ms0` option so that the compiler generates only the software-pipelined code and never generates a redundant loop; however, loops with an unknown trip count, or where the trip count is less than the minimum 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.

You can use the `_nassert` intrinsic to convey many different types of information about the trip count to the compiler.

- It can convey that the trip count will always equal some value.

```
/* This loop will always execute exactly 30 times */
_nassert(x == 30);
for (j = 0; j < x; j++)
```

- It can convey that the trip count will be greater than some minimum value or smaller than some maximum value. The latter is useful when interrupts need to occur inside of loops and you are using the `-mi<n>` option. Refer to section 8.4, *Interruptible Loops*.

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

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

```
/* The trip count will execute some multiple of 4 times */
_nassert((x % 4) == 0);
for (j = 0; j < x; j++)
```

- It can convey information about the alignment of pointers and arrays.

```
void vecsum(short *a, const short *b, const short *c)
{
    _nassert(((int) a & 0x3) == 0);
    _nassert(((int) b & 0x3) == 0);
    _nassert(((int) c & 0x3) == 0);
    . . .
}
```

This information call all be combined as well into a single C statement:

```
_nassert((x >= 8) && (x <= 48) && ((x % 8) == 0));
for (j = 0; j < x; j++)
```

The compiler knows that this loop will execute some multiple of 8 (between 8 and 48) times. This information is useful in providing more information about unrolling a loop or the ability to perform word accesses on a loop.

Several examples in this chapter and in section 8.4.4 show all of the different ways that the `_nassert` intrinsic can be used.

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 4–20, 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 4–20. 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 4–21), 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 4–21. 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 4–22 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 4–22 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 4–22. 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 4–23, 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 4–23. 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 4–24.

In general unrolling may be a good idea if you have an uneven partition or if your loop carried dependency bound is greater than the partition bound. (Refer to section 7.7, *Loop Carry Paths* and section 3.2 in the *TMS320C6x Optimizing C Compiler User's Guide*. This information can be obtained by using the `-mw` option and looking at the comment block before the loop.

Example 4–24. 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.

The heuristic the compiler uses to determine if it should unroll the loops needs to know either of the following pieces of information. Without knowing either of these the compiler will never unroll a loop.

- The exact trip count of the loop
- Or that the trip count of the loop is some multiple of two

The second requirement can be passed to the compiler through the `_nassert` intrinsic. In section 4.3.3.3, *Communicating Trip-Count Information to the Compiler*, it is explained that `_nassert` can be used to provide information about loop unrolling. By using the modulus operator, you can specify that the trip count is a multiple or power of two.

```

_nassert((n % 2) == 0);

```

Example 4–25 shows how the compiler can perform simple loop unrolling of replicating the loop body. The `_nassert` intrinsics tell the compiler that the loop will execute an even number of times greater than 20. This compiler will unroll the loop once to take advantage of the performance gain that results from the unrolling.

Example 4–25. Vector Sum

```

void func(short *a, const short *b, const short *c, int n)
{
    int i;
    _nassert (((n % 2) == 0) && (n >= 20));
    for (i = 0; i < n; i++) a[i] = b[i] + c[i];
}

<compiler output for above code>
L2:      ; PIPED LOOP KERNEL

        ADD     .L1X   B4,A3,A3
|| [ B0]  B       .S1   L2
||
||      LDH     .D1T1  *++A4(4),A3
||      LDH     .D2T2  *++B5(4),B4

        [!A1]  STH     .D1T1  A3,*++A0(4)
||
||      ADD     .L2X   B6,A5,B6
||      LDH     .D2T2  *+B5(2),B6

        [ A1]  SUB     .L1   A1,1,A1
|| [!A1]  STH     .D2T2  B6,*++B7(4)
|| [ B0]  SUB     .L2   B0,1,B0
||      LDH     .D1T1  *+A4(2),A5

```

Note: When the interrupt threshold option is used, unrolling can be used to regain lost performance. Refer to section 8.4.4 *Getting the Most Performance Out of Interruptible Code*.

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 Software Pipelining Retry (*-mx option*)

Use the *-mx* option whenever you are concerned about getting the best possible pipelined schedule out of a loop. Using the *-mx* option tells the compiler to take more time to search for other possible schedules of the loop. The compiler will select the best version of the pipelined loop and generate assembly instructions for that version, based on the trip count information about the loop. Since the compiler needs as much information about the loop as possible, it is recommended that you use the *-o3* and *-pm* options with *-mx*, or use the `_nassert` intrinsics to describe the trip count characteristics of your important loops.

4.3.3.7 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, including code that will call the run-time support routines.

```
for (i = 0; i < 100; i++)
    x[i] = x[i] % 5;
```

This will call the run-time support `_remi` routine.

- ❑ You must not have a conditional break (early exit) in the loop. You need to rewrite your code to use `if` statements instead. Use the `if` statements only around code that updates memory (stores to pointers and arrays) and around variables whose values calculated inside the loop and are used outside the loop. Also, do not nest `if` statements. The compiler cannot software pipeline a loop that contains nested `if` statements. Example 4–26 shows how to combine the nested conditions using `&&` into one `if` condition.

Example 4–26. Use of If Statements in Float Collision Detection

(a) Original Code

```

int colldet(const float *x, const float *p, float point,
float distance)
{
    int I, retval = 0;
    float sum0, sum1, dist0, dist1;
    for (I = 0; I < (28 * 3); I += 6)
    {
        sum0 = x[I+0]*p[0] + x[I+1]*p[1] + x[I+2]*p[2];
        sum1 = x[I+3]*p[0] + x[I+4]*p[1] + x[I+5]*p[2];
        dist0 = sum0 - point;
        dist1 = sum1 - point;
        dist0 = fabs(dist0);
        dist1 = fabs(dist1);
        if (dist0 < distance)
        {
            retval = (int)&x[I + 0];
            break;
        }
        if (dist1 < distance)
        {
            retval = (int)&x[I + 3];
            break;
        }
    }
    return retval;
}

```

(b) Modified Code

```

int colldet_new(const float *x, const float *p, float point,
float distance)
{
    int I, retval = 0;
    float sum0, sum1, dist0, dist1;
    for (I = 0; I < (28 * 3); I += 6)
    {
        sum0 = x[I+0]*p[0] + x[I+1]*p[1] + x[I+2]*p[2];
        sum1 = x[I+3]*p[0] + x[I+4]*p[1] + x[I+5]*p[2];
        dist0 = sum0 - point;
        dist1 = sum1 - point;
        dist0 = fabs(dist0);
        dist1 = fabs(dist1);
        if ((dist0<distance)&&!retval) retval = (int)&x[I+0];
        if ((dist1<distance)&&!retval) retval = (int)&x[I+3];
    }
    return retval;
}

```

- The loop cannot have an incrementing loop counter. Run the optimizer with the `-o2` or `-o3` option to convert as many loops as possible into down-counting loops.

- ❑ If the trip counter is modified within the body of the loop, it typically cannot be converted into a downcounting loop. If possible, rewrite the loop to not modify the trip counter. For example, the following code will not software pipeline:

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

- ❑ A conditionally incremented loop control variable is not software pipelined. Again, if possible, rewrite the loop to not conditionally modify the trip counter. For example the following code will not software pipeline:

```
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. Either try to simplify the loop or break the loop up into multiple smaller loops.
- ❑ If a register value is live too long, the code is not software-pipelined. See section 7.6.6.2, *Live Too Long*, on page 7-68 and section 7.10, *Live-Too-Long Issues*, on page 7-102 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. Try to eliminate or combine these conditions.

Part I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Linking Issues

This chapter contains useful information about other problems and questions that might arise while building your projects, including:

- What to do with the *relocation value truncated* linker and assembler messages
- How to save on-chip memory by moving the RTS off-chip
- How to build your application with RTS calls either *near* or *far*
- How to change the default RTS data from *far* to *near*

Topic	Page
5.1 How to Use Linker Error Messages	5-2
5.2 How to Save On-Chip Memory by Placing RTS Off-Chip	5-5

5.1 How to Use Linker Error Messages

When you try to call a function which, due to how you linked your application, is too far away from a call site to be reached with the normal PC-relative branch instruction, you will see the following linker error message:

```
>> PC-relative displacement overflow. Located in file.obj,
section .text, SPC offset 000000bc
```

This message means that in the named object file in that particular section, is a PC-relative branch instruction trying to reach a call destination that is too far away. The SPC offset is the section program counter (SPC) offset within that section where the branch occurs. For C code, the section name will be `.text` (unless a `CODE_SECTION` pragma is in effect).

You might also see this message in connection with an MVK instruction:

```
>> relocation value truncated at 0xa4 in section .text,
file file.obj
```

Or, an MVK can be the source of this message:

```
>> Signed 16-bit relocation out of range, value truncated.
Located in file.obj, section .text, SPC offset 000000a4
```

These messages are similar. The file is `file.obj`, the section is `.text`, and the SPC offset is `0xa4`. If this happens to you when you are linking C code, here is what you do to find the problem:

- Recompile the C source file as you did before but include `-s -al` in the options list

```
cl6x <other options> -s -al file.c
```

This will give you C interlisted in the assembly output and create an assembler listing file with the extension `.lst`.

- Edit the resulting `.lst` file, in this case `file.lst`.
- Each line in the assembly listing has several fields. For a full description of those fields see section 3.10 of the *TMS320C6x Assembly Language Tools User's Guide*. The field you are interested in here is the second one, the section program counter (SPC) field. Find the line with the same SPC field as the SPC offset given in the linker error message. It will look like:

```
245 000000bc 0FFFE10!      B      .S1  _atoi      ; |56|
```

In this case, the call to the function `atoi` is too far away from the location where this code is linked.

It is possible that use of `-s` will cause instructions to move around some and thus the instruction at the given SPC offset is not what you expect. The branch

or MVK nearest to that instruction is the most likely cause. Or, you can rebuild the whole application with `-s -al` and relink to see the new SPC offset of the error.

If you are tracing a problem in a hand-coded assembly file, the process is similar, but you merely re-assemble with the `-I` option instead of recompiling.

To fix a branch problem, your choices are:

- Use the `-mr1` option to force the call to `atoi`, and all other RTS functions, to be far.
- Compile with `-ml1` or higher to force all calls to be far.
- Rewrite your linker command file (looking at a map file usually helps) so that all the calls to `atoi` are close (within 0x100000 words) to where `atoi` is linked.

If the problem instruction is an MVK, then you need to understand why the constant expression does not fit.

For C code, you might find the instruction looks like:

```
50 000000a4 0200002A%      MVK  (_ary-$bss),B4 ; |5|
```

In this case, the address of the C object `ary` is being computed as if `ary` is declared near (the default), but because it falls outside of the 15-bit address range the compiler presumes for near objects, you get the warning. To fix this problem, you can declare `ary` to be far, or you can use the correct `cl6x -ml<n>` memory model option to automatically declare `ary` and other such data objects to be far. See chapter 2 of the *TMS320C6x Optimizing C Compiler User's Guide* for more information on `-ml<n>`.

It is also possible that `ary` is defined as far in one file and declared as near in this file. In that case, insure `ary` is defined and declared consistently to all files in the project.

If the MVK instruction is just a simple load of an address:

```
123 000000a4 0200002A!      MVK      sym,B4
```

Then the linker warning message is telling you that `sym` is greater than 32767, and you will end up with something other than the value of `sym` in `B4`. In most cases, this instruction is accompanied by:

```
124 000000a8 0200006A!      MVKH     sym,B4
```

When this is the case, the solution is to change the MVK to MVKL.

On any other MVK problem, it usually helps to look up the value of the symbol(s) involved in the linker map file.

5.1.1 Executable Flag

You may also see the linker message:

```
>> warning: output file file.out not executable
```

If this is due solely to MVK instructions, paired with MVKH, which have yet to be changed to MVKL, then this warning may safely be ignored. The loaders supplied by TI will still load and execute this .out file.

If you implement your own loader, please be aware this warning message means the F_EXEC flag in the file header is not set. If your loader depends on this flag, then you will have to fix your MVK instructions, or use the switches described above to turn off these warnings.

5.2 How to Save On-Chip Memory by Placing RTS Off-Chip

One of many techniques you might use to save valuable on-chip space is to place the code and data needed by the runtime-support (RTS) functions in off-chip memory.

Placing the RTS in off-chip memory has the advantage of saving valuable on-chip space. However, it comes at a cost. The RTS functions will run much slower. Depending on your application, this may or may not be acceptable. It is also possible your application doesn't use the RTS library much, and placing the RTS off-chip saves very little on-chip memory.

Table 5–1. Definitions

Term	Means
Normal RTS functions	Ordinary RTS functions. Example: strcpy
Internal RTS functions	Functions which implement atomic C operations such as divide or floating point math on the C62xx. Example: _divu performs 32-bit unsigned divide.
near calls	Function calls performed with a ordinary PC-relative branch instruction. The destination of such branches must be within 1 048 576 (0x100000) words of the branch. Such calls use 1 instruction word and 1 cycle.
far calls	Function calls performed by loading the address of the function into a register and then branching to the address in the register. There is no limit on the range of the call. Such calls use 3 instruction words and 3 cycles.

5.2.1 How to Compile

Make use of shell (cl6x) options for controlling how RTS functions are called:

Table 5–2. Command Line Options for RTS Calls

Option	Internal RTS calls	Normal RTS calls
Default	Same as user	Same as user
–mr0	Near	Near
–mr1	Far	Far

By default, RTS functions are called with the same convention as ordinary user-coded functions. If you do not use a –ml<n> option to enable one of large-memory models, then these calls will be near. The option –mr0 causes calls

to RTS functions to be near, regardless of the setting of the `-ml<n>` switch. This option is for special situations, and typically isn't needed. The option `-mr1` will cause calls to RTS functions to be far, regardless of the setting of the `-ml<n>` switch.

Note these options only address how RTS functions are called. Calling functions with the far method does not mean those functions must be in off-chip memory. It simply means those functions can be placed at any distance from where they are called.

5.2.2 Must #include Header Files

When you call a RTS function, you must include the header file which corresponds to that function. For instance, when you call `memcmp`, you must `#include <string.h>`. If you do not include the header, the `memcmp` call looks like a normal user call to the compiler, and the effect of using `-mr1` does not occur.

5.2.3 RTS Data

Most RTS functions do not have any data of their own. Data is typically passed as arguments or through pointers. However, a few functions do have their own data. All of the "is<xxx>" character recognition functions defined in `ctype.h` refer to a global table. Also, many of the floating point math functions have their own constant look-up tables. All RTS data is defined to be far data, for example, accessed without regard to where it is in memory. Again, this does not necessarily mean this data is in off-chip memory.

Details on how to change access of RTS data are given in section 5.2.7

5.2.4 How to Link

You place the RTS code and data in off-chip memory through the linking process. Here is an example linker command file you could use instead of the `lnk.cmd` file provided in the `\lib` directory.

```

/*****/
/* farlnk.cmd - Link command file which puts RTS off-chip */
/*****/
-c
-heap 0x2000
-stack 0x4000

/* Memory Map 1 - the default */
MEMORY
{
    PMEM:  o = 00000000h  l = 00010000h
    EXT0:  o = 00400000h  l = 01000000h
    EXT1:  o = 01400000h  l = 00400000h
    EXT2:  o = 02000000h  l = 01000000h
    EXT3:  o = 03000000h  l = 01000000h
    BMEM:  o = 80000000h  l = 00010000h
}

SECTIONS
{
    /*-----*/
    /* Sections defined only in RTS. */
    /*-----*/
    .stack      >      BMEM
    .systemem   >      BMEM
    .cio        >      EXT0

    /*-----*/
    /* Sections of user code and data */
    /*-----*/
    .text       >      PMEM
    .bss        >      BMEM
    .const      >      BMEM
    .data       >      BMEM
    .switch     >      BMEM
    .far        >      EXT2

    /*-----*/
    /* All of .cinit, including from RTS, must be collected together */
    /* in one step. */
    /*-----*/
    .cinit      >      BMEM

```



```

/*-----*/
/* RTS code - placed off chip */
/*-----*/
.rtscode { -lrts6201.lib(.text) } > EXT0

/*-----*/
/* RTS data - undefined sections - placed off chip */
/*-----*/
.rtsbss { -lrts6201.lib(.bss)
         -lrts6201.lib(.far) } > EXT0

/*-----*/
/* RTS data - defined sections - placed off chip */
/*-----*/
.rtsdata { -lrts6201.lib(.const)
          -lrts6201.lib(.switch) } > EXT0
}

```

User sections (.text, .bss, .const, .data, .switch, .far) are built and allocated normally.

The .cinit section is built normally as well. It is important to not allocate the RTS .cinit sections separately as is done with the other RTS sections. All of the .cinit sections must be combined together into one section for auto-initialization of global variables to work properly.

The .stack, .system, and .cio sections are entirely created from within the RTS. So, you don't need any special syntax to build and allocate these sections separately from user sections. Typically, you place the .stack (system stack) and .system (heap of memory used by malloc, etc.) sections in on-chip memory for performance reasons. The .cio section is a buffer used by printf and related functions. You can typically afford slower performance of such I/O functions, so it is placed in off-chip memory.

The .rtstext section collects all the .text, or code, sections from RTS and allocates them to external memory name EXT0. If needed, replace the library name rts6201.lib with the library you normally use, perhaps rts6701.lib. The -l is required, and no space is allowed between the -l and the name of the library. The choice of EXT0 is arbitrary. Use the memory range which makes the most sense in your application.

The .rtsbss section combines all of the undefined data sections together. Undefined sections reserve memory without any initialization of the contents of that memory. You use .bss and .usect assembler directives to create undefined data sections.

The `.rtsdata` section combines all of the defined data sections together. Defined data sections both reserve and initialize the contents of a section. You use the `.sect` assembler directive to create defined sections.

It is necessary to build and allocate the undefined data sections separately from the defined data sections. When a defined data section is combined together with an undefined data section, the resulting output section is a defined data section, and the linker must fill the range of memory corresponding to the undefined section with a value, typically the default value of 0. This has the undesirable effect of making your resulting `.out` file much larger.

You may get a linker warning like:

```
>> farlnk.cmd, line 65: warning: rts6201.lib(.switch) not found
```

That means none of the RTS functions needed by your application define a `.switch` section. Simply delete the corresponding `-l` entry in the linker command file to avoid the message. If your application changes such that you later do include an RTS function with a `.switch` section, it will be linked next to the `.switch` sections from your code. This is fine, except it is taking up that valuable on-chip memory. So, you may want to check for this situation occasionally by looking at the linker map file you create with the linker `-m` option.

5.2.5 Example Compiler Invocation

A typical build could look like:

```
cl6x -mr1 <other options> <C files> -z -o app.out  
-m app.map farlnk.cmd
```

In this one step you both compile all the C files and link them together. The C6x executable image file is named `app.out` and the linker map file is named `app.map`.

Refer to section 4.4.1 to learn about the linker error messages when calls go beyond the PC relative boundary.

5.2.6 Header File Details

Look at the file `linkage.h` in the `\include` directory of the release. Depending on the value of the `_FAR_RTS` macro, the macro `_CODE_ACCESS` is set to force calls to RTS functions to be either user default, near, or far. The `_FAR_RTS` macro is set according to the use of the `-mr<n>` switch.

Table 5–3. How `_FAR_RTS` is Defined in `Linkage.h` With `-mr`

Option	Internal RTS calls	Normal RTS calls	<code>_FAR_RTS</code>
Default	Same as user	Same as user	Undefined
<code>-mr0</code>	Near	Near	0
<code>-mr1</code>	Far	Far	1

The `_DATA_ACCESS` macro is set to always be far.

The `_IDECL` macro determines how inline functions are declared.

All of the RTS header files which define functions or data include `linkage.h` header file. Functions are modified with `_CODE_ACCESS`:

```
extern _CODE_ACCESS void exit(int _status);
```

and data is modified with `_DATA_ACCESS`:

```
extern _DATA_ACCESS unsigned char _ctype_[];
```

5.2.7 Changing RTS Data to near

If for some reason you do not want accesses of RTS data to use the far access method, take these steps:

Go to the `\include` directory of the release.

Edit `linkage.h`, and change the:

```
#define _DATA_ACCESS far
```

```
macro to
```

```
#define _DATA_ACCESS near
```

to force all access of RTS data to use near access, or change it to

```
#define _DATA_ACCESS
```

if you want RTS data access to use the same method used when accessing ordinary user data.

Copy `linkage.h` to the `\lib` directory.

Go to the `\lib` directory.

Replace the `linkage.h` entry in the source library:

```
ar6x -r rts.src linkage.h
```

Delete `linkage.h`.

- Rename or delete the object library you use when linking.
- Rebuild the object library you use with the library build command listed in the readme file for that release.

Note that you will have to perform this process each time you install an update of the code generation toolset.

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
6.1 Labels	6-2
6.2 Parallel Bars	6-2
6.3 Conditions	6-3
6.4 Instructions	6-4
6.5 Functional Units	6-6
6.6 Operands	6-8
6.7 Comments	6-9

6.1 Labels

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

Figure 6–1 shows the position of the label in a line of assembly code. The colon following the label is optional.

Figure 6–1. 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.

6.2 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 6–2. Parallel Bars in Assembly Code

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

6.3 Conditions

Five registers in the 'C6x are available for conditions: A1, A2, B0, B1, and B2. Figure 6–3 shows the position of a condition in a line of assembly code.

Figure 6–3. 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

6.4 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 6–1.
- ❑ *Processor mnemonics* are the actual microprocessor instructions that execute at runtime and perform the operations in the program. Table 6–2 summarizes the 'C6x mnemonics. Processor mnemonics must begin in column 2 or greater.

Figure 6–4 shows the position of the instruction in a line of assembly code.

Figure 6–4. Instructions in Assembly Code

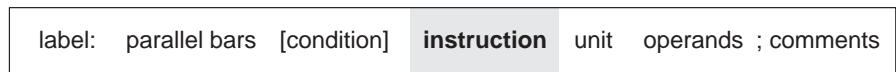


Table 6–1. Selected TMS320C6x Directives

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	Reserve 32 bits in memory and fill with specified value
.long value	
.word value	
.short value	Reserve 16 bits in memory and fill with specified value
.half 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.

Table 6–2. 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†				CMPGTD†	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.

6.5 Functional Units

The 'C6x CPU contains eight functional units, which are shown in Figure 6–5 and described in Table 6–3.

Figure 6–5. TMS320C6x Functional Units

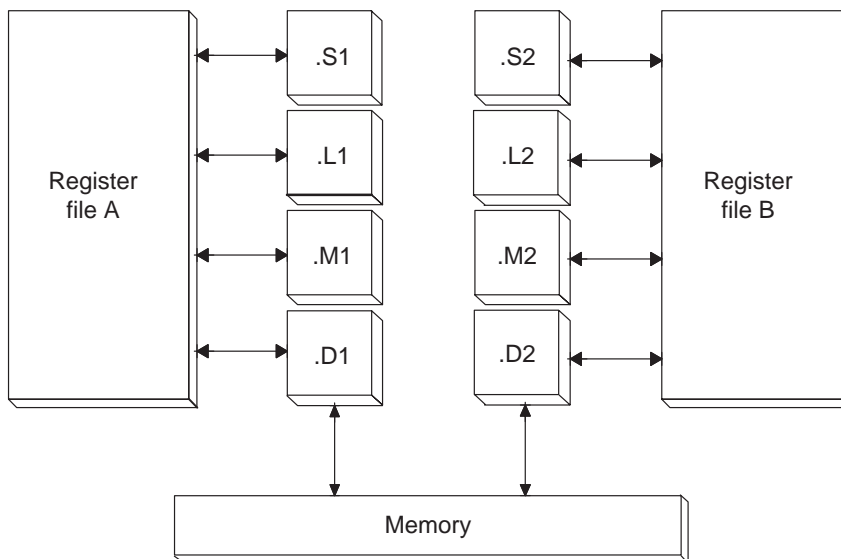


Table 6–3. Functional Units and Descriptions

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 × 16 bit multiplies 32 × 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 6–6 shows the position of the unit in a line of assembly code.

Figure 6–6. 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.

6.6 Operands

The 'C6x architecture requires that memory reads and writes move data between memory and a register. Figure 6–7 shows the position of the operands in a line of assembly code.

Figure 6–7. Operands in the Assembly Code

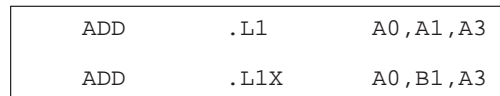


Instructions have the following requirements for operands in the assembly code:

- 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 6–8, 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 6–8. Operands in Instructions



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

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

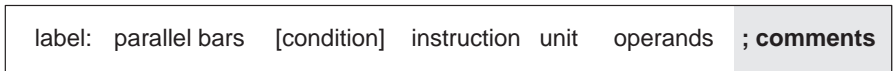
- Register operands indicate a register that contains the data.
- Constant operands specify the data within the assembly code.
- Pointer operands contain addresses of data values.

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

6.7 Comments

As with all programming languages, comments provide code documentation. Figure 6–9 shows the position of the comment in a line of assembly code.

Figure 6–9. 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.

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
7.1 Assembly Code	7-2
7.2 Assembly Optimizer Options and Directives	7-4
7.3 Writing Parallel Code	7-9
7.4 Using Word Access for Short Data and Doubleword Access for Floating-Point Data	7-20
7.5 Software Pipelining	7-30
7.6 Modulo Scheduling of Multicycle Loops	7-59
7.7 Loop Carry Paths	7-78
7.8 If-Then-Else Statements in a Loop	7-87
7.9 Loop Unrolling	7-95
7.10 Live-Too-Long Issues	7-102
7.11 Redundant Load Elimination	7-111
7.12 Memory Banks	7-119
7.13 Software Pipelining the Outer Loop	7-132
7.14 Outer Loop Conditionally Executed With Inner Loop	7-137

7.1 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 7.3 and section 7.4 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 7.5 and section 7.6 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 7.7 uses an IIR filter algorithm to discuss the problems with loop carry paths.
- Section 7.8 and section 7.9 discuss the problems encountered with if-then-else statements in a loop and how loop unrolling can be used to resolve them.
- Section 7.10 introduces live-too-long issues in your code.
- Section 7.11 uses a simple FIR filter algorithm to discuss redundant load elimination.
- Section 7.12 discusses the same FIR filter in terms of the interleaved memory bank scheme used by 'C6x devices.
- Section 7.13 and section 7.14 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 three sections following section 7.2, 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 three sections following section 7.2 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.

7.2 Assembly Optimizer Options and Directives

All directives and options that are described in the following sections are listed in greater detail in Chapter 4 of the *TMS320C6x Optimizing C Compiler User's Guide*.

7.2.1 The `-mt` Option and the `.no_mdep` Directive

Because the assembly optimizer has no idea where objects you are accessing are located when you perform load and store instructions, the assembly optimizer is by default very conservative in determining dependencies between memory operations. For example, let us say you have the following loop defined in a linear assembly file:

Example 7–1. Linear Assembly Block Copy

```
loop:
    ldw *reg1++, reg2
    add reg2, reg3, reg4
    stw reg4, *reg5++
[reg6] add -1, reg6, reg6
[reg6] b loop
```

The assembly optimizer will make sure that each store to “reg5” completes before the next load of “reg1”. A suboptimal loop would result if the store to address in reg5 is not in the next location to be read by “reg1”. For loops where “reg5” is pointing to the next location of “reg1”, this is necessary and implies that the loop has a loop carry path (Refer to Section 7.7 Loop Carry Paths for more information). For most loops, this is not the case, and you can inform the assembly optimizer to be more aggressive about scheduling memory operations. You can do this either by including the “.no_mdep” (no memory dependencies) directive in your linear assembly function or with the `-mt` option when you are compiling the linear assembly file. Be aware that if you are compiling both C code and linear assembly code in your application, that the `-mt` option has different meanings for both C and linear assembly code. In this case, use the `.no_mdep` directive in your linear assembly source files. For a full description on the implications of `.no_mdep` and the `-mt` option, refer to Appendix A, Memory Alias Disambiguation. Refer to Chapter 4 of the *Optimizing C Compiler User's Guide* for more information on both the `-mt` option and the `.no_mdep` directive.

7.2.2 The `.mdep` Directive

Should you need to specify a dependence between two or more memory references, use the `.mdep` directive. Annotate your code with memory reference symbols and add the `.mdep` directive to your linear assembly function.

Example 7–2. Block copy With `.mdep`

```
.mdep ld1, st1
LDW *p1++ {ld1}, inpl ; annotate memory reference ld1
; other code ...
STW outp2,*p2++ {st1} ; annotate memory reference st1
```

The `.mdep` directive indicates there is a memory dependence from the LDW instruction to the STW instruction. This means that the STW instruction must come after the LDW instruction. The `.mdep` directive does not imply that there is a memory dependence from the STW to the LDW. Another `.mdep` directive would be needed to handle that case.

7.2.3 The `.mptr` Directive

The `.mptr` directive gives the assembly optimizer information on how to avoid memory bank conflicts. The assembly optimizer will rearrange the memory references generated in the assembly code to avoid the memory bank conflicts that were specified with the `.mptr` directive. This means that code generated by the assembly optimizer will be faster by avoiding the memory bank conflicts. Example 7–3 shows linear assembly code and the generated loop kernel for a dot product without the `.mptr` directive.

Example 7–3. Linear Assembly Dot Product

```
dotp: .cproc ptr_a, ptr_b, cnt
      .reg  val1, val2, val3, val4
      .reg  prod1, prod2, sum1, sum2
      zero  sum1
      zero  sum2
loop: .trip 20, 20
```

Example 7–3. Linear Assembly Dot Product (Continued)

```

    ldh    *ptr_a++, val1
    ldh    *ptr_b++, val2
    mpy    val1, val2, prod1
    add    sum1, prod1, sum1
    ldh    *ptr_a++, val1
    ldh    *ptr_b++, val2
    mpy    val3, val4, prod2
    add    sum2, prod2, sum2
[cnt] add    -1, cnt, cnt
[cnt] b     loop
    add    sum1, sum2, sum1
    return sum1
    .endproc

<loop kernel generated>

loop:      ; PIPED LOOP KERNEL
[!A1]     ADD     .L2     B4,B6,B4
||
||     MPY     .M2X     B7,A0,B6
||
[ B0]     B      .S1     loop
||
||     LDH     .D2T2    *-B5(2),B6
||     LDH     .D1T1    *-A4(2),A0
||
[ A1]     SUB     .S1     A1,1,A1
||
[!A1]     ADD     .L1     A5,A3,A5
||     MPY     .M1X     B6,A0,A3
||
[ B0]     ADD     .L2     -1,B0,B0
||     LDH     .D2T2    *B5++(4),B7
||     LDH     .D1T1    *A4++(4),A0

```

If the arrays pointed to by `ptr_a` and `ptr_b` begin on the same bank, then there will be memory bank conflicts at every cycle of the loop due to how the LDH instructions are paired.

By adding the `.mptr` directive information, you can avoid the memory bank conflicts. Example 7–4 shows the linear assembly dot product with the `.mptr` directive and the resulting loop kernel.

Example 7-4. Linear Assembly Dot Product With `.mptr`

```

dotp: .cproc ptr_a, ptr_b, cnt
      .reg  val1, val2, val3, val4
      .reg  prod1, prod2, sum1, sum2
      zero sum1
      zero sum2
      .mptr ptr_a, x, 4
      .mptr ptr_b, x, 4
loop: .trip 20, 20
      ldh  *ptr_a++, val1
      ldh  *ptr_b++, val2
      mpy  val1, val2, prod1
      add  sum1, prod1, sum1
      ldh  *ptr_a++, val3
      ldh  *ptr_b++, val4
      mpy  val3, val4, prod2
      add  sum2, prod2, sum2
[cnt] add  -1, cnt, cnt
[cnt] b    loop
      add  sum1, sum2, sum1
      return sum1
      .endproc

```

<loop kernel generated>

```

loop:      ; PIPED LOOP KERNEL
[!A1]     ADD      .L2      B4,B6,B4
||
||      MPY      .M2X      B8,A0,B6
||
[ B0]     B        .S1      loop
||
||      LDH      .D2T2     *B5++(4),B8
||      LDH      .D1T1     *-A4(2),A0

[ A1]     SUB      .S1      A1,1,A1
||
[!A1]     ADD      .L1      A5,A3,A5
||      MPY      .M1X      B7,A0,A3
||
[ B0]     ADD      .L2      -1,B0,B0
||      LDH      .D2T2     *-B5(2),B7
||      LDH      .D1T1     *A4++(4),A0

```

The above loop kernel has no memory bank conflicts in the case where `ptr_a` and `ptr_b` point to the same bank. This means that you have to know how your data is aligned in C code before using the `.mptr` directive in your linear assembly code. The 'C6x compiler supports pragmas in C that align your data to a particular boundary (`DATA_ALIGN`, for example). Use these pragmas to align your data properly, so that the `.mptr` directives work in your linear assembly code.

7.2.4 The `.trip` Directive

The `.trip` directive is analogous to the `_nassert` intrinsic for C. The `.trip` directive looks like:

```
label: .trip minimum_value[, maximum_value[, factor]]
```

For example if you wanted to say that the linear assembly loop will execute some minimum number of times, use the `.trip` directive with just the first parameter. This example tells the assembly optimizer that the loop will iterate at least ten times.

```
loop: .trip 10
```

You can also tell the assembly optimizer that your loop will execute exactly some number of times by setting the `minimum_value` and `maximum_value` parameters to exactly the same value. This next example tells the assembly optimizer that the loop will iterate exactly 20 times.

```
loop: .trip 20, 20
```

The `maximum_value` parameter can also tell the assembly optimizer that the loop will iterate between some range. The `factor` parameter allows the assembly optimizer to know that the loop will execute a factor of value times. For example, the next loop will iterate either 8, 16, 24, 32, 40, or 48 times when this particular linear assembly loop is called.

```
loop: .trip 8, 48, 8
```

The `maximum_value` and `factor` parameters are especially useful when your loop needs to be interruptible. Refer to section 8.4.4, *Getting the Most Performance Out of Interruptible Code*.

7.3 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.

7.3.1 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 7–5 is a fixed-point dot product algorithm. The C code in Example 7–6 is a floating-point dot product algorithm.

Example 7–5. 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 7–6. 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);
}
```


7.3.2 Translating C Code to Linear Assembly

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

7.3.2.1 Fixed-Point Dot Product

Example 7–7 shows the linear assembly instructions used for the inner loop of the fixed-point dot product C code.

Example 7–7. 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.

7.3.2.2 Floating-Point Dot Product

Example 7–8 shows the linear assembly instructions used for the inner loop of the floating-point dot product C code.

Example 7–8. 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.

7.3.3 Linear Assembly Resource Allocation

The following rules affect the assignment of functional units for Example 7–7 and Example 7–8 (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 7–7 and Example 7–8, they are assigned as listed above.

The ADDSP instruction in Example 7–8 must use a .L unit.

7.3.4 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*.

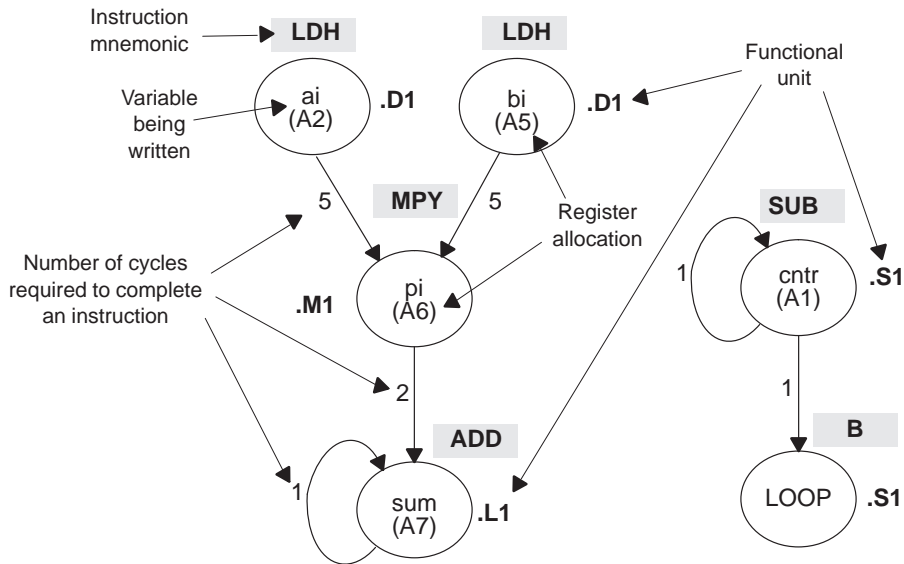
Use the following steps to draw a dependency graph:

- 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.

7.3.4.1 Fixed-Point Dot Product

Figure 7–1 shows the dependency graph for the fixed-point dot product assembly instructions shown in Example 7–7 and their corresponding register allocations.

Figure 7–1. 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 7.7 on page 7-78 for more information on loop carry paths.)

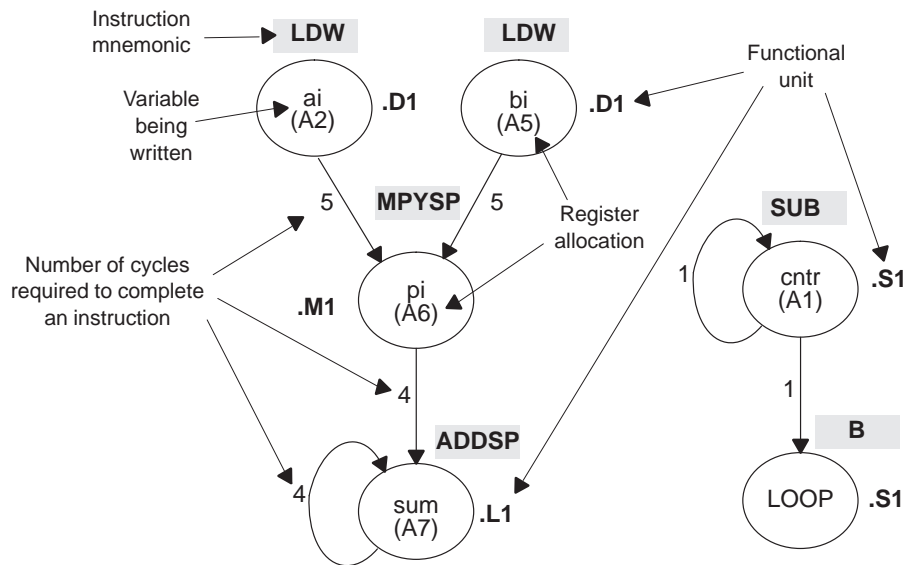
The dependency graph for this dot product algorithm has two separate parts because the decrement of the loop counter and the branch do not read or write any variables from the other part.

- ❑ 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.

7.3.4.2 Floating-Point Dot Product

Similarly, Figure 7–2 shows the dependency graph for the floating-point dot product assembly instructions shown in Example 7–8 and their corresponding register allocations.

Figure 7–2. 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 7.7 on page 7-78 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.

7.3.5 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.

7.3.5.1 Fixed-Point Dot Product

Example 7–9 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 7–9. Nonparallel Assembly Code for Fixed-Point Dot Product

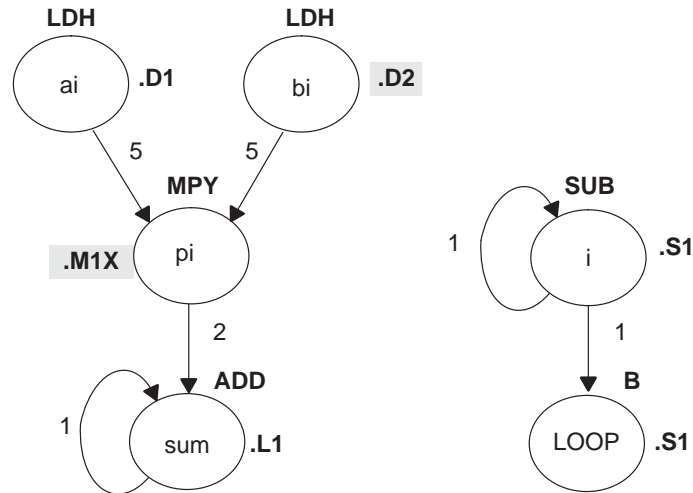
```

        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 7–3. The parallel assembly code is shown in Example 7–10.

Figure 7–3. Dependency Graph of Fixed-Point Dot Product with Parallel Assembly



Example 7–10. 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
    
```

Part III

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 7–9.

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.

7.3.5.2 Floating-Point Dot Product

Similarly, Example 7–11 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 7–11. Nonparallel Assembly Code for Floating-Point Dot Product

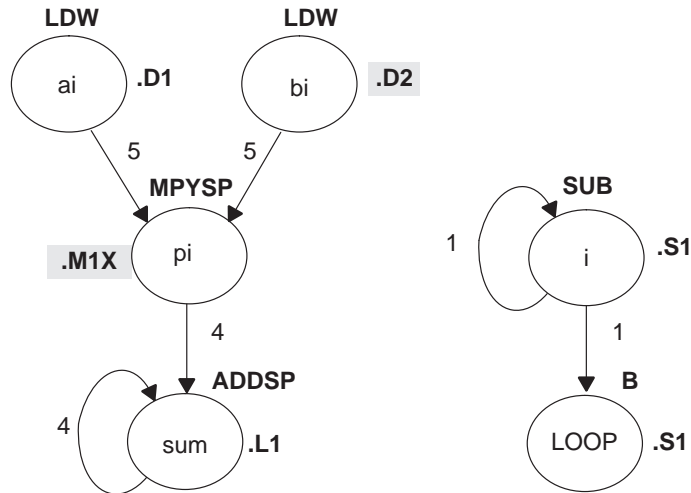
```

        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 7–4. The parallel assembly code is shown in Example 7–12.

Figure 7–4. Dependency Graph of Floating-Point Dot Product with Parallel Assembly



Example 7–12. 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
  
```

Part III

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 7–11 on page 7-17.

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.

7.3.6 Comparing Performance

Executing the fixed-point dot product code in Example 7–10 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 7–1 compares the performance of the nonparallel code with the parallel code for the fixed-point example.

Table 7–1. Comparison of Nonparallel and Parallel Assembly Code for Fixed-Point Dot Product

Code Example	100 Iterations	Cycle Count
Example 7–9 Fixed-point dot product nonparallel assembly	$2 + 100 \times 16$	1602
Example 7–10 Fixed-point dot product parallel assembly	$1 + 100 \times 8$	801

Executing the floating-point dot product code in Example 7–12 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 7–2 compares the performance of the nonparallel code with the parallel code for the floating-point example.

Table 7–2. Comparison of Nonparallel and Parallel Assembly Code for Floating-Point Dot Product

Code Example	100 Iterations	Cycle Count
Example 7–11 Floating-point dot product nonparallel assembly	$2 + 100 \times 21$	2102
Example 7–12 Floating-point dot product parallel assembly	$1 + 100 \times 10$	1001

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

The parallel code for the fixed-point example in section 7.3 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.

7.4.1 Unrolled Dot Product C Code

The fixed-point C code in Example 7–13 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 7–14. (For another example of loop unrolling, see section 7.9 on page 7-95.)

Example 7–13. 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);
}
```

Example 7–14. 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);
}
```

7.4.2 Translating C Code to Linear Assembly

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

7.4.2.1 Fixed-Point Dot Product

Example 7–15 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 7–15. Linear Assembly for Fixed-Point Dot Product Inner Loop with LDW

```
LDW    *a++,ai_il    ; load ai & a1 from memory
LDW    *b++,bi_il    ; load bi & b1 from memory
MPY    ai_il,bi_il,pi ; ai * bi
MPYH   ai_il,bi_il,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.

7.4.2.2 Floating-Point Dot Product

Example 7–16 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 7–16. 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.

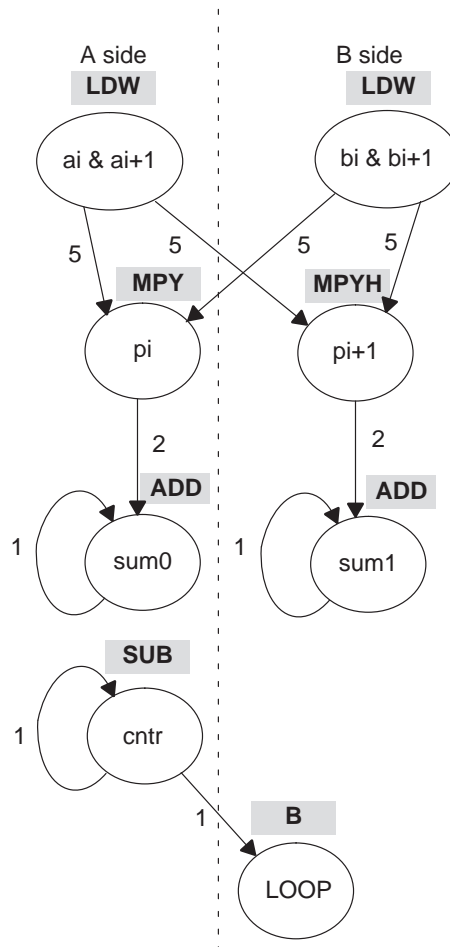
Two MPYSP instructions are now necessary to multiply the second set of array elements.

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

7.4.3 Drawing a Dependency Graph

The dependency graph in Figure 7–5 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.

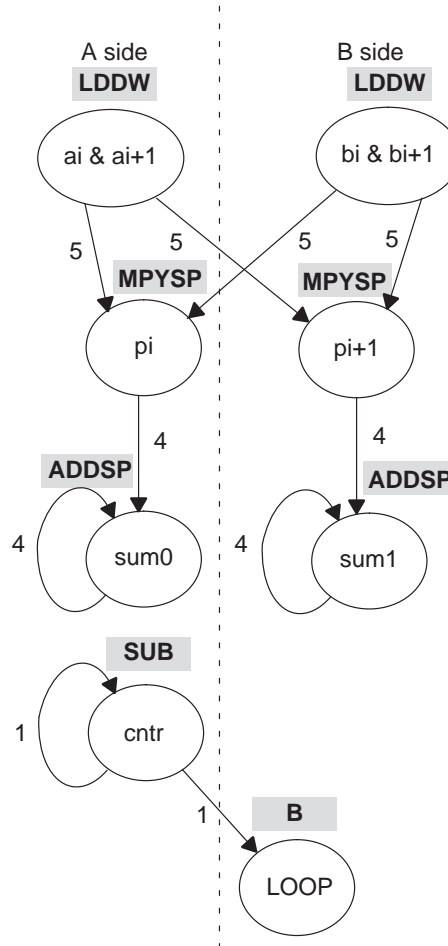
Figure 7–5. Dependency Graph of Fixed-Point Dot Product With LDW



Similarly, the dependency graph in Figure 7–6 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.

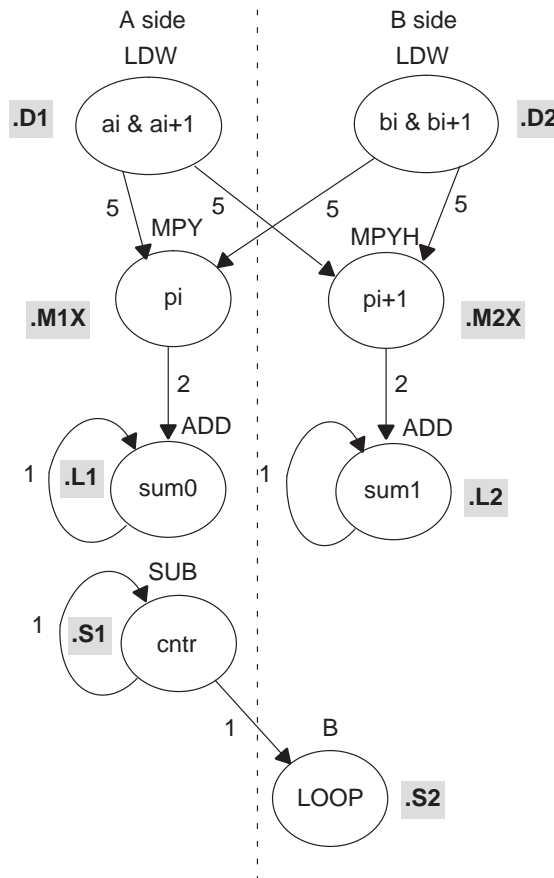
Figure 7–6. Dependency Graph of Floating-Point Dot Product With LDDW



7.4.4 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 7–7 and Figure 7–8 and in the instructions in Example 7–17 and Example 7–18. The .M1X and .M2X represent a path in the dependency graph crossing from one side to the other.

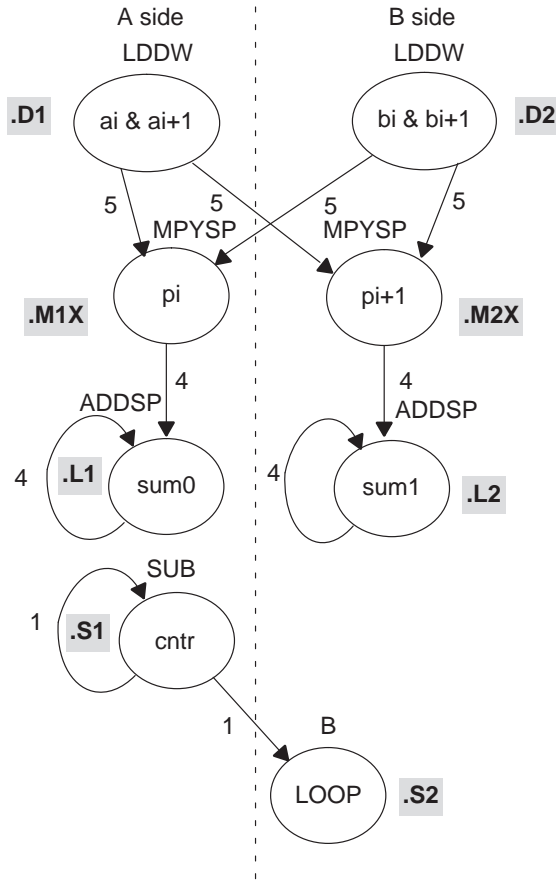
Figure 7–7. Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)



Example 7–17. 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 7–8. Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)



Example 7–18. 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

7.4.5 Final Assembly

Example 7–19 shows the final assembly code for the unrolled loop of the fixed-point dot product and Example 7–20 shows the final assembly code for the unrolled loop of the floating-point dot product.

7.4.5.1 Fixed-Point Dot Product

Example 7–19 uses LDW instructions instead of LDH instructions.

Example 7–19. 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 7–19 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.

7.4.5.2 Floating-Point Dot Product

Example 7–20 uses LDDW instructions instead of LDW instructions.

Example 7–20. 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 7–20 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.

7.4.6 Comparing Performance

Executing the fixed-point dot product with the optimizations in Example 7–19 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 ($1 + 8 \times 50 + 1$).

Table 7–3 compares the performance of the different versions of the fixed-point dot product code discussed so far.

Table 7–3. Comparison of Fixed-Point Dot Product Code With Use of LDW

Code Example	100 Iterations	Cycle Count
Example 7–9 Fixed-point dot product nonparallel assembly	$2 + 100 \times 16$	1602
Example 7–10 Fixed-point dot product parallel assembly	$1 + 100 \times 8$	801
Example 7–19 Fixed-point dot product parallel assembly with LDW	$1 + (50 \times 8) + 1$	402

Executing the floating-point dot product with the optimizations in Example 7–20 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 ($1 + 10 \times 50 + 7$).

Table 7–4 compares the performance of the different versions of the floating-point dot product code discussed so far.

Table 7–4. Comparison of Floating-Point Dot Product Code With Use of LDDW

Code Example	100 Iterations	Cycle Count
Example 7–11 Floating-point dot product nonparallel assembly	$2 + 100 \times 21$	2102
Example 7–12 Floating-point dot product parallel assembly	$1 + 100 \times 10$	1001
Example 7–20 Floating-point dot product parallel assembly with LDDW	$1 + (50 \times 10) + 7$	508

7.5 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 7–19 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 7–9 shows the dependency graph for the fixed-point dot product instructions. Example 7–21 shows the same dot product assembly code in Example 7–17 on page 7-25, 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 non-zero 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.

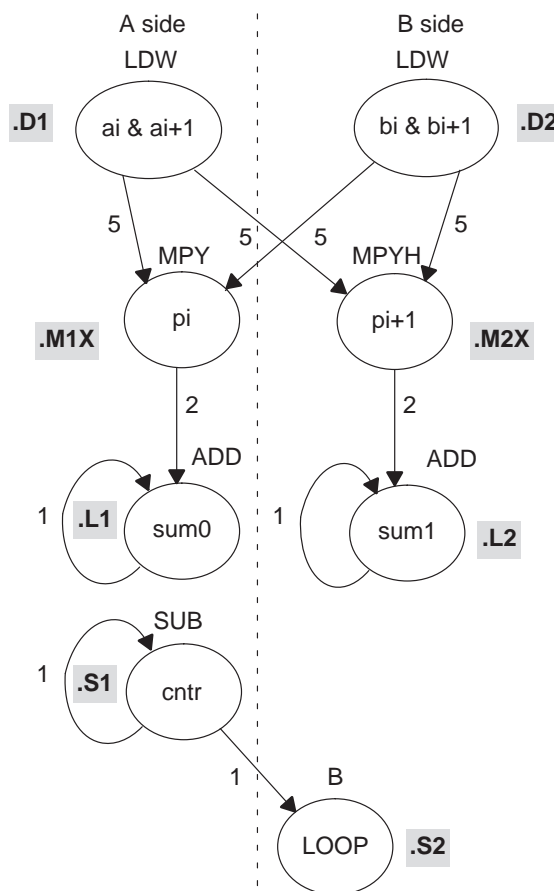
The floating-point dot product code in Example 7–20 needs ten cycles for each iteration of the loop: five cycles for the LDDWs, four cycles for the MPYSPs, and one cycle for the ADDSPs.

Figure 7–10 shows the dependency graph for the floating-point dot product instructions. Example 7–22 shows the same dot product assembly code in Example 7–18 on page 7-26, 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.

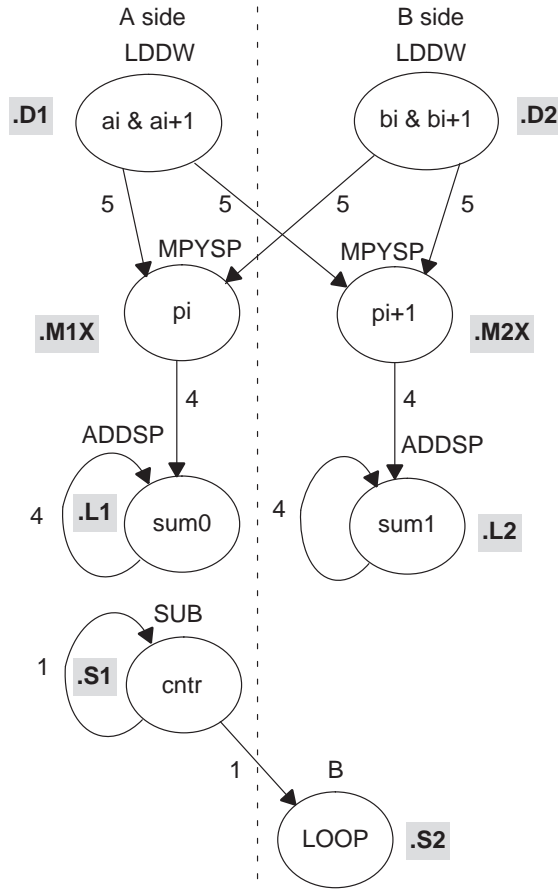
Figure 7–9. Dependency Graph of Fixed-Point Dot Product With LDW
(Showing Functional Units)



Example 7–21. 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 7–10. Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)



Example 7–22. 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

7.5.1 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.

7.5.1.1 Fixed-Point Example

The fixed-point code in Example 7–19 needs eight cycles for each iteration of the loop, so the iteration interval is eight.

Table 7–5 shows a modulo iteration interval scheduling table for the fixed-point dot product loop before software pipelining (Example 7–19). 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 7–5. 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.

7.5.1.2 Floating-Point Example

The floating-point code in Example 7–20 needs ten cycles for each iteration of the loop, so the iteration interval is ten.

Table 7–6 shows a modulo iteration interval scheduling table for the floating-point dot product loop before software pipelining (Example 7–20). 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 7–6. 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.

7.5.1.3 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 7–9 and Figure 7–10, 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.

7.5.1.4 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 7–7 shows a fully pipelined schedule for the fixed-point dot product example.

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

Unit / Cycle	Loop Prolog							
	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 7–7 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 7–8 shows a fully pipelined schedule for the floating-point dot product example.

Table 7–8. Modulo Iteration Interval Table for Floating-Point Dot Product (After Software Pipelining)

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

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.

7.5.1.5 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 7–23).

Example 7–23. 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 7–9 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 7–9. 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

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 7–27 on page 7-44.

7.5.2 Using the Assembly Optimizer to Create Optimized Loops

Example 7–24 shows the linear assembly code for the full fixed-point dot product loop. Example 7–25 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 7–24. Linear Assembly for Full Fixed-Point Dot Product

```

        .global _dotp
_dotp:  .cproc   a, b

        .reg    sum, sum0, sum1, cntr
        .reg    ai_il, bi_il, pi, pil

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

LOOP:   .trip 50
        LDW    *a++,ai_il       ; load ai & ai+1 from memory
        LDW    *b++,bi_il       ; load bi & bi+1 from memory
        MPY    ai_il,bi_il,pi    ; ai * bi
        MPYH   ai_il,bi_il,pil   ; ai+1 * bi+1
        ADD    pi,sum0,sum0     ; sum0 += (ai * bi)
        ADD    pil,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 7–25. Linear Assembly for Full Floating-Point Dot Product

```

        .global _dotp
_dotp:  .cproc   a, b

        .reg    sum, sum0, sum1, a, b
        .reg    ai:a1l, bi:b1l, pi, p1l

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

LOOP:   .trip 50
        LDDW    *a++,ai:a1l      ; load ai & ai+1 from memory
        LDDW    *b++,bi:b1l      ; load bi & bi+1 from memory
        MPYSP   a0,b0,pi         ; ai * bi
        MPYSP   a1,b1,p1l        ; ai+1 * bi+1
        ADDSP   pi,sum0,sum0     ; sum0 += (ai * bi)
        ADDSP   p1l,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

```

7.5.3 Final Assembly

Example 7–26 shows the assembly code for the fixed-point software-pipelined dot product in Table 7–7 on page 7-36. Example 7–27 shows the assembly code for the floating-point software-pipelined dot product in Table 7–8 on page 7-37. 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 7–7 and Table 7–8, 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.

7.5.3.1 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 7–26 requires a total of 58 cycles (7 + 50 + 1), which is a significant improvement over the 402 cycles required by the code in Example 7–19.

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 7–26. 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

```

7.5.3.2 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 7–27 requires a total of 74 cycles (9 + 50 + 15), which is a significant improvement over the 508 cycles required by the code in Example 7–20.

Example 7–27. 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      ; pil = 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      ;* pil = a1 b1
|| [A1] B        .S2    LOOP          ;** branch to loop
|| [A1] SUB      .S1    A1,1,A1        ;*** decrement loop counter

```

Example 7–27. 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  ;** pil = 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  ;*** pil = 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  ;**** pil = 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

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

    NOP

    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

```

7.5.3.3 Removing Extraneous Instructions

The code in Example 7–26 and Example 7–27 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 7–26:

- 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 7–27:

- 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 7–28 on page 7-48 and Example 7–29 on page 7-49.

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 7–28 indicate the changes in this code.)

Executing the dot product code in Example 7–28 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 7–29 indicate the changes in this code.)

Executing the dot product code in Example 7–29 with no extraneous LDDWs still requires a total of 74 cycles (9 + 41 + 9 + 15), but the code size is now larger.

Example 7–28. 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 7–28. Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads) (Continued)

LOOP:				ADDs	MPYs
	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	.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		

Example 7–29. Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads)

	MVK	.S1	A1,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	; pil = 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	;* pil = 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	** pil = 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	*** pil = a1 b1
[A1]	B	.S2	LOOP	**** branch to loop
[A1]	SUB	.S1	A1,1,A1	***** decrement loop counter

Example 7–29. 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	;**** pil = 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	; pil = 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	; pil = 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	; pil = 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	
	MPYSP	.M2X	A7,B7,B5	; pil = a1 b1	④
	ADDSP	.L1	A5,A8,A8	; sum0 += (ai bi)	④
	ADDSP	.L2	B5,B8,B8	; sum1 += (ai+1 bi+1)	
	MPYSP	.M1X	A6,B6,A5	; pi = a0 b0	
	MPYSP	.M2X	A7,B7,B5	; pil = a1 b1	⑤
	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)	⑧
	ADDSP	.L2	B5,B8,B8	; sum1 += (ai+1 bi+1)	
	ADDSP	.L1	A5,A8,A8	; sum0 += (ai bi)	⑨
	ADDSP	.L2	B5,B8,B8	; sum1 += (ai+1 bi+1)	

Example 7–29. 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    ;
```

7.5.3.4 Priming the Loop

Although Example 7–28 and Example 7–29 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 7–26 or Example 7–27, which do not have epilogs and, therefore, contain fewer instructions. (This technique can be used equally well with Example 7–28 or Example 7–29.)

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 7–30 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 7–30 requires a total of 65 cycles ($7 + 57 + 1$). Therefore, you are reducing the code size with a slight loss in performance.

Example 7–30. 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 7–31 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 7–31 requires a total of 81 cycles (7 + 59 + 15). Therefore, you are reducing the code size with a slight loss in performance.

Example 7–31. 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 7–31. 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    ;**** pil = 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
;

```

7.5.3.5 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 7–32 and Example 7–33. 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 7–30 and Example 7–31.) 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 7–30 and Example 7–31. The loop in Example 7–32 requires 63 cycles ($5 + 57 + 1$) and the loop in Example 7–31 requires 79 cycles ($5 + 59 + 15$).

Example 7–32. 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

```

Example 7–33. 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       ;**** pil = 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
;

```


7.5.4 Comparing Performance

Table 7–10 compares the performance of all versions of the fixed-point dot product code. Table 7–11 compares the performance of all versions of the floating-point dot product code.

Table 7–10. Comparison of Fixed-Point Dot Product Code Examples

Code Example	100 Iterations	Cycle Count
Example 7–9 Fixed-point dot product linear assembly	$2 + 100 \times 16$	1602
Example 7–10 Fixed-point dot product parallel assembly	$1 + 100 \times 8$	801
Example 7–19 Fixed-point dot product parallel assembly with LDW	$1 + (50 \times 8) + 1$	402
Example 7–26 Fixed-point software-pipelined dot product	$7 + 50 + 1$	58
Example 7–28 Fixed-point software-pipelined dot product with no extraneous loads	$7 + 43 + 7 + 1$	58
Example 7–30 Fixed-point software-pipelined dot product with no prolog or epilog	$7 + 57 + 1$	65
Example 7–32 Fixed-point software-pipelined dot product with smallest code size	$5 + 57 + 1$	63

Table 7–11. Comparison of Floating-Point Dot Product Code Examples

Code Example	100 Iterations	Cycle Count
Example 7–11 Floating-point dot product nonparallel assembly	$2 + 100 \times 21$	2102
Example 7–12 Floating-point dot product parallel assembly	$1 + 100 \times 10$	1001
Example 7–20 Floating-point dot product parallel assembly with LDDW	$1 + (50 \times 10) + 7$	508
Example 7–27 Floating-point software-pipelined dot product	$9 + 50 + 15$	74
Example 7–29 Floating-point software-pipelined dot product with no extraneous loads	$9 + 41 + 9 + 15$	74
Example 7–31 Floating-point software-pipelined dot product with no prolog or epilog	$7 + 59 + 15$	81
Example 7–33 Floating-point software-pipelined dot product with smallest code size	$5 + 59 + 15$	79

7.6 Modulo Scheduling of Multicycle Loops

Section 7.5 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.

7.6.1 Weighted Vector Sum C Code

Example 7–34 shows the C code for a weighted vector sum.

Example 7–34. 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];
    }
}
```

7.6.2 Translating C Code to Linear Assembly

Example 7–35 shows the linear assembly that executes the weighted vector sum in Example 7–34. 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 7–35. 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
```

7.6.3 Determining the Minimum Iteration Interval

Example 7–35 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.

7.6.3.1 Unrolling the Weighted Vector Sum C Code

Example 7–36 shows the C code for an unrolled version of the weighted vector sum.

Example 7–36. 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];
    }
}
```

7.6.3.2 Translating Unrolled Inner Loop to Linear Assembly

Example 7–37 shows the linear assembly that calculates $c[i]$ and $c[i+1]$ for the weighted vector sum in Example 7–36.

- 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 7–37. 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

```

7.6.3.3 Determining a New Minimum Iteration Interval

Use the following considerations to determine the minimum iteration interval for the assembly instructions in Example 7–37:

- 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 7–35.

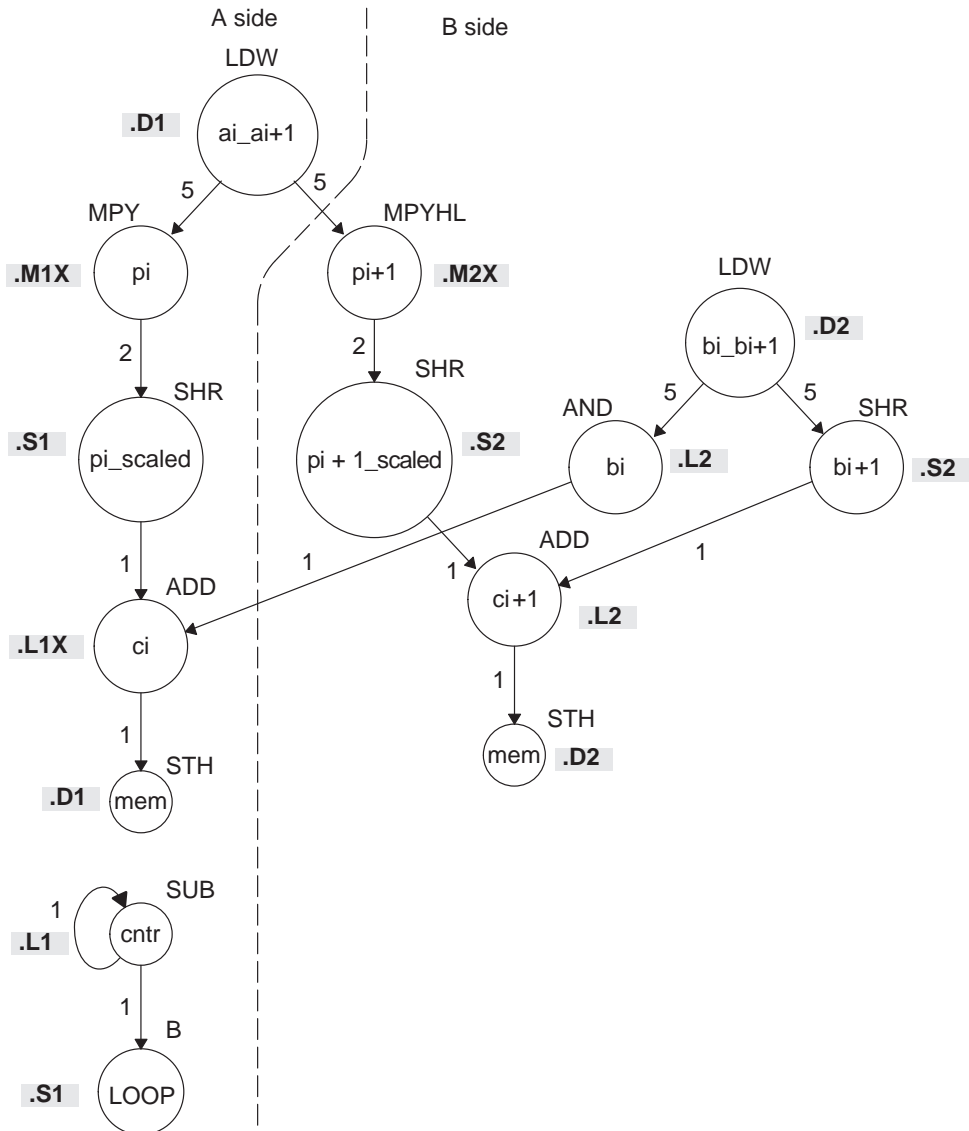
By using LDWs instead of LDHs, the program can do twice as much work in the same number of cycles.

7.6.4 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 7–11 shows the dependency graph divided evenly with a minimum iteration interval of 2.

Figure 7–11. Dependency Graph of Weighted Vector Sum



7.6.5 Linear Assembly Resource Allocation

Using the dependency graph, you can allocate functional units and registers as shown in Example 7–38. 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 7–38. 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

```

7.6.6 Modulo Iteration Interval Scheduling

Table 7–12 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 7–12 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 7–12 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 7–12. 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.

7.6.6.1 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 7–13 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 7–12. Dependency Graph of Weighted Vector Sum (Showing Resource Conflict)

Part III

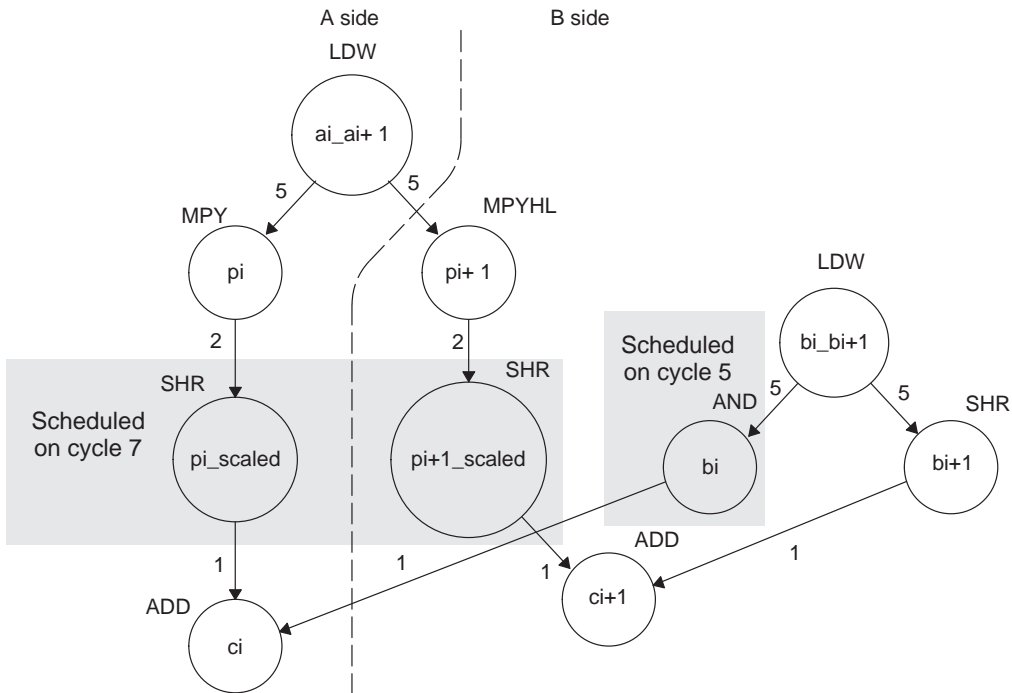


Table 7–13. Modulo Iteration Interval Table for Weighted Vector Sum With SHR Instructions

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 _{i+1}	* SHR bi _{i+1}	** SHR bi _{i+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
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 7–12.

7.6.6.2 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.

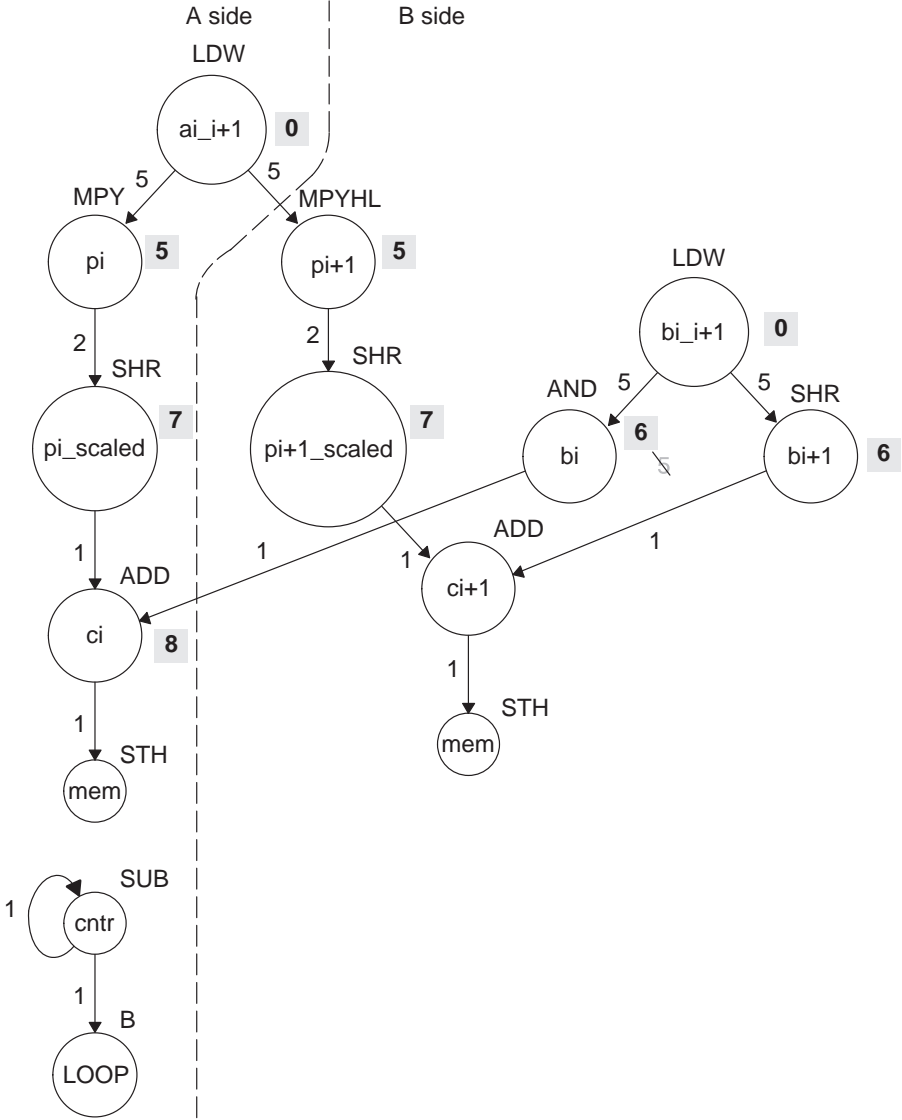
No value can be live in a register for more than the number of cycles in the loop. Otherwise, iteration $n + 1$ writes into the register before iteration n has read that register. Therefore, in a 2-cycle loop, a value is written to a register at the end of cycle n , then all children of that value must read the register before the end of cycle $n + 2$.

7.6.6.3 Solving the Live-Too-Long Problem

The live-too-long problem in Table 7–13 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 7–13 and Table 7–14.

Figure 7–13. Dependency Graph of Weighted Vector Sum (With Resource Conflict Resolved)



Note: Shaded numbers indicate the cycle in which the instruction is first scheduled.

Part III

Table 7–14. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

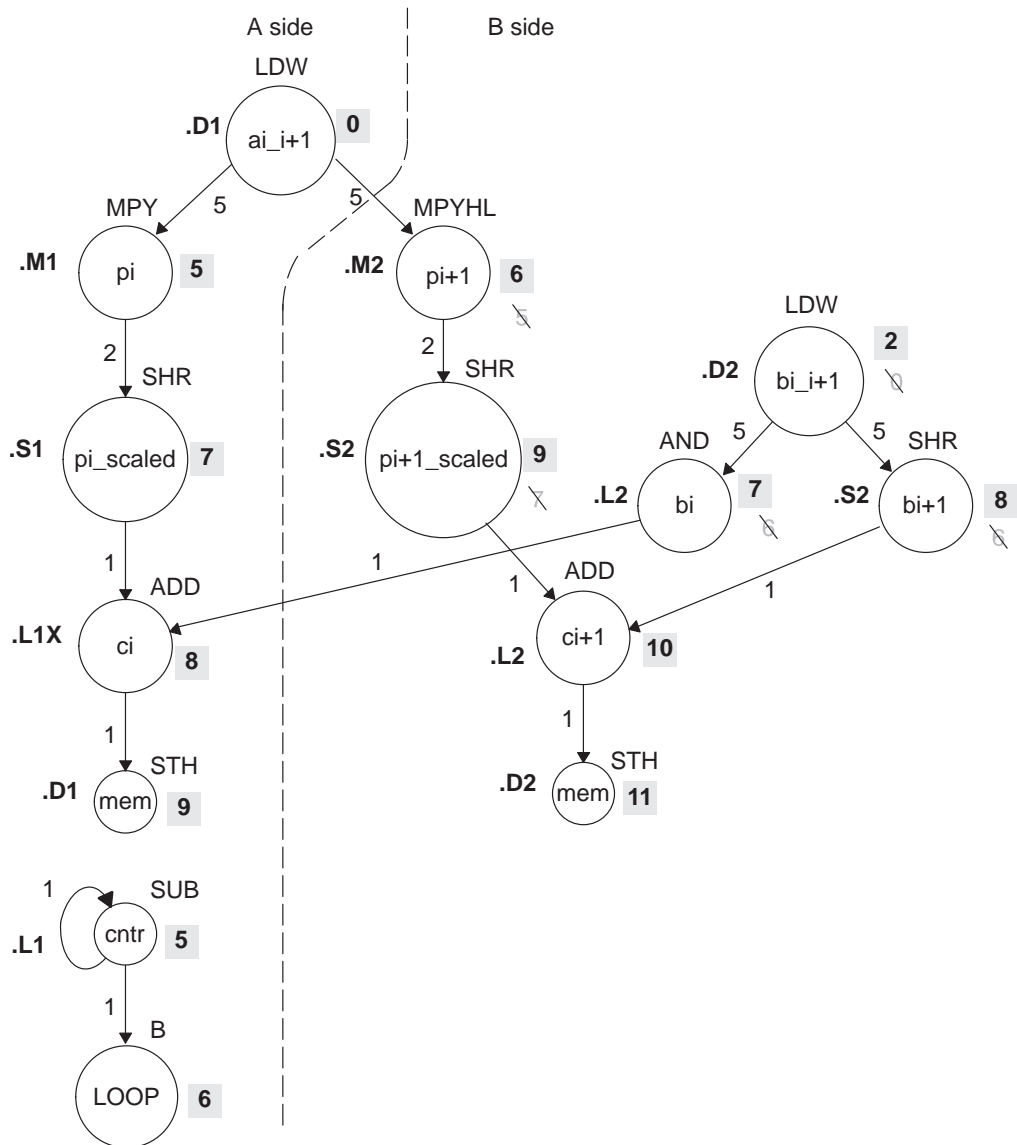
Unit/Cycle	0	2	4	6	8	10
.D1	LDW ai _{i+1}	* LDW ai _{i+1}	** LDW ai _{i+1}	*** LDW ai _{i+1}	**** LDW ai _{i+1}	***** LDW ai _{i+1}
.D2	LDW bi _{i+1}	* LDW bi _{i+1}	** LDW bi _{i+1}	*** LDW bi _{i+1}	**** LDW bi _{i+1}	***** LDW bi _{i+1}
.M1						
.M2						
.L1					ADD ci	* ADD ci
.L2				AND bi	* AND 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 7–13.

7.6.6.4 Scheduling the Remaining Instructions

Figure 7–14 shows the dependency graph with additional scheduling changes. The final version of the loop, with all instructions scheduled correctly, is shown in Table 7–15.

Figure 7–14. Dependency Graph of Weighted Vector Sum (Scheduling $ci + 1$)



Part III

Note: Shaded numbers indicate the cycle in which the instruction is first scheduled.

Table 7–15 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 7–15 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 7–15. 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 7–14.

7.6.7 Using the Assembly Optimizer for the Weighted Vector Sum

Example 7–39 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 7–39. Linear Assembly for Weighted Vector Sum

```

.global _w_vec

_w_vec: .cproc    a, b, c, m

        .reg     ai_il, bi_il, pi, pil, pi_il, pi_s, pil_s
        .reg     mask, bi, bil, ci, cil, 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_il ; ai & ai+1
        LDW      .D2      *b++,bi_il ; bi & bi+1
        MPY      .M1X     ai_il,m,pi  ; m * ai
        MPYHL    .M2X     ai_il,m,pil ; m * ai+1
        SHR      .S1      pi,15,pi_s ; (m * ai) >> 15
        SHR      .S2      pil,15,pil_s ; (m * ai+1) >> 15
        AND      .L2      bi_il,mask,bi ; bi
        SHR      .S2      bi_il,16,bil ; bi+1
        ADD      .L1X     pi_s,bi,ci  ; ci = (m * ai) >> 15 + bi
        ADD      .L2      pil_s,bil,cil ; ci+1 = (m * ai+1) >> 15 + bi+1
        STH      .D1      ci,*c++[2] ; store ci
        STH      .D2      cil,*c1++[2] ; store ci+1
[cntr] SUB      .L1      cntr,1,cntr ; decrement loop counter
[cntr] B        .S1      LOOP        ; branch to loop

        .endproc

```

7.6.8 Final Assembly

Example 7–40 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 7–40. 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 7–40. 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

```

7.7 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.

7.7.1 IIR Filter C Code

Example 7–41 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 7–41. 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;
    }
}
```

7.7.2 Translating C Code to Linear Assembly (Inner Loop)

Example 7–42 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.

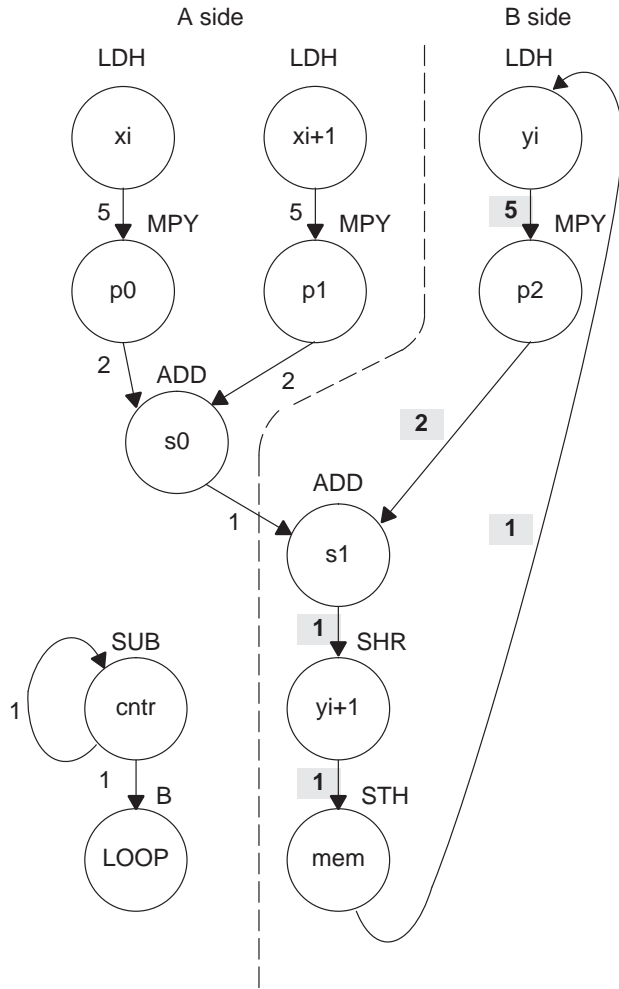
Example 7–42. Linear Assembly for IIR Inner Loop

LDH	<code>*xptr++,xi</code>	<code>; xi+1</code>
MPY	<code>c1,xi,p0</code>	<code>; c1 * xi</code>
LDH	<code>*xptr,xi+1</code>	<code>; xi+1</code>
MPY	<code>c2,xi+1,p1</code>	<code>; c2 * xi+1</code>
ADD	<code>p0,p1,s0</code>	<code>; c1 * xi + c2 * xi+1</code>
LDH	<code>*yptr++,yi</code>	<code>; yi</code>
MPY	<code>c3,yi,p2</code>	<code>; c3 * yi</code>
ADD	<code>s0,p2,s1</code>	<code>; c1 * xi + c2 * xi+1 + c3 * yi</code>
SHR	<code>s1,15,yi+1</code>	<code>; yi+1</code>
STH	<code>yi+1,*yptr</code>	<code>; store yi+1</code>
[cntr]SUB	<code>cntr,1,cntr</code>	<code>; decrement loop counter</code>
[cntr]B	<code>LOOP</code>	<code>; branch to loop</code>

7.7.3 Drawing a Dependency Graph

Figure 7–15 shows the dependency graph for the IIR filter. A loop carry path exists from the store of y_{i+1} to the load of y_i . 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 7–15. Dependency Graph of IIR Filter



Note: The shaded numbers show the loop carry path: $5 + 2 + 1 + 1 + 1 = 10$.

7.7.4 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 7–16, 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 7–16, the minimum iteration interval for the IIR filter is 2 because the total non-.M units on the A side is 5 ($5 \div 3$ 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 7–16. Resource Table for IIR Filter

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1	2 MPYs	2	.M2	MPY	1
.S1	B	1	.S2	SHR	1
.D1	2 LDHs	2	.D2	STH	1
.L1, .S1, or .D1	ADD & SUB	2	.L2 or .S2, .D2	ADD	1
Total non-.M units		5	Total non-.M units		3

However, the IIR has a data dependency constraint defined by its loop carry path. Figure 7–15 shows that if you schedule LDH y_i on cycle 0:

- The earliest you can schedule MPY p_2 is on cycle 5.
- The earliest you can schedule ADD s_1 is on cycle 7.
- SHR y_{i+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 second iteration can begin is cycle 10.

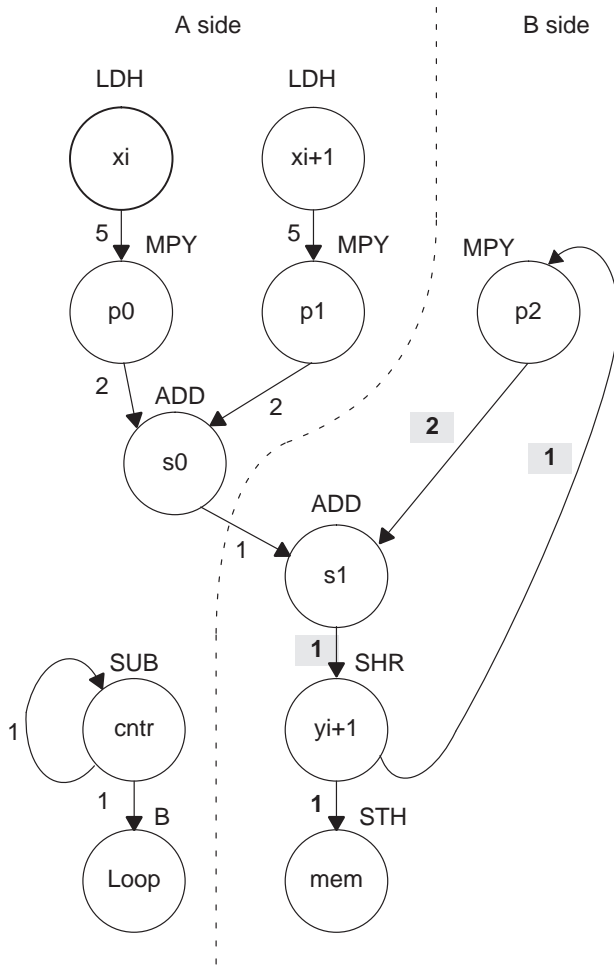
To determine the minimum loop carry path, add all of the numbers along the loop paths in the dependency graph. This means that this loop carry path is 10 ($5 + 2 + 1 + 1 + 1$).

Although the minimum iteration interval is the greater of the resource limits and data dependency constraints, an interval of 10 seems slow. Figure 7–16 shows how to improve the performance.

7.7.4.1 Drawing a New Dependency Graph

Figure 7–16 shows a new graph with a loop carry path of 4 (2 + 1 + 1). because the MPY p2 instruction can read y_{i+1} while it is still in a register, you can reduce the loop carry path by six cycles. LDH y_i 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 7–16. Dependency Graph of IIR Filter (With Smaller Loop Carry)



Note: The shaded numbers show the loop carry path: 2 + 1 + 1 = 4.

7.7.4.2 New 'C6x Instructions (Inner Loop)

Example 7–43 shows the new linear assembly from the graph in Figure 7–16, 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 7–43. 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
[cnt]	SUB	cnt,r1,cnt	; decrement loop counter
[cnt]	B	LOOP	; branch to loop

7.7.5 Linear Assembly Resource Allocation

Example 7–44 shows the same linear assembly instructions as those in Example 7–43 with the functional units and registers assigned.

Example 7–44. 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

7.7.6 Modulo Iteration Interval Scheduling

Table 7–17 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 7–17. 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.

7.7.7 Using the Assembly Optimizer for the IIR Filter

Example 7–45 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 7–45. Linear Assembly for IIR Filter

```

.global _iir
_iir: .cproc x, y, c1, c2, c3

    .reg    xi, xil, yil
    .reg    p0, p1, p2, s0, s1, cntr

    MVK     100,cntr                ; cntr = 100

    LDH     .D2 *y++,yil           ; yi+1

LOOP: .trip 100
    LDH     .D1 *x++,xi            ; xi
    MPY     .M1 c1,xi,p0           ; c1 * xi
    LDH     .D1 *x,xil            ; xi+1
    MPY     .M1X c2,xil,p1        ; c2 * xi+1
    ADD     .L1 p0,p1,s0          ; c1 * xi + c2 * xi+1
    MPY     .M2X c3,yil,p2        ; c3 * yi
    ADD     .L2X s0,p2,s1         ; c1 * xi + c2 * xi+1 + c3 * yi
    SHR     .S2 s1,15,yil         ; yi+1
    STH     .D2 yil,*y++         ; store yi+1
[cntr] SUB .L1 cntr,1,cntr        ; decrement loop counter
[cntr] B   .S1 LOOP              ; branch to loop

    .endproc

```

7.7.8 Final Assembly

Example 7–46 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 7–46 requires 408 cycles: $(4 \times 100) + 8$.

Example 7–46. 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

```

7.8 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.

7.8.1 If-Then-Else C Code

Example 7–47 contains a loop with an if-then-else statement. You either add `a[i]` to `sum` or subtract `a[i]` from `sum`.

Example 7–47. 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.

7.8.2 Translating C Code to Linear Assembly

Example 7–48 shows the linear assembly instructions needed to execute inner loop of the C code in Example 7–47.

Example 7–48. 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).

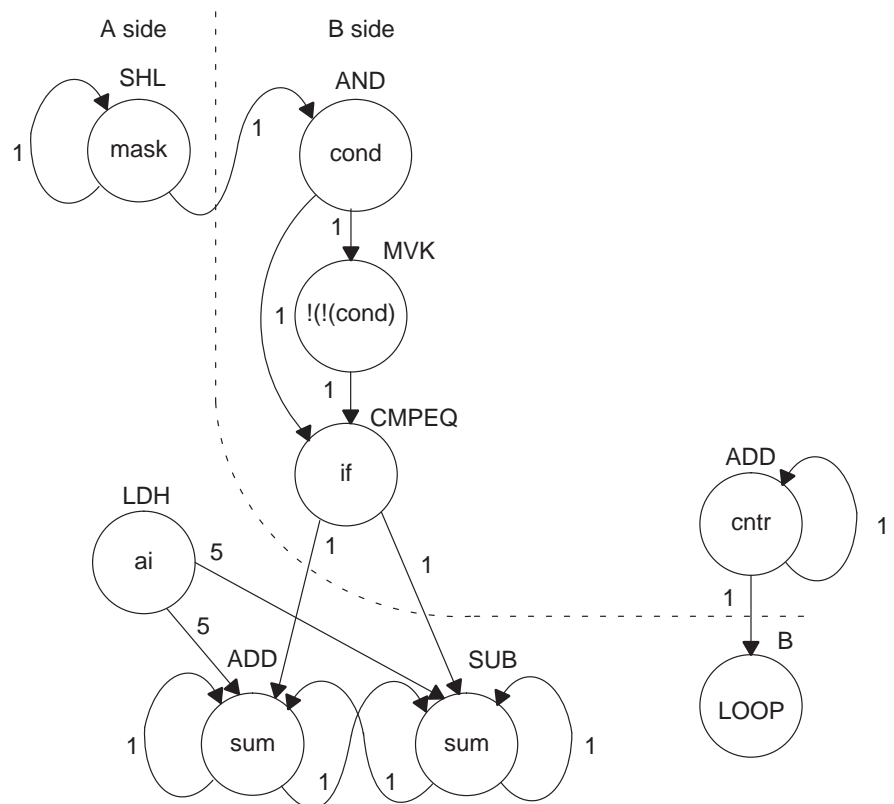
A conditional MVK performs the !(!(cond)) C statement. If the result of the bitwise AND is nonzero, a 1 is written into cond; if the result of the AND is 0, cond remains at 0.

7.8.3 Drawing a Dependency Graph

Figure 7–17 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 7–17. Dependency Graph of If-Then-Else Code



7.8.4 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 7–17 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 7–18, you can see that no one resource is used more than two times, so the minimum iteration interval is still 2.

Table 7–18. Resource Table for If-Then-Else Code

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1		0	.M2		0
.S1	SHL & B	2	.S2	MVK	1
.D1	LDH	1	.L2	CMPEQ	1
.L1, .S1, or .D1	ADD & SUB	2	.L2 or .S2	AND	1
			.L2, .S2, or .D2	ADD	1
Total non-.M units		5	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.

7.8.5 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 that no resource is used more than twice.

Example 7–49 shows the linear assembly with the functional units and registers that are used in the inner loop.

Example 7–49. 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

```

7.8.6 Final Assembly

Example 7–50 shows the final assembly code after software pipelining. The performance of this loop is 70 cycles ($2 \times 32 + 6$).

Example 7–50. 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

```

7.8.7 Comparing Performance

You can improve the performance of the code in Example 7–50 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 7–51. 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 7–51 shows the improved loop with a cycle count of 68 ($2 \times 32 + 4$). Table 7–19 compares the performance of Example 7–50 and Example 7–51.

Table 7–19. Comparison of If-Then-Else Code Examples

Code Example	Cycles	Cycle Count
Example 7–50 If-then-else assembly code	$(2 \times 32) + 6$	70
Example 7–51 If-then-else assembly code with loop count greater than 3	$(2 \times 32) + 4$	68

7.9 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 7–52, 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 7–51.

7.9.1 Unrolled If-Then-Else C Code

Example 7–52 shows the unrolled version of the if-then-else C code in Example 7–47 on page 7-87.

Example 7–52. 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);
}
```

7.9.2 Translating C Code to Linear Assembly

Example 7–53 shows the unrolled inner loop with 16 instructions and the possibility of achieving a loop with a minimum iteration interval of 3.

Example 7–53. 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

```

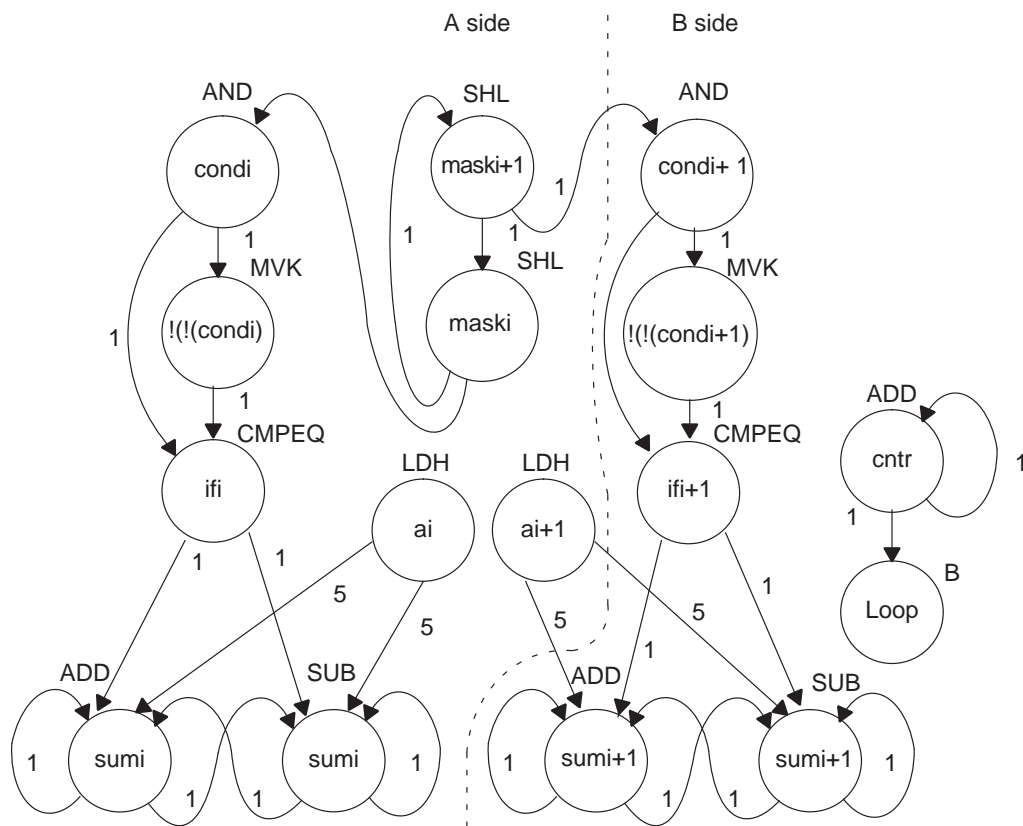
7.9.3 Drawing a Dependency Graph

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 7–18 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 7–18. Dependency Graph of If-Then-Else Code (Unrolled)



7.9.4 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 7–20, 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 7–20. Resource Table for Unrolled If-Then-Else Code

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1		0	.M2		0
.S1	MVK and 2 SHLs	3	.S2	MVK and B	2
.D1	2 LDHs	2	.L2	CMPEQ	1
.L1	CMPEQ	1	.L2 pr.S2	AND	1
.L1 or .S1	AND	1	.L2 ,.S2, or .D2	SUB and 2 ADDs	3
.L1, .S1, or .D1	ADD and SUB	2			
Total non-.M units		9	Total non-.M units		7

Part III

7.9.5 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 7–54 shows the linear assembly code with the functional units and registers.

Example 7–54. 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, ifil, a, ai, ail, cntr
        .reg   cdi, cdil, sumi, sumil, 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   sumil               ; sumi+1 = 0

LOOP:   .trip 32
        AND    .L1X cword,mask,cdi ; cdi = codeword & maski
[cdi]   MVK    .S1 1,cdil          ; !(!(cdi))
        CMPEQ  .L1X theta,cdil,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,cdil ; cdi+1 = codeword & maski+1
[cdil]  MVK    .S2 1,cdil         ; !(!(cdi+1))
        CMPEQ  .L2 theta,cdil,ifil ; (theta == !(!(cdi+1)))
        LDH    .D1 *a++,ail        ; a[i+1]
[ifil]  ADD    .L2 sumil,ail,sumil ; sum += a[i+1]
[!ifil] SUB    .D2 sumil,ail,sumil ; 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,sumil,sum      ; Add sumi and sumi+1 for ret value

        .return sum

        .endproc

```

7.9.6 Final Assembly

Example 7–55 shows the final assembly code after software pipelining. The cycle count of this loop is now 53: $(3 \times 16) + 5$.

Example 7–55. 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

```

7.9.7 Comparing Performance

Table 7–21 compares the performance of all versions of the if-then-else code examples.

Table 7–21. Comparison of If-Then-Else Code Examples

Code Example	Cycles	Cycle Count
Example 7–50 If-then-else assembly code	$(2 \times 32) + 6$	70
Example 7–51 If-then-else assembly code with loop count greater than 3	$(2 \times 32) + 4$	68
Example 7–55 Unrolled if-then-else assembly code	$(3 \times 16) + 5$	53

7.10 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 7.6.6.1, *Resource Conflicts*, on page 7-66 showed how to solve this problem simply by moving the parent to a later cycle. This is not always a valid solution.

7.10.1 C Code With Live-Too-Long Problem

Example 7–56 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 7–19 on page 7-104 shows a *split-join* path that causes this live-too-long problem.

Example 7–56. 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);
}
```

7.10.2 Translating C Code to Linear Assembly

Example 7–57 shows the assembly instructions that execute the inner loop in Example 7–56.

Example 7–57. 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

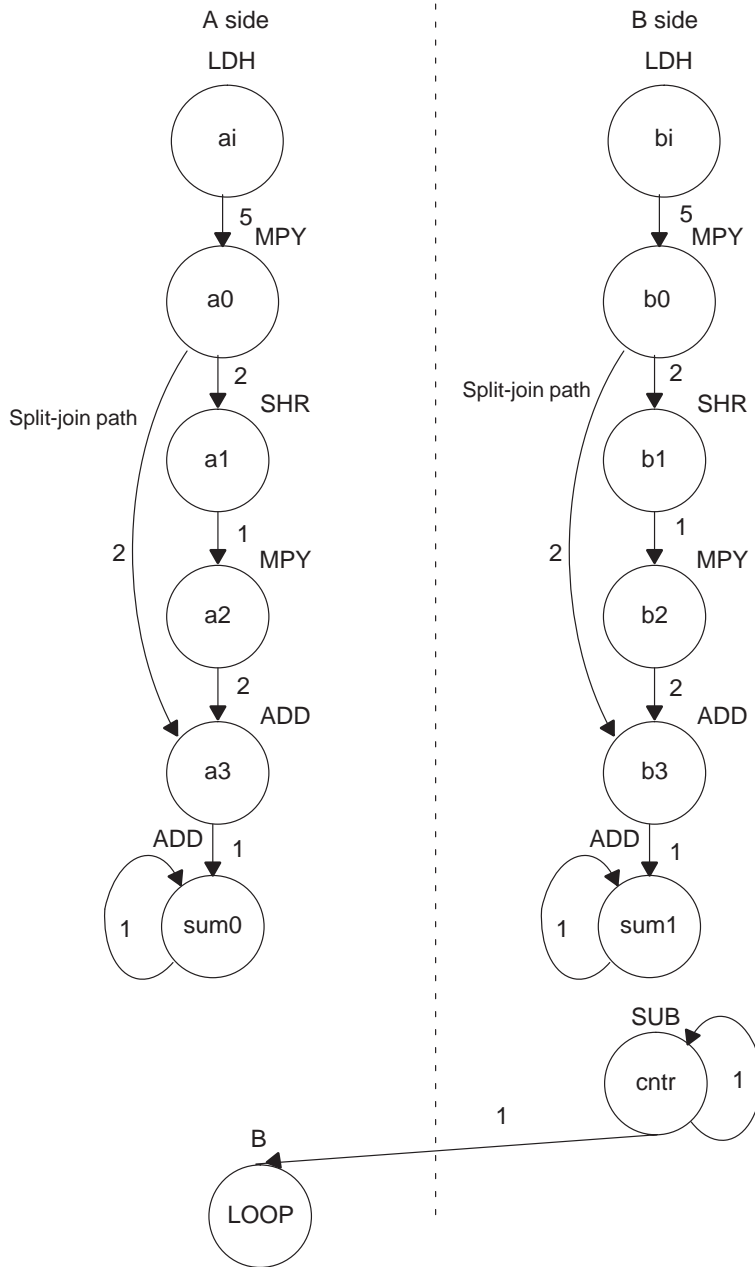
[ctr]SUB  ctr,1,ctr   ; decrement loop counter
[ctr]B    LOOP       ; branch to loop

```

7.10.3 Drawing a Dependency Graph

Figure 7–19 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 7–19. Dependency Graph of Live-Too-Long Code



7.10.4 Determining the Minimum Iteration Interval

Table 7–22 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 7–22. Resource Table for Live-Too-Long Code

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1	MPY	1	.M2	MPY	1
.S1	B and SHR	2	.S2	SHR	1
.D1	LDH	1	.D2	LDH	1
.L1, .S1, or .D1	2 ADDs	2	.L2, .S2, or .D2	2 ADDs and SUB	3
Total non-.M units		5	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 7.7, *Loop Carry Paths*, on page 7-78. In this example, a live-too-long problem determines the minimum iteration interval.

7.10.4.1 Split-Join-Path Problems

In Figure 7–19, 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.

7.10.4.2 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.

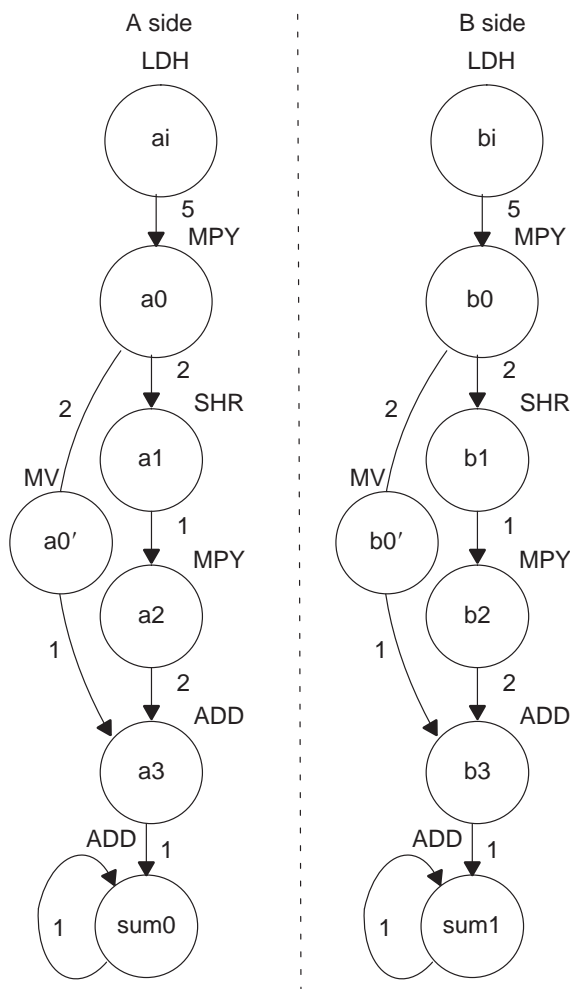
7.10.4.3 Inserting Moves

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.

7.10.4.4 Drawing a New Dependency Graph

Figure 7–20 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 7–20. Dependency Graph of Live-Too-Long Code (Split-Join Path Resolved)



7.10.5 Linear Assembly Resource Allocation

Example 7–58 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 7–58. 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

```

7.10.6 Final Assembly With Move Instructions

Example 7–59 shows the final assembly code after software pipelining. The performance of this loop is 212 cycles ($2 \times 100 + 11 + 1$).

Example 7–59. 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

```

Example 7–59. 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

```

7.11 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.

7.11.1 FIR Filter C Code

Example 7–60 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 7–60. 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.).

7.11.1.1 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.

7.11.1.2 New FIR Filter C Code

Example 7–61 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 7–61. 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;
    }
}
```

7.11.2 Translating C Code to Linear Assembly

Example 7–62 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.

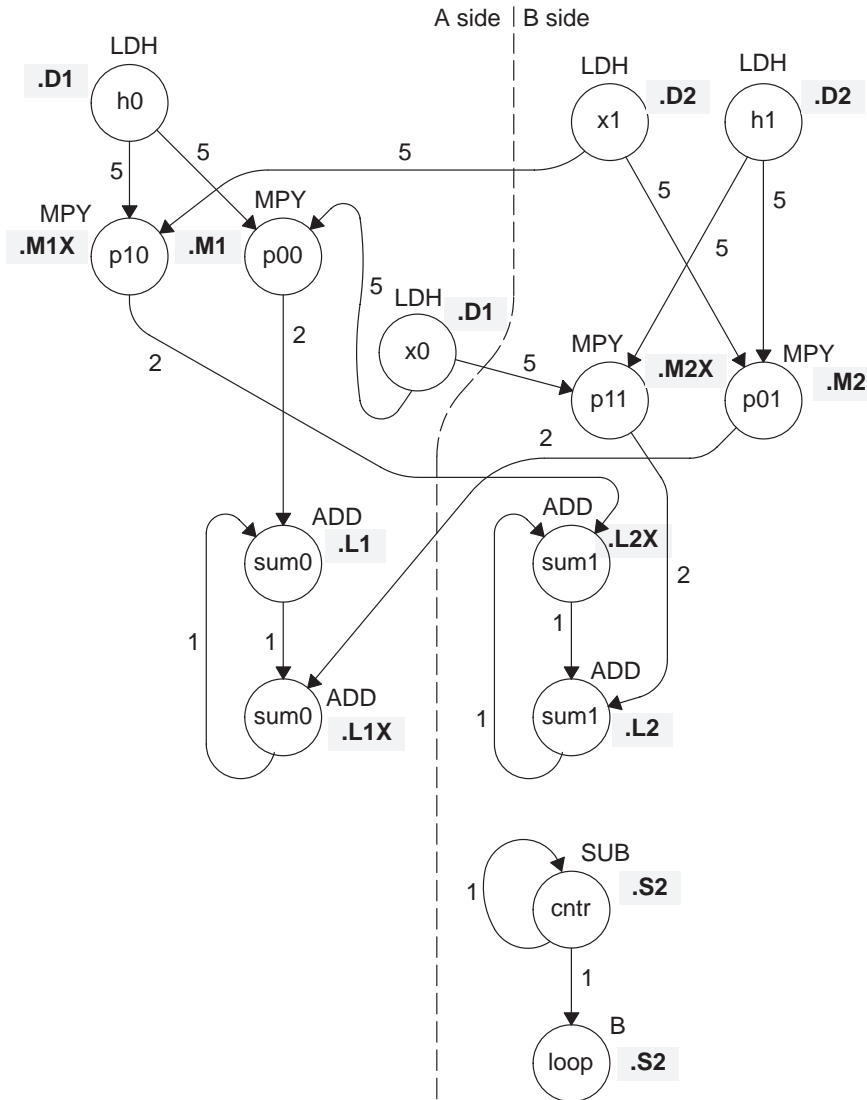
Example 7–62. Linear Assembly for FIR Inner Loop

	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

7.11.3 Drawing a Dependency Graph

Figure 7–21 shows the dependency graph of the FIR filter with redundant load elimination.

Figure 7–21. Dependency Graph of FIR Filter (With Redundant Load Elimination)



7.11.4 Determining the Minimum Iteration Interval

Table 7–23 shows that the minimum iteration interval is 2. An iteration interval of 2 means that two multiply-accumulates are executing per cycle.

Table 7–23. Resource Table for FIR Filter Code

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1	2 MPYs	2	.M2	2 MPYs	2
.S1		0	.S2	B	1
.D1	2 LDHs	2	.D2	2 LDHs	2
.L1, .S1, or .D1	2 ADDs	2	.L2, .S2, .D2	2 ADDs and SUB	3
Total non-.M units		4	Total non-.M units		6
1X paths		2	2X paths		2

7.11.5 Linear Assembly Resource Allocation

Example 7–63 shows the linear assembly with functional units and registers assigned.

Example 7–63. 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 7–63. Linear Assembly for Full FIR Code (Continued)

```

LOOP:      .trip 16

          LDH      .D2      *x_1++[2],x1      ; x1 = x[j+i+1]
          LDH      .D1      *h_1++[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

```

7.11.6 Final Assembly

Example 7–64 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 7.13, *Software Pipelining the Outer Loop*, on page 7-132 for information on how to reduce this overhead.

Example 7–64. 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]	

Example 7-64 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 cntnr
    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
    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
    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
    
```

①

②

③

④

⑤

⑥

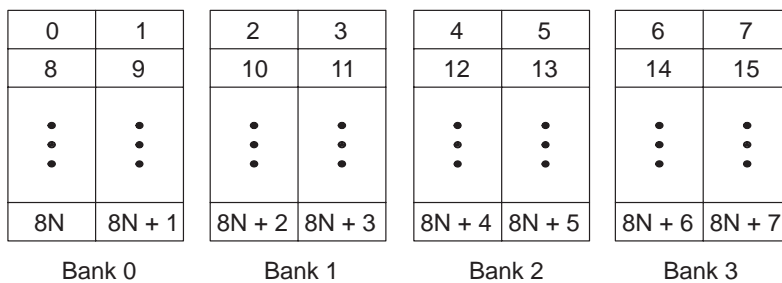
7.12 Memory Banks

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 7–22. 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.

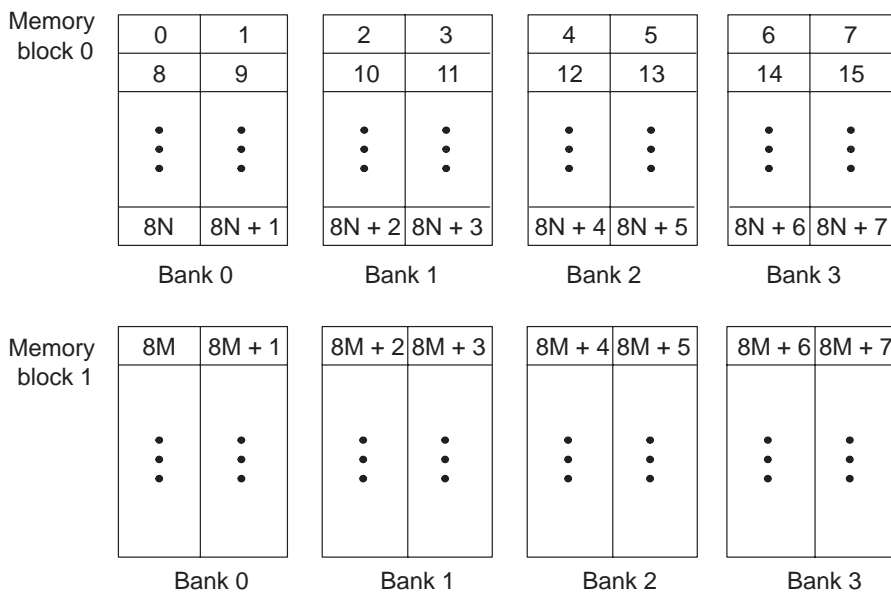
Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle, while the second value is read from memory. Two memory operations per cycle are allowed without any stall, as long as they do not access the same bank.

Figure 7–22. 4-Bank Interleaved Memory



For devices that have more than one memory block (see Figure 7–23), 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 7–23. 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 7–61 on page 7-112 is sufficient. This section describes a solution when two arrays must reside in the same memory block.

7.12.1 FIR Filter Inner Loop

Example 7–65 shows the inner loop from the final assembly in Example 7–64. 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 7–65 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 7–65. 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]

```

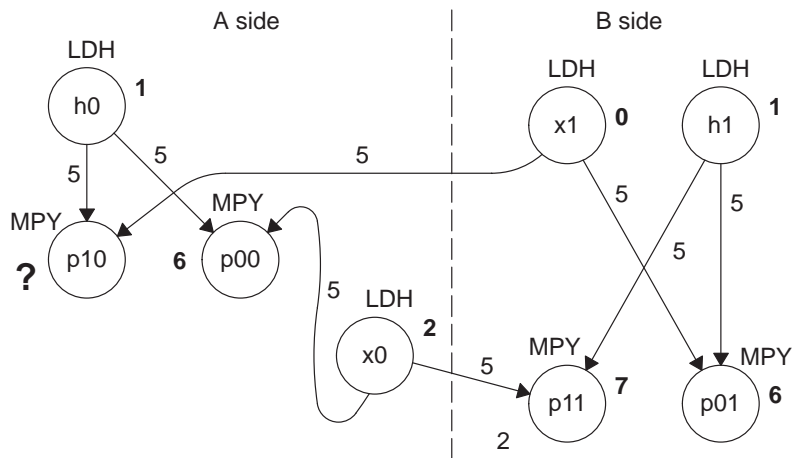
It is not always possible to fully control how arrays are aligned, especially if one of the arrays is passed into a function as a pointer and that pointer has different alignments each time the function is called. One solution to this problem is to write an FIR filter that avoids memory hits, regardless 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 7–24. 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 7–24. 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 7–24 *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.

7.12.2 Unrolled FIR Filter C Code

The main limitation in solving the problem in Figure 7–24 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 7–66 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 7–66. 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;
    }
}
```

7.12.3 Translating C Code to Linear Assembly

Example 7–67 shows the linear assembly for the unrolled inner loop of the FIR filter C code.

Example 7–67. 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

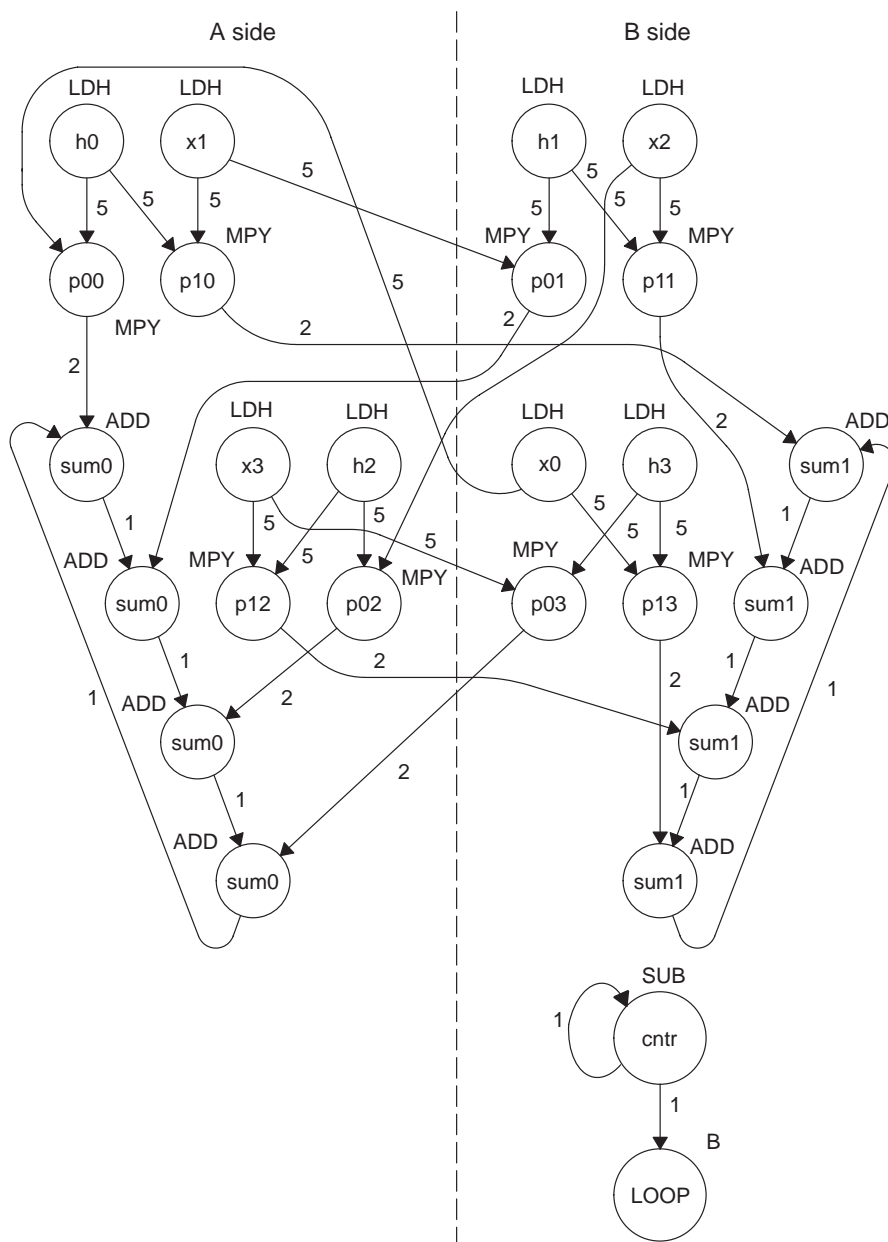
[cntr] SUB      cntr,1,cntr      ; decrement loop counter
[cntr] B        LOOP            ; branch to loop

```

7.12.4 Drawing a Dependency Graph

Figure 7–25 shows the dependency graph of the FIR filter with no memory hits.

Figure 7–25. Dependency Graph of FIR Filter (With No Memory Hits)



7.12.5 Linear Assembly for Unrolled FIR Inner Loop With .mptr Directive

Example 7–68 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 7–68. 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 7–68. 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_1++[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_1++[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_1++[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_1++[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

```

7.12.6 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.

7.12.7 Determining the Minimum Iteration Interval

Based on Table 7–24, 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 7–24. Resource Table for FIR Filter Code

(a) A side

(b) B side

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1	4 MPYs	4	.M2	4 MPYs	4
.S1		0	.S2	B	1
.D1	4 LDHs	4	.D2	4 LDHs	4
.L1, .S1, or .D1	4 ADDs	4	.L2, .S2, or .D2	4 ADDs and SUB	5
Total non-.M units		8	Total non-.M units		10
1X paths		4	2X paths		4

7.12.8 Final Assembly

Example 7–69 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.

7.12.9 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 7.13 addresses how to reduce this overhead by software pipelining the outer loop.

Table 7–25. Comparison of FIR Filter Code

Code Example		Cycles	Cycle Count
Example 7–64	FIR with redundant load elimination	$50(16 \times 2 + 9 + 6) + 2$	2352
Example 7–69	FIR with redundant load elimination and no memory hits	$50(8 \times 4 + 10 + 6) + 2$	2402

Example 7–69. 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]

```

Example 7–69. 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

```

7.13 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.

7.13.1 Unrolled FIR Filter C Code

Example 7–70 shows the FIR filter C code after unrolling the inner loop (identical to Example 7–66 on page 7-123).

Example 7–70. 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;
    }
}
```

7.13.2 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 7–69 on page 7-130) 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.

7.13.3 Final Assembly

Example 7–71 shows the final assembly for the FIR filter with a software-pipelined outer loop. Below the inner loop (starting on page 7-135), 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.

Example 7–71. 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 7–71. 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

```

Example 7–71. 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

```

7.13.4 Comparing Performance

The improved cycle count for this loop is 2006 cycles: $50 ((7 \times 4) + 6 + 6) + 6$. 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 7–26. Comparison of FIR Filter Code

Code Example	Cycles	Cycle Count
Example 7–64 FIR with redundant load elimination	$50 (16 \times 2 + 9 + 6) + 2$	2352
Example 7–69 FIR with redundant load elimination and no memory hits	$50 (8 \times 4 + 10 + 6) + 2$	2402
Example 7–71 FIR with redundant load elimination and no memory hits with outer loop software-pipelined	$50 (7 \times 4 + 6 + 6) + 6$	2006

7.14 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.

7.14.1 Unrolled FIR Filter C Code

Example 7–72 shows the same unrolled FIR filter C code that used in the previous example.

Example 7–72. 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;
    }
}
```


7.14.2 Translating C Code to Linear Assembly (Inner Loop)

Example 7–73 shows a list of linear assembly for the inner loop of the FIR filter C code (identical to Example 7–67 on page 7-124).

Example 7–73. 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

```

7.14.3 Translating C Code to Linear Assembly (Outer Loop)

Example 7–74 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 7–74. 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

7.14.4 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 7–75 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 7–72 on page 7-137.

Example 7–75. 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;
    }
}
```

7.14.5 Translating C Code to Linear Assembly (Inner Loop)

Example 7–76 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 7–76. 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

[sctr]  MPY      h01,x01,p00     ; p00 = h[i+0]*x[j+i+0]
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 7–76. 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

7.14.6 Translating C Code to Linear Assembly (Inner Loop and Outer Loop)

Example 7–77 shows the linear assembly with functional units assigned. (As in Example 7–68 on page 7-126, 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 7–77. 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 7-77. 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
[sctr]	MPY	.M1	h01,x01,p00	; p00 = h[i+0]*x[j+i+0]
	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 7–77. 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

```

7.14.7 Determining the Minimum Iteration Interval

Based on Table 7–27, the minimum iteration interval is 8. An iteration interval of 8 means that two multiply-accumulates per cycle are still executing.

Table 7–27. 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

7.14.8 Final Assembly

Example 7–78 shows the final assembly for the FIR filter with the outer loop conditionally executing in parallel with the inner loop.

Example 7–78. 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

|| [!A1] MPYLH  .M2X   A8,B8,B4        ; p10 = h[i+0]*x[j+i+1]
||      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

```

Example 7–78. 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]
  || MPYHL      .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
  || MPYHL      .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
  || MPYHL      .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 7–78. 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)

7.14.9 Comparing Performance

The cycle count of this code is 1612: $50(8 \times 4 + 0) + 12$. The overhead due to the outer loop has been completely eliminated.

Table 7–28. Comparison of FIR Filter Code

Code Example	Cycles	Cycle Count
Example 7–61 FIR with redundant load elimination	$50(16 \times 2 + 9 + 6) + 2$	2352
Example 7–69 FIR with redundant load elimination and no memory hits	$50(8 \times 4 + 10 + 6) + 2$	2402
Example 7–71 FIR with redundant load elimination and no memory hits with outer loop software-pipelined	$50(7 \times 4 + 6 + 6) + 6$	2006
Example 7–74 FIR with redundant load elimination and no memory hits with outer loop conditionally executed with inner loop	$50(8 \times 4 + 0) + 12$	1612

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
8.1 Overview of Interrupts	8-2
8.2 Single Assignment vs. Multiple Assignment	8-3
8.3 Interruptible Loops	8-5
8.4 Interruptible Code Generation	8-6
8.5 Interrupt Subroutines	8-11

8.1 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

8.2 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 8–1 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 8–1. 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 8–2 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 8–2. 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 8–2, can result in higher register pressure (Example 8–2 uses one more register than Example 8–1).

The next section describes how to generate interruptible and non-interruptible code with the 'C6x code generation tools.

8.3 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 7.9, *Loop Unrolling* and section 8.4.4, *Getting the Most Performance Out of Interruptible Code*). The disadvantage is that code size increases.

The next section describes how to automatically generate these different options with the 'C6x code generation tools.

8.4 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 allocate registers 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 a problem for looped code, use the `-mw` feedback option. If you see a “Cannot allocate machine registers” message after the message about searching for a software pipeline schedule, then you have a register pressure problem.
- ❑ 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 8.4.4.)
- ❑ 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 description of interruptible code generation, see the *TMS320C6x Optimizing C Compiler User's Guide*.

8.4.1 Level 0 - Specified Code is Guaranteed to Not Be Interrupted

The compiler does not disable interrupts. Thus, it is up to you 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` (no value specified) 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);
```

8.4.2 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);
```

8.4.3 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);
```

8.4.4 Getting the Most Performance Out of Interruptible Code

As stated in Chapter 4 and Chapter 7, the `.trip` directive and the `_nassert` intrinsic can be used to specify a maximum value for the trip count of a loop. This information can help to prevent performance loss when your loops need to be interruptible as in Example 8–3.

For example, if your application has an interrupt threshold of 100 cycles, you will use the `-mi100` option when compiling your application. Assume that there is a dot product routine in your application as follows:

Example 8–3. Dot Product With `_nassert` Guaranteeing Minimum Trip Count

```
int dot_prod(short *a, short *b, int n)
{
    int i, sum = 0;
    _nassert (n >= 20);
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

With the `_nassert` intrinsic, the compiler only knows that this loop will execute at least 20 times. Even with the interrupt threshold set at 100 by the `-mi` option, the compiler will still produce a 6-cycle loop for this code (with only one result computed during those six cycles) because the compiler has to expect that a value of greater than 100 may be passed into `n`.

After looking at the application, you discover that `n` will never be passed a value greater than 50 in the dot product routine. Example 8–4 adds this information to the `_nassert` statement as follows:

Example 8–4. Dot Product With `_nassert` Guaranteeing Trip Count Range

```
int dot_prod(short *a, short *b, int n)
{
    int i, sum = 0;
    _nassert ((n >= 20) && (n <= 50));
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

Now the compiler knows that the loop will complete in less than 100 cycles when it generates a 1-cycle kernel that must execute 50 times (which equals 50 cycles). The total cycle count of the loop is now known to be less than the interrupt threshold, so the compiler will generate the optimal 1-cycle kernel loop. You can do the same thing in linear assembly code by specifying both the minimum and maximum trip counts with the `.trip` directive.

Note: The compiler does not take memory bank conflicts into account. Because of this it is recommended that you are conservative with the threshold value.

Let us now assume the worst case scenario - the application needs to be interruptible at any given cycle. In this case, you will build your application with an interrupt threshold of one. It is still possible to regain some performance lost from setting the interrupt threshold to one. Example 8–5 shows this is where the `factor` option in `.trip` and using the modulus operator in an `_nassert` intrinsic are useful. (Refer to section 4.3.3.4, *Loop Unrolling*.)

Example 8–5. Dot Product With `_nassert` Guaranteeing Trip Count Range and Factor of 2

```
int dot_prod(short *a, short *b, int n)
{
    int i, sum = 0;
    _nassert ((n >= 20) && (n <= 50) && ((n%2)==0));
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

By enabling unrolling, performance has doubled from one result per 6-cycle kernel to two results per 6-cycle kernel. By allowing the compiler to maximize unrolling when using the interrupt threshold of one, you can get most of the performance back. Example 8–6 shows a dot product loop that will execute a factor of 4 between 16 and 48 times.

Example 8–6. Dot Product With `_nassert` Guaranteeing Trip Count Range and Factor of 4

```
int dot_prod(short *a, short *b, int n)
{
    int i, sum = 0;
    _nassert ((n >= 16) && (n <= 48) && ((n%4)==0));
    for (i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

The compiler knows that the trip count is some factor of four. The compiler will unroll this loop such that four iterations of the loop (four results are calculated) occur during the six cycle loop kernel. This is an improvement of four times over the first attempt at building the code with an interrupt threshold of one. The one drawback of unrolling the code is that code size increases, so using this type of optimization should only be done on key loops.

8.5 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.

8.5.1 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.

8.5.2 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 8–7. 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
```

8.5.3 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 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 8–8. 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 I
Introduction

Part II
C Code

Part III
Assembly Code

Part IV
Appendix

Memory Alias Disambiguation

This appendix is a tutorial and practical treatment on the problem of memory alias disambiguation on the 'C6x. If you write 'C6x linear assembly or hand-coded assembly, you will gain direct practical knowledge and advice on how to use the tools to handle this problem. If you write in C, you will gain insight into how the compiler handles this problem, as well as some practical advice.

The keywords to keep in mind are: memory aliases, dependence graphs, instruction scheduling

Topic	Page
A.1 Overview	A-2
A.2 Background	A-3
A.3 Tools Solution	A-12
A.4 Examples of Memory Alias Disambiguation	A-15
A.5 C Compiler and Alias Disambiguation	A-26
A.6 Memory Alias Disambiguation versus Memory Bank Conflict Detection	A-27
A.7 Summary	A-28

A.1 Overview

Memory alias disambiguation analyzes whether a dependence, through a memory location, exists between a given pair of instructions. Dependences between instructions are then used to determine the fastest legal schedule of those instructions.

This appendix begins by covering the topic of dependence. Next is a description of how dependences are represented in dependence graphs. These concepts are then extended to cover loops. Then, it addresses how dependence affects instruction scheduling. Next, the term memory alias disambiguation is introduced.

The focus then shifts to how the tools, particularly the assembly optimizer, handle memory alias disambiguation. However, if you write hand-coded assembly, you will find some useful concepts in these sections. Several detailed examples are presented.

Two final sections discuss how the C compiler handles memory alias disambiguation, and the differences between memory alias disambiguation and memory bank conflict detection.

Note that this appendix describes the 'C6x code generation tools for release 2.10 or greater.

A.2 Background

A.2.1 Data Dependence Between Instructions

One dictionary definition of dependence is "the state of being determined, influenced, or controlled by something else". In the world of software, the objects being influenced can be modules of code, specific functions, blocks within functions, individual statements, data structures, variables, etc. Further, the relationship can be interdependent, for example, two objects can depend on each other. This appendix refers to only one kind of dependence relationship: the data dependence between individual assembly language instructions.

At this level, dependence is evaluated between pairs of instructions. Two instructions have a dependence when they reference (read or write) the same machine resource, for example, register, memory location, status bit, and so forth. So, a dependence is characterized by the following pieces of information:

- The first instruction
- The second instruction
- The resource both instructions reference
- The first instruction reference - read or write?
- The second instruction reference - read or write?

This information is summarized in the following table. The entries in the table are the formal name for that form of dependence.

Table A–1. Dependence Table

		<i>Instruction 2 Reference</i>	
		Read	Write
<i>Instruction 1 Reference</i>	Read	Input	Anti-
	Write	Flow	Output

Flow dependence is the most common and intuitive form of dependence. In this relationship, one instruction writes an output which a following instruction reads as an input. For example:

```
I1:   ADDK  10,A2    ; writes output to A2
I2:   STW   A2,*A3   ; reads input from A2
```

Instruction I1 writes an output in the register A2, and instruction I2 reads A2 as an input.

Anti-dependence is less common than flow dependence, but it is no less important. In this relationship, one instruction reads a resource as an input, and a following instruction writes a result to that same resource. For example:

```
I3:   STWA3,*A4   ; reads A4
I4:   ZERO  A4   ; writes A4
```

Instruction I3 reads A4 for a data address, while instruction I4 clears out A4 for some later computation.

Note a key difference from flow dependence: the anti-dependence exists because of the reuse of the resource, and not because of a transfer of actual data. So, one easy way to remove an anti-dependence is to choose a different resource in the second instruction. In this example, instruction I4 could use the register A5 instead:

```
I3:   STWA3,*A4   ; reads A4
I4$:  ZERO  A5   ; writes A5 ==> no anti-dependence
```

Since anti-dependence through a register is so easy to avoid, it is less common. However, anti-dependence through a memory location is usually not as easy to rewrite.

Output dependence is also not very common. One example is using a register to pass a value to a function. You will see a register load followed by a branch to a function which is known or presumed to overwrite that same register.

```
I5:   LDW*A8,A4   ; load A4
I6:   B  func     ; branch to func ==> overwrites A4
```

The relationship between these two instructions is an output dependence.

Input dependence is common, but it is usually ignored. One exception is accessing memory mapped peripherals. In that case, reading a memory location can trigger a side effect such as incrementing register, or starting a memory block transfer. You generally want to recognize a dependence between any instruction which triggers such side effects and any other memory reference.

The term independent is used to describe two instructions which do not reference any of the same resources. Note the difference between the terms independent and anti-dependence.

A.2.2 Dependence Graphs

Dependence graphs are used to represent the dependences between a set of instructions.

From the C fragment:

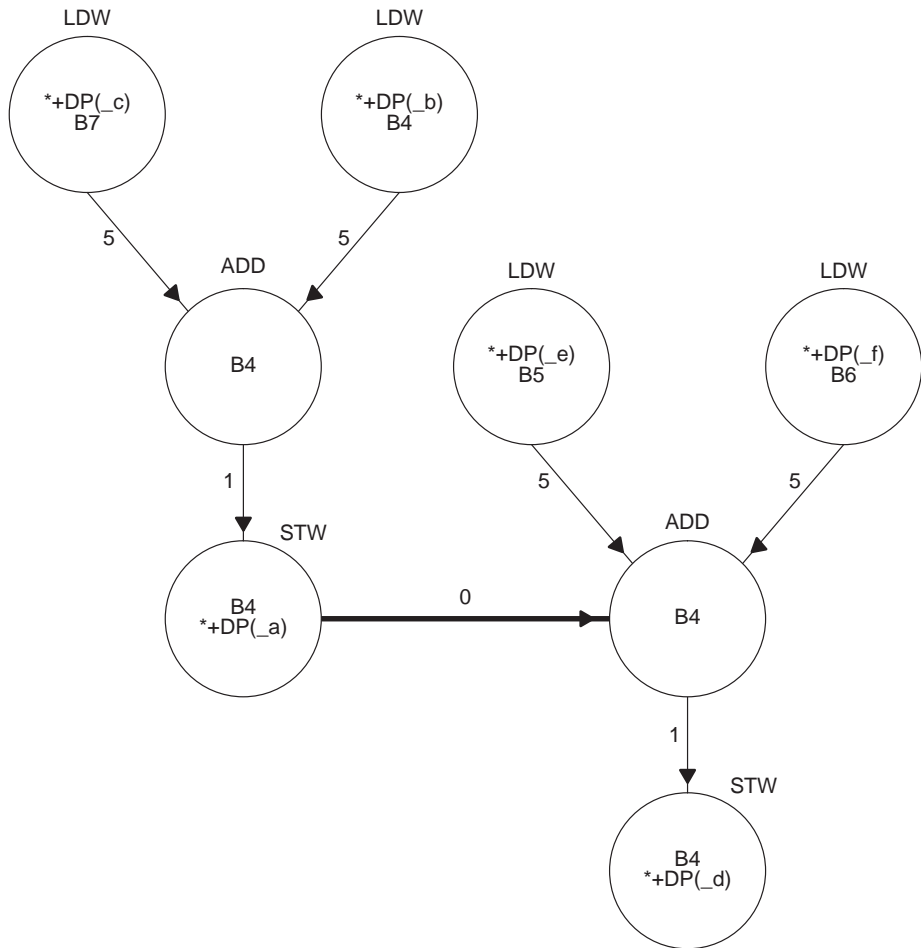
```
a = b + c;
```

```
d = e + f;
```

Here is hand modified compiler output which illustrates the serial instruction stream for those statements:

```
;-----  
; 5 | a = b + c;  
;-----  
LDW*+DP(_b),B4 ; |5|  
LDW*+DP(_c),B7 ; |5|  
ADDB7,B4,B4 ; |5|  
STWB4,*+DP(_a) ; |5|  
;-----  
; 6 | d = e + f;  
;-----  
LDW*+DP(_e),B5 ; |6|  
LDW*+DP(_f),B6 ; |6|  
ADDB6,B5,B4 ; |6|  
STWB4,*+DP(_d) ; |6|
```

Here is the dependence graph:



The circles are called nodes, and the arrows connecting the circles are called edges. There is one node for each instruction, and one edge for each dependence. Since instructions can have multiple dependences as both input and output, nodes in the graph can have multiple edges leading in and out.

With regard to a single edge, the node at the head of the arrow is termed the “parent” of the “child” node at the tail of the arrow.

The instruction is written immediately over the corresponding node.

For loads and stores, both operands are written inside the node. For other instructions, only the result operand is written, because the input operands are the result operands from the parent nodes.

The numbers next to the edges indicate how many cycles of pipeline latency you must wait for that result to be available to the child node. A latency of 0 indicates those instructions can be scheduled in parallel.

A common misconception is to imagine data flowing along the edges. That is true for the common case of flow dependence (thus the name). But note the edge from the first `STW` to the second `ADD`. That is an anti-dependence on `B4`. No data is flowing in this dependence. The dependence is based on the reuse of register `B4`. Anti-dependence is shown in the graph with a boldface arrow. All of the other edges are flow dependences. In flow dependence, the associated operand is always the last (or only) operand shown in the parent node. For anti-dependence, the associated operand is the first (or only) operand shown in the child node.

In other literature, a node may be called a vertex (vertices for plural), and an edge may be called an arc.

If you are accustomed to the dependence graphs that appear in Chapter 7 of the 'C6x Programmer's Guide, you will notice some differences. The graphs are called dependency graphs. An edge is called a path. Only one operand is shown in load/store instructions, and anti-dependence is not addressed.

A.2.3 Data Dependence in Loops

So far, we have only looked at relationships between instructions in a simple straight-line block of code. Considering dependence between instructions in a loop requires some extensions to those concepts.

A dependence graph for instructions in a loop looks the same, but there is a key difference. Each node, instead of representing one instruction, now represents every instance of that instruction in every loop iteration. The same is true of the edges.

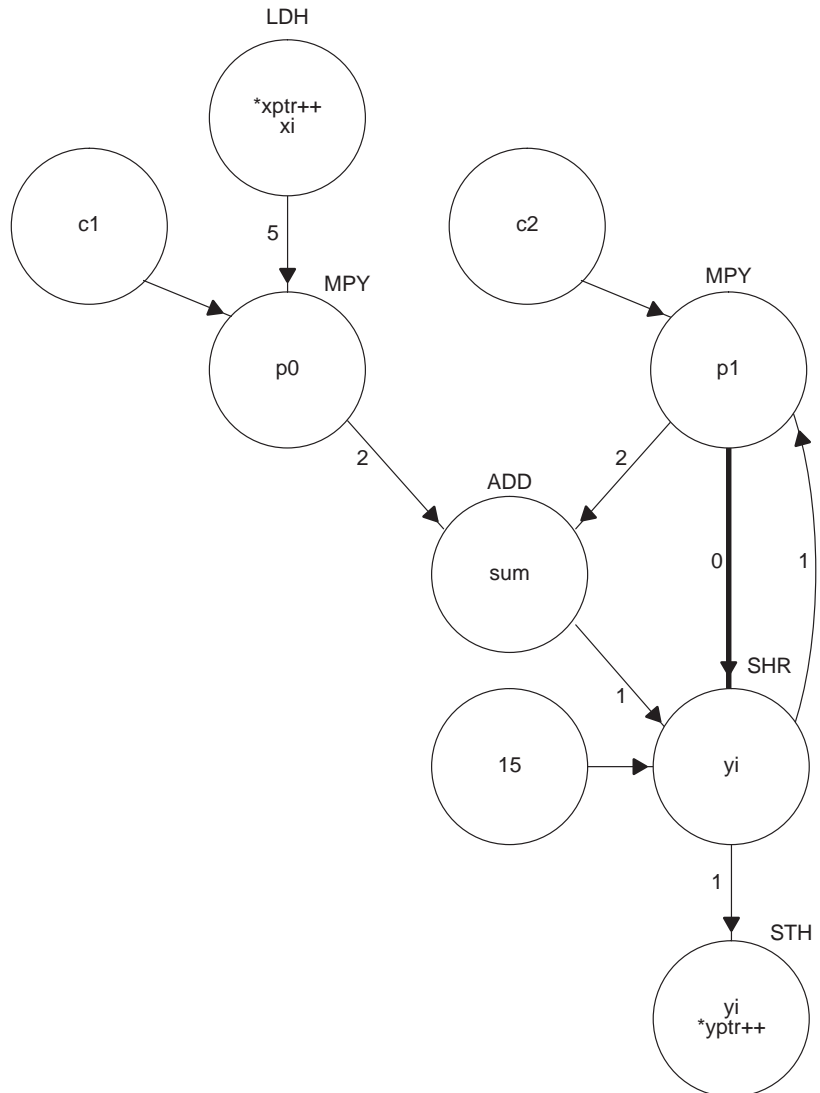
When considering dependence graphs in straight-line code, you do not have to worry about the direction of the dependence, because it is always the same: from an earlier serial instruction to a later one. In loops, however, an instruction late in the loop can generate a result which is used, in the next loop iteration, by an instruction earlier in the loop. We say such dependences are carried by the loop. In that case, the edge in the dependence graph goes the other direction. Here is a linear assembly code fragment:

```

loop:
    LDH*xptr++,xi
    MPYc1,xi,p0
    MPYc2,yi,p1 ; reads yi from prior iteration
    ADDp0,p1,sum
    SHRsum,15,yi ; writes yi for next iteration
    STHyi,*yptr++
; decrement and branch to loop

```

Here is the dependence graph:



Consider the instructions which reference y_i . Note the flow dependence, carried to the next iteration by the loop, on y_i from the `SHR` to `MPY`, because the `SHR` writes a value to y_i which `MPY` reads. Also note the anti-dependence, not carried by the loop, on y_i from the `MPY` to the `SHR`, because the `MPY` must read y_i before `SHR` writes to it. Note how the two dependences are in opposite directions.

Nodes which are not loads and have no parents (`c1`, `c2`, `15`) are either invariant in the loop or constants. No latency is shown by the edge since the operand is always available.

This appendix only examines simple loops which contain no other loops. Data dependence in the presence of nested loops is beyond the scope of this chapter. With regard to the differences introduced by nested loops, the 'C6x assembly optimizer capability and features, as well as this appendix, work together to provide you with a conservatively safe solution. That is, the solutions we provide are generally optimal for simple loops, and safe, though sometimes less than optimal, for nested loops. The C compiler, on the other hand, performs very sophisticated dependence analysis on nested loops.

A.2.4 How Dependence Affects Instruction Scheduling

Instruction scheduling is solving the problem of choosing a schedule for a serial stream of instructions which satisfies two competing constraints: it preserves the computational effect of the serial instructions, for example, the code still works, and, it has the best performance.

Instruction scheduling algorithms are built around one central concept: while you do not have to honor the serial instruction order, you do have to honor the order imposed by the instruction dependences.

We can examine this concept at the C statement level. Take the C fragment presented earlier and simply swap the order of the statements:

```
d = e + f;
a = b + c;
```

It is obvious this will generate the same answer. Why? Because the two statements are independent; they do not reference any of the same variables. Consider this fragment ...

```
x = y + z; /* #1 */
z = x + 1; /* #2 */
```

Obviously, if you reorder these statements, you will get a different answer. Consider the variable x . Statement 1 writes a value to x which statement 2

reads; a flow dependence on x . Consider the variable z . Statement 1 reads a value from z while statement 2 writes a value to z ; an anti-dependence on z . Either dependence alone prevents reordering the statements.

It may be somewhat surprising that the forms of dependence, flow or anti- or output, all have the same effect on the statement order. In every case, the dependence constrains those statements to be in that order.

Input dependence is ignored with regard to scheduling; the order you read from memory usually does not matter. Instances where you may be concerned about input dependence include considering the effect on cache behavior, or accessing memory mapped peripherals.

These same ideas transfer directly to assembly language instructions. Instructions which are independent can be reordered, instructions which have one or more dependences cannot be reordered. Further, the latencies associated with the dependences must be honored.

Since dependences force instruction orderings, it follows that fewer dependences mean fewer constraints on instruction orderings. Put another way, fewer dependences mean more degrees of freedom in choosing an instruction schedule. On a chip architecture like the 'C6x, which has many opportunities for parallelism in combination with a deep pipeline, you can never have too much freedom in choosing an instruction schedule.

The details of how instruction scheduling algorithms really work is also beyond the scope of this appendix. But here is the compiler generated schedule for the original C fragment presented earlier:

```
LDW  .D2T2 *+DP(_b),B4 ; |5|
LDW  .D2T2 *+DP(_c),B7 ; |5|
LDW  .D2T2 *+DP(_f),B6 ; |6|
LDW  .D2T2 *+DP(_e),B5 ; |6|
NOP  3
ADD  .L2  B7,B4,B4 ; |5|
STW  .D2T2 B4,*+DP(_a) ; |5|
||  ADD  .L2  B6,B5,B4 ; |6|
      STW  .D2T2 B4,*+DP(_d) ; |6|
```

Note how the instructions from the two C statements are interspersed. The load statements are scheduled early, to better hide the latency of a load. The rest of the instructions are scheduled as soon as the latencies of the instructions they depend on are satisfied. Use the dependence graph from the earlier section as a guide.

A.2.5 Memory Alias Disambiguation Defined

This concept has an analogue in computer programs. When there are two (or more) different ways to reference a memory location, we say there are aliases to that memory location.

Given this linear assembly fragment:

```
I7:    LDW *A4, A2
        ...      ; other instructions
I8:    STWA3, *A6
```

do `A4` and `A6` reference the same memory location or not? If they do, they are memory aliases to that memory location. If they are memory aliases, then these two instructions have an anti-dependence between them; the read must occur before the write. In the instruction schedule, this dependence means those instructions must remain in that order.

On the other hand, if `A4` and `A6` do not reference the same memory location, they are not memory aliases. The instructions are independent. In the instruction schedule, these instructions can safely be placed in any order.

Note that unlike an anti-dependence on registers, there is no way to rewrite these instructions to remove the anti-dependence.

How can you determine whether `*A4` is an alias for `*A6` or not? Given the information we have here, you cannot. Thus, we call this an ambiguous alias. Maybe it is alias, maybe it is not.

Memory alias disambiguation, then, is the process of determining whether any given pair of memory references are aliases to one another. The outcome of that process is one of three states:

State	Means
Not aliases	Instructions are independent.
Are aliases	Instructions have a dependence between them.
Still ambiguous	Tool convention or user information is needed.

If a dependence is found, it imposes an ordering on the instruction schedule.

A.3 Tools Solution

A.3.1 Overview of the Assembly Optimizer Solution

First, the assembly optimizer attempts to automatically disambiguate as many memory references as possible. For all remaining memory references, the default is to presume they may access the same memory location, i.e. they are aliases. While that presumption is safe for all input, it is usually too conservative. So, a command line switch (`-mt`) and a function level directive (`.no_mdep`) can reverse that presumption, i.e. presume that any ambiguous aliases do not access the same memory location. If you have a small number of possible aliases in your code, you can use an additional directive (`.mdep`) to mark those instruction pairs.

A.3.2 Automatic Disambiguation

So, what can the assembly optimizer disambiguate automatically?

When you have memory references that use the same base register but different constant offsets, with no intervening modification of that base register, the assembly optimizer will recognize those references cannot possibly access the same memory location. For example ...

```
I9:   LDW*AR2[4],A3   ; not an alias to I10
      ...           ; no changes to AR2
I10:  LDW*AR2[5],A4   ; not an alias to I9
```

Note the trivial case of 0 offset, e.g. `*AR2`, is included in this analysis.

Any other combination of memory references is an ambiguous alias.

User information about aliases, whether in the form of command line options or directives, is used only after automatic disambiguation fails to yield an answer. You cannot override any automatic disambiguation, whether the answer is “not aliases” or “are aliases”. Implications of this policy are detailed later.

A.3.3 Default Presumption is Pessimistic

The default presumption, any ambiguous alias must be an alias, is a worst case, or pessimistic, assumption. While it is common to have instructions in your linear assembly which possibly access the same memory location, it relatively rare for that possibility to come true. Still, this pessimistic assumption is key to giving you the ability to balance correct handling of memory aliases with good performance.

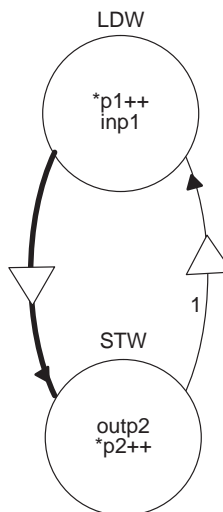
The pessimistic assumption can have a drastic effect on software pipelining. Many linear assembly loops fit this general form:

```

loop:
I11:  LDW *p1++, inp1
      ...           ; compute something into outp2
I12:  STW outp2, *p2++
      ...
; decrement and branch to loop

```

Under the default assumption, `p1` and `p2` may reference the same memory location. This means two more dependence edges are added to the dependence graph: an anti-dependence edge between `I11` and `I12`, and a flow dependence edge between `I12` and `I11`.



When a dependence edge is associated with a memory reference, and not a register operand, you will see a triangle imposed over the edge.

These dependences mean those instructions must remain in that order for every loop iteration. Because these are the first and (nearly the) last instructions in the loop, and they cannot be moved past each other, software pipelining is all but completely disabled.

However, in many cases, `p1` and `p2` point to completely different arrays, and thus never reference the same memory location. So, what is a user to do?

A.3.4 Change the Default Presumption to Optimistic

There are two methods for changing the presumption to the optimistic view that ambiguous memory aliases never access the same location. You can use a command line option:

```
cl6x -mt ...
```

Or, you can use a function level directive:

```
.no_mdep
```

The command line option affects every function in your linear assembly file. The `.no_mdep` directive can only appear within the `.(c)proc/.endproc` block of a linear assembly function, and affects only that function.

If you are certain you have no memory aliases in your code, then switching to the optimistic assumption is all you need to do. This will be a common case. If you ever do have a memory alias in your code, now you know how to handle it: get this appendix out again.

Many users will want to switch to the optimistic assumption, except for a small number aliases they know about in their code. If that is you, the solution is to switch to the optimistic assumption, and then use the `.mdep` directive to mark those few aliases you have.

A.3.5 Using `.mdep` to Mark Aliases

Marking an instance of a memory alias is a two step process. First you attach symbolic names to your memory references in the linear assembly stream:

```
LDW*p1++{ld1}, inp1 ; name memory reference "ld1"
...
```

```
STWoutp2, *p2++{st1} ; name memory reference "st1"
```

The names in the “{}” are assembly symbols like any other. You cannot use the same symbol as a memory reference name and a symbolic register. Then you note the specific memory dependence:

```
.mdep ld1,st1
```

This means whenever `ld1` references some memory location `X`, at some later time in code execution, `st1` may also reference location `X`. This is equivalent to adding an edge between these two instructions in the dependence graph. In terms of the instruction schedule, these instructions must remain in that order. The `ld1` reference must always occur before the `st1` reference.

Recall how the direction of a given dependence is important when considering loops. The direction implied by `.mdep` is from the first named memory reference to the second; in this case, from `ld1` to `st1`. The opposite direction, from `st1` to `ld1`, is not implied.

A.4 Examples of Memory Alias Disambiguation

A.4.1 How .mdep Affects Instruction Scheduling

The following are some complete examples. This example illustrates how an `.mdep` may, or may not, affect the instruction schedule. It also shows how the direction of a dependence, as indicated by the order of the operands to the `.mdep` directive, affects the instruction schedule.

Full understanding of all the examples presumes an understanding of the general concepts of software pipelining. For background information on software pipelining, see Chapter 7.

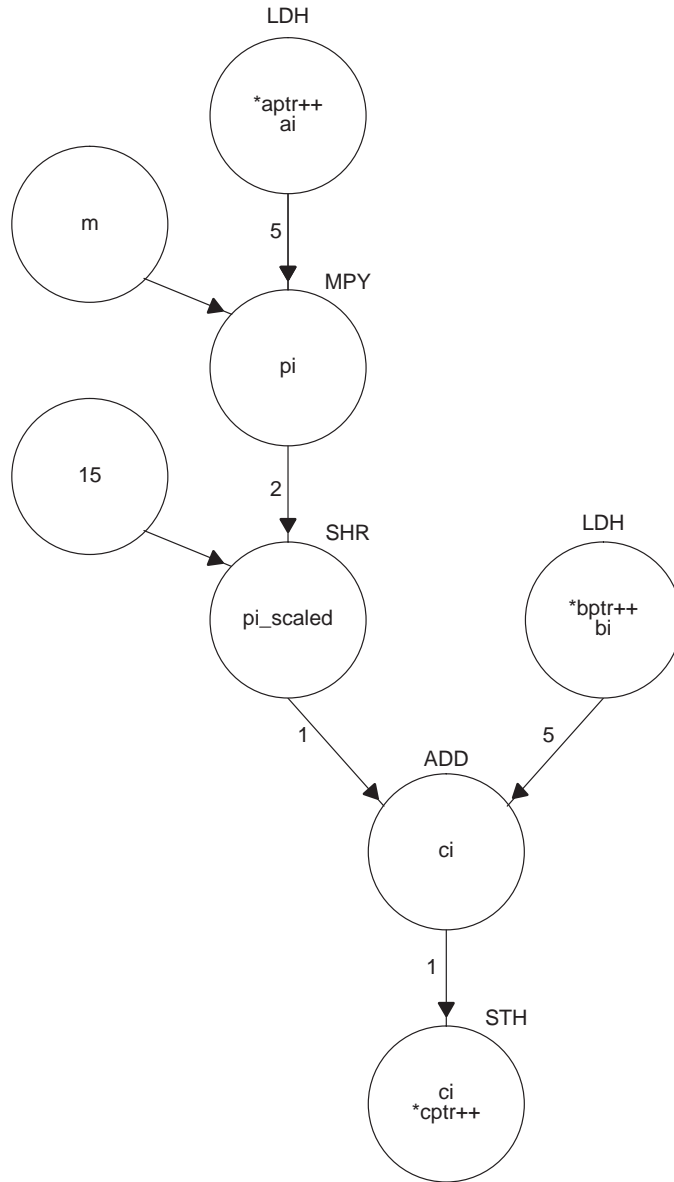
This linear assembly function is adapted from the weighted vector sum example. A typical call to this function could look like:

```
.call wvs(a, b, c, m)
```

Here is the source:

```
wvs:  .cproc aptr, bptr, cptr, m
      .reg  cntr, ai, bi, pi, pi_scaled, ci
      MVK100,cntr
      .no_mdep      ; presume no memory aliases
loop: .trip 100
      LDH*aptr++,ai
      LDH*bptr++,bi
      MPYm,ai,pi
      SHRpi,15,pi_scaled
      ADDpi_scaled,bi,ci
      STHci,*cptr++
[cntr]SUB cntr,1,cntr
[cntr]B  loop
      .endproc
```

Here is the dependence graph (without the decrement and branch instructions):



The assembly optimizer generates a 2-cycle loop for this code, which is optimal for this input.

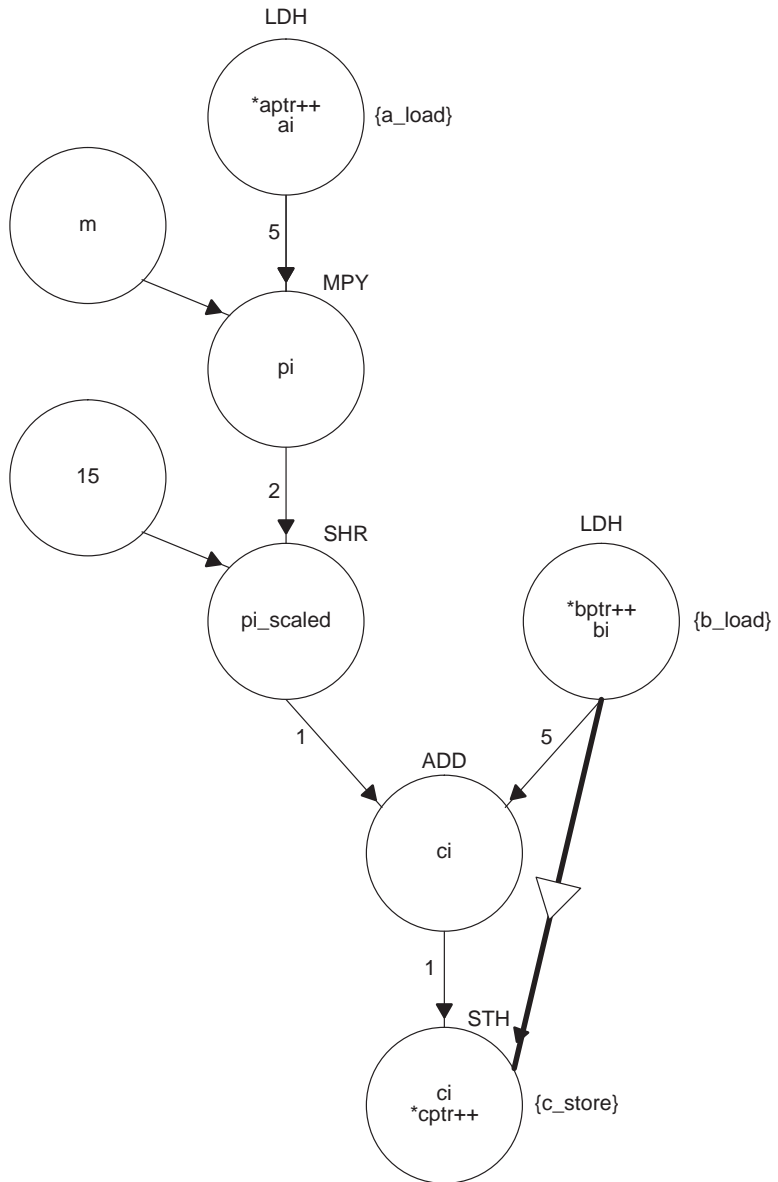
Suppose you know some calls to `wvs` pass the same array as the `b` input array and the `c` output array, just to save some space:

```
.call wvs(a, b, b, m)
```

So, every loop iteration reads an element from the `b` array, and then immediately writes a result back to that same element. The correct way to model that is:

```
wvs: .cproc aptr, bptr, cptr, m
     .reg  cntr, ai, bi, pi, pi_scaled, ci
     MVK100,cntr
     .no_mdep          ; presume no memory aliases
     .mdep b_load,c_store ; except this one
loop: .trip 100
     LDH*aptr++ {a_load},ai
     LDH*bptr++ {b_load},bi
     MPYm,ai,pi
     SHRpi,15,pi_scaled
     ADDpi_scaled,bi,ci
     STHci,*cptr++ {c_store}
[cntr]SUB cntr,1,cntr
[cntr]B  loop
     .endproc
```

Here is the dependence graph:



Note the addition of the anti-dependence memory edge between the `b_load` and the `c_store`. The assembly optimizer still generates a 2-cycle loop for this code; the addition of the `.mddep` makes no difference. Why?

Well, there is already a chain of flow dependences, through registers, from `b_load` to `c_store`, and that chain of dependences imposes an ordering on the instructions in the chain. So, the instruction ordering constraint imposed

by the anti-dependence edge is no different than the constraints already imposed by the flow dependence chain. Therefore, the instruction schedule doesn't change.

Or, you can think about it strictly in terms of the new anti-dependence memory edge. It means every time `b_load` references memory location `X`, at some *later* time in execution, `c_store` may also reference location `X`. This means that each `b_load` must occur before each `c_store`. More importantly, it also means each `c_store` does *not* have to occur before each `b_load`. So, the `b_load` for the next loop iteration can start before the `c_store` from the previous iteration finishes. Here is an illustration of the software pipeline where each iteration is in a separate column, and instructions which can run in parallel are on the same horizontal line:

Iteration n	Iteration n+1
...	
LDH b_load	...
...	LDH b_load
STH c_store	...
	STH c_store

Well, the software pipeline was structured like that before the `.mdep`. So, no change.

While it makes for a contrived example, consider what happens if you call `wvs` like this:

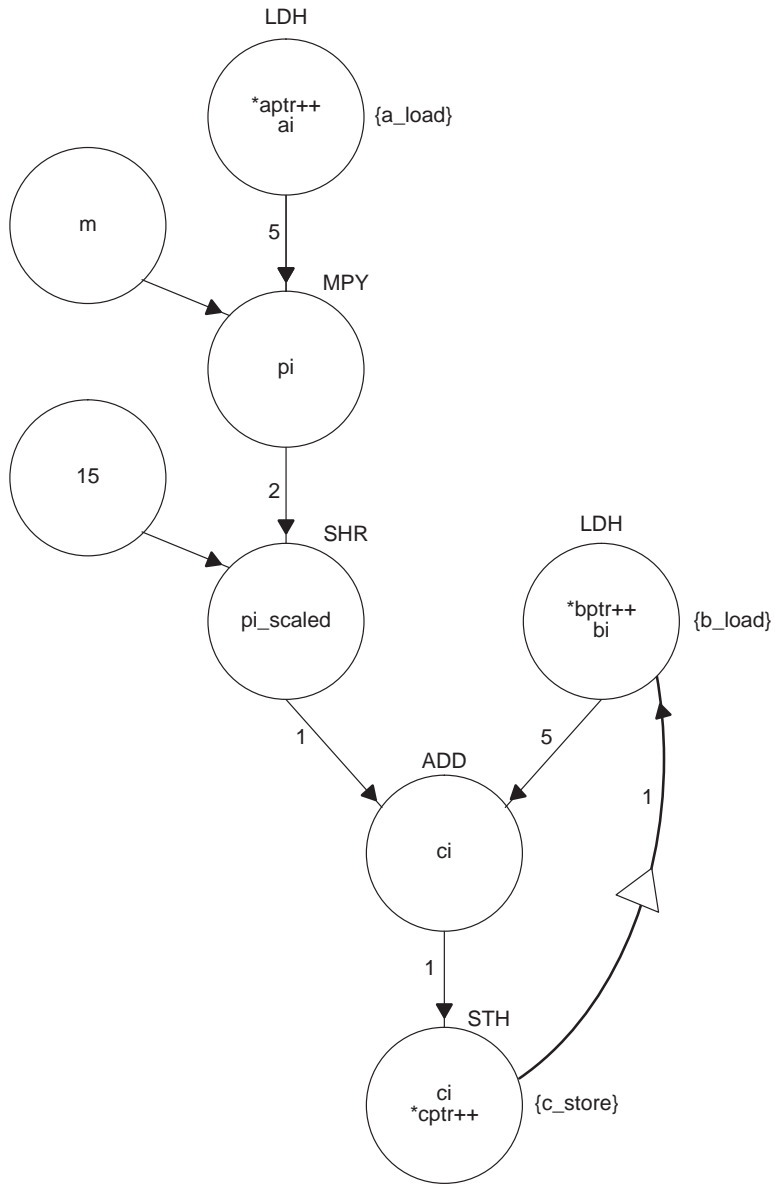
```
ADD    b,2,c          ; c points to b[1]
.call  wvs(a, b, c, m)
```

So, `c_store` writes its result to an array element which `b_load` reads on the next loop iteration. Here is the correct way to model that:

```
.no_mdep                ; presume no memory aliases
.mdep c_store,b_load    ; except this one
<exactly as before>
```

Note the `.mdep` is the same as the previous example, except the operands are reversed.

Here is the dependence graph:



Now, instead of the anti-dependence memory edge, there is a flow dependence memory edge from `c_store` to `b_load`. Note this dependence is carried by the loop. Now the assembly optimizer generates a 7 cycle loop. Why?

Recall the chain of flow dependences, on registers, from the `b_load` to the `c_store`. Now that chain is extended, and carried by the loop, to the `b_load` for the next iteration. Before you can start that `b_load` for the next loop itera-

tion, you have to wait for the `c_store` from the present iteration to finish. Here is how the software pipeline looks:

Iteration n	Iteration n+1
...	
LDH b_load	
...	
STH c_store	...
	LDH b_load
	...
	STH c_store

So, as you can see, the direction, as implied by the operand order, is a very important characteristic of an `.mdep`.

A.4.2 Handling Indexed Addressing Mode

Indexed addressing, e.g. `*+A4[A5]`, typically means you are accessing memory without any clear pattern. How should you handle this case?

Here is an example ...

```

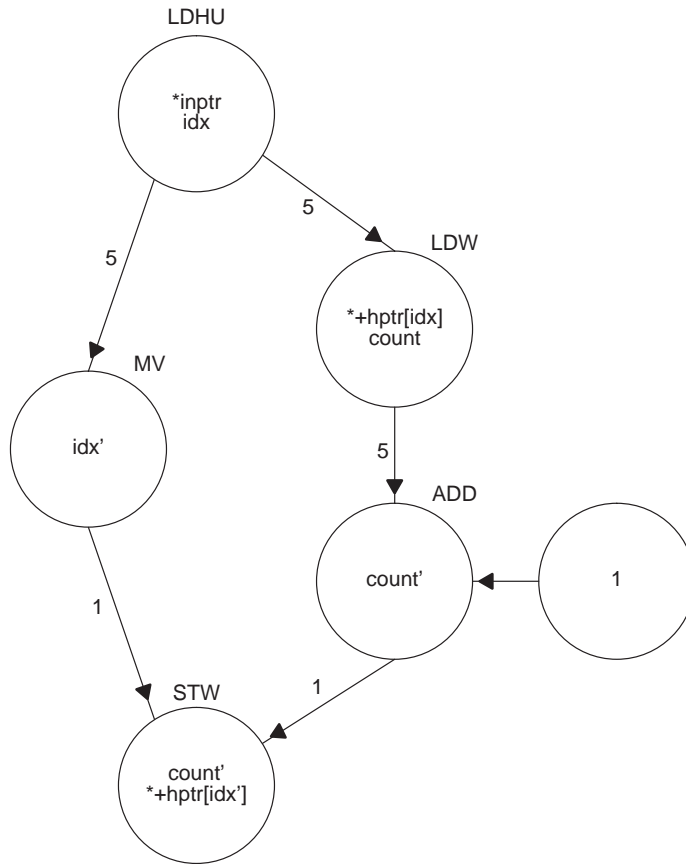
histogram: .cproc inptr, hptr, len
    .reg  idx, count
    .no_mdep          ; no memory aliases

loop: .trip 8
    LDHU  *inptr++,idx
    LDW*+hptr[idx],count
    ADDcount,1,count
    STWcount,*+hptr[idx]

[len] SUBlen,1,len
[len] B  loop
    .endproc

```

Here is the dependence graph:



Note the assembly optimizer splits the lifetime of the `idx` register by adding the `MV` instruction; the new variable is shown as `idx'`. The lifetime of `count` is similarly split at the `ADD` instruction.

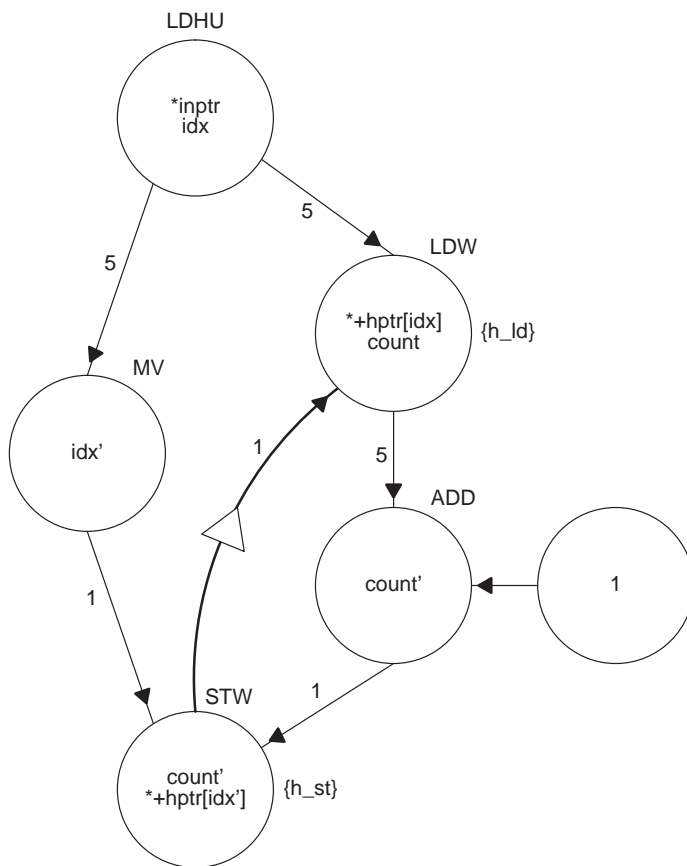
The assembly optimizer generates a 2 cycle loop, but it will not work. Why? This loop is computing, into the array `hptr`, a histogram of all the values in the array `inptr`. What if the value 10 occurs in the `inptr` array two times in a row? In that case, the location `*hptr[10]` is incremented on successive loop iterations. Look at the dependence graph. Do you see a dependence edge from the `STW` to the `LDW`? No? Well, that is the problem. The `LDW` for the next loop iteration has permission to get started without waiting for the `STW` from the previous loop iteration, which it does. To fix this situation we add the `.mdop`:

```

histogram: .cproc inptr, hptr, len
            .reg  idx, count
            .no_mdep          ; no memory aliases
            .mdep h_st, h_ld   ; except this one
loop:      .trip 8
            LDHU  *inptr++,idx
            LDW  *+hptr[idx] {h_ld},count
            ADD  count,1,count
            STW  count,*+hptr[idx] {h_st}
[len]     SUB  len,1,len
[len]     B    loop
            .endproc

```

Here is the dependence graph:



Note the flow dependence memory edge, carried by the loop, from the `STW` to the `LDW`. Now the assembly optimizer generates a 7 cycle loop. Much slower, but it works.

Do you need the `.mdep` in the other direction, from the `h_ld` to the `h_st`? If you simply want the code to run, and you do not care why, then the answer is no. Because of the chain of flow dependences on registers from `h_ld` to `h_st`, this ...

```
.no_mdep           ; no memory aliases
.mdep h_st, h_ld   ; except this one
.mdep h_ld, h_st   ; and this one
```

does not change the instruction schedule. But the dependence actually does exist, so it is advisable to add this `.mdep` because it makes the code self-documenting.

In the face of indexed addressing, you may be tempted to just rely on the default pessimistic assumption. Be careful. In this case, that will hand you a 13-cycle loop. Why? Because under the pessimistic assumption a dependence is recognized from the `STW` (at the bottom of the loop) to the `LDHU` (at the top of the loop). That means the load at the top of iteration `n+1` cannot start until the store at the end of iteration `n` is finished.

A.4.3 Peripherals Access Example

Recall that you cannot override any automatic disambiguation performed by the assembly optimizer. If it can determine that two memory references (must/must not) access the same memory location it (will/will not) recognize a dependence between the associated instructions. This is true despite any command line options or `.mdep` directives which may be in effect. This means there is no way to guarantee the assembly optimizer will use a particular pattern of access to memory. In general code, this is preferable behavior. But it can be an issue when you consider code which accesses memory mapped peripherals. Here is an example:

```
; base of multi-channel buffered serial port 0
MCBSP0_BASE .set 0x018C0000
mcbasp0_dxr: .cproc
    MVKMCBSP0_BASE,B4 ; load base of McBSP0 regs
    MVKH  MCBSP0_BASE,B4
    ...
    STWB5,*+B4(0x10) ; init XCR for transfer
    STWB6,*+B4(0x4) ; transfer word through DXR
    ...
.endproc
```

It is clear that the two STW memory references are not accessing the same memory location; they are using the same base register with different offsets. So, even if you use:

```
.mdep wrt_xcr,wrt_dxr
STWB5,*+B4(0x10) {wrt_xcr}
STWB6,*+B4(0x4) {wrt_dxr}
```

the assembly optimizer may still reorder the writes. In general code, this is fine, and often an improvement. But when accessing peripherals like a serial port, or whenever writing to a memory location can trigger a side effect, reordering the memory references is wrong.

Presently, there are two ways to solve this problem. You can write the code in C, being careful to use the keyword “volatile” for any memory reference which has a side effect. Or, you can bypass the assembly optimizer by writing these routines in hand-coded assembly.

A.5 C Compiler and Alias Disambiguation

The C compiler provides several methods, both command line options and in the source, for addressing the problem of memory alias disambiguation. Having read this appendix, you should now have a much better understanding of the issue. This section will briefly review each of these methods. For all the details, consult either this book or the *TMS320C6x Optimizing C Compiler User's Guide*.

The compiler does a much better job of alias disambiguation than the assembly optimizer. C source provides much more information to work with. So, the default presumption on aliases which cannot be disambiguated is the pessimistic one: they are aliases.

Still, there are a few very esoteric cases of memory aliases which the compiler presumes do not occur. If your code violates those presumptions use:

```
cl6x -ma
```

On rare occasions, you may need it.

The command line option:

```
cl6x -mt
```

means something different to the compiler than it does to the assembly optimizer. As presented already, this option reverses the assembly optimizer's pessimistic assumption that memory references it cannot disambiguate must be aliases. To the compiler, this same option means several specific instances of memory aliases do not occur in your C code.

The command line options:

```
cl6x -pm -o3
```

have several effects, of which improved alias disambiguation only one. These options work together to provide program level optimization. The `-pm` option combines all of your source files into one intermediate file, and `-o3` carries out the program level optimization on that intermediate file. Seeing all the functions at once yields optimization opportunities which generally do not occur otherwise. If the compiler can see all the calls to this function:

```
void foo(int *p1, int *p2)
```

it can easily determine that the same array is never passed in for `p1` and `p2`, and therefore `p1` and `p2` references are not aliases.

Aggressive, but correct, use of the `const` keyword helps the alias disambiguation problem. The `const` keyword tells the compiler that data object will not be modified. Not even by an alias. So, any `const` qualified memory read cannot possibly be an alias to a memory write. If an alias does modify a `const` object, that is a user bug.

A.6 Memory Alias Disambiguation versus Memory Bank Conflict Detection

If a 'C6x execute packet (a set of instructions which execute in parallel) includes two references to memory, and both of those references are to the same memory bank, because each bank is single-ported memory, the pipeline stalls for one cycle while the second memory word is accessed. This is called a bank conflict, and it is obviously worth avoiding. The assembly optimizer provides a directive called `.mptx` for the purpose of solving this problem. See the *TMS320C6x Optimizing C Compiler User's Guide* for all the details.

It is easy to confuse the topic of memory alias disambiguation with memory bank conflict detection. The terms sound similar. And they are both concerned with how memory references affect the instruction schedule. But there are some striking differences ...

	Alias Disambiguation	Bank Conflict Detection
The problem is ...	Whether two memory references access exactly the same location	Whether two memory references access the same memory bank
The answer affects ...	The ordering constraints imposed on the instruction schedule	Whether to schedule a pair of memory references in parallel
Get it wrong and ...	Your code does not work	The execute packet takes 1 cycle longer
Occurrences of ...	Memory aliases are relatively rare	Potential memory bank conflicts are common

You have to solve the problem of memory alias disambiguation before you can consider the problem of memory bank conflict detection. One of the presumptions of memory bank conflict detection is that the two memory references can be scheduled in parallel. That is true only if you have already determined the instructions are independent; they are not aliases to one another.

In your linear assembly, it is best to simply keep these problems, and their solutions, entirely separate. Enter your `.mdop` directives without any regard to your `.mptx` directives, and vice versa.

A.7 Summary

- Dependence is a relationship between two instructions which read or write the same machine resource.
- Dependence graphs represent the dependence between instructions. Nodes (circles) are instructions. Edges (arrows) are dependences. Data often flows along edges, but not always.
- In loops, nodes represent every instance of that instruction in every loop iteration, and dependence direction is important.
- Dependences force an ordering on the instruction schedule.
- Generally, the fewer the dependences, the better the schedule.
- Multiple references to the same memory location are called aliases.
- Aliases imply a dependence between the associated instructions.
- Memory alias disambiguation is the process of determining whether a pair of references are aliases, for example, whether a dependence is recognized between the instructions.
- The assembly optimizer automatically disambiguates some references, then uses command line options and directives to disambiguate the remaining references.
- The default presumption for remaining aliases is pessimistic; they are aliases.
- The assembly optimizer command line option:

```
cl6x -mt ...
```

reverses the default presumption to optimistic; they are not aliases. It applies to all functions in the file.

- The function level directive ...

```
.no_mdep
```

also changes the presumption to optimistic, but applies only to the function which contains it.

- To mark a specific memory dependence, first annotate memory references ...

```
LDW*p1++{ld1}, inp1 ; name memory reference "ld1"
...
```

```
STWoutp2, *p2++{st1} ; name memory reference "st1"
```

Then note the specific dependence:

```
.mdep ld1,st1
```

- You cannot force the assembly optimizer to recognize a dependence between instructions, which can be an issue when accessing memory mapped peripherals.
- The C compiler offers the user several methods for influencing the handling of memory aliases
- Do not confuse memory alias disambiguation with memory bank conflict detection. Solve the problems separately.



Sy

- [] in assembly code 6-3
- @ symbol in assembly output 2-15
- || (parallel bars) in assembly code 6-2
- _ (underscore) in intrinsics 4-13

A

- _add2 intrinsic 4-18
 - tutorial 2-19
 - aliasing 4-8
 - allocating resources
 - conflicts 7-66
 - dot product 7-24
 - if-then-else 7-91, 7-98
 - IIR filter 7-83
 - in writing parallel code 7-11
 - live-too-long resolution 7-107
 - weighted vector sum 7-63
 - AND instruction, mask for 7-75
 - arrays, controlling alignment 7-121
 - assembler directives 6-4
 - assembly code
 - comments in 6-9
 - conditions in 6-3
 - directives in 6-4
 - dot product, fixed-point
 - nonparallel* 7-15
 - parallel* 7-16
 - final
 - dot product, fixed-point* 7-27, 7-47, 7-53, 7-56
 - dot product, floating-point* 7-49, 7-54, 7-57
 - FIR filter* 7-121, 7-130, 7-134 to 7-137, 7-148 to 7-151
 - FIR filter with redundant load elimination* 7-117
 - if-then-else* 7-92, 7-93, 7-100
 - IIR filter* 7-86
 - live-too-long, with move instructions* 7-109
 - weighted vector sum* 7-76
 - functional units in 6-6
 - instructions in 6-4
 - labels in 6-2
 - linear
 - dot product, fixed-point* 7-10, 7-21, 7-25, 7-31, 7-40
 - dot product, floating-point* 7-22, 7-26, 7-32, 7-41
 - FIR filter* 7-113, 7-115, 7-124, 7-126
 - FIR filter, outer loop* 7-139
 - FIR filter, outer loop conditionally executed with inner loop* 7-142, 7-144
 - FIR filter, unrolled* 7-138
 - if-then-else* 7-88, 7-91, 7-96, 7-99
 - IIR filter* 7-79, 7-83
 - live-too-long* 7-103, 7-108
 - weighted vector sum* 7-59, 7-61, 7-63
 - mnemonics in 6-4
 - operands in 6-8
 - optimizing (phase 3 of flow), description 7-2
 - parallel bars in 6-2
 - structure of 6-1 to 6-11
 - writing parallel code 7-4, 7-9
 - assembly optimizer
 - for dot product 7-42
 - tutorial 2-26, 2-29
 - using to create optimized loops 7-40
 - assistance from TI vi
- ## B
- big-endian mode
 - and MPY operation 7-22
 - runtime support (rts6201e.lib) 2-6

biquad filter
 inner loop kernel
 assembly from C with intrinsics 2-24
 linear assembly 2-30
 original assembly code 2-17
 linear assembly 2-28
 original C code 2-4
 with word instructions and intrinsics 2-21

branch target, for software-pipelined dot product 7-42, 7-44

branching to create if-then-else 7-87

C

C code
 analyzing performance of 4-2
 basic vector sum 4-7
 dot product 4-20
 fixed-point 7-9, 7-20
 floating-point 7-21
 FIR filter 4-20, 4-38, 7-111, 7-123
 inner loop completely unrolled 4-39
 optimized form 4-21
 unrolled 7-132, 7-137, 7-140
 with redundant load elimination 7-112
 if-then-else 7-87, 7-95
 IIR filter 7-78
 live-too-long 7-102
 refining (phase 2 of flow), in flow diagram 1-3
 saturated add 4-13
 trip counters 4-33
 vector sum
 with const keywords 4-9
 with const keywords, _nassert, word reads 4-18, 4-19
 with const keywords, _nassert, word reads, unrolled 4-37
 with three memory operations 4-36
 word-aligned 4-36
 weighted vector sum 7-59
 unrolled version 7-60
 writing 4-2

C_OPTIONS environment variable 2-6

'C6x mnemonics 6-5

char data type 4-2

child node 7-11

cl6x command 2-5, 4-4

clock () function 4-2

code development flow diagram 1-3
 phase 1: develop C code 1-3, 2-15 to 2-17
 phase 2: refine C code 1-3, 2-18 to 2-25
 phase 3: write linear assembly 1-3, 2-26 to 2-31

code development steps 3-2

code documentation 6-9

comments in assembly code 6-9

compiler options
 -ms 4-34
 -o3 4-34
 -pm 4-34

conditional execution of outer loop with inner loop 7-139

conditional instructions to execute if-then-else 7-88

conditional SUB instruction 7-30

conditions in assembly code 6-3

const keyword 4-6, 4-8
 in vector sum 4-18

constant operands 6-8

.cproc directive 2-26

CPU elements 1-2

cycle count
 for biquad filter 2-30
 for functions in demo1.c 2-14
 for multiply accumulate 2-14
 for vector multiply 2-23
 formula for calculating 2-14

D

data types 4-2

demo1.c example code 2-3

demo2.c example code 2-22

demo3.c example code 2-29

dependency graph
 dot product, fixed-point 7-12
 dot product, fixed-point
 parallel execution 7-16
 with LDW 7-23, 7-25, 7-31
 dot product, floating-point, with LDW 7-24, 7-26, 7-32

drawing 7-11
 steps in 7-12

FIR filter
 with arrays aligned on same loop
 cycle 7-122
 with no memory hits 7-125
 with redundant load elimination 7-114

- if-then-else 7-89, 7-97
 - IIR filter 7-80, 7-82
 - live-too-long code 7-104, 7-107
 - showing resource conflict 7-66
 - resolved 7-69
 - vector sum 4-7
 - weighted 7-62, 7-66, 7-69, 7-71
 - with *const* keywords 4-9
 - weighted vector sum 7-69
- destination operand 6-8
- dot product
- C code 7-9
 - fixed-point 7-9
 - translated to linear assembly, fixed-point 7-10
 - with intrinsics 4-20
- dependency graph of basic 7-12
- fixed-point
- assembly code with LDW before software pipelining 7-27
 - assembly code with no extraneous loads 7-47
 - assembly code with no prolog or epilog 7-53
 - assembly code with smallest code size 7-56
 - assembly code, fully pipelined 7-43
 - assembly code, nonparallel 7-15
 - C code with loop unrolling 7-20
 - dependency graph of parallel assembly code 7-16
 - dependency graph with LDW 7-25
 - fully pipelined 7-42
 - linear assembly for full code 7-40
 - linear assembly for inner loop with conditional SUB instruction 7-31
 - linear assembly for inner loop with LDW 7-21
 - linear assembly for inner loop with LDW and allocated resources 7-25
 - nonparallel assembly code 7-15
 - parallel assembly code 7-16
- floating-point
- assembly code with LDW before software pipelining 7-28
 - assembly code with no extraneous loads 7-49
 - assembly code with no prolog or epilog 7-54
 - assembly code with smallest code size 7-57
 - assembly code, fully pipelined 7-44
 - C code with loop unrolling 7-21
 - linear assembly for inner loop with conditional SUB instruction 7-32
 - fully pipelined 7-44
 - linear assembly for full code 7-41
 - linear assembly for inner loop with LDW 7-22
 - linear assembly for inner loop with LDW and allocated resources 7-26
 - word accesses in 4-19
- double data type 4-2
- ## E
- .endproc directive 2-26
 - epilog 4-32
 - execute packet 2-14, 2-16, 7-41
 - execution cycles, reducing number of 7-9
 - extraneous instructions, removing 7-46
 - SUB instruction 7-56
- ## F
- feedback, from compiler or assembly optimizer 3-4
 - File menu (debugger) 2-12
 - FIR filter
 - C code 4-20, 7-111
 - optimized form 4-21
 - unrolled 7-137, 7-140
 - with inner loop unrolled 7-132
 - with redundant load elimination 7-112
 - final assembly 7-148
 - for inner loop 7-121
 - with redundant load elimination 7-117
 - with redundant load elimination, no memory hits 7-130
 - with redundant load elimination, no memory hits, outer loop software-pipelined 7-134
 - linear assembly
 - for inner loop 7-113
 - for outer loop 7-139
 - for unrolled inner loop 7-124
 - for unrolled inner loop with .mptr directive 7-126
 - with inner loop unrolled 7-138
 - with outer loop conditionally executed with inner loop 7-142, 7-144
 - software pipelining the outer loop 7-132
 - using word access in 4-20

- with inner loop unrolled 7-123
- fixed-point, dot product
 - linear assembly for inner loop with LDW 7-21
 - linear assembly for inner loop with LDW and allocated resources 7-25
- float data type 4-2
- floating-point, dot product
 - dependency graph with LDW 7-26
 - linear assembly for inner loop with LDDW 7-22
 - linear assembly for inner loop with LDDW with allocated resources 7-26
- flow diagram, code development 1-3
- functional units
 - description 6-7
 - in assembly code 6-7
 - reassigning for parallel execution 7-15, 7-17
- functions
 - clock () 4-2
 - printf () 4-2

G

- g option 2-5

I

- if-then-else
 - branching versus conditional instructions 7-87
 - C code 7-87, 7-95
 - final assembly 7-92, 7-93, 7-100
 - linear assembly 7-88, 7-91, 7-96, 7-99
- IIR filter, C code 7-78
- iir1.asm, inner loop kernel 2-17
- iir1.c example code 2-4
- in-flight value 8-3
- information elements in tutorial 2-2
- inserting moves 7-106
- instructions, placement in assembly code 6-4
- int data type 4-2
- interrupt subroutines 8-11 to 8-14
 - hand-coded assembly allowing nested interrupts 8-13
 - nested interrupts 8-12
 - with hand-coded assembly 8-12
 - with the C compiler 8-11
- interrupts
 - overview 8-2

- single assignment versus multiple assignment 8-3 to 8-4
- intrinsics
 - _add2 () 4-18
 - _mpy () 4-19
 - _mpyh () 4-19
 - _mpyhl () 4-18
 - _mpylh () 4-18
 - _nassert 4-34
 - described 2-19, 4-13
 - in saturated add 4-13
 - summary table 4-14 to 4-16
- iteration interval, defined 7-33

K

- k compiler option 2-5, 4-4
- kernel
 - loop 2-15, 4-9, 4-32
 - of iir1.asm code 2-17
 - of iir2.asm code 2-24
 - of iir3.asm code 2-30
 - of mac1.asm code 2-15
 - of vec_mpy1.asm code 2-16
 - of vec_mpy2.asm code 2-23

L

- l linker option 2-6
- labels in assembly code 6-2
- linear, optimizing (phase 3 of flow), in flow diagram 1-3
- linear assembly 2-26
 - code
 - dot product, fixed-point* 7-10
 - dot product, fixed-point* 7-15, 7-21, 7-25, 7-31, 7-40
 - dot product, floating-point* 7-22, 7-26, 7-32, 7-41
 - FIR filter* 7-113, 7-115, 7-124, 7-126
 - FIR filter with outer loop conditionally executed with inner loop* 7-142, 7-144
 - FIR filter, outer loop* 7-139
 - FIR filter, unrolled* 7-138
 - if-then-else* 7-91, 7-99
 - live-too-long* 7-108
 - weighted vector sum* 7-63
- resource allocation
 - conflicts* 7-66

- dot product* 7-24
 - if-then-else* 7-91, 7-98
 - IIR filter* 7-83
 - in writing parallel code* 7-11
 - live-too-long resolution* 7-107
 - weighted vector sum* 7-63
 - linker command file 2-6
 - little-endian mode, runtime support (rts6201.lib) 2-6
 - little-endian mode, and MPY operation 7-22
 - live-too-long
 - code 7-68
 - C code* 7-102
 - inserting move (MV) instructions* 7-106
 - unrolling the loop* 7-106
 - issues 7-102
 - and software pipelining* 4-43
 - created by split-join paths* 7-105
 - load
 - doubleword (LDDW) instruction 7-20
 - word (LDW) instruction 7-20
 - Load Program File dialog box (debugger) 2-12
 - load6x 2-11
 - long data type 4-2
 - loop
 - carry path, described 7-78
 - counter, handling odd-numbered 4-19
 - iterations 4-33
 - kernel 2-15
 - unrolling
 - dot product* 7-20
 - for simple loop structure* 4-38
 - if-then-else code* 7-95
 - in FIR filter* 7-123, 7-126, 7-132, 7-137, 7-139
 - in live-too-long solution* 7-106
 - in vector sum* 4-36
- M**
- mac1.asm kernel, inner loop 2-15
 - mac1.c example code 2-3
 - memory bank scheme, interleaved 7-119 to 7-121
 - memory dependency. *See* dependency
 - mg compiler option 2-5
 - minimum iteration interval, determining 7-35
 - for FIR code 7-115, 7-129, 7-147
 - for if-then-else code 7-90, 7-98
 - for IIR code 7-81
 - for live-too-long code 7-105
 - for weighted vector sum 7-60, 7-61
 - mnemonic (instruction) 6-4
 - modulo iteration interval table
 - dot product, fixed-point
 - after software pipelining* 7-36
 - before software pipelining* 7-33
 - dot product, floating-point
 - after software pipelining* 7-37
 - before software pipelining* 7-34
 - IIR filter, 4-cycle loop 7-84
 - weighted vector sum
 - 2-cycle loop* 7-65, 7-70, 7-73
 - with SHR instructions* 7-67
 - modulo-scheduling technique, multicycle loops 7-59
 - move (MV) instruction 7-106
 - _mpy intrinsic 4-19
 - tutorial 2-19
 - _mpyh () intrinsic 4-19
 - _mpyhl intrinsic 4-18
 - _mpyhh intrinsic 4-18
 - tutorial 2-19
 - multicycle instruction, staggered accumulation 7-38
 - multiple assignment, code example 8-3
 - multiply accumulate function
 - inner loop kernel of original assembly code 2-15
 - original C code 2-3
 - mw compiler option 3-3
- N**
- _nassert intrinsic 4-16, 4-18, 4-34
 - node 7-11
- O**
- o compiler option 2-5, 4-4, 4-32, 4-34
 - o linker option 2-6
 - operands
 - placement in assembly code 6-8
 - types of 6-8
 - optimization checklist 3-1 to 3-8
 - optimizing assembly code, introduction 7-2

optional tasks in tutorial 2-2
 outer loop conditionally executed with inner loop 7-137
 OUTLOOP 7-116, 7-129

P

parallel bars, in assembly code 6-2
 parent instruction 7-11
 parent node 7-11
 path in dependency graph 7-11
 performance analysis
 of C code 4-2
 of dot product examples 7-19, 7-29, 7-58
 of FIR filter code 7-129, 7-136, 7-150
 of if-then-else code 7-94, 7-101
 pipeline in 'C6x 1-2
 -pm compiler option 4-4, 4-6, 4-11, 4-34
 pointer operands 6-8
 preparation for tutorial 2-1
 primary tasks in tutorial 2-2
 priming the loop, described 7-52
 priming the pipeline 4-33
 printf () function 4-2
 processor mnemonics 6-5
 Profile
 Marking dialog box 2-12
 menu (debugger) 2-12
 Run dialog box 2-13
 profiling 2-8 to 2-14
 program-level optimization 4-6
 prolog 4-32, 7-52, 7-54
 pseudo-code, for single-cycle accumulator with ADDSP 7-38

R

redundant
 load elimination 7-111
 loops 4-34
 .reg directive 2-26, 7-21, 7-22
 register
 allocation 7-128
 operands 6-8
 related documentation iv

resource
 conflicts
 described 7-66
 live-too-long issues 7-68, 7-102
 table
 FIR filter code 7-115, 7-129, 7-147
 if-then-else code 7-90, 7-98
 IIR filter code 7-81
 live-too-long code 7-105
 rts6201.lib file 2-6
 rts6201e.lib file 2-6

S

.sa extension 2-26
 _sadd intrinsic 4-13, 4-16
 scheduling table. *See* modulo iteration interval table
 shell program (cl6x) 2-5, 4-4
 short
 arrays 4-18
 data type 4-2, 4-18
 single assignment, code example 8-4
 software pipeline 4-32, 4-37
 accumulation, staggered results due to 3-cycle delay 7-39
 described 7-30
 when not used 4-41
 software-pipelined schedule, creating 7-35
 source operands 6-8
 split-join path 7-102, 7-103, 7-105
 stand-alone simulator (load6x) 4-2
 SunOS shell initialization 2-7
 symbolic names, for data and pointers 7-21, 7-22

T

techniques
 for priming the loop 7-52
 for refining C code 4-13
 for removing extra instructions 7-46, 7-56
 using intrinsics 4-13
 word access for short data 4-18
 TMS320C6x pipeline 1-2
 translating C code to 'C6x instructions
 dot product
 fixed-point, unrolled 7-21
 floating-point, unrolled 7-22

- IIR filter 7-79
 - with reduced loop carry path* 7-83
- weighted vector sum 7-59
 - unrolled inner loop* 7-61
- translating C code to linear assembly, dot product, fixed-point 7-10
- trip count 2-26, 4-33
 - communicating information to the compiler 4-34
 - determining the minimum 4-33
- trip counter, defined 4-33
- .trip directive 2-26

V

- vec_mpy1.asm kernel, inner loop 2-16
- vec_mpy1.c example code 2-4
- vector multiply function
 - C with word instructions and intrinsics 2-19
 - inner loop kernel
 - of assembly from C with intrinsics* 2-23
 - of original assembly code* 2-16
 - original C code 2-4
 - tutorial C code example (vec_mpy1.c) 2-4
- vector sum function
 - See also* weighted vector sum
 - C code 4-7
 - with const keyword* 4-9
 - with const keywords, _nassert, word reads* 4-18
 - with const keywords, _nassert, word reads, and loop unrolling* 4-37

- with const keywords, _nassert, and word reads (generic)* 4-19
- with three memory operations* 4-36
- word-aligned* 4-36
- compiler output (original assembly code) 4-10
- dependency graph 4-7, 4-9
- handling odd-numbered loop counter with 4-19
- handling short-aligned data with 4-19
- rewriting to use word accesses 4-18
- VelociTI 1-2
- very long instruction word (VLIW) 1-2

W

- weighted vector sum
 - C code 7-59
 - unrolled version* 7-60
 - final assembly 7-76
 - linear assembly 7-74
 - for inner loop* 7-59
 - with resources allocated* 7-63
 - translating C code to assembly instructions 7-61
- word access
 - in dot product 4-19 to 4-20
 - in FIR filter 4-20
 - using for short data 4-18 to 4-31

Z

- z compiler option 2-6