TMS320C6000
Programmer's Guide

Literature Number: SPRU198D
March 2000

TEXAS INSTRUMENTS

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.

Preface

Read This First

About This Manual

This manual is a reference for programming TMS320C6000 digital signal processor (DSP) devices.

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

This book is organized in five major parts:

❑ **Part I: Introduction** includes a brief description of the ’C6000 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: ’C64x Programming Techniques** describes programming considerations for the ’C64x.

❑ **Part IV: Appendix** provides a summary of feedback solutions and memory alias disambiguation.

Related Documentation From Texas Instruments

The following books describe the TMS320C6000 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 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.

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.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/6701 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.

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.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This chapter introduces some features of the ’C6000 microprocessor and discusses the basic process for creating code and understanding feedback. Any reference to ’C6000 pertains to the ’C62x (fixed-point), ’C64x (fixed-point), and the ’C67x (floating-point) devices. All techniques are applicable to each device, even though most of the examples shown are fixed-point specific.

Topic **Page**

1.1 TMS320C6000 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 ’C64x is a fixed-point DSP with the same features. It is the third DSP to use the VelociTI™ architecture.

The ’C6000 DSPs are based on the ’C6000 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 (’C62x and ’C67x)
❑ Sixty-four 32-bit registers (’C64x)
❑ Control registers
❑ Control logic
❑ Test, emulation, and interrupt logic

1.2 TMS320C6000 Pipeline

The ’C6000 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.

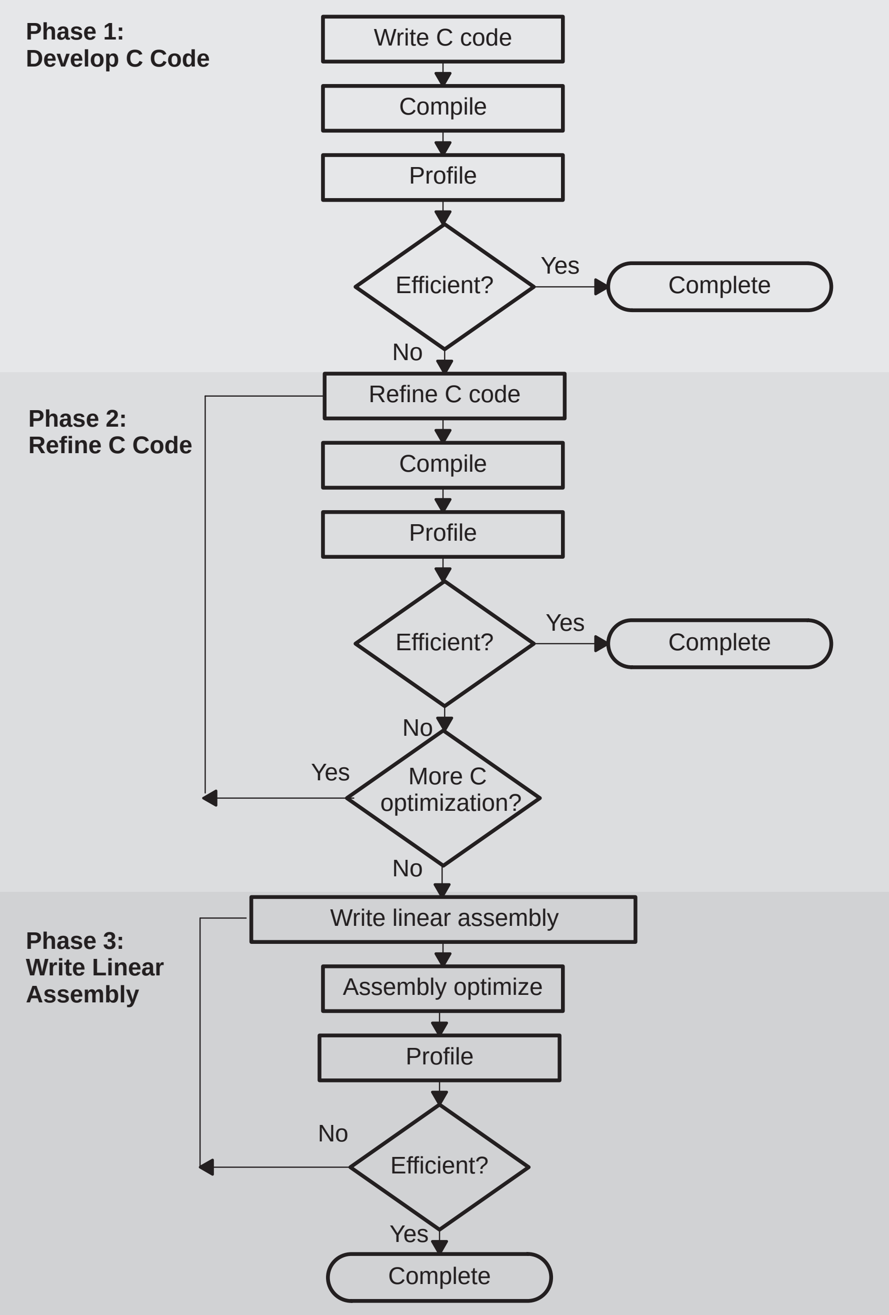
1.3 Code Development Flow to Increase Performance

Traditional development flows in the DSP industry have involved validating a C model for correctness on a host PC or Unix workstation and then painstakingly porting that C code to hand coded DSP assembly language. This is both time consuming and error prone, not to mention the difficulties that can arise from maintaining the code over several projects.

The recommended code development flow involves utilizing the ’C6000 code generation tools to aid in your optimization rather than forcing you to code by hand in assembly. The advantages are obvious. Let the compiler do all the laborious work of instruction selection, parallelizing, pipelining, and register allocation, and you focus on getting the product to market quickly. Because of these features, maintaining the code becomes easy, as everything resides in a C framework that is simple to maintain, support and upgrade.

The recommended code development flow for the ’C6000 involves the phases described below. The tutorial section of the Programmer’s Guide focuses on phases 1 – 3, and will show you when to go to the tuning stage of phase 3. What you learn is the importance of giving the compiler enough information to fully maximize its potential. What’s even better is that this compiler gives you direct feedback on all your high MIPS areas (loops). Based on this feedback, there are some very simple steps you can take to pass more, or better, information to the compiler allowing you to quickly begin maximizing compiler peformance.

You can achieve the best performance from your ’C6000 code if you follow this code development 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 ’C6000. Use the ’C6000 profiling tools that are described in the *Code Composer Studio User’s Guide* to identify any inefficient areas that you might have in your C code. To improve the performance of your code, proceed to phase 2.
2	Use the intrinsics, shell options, and techniques that are described in this book to improve your C code. Use the ’C6000 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.

Because most of the millions of instructions per second (MIPS) in DSP applications occur in tight loops, it is important for the ’C6000 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 ’C6000 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 feedback list that can be used to optimize loops with references to more detailed documentation.

Table 1–1 describes the recommended code development flow for developing code which achieves the highest performance on loops.

Table 1–1. Code Development Steps

Step	Description
1	Compile and profile native C/C++ code ❑ Validates original C/C++ code ❑ Determines which loops are most important in terms of MIPS requirements
2	Add const declarations, loop, memory bank, and data alignment information. ❑ Reduces potential pointer aliasing problems ❑ Allows loops with indeterminate iteration counts to execute epilogs ❑ Uses pragmas to pass count information to the compiler ❑ Uses memory bank pragmas and _nassert intrinsic to pass memory bank and alignment information to the compiler.
3	Optimize C code using other ’C6000 intrinsics and other methods ❑ Facilitates use of certain ’C6000 instructions not easily represented in C ❑ Optimizes data flow bandwidth (uses word access for short (’C62x, ’C64x, and ’C67x) data, and double word access for word (’C64x, and ’C67x) data).
4a	Write linear assembly ❑ Allows control in determining exact ’C6000 instructions to be used ❑ Provides flexibility of hand-coded assembly without worry of pipelining, parallelism, or register allocation ❑ Can pass memory bank information to the tools ❑ Uses .trip directive to convey loop count information
4b	Add partitioning information to the linear assembly ❑ Can improve partitioning of loops when necessary ❑ Can avoid bottlenecks of certain hardware resources
	Code size considerations ❑ Can trade small performance degradation for smaller code on loops ❑ 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 ’C6000 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/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/C++ can offer, works within the framework of C/C++, and is much like programming in higher level C. For more information on the assembly optimizer, see the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* and Chapter 6, *Optimizing Assembly Code via Linear Assembly*, in this book.

In order to aid the development process, some feedback is enabled by default in the code generation tools. Additional feedback is generated with the -mw option. Example 1–1 shows output from the compiler and/or assembly optimizer of a particular loop.

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

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     2        3
;*      .S units                     4        4
;*      .D units                     1        0
;*      .M units                     0        0
;*      .X cross paths               1        3
;*      .T address paths             1        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*----------------------------------------------------------------------------*
```

This feedback is important in determining which optimizations might be useful for further improved performance. The following section, Understanding Feedback, 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.

1.4 Understanding Feedback

The compiler provides some feedback by default. Additional feedback is generated with the -mw option. The feedback is located in the .asm file that the compiler generates. In order to view the feedback, you must also enable -k which retains the .asm output from the compiler. By understanding feedback, you can quickly tune your C code to obtain the highest possible performance.

The feedback in Example 1–1 is for an innermost loop. On the ’C6000, C code loop performance is greatly affected by how well the compiler can software pipeline. The feedback is geared for explaining exactly what all the issues with pipelining the loop were and what the results obtained were. Understanding feedback will focus on all the components in the software pipelining feedback window.

The compiler goes through three basic stages when compiling a loop. Here we will focus on the comprehension of these stages and the feedback produced by them. This, combined with the Feedback Solutions in Appendix A will send you well on your way to fully optimizing your code with the ’C6000 compiler. The three stages are:

1) Qualify the loop for software pipelining

2) Collect loop resource and dependency graph information

3) Software pipeline the loop

1.4.1 Stage 1: Qualify the Loop for Software Pipelining

The result of this stage will show up as the first three or four lines in the feedback window as long as the compiler qualifies the loop for pipelining:

Example 1–2. Stage 1 Feedback

```
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
```

The compiler tries to identify what the loop counter (named trip counter because of the number of trips through a loop) is and any information about the loop counter such as minimum value (known minimum trip count), and whether it is a multiple of something (has a known maximum trip count factor).

If factor information is known about a loop counter, the compiler can be more aggressive with performing packed data processing and loop unrolling optimizations. For example, if the exact value of a loop counter is not known but it is known that the value is a multiple of some number, the compiler may be able to unroll the loop to improve performance.

There are several conditions that must be met before software pipelining is allowed, or legal, from the compiler’s point of view. These conditions are:

- ❑ It cannot have too many instructions in the loop. Loops that are too big, typically require more registers than are available and require a longer compilation time.
- ❑ It cannot call another function from within the loop unless the called function is inlined. Any break in control flow makes it impossible to software pipeline as multiple iterations are executing in parallel.

If any of the conditions for software pipelining are not met, qualification of the pipeline will halt and a disqualification messages will appear. For more information about what disqualifies a loop from being software-pipelined, see section 3.4.3.6, on page 3-50.

1.4.2 Stage 2: Collect loop resource and dependency graph information

The second stage of software pipelining a loop is collecting loop resource and dependency graph information. The results of stage 2 will be displayed in the feedback window as follows:

Example 1–3. Stage 2 Feedback

```
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     2        3
;*      .S units                     4        4
;*      .D units                     1        0
;*      .M units                     0        0
;*      .X cross paths               1        3
;*      .T address paths             1        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
```

- **Iteration interval** (ii). The number of cycles between the initiation of successive iterations of the loop. The smaller the iteration interval, the fewer cycles it takes to execute a loop. All of the numbers shown in each row of the feedback imply something about what the minimum iteration interval (mii) will be for the compiler to attempt initial software pipelining.

  Several things will determine what the mii of the loop is and are described in the following sections. The mii is simply the maximum of any of these individual mii’s.

- **Loop carried dependency bound**. The distance of the largest loop carry path, if one exists. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol in the assembly code saved with the –k option in the *.asm file. The number shown for the loop carried dependency bound is the minimum iteration interval due to a loop carry dependency bound for the loop.

  Often, this loop carried dependency bound is due to lack of knowledge by the compiler about certain pointer variables. When exact values of pointers are not known, the compiler must assume that any two pointers might point to the same location. Thus, loads from one pointer have an implied dependency to another pointer performing a store and vice versa. This can create large (and usually unnecessary) dependency paths. When the Loop Carried Dependency Bound is larger than the Resource Bound, this is often the culprit. Potential solutions for this are shown in Appendix A, Feedback Solutions.

- ❑ **Unpartitioned resource bound across all resources**. The best case resource bound mii before the compiler has partitioned each instruction to the A or B side. In Example 1–3, the unpartitioned resource bound is 4 because the .S units are required for 8 cycles, and there are 2 .S units.

- ❑ **Partitioned resource bound across all resources**. The mii after the instructions are partitioned to the A and B sides. In Example 1–3, after partitioning, we can see that the A side .L, .S, and .D units are required for a total of 13 cycles, making the partitioned resource bound $\lceil 13/5 \rceil = 5$. For more information, see the description of **Bound** (.L .S .D .LS .LSD) later in this section.

- ❑ **Resource partition table**. Summarizes how the instructions have been assigned to the various machine resources and how they have been partitioned between the A and B side. An asterisk is used to mark those entries that determine the resource bound value – in other words the maximum mii. Because the resources on the C6000 architecture are fairly orthogonal, many instructions can execute 2 or more different functional units. For this reason, the table breaks these functional units down by the possible resource combinations. The table entries are described below:

  - ■ **Individual Functional Units** (.L .S .D .M) show the total number of instructions that specifically require the .L, .S, .D, or .M functional units. Instructions that can operate on multiple different functional units are not included in these counts. They are described below in the Logical Ops (.LS) and Addition Ops (.LSD) rows.

  - ■ **.X cross paths** represents the total number of AtoB and BtoA. When this particular row contains an asterisk, it has a resource bottleneck and partitioning may be a problem.

  - ■ **.T address paths** represents the total number of address paths required by the loads and stores in the loop. This is actually different from the number .D units needed as some other instructions may use the .D unit. In addition, there can be cases where the number of .T address paths on a particular side might be higher than the number of .D units if .D units are partitioned evenly between A and B and .T address paths are not.

  - ■ **Long read path** represents the total number of long read port paths . All long operations with long sources use this port to do extended width (40-bit) reads. Store operations share this port so they also count toward this total. Long write path represents the total number of long write port paths. All instructions with long (40bit) results will be counted in this number.

- **Logical ops** (.LS) represents the total number of instructions that can use either the .L or .S unit.
- **Addition ops** (.LSD) represents the total number of instructions that can use either the .L or .S or .D unit.
- **Bound** (.L .S .LS) represents the resource bound value as determined by the number of instructions that use the .L and .S units. It is calculated with the following formula:

  ```
  Bound(.L .S .LS ) = ceil((.L + .S + .LS) / 2)
  ```

  Where ceil represents the ceiling function. This means you always round up to the nearest integer. In Example 1–3, if the B side needs:

  3 .L unit only instructions

  4 .S unit only instructions

  1 logical .LS instruction

  you would need at least $\lceil 8/2 \rceil$ cycles or 5 cycles to issue these.

- **Bound** (.L .S .D .LS .LSD) represents the resource bound value as determined by the number of instructions that use the .D, .L and .S unit. It is calculated with the following formula:

  ```
  Bound(.L .S .D .LS .LSD) = ceil((.L + .S + .D + .LS
  + .LSD) / 3)
  ```

  Where ceil represents the ceiling function. This means you always round up to the nearest integer. In Example 1–3, the A side needs:

  2 .L unit only instructions, 4 .S unit only instructions, 1 .D unit only instructions, 0 logical .LS instructions, and 6 addition .LSD instructions

  you would need at least $\lceil 13/3 \rceil$ cycles or 5 cycles to issue these.

1.4.3 Stage 3: Software pipeline the loop

Once the compiler has completed qualification of the loop, partitioned it, and analyzed the necessary loop carry and resource requirements, it can begin to attempt software pipelining. This section will focus on the following lines from the feedback example:

Example 1–4. Stage 3 Feedback

```
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
```

The first thing the compiler attempts during this stage, is to schedule the loop at an iteration interval (ii) equal to the mii determined in stage 2: collect loop resource and dependency graph information. In the example above, since 11 .M units on the A side was the mii bottleneck, our example starts with:

```
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
```

If the attempt was not successful, the compiler provides additional feedback to help explain why. In this case, the compiler cannot find a schedule at 11 cycles because register is live too long. For more information about live too long issues, see section 6.10, Live-Too-Long Issues, on page 6-101.

Sometimes the compiler finds a valid software pipeline schedule but one or more of the values is live too long. Lifetime of a register is determined by the cycle a value is written into it and by the last cycle this value is read by another instruction. By definition, a variable can never be live longer than the ii of the loop, because the next iteration of the loop will overwrite that value before it is read.

The compiler then proceeds to:

```
ii = 6 Did not find schedule
```

Sometimes, due to a complex loop or schedule, the compiler simply cannot find a valid software pipeline schedule at a particular iteration interval.

```
Regs Live Always : 1/5 (A/B–side)
Max Regs Live : 14/19
Max Cond Regs Live : 1/0
```

❑ **Regs Live Always** refers to the number of registers needed for variables to be live every cycle in the loop. Data loaded into registers outside the loop and read inside the loop will fall into this category.

❑ **Max Regs Live** refers to the maximum number of variable live on any one cycle in the loop. If there are 33 variables live on one of the cycles inside the loop, a minimum of 33 registers is necessary and this will not be possible with the 32 registers available on the ’C62x and ’C67x cores. In addition, this is broken down between A and B side, so if there is uneven partitioning with 30 values and there are 17 on one side and 13 on the other, the same problem will exist. This situation does not apply to the 64 registers available on the ’C64x core.

❑ **Max Cond Regs Live** tells us if there are too many conditional values needed on a given cycle. The ’C62x and ’C67x cores have 2 A side and 3 B side condition registers available. The ’C64x core has 3 A side and 3 B side condition registers available.

After failing at ii = 6, the compiler proceeds to ii = 7:

```
ii = 7 Schedule found with 3 iterations in parallel
```

It is successful and finds a valid schedule with 3 iterations in parallel. This means it is pipelined 3 deep. In other words, before iteration n has completed, iterations n+1 and n+2 have begun.

Each time a particular iteration interval fails, the ii is increased and retried. This continues until the ii is equal to the length of a list scheduled loop (no software pipelining). This example shows two possible reasons that a loop was not software pipelined. To view the full detail of all possible messages and their descriptions, see Feedback Solutions in Appendix A.

After a successful schedule is found at a particular iteration interval, more information about the loop is displayed. This information may relate to the load threshold, epilog/prolog collapsing, and projected memory bank conflicts.

```
Speculative Load Threshold : 12
```

When an epilog is removed, the loop is run extra times to finish out the last iterations, or pipe–down the loop. In doing so, extra loads from new iterations of the loop will speculatively execute (even though their results will never be used). In order to ensure that these memory accesses are not pointing to invalid memory locations, the Load Threshold value tells you how many extra bytes of data beyond your input arrays must be valid memory locations (not a memory mapped I/O etc) to ensure correct execution. In general, in the large address space of the ’C6000 this is not usually an issue, but you should be aware of this.

```
    Epilog not entirely removed
    Collapsed epilog stages : 1
```

This refers to the number of epilog stages, or loop iterations that were removed. This can produce a large savings in code size. The –mh enables speculative execution and improves the compiler’s ability to remove epilogs and prologs. However, in some cases epilogs and prologs can be partially or entirely removed without speculative execution. Thus, you may see nonzero values for this even without the –mh option.

```
   Prolog not removed
   Collapsed prolog stages : 0
```

This means that the prolog was not removed. For various technical reasons, prolog and epilog stages may not be partially or entirely removed.

```
   Minimum required memory pad : 2 bytes
```

The minimum required memory padding to use -mh is 2 bytes. See the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for more information on the -mh option and the minimum required memory padding.

```
   Minimum safe trip count :2
```

This means that the loop must execute at lease twice to safely use the software pipelined version of the loop. If this value is less than the known minimum trip count, two versions of the loop will be generated. For more information on eliminating redundant loops, see section 3.4.3.2, Eliminating Redundant Loops, on page 3-43.

Chapter 2

Compiler Optimization Tutorial

This chapter walks you through the code development flow and introduces you to compiler optimization techniques that were 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 these tutorials, you should install Code Composer Studio.

The sample code that is used in this tutorial is included on the code generation tools and Code Composer Studio CD-ROM. When you install your code generation tools, the example code is installed in c:\ti\c6000\tutorial\c_tutorial. 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 Introduction: Simple C Tuning

The ’C6000 compiler delivers the industry’s best ”out of the box” C performance. In addition to performing many common DSP optimizations, the ’C6000 compiler also performs software pipelining on various MIPS intensive loops. This feature is important for any pipelined VLIW machine to perform. In order to take full advantage of the eight available independent functional units, the dependency graph of every loop is analyzed and then scheduled by software pipelining. The more information the compiler gathers about the dependency graph, the better the resulting schedule. Because of this, the ’C6000 compiler provides many features that facilitate sending information to the compiler to ”tune” your C code.

These tutorial lessons focus on four key areas where tuning your C code can offer great performance improvements. In this tutorial, a single code example is used to demonstrate all four areas. The following example is the vector summation of two weighted vectors.

Example 2–1. Vector Summation of Two Weighted Vectors

```
void lesson_c(short *xptr, short *yptr, short *zptr, short *w_sum, int N){
   int i, w_vec1, w_vec2;
   short w1,w2;

   w1 = zptr[0];
   w2 = zptr[1];
   for (i = 0; i < N; i++){
      w_vec1 = xptr[i] * w1;
      w_vec2 = yptr[i] * w2;
      w_sum[i] = (w_vec1 + w_vec2) >> 15;
      }
   }
```

2.1.1 Project Familiarization

In order to load and run the provided example project workspace, C_tutorial.wks, you must select the appropriate target from Code Composer Setup. The included C_tutorial project was built and saved as a workspace (c_tutorial.wks). This workspace assumes a C62x fast simulator little endian target. Therefore, you need to import the same target from Code Composer Setup:

Set Up Code Composer Studio for C62x Fast Simulator Little Endian

1) Click on Setup CCStudio to setup the target.

2) From the import configuration window, select C62xx Fast Sim Ltl Endian.

3) Click on the ”Add to system configuration” button.

4) Click on the close button and exit setup.

5) Save the configuration on exit.

Load the Tutorial Workspace

1) Start Code Composer Studio.

2) From the menu bar, select File –>Workspace –>Load Workspace.

   Browse to: ti\c6000\tutorial\c_tutorial\C_tutorial.wks

3) Select C_tutorial.wks, and click Open to load the workspace.

Build tutor.out

From the menu bar, select Project –> Rebuild All

Load tutor.out

1) From the menu bar, select File –>Load Program.

   Browse to ti\c6000\tutorial\c_tutorial\

2) Select tutor.out, and click Open to load the file.

   The disassembly window with a cursor at c_int00 is displayed and highlighted in yellow.

Profile the C_tutorial project

1) From the menu bar, select Profiler–>Enable Clocks.

   The Profile Statistics window shows profile points that are already set up for each of the four functions, tutor1–4.

2) From the menu bar, select Debug–>Run.

   This updates the Profile Statistics and Dis–Assembly window. You can also click on the Run icon, or F5 key to run the program.

3) Click on the location bar at the top of the Profile Statistics window.

The second profile point in each file (the one with the largest line number) contains the data you need. This is because profile points (already set up for you at the beginning and end of each function) count from the previous profile point. Thus, the cycle count data of the function is contained in the second profile point.

You can see cycle counts of 414, 98, 79, and 55 for functions in tutor1–4, running on the C6xxx simulator. Each of these functions contains the same C code but has some minor differences related to the amount of information to which the compiler has access.

The rest of this tutorial discusses these differences and teaches you how and when you can tune the compiler to obtain performance results comparable to fully optimized hand–coded assembly.

2.1.2 Getting Ready for Lesson 1

Compile and rerun the project

1) From the menu bar, select Project–>Rebuild All, or click on the Rebuild All icon.

   All of the files are built with compiler options, –gk –mw –mhh –o3 –fr C:\ti\c6000\tutorial\c_tutorial.

2) From the menu bar, choose File–>Reload Program.

   This reloads tutor.out and returns the cursor to c_int00.

3) From the menu bar, choose Debug Run, or click the Run icon.

   The count in the Profile Statistics window now equals 2 with the cycle counts being an average of the two runs.

4) Right–click in the Profile Statistics window and select clear all.

   This clears the Profile Statistics window.

5) From the menu bar, select Debug–>Reset DSP.

6) From the menu bar, select Debug–>Restart.

This restarts the program from the entry point. You are now ready to start lesson 1.

2.2 Lesson 1: Loop Carry Path From Memory Pointers

Open lesson_c.c

In the Project View window, right–click on lesson_c.c and select Open.

Example 2–2. lesson_c.c

```
void lesson_c(short *xptr, short *yptr, short *zptr, short *w_sum, int N) {
   int i, w_vec1, w_vec2;
   short w1,w2;

   w1 = zptr[0];
   w2 = zptr[1];
   for (i = 0; i < N; i++){
      w_vec1 = xptr[i] * w1;
      w_vec2 = yptr[i] * w2;
      w_sum[i] = (w_vec1 + w_vec2) >> 15;
      }
   }
```

Compile the project and analyze the feedback in lesson_c.asm

When you rebuilt the project in Getting Ready for Lesson 1, each file was compiled with –gk –mhh –o3. Because option –k was used, a *.asm file for each *.c file is included in the rebuilt project. The –g option is used here to enable symbolic debugging which is required in CCS for profiling, but would normally be removed to enable additional optimizations across C statements.

1) Select File –>Open. From the Files of Type drop–down menu, select *.asm.

2) Select lesson_c.asm and click Open.

   Each .asm file contains software pipelining information. You can see the results in Example 2–3, Feedback From lesson_c.asm:

Example 2–3. Feedback From lesson_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 10
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     1        1
;*      .D units                     2*       1
;*      .M units                     1        1
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              1        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     2*       1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 10 Schedule found with 1 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 0
;*      Collapsed prolog stages     : 0
;*
;*      Minimum safe trip count     : 1
;*
;*----------------------------------------------------------------------------*
;*   SINGLE SCHEDULED ITERATION
;*
;*   C17:
;*              LDH     .D1T1   *A4++,A0          ;  ^ |32|
;*   ||         LDH     .D2T2   *B4++,B6          ;  ^ |32|
;*              NOP             2
;*      [ B0]   SUB     .L2     B0,1,B0           ; |33|
;*      [ B0]   B       .S2     C17               ; |33|
;*              MPY     .M1     A0,A5,A0          ;  ^ |32|
;*   ||         MPY     .M2     B6,B5,B6          ;  ^ |32|
;*              NOP             1
;*              ADD     .L1X    B6,A0,A0          ;  ^ |32|
;*              SHR     .S1     A0,15,A0          ;  ^ |32|
;*              STH     .D1T1   A0,*A3++          ;  ^ |32|
;*----------------------------------------------------------------------------*
```

A schedule with ii = 10, implies that each iteration of the loop takes ten cycles. Obviously, with eight resources available every cycle on such a small loop, we would expect this loop to do better than this.

Q Where are the problems with this loop?

A A closer look at the feedback in lesson_c.asm gives us the answer.

Q Why did the loop start searching for a software pipeline at ii=10 (for a 10–cycle loop)?

A The first iteration interval attempted by the compiler is always the maximum of the Loop Carried Dependency Bound and the Partitioned Resource Bound. In such a case, the compiler thinks there is a loop carry path equal to ten cycles:

```
;* Loop Carried Dependency Bound(^) : 10
```

The ^ symbol is interspersed in the assembly output in the comments of each instruction in the loop carry path, and is visible in lesson_c.asm.

Example 2–4. lesson_c.asm

```
L2:    ; PIPED LOOP KERNEL

           LDH     .D1T1   *A4++,A0          ;  ^ |32|
||         LDH     .D2T2   *B4++,B6          ;  ^ |32|

           NOP             2
   [ B0]   SUB     .L2     B0,1,B0           ; |33|
   [ B0]   B       .S2     L2                ; |33|

           MPY     .M1     A0,A5,A0          ;  ^ |32|
||         MPY     .M2     B6,B5,B6          ;  ^ |32|

           NOP             1
           ADD     .L1X    B6,A0,A0          ;  ^ |32|
           SHR     .S1     A0,15,A0          ;  ^ |32|
           STH     .D1T1   A0,*A3++          ;  ^ |32|
```

You can also use a dependency graph to analyze feedback, for example:

Q Why is there a dependency between STH and LDH? They do not use any common registers so how can there be a dependency?

A If we look at the original C code in lesson_c.c, we see that the LDHs correspond to loading values from xptr and yptr, and the STH corresponds to storing values into w_sum array.

Q Is there any dependency between xptr, yptr, and w_sum?

A If all of these pointers point to different locations in memory there is no dependency. However, if they do, there could be a dependency.

Because all three pointers are passed into lesson_c, there is no way for the compiler to be sure they don't alias, or point to the same location as each other. This is a memory alias disambiguation problem. In this situation, the compiler must be conservative to guarantee correct execution. Unfortunately, the requirement for the compiler to be conservative can have dire effects on the performance of your code.

We know from looking at the main calling function in tutor_d.c that in fact, these pointers all point to separate arrays in memory. However, from the compiler's local view of lesson_c, this information is not available.

Q How can you pass more information to the compiler to improve its performance?

A The next example, lesson1_c provides the answer:

Open lesson1_c.c and lesson1_c.asm

Example 2–5. lesson1_c.c

```
void lesson1_c(short * restrict xptr, short * restrict yptr, short *zptr,
               short *w_sum, int N)
{
    int i, w_vec1, w_vec2;
    short w1,w2;

    w1 = zptr[0];
    w2 = zptr[1];
    for (i = 0; i < N; i++)
    {
        w_vec1 =  xptr[i] * w1;
        w_vec2 =  yptr[i] * w2;
        w_sum[i] = (w_vec1 + w_vec2) >> 15;
    }

}
```

The only change made in lesson1_c is the addition of the restrict type qualifier for xptr and yptr. Since we know that these are actually separate arrays in memory from w_sum, in function lesson1_c, we can declare that nothing else points to these objects. No other pointer in lesson1_c.c points to xptr and no other pointer in lesson1_c.c points to zptr. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information on the restrict type qualifier. Because of this declaration, the compiler knows that there are no possible dependency between xptr, yptr, and w_sum. Compiling this file creates feedback as shown in Example 2–6, lesson1_c.asm:

Example 2–6. lesson1_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     1        1
;*      .D units                     2*       1
;*      .M units                     1        1
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              1        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     2*       1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 4
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count     : 1
;*
;*----------------------------------------------------------------------------*
;*   SINGLE SCHEDULED ITERATION
;*
;*   C17:
;*              LDH     .D1T1   *A0++,A4          ; |32|
;*   ||         LDH     .D2T2   *B4++,B6          ; |32|
;*              NOP             2
;*      [ B0]   SUB     .L2     B0,1,B0           ; |33|
;*      [ B0]   B       .S2     C17               ; |33|
;*              MPY     .M1     A4,A5,A3          ; |32|
;*   ||         MPY     .M2     B6,B5,B7          ; |32|
;*              NOP             1
;*              ADD     .L1X    B7,A3,A3          ; |32|
;*----------------------------------------------------------------------------*
```

At this point, the Loop Carried Dependency Bound is zero. By simply passing more information to the compiler, we allowed it to improve a 10–cycle loop to a 2–cycle loop.

Lesson 4 in this tutorial shows how the compiler retrieves this type of information automatically by gaining full view of the entire program with program level optimization switches.

A special option in the compiler, -mt, tells the compiler to ignore alias disambiguation problems like the one described in lesson_c. Try using this option to rebuild the original lesson_c example and look at the results.

Rebuild lesson_c.c using the –mt option

1) From the menu bar, select Project–>Options.

   The Build Options dialog window appears.

2) Select the Compiler tab.

3) In the Category box, select Advanced.

4) In the Aliasing drop-down box, select No Bad Alias Code.

   The -mt option will appear in the options window.

5) Click OK to set the new options.

6) Select lesson_c.c by selecting it in the project environment, or double–clicking on it in the Project View window.

7) Select Project–>Build, or click on the Build icon.

   If prompted, reload lesson_c.asm.

8) From the menu bar, select File–>Open, and select lesson_c.asm in Open window.

You can now view lesson_c.asm in the main window. In the main window, you see that the file header contains a description of the options that were used to compile the file under Global File Parameters. The following line implies that –mt was used:

```
;* Memory Aliases : Presume not aliases (optimistic)
```

9) Scroll down until you see the feedback embedded in the lesson_c.asm file.

   You now see the following:

```
;* Loop Carried Dependency Bound(^) : 0
;* ii = 2 Schedule found with 5 iterations in parallel
```

This indicates that a 2–cycle loop was found. Lesson 2 will address information about potential improvements to this loop.

Table 2–1. Status Update: Tutorial example lesson_c lesson1_c

Tutorial Example	**Lesson_c**	**Lesson1_c**
Potential pointer aliasing info (discussed in Lesson 1)	×	✓
Loop count info – minimum trip count (discussed in Lesson 2)	×	×
Loop count info – max trip count factor (discussed in Lesson 2)	×	×
Alignment info – xptr & yptr aligned on a word boundary (discussed in Lesson 3)	×	×
Cycles per iteration (discussed in Lesson 1–3)	10	2

2.3 Lesson 2: Balancing Resources With Dual-Data Paths

Lesson 1 showed you a simple way to make large performance gains in lesson_c. The result is lesson1_c with a 2–cycle loop.

Q Is this the best the compiler can do? Is this the best that is possible on the VelociTI architecture?

A Again, the answers lie in the amount of knowledge to which the compiler has access. Let’s analyze the feedback of lesson1_c to determine what improvements could be made:

Open lesson1_c.asm

Example 2–7. lesson1_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     1        1
;*      .D units                     2*       1
;*      .M units                     1        1
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              1        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     2*       1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Collapsed epilog stages     : 4
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 2
;*
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count     : 1
;*
;*----------------------------------------------------------------------------*
;*   SINGLE SCHEDULED ITERATION
;*
;*   C17:
;*              LDH     .D1T1   *A0++,A4          ; |32|
;*   ||         LDH     .D2T2   *B4++,B6          ; |32|
;*              NOP             2
;*      [ B0]   SUB     .L2     B0,1,B0           ; |33|
;*      [ B0]   B       .S2     C17               ; |33|
;*              MPY     .M1     A4,A5,A3          ; |32|
;*   ||         MPY     .M2     B6,B5,B7          ; |32|
;*              NOP             1
;*              ADD     .L1X    B7,A3,A3          ; |32|
;*----------------------------------------------------------------------------*
```

The first iteration interval (ii) attempted was two cycles because the Partitioned Resource Bound is two. We can see the reason for this if we look below at the .D units and the .T address paths. This loop requires two loads (from xptr and yptr) and one store (to w_sum) for each iteration of the loop.

Each memory access requires a .D unit for address calculation, and a .T address path to send the address out to memory. Because the ’C6000 has two .D units and two .T address paths available on any given cycle (A side and B side), the compiler must partition at least two of the operations on one side (the A side). That means that these operations are the bottleneck in resources (highlighted with an *) and are the limiting factor in the Partitioned Resource Bound. The feedback in lesson1_c.asm shows that there is an imbalance in resources between the A and B side due, in this case, to an odd number of operations being mapped to two sides of the machine.

Q Is it possible to improve the balance of resources?

A One way to balance an odd number of operations is to unroll the loop. Now, instead of three memory accesses, you will have six, which is an even number. You can only do this if you know that the loop counter is a multiple of two; otherwise, you will incorrectly execute too few or too many iterations. In tutor_d.c, LOOPCOUNT is defined to be 40, which is a multiple of two, so you are able to unroll the loop.

Q Why did the compiler not unroll the loop?

A In the limited scope of lesson1_c, the loop counter is passed as a parameter to the function. Therefore, it might be any value from this limited view of the function. To improve this scope you must pass more information to the compiler. One way to do this is by inserting a MUST_ITERATE pragma. A MUST_ITERATE pragma is a way of passing iteration information to the compiler. There is no code generated by a MUST_ITERATE pragma; it is simply read at compile time to allow the compiler to take advantage of certain conditions that may exist. In this case, we want to tell the compiler that the loop will execute a multiple of 2 times; knowing this information, the compiler can unroll the loop automatically.

Unrolling a loop can incur some minor overhead in loop setup. The compiler does not unroll loops with small loop counts because unrolling may not reduce the overall cycle count. If the compiler does not know what the minimum value of the loop counter is, it will not automatically unroll the loop. Again, this is information the compiler needs but does not have in the local scope of lesson1_c. You know that LOOPCOUNT is set to 40, so you can tell the compiler that N is greater than some minimum value. lesson2_c demonstrates how to pass these two pieces of information.

Open lesson2_c.c

Example 2–8. lesson2_c.c

```
void lesson2_c(short * restrict xptr, short * restrict yptr, short *zptr,
               short *w_sum, int N)
{
    int i, w_vec1, w_vec2;
    short w1,w2;

    w1 = zptr[0];
    w2 = zptr[1];
    #pragma MUST_ITERATE(20, , 2);
    for (i = 0; i < N; i++)
    {
        w_vec1 =  xptr[i] * w1;
        w_vec2 =  yptr[i] * w2;
        w_sum[i] = (w_vec1+w_vec2) >> 15;
    }

}
```

In lesson2_c.c, no code is altered, only additional information is passed via the MUST_ITERATE pragma. We simply guarantee to the compiler that the trip count (in this case the trip count is N) is a multiple of two and that the trip count is greater than or equal to 20. The first argument for MUST_ITERATE is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information about the MUST_ITERATE pragma.

For this example, we chose a trip count large enough to tell the compiler that it is more efficient to unroll. Always specify the largest minimum trip count that is safe.

Open lesson2_c.asm and examine the feedback

Example 2–9. lesson2_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 10
;*      Known Maximum Trip Count         : 1073741823
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     2        1
;*      .D units                     3*       3*
;*      .M units                     2        2
;*      .X cross paths               1        1
;*      .T address paths             3*       3*
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        1     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 2
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 3
;*
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count    : 4
```

Notice the following things in the feedback:

A schedule with three cycles (ii=3): You can tell by looking at the .D units and .T address paths that this 3–cycle loop comes after the loop has been unrolled because the resources show a total of six memory accesses evenly balanced between the A side and B side. Therefore, our new effective loop iteration interval is 3/2 or 1.5 cycles.

A Known Minimum Trip Count of 10: This is because we specified the count of the original loop to be greater than twenty and a multiple of two (at least twenty–two) and after unrolling, this is cut in half. Also, a new line, Known Maximum Trip Count, is displayed in the feedback. This represents the maximum signed integer value divided by two, or 3FFFFFFFh.

Therefore, by passing information without modifying the loop code, compiler performance improves from a 10–cycle loop to 2 cycles and now to 1.5 cycles.

Q Is this the lower limit?

A Check out Lesson 3 to find out!

Table 2–2. Status Update: Tutorial example lesson_c lesson1_c lesson2_c

Tutorial Example	**Lesson_c**	**Lesson1_c**	**Lesson2_c**
Potential pointer aliasing info (discussed in Lesson 1)	×	✓	✓
Loop count info – minimum trip count (discussed in Lesson 2)	×	×	✓
Loop count info – max trip count factor (discussed in Lesson 2)	×	×	✓
Alignment info – xptr & yptr aligned on a word boundry (discussed in Lesson 3)	×	×	×
Cycles per iteration (discussed in Lesson 1–3)	10	2	1.5

2.4 Lesson 3: Packed Data Optimization of Memory Bandwidth

Lesson 2 produced a 3–cycle loop that performed two iterations of the original *vector sum of two weighted vectors*. This means that each iteration of our loop now performs six memory accesses, four multiplies, two adds, two shift operations, a decrement for the loop counter, and a branch. You can see this phenomenon in the feedback of lesson2_c.asm.

Open lesson2_c.asm

Example 2–10. lesson2_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 10
;*      Known Maximum Trip Count         : 1073741823
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 3
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     2        1
;*      .D units                     3*       3*
;*      .M units                     2        2
;*      .X cross paths               1        1
;*      .T address paths             3*       3*
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        1     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 2
;*
;*      Prolog not entirely removed
;*      Collapsed prolog stages     : 3
;*
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count     : 4
;*
;*----------------------------------------------------------------------------*
```

The six memory accesses appear as .D and .T units. The four multiplies appear as .M units. The two shifts and the branch show up as .S units. The decrement and the two adds appear as .LS and .LSD units. Due to partitioning, they don't all show up as .LSD operations. Two of the adds must read one value from the opposite side. Because this operation cannot be performed on the .D unit, the two adds are listed as .LS operations.

By analyzing this part of the feedback, we can see that resources are most limited by the memory accesses; hence, the reason for an asterisk highlighting the .D units and .T address paths.

Q Does this mean that we cannot make the loop operate any faster?

A Further insight into the 'C6000 architecture is necessary here.

The C62x fixed-point device loads and/or stores 32 bits every cycle. In addition, the C67x floating-point and 'C64x fixed-point device loads two 64-bit values each cycle. In our example, we load four 16-bit values and store two 16–bit values every three cycles. This means we only use 32 bits of memory access every cycle. Because this is a resource bottleneck in our loop, increasing the memory access bandwidth further improves the performance of our loop.

In the unrolled loop generated from lesson2_c, we load two consecutive 16-bit elements with LDHs from both the xptr and yptr array.

Q Why not use a single LDW to load one 32-bit element, with the resulting register load containing the first element in one-half of the 32-bit register and the second element in the other half?

A This is called Packed Data optimization. Two 16-bit loads are effectively performed by one single 32-bit load instruction.

Q Why doesn't the compiler do this automatically in lesson2_c?

A Again, the answer lies in the amount of information the compiler has access to from the local scope of lesson2_c.

In order to perform a LDW (32–bit load) on the 'C62x and 'C67x cores, the address must be aligned to a word address; otherwise, incorrect data is loaded. An address is word–aligned if the lower two bits of the address are zero. Unfortunately, in our example, the pointers, xptr and yptr, are passed into lesson2_c and there is no local scope knowledge as to their values. Therefore, the compiler is forced to be conservative and assume that these pointers might not be aligned. Once again, we can pass more information to the compiler, this time via the _nassert statement.

Open lesson3_c.c

Example 2–11. lesson3_c.c

```
#define WORD_ALIGNED(x) (_nassert(((int)(x) & 0x3) == 0))

void lesson3_c(short * restrict xptr, short * restrict yptr, short *zptr,
               short *w_sum, int N)
{
    int i, w_vec1, w_vec2;
    short w1,w2;

    WORD_ALIGNED(xptr);
    WORD_ALIGNED(yptr);

    w1 = zptr[0];
    w2 = zptr[1];
    #pragma MUST_ITERATE(20, , 2);
    for (i = 0; i < N; i++)
    {
        w_vec1 =  xptr[i] * w1;
        w_vec2 =  yptr[i] * w2;
        w_sum[i] = (w_vec1+w_vec2) >> 15;
    }
}
```

By asserting that xptr and yptr addresses ”anded” with 0x3 are equal to zero, the compiler knows that they are word aligned. This means the compiler can perform LDW and packed data optimization on these memory accesses.

Open lesson3_c.asm

Example 2–12. lesson3_c.asm

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Trip Count         : 10
;*      Known Maximum Trip Count         : 1073741823
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     2*       1
;*      .D units                     2*       2*
;*      .M units                     2*       2*
;*      .X cross paths               1        1
;*      .T address paths             2*       2*
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           1        1     (.L or .S unit)
;*      Addition ops (.LSD)          0        1     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2*       1
;*      Bound(.L .S .D .LS .LSD)     2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 6 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 2
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 8 bytes
;*
;*      Minimum safe trip count     : 8
;*
```

Success! The compiler has fully optimized this loop. You can now achieve two iterations of the loop every two cycles for one cycle per iteration throughout.

The .D and .T resources now show four (two LDWs and two STHs for two iterations of the loop).

Table 2–3. Status Update: Tutorial example lesson_c lesson1_c lesson2_c lesson3_c

Tutorial Example	Lesson_c	Lesson1_c	Lesson2_c	Lesson3_c
Potential pointer aliasing info (discussed in Lesson 1)	×	✓	✓	✓
Loop count info – minimum trip count (discussed in Lesson 2)	×	×	✓	✓
Loop count info – max trip count factor (discussed in Lesson 2)	×	×	✓	✓
Alignment info – xptr & yptr aligned on a word boundary (discussed in Lesson 3)	×	×	×	✓
Cycles per iteration (discussed in Lessons 1–3)	10	2	1.5	1

2.5 Lesson 4: Program Level Optimization

In Lesson 3, you learned how to pass information to the compiler. This increased the amount of information visible to the compiler from the local scope of each function.

Q Is this necessary in all cases?

A The answer is no, not in all cases. First, if this information already resides locally inside the function, the compiler has visibility here and restrict and MUST_ITERATE statements are not usually necessary. For example, if xptr and yptr are declared as local arrays, the compiler does not assume a dependency with w_sum. If the loop count is defined in the function or if the loop simply described from one to forty, the MUST_ITERATE pragma is not necessary.

Secondly, even if this type of information is not declared locally, the compiler can still have access to it in an automated way by giving it a program level view. This module discusses how to do that.

The ’C6000 compiler provides two valuable switches, which enable program level optimization: –pm and –op2. When these two options are used together, the compiler can automatically extract all of the information we passed in the previous examples. To tell the compiler to use program level optimization, you need to turn on –pm and –op2.

Enable program level optimization

1) From the menu bar, choose Project –> Options, and click on the Basic category.

2) Select No External Refs in the Program Level Optimization drop-down box. This adds –pmm (same as –pm) and –op2 to the command line.

View profile statistics

1) Clear the Profile Statistics window by right clicking on it and selecting Clear All.

2) Rebuild the program by selecting Project –> Rebuild All.

3) Reload the program by selecting File –> Reload Program.

4) Now run the program by selecting Debug –> Run.

   The new profile statistics should appear in the Profile Statistics window, as in Example 2–13.

Example 2–13. Profile Statistics

```
Location                   Count  Average  Total  Maximum  Minimum
lesson_c.c line 27             1   5020.0   5020     5020     5020
lesson_c.c line 36             1     60.0     60       60       60
lesson1_c.c line 37            1     60.0     60       60       60
lesson2_c.c line 39            1     60.0     60       60       60
lesson3_c.c line 44            1     60.0     60       60       60
lesson1_c.c line 27            1     12.0     12       12       12
lesson2_c.c line 29            1     12.0     12       12       12
lesson3_c.c line 35            1     12.0     12       12       12
```

This is quite a performance improvement. The compiler automatically extracts and acts upon all the information that we passed in Lessons 1 to 3. Even the original untouched tutor1 is 100% optimized by discounting memory dependencies, unrolling, and performing packed data optimization.

Table 2–4. Status Update: Tutorial example lesson_c lesson1_c lesson2_c lesson3_c

Tutorial Example	Lesson_c	Lesson1_c	Lesson2_c	Lesson3_c
Potential pointer aliasing info (discussed in Lesson 1)	×	✓	✓	✓
Loop count info – minimum trip count (discussed in Lesson 2)	×	×	✓	✓
Loop count info – max trip count factor (discussed in Lesson 2)	×	×	✓	✓
Alignment info – xptr & yptr aligned on a word boundary (discussed in Lesson 3)	×	×	×	✓
Cycles per iteration (discussed in Lesson 1–3)	10	2	1.5	1
Cycles per iteration with program level optimization (discussed in Lesson 4)	1	1	1	1

This tutorial has shown you that a lot can be accomplished by both tuning your C code and using program level optimization. Many different types of tuning optimizations can be done in addition to what was presented here.

We recommend you use Appendix A, Feedback Solutions, when tuning your code to get “how to” answers on all of your optimizing C questions. You can also use the Feedback Solutions Appendix as a tool during development. We believe this offers a significant advantage to TI customers and we plan on continuing to drive a more developer–friendly environment in our future releases.

2.6 Lesson 5: Writing Linear Assembly

When the compiler does not fully exploit the potential of the ’C6000 architecture, you may be able to get better performance by writing your loop in linear assembly. Linear assembly is the input for the assembly optimizer.

Linear assembly is similar to regular ’C6000 assembly code in that you use ’C6000 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 ’C6000 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.

Let’s look at a new example, iircas4, which will show the benefit of using linear assembly. The compiler does not not optimally partition this loop. Thus, the iircas4 function does not improve with the C modification techniques we saw in the first portion of the chapter. In order to get the best partition, we must write the function in partitioned linear assembly.

In order to follow this example in Code Composer Studio, you must open the workspace, l_tutorial.wks, located in c:\ti\c6000\tutorial\c_tutorial. Build the program and look at the software pipeline information feedback in the generated assembly files.

Example 2–14. Using the iircas4 Function in C

```
void iircas4_1(const int n, const short (* restrict c)[4], int (*d)[2],
               int *y)
{
    int k0, k1, i;
    int y0 = y[0];
    int y1 = y[1];

    _nassert(((int)(c) & 0x3) == 0));

    #pragma MUST_ITERATE(10);

    for (i = 0; i < n; i++)
    {
        k0      = c[i][1] * (d[i][1]>>16) + c[i][0] * (d[i][0]>>16) + y0;
        y0      = c[i][3] * (d[i][1]>>16) + c[i][2] * (d[i][0]>>16) + k0;
        k1      = c[i][1] * (d[i][0]>>16) + c[i][0] * (k0>>16) + y1;
        y1      = c[i][3] * (d[i][0]>>16) + c[i][2] * (k0>>16) + k1;

        d[i][1] = k0;
        d[i][0] = k1;
    }

    y[0] = y0;
    y[1] = y1;
}
```

Example 2–15 shows the assembly output from Example 2–14

Example 2–15. Software Pipelining Feedback From the iircas4 C Code

```
;*-------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 10
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 2
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     2        4
;*      .M units                     4        4
;*      .X cross paths               5*       3
;*      .T address paths             2        4
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           2        1    (.L or .S unit)
;*      Addition ops (.LSD)          4        3    (.L or .S or .D unit)
;*      Bound(.L .S .LS)             2        1
;*      Bound(.L .S .D .LS .LSD)     3        3
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Schedule found with 4 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 2
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 16 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*-------------------------------------------------------------------------*
```

From the feedback in the generated .asm file, we can see that the compiler generated a suboptimal partition. Partitioning is placing operations and operands on the A side or B side. We can see that the Unpartioned Resource Bound is 4 while the Partitioned Resource Bound is 5. When the Partitioned

Resource Bound is higher, this usually means we can make a better partition by writing the code in linear assembly.

Notice that there are 5 cross path reads on the A side and only 3 on the B side. We would like 4 cross path reads on the A side and 4 cross path reads on the B side. This would allow us to schedule at an iteration interval (ii) of 4 instead of the current ii of 5. Example 2–16 shows how to rewrite the iircas4 ( ) function Using Linear Assembly.

Example 2–16. Rewriting the iircas4 ( ) Function in Linear Assembly

```
        .def    _iircas4_sa

_iircas4_sa:    .cproc  AI,C,BD,AY

        .no_mdep

        .reg    BD0,BD1,AA,AB,AJ0,AF0,AE0,AG0,AH0,AY0,AK0,AM0,BD00
        .reg    BA2,BB2,BJ1,BF1,BE1,BG1,BH1,BY1,BK1,BM1

        LDW     .D2     *+AY[0],AY0
        LDW     .D2     *+AY[1],BY1

        .mptr   C,  bank+0, 8
        .mptr   BD, bank+4, 8

LOOP:   .trip   10
        LDW     .D2T1   *C++, AA        ; a0 = c[i][0], a1 = c[i][1]
        LDW     .D2T1   *C++, AB        ; b0 = c[i][2], b1 = c[i][3]
        LDW     .D1T2   *BD[0], BD0     ; d0 = d[i][0]
        LDW     .D1T2   *BD[1], BD1     ; d1 = d[i][1]

        MPYH    .1      BD1, AA, AE0    ; e0 = (d1 >> 16) * a1
        MPYHL   .1      BD0, AA, AJ0    ; j0 = (d0 >> 16) * a0
        MPYH    .1      BD1, AB, AG0    ; g0 = (d1 >> 16) * b1
        MPYHL   .1      BD0, AB, AF0    ; f0 = (d0 >> 16) * b0

        ADD     .1      AJ0, AE0, AH0   ; h0 = j0 + e0
        ADD     .1      AH0, AY0, AK0   ; k0 = h0 + y0
        ADD     .1      AF0, AG0, AM0   ; m0 = f0 + g0
        ADD     .1      AM0, AK0, AY0   ; y0 = m0 + k0

        MV      .2      AA,BA2
        MV      .2      AB,BB2
        MV      .2      BD0,BD00
        STW     .D1T1   AK0, *BD[1]     ; d[i][1] = k0

        MPYH    .2      BD00, BA2, BE1  ; e1 = (d0 >> 16) * a1
        MPYHL   .2      AK0, BA2, BJ1   ; j1 = (k0 >> 16) * a0
        MPYH    .2      BD00, BB2, BG1  ; g1 = (d0 >> 16) * b1
        MPYHL   .2      AK0, BB2, BF1   ; f1 = (k0 >> 16) * b0

        ADD     .2      BJ1, BY1, BH1   ; h1 = j1 + y1
        ADD     .2      BH1, BE1, BK1   ; k1 = h1 + e1
        ADD     .2      BF1, BG1, BM1   ; m1 = f1 + g1
        ADD     .2      BM1, BK1, BY1   ; y1 = m1 + k1

        STW     .D1T2   BK1, *BD++[2]   ; d[i][0] = k1

        SUB     .1      AI,1,AI         ; i--
[AI]    B       .1      LOOP            ; for

        STW     .D2T1   AY0,*+AY[0]
        STW     .D2T2   BY1,*+AY[1]

        .endproc
```

The following example shows the software pipeline feedback from Example 2–16.

Example 2–17. Software Pipeline Feedback from Linear Assembly

```
;*------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Known Minimum Trip Count         : 10
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 3
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 4
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     0        0
;*      .S units                     1        0
;*      .D units                     4*       2
;*      .M units                     4*       4*
;*      .X cross paths               4*       4*
;*      .T address paths             3        3
;*      Long read paths              1        1
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        2    (.L or .S unit)
;*      Addition ops (.LSD)          5        5    (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        1
;*      Bound(.L .S .D .LS .LSD)     4*       3
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 4  Schedule found with 5 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 3
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 24 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*------------------------------------------------------------------------*
```

Notice in Example 2–16 that each instruction is manually partitioned. From the software pipeline feedback information in Example 2–17, you can see that a software pipeline schedule is found at ii = 4. This is a result of rewriting the iircas4 ( ) function in linear assembly, as shown in Example 2–16.

Chapter 3

Optimizing C/C++ Code

You can maximize C/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**

3.1 Writing C/C++ Code

This chapter shows you how to analyze and tailor your code to be sure you are getting the best performance from the ’C6000 architecture.

3.1.1 Tips on Data Types

Give careful consideration to the data type size when writing your code. The ’C6000 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 ’C6000 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 ’C6000 (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 ’C6700, 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.

- When using the ’C6400 device, use the –mv6400 compiler switch so the code generated will use the device’s additional hardware and instructions.

3.1.2 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/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 *TMS320C6000 Optimizing C/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/C++ code, you can then evaluate the performance results by running the code and looking at the instructions generated by the compiler.

3.2 Compiling C/C++ Code

The 'C6000 compiler offers high-level language support by transforming your C/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/C++ compiler and the options discussed in this chapter, see the *TMS320C6000 Optimizing C/C++ Compiler User's Guide (SPRU187)*.

3.2.1 Compiler Options

Options control the operation of the compiler. This section introduces you to the recommendd options for performance, optimization, and code size. Considerations of optimization versus performance are also discussed.

The options described in Table 3–1 are obsolete or intended for debugging, and could potentially decrease performance and increase code size. Avoid using these options with performance critical code.

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

Option	Description
–g/–s/ –ss/–mg	These options limit the amount of optimization across C statements leading to larger code size and slower execution.
–mu	Disables software pipelining for debugging. Use –ms2/–ms3 instead to reduce code size which will disable software pipelining among other code size optimizations.
–o1/–o0	Always use –o2/–o3 to maximize compiler analysis and optimization. Use code size flags (–ms*n*) to tradeoff between performance and code size.
–mz	Obsolete. On pre–3.00 tools, this option may have improved your code, but with 3.00+ compilers, this option will decrease performance and increase code size.

The options in Table 3–2 can improve performance but require certain characteristics to be true, and are described below.

Table 3–2. Compiler Options for Performance

Option	Description
–mh <*n*>	Allows speculative execution. The appropriate amount of padding must be available in data memory to insure correct execution. This is normally not a problem but must be adhered to.
–mi<*n*>	Describes the interrupt threshold to the compiler. If you know that NO interrupts will occur in your code, the compiler can avoid enabling and disabling interrupts before and after software pipelined loops for a code size and performance improvement. In addition, there is potential for performance improvement where interrupt registers may be utilized in high register presure loops.(See section 7.4.)
–ms0/–ms1	Optimizes primarily for performance, and secondly for code size. Should be used on all but the most performance critical routines.
–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. (See section 6.2)
–o3†	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
–oi0	Disables all automatic size–controlled inlining, (which is enabled by –o3). User specified inlining of functions is still allowed.
–op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
–pm‡	Combines source files to perform program–level optimization.

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

The options described in Table 3–3 are recommended for control code, and will result in smaller code size with minimal performance degradation.

Table 3–3. Compiler Options for Control Code

Option	Description
–o3[†]	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
–pm[‡]	Combines source files to perform program–level optimization.
–op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
–oi0	Disables all automatic size–controlled inlining, (which is enabled by –o3). User specified inlining of functions is still allowed.
–ms2–ms3	Optimizes primarily for code size, and secondly for performance.

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

The options described in Table 3–4 provide information, but do not affect performance or code size.

Table 3–4. Compiler Options for Information

Option	Description
–mw	Use this option to produce additional compiler feedback. This option has no performance or code size impact.
–k	Keeps the assembly file so that you can inspect and analyze compiler feedback. This option has no performance or code size impact.
–mg	Enables automatic function level profiling with the loader. Can result in minor performance degradation around function call boundaries only.
–s/–ss	Interlists C/C++ source or optimizer comments in assembly.

3.2.2 Memory Dependencies

To maximize the efficiency of your code, the ’C6000 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:

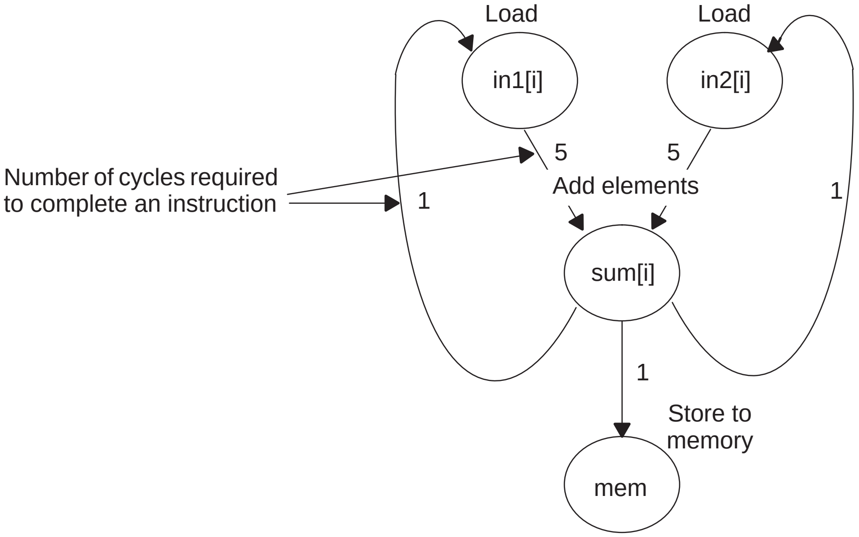
- Use the const keyword to indicate which objects are not changed by a function.
- Use the restrict keyword to indicate that a pointer is the only pointer that can point to a particular object in the scope in which the pointer is declared.
- Use the –pm (program-level optimization) option, which gives 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 *TMS320C6000 Optimizing C/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 3–1 shows the C code for a basic vector sum. Figure 3–1 shows the dependency graph for this basic vector sum. (For more information, see section 6.3.4, *Drawing a Dependency Graph,* on page 6-11.)

Example 3–1. 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 3–1. Dependency Graph for Vector Sum #1



The dependency graph in Figure 3–1 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);
  ```

3.2.2.1 The const Keyword

In Figure 3–1, the reads from in1 and in2 finish before the write to sum within a single iteration. However, the ’C6000 compiler uses software pipelining to execute multiple iterations in parallel and, therefore, must determine memory dependencies that exist across loop iterations.

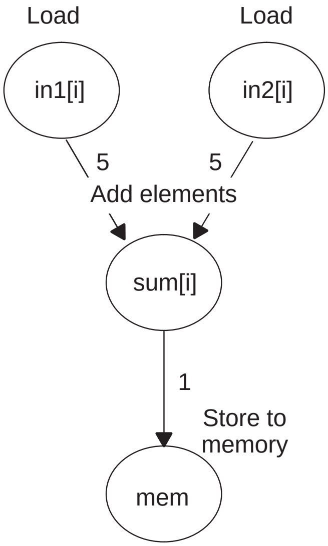
To help the compiler, you can qualify an object with the const keyword, which indicates that a variable or the memory referenced by a variable will not be changed, but will remain constant. It is good coding practice to use the const keyword wherever you can, because it is a simple way to increase the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

Example 3–2 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 3–2 shows the revised dependency graph for the code in the inner loop.

Example 3–2. 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 3–2. Dependency Graph for Vector Sum #2



Example 3–3 shows the output of the compiler for the vector sum in Example 3–2. 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 3–3. Compiler Output for Vector Sum Code

```
L2:    ; PIPED LOOP KERNEL

           ADD     .L1X    B5,A3,A5           ; |6|
|| [ B0]   B       .S2     L2                 ; @@|6|
|| [ A1]   LDH     .D1T1   *A4++,A3           ; @@@|6|

   [ A2]   SUB     .L1     A2,1,A2            ;
|| [ A1]   SUB     .S1     A1,1,A1            ;
|| [!A2]   STH     .D1T1   A5,*A0++           ; |6|
|| [ B0]   SUB     .L2     B0,1,B0            ; @@@|6|
|| [ A1]   LDH     .D2T2   *B4++,B5           ; @@@|6|
```

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.

Do *not* use the const keyword with code such as listed in Example 3–4. By using the const keyword in Example 3–4, 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 can cause an incorrect program because both *a* and *b* point to the same object —*array*.

Example 3–4. 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, 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, 41, 42,
                           43, 44};
       short *ptr1, *ptr2;

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

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

3.2.2.2 The Restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that may be applied to pointers, references, and arrays. Its use represents a guarantee by the programmer that within the scope of the pointer declaration, the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In the example that follows, you can use the restrict keyword to tell the compiler that a and b never point to the same object in foo (and the objects' memory that foo accesses does not overlap).

Example 3–5. Use of the Restrict Type Qualifier With Pointers

```
void foo(int * restrict a, int * restrict b)
{
  /* foo’s code here */
}
```

This example is a use of the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

Example 3–6. Use of the Restrict Type Qualifier With Arrays

```
void func1(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

3.2.2.3 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.

3.2.2.4 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 3–1, the compiler uses the assumption that that in1 and in2 do not alias memory pointed to by sum and, therefore, eliminates memory dependencies among the instructions that access those variables.

You would get the same loop kernel listed in Example 3–3. 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 *TMS320C6000 Optimizing C/C++ Compiler User’s Guide*.

3.3 Profiling Your Code

In large applications, it makes sense to optimize the most important sections of code first. You can use the information generated by profiling options to get started. You can use several different methods to profile your code. These methods are described below.

3.3.1 Using the Standalone Simulator (load6x) to Profile

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, you can use the –g 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.

3.3.1.1 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 are stored in a file called by the same name as the .out file, but with the .vaa extension.

If you used the –mg profiling option when compiling and linking ”example.out”, use the –g option to create a file in which you can view the profiling results. For example, enter the following on your command line:

```
load6x –g example.out
```

Now, you can view the file ”example.vaa” to see the results of the profile session created with the –mg option on the .out file.

Your new file, ”example.vaa” should have been created in the same directory as the .out file. You can edit the .vaa file with a text editor. You should see something like this:

```
Program Name: example.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.

3.3.1.2 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. The following example demonstrates how to include the clock() function in your C code.

Example 3–7. Including the clock( ) Function

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

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

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

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

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

start = clock();
iir1(coefs, x, optr, state);
stop = clock();
printf(”iir1 cycles: %d\n”, stop – start – overhead);
}
Note: When using this method, remember to calculate the
overhead and include the appropriate header files.
```

3.3.2 Profiling in Code Composer Studio

The Compiler Optimization Tutorial, Chapter 2, describes how profiling is achieved with CCS. The introduction provides you with an example of how to set probe points and provides you with example code to walk through.

3.4 Refining C/C++ Code

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

- Using intrinsics to replace complicated C/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 (’C64x and ’C67x only)

3.4.1 Using Intrinsics

The ’C6000 compiler provides intrinsics, special functions that map directly to inlined ’C62x/’C64x/’C67x instructions, to optimize your C/C++ code quickly. All instructions that are not easily expressed in C/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/C++ code only by writing a multicycle function, such as the one in Example 3–8.

Example 3–8. 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 3–9).

Example 3–9. Saturated Add With Intrinsics

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

Table 3–5 lists the ’C6000 intrinsics. For more information on using intrinsics, see the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide*.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics

C Compiler Intrinsic	Assembly Instruction	Description	Device

int **_abs(**int *src2***);**
int**_labs(**long *src2***);**
	ABS	Returns the saturated absolute value of src2.	

int **_abs2 (**int src**);**	**ABS2**	Calculates the absolute value for each 16–bit value.	’C64x
int **_add2(**int *src1*, int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.	
int **_add4 (**int *src1*, int *src2***);**	**ADD4**	Performs 2s–complement addition to pairs of packed 8–bit numbers.	’C64x
int **_avg2 (**int *src1*, *int src2***);**	**AVG2**	Calculates the average for each pair of signed 16–bit values.	’C64x
unsigned **_avgu4(**unsigned, unsigned**);**	**AVGU4**	Calculates the average for each pair of unsigned 8–bit values.	’C64x
unsigned **_bitc4 (**unsigned src**);**	**BITC4**	For each of the 8–bit quantities in src, the number of 1 bits is written to the corresponding position in the return value.	’C64x
unsigned **_bitr (**unsigned src**);**	**BITR**	Reverses the order of the bits.	’C64x
uint _**clr(**uint *src2,* uint *csta*, uint *cstb***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.	
unsigned _**clrr(**uint *src1*, int *src2***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of the source register.	
int **_cmpeq2 (**int *src1*, int *src2*);	**CMPEQ2**	Performs equality comparisons on each pair of 16–bit values. Equality results are packed into the two least–significant bits of the return value.	’C64x

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_cmpeq4 (**int *src1*, int *src2***);**	**CMPEQ4**	Performs equality comparisons on each pair of 8–bit values. Equality results are packed into the four least–significant bits of the return value.	’C64x
int **_cmpgt2 (**int *src1*, int *src2***);**	**CMPGT2**	Compares each pair of signed 16–bit values. Results are packed into the two least–significant bits of the return value.	’C64x
unsigned **_cmpgtu4 (**unsigned *src1*, unsigned *src2***);**	**CMPGTU4**	Compares each pair of unsigned 8–bit values. Results are packed into the four least–significant bits of the return value.	’C64x
unsigned **_deal (**unsigned src**);**	**DEAL**	The odd and even bits of src are extracted into two separate 16–bit values.	’C64x

int **_dotp2 (**int *src1*, int *src2***);**
long **_ldotp2 (**int *src1*, int *src2***);**
	DOTP2
	LDOTP2
		The product of signed 16–bit values in src1 is added to the product of signed 16–bit values in src2.	’C64x

int **_dotpn2 (**int *src1*, int *src2***);**	**DOTPN2**	The product of signed 16–bit values in src2 is subtracted from the product of signed 16–bit values in src1.	’C64x
int **_dotpnrsu2 (**int *src1*, unsigned *src2***);**	**DOTPNRSU2**	The product of unsigned 16–bit values in src2 is subtracted from the product of signed 16–bit values in src1. 2^{15} is added and the result is sign shifted right by 16.	’C64x
int **_dotprsu2 (**int *src1*, unsigned *src2***);**	**DOTPRSU2**	Adds the result of the product of the first signed pair and the unsigned second pair of 16–bit values. 2^{15} is added and the result is sign shifted right by 16.	’C64x

unsigned **_dotpu4 (**unsigned *src1*, unsigned *src2***);**
int **_dotpsu4 (**int *src1*, unsigned *src2***);**
	DOTPU4
	DOTPSU4
		For each pair of 8–bit values in src1 and src2, the 8–bit value from src1 is multiplied with the 8–bit value from src2. The four products are summed together.	’C64x

int**_dpint(**double**)**;	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register.	’C67x

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
int **_ext(**uint *src2,* uint *csta*, int *cstb***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	
int **_extr(**int *src2*, int *src1***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_extu(**uint *src2,* uint *csta*, uint *cstb***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_extur(**uint *src2*, int *src1***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right; csta and cstb are the shift left and shift right amounts, respectively.	
uint **_ftoi(**float**);**		Reinterprets the bits in the float as an unsigned integer. (Ex: _ftoi(1.0) == 1065353216U)	’C67x
int **_gmpy4 (**int *src1*, int *src2***);**	**GMPY4**	Performs the galois field multiply on 4 values in src1 with 4 parallel values in src2. The 4 products are packed into the return value.	’C64x
uint **_hi(**double**);**		Returns the high 32 bits of a double as an integer.	’C67x, ’C64x
double **_itod(**uint, uint**);**		Creates a new double register pair from two unsigned integers.	’C67x, ’C64x
float **_itof(**uint**);**		Reinterprets the bits in the unsigned integer as a float. (Ex: _itof(0x3f800000) == 1.0)	’C67x

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
double & **_memd8(**void * ptr**);**	**LDNDW/** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory.	’C64x
int & **_mem4(**void * ptr**);**	**LDNW/** **STNW**	Allows unaligned loads and stores of 4 bytes to memory.	’C64x

long **_ldotp2 (**int *src1*, int *src2***);**
int **_dotp2 (**int *src1*, int *src2***);**
	LDOTP2
	DOTP2
		The product of signed 16–bit values in src1 is added to the product of signed 16–bit values in src2.	’C64x

uint **_lmbd(**uint s*rc1,* uint *src2***);**	**LMBD**	Searches for a leftmost 1 or 0 of src2 determined by the LSB of src1. Returns the number of bits up to the bit change.	
uint **_lo(**double**);**		Returns the low (even) register of a double register pair as an integer.	’C67x, ’C64x

int **_max2 (**int *src1*, int *src2***);**
int **_min2 (**int *src1*, int *src2***);**
unsigned **_maxu4 (**unsigned *src1*, unsigned *src2***);**
unsigned **_minu4 (**unsigned *src1*, unsigned *src2***);**
	MAX2
	MIN2
	MAXU4
	MINU4
		Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16–bit signed or 8–bit unsigned.	’C64x

double **_mpy2 (**int *src1*, int *src2***);**	**MPY2**	Returns the products of the lower and higher 16–bit values in src1 and src2.	’C64x

double **_mpyhi (**int *src1*, int *src2***);**
double **_mpyli (**int *src1*, int *src2***);**
	MPYHI
	MPYLI
		Produces a 16 by 32 multiply. The result is placed into the lower 48 bits of the returned double. Can use the upper or lower 16 bits of src1.	’C64x

int **_mpyhir (**int *src1*, int *src2***);**
int**_mpylir (**int *src1*, int *src2***);**
	MPYHIR
	MPYLIR
		Produces a signed 16 by 32 multiply. The result is shifted right by 15 bits. Can use the upper or lower 16 bits of src1.	’C64x

double **_mpysu4 (**int *src1*, unsigned *src2***);**
double **_mpyu4 (**unsigned *src1*, unsigned *src2***);**
	MPYSU4
	MPYU4
		For each 8–bit quantity in src1 and src2, performs an 8–bit by 8–bit multiply. The four 16–bit results are packed into a double. The results can be signed or unsigned.	’C64x

int **_mpy(**int *src1,* int *src2***);**
int **_mpyus(**uint *src1,* int *src2***);**
int **_mpysu(**int *src1,* uint *src2***);**
uint **_mpyu(**uint *src1,* uint *src2***);**
	MPY
	MPYUS
	MPYSU
	MPYU
		Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device

int **_mpyh(**int *src1*, int *src2***);**
int **_mpyhus(**uint *src1*, int *src2***);**
int **_mpyhsu(**int *src1*, uint *src2***);**
uint **_mpyhu(**uint *src1*, uint *src2***);**
	MPYH
	MPYHUS
	MPYHSU
	MPYHU
		Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	

int **_mpyhl(**int *src1*, int *src2***);**
int **_mpyhuls(**uint *src1*, int *src2***);**
int **_mpyhslu(**int *src1*, uint *src2***);**
uint **_mpyhlu(**uint *src1*, uint *src2***);**
	MPYHL
	MPYHULS
	MPYHSLU
	MPYHLU
		Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	

int **_mpylh(**int *src1*, int *src2***);**
int **_mpyluhs(**uint *src1*, int *src2***);**
int **_mpylshu(**int *src1*, uint *src2***);**
uint **_mpylhu(**uint *src1*, uint *src2***);**
	MPYLH
	MPYLUHS
	MPYLSHU
	MPYLHU
		Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	

int **_mvd (**int src**);**	**MVD**	Moves the data from the src to the return value over 4 cycles using the multipler pipeline.	’C64x
void **_nassert(**int**);**		Generates no code. Tells the optimizer that the expression declared with the assert function is true. This gives a hint to the compiler as to what optimizations might be valid (trip count information for software pipelined loops and about using word-wide optimizations).	

uint **_norm(**int *src2***);**
uint **_lnorm(**long *src2***);**
	NORM	Returns the number of bits up to the first nonredundant sign bit of src2.	

unsigned **_pack2 (**unsigned *src1*, unsigned *src2***);**
unsigned **_packh2 (**unsigned *src1*, unsigned *src2***);**
	PACK2
	PACKH2
		The lower/upper half–words of src1 and src2 are placed in the return value.	’C64x

unsigned **_packh4 (**unsigned *src1*, unsigned *src2***);**
unsigned **_packl4 (**unsigned *src1*, unsigned *src2***);**
	PACKH4
	PACKL4
		Packs alternate bytes into return value. Can pack high or low bytes.	’C64x

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device

unsigned **_packhl2 (**unsigned *src1*, unsigned *src2***);**
unsigned **_packlh2 (**unsigned *src1*, unsigned *src2***);**
	PACKHL2
	PACKLH2
		The upper/lower half–word of src1 is placed in the upper half–word the return value. The lower/upper half–word of src2 is placed in the lower half–word the return value.	’C64x

double _**rcpdp(**double**)**;	**RCPDP**	Computes the approximate 64-bit double reciprocal.	’C67x
float _**rcpsp(**float**)**;	**RCPSP**	Computes the approximate 64-bit double reciprocal.	’C67x
unsigned **_rotl (**unsigned *src1*, unsigned *src2***);**	**ROTL**	Rotates src2 to the left by the amount in src1.	’C64x
double _**rsqrdp(**double *src***)**;	**RSQRDP**	Computes the approximate 64-bit double reciprocal square root.	’C67x
float _**rsqrsp(**float *src***)**;	**RSQRSP**	Computes the approximate 32-bit float reciprocal square root.	’C67x

int _**sadd(**int *src1,* int *src2***);**
long _**lsadd(**int *src1,* long *src2***):**
	SADD	Adds src1 to src2 and saturates the result. Returns the result.	

unsigned **_saddu4 (**unsigned *src1*, unsigned *src2***);**	**SADDU4**	Performs saturated addition between pairs of 8-bit unsigned values in src1 and src2.	’C64x

int **_sadd2 (**int *src1*, int *src2***);**
int **_saddus2 (**unsigned *src1*, int *src2***);**
	SADD2
	SADDUS2
		Performs saturated addition between pairs of 16–bit values in src1 and src2. Src1 values can be signed or unsigned.	’C64x

int _**sat(**long *src2***);**	**SAT**	Converts a 40-bit value to an 32-bit value and saturates if necessary.	
uint _**set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csta and cstb, respectively.	
unsigned _**setr(**unsigned, int**);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of the source register.	

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device
unsigned **_shfl (**unsigned src**);**	**SHFL**	The lower 16 bits of src are placed in the even bit positions, and the upper 16 bits of src are placed in the odd bit positions.	’C64x

unsigned **_shlmb (**unsigned *src1*, unsigned *src2***);**
unsigned **_shrmb (**unsigned *src1*, unsigned *src2***);**
	SHLMB
	SHRMB
		Shifts src2 left/right by one byte, and the most/least significant byte of src1 is merged into the least/most significant byte position.	’C64x

int **_shr2 (**int *src1*, unsigned *src2***);**
unsigned **_shru2 (**unsigned *src1*, unsigned *src2***);**
	SHR2
	SHRU2
		For each 16-bit quantity in src2, the quantity is arithmetically or logically shifted right by src1 number of bits. src2 can contain signed or unsigned values.	’C64x

int **_smpy(**int *src1*, int *sr2***);**
int **_smpyh(**int *src1*, int *sr2***);**
int **_smpyhl(**int *src1*, int *sr2***);**
int **_smpylh(**int *src1*, int *sr2***);**
	SMPY
	SMPYH
	SMPYHL
	SMPYLH
		Multiplies src1 by src2, left shifts the result by one, and returns the result. If the result is 0x8000 0000, saturates the result to 0x7FFF FFFF.	

double **_smpy2 (**int *src1*, int *src2***);**	**SMPY2**	Performs 16-bit multiplication between pairs of signed packed 16-bit values, with an additional 1 bit left–shift and saturate into a double result.	’C64x
int **_spack2 (**int *src1*, int *src2***);**	**SPACK2**	Two signed 32–bit values are saturated to 16–bit values and packed into the return value.	’C64x
unsigned **_spacku4 (**int *src1*, int *src2***);**	**SPACKU4**	Four signed 16–bit values are saturated to 8–bit values and packed into the return value.	’C64x
int **_spint(**float**)**;	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register.	’C67x

int **_sshvl (**int *src1*, int *src2***);**
int **_sshvr (**int *src1*, int *src2***);**
	SSHVL
	SSHVR
		Shifts src2 to the left/right of src1 bits. Saturates the result if the shifted value is greater than MAX_INT or less than MIN_INT	’C64x

uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.	

Note: Instructions not specified with a device apply to all ’C6000 devices.

Table 3–5. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device

int **_ssub(**int *src1,* int *src2***);**
long **_lssub(**int *src1,* long *src2***):**
	SSUB	Subtracts src2 from src1, saturates the result size, and returns the result.	

uint **_subc(**uint *src1,* uint *src2***);**	**SUBC**	Conditional subtract divide step.	
int **_sub2(**int *src1*, int *src2***);**	**SUB2**	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1, and returns the result. Any borrowing from the lower half subtract does not affect the upper half subtract.	
int **_sub4 (**int *src1*, int *src2***);**	**SUB4**	Performs 2s–complement subtraction between pairs of packed 8–bit values.	’C64x
int **_subabs4 (**int *src1*, int *src2***);**	**SUBABS4**	Calculates the absolute value of the differences for each pair of packed 8–bit values.	’C64x
unsigned **_swap4 (**unsigned src**);**	**SWAP4**	Exchanges pairs of bytes (an endian swap) within each 16–bit value.	’C64x
unsigned **_unpkhu4 (**unsigned src**);**	**UNPKHU4**	Unpacks the two high unsigned 8–bit values into unsigned packed 16–bit values.	’C64x
unsigned **_unpklu4 (**unsigned src**);**	**UNPKLU4**	Unpacks the two low unsigned 8–bit values into unsigned packed 16–bit values.	’C64x
unsigned **_xpnd2 (**unsigned src**);**	**XPND2**	Bits 1 and 0 of src are replicated to the upper and lower halfwords of the result, respectively.	’C64x
unsigned **_xpnd4 (**unsigned src**);**	**XPND4**	Bits 3 through 0 are replicated to bytes 3 through 0 of the result.	’C64x

Note: Instructions not specified with a device apply to all ’C6000 devices.

3.4.2 Using Word Access for Short Data

The ’C6000 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 3–10) doubles the performance of the loop. See section 6.4, *Loading Two Data Values with LDW*, on page 6-19 for more information. This type of optimization is called SIMD (Single Instruction Multiple Data).

Example 3–10. 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;

  #pragma MUST_ITERATE (10);

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

Note:

The MUST_ITERATE intrinsic tells the compiler that the following loop will iterate at least the specified number of times.

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);
```

On the ’C64x, nonaligned accesses to memory are allowed in C through the _mem4 and _memd8 intrinsics.

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 3–11 provides a generic vecsum( ) function that handles all types of alignments and array sizes.

Example 3–11. Vector Sum With const Keywords, MUST_ITERATE pragma, and Word Reads (Generic Version)

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

     /* test to see if sum, in2, and in1 are aligned to a word boundary */

  if (((int)sum | (int)in2 | (int)in1) & 0x2)
  {
    #pragma MUST_ITERATE (20);
    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;

#pragma MUST_ITERATE (10);
    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];
  }
}
```

3.4.2.1 Using Word Access in Dot Product

Other intrinsics that are useful for reading short data as words are the multiply intrinsics. Example 3–12 is a dot product example that reads word-aligned short data and uses the _mpy( ) and _mpyh( ) intrinsics. The _mpyh( ) intrinsic uses the ’C6000 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 3–12. 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;
}
```

3.4.2.2 Using Word Access in FIR Filter

Example 3–13 shows an FIR filter that can be optimized with word reads of short data and multiply intrinsics.

Example 3–13. 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 3–14 shows an optimized version of Example 3–13. 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 3–14. 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);

  #pragma MUST_ITERATE (8);

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

    #pragma MUST_ITERATE (8);
    for (i = 0; i < (n >> 1); i++)
    {
      y0 += _mpy (x[i + j],   h[i]);
      y0 += _mpyh (x[i + j],   h[i]);
      y1 += _mpyhl(x[i + j],   h[i]);
      y1 += _mpylh(x[i + j + 1], h[i]);
    }

    *y++ = (int)(y0 >> s);
    *y++ = (int)(y1 >> s);
  }
}
 short x[SIZE_X], h[SIZE_H], y[SIZE_Y]; void main(){
     fir1((int *)x, (int *)h, y, n,m, s;
}
```

3.4.2.3 Using Double Word Access for Word Data (’C64x and ’C67x Specific)

The ’C64x and ’C67x families have 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 3–15. 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 3–15. 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 3–16, 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() instrinsics return integer values, the _itof() intrinsic subverts the C typing system by interpreting an integer value as a float value. In this version of the loop, 2 float results are computed every 4 cycles. 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 3–16 and Example 3–17 show these pragmas.

Example 3–16. 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 3–17, 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 3–17. 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 */
/* the C62xx, C64xx, 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);
}
```

3.4.2.4 Using _nassert(), Word Accesses, and the MUST_ITERATE pragma

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 ‘C6000 intrinsics.

Example 3–18 shows how the compiler can automatically do this optimization.

Example 3–18. 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);
    #pragma MUST_ITERATE (40, 40);
    for (i = 0; i < N; i++)
    sum += a[i] * b[i];
    return sum;
}
```

Compile Example 3–18 with the following options: –o -k. Open up the assembly file and look at the loop kernel. The results are the exact same as those produced by Example 3–12. The first 2 _nassert() intrinsics in Example 3–18 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 3–12 (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 MUST_ITERATE pragma conveys loop count information to the compiler. For example, #pragma MUST_ITERATE (40, 40), tells the compiler the loop immediately following this pragma will execute a minimum of 40 times (the first argument), and a maximum of 40 times (the second argument). An optional third argument tells the compiler what the trip count is a multiple of. See the *TMS320C6000 C/C++ Compiler User's Guide* for more information about the MUST_ITERATE pragma.

Example 3–19 and Example 3–20 show how to use the _nassert() intrinsic and MUST_ITERATE pragma to get word accesses on the vector sum and the FIR filter.

Example 3–19. 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);
      #pragma MUST_ITERATE (40, 40);
      for (i = 0; i < N; i++)
         sum[i] = in1[i] + in2[i];
   }
```

Example 3–20. 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);
   for (j = 0; j < m; j++)
   {
      y0 = round;
      #pragma MUST_ITERATE (40, 40);
      for (i = 0; i < n; i++)
         y0 += x[i + j] * h[i];
      y[j] = (int)(y0 >> s);
   }
}
```

As you can see from Example 3–20, the optimization done by the compiler is not as optimal as the code produced in Example 3–14, but it is more optimal than the code in Example 3–13.

Example 3–21. Compiler Output From Example 3–20

```
L3:    ; PIPED LOOP KERNEL

   [!B0]   ADD     .L1     A9,A7:A6,A7:A6    ; |21|
||         MPY     .M2X    A3,B3,B2          ; |21|
||         MPYHL   .M1X    B3,A0,A0          ; |21|
|| [ A1]   B       .S2     L3                ; @|21|
||         LDH     .D2T2   *++B9(8),B3       ; @@|21|
||         LDH     .D1T1   *+A8(4),A3        ; @@|21|

   [!B0]   ADD     .L2     B3,B5:B4,B5:B4    ; |21|
||         MPY     .M1X    A0,B1,A9          ; @|21|
||         LDW     .D2T2   *+B8(4),B3        ; @@|21|
||         LDH     .D1T1   *+A8(6),A0        ; @@|21|

   [ B0]   SUB     .S2     B0,1,B0           ;
|| [!B0]   ADD     .L2     B2,B7:B6,B7:B6    ; |21|
|| [!B0]   ADD     .L1     A0,A5:A4,A5:A4    ; |21|
||         MPYHL   .M2     B1,B3,B3          ; @|21|
|| [ A1]   SUB     .S1     A1,1,A1           ; @@|21|
||         LDW     .D2T2   *++B8(8),B1       ; @@@|21|
||         LDH     .D1T1   *++A8(8),A0       ; @@@|21|
```

Example 3–22. Compiler Output From Example 3–14

```
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
```

Example 3–23. Compiler Output From Example 3–13

```
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.

If your code operates on global arrays as in Example 3–24, 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 3–24. 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 file1.c file2.c
```

Below is the resulting assembly file (file1.asm). Notice that the dot product loop uses word accesses and the ‘C6000 intrinsics.

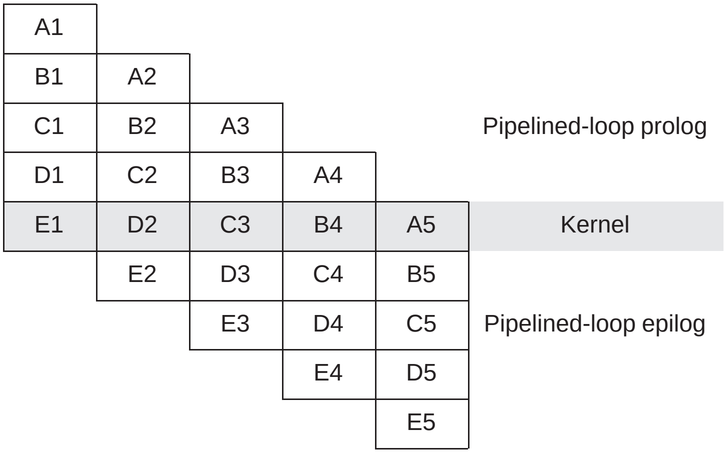
```
L2:    ; PIPED LOOP KERNEL
   [!A1]   ADD     .L2     B6,B7,B7
|| [!A1]   ADD     .L1     A6,A0,A0
||         MPY     .M2X    B5,A4,B6
||         MPYH    .M1X    B5,A4,A6
|| [ B0]   B       .S1     L2
||         LDW     .D1T1   *+A5(4),A4
||         LDW     .D2T2   *+B4(4),B6

   [ 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
```

3.4.3 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 3–3 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 3–3. Software-Pipelined Loop



A1					
B1	A2				
C1	B2	A3			Pipelined-loop prolog
D1	C2	B3	A4		
E1	D2	C3	B4	A5	Kernel
	E2	D3	C4	B5	
		E3	D4	C5	Pipelined-loop epilog
			E4	D5	
				E5	

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

3.4.3.1 Trip Count Issues

A trip count is the number of loop iterations executed. The trip counter is the variable used to count each iteration. When the trip count reaches a limit equal to the trip count, the loop terminates.

If the compiler can guarantee that at least *n* loop iterations will be executed, then *n* is the known minimum trip count. Sometimes the compiler can determine this information automatically. Alternatively, the user can provide this information using the MUST_ITERATE and PROB_ITERATE pragma. For more information about pragmas, see the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* (SPRU187).

The minimum safe trip count is the number of iterations of the loop that are necessary to safely execute the software pipelined version of the loop.

All software pipelined loops have a minimum safe trip count requirement. If the known minimum trip count is not above the minimum safe trip count, redundant loops will be generated.

The known minimum trip count and the minimum safe trip count for a given software pipelined loop can be found in the compiler-generated comment block for that loop.

In general, loops that can be most efficiently software pipelined 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 3–25(a) to something like the code in Example 3–25(b).

Example 3–25. 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 */
```

3.4.3.2 Eliminating Redundant Loops

Sometimes the compiler cannot determine if the loop always executes more than the minimum safe trip count. Therefore, the compiler will generate two versions of the loop:

❑ An unpipelined version that executes if the trip count is less than the minimum safe trip count.

❑ A software-pipelined version that executes if the trip count is equal to or greater than the minimum safe trip count.

Obviously, the need for redundant loops will hurt both codesize and to a lesser extent, performance.

You can use the -mh option, when safe, to reduce the need for a redundant loop.

To indicate to the compiler that you do not want two versions of the loop, you can use the -ms0 or -ms1 option. The compiler will generate the software pipelined version of the loop only if it can prove the minumum trip count will always be equal or greater than the effective minimum trip count of the software pipelined version of the loop. Otherwise, the non pipelined version will be generated. In order to help the compiler generate only the software pipelined version of the loop, use the MUST_ITERATE pragma and/or the -pm option to help the compiler determine the known minimum trip count.

Note: Use of -ms0 or -ms1 may result in a performance degredation

Using -ms0 or -ms1 may cause the compiler not to software pipeline a loop. This can cause the performance of the loop to suffer.

The compiler performs an optimization called prolog/epilog collapsing to reduce code size of pipelined loops. In particular, this optimization involves rolling the prolog and/or epilog (or parts thereof) back into the kernel. This can result in a major code size reduction. This optimization can also reduce the minimum trip count needed to safely execute the software-pipelined loop, thereby eliminating the need for redundant loops in many cases.

The user can increase the compiler's ability to perform this optimization by using the -mh, or -mhn option whenever possible. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information about options.

3.4.3.3 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 MUST_ITERATE pragma 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 MUST_ITERATE and PROB_ITERATE pragma to convey many different types of information about the trip count to the compiler.

❑ The MUST_ITERATE pragma can convey that the trip count will always equal some value.

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

❑ The MUST_ITERATE pragma 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 7.4, *Interruptible Loops*.

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

❑ The MUST_ITERATE pragma can convey that the trip count is always divisible by a value.

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

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

```
   #pragma MUST_ITERATE (8, 48, 8);
   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 7.4.4 show all of the different ways that the MUST_ITERATE pragma and _nassert intrinsic can be used.

The _nassert intrinsic 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);
   . . .
}
```

See the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for a complete discussion of the –ms, –o3, and –pm options, the _nassert intrinsic, and the MUST_ITERATE and PROB_ITERATE pragmas.

3.4.3.4 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 ’C6000 architecture.

There are three ways loop unrolling can be performed:

1) The compiler may automatically unroll the loop.

2) You can suggest that the compiler unroll the loop using the UNROLL pragma.

3) You can Unroll the C/C++ code yourself

In Example 3–26, 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 3–26. 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 3–27), 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 3–27. 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;

  #pragma MUST_ITERATE (10);

  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 3–28 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 3–28 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 3–28. Vector Sum Using const Keywords, MUST_ITERATE pragma, Word Reads, and Loop Unrolling

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

  #pragma MUST_ITERATE (10);

  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 3–29, 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 3–29. 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 3–30.

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 6.7, *Loop Carry Paths* and section 3.2 in the *TMS320C6000 Optimizing C/C++ Compiler User's Guide*. This information can be obtained by using the option and looking at the comment block before the loop.

Example 3–30. 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 first requirement can be communicated using the MUST_ITERATE pragma. The second requirement can also be passed to the compiler through the MUST_ITERATE pragma. In section 3.4.3.3, *Communicating Trip-Count Information to the Compiler*, it is explained that the MUST_ITERATE pragma can be used to provide information about loop unrolling. By using the third argument, you can specify that the trip count is a multiple or power of two.

```
#pragma MUST_ITERATE (n,n, 2);
```

Example 3–31 shows how the compiler can perform simple loop unrolling of replicating the loop body. The MUST_ITERATE pragma tells the compiler that the loop will execute an even number of 20 or more times. This compiler will unroll the loop once to take advantage of the performance gain that results from the unrolling.

Example 3–31. Vector Sum

```
void func(short *a, const short *b, const short *c, int n)
{
   int i;
   #pragma MUST_ITERATE (20, , 2);
   for (i = 0; i < n; i++) a[i] = b[i] + c[i];
}
   <compiler output for above code>
L2:    ; PIPED LOOP KERNEL

           ADD     .L1X    B7,A3,A3          ; |5|
|| [ B0]   B       .S1     L2                ; @|5|
||         LDH     .D1T1   *++A4(4),A3       ; @@|5|
||         LDH     .D2T2   *++B4(4),B7       ; @@|5|

   [!A1]   STH     .D1T1   A3,*++A0(4)       ; |5|
||         ADD     .L2X    B6,A5,B6          ; |5|
||         LDH     .D2T2   *+B4(2),B6        ; @@|5|

   [ A1]   SUB     .L1     A1,1,A1           ;
|| [!A1]   STH     .D2T2   B6,*++B5(4)       ; |5|
|| [ B0]   SUB     .L2     B0,1,B0           ; @@|5|
||         LDH     .D1T1   *+A4(2),A5        ; @@|5|
```

Note:

When the interrupt threshold option is used, unrolling can be used to regain lost performance. Refer to section 7.4.4 *Getting the Most Performance Out of Interruptible Code*.

If the compiler does not automatically unroll the loop, you can suggest that the compiler unroll the loop by using the UNROLL pragma. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information.

3.4.3.5 Speculative Execution (–mh option)

The –mh option facilitates the compiler's ability to remove prologs and epilogs. Indirectly, it can reduce register pressure. Use of this option can lead to better code size and better performance. This option may cause a loop to read past the end of an array. Thus, the user assumes responsibility for safety. For a complete discussion of the -mh option, including how to use it safely, see the *TMS320C6000 Optimizing C/C++ Compiler User's Guide*.

3.4.3.6 What Disqualifies a Loop from Being Software-Pipelined

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

❑ If a register value is live too long, the code is not software-pipelined. See section 6.6.6.2, *Live Too Long*, on page 6-67 and section 6.10, *Live-Too-Long Issues*, on page 6-101 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 'C6000 condition registers on the 'C62x and 'C67x, or six condition registers for the 'C64x, the loop is not software pipelined. Try to eliminate or combine these conditions.

❑ Although a software pipelined loop can contain intrinsics, it cannot contain function calls, including code that will call the run-time support routines. The exception is function calls that will be inlined.

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

This will call the run-time support _remi routine.

❑ In general, you should not have a conditional break (early exit) in the loop. You may need to rewrite your code to use if statements instead. In some, but not all cases, the compiler can do this automatically. 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.

- In the loop in Example 3–32, there is an early exit. If dist0 or dist1 is less than distance, then execution breaks out of the loop early. If the compiler could not perform transformations to the loop to software pipeline the loop, you would have to modify the code. Example 3–33 shows how the code would be modified so the compiler could software pipeline this loop. In this case however, the compiler can actually perform some transformations and software pipeline this loop better than it can the modified code in Example 3–33.

Example 3–32. Use of If Statements in Float Collision Detection (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;
   }
```

Example 3–33. Use of If Statements in Float Collision Detection (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 downcounting 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 ‘C62x and ’C67x, or 64 registers on the ’C64x, it is not software pipelined. Either try to simplify the loop or break the loop up into multiple smaller loops.

Chapter 4

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**

4.1 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 *TMS320C6000 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 0FFFEC10!     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 –l 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 *TMS320C6000 Optimizing C/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.

4.1.1 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.

4.2 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 4–1. 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 C62x and C64x. 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.

4.2.1 How to Compile

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

Table 4–2. 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.

4.2.2 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.

4.2.3 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 4.2.7

4.2.4 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
    .sysmem     >       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                                    */
    /*---------------------------------------------------------------*/
    .rtstext   { –lrts6200.lib(.text)   } > EXT0

    /*---------------------------------------------------------------*/
    /* RTS data – undefined sections – placed off chip               */
    /*---------------------------------------------------------------*/
    .rtsbss    { –lrts6200.lib(.bss)
                 –lrts6200.lib(.far)    } > EXT0

    /*---------------------------------------------------------------*/
    /* RTS data – defined sections – placed off chip                 */
    /*---------------------------------------------------------------*/
    .rtsdata   { –lrts6200.lib(.const)
                 –lrts6200.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, .sysmem, 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 .sysmem (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 rts6200.lib with the library you normally use, perhaps rts6700.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: rts6200.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.

4.2.5 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 C6000 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.

4.2.6 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 4–3. How _FAR_RTS is Defined in Linkage.h With –mr

Option	Internal RTS calls	Normal RTS calls	_FAR_RTS
Default	Same as user	Same as user	Undefined
–mr0	Near	Near	0
–mr1	Far	Far	1

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

```
The _IDECL macro determines how inline functions are de-
clared.
```

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 _ctypes_[];
```

4.2.7 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.

Chapter 5

Structure of Assembly Code

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

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

Topic **Page**

5.1 Labels

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

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

Figure 5–1. Labels in Assembly Code

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

Labels must meet the following conditions:

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

5.2 Parallel Bars

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

Figure 5–2. Parallel Bars in Assembly Code

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

5.3 Conditions

Five registers in the ’C6000 are available for conditions: A1, A2, B0, B1, and B2. Figure 5–3 shows the position of a condition in a line of assembly code.

Figure 5–3. Conditions in Assembly Code

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

All ’C6000 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

5.4 Instructions

Assembly code instructions are either directives or mnemonics:

- *Assembler directives* are commands for the assembler (asm6x) that control the assembly process or define the data structures (constants and variables) in the assembly language program. All assembler directives begin with a period, as shown in the partial list in Table 5–1. See the *TMS320C6000 Assembly Language Tools User's Guide* for a complete list of directives.

- *Processor mnemonics* are the actual microprocessor instructions that execute at runtime and perform the operations in the program. Processor mnemonics must begin in column 2 or greater. For more information about processor mnemonics, see the *TMS320C6000 CPU and Instruction Set User's Guide*.

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

Figure 5–4. Instructions in Assembly Code

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

Table 5–1. Selected TMS320C6x Directives

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

See the *TMS320C6000 Assembly Language Tools User's Guide* for a complete list of directives.

5.5 Functional Units

The ’C6000 CPU contains eight functional units, which are shown in Figure 5–5 and described in Table 5–2.

Figure 5–5. TMS320C6x Functional Units

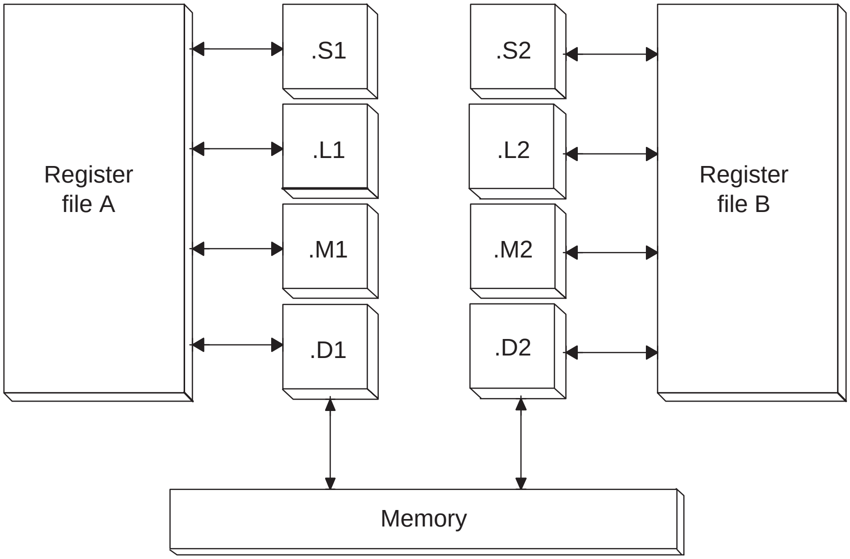


Table 5–2. Functional Units and Operations Performed

Functional Unit	**Fixed–Point Operations**	**Floating–Point Operations**
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits **Byte shifts** **Data packing/unpacking** **5-bit constant generation** **Dual 16-bit arithmetic operations** **Quad 8-bit arithmetic operations** **Dual 16-bit min/max operations** **Quad 8-bit min/max operations**	Arithmetic operations DP → SP, INT → DP, INT → SP conversion operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) **Byte shifts** **Data packing/unpacking** **Dual 16-bit compare operations** **Quad 8-bit compare operations** **Dual 16-bit shift operations** **Dual 16-bit saturated arithmetic operations** **Quad 8-bit saturated arithmetic operations**	Compare Reciprocal and reciprocal square–root operations Absolute value operations SP → DP conversion operations

Table 5–2. Functional Units and Operations Performed (Continued)

Functional Unit	Fixed–Point Operations	Floating–Point Operations
.M unit (.M1, .M2)	16 x 16 multiply operations **16 x 32 multiply operations** **Quad 8 x 8 multiply operations** **Dual 16 x 16 multiply operations** **Dual 16 x 16 multiply with add/subtract operations** **Quad 8 x 8 multiply with add operation** **Bit expansion** **Bit interleaving/de-interleaving** **Variable shift operations** **Rotation** **Galois Field Multiply**	32 X 32–bit fixed–point multiply operations Floating–point multiply operations
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only) **Load and store double words with 5-bit constant** **Load and store non-aligned words and double words** **5-bit constant generation** **32-bit logical operations**	Load doubleword with 5–bit constant offset

Note: Fixed-point operations are available on all three devices. Floating-point operations and 32 x 32-bit fixed-point multiply are available only on the ’C67x. Additonal ’C64x functions are shown in bold.

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

Figure 5–6. Units in the Assembly Code

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

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

5.6 Operands

The ’C6000 architecture requires that memory reads and writes move data between memory and a register. Figure 5–7 shows the position of the operands in a line of assembly code.

Figure 5–7. Operands in the Assembly Code

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

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

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

  When an operand comes from the other register file, the unit includes an X, as shown in Figure 5–8, indicating that the instruction is using one of the cross paths. (See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information on cross paths.)

Figure 5–8. Operands in Instructions

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

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

The ’C6000 instructions use three types of operands to access data:

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

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

5.7 Comments

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

Figure 5–9. Comments in Assembly Code

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

The following are guidelines for using comments in assembly code:

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

Chapter 6

Optimizing Assembly Code via Linear Assembly

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

This chapter encompasses phase 3 of the code development flow. After you have developed and optimized your C code using the ’C6000 compiler, extract the inefficient areas from your C code and rewrite them in linear assembly (assembly code that has not been register-allocated and is unscheduled).

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

Topic **Page**

6.1 Assembly Code

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 ’C6000 to specify the functional unit or register used, this may restrict the compiler’s ability to fully optimize your code. See the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for more information.

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

- ❑ Section 6.3 and section 6.4 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 6.5 and section 6.6 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 6.7 uses an IIR filter algorithm to discuss the problems with loop carry paths.

- ❑ Section 6.8 and section 6.9 discuss the problems encountered with if-then-else statements in a loop and how loop unrolling can be used to resolve them.

- ❑ Section 6.10 introduces live-too-long issues in your code.

- ❑ Section 6.11 uses a simple FIR filter algorithm to discuss redundant load elimination.

- ❑ Section 6.12 discusses the same FIR filter in terms of the interleaved memory bank scheme used by ’C6000 devices.

- ❑ Section 6.13 and section 6.14 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 ’C6000: C/C++ code (which is input for the C/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 6.2, 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 6.2 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.

6.2 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 *TMS320C6000 Optimizing C/C++ Compiler User's Guide*.

6.2.1 The –0*n* Option

Software pipelining requires the -o2 or -o3 option. Not specifying -o2 or -o3 facilitates faster compile time and ease of development through reduced optimization.

6.2.2 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 6–1. 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 in 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. See section 6.7, Loop Carry Paths, on page 6-77 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 B, Memory Alias Disambiguation. Refer to the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information on both the -mt option and the .no_mdep directive.

6.2.3 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 6–2. Block copy With .mdep

```
.mdep  ld1, st1
LDW *p1++ {ld1}, inp1 ; 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.

6.2.4 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 6–3 shows linear assembly code and the generated loop kernel for a dot product without the .mptr directive.

Example 6–3. 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 6–3. 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 6–4 shows the linear assembly dot product with the .mptr directive and the resulting loop kernel.

Example 6–4. 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 ’C6000 compiler supports pragmas in C/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.

6.2.5 The .trip Directive

The .trip directive is analogous to the _nassert intrinsic for C/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 7.4.4, *Getting the Most Performance Out of Interruptible Code*.

6.3 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.

6.3.1 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 6–5 is a fixed-point dot product algorithm. The C code in Example 6–6 is a floating-point dot product algorithm.

Example 6–5. 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 6–6. 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);
}
```

6.3.2 Translating C Code to Linear Assembly

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

6.3.2.1 Fixed-Point Dot Product

Example 6–7 shows the linear assembly instructions used for the inner loop of the fixed-point dot product C code.

Example 6–7. 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.

6.3.2.2 Floating-Point Dot Product

Example 6–8 shows the linear assembly instructions used for the inner loop of the floating-point dot product C code.

Example 6–8. 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.

6.3.3 Linear Assembly Resource Allocation

The following rules affect the assignment of functional units for Example 6–7 and Example 6–8 (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 6–7 and Example 6–8, they are assigned as listed above.

The ADDSP instruction in Example 6–8 must use a .L unit.

6.3.4 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*.

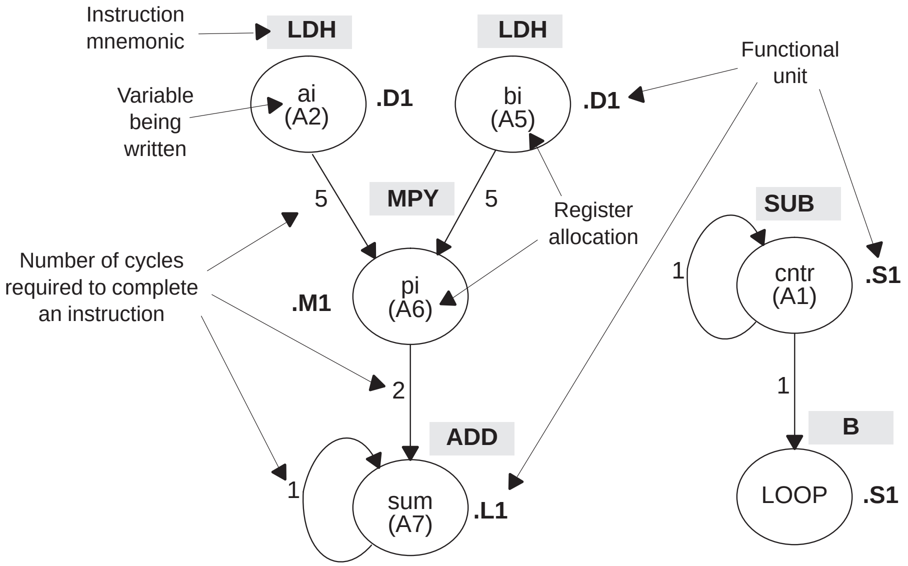
Use the following steps to draw a dependency graph:

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.

6.3.4.1 Fixed-Point Dot Product

Figure 6–1 shows the dependency graph for the fixed-point dot product assembly instructions shown in Example 6–7 and their corresponding register allocations.

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



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

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

- ❑ The ADD instruction adds pi (the result of the MPY) to sum. The output of the ADD instruction feeds back to become an input on the next iteration and, thus, creates a *loop carry* path. (See section 6.7 on page 6-77 for more information on loop carry paths.)

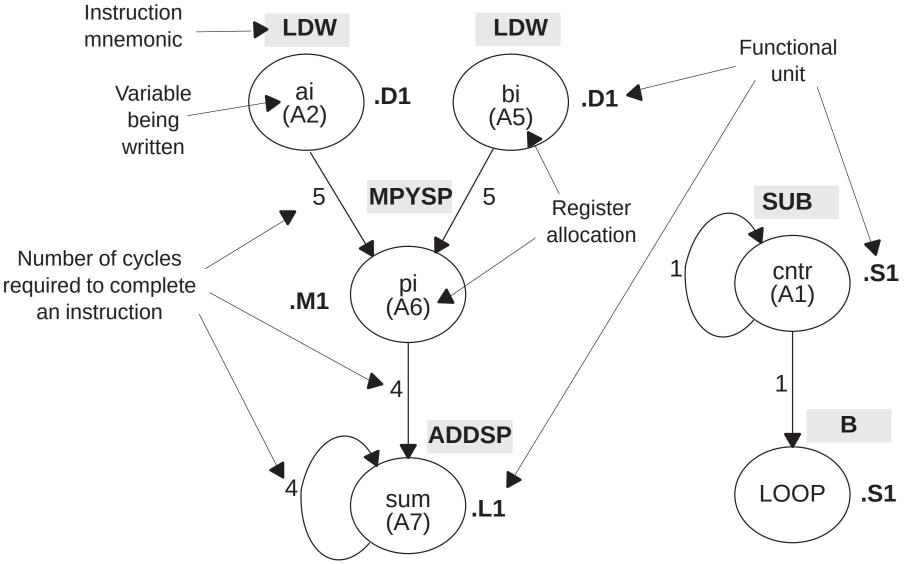
The dependency graph for this dot product algorithm has two separate parts because the decrement of the loop counter and the branch do not read or write any variables from the other part.

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

6.3.4.2 Floating-Point Dot Product

Similarly, Figure 6–2 shows the dependency graph for the floating-point dot product assembly instructions shown in Example 6–8 and their corresponding register allocations.

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



- The two LDW instructions, which write the values of ai and bi, are parents of the MPYSP instruction. It takes five cycles for the parent (LDW) instruction to complete. Therefore, if LDW is scheduled on cycle i, then its child (MPYSP) cannot be scheduled until cycle i + 5.
- The MPYSP instruction, which writes the product pi, is the parent of the ADDSP instruction. The MPYSP instruction takes four cycles to complete.
- The ADDSP instruction adds pi (the result of the MPYSP) to sum. The output of the ADDSP instruction feeds back to become an input on the next iteration and, thus, creates a *loop carry* path. (See section 6.7 on page 6-77 for more information on loop carry paths.)

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

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

6.3.5 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.

6.3.5.1 Fixed-Point Dot Product

Example 6–9 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.

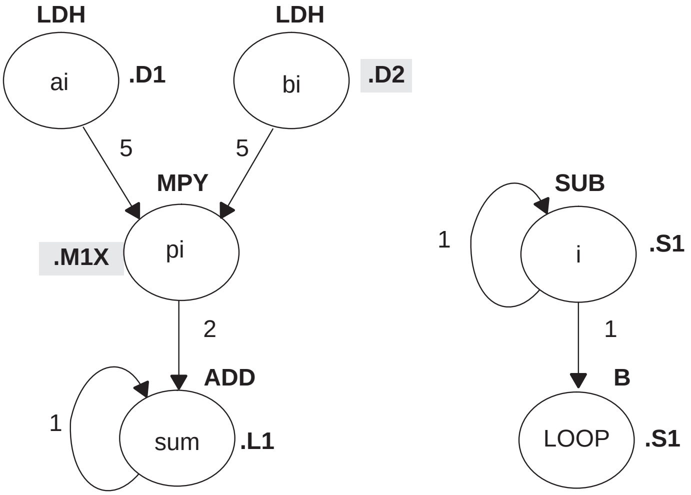
Executing this dot product code serially requires 16 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator; 100 iterations require 1602 cycles.

Example 6–9. 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 6–3. The parallel assembly code is shown in Example 6–10.

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



Example 6–10. 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
```

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

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

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

Rearranging the order of the instructions also improves the performance of the code. The SUB instruction can take the place of one of the NOP delay slots for the LDH instructions. Moving the B instruction after the SUB removes the need for the NOP 5 used at the end of the code in Example 6–9.

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.

6.3.5.2 Floating-Point Dot Product

Similarly, Example 6–11 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.

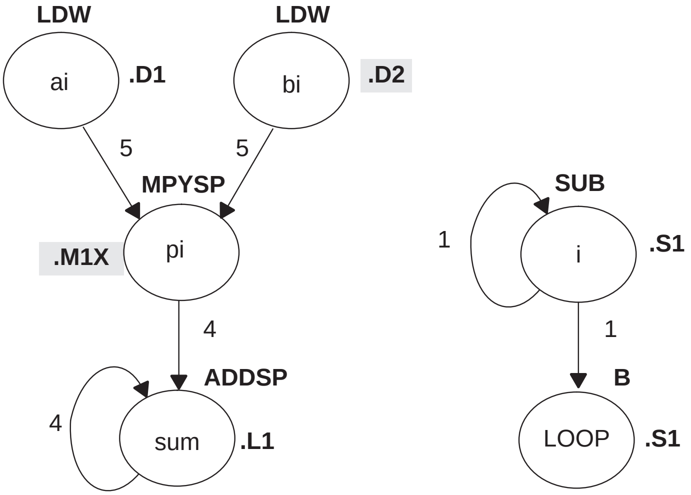
Executing this dot product code serially requires 21 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator; 100 iterations require 2102 cycles.

Example 6–11. 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 6–4. The parallel assembly code is shown in Example 6–12.

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



Example 6–12. 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
```

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

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

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

Rearranging the order of the instructions also improves the performance of the code. The SUB instruction replaces one of the NOP delay slots for the LDW instructions. Moving the B instruction after the SUB removes the need for the NOP 5 used at the end of the code in Example 6–11 on page 6-16.

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

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

6.3.6 Comparing Performance

Executing the fixed-point dot product code in Example 6–10 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 6–1 compares the performance of the nonparallel code with the parallel code for the fixed-point example.

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

Code Example		100 Iterations	Cycle Count
Example 6–9	Fixed-point dot product nonparallel assembly	2 + 100 × 16	1602
Example 6–10	Fixed-point dot product parallel assembly	1 + 100 × 8	801

Executing the floating-point dot product code in Example 6–12 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 6–2 compares the performance of the nonparallel code with the parallel code for the floating-point example.

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

Code Example		100 Iterations	Cycle Count
Example 6–11	Floating-point dot product nonparallel assembly	2 + 100 × 21	2102
Example 6–12	Floating-point dot product parallel assembly	1 + 100 × 10	1001

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

The parallel code for the fixed-point example in section 6.3 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 *TMS320C6000 CPU and Instruction Set User's Guide* for more specific information on the LDDW instruction.

Note:

The load doubleword (LDDW) instruction is available on the 'C64x (fixed point) and 'C67x (floating-point) device.

6.4.1 Unrolled Dot Product C Code

The fixed-point C code in Example 6–13 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 6–14. (For another example of loop unrolling, see section 6.9 on page 6-94.)

Example 6–13. 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 6–14. 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);
}
```

6.4.2 Translating C Code to Linear Assembly

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

6.4.2.1 Fixed-Point Dot Product

Example 6–15 shows the list of ’C6000 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 *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for more information on writing linear assembly code.

Example 6–15. Linear Assembly for Fixed-Point Dot Product Inner Loop with LDW

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

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

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

- The first MPY instruction multiplies the 16 least significant bits (LSBs) in each source register: a[i] × b[i].
- The MPYH instruction multiplies the 16 most significant bits (MSBs) of each source register: a[i+1] × 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 *TMS320C6000 Peripherals Reference Guide* for more information.

6.4.2.2 Floating-Point Dot Product

Example 6–16 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 *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for more information on writing linear assembly code.

Example 6–16. 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.

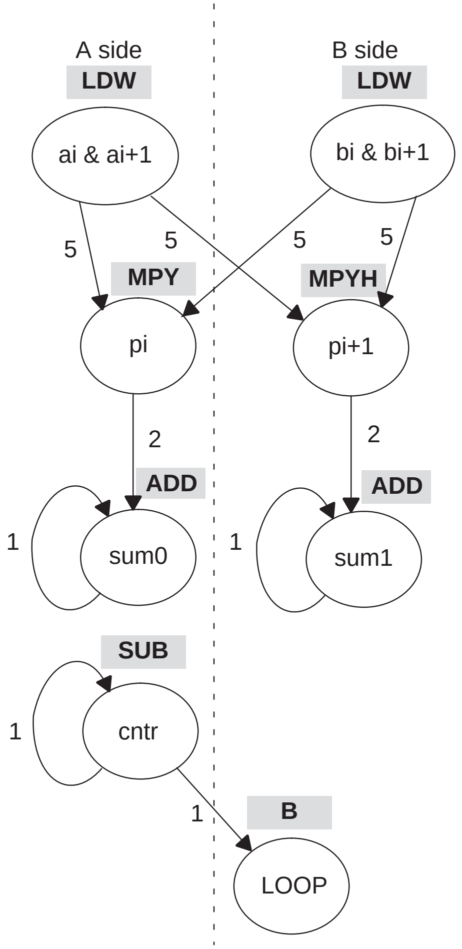
Two MPYSP instructions are now necessary to multiply the second set of array elements.

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

6.4.3 Drawing a Dependency Graph

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

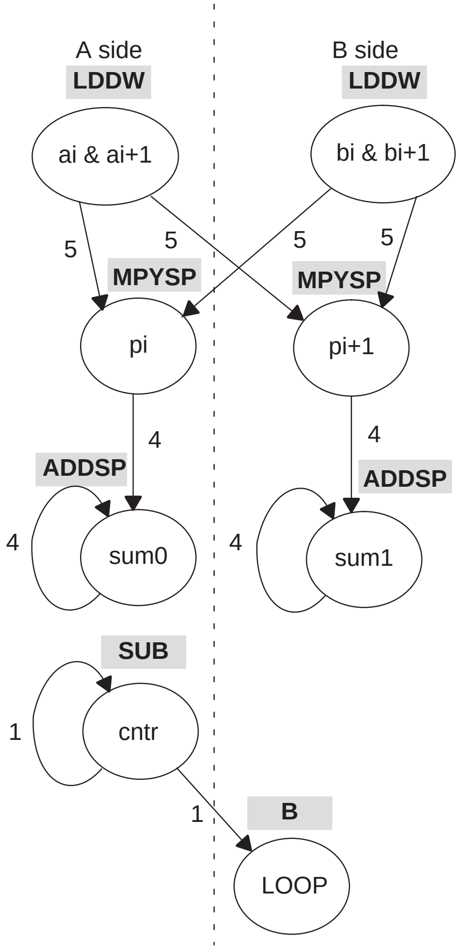
Figure 6–5. Dependency Graph of Fixed-Point Dot Product With LDW



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

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

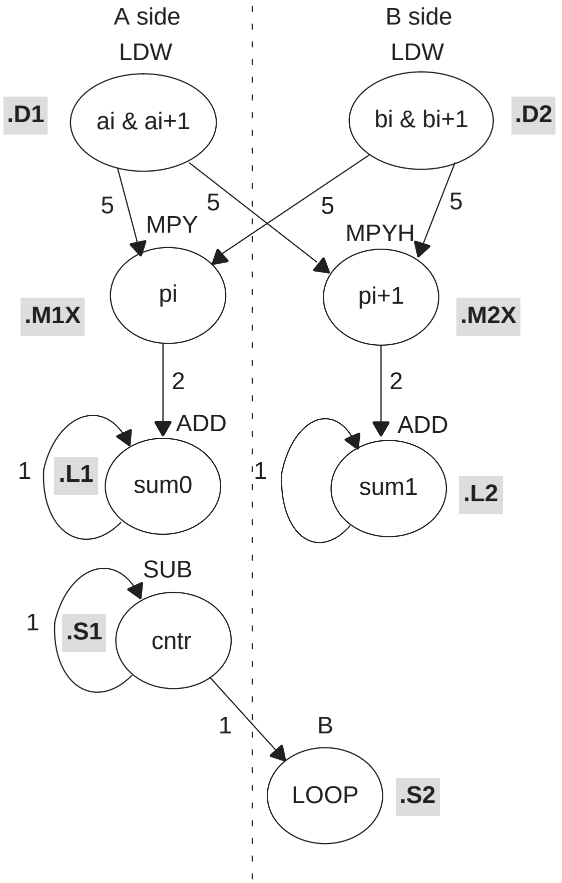
Figure 6–6. Dependency Graph of Floating-Point Dot Product With LDDW



6.4.4 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 6–7 and Figure 6–8 and in the instructions in Example 6–17 and Example 6–18. The .M1X and .M2X represent a path in the dependency graph crossing from one side to the other.

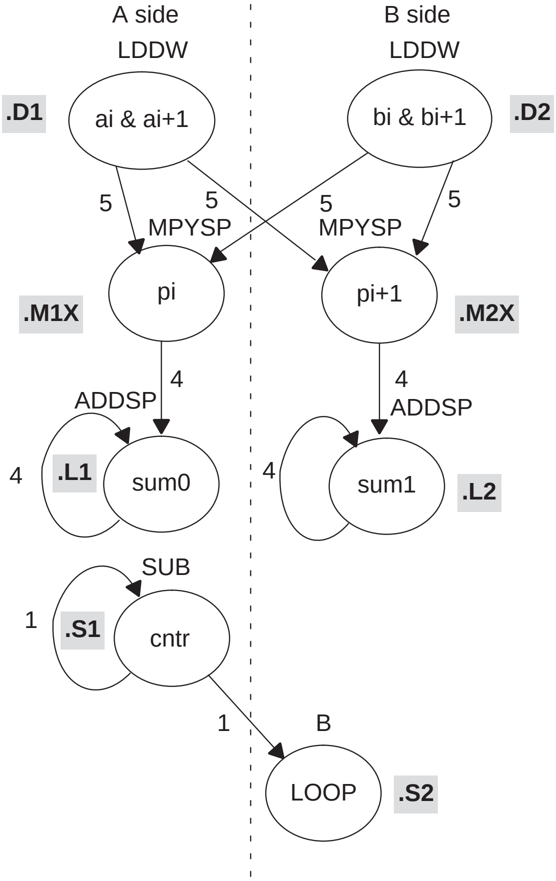
Figure 6–7. Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)



Example 6–17. Linear Assembly for Fixed-Point Dot Product Inner Loop With LDW (With Allocated Resources)

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

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



Example 6–18. 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
```

6.4.5 Final Assembly

Example 6–19 shows the final assembly code for the unrolled loop of the fixed-point dot product and Example 6–20 shows the final assembly code for the unrolled loop of the floating-point dot product.

6.4.5.1 Fixed-Point Dot Product

Example 6–19 uses LDW instructions instead of LDH instructions.

Example 6–19. 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 6–19 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.

6.4.5.2 Floating-Point Dot Product

Example 6–20 uses LDDW instructions instead of LDW instructions.

Example 6–20. 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 6–20 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.

6.4.6 Comparing Performance

Executing the fixed-point dot product with the optimizations in Example 6–19 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 × 50 + 1).

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

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

Code Example		100 Iterations	Cycle Count
Example 6–9	Fixed-point dot product nonparallel assembly	2 + 100 × 16	1602
Example 6–10	Fixed-point dot product parallel assembly	1 + 100 × 8	801
Example 6–19	Fixed-point dot product parallel assembly with LDW	1 + (50× 8)+ 1	402

Executing the floating-point dot product with the optimizations in Example 6–20 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 × 50 + 7).

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

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

Code Example		100 Iterations	Cycle Count
Example 6–11	Floating-point dot product nonparallel assembly	2 + 100 × 21	2102
Example 6–12	Floating-point dot product parallel assembly	1 + 100 × 10	1001
Example 6–20	Floating-point dot product parallel assembly with LDDW	1 + (50× 10)+ 7	508

6.5 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 6–19 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 6–9 shows the dependency graph for the fixed-point dot product instructions. Example 6–21 shows the same dot product assembly code in Example 6–17 on page 6-24, except that the SUB instruction is now conditional on the loop counter (A1).

Note:

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

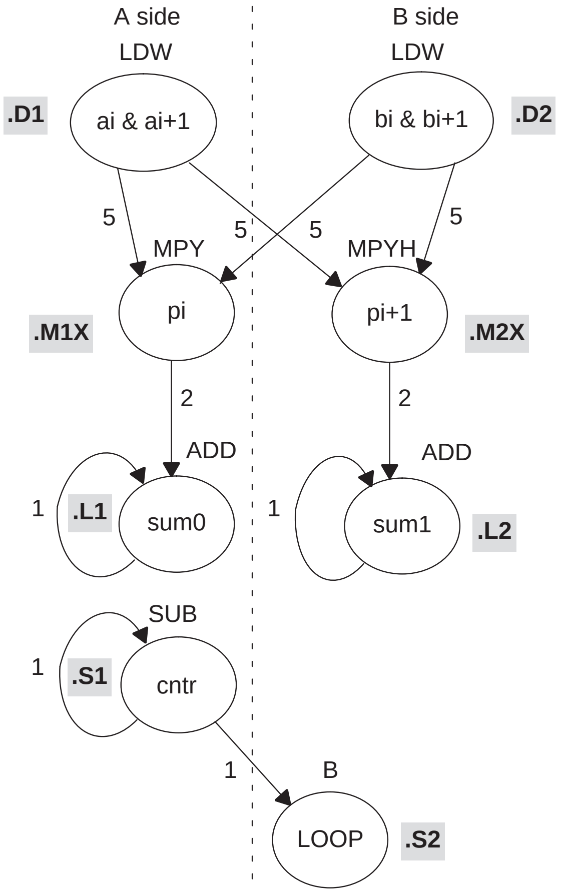
The floating-point dot product code in Example 6–20 needs ten cycles for each iteration of the loop: five cycles for the LDDWs, four cycles for the MPYSPs, and one cycle for the ADDSPs.

Figure 6–10 shows the dependency graph for the floating-point dot product instructions. Example 6–22 shows the same dot product assembly code in Example 6–18 on page 6-25, except that the SUB instruction is now conditional on the loop counter (A1).

Note:

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

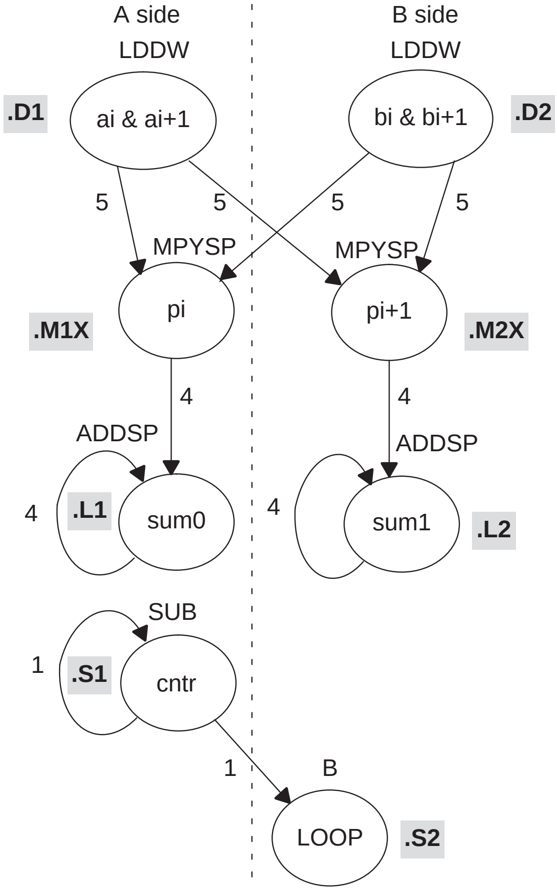
Figure 6–9. Dependency Graph of Fixed-Point Dot Product With LDW (Showing Functional Units)



Example 6–21. 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 6–10. Dependency Graph of Floating-Point Dot Product With LDDW (Showing Functional Units)



Example 6–22. 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
```

6.5.1 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.

6.5.1.1 Fixed-Point Example

The fixed-point code in Example 6–19 needs eight cycles for each iteration of the loop, so the iteration interval is eight.

Table 6–5 shows a modulo iteration interval scheduling table for the fixed-point dot product loop before software pipelining (Example 6–19). 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 6–5. Modulo Iteration Interval Scheduling Table for Fixed-Point Dot Product (Before Software Pipelining)

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

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

6.5.1.2 Floating-Point Example

The floating-point code in Example 6–20 needs ten cycles for each iteration of the loop, so the iteration interval is ten.

Table 6–6 shows a modulo iteration interval scheduling table for the floating-point dot product loop before software pipelining (Example 6–20). 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 6–6. Modulo Iteration Interval Scheduling Table for Floating-Point Dot Product (Before Software Pipelining)

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

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

6.5.1.3 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 6–9 and Figure 6–10, all eight instructions use a different functional unit and no two instructions use the same cross paths (1X and 2X). Because no two instructions use the same resource, the minimum iteration interval based on resources is 1.

Note:

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

6.5.1.4 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 6–7 shows a fully pipelined schedule for the fixed-point dot product example.

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

Loop Prolog

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

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

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

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

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

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

Floating-Point Example

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

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

Loop Prolog

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

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

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

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

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

Note:

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

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

6.5.1.5 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 6–23).

Example 6–23. 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 6–9 shows the results of the loop kernel for a single-cycle accumulator using a multicycle add instruction; in this case, the ADDSP, which has three delay slots (a 4-cycle instruction).

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

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

† where i is a multiple of 4

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

The current value in the accumulator “sum” depends on which iteration is being done. After the completion of the loop, the last four sums should be written into separate registers and then added together to give the final result. This is shown in Example 6–27 on page 6-43.

6.5.2 Using the Assembly Optimizer to Create Optimized Loops

Example 6–24 shows the linear assembly code for the full fixed-point dot product loop. Example 6–25 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 *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information on the assembly optimizer.

Example 6–24. Linear Assembly for Full Fixed-Point Dot Product

```
         .global _dotp

_dotp:  .cproc   a, b

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

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

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

         ADD      sum0,sum1,sum    ; compute final result

         .return sum

         .endproc
```

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

Example 6–25. Linear Assembly for Full Floating-Point Dot Product

```
         .global _dotp

_dotp:  .cproc   a, b

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

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

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

         ADDSP    sum,sum1,sum0   ; compute final result

         .return sum

         .endproc
```

6.5.3 Final Assembly

Example 6–26 shows the assembly code for the fixed-point software-pipelined dot product in Table 6–7 on page 6-35. Example 6–27 shows the assembly code for the floating-point software-pipelined dot product in Table 6–8 on page 6-36. The accumulators are initialized to 0 and the loop counter is set up in the first execute packet in parallel with the first load instructions. The asterisks in the comments correspond with those in Table 6–7 and Table 6–8, respectively.

Note:

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

See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information about pipeline operation.

6.5.3.1 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 6–26 requires a total of 58 cycles (7 + 50 + 1), which is a significant improvement over the 402 cycles required by the code in Example 6–19.

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 6–26. 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
```

6.5.3.2 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 6–27 requires a total of 74 cycles (9 + 50 + 15), which is a significant improvement over the 508 cycles required by the code in Example 6–20.

Example 6–27. 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         ; pi1 = a1  b1
||[A1] B         .S2    LOOP             ;* branch to loop
||[A1] SUB       .S1    A1,1,A1          ;** decrement loop counter

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

Example 6–27. 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         ;** pi1 = a1  b1
||[A1] B         .S2   LOOP             ;*** branch to loop
||[A1] SUB       .S1   A1,1,A1          ;**** decrement loop counter

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

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

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

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

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

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

       NOP                              ; wait for B0

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

       NOP                              ; wait for next B0

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

       NOP             3

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

       NOP             3                ;
```

6.5.3.3 Removing Extraneous Instructions

The code in Example 6–26 and Example 6–27 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 6–26:

❑ 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 6–27:

❑ 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 6–28 on page 6-47 and Example 6–29 on page 6-48.

Fixed-Point Example

To eliminate LDWs in the fixed-point dot product from iterations 51 through 57, run the loop seven fewer times. This brings the loop counter to 43 (50 – 7), which means you still must execute seven more cycles of ADD instructions and five more cycles of MPY instructions. Five pairs of MPYs and seven pairs of ADDs are now outside the loop. The LDWs, MPYs, and ADDs all execute exactly 50 times. (The shaded areas of Example 6–28 indicate the changes in this code.)

Executing the dot product code in Example 6–28 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 6–29 indicate the changes in this code.)

Executing the dot product code in Example 6–29 with no extraneous LDDWs still requires a total of 74 cycles (9 + 41 + 9 + 15), but the code size is now larger.

Example 6–28. 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 6–28. Assembly Code for Fixed-Point Dot Product (Software Pipelined With No Extraneous Loads) (Continued)

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

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

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


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

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

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

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

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

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

Example 6–29. Assembly Code for Floating-Point Dot Product (Software Pipelined With No Extraneous Loads)

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

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

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

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

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

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

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

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


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

Example 6–29. 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     ;**** pi1 = a1  b1
||     ADDSP     .L1    A5,A8,A8     ; sum0 += (ai  bi)
||     ADDSP     .L2    B5,B8,B8     ; sum1 += (ai+1  bi+1)
||[A1] B         .S2    LOOP         ;***** branch to loop
||[A1] SUB       .S1    A1,1,A1      ;****** decrement loop counter
; Branch occurs here
                                                               ADDSPs    MPYSPs
       MPYSP     .M1X   A6,B6,A5     ; pi = a0  b0
||     MPYSP     .M2X   A7,B7,B5     ; pi1 = a1  b1                         ①
||     ADDSP     .L1    A5,A8,A8     ; sum0 += (ai  bi)            ①
||     ADDSP     .L2    B5,B8,B8     ; sum1 += (ai+1  bi+1)

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

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

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

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

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

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

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

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

Example 6–29. 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             ;
```

6.5.3.4 Priming the Loop

Although Example 6–28 and Example 6–29 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 6–26 or Example 6–27, which do not have epilogs and, therefore, contain fewer instructions. (This technique can be used equally well with Example 6–28 or Example 6–29.)

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 6–30 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 6–30 requires a total of 65 cycles (7 + 57+ 1). Therefore, you are reducing the code size with a slight loss in performance.

Example 6–30. 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 6–31 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 6–31 requires a total of 81 cycles (7 + 59+ 15). Therefore, you are reducing the code size with a slight loss in performance.

Example 6–31. 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 6–31. 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      ;**** pi1 = a1  b1
||       ADDSP    .L1      A5,A8,A8      ; sum0 += (ai  bi)
||       ADDSP    .L2      B5,B8,B8      ; sum1 += (ai+1  bi+1)
||[A1]   B        .S2      LOOP          ;***** branch to loop
||[A1]   SUB      .S1      A1,1,A1       ;****** decrement loop counter
; Branch occurs here

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

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

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

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

         NOP                             ; wait for B0

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

         NOP                             ; wait for next B0

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

         NOP               3

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

         NOP               3             ;
```

6.5.3.5 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 6–32 and Example 6–33. 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 6–30 and Example 6–31.) 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 6–30 and Example 6–31. The loop in Example 6–32 requires 63 cycles (5 + 57 + 1) and the loop in Example 6–31 requires 79 cycles (5 + 59 + 15).

Example 6–32. 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 6–33. 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        ;**** pi1 = a1  b1
||     ADDSP  .L1    A5,A8,A8        ; sum0 += (ai  bi)
||     ADDSP  .L2    B5,B8,B8        ; sum1 += (ai+1  bi+1)
||[A1] B      .S2    LOOP            ;***** branch to loop
||[A1] SUB    .S1    A1,1,A1         ;****** decrement loop counter
; Branch occurs here

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

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

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

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

       NOP                           ; wait for B0

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

       NOP                           ; wait for next B0

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

       NOP           3

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

       NOP           3               ;
```

6.5.4 Comparing Performance

Table 6–10 compares the performance of all versions of the fixed-point dot product code. Table 6–11 compares the performance of all versions of the floating-point dot product code.

Table 6–10. Comparison of Fixed-Point Dot Product Code Examples

Code Example		100 Iterations	Cycle Count
Example 6–9	Fixed-point dot product linear assembly	2 + 100 × 16	1602
Example 6–10	Fixed-point dot product parallel assembly	1 + 100 × 8	801
Example 6–19	Fixed-point dot product parallel assembly with LDW	1 + (50 × 8) + 1	402
Example 6–26	Fixed-point software-pipelined dot product	7 + 50 + 1	58
Example 6–28	Fixed-point software-pipelined dot product with no extraneous loads	7 + 43 + 7 + 1	58
Example 6–30	Fixed-point software-pipelined dot product with no prolog or epilog	7 + 57 + 1	65
Example 6–32	Fixed-point software-pipelined dot product with smallest code size	5 + 57 + 1	63

Table 6–11. Comparison of Floating-Point Dot Product Code Examples

Code Example		100 Iterations	Cycle Count
Example 6–11	Floating-point dot product nonparallel assembly	2 + 100 × 21	2102
Example 6–12	Floating-point dot product parallel assembly	1 + 100 × 10	1001
Example 6–20	Floating-point dot product parallel assembly with LDDW	1 + (50 × 10) + 7	508
Example 6–27	Floating-point software-pipelined dot product	9 + 50 + 15	74
Example 6–29	Floating-point software-pipelined dot product with no extraneous loads	9 + 41 + 9 + 15	74
Example 6–31	Floating-point software-pipelined dot product with no prolog or epilog	7 + 59 + 15	81
Example 6–33	Floating-point software-pipelined dot product with smallest code size	5 + 59 + 15	79

6.6 Modulo Scheduling of Multicycle Loops

Section 6.5 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.

6.6.1 Weighted Vector Sum C Code

Example 6–34 shows the C code for a weighted vector sum.

Example 6–34. 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];
       }
}
```

6.6.2 Translating C Code to Linear Assembly

Example 6–35 shows the linear assembly that executes the weighted vector sum in Example 6–34. 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 6–35. 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
```

6.6.3 Determining the Minimum Iteration Interval

Example 6–35 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.

6.6.3.1 Unrolling the Weighted Vector Sum C Code

Example 6–36 shows the C code for an unrolled version of the weighted vector sum.

Example 6–36. 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];
       }
}
```

6.6.3.2 Translating Unrolled Inner Loop to Linear Assembly

Example 6–37 shows the linear assembly that calculates c[i] and c[i+1] for the weighted vector sum in Example 6–36.

- 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 6–37. 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
```

6.6.3.3 Determining a New Minimum Iteration Interval

Use the following considerations to determine the minimum iteration interval for the assembly instructions in Example 6–37:

- 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 6–35.

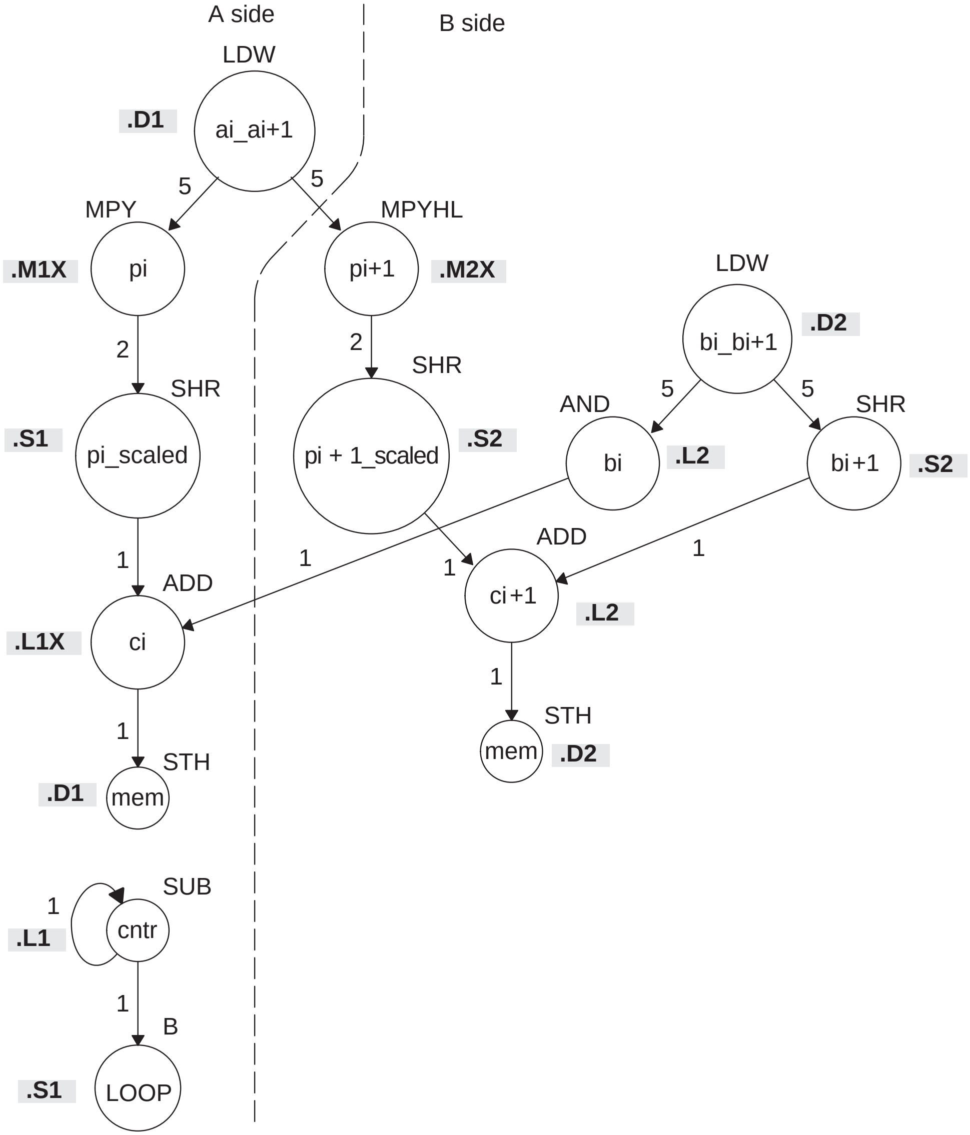
By using LDWs instead of LDHs, the program can do twice as much work in the same number of cycles.

6.6.4 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 6–11 shows the dependency graph divided evenly with a minimum iteration interval of 2.

Figure 6–11. Dependency Graph of Weighted Vector Sum



6.6.5 Linear Assembly Resource Allocation

Using the dependency graph, you can allocate functional units and registers as shown in Example 6–38. 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 6–38. 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
```

6.6.6 Modulo Iteration Interval Scheduling

Table 6–12 provides a method to keep track of resources that are a modulo iteration interval away from each other. In the single-cycle dot product example, every instruction executed every cycle and, therefore, required only one set of resources. Table 6–12 includes two groups of resources, which are necessary because you are scheduling a two-cycle loop.

- Instructions that execute on cycle k also execute on cycle k + 2, k + 4, etc. Instructions scheduled on these even cycles cannot use the same resources.

- Instructions that execute on cycle k + 1 also execute on cycle k + 3, k + 5, etc. Instructions scheduled on these odd cycles cannot use the same resources.

- Because two instructions (MPY and ADD) use the 1X path but do not use the same functional unit, Table 6–12 includes two rows (1X and 2X) that help you keep track of the cross path resources.

Only seven instructions have been scheduled in this table.

- ❑ The two LDWs use the .D units on the even cycles.
- ❑ The MPY and MPYH are scheduled on cycle 5 because the LDW has four delay slots. The MPY instructions appear in two rows because they use the .M and cross path resources on cycles 5, 7, 9, etc.
- ❑ The two SHR instructions are scheduled two cycles after the MPY to allow for the MPY’s single delay slot.
- ❑ The AND is scheduled on cycle 5, four delay slots after the LDW.

Table 6–12. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

Unit/Cycle	**0**	**2**	**4**	**6**	**8**	**10**
.D1	**LDW ai_i+1**	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	**LDW bi_i+1**	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1						
.L2						
.S1						
.S2						
1X						
2X						
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11**
.D1						
.D2						
.M1			**MPY pi**	* MPY pi	** MPY pi	*** MPY pi
.M2			**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1			**AND bi**	* AND bi	** AND bi	*** AND bi
.L2						
.S1				**SHR pi_s**	* SHR pi_s	** SHR pi_s
.S2				**SHR pi+1_s**	* SHR pi+1_s	** SHR pi+1_s
1X			**MPY pi**	* MPY pi	** MPY pi	*** MPY pi
2X			**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

Note: The asterisks indicate the iteration of the loop; shaded cells indicate cycle 0.

6.6.6.1 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 6–13 shows the addition of the SHR bi+1 instruction. This must avoid a conflict of resources in cycles 5 and 7, which are one iteration interval away from each other.

Even though LDW bi_i+1 (.D2, cycle 0) finishes on cycle 5, its child, SHR bi+1, cannot be scheduled on .S2 until cycle 6 because of a resource conflict with SHR pi+1_scaled, which is on .S2 in cycle 7.

Figure 6–12. Dependency Graph of Weighted Vector Sum (Showing Resource Conflict)

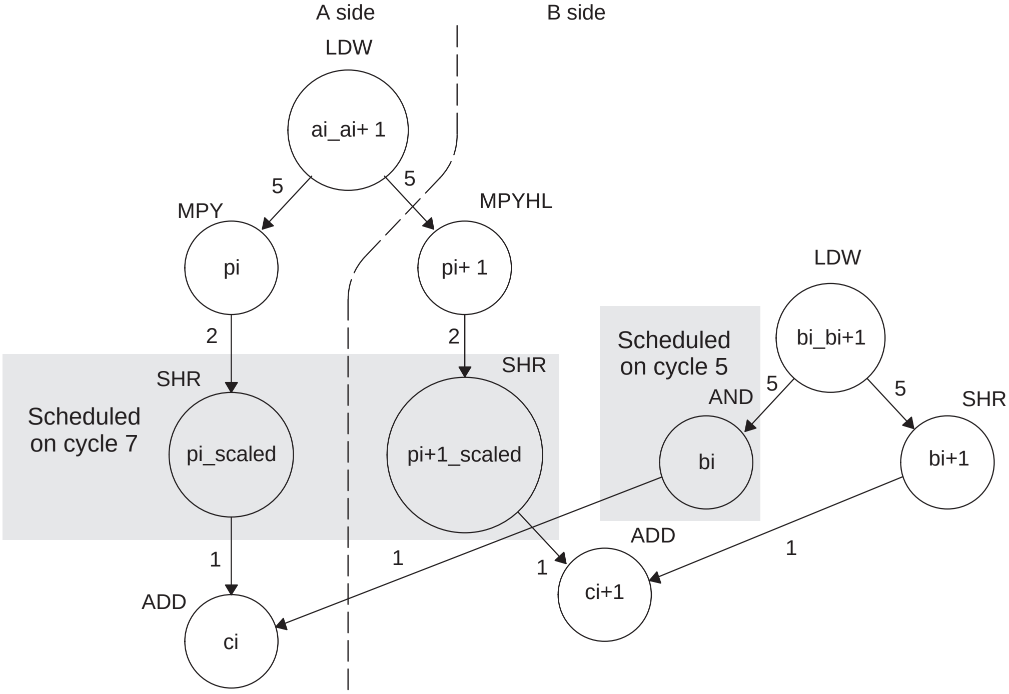


Table 6–13. Modulo Iteration Interval Table for Weighted Vector Sum With SHR Instructions

Unit / Cycle	0	2	4	6	8	10, 12, 14, ...
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	LDW bi_i+1	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1						
.L2						
.S1						
.S2				SHR bi+1	* SHR bi+1	** SHR bi+1
1X						
2X						

Unit / Cycle	1	3	5	7	9	11, 13, 15, ...
.D1						
.D2						
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1			AND bi	* AND bi	** AND bi	*** AND bi
.L2						
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2				SHR pi+1_s	* SHR pi+1_s	** SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi
2X			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–12.

6.6.6.2 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.

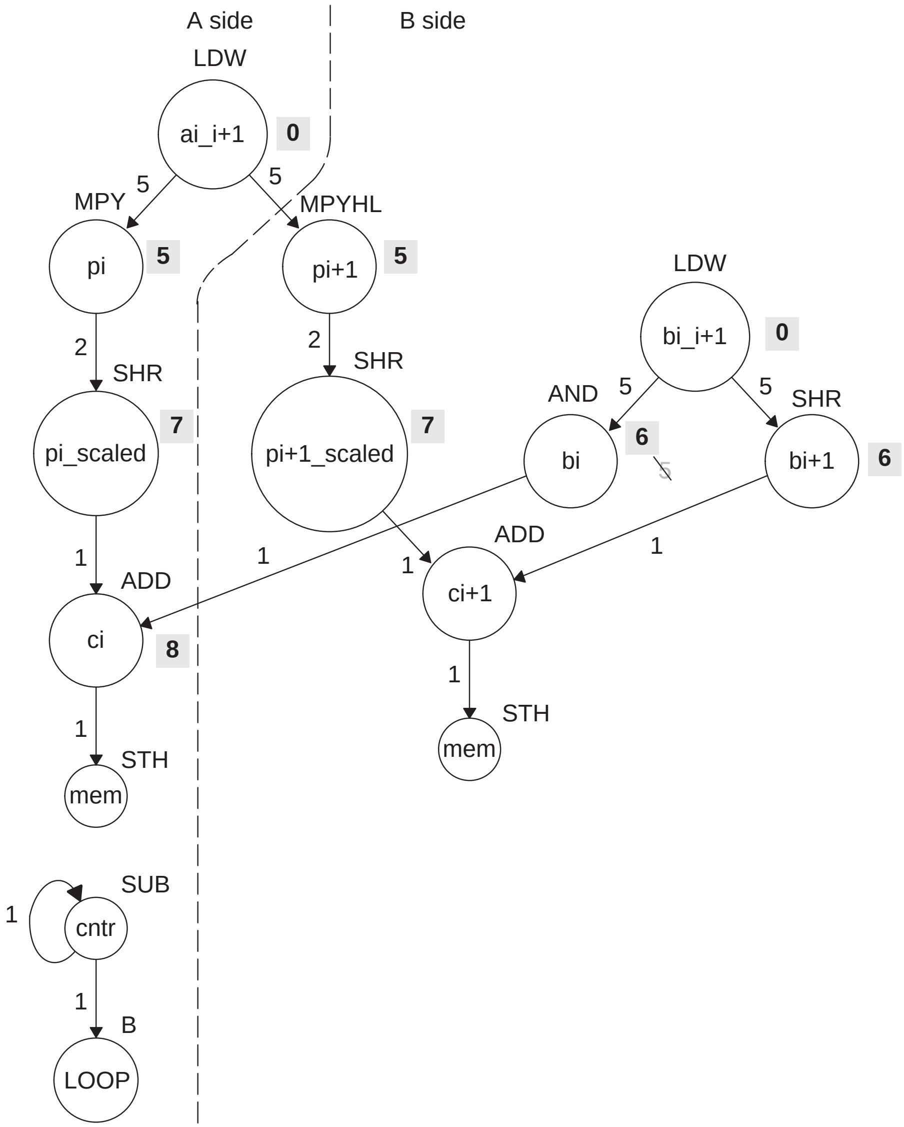
No value can be live in a register for more than the number of cycles in the loop. Otherwise, iteration n + 1 writes into the register before iteration n has read that register. Therefore, in a 2-cycle loop, a value is written to a register at the end of cycle n, then all children of that value must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

6.6.6.3 Solving the Live-Too-Long Problem

The live-too-long problem in Table 6–13 means that the bi value would have to be live from cycles 6–8, or 3 cycles. *No loop variable can live longer than the iteration interval,* because a child would then read the parent value for the next iteration.

To solve this problem move AND bi to cycle 6 so that you can schedule ADD ci to read the correct value on cycle 8, as shown in Figure 6–13 and Table 6–14.

Figure 6–13. Dependency Graph of Weighted Vector Sum (With Resource Conflict Resolved)



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

Table 6–14. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

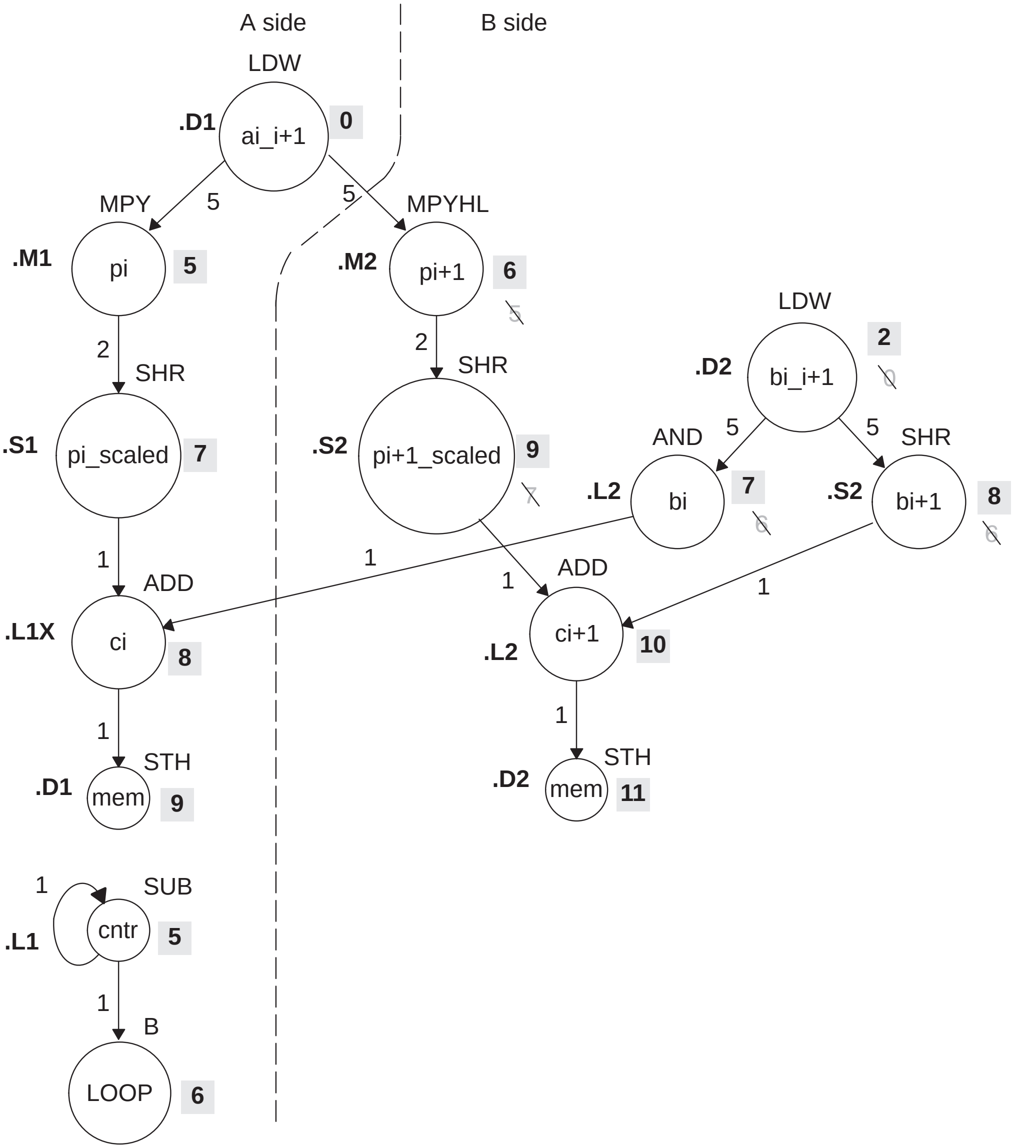
Unit/Cycle	**0**	**2**	**4**	**6**	**8**	**10**
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2	LDW bi_i+1	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1	***** LDW bi_i+1
.M1						
.M2						
.L1					**ADD ci**	* ADD ci
.L2				**AND bi**	* AND bi	** AND bi
.S1						
.S2				SHR bi+1	* SHR bi+1	** SHR bi+1
1X						
2X						
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11**
.D1						
.D2						
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1
.L1						
.L2						
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2				SHR pi+1_s	* SHR pi+1_s	** SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi
2X			MPYHL pi+1	* MPYHL pi+1	** MPYHL pi+1	*** MPYHL pi+1

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–13.

6.6.6.4 Scheduling the Remaining Instructions

Figure 6–14 shows the dependency graph with additional scheduling changes. The final version of the loop, with all instructions scheduled correctly, is shown in Table 6–15.

Figure 6–14. Dependency Graph of Weighted Vector Sum (Scheduling ci +1)



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

Table 6–15 shows the following additions:

- ❑ B LOOP (.S1, cycle 6)
- ❑ SUB cntr (.L1, cycle 5)
- ❑ ADD ci+1 (.L2, cycle 10)
- ❑ STH ci (cycle 9)
- ❑ STH ci+1 (cycle 11)

To avoid resource conflicts and live-too-long problems, Table 6–15 also includes the following additional changes:

- ❑ LDW bi_i+1 (.D2) moved from cycle 0 to cycle 2.
- ❑ AND bi (.L2) moved from cycle 6 to cycle 7.
- ❑ SHR pi+1_scaled (.S2) moved from cycle 7 to cycle 9.
- ❑ MPYHL pi+1 moved from cycle 5 to cycle 6.
- ❑ SHR bi+1 moved from cycle 6 to 8.

From the table, you can see that this loop is pipelined six iterations deep, because iterations n and n + 5 execute in parallel.

Table 6–15. Modulo Iteration Interval Table for Weighted Vector Sum (2-Cycle Loop)

Unit/Cycle	**0**	**2**	**4**	**6**	**8**	**10, 12, 14, ...**
.D1	LDW ai_i+1	* LDW ai_i+1	** LDW ai_i+1	*** LDW ai_i+1	**** LDW ai_i+1	***** LDW ai_i+1
.D2		**LDW bi_i+1**	* LDW bi_i+1	** LDW bi_i+1	*** LDW bi_i+1	**** LDW bi_i+1
.M1						
.M2				**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1
.L1					**ADD ci**	* ADD ci
.L2						**ADD ci+1**
.S1				**B LOOP**	* B LOOP	** B LOOP
.S2					**SHR bi+1**	* SHR bi+1
1X					**ADD ci**	* ADD ci
2X				**MPYHL pi+1**	* MPYHL pi+1	** MPYHL pi+1
Unit/Cycle	**1**	**3**	**5**	**7**	**9**	**11, 13, 15, ...**
.D1					**STH ci**	* STH ci
.D2						**STH ci+1**
.M1			MPY pi	* MPY pi	** MPY pi	*** MPY pi
.M2						
.L1			**SUB cntr**	* SUB cntr	** SUB cntr	*** SUB cntr
.L2				**AND bi**	* AND bi	** AND bi
.S1				SHR pi_s	* SHR pi_s	** SHR pi_s
.S2					**SHR pi+1_s**	* SHR pi+1_s
1X			MPY pi	* MPY pi	** MPY pi	*** MPY pi
2X						

Note: The asterisks indicate the iteration of the loop; shading indicates changes in scheduling from Table 6–14.

6.6.7 Using the Assembly Optimizer for the Weighted Vector Sum

Example 6–39 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 6–39. Linear Assembly for Weighted Vector Sum

```
         .global _w_vec

_w_vec: .cproc    a, b, c, m

         .reg     ai_i1, bi_i1, pi, pi1, pi_i1, pi_s, pi1_s
         .reg     mask, bi, bi1, ci, ci1, c1, cntr

         MVK      –1,mask               ; set to all 1s to create 0xFFFFFFFF
         MVKH     0,mask                ; clear upper 16 bits to create 0xFFFF
         MVK      50,cntr               ; cntr = 100/2
         ADD      2,c,c1                ; point to c[1]

LOOP:    .trip 50
         LDW      .D1      *a++,ai_i1   ; ai & ai+1
         LDW      .D2      *b++,bi_i1   ; bi & bi+1
         MPY      .M1X     ai_i1,m,pi   ; m * ai
         MPYHL    .M2X     ai_i1,m,pi1  ; m * ai+1
         SHR      .S1      pi,15,pi_s   ; (m * ai) >> 15
         SHR      .S2      pi1,15,pi1_s ; (m * ai+1) >> 15
         AND      .L2      bi_i1,mask,bi; bi
         SHR      .S2      bi_i1,16,bi1 ; bi+1
         ADD      .L1X     pi_s,bi,ci   ; ci = (m * ai) >> 15 + bi
         ADD      .L2      pi1_s,bi1,ci1; ci+1 = (m * ai+1) >> 15 + bi+1
         STH      .D1      ci,*c++[2]   ; store ci
         STH      .D2      ci1,*c1++[2] ; store ci+1
[cntr]   SUB      .L1      cntr,1,cntr  ; decrement loop counter
[cntr]   B        .S1      LOOP         ; branch to loop

         .endproc
```

6.6.8 Final Assembly

Example 6–40 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 6–40. 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 6–40. 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
```

6.7 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.

6.7.1 IIR Filter C Code

Example 6–41 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 6–41. 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;
       }
}
```

6.7.2 Translating C Code to Linear Assembly (Inner Loop)

Example 6–42 shows the ’C6000 instructions that execute the inner loop of the IIR filter C code. In this example:

- xptr is not postincremented after loading xi+1, because xi of the next iteration is actually xi+1 of the current iteration. Thus, the pointer points to the same address when loading both xi+1 for one iteration and xi for the next iteration.

- yptr is also not postincremented after storing yi+1, because yi of the next iteration is yi+1 for the current iteration.

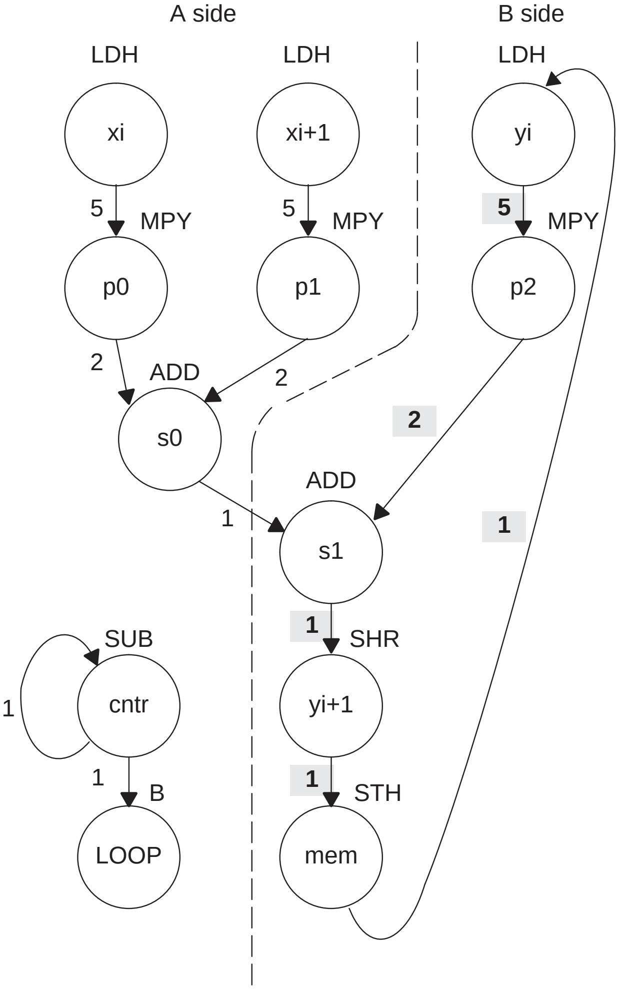
Example 6–42. Linear Assembly for IIR Inner Loop

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

6.7.3 Drawing a Dependency Graph

Figure 6–15 shows the dependency graph for the IIR filter. A loop carry path exists from the store of yi+1 to the load of yi. The path between the STH and the LDH is one cycle because the load and store instructions use the same memory pipeline. Therefore, if a store is issued to a particular address on cycle n and a load from that same address is issued on the next cycle, the load reads the value that was written by the store instruction.

Figure 6–15. Dependency Graph of IIR Filter



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

6.7.4 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 6–16, the minimum iteration interval is 2.

Note:

There are six non-.M units available: three on the A side (.S1, .D1, .L1) and three on the B side (.S2, .D2, .L2). Therefore, to determine resource constraints, divide the total number of non-.M units used on each side by 3 (3 is the total number of non-.M units available on each side).

Based on non-.M unit resources in Table 6–16, the minimum iteration interval for the IIR filter is 2 because the total non-.M units on the A side is 5 (5 ÷ 3 is greater than 1 so you round up to the next whole number). The B side uses only three non-.M units, so this does not affect the minimum iteration interval, and no other unit is used more than twice.

Table 6–16. Resource Table for IIR Filter

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	2 MPYs	2
.S1	B	1
.D1	2 LDHs	2
.L1,.S1, or .D1	ADD & SUB	2
Total non-.M units		5

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	MPY	1
.S2	SHR	1
.D2	STH	1
.L2 or .S2, .D2	ADD	1
Total non-.M units		3

However, the IIR has a data dependency constraint defined by its loop carry path. Figure 6–15 shows that if you schedule LDH yi on cycle 0:

❑ The earliest you can schedule MPY p2 is on cycle 5.

❑ The earliest you can schedule ADD s1 is on cycle 7.

❑ SHR yi+1 must be on cycle 8 and STH on cycle 9.

❑ Because the LDH must wait for the STH to be issued, the earliest the the second iteration can begin is cycle 10.

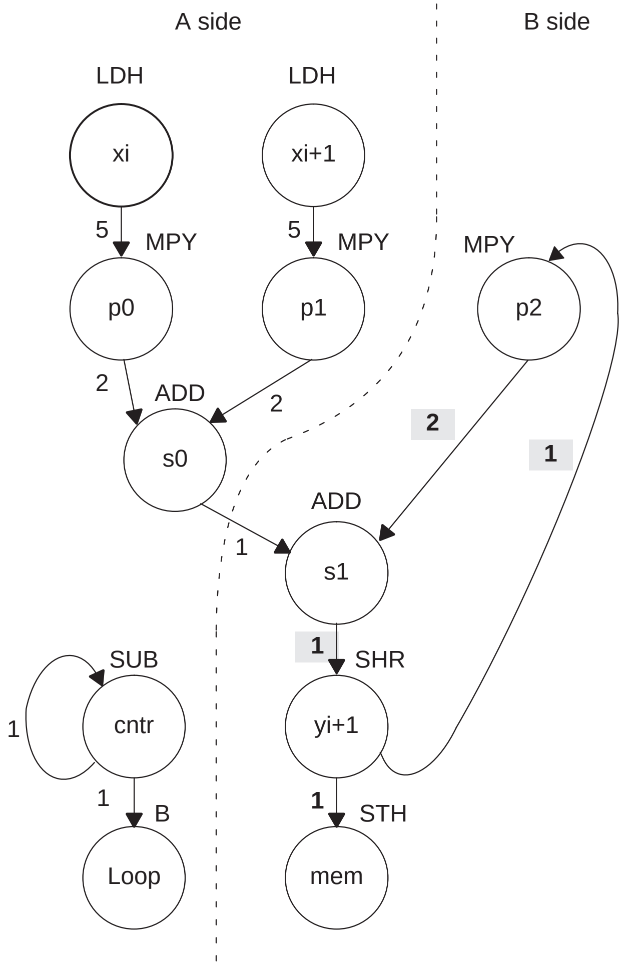
To determine the minimum loop carry path, add all of the numbers along the loop paths in the dependency graph. This means that this loop carry path is 10 (5 + 2 + 1 + 1 + 1).

Although the minimum iteration interval is the greater of the resource limits and data dependency constraints, an interval of 10 seems slow. Figure 6–16 shows how to improve the performance.

6.7.4.1 Drawing a New Dependency Graph

Figure 6–16 shows a new graph with a loop carry path of 4 (2 +1 + 1). because the MPY p2 instruction can read yi+1 while it is still in a register, you can reduce the loop carry path by six cycles. LDH yi is no longer in the graph. Instead, you can issue LDH y[0] once outside the loop. In every iteration after that, the y+1 values written by the SHR instruction are valid y inputs to the MPY instruction.

Figure 6–16. Dependency Graph of IIR Filter (With Smaller Loop Carry)



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

6.7.4.2 New ’C6x Instructions (Inner Loop)

Example 6–43 shows the new linear assembly from the graph in Figure 6–16, where LDH yi was removed. The one variable y that is read and written is yi for the MPY p2 instruction and yi+1 for the SHR and STH instructions.

Example 6–43. 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
 [cntr]SUB    cntr,1,cntr   ; decrement loop counter
 [cntr]B      LOOP          ; branch to loop
```

6.7.5 Linear Assembly Resource Allocation

Example 6–44 shows the same linear assembly instructions as those in Example 6–43 with the functional units and registers assigned.

Example 6–44. 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
```

6.7.6 Modulo Iteration Interval Scheduling

Table 6–17 shows the modulo iteration interval table for the IIR filter. The SHR instruction on cycle 10 finishes in time for the MPY p2 instruction from the next iteration to read its result on cycle 11.

Table 6–17. Modulo Iteration Interval Table for IIR (4-Cycle Loop)

Unit/Cycle	0	4	8, 12, 16, ...	Unit/Cycle	1	5	9, 13, 17, ...
.D1	LDH xi	* LDH xi	** LDH xi	.D1	LDH xi+1	* LDH xi+1	** LDH ci+1
.D2			ADD s0	.D2			
.M1				.M1		MPY p0	* MPY p0
.M2				.M2			
.L1				.L1		SUB cntr	* SUB cntr
.L2				.L2			ADD s1
.S1				.S1			
.S2				.S2			
1X				1X			
2X				2X			ADD s1

Unit/Cycle	2	6	10, 14, 18, ...	Unit/Cycle	3	7	11, 15, 19, ...
.D1				.D1			
.D2				.D2			STH yi+1
.M1		MPY p1	* MPY p1	.M1			
.M2				.M2		MPY p2	***** **MPY p2**
.L1				.L1			
.L2				.L2			
.S1		B LOOP	* B LOOP	.S1			
.S2			**SHR yi+1**	.S2			
1X		MPY p1	* MPY p1	1X			
2X				2X		MPY p2	* MPY p2

Note: The asterisks indicate the iteration of the loop.

6.7.7 Using the Assembly Optimizer for the IIR Filter

Example 6–45 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 6–45. Linear Assembly for IIR Filter

```
       .global _iir

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

       .reg     xi, xi1, yi1
       .reg     p0, p1, p2, s0, s1, cntr

       MVK      100,cntr                  ; cntr = 100

       LDH      .D2  *y++,yi1             ; yi+1

LOOP:  .trip 100
       LDH      .D1  *x++,xi              ; xi
       MPY      .M1  c1,xi,p0             ; c1 * xi
       LDH      .D1  *x,xi1               ; xi+1
       MPY      .M1X c2,xi1,p1            ; c2 * xi+1
       ADD      .L1  p0,p1,s0             ; c1 * xi + c2 * xi+1
       MPY      .M2X c3,yi1,p2            ; c3 * yi
       ADD      .L2X s0,p2,s1             ; c1 * xi + c2 * xi+1 + c3 * yi
       SHR      .S2  s1,15,yi1            ; yi+1
       STH      .D2  yi1,*y++             ; store yi+1
[cntr] SUB      .L1  cntr,1,cntr          ; decrement loop counter
[cntr] B        .S1  LOOP                 ; branch to loop

       .endproc
```

6.7.8 Final Assembly

Example 6–46 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 6–46 requires 408 cycles: (4 × 100) + 8.

Example 6–46. 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
```

6.8 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 ’C6000 instructions can be conditional on one of five general-purpose registers on the ’C62x and ’C67x and one of 6 on the ’C64x. Conditional instructions can handle both the true and false cases of the if-then-else C statement.

6.8.1 If-Then-Else C Code

Example 6–47 contains a loop with an if-then-else statement. You either add a[i] to sum or subtract a[i] from sum.

Example 6–47. 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.

6.8.2 Translating C Code to Linear Assembly

Example 6–48 shows the linear assembly instructions needed to execute inner loop of the C code in Example 6–47.

Example 6–48. 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.

6.8.3 Drawing a Dependency Graph

Figure 6–17 shows the dependency graph for the if-then-else C code. This graph illustrates the following arrangement:

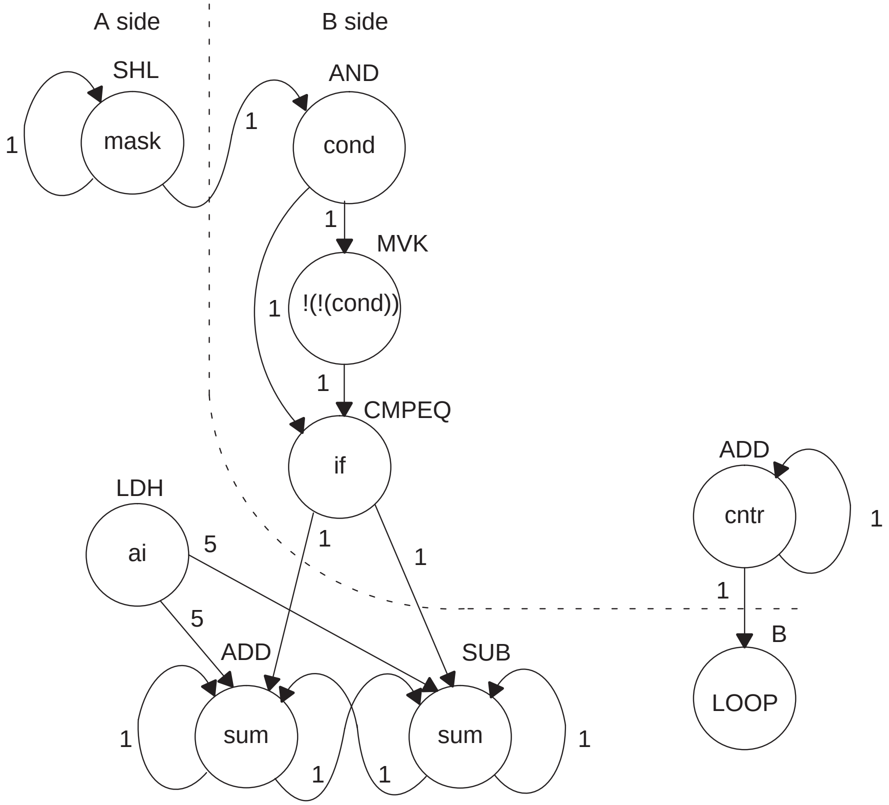
❑ Two nodes on the graph contain sum: one for the ADD and one for the SUB. Because some iterations are performing an ADD and others are performing a SUB, each of these nodes is a possible input to the next iteration of either node.

❑ The LDH ai instruction is a parent of both ADD sum and SUB sum, because both instructions read ai.

❑ CMPEQ if is also a parent to ADD sum and SUB sum, because both read IF for the conditional execution.

❑ The result of SHL mask is read on the next iteration by the AND cond instruction.

Figure 6–17. Dependency Graph of If-Then-Else Code



6.8.4 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 6–17 is split, five instructions are on the A side and four are on the B side. Because none of the instructions are MPYs, all instructions must go on the .S, .D, or .L units, which means you have a total of six resources.

- LDH must be on a .D unit.
- SHL, B, and MVK must be on a .S unit.
- The ADDs and SUB can be on the .S, .L, or .D units.
- The AND can be on a .S or .L unit, or .D unit (’C64x only)

From Table 6–18, you can see that no one resource is used more than two times, so the minimum iteration interval is still 2.

Table 6–18. Resource Table for If-Then-Else Code

(a) A side

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

(b) B side

Unit(s)	Instructions	Total/Unit
.M2		0
.S2	MVK	1
.L2	CMPEQ	1
.L2 or .S2	AND	1
.L2, .S2, or .D2	ADD	1
Total non-.M units		4

The minimum iteration interval is also affected by the total number of instructions. Because three units can perform nonmultiply operations on a given side, a total of five instructions can be performed with a minimum iteration interval of 2. Because only four instructions are on the B side, the minimum iteration interval is still 2.

6.8.5 Linear Assembly Resource Allocation

Now that the graph is split and you know the minimum iteration interval, you can allocate functional units and registers to the instructions. You must ensure that no resource is used more than twice.

Example 6–49 shows the linear assembly with the functional units and registers that are used in the inner loop.

Example 6–49. 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
```

6.8.6 Final Assembly

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

Example 6–50. 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
```

6.8.7 Comparing Performance

You can improve the performance of the code in Example 6–50 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 6–51. 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 6–51 shows the improved loop with a cycle count of 68 (2 × 32 + 4). Table 6–19 compares the performance of Example 6–50 and Example 6–51.

Table 6–19. Comparison of If-Then-Else Code Examples

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

6.9 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 6–52, 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 6–51.

6.9.1 Unrolled If-Then-Else C Code

Example 6–52 shows the unrolled version of the if-then-else C code in Example 6–47 on page 6-86.

Example 6–52. 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);
}
```

6.9.2 Translating C Code to Linear Assembly

Example 6–53 shows the unrolled inner loop with 16 instructions and the possibility of achieving a loop with a minimum iteration interval of 3.

Example 6–53. 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
```

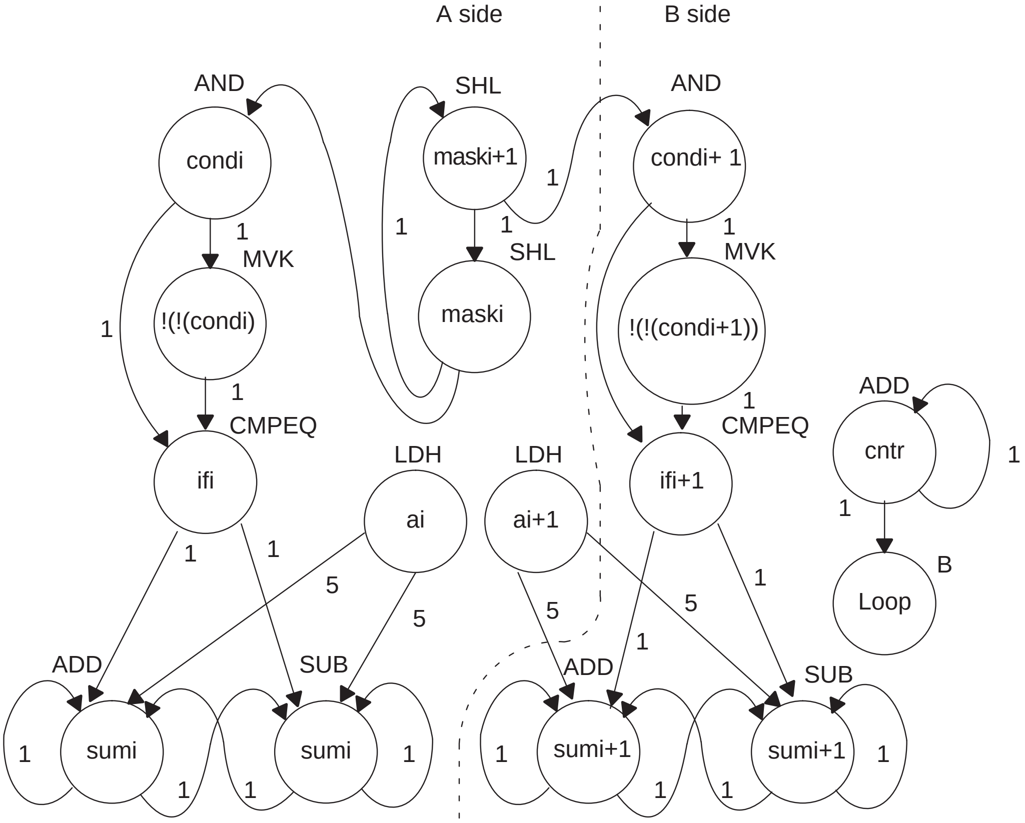
6.9.3 Drawing a Dependency Graph

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 6–18 shows the dependency graph for the unrolled if-then-else code. Nine instructions are on the A side, and seven instructions are on the B side.

Figure 6–18. Dependency Graph of If-Then-Else Code (Unrolled)



6.9.4 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, or .D unit (’C64x only)

From Table 6–20, you can see that no one resource is used more than three times so that the minimum iteration interval is still 3.

Checking the total number of non-.M instructions on each side shows that a total of nine instructions can be performed with the minimum iteration interval of 3. because only seven non-.M instructions are on the B side, the minimum iteration interval is still 3.

Table 6–20. Resource Table for Unrolled If-Then-Else Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1		0
.S1	MVK and 2 SHLs	3
.D1	2 LDHs	2
.L1	CMPEQ	1
.L1 or .S1	AND	1
.L1, .S1, or .D1	ADD and SUB	2
Total non-.M units		9

(b) B side

Unit(s)	Instructions	Total/Unit
.M2		0
.S2	MVK and B	2
.L2	CMPEQ	1
.L2 pr.S2	AND	1
.L2 ,.S2, or .D2	SUB and 2 ADDs	3
Total non-.M units		7

6.9.5 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 6–54 shows the linear assembly code with the functional units and registers.

Example 6–54. 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, ifi1, a, ai, ai1, cntr
         .reg   cdi, cdi1, sumi, sumi1, sum

         MV     A4,a                 ; C callable register for 1st operand
         MV     B4,cword             ; C callable register for 2nd operand
         MV     A6,mask              ; C callable register for 3rd operand
         MV     B6,theta             ; C callable register for 4th operand
         MVK    16,cntr              ; cntr = 32/2
         ZERO   sumi                 ; sumi = 0
         ZERO   sumi1                ; sumi+1 = 0

LOOP:    .trip 32
         AND    .L1X cword,mask,cdi  ; cdi = codeword & maski
  [cdi]  MVK    .S1  1,cdi           ; !(!(cdi))
         CMPEQ  .L1X theta,cdi,ifi   ; (theta == !(!(cdi)))
         LDH    .D1  *a++,ai         ; a[i]
  [ifi]  ADD    .L1  sumi,ai,sumi    ; sum += a[i]
 [!ifi]  SUB    .D1  sumi,ai,sumi    ; sum –= a[i]
         SHL    .S1  mask,1,mask     ; maski+1 = maski << 1;

         AND    .L2X cword,mask,cdi1 ; cdi+1 = codeword & maski+1
 [cdi1]  MVK    .S2  1,cdi1          ; !(!(cdi+1))
         CMPEQ  .L2  theta,cdi1,ifi1 ; (theta == !(!(cdi+1)))
         LDH    .D1  *a++,ai1        ; a[i+1]
 [ifi1]  ADD    .L2  sumi1,ai1,sumi1 ; sum += a[i+1]
[!ifi1]  SUB    .D2  sumi1,ai1,sumi1 ; sum –= a[i+1]
         SHL    .S1  mask,1,mask     ; maski = maski+1 << 1;

 [cntr]  ADD    .D2  –1,cntr,cntr    ; decrement counter
 [cntr]  B      .S2  LOOP            ; for LOOP

         ADD    sumi,sumi1,sum       ; Add sumi and sumi+1 for ret value

         .return sum

         .endproc
```

6.9.6 Final Assembly

Example 6–55 shows the final assembly code after software pipelining. The cycle count of this loop is now 53: (3×16) + 5.

Example 6–55. 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
```

6.9.7 Comparing Performance

Table 6–21 compares the performance of all versions of the if-then-else code examples.

Table 6–21. Comparison of If-Then-Else Code Examples

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

6.10 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 6.6.6.1, *Resource Conflicts,* on page 6-65 showed how to solve this problem simply by moving the parent to a later cycle. This is not always a valid solution.

6.10.1 C Code With Live-Too-Long Problem

Example 6–56 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 6–19 on page 6-103 shows a *split-join* path that causes this live-too-long problem.

Example 6–56. 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);
}
```

6.10.2 Translating C Code to Linear Assembly

Example 6–57 shows the assembly instructions that execute the inner loop in Example 6–56.

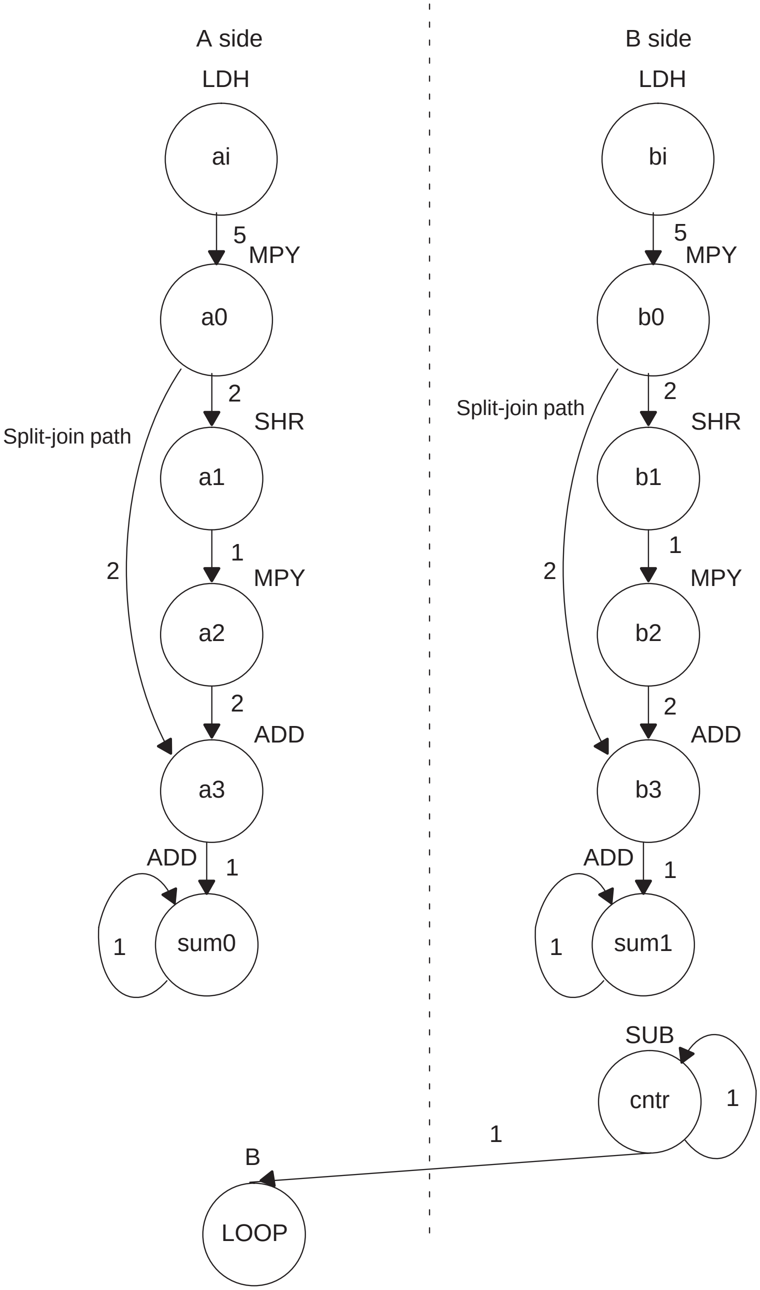
Example 6–57. Linear Assembly for Live-Too-Long Inner Loop

```
       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

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

6.10.3 Drawing a Dependency Graph

Figure 6–19 shows the dependency graph for the live-too-long code. This algorithm includes three separate and independent graphs. Two of the independent graphs have split-join paths: from a0 to a3 and from b0 to b3.

Figure 6–19. Dependency Graph of Live-Too-Long Code



6.10.4 Determining the Minimum Iteration Interval

Table 6–22 shows the functional unit resources for the loop. Based on the resource usage, the minimum iteration interval is 2 for the following reasons:

- ❑ No specific resource is used more than twice, implying a minimum iteration interval of 2.
- ❑ A total of five non-.M units on each side also implies a minimum iteration interval of 2, because three non-.M units can be used on a side during each cycle.

Table 6–22. Resource Table for Live-Too-Long Code

(a) A side

Unit(s)	Instructions	Total/Unit
.M1	MPY	1
.S1	B and SHR	2
.D1	LDH	1
.L1, .S1, or .D1	2 ADDs	2
Total non-.M units		5

(b) B side

Unit(s)	Instructions	Total/Unit
.M2	MPY	1
.S2	SHR	1
.D2	LDH	1
.L2, .S2, or .D2	2 ADDs and SUB	3
Total non-.M units		5

However, the minimum iteration interval is determined by both resources and data dependency. A loop carry path determined the minimum iteration interval of the IIR filter in section 6.7, *Loop Carry Paths*, on page 6-77. In this example, a live-too-long problem determines the minimum iteration interval.

6.10.4.1 Split-Join-Path Problems

In Figure 6–19, the two split-join paths from a0 to a3 and from b0 to b3 create the live-too-long problem. Because the ADD a3 instruction cannot be scheduled until the SHR a1 and MPY a2 instructions finish, a0 must be live for at least four cycles. For example:

- ❑ If MPY a0 is scheduled on cycle 5, then the earliest SHR a1 can be scheduled is cycle 7.
- ❑ The earliest MPY a2 can be scheduled is cycle 8.
- ❑ The earliest ADD a3 can be scheduled is cycle 10.

Because a0 is written at the end of cycle 6, it must be live from cycle 7 to cycle 10, or four cycles. No value can be live longer than the minimum iteration interval, because the next iteration of the loop will overwrite that value before the current iteration can read the value. Therefore, if the value has to be live for four cycles, the minimum iteration interval must be at least 4. A minimum iteration interval of 4 means that the loop executes at half the performance that it could based on available resources.

6.10.4.2 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.

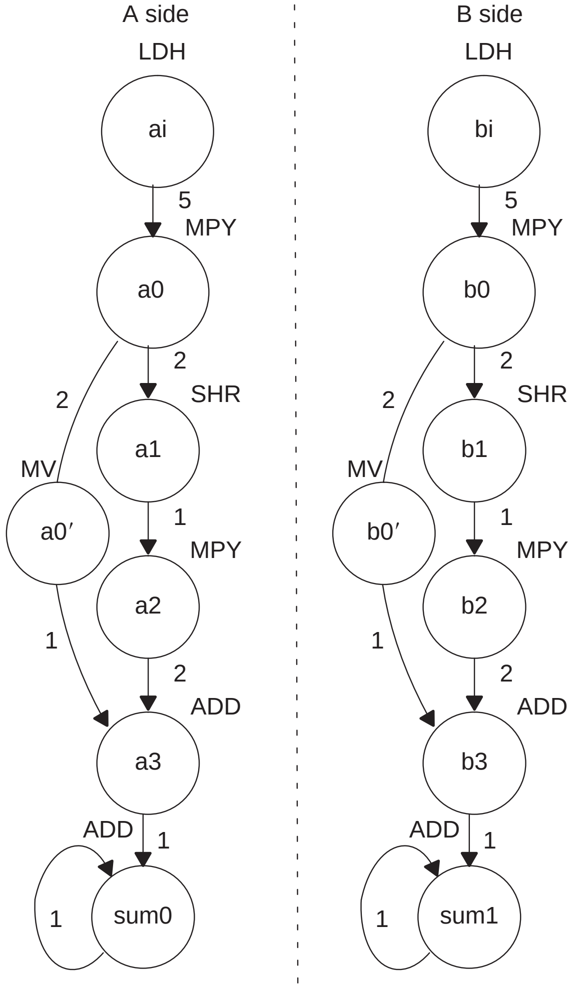
6.10.4.3 Inserting Moves

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.

6.10.4.4 Drawing a New Dependency Graph

Figure 6–20 shows the new dependency graph with the MV instructions. Now the left paths of the split-join paths are broken into two pieces. Each value, a0 and a0’, can be live for minimum iteration interval cycles. If MPY a0 is scheduled on cycle 5 and ADD a3 is scheduled on cycle 10, you can achieve a minimum iteration interval of 2 by scheduling MV a0’ on cycle 8. Then a0 is live on cycles 7 and 8, and a0’ is live on cycles 9 and 10. Because no values are live more than two cycles, the minimum iteration interval for this graph is 2.

Figure 6–20. Dependency Graph of Live-Too-Long Code (Split-Join Path Resolved)



6.10.5 Linear Assembly Resource Allocation

Example 6–58 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 6–58. 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
```

6.10.6 Final Assembly With Move Instructions

Example 6–59 shows the final assembly code after software pipelining. The performance of this loop is 212 cycles (2 × 100 + 11 + 1).

Example 6–59. 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 6–59. 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
```

6.11 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.

6.11.1 FIR Filter C Code

Example 6–60 shows C code for a simple FIR filter. There are two memory reads (x[i+j] and h[i]) for each multiply. Because the ’C6000 can perform only two LDHs per cycle, it seems, at first glance, that only one multiply-accumulate per cycle is possible.

Example 6–60. 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.).

6.11.1.1 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.

6.11.1.2 New FIR Filter C Code

Example 6–61 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 6–61. 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;
         }
}
```

6.11.2 Translating C Code to Linear Assembly

Example 6–62 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.

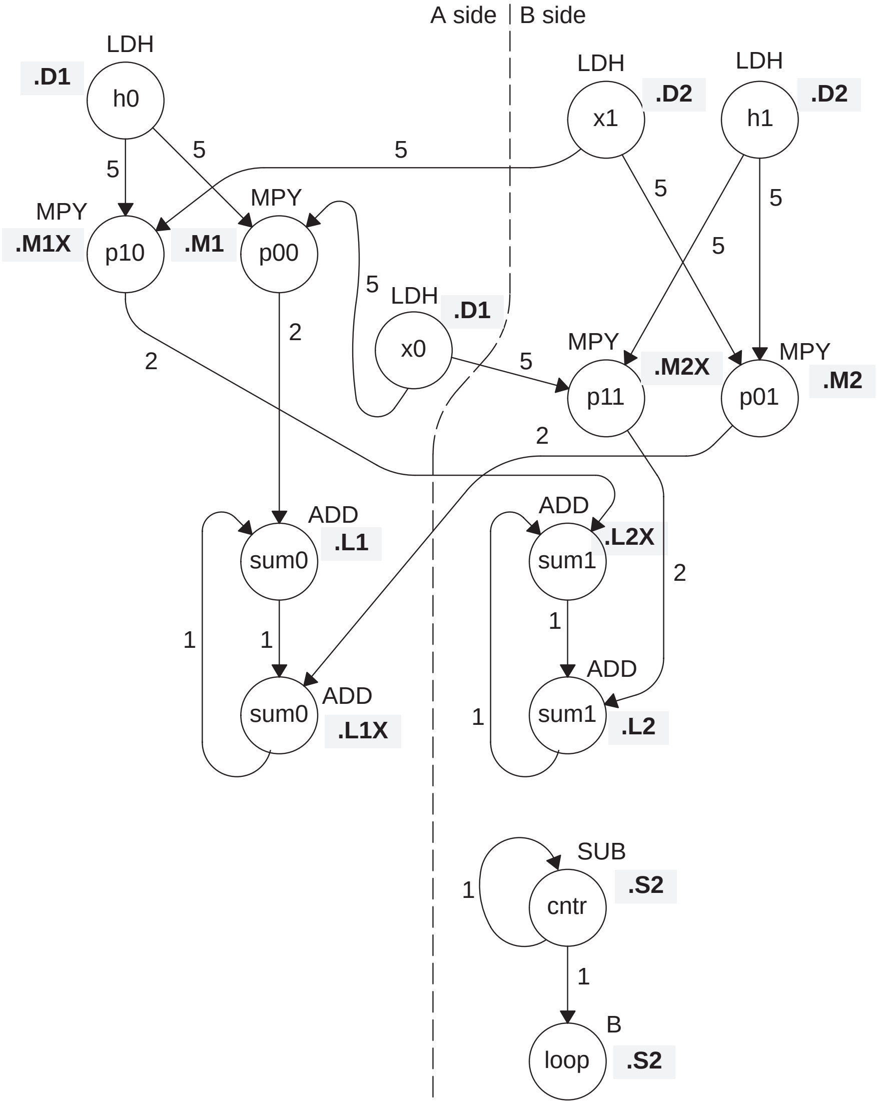
Example 6–62. Linear Assembly for FIR Inner Loop

```
        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
```

6.11.3 Drawing a Dependency Graph

Figure 6–21 shows the dependency graph of the FIR filter with redundant load elimination.

Figure 6–21. Dependency Graph of FIR Filter (With Redundant Load Elimination)



6.11.4 Determining the Minimum Iteration Interval

Table 6–23 shows that the minimum iteration interval is 2. An iteration interval of 2 means that two multiply-accumulates are executing per cycle.

Table 6–23. Resource Table for FIR Filter Code

(a) A side | *(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

6.11.5 Linear Assembly Resource Allocation

Example 6–63 shows the linear assembly with functional units and registers assigned.

Example 6–63. 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 6–63. Linear Assembly for Full FIR Code (Continued)

```
LOOP:    .trip 16

         LDH      .D2      *x_1++[2],x1      ; x1 = x[j+i+1]
         LDH      .D1      *h++[2],h0        ; h0 = h[i]
         MPY      .M1      x0,h0,p00         ; x0 * h0
         MPY      .M1X     x1,h0,p10         ; x1 * h0
         ADD      .L1      p00,sum0,sum0     ; sum0 += x0 * h0
         ADD      .L2X     p10,sum1,sum1     ; sum1 += x1 * h0

         LDH      .D1      *x++[2],x0        ; x0 = x[j+i+2]
         LDH      .D2      *h_1++[2],h1      ; h1 = h[i+1]
         MPY      .M2      x1,h1,p01         ; x1 * h1
         MPY      .M2X     x0,h1,p11         ; x0 * h1
         ADD      .L1X     p01,sum0,sum0     ; sum0 += x1 * h1
         ADD      .L2      p11,sum1,sum1     ; sum1 += x0 * h1

 [ctr]   SUB      .S2      ctr,1,ctr         ; decrement loop counter
 [ctr]   B        .S2      LOOP              ; branch to loop
         SHR      sum0,15,sum0               ; sum0 >> 15
         SHR      sum1,15,sum1               ; sum1 >> 15
         STH      sum0,*y++                  ; y[j] = sum0 >> 15
         STH      sum1,*y++                  ; y[j+1] = sum1 >> 15
         SUB      x,rstx,x                   ; reset x pointer to x[j]
         SUB      h_1,rsth,h_1               ; reset h pointer to h[0]
 [octr]  B        OUTLOOP                    ; branch to outer loop

         .endproc
```

6.11.6 Final Assembly

Example 6–64 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 × 2 + 9 + 6) + 2. Fifteen cycles are overhead for each outer loop:

- Nine cycles execute the inner loop prolog.
- Six cycles execute the branch to the outer loop.

See section 6.13, *Software Pipelining the Outer Loop*, on page 6-131 for information on how to reduce this overhead.

Example 6–64. 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 6–64 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 cntr
||     LDH    .D2    *B4++[2],B0   ;**** h1 = h[i+1]
||     LDH    .D1    *A4++[2],A0   ;**** x0 = x[j+i+2]
       ; inner loop branch occurs here

  [A2] B      .S1    OUTLOOP              ; branch to outer loop                 ①
||     SUB    .L1    A4,A3,A4      ; reset x pointer to x[j]
||     SUB    .L2    B4,B6,B4      ; reset h pointer to h[0]

       SHR    .S1    A9,15,A9      ; sum0 >> 15                                 ②
||     SHR    .S2    B9,15,B9      ; sum1 >> 15

       STH    .D1    A9,*A6++      ; y[j] = sum0 >> 15                          ③

       STH    .D1    B9,*A6++      ; y[j+1] = sum1 >> 15                        ④

       NOP    2                    ; branch delay slots                         ⑤
       ; outer loop branch occurs here                                          ⑥
```

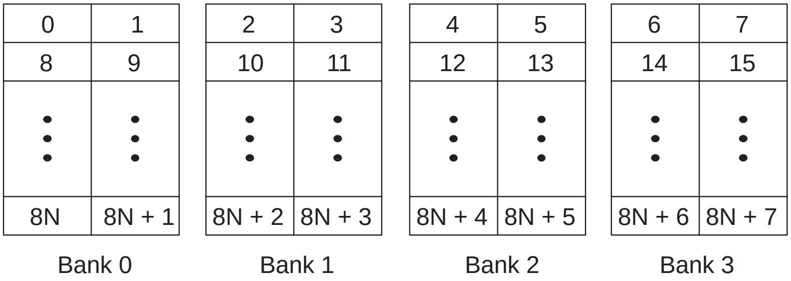
6.12 Memory Banks

The internal memory of the ’C6000 family varies from device to device. See the *TMS320C6000 Peripherals Reference Guide* to determine the memory blocks in your particular device. This section discusses how to write code to avoid memory bank conflicts.

Most ’C6x devices use an interleaved memory bank scheme, as shown in Figure 6–22. Each number in the boxes represents a byte address. A load byte (LDB) instruction from address 0 loads byte 0 in bank 0. A load halfword (LDH) from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. An LDW from address 0 loads bytes 0 through 3 in banks 0 and 1.

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

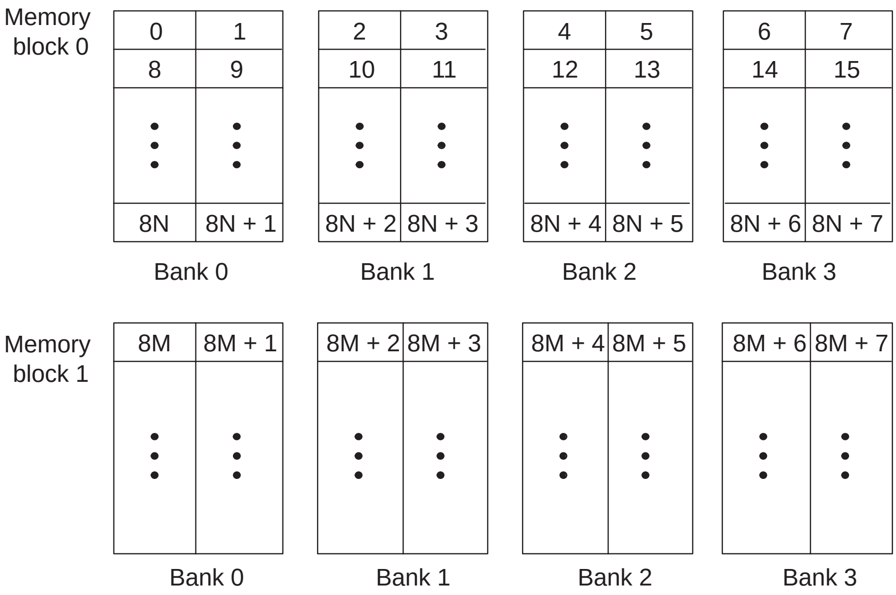
Figure 6–22. 4-Bank Interleaved Memory



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
8N	8N + 1	8N + 2	8N + 3	8N + 4	8N + 5	8N + 6	8N + 7
Bank 0		Bank 1		Bank 2		Bank 3	

For devices that have more than one memory block (see Figure 6–23), an access to bank 0 in one block does not interfere with an access to bank 0 in another memory block, and no pipeline stall occurs.

Figure 6–23. 4-Bank Interleaved Memory With Two Memory Blocks



If each array in a loop resides in a separate memory block, the 2-cycle loop in Example 6–61 on page 6-111 is sufficient. This section describes a solution when two arrays must reside in the same memory block.

6.12.1 FIR Filter Inner Loop

Example 6–65 shows the inner loop from the final assembly in Example 6–64. 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 6–65 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 6–65. 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]
```

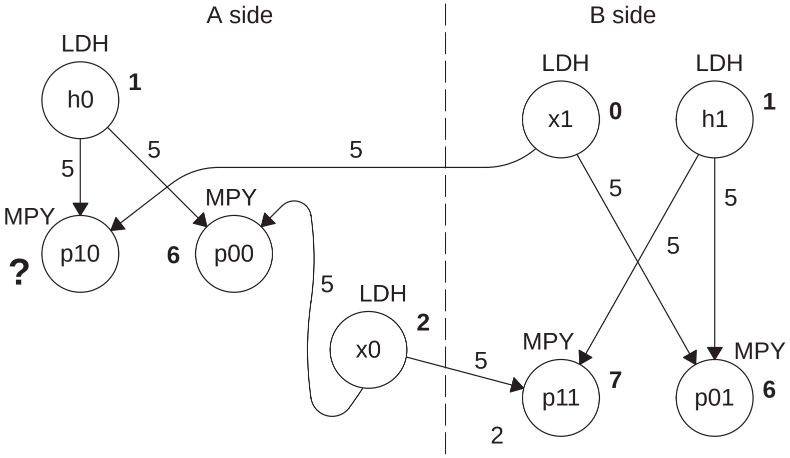
It is not always possible to fully control how arrays are aligned, especially if one of the arrays is passed into a function as a pointer and that pointer has different alignments each time the function is called. One solution to this problem is to write an FIR filter that avoids memory hits, regardless 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 6–24. In this example, a valid 2-cycle software-pipelined loop without memory constraints is ruled by the following constraints:

- ❑ LDH h0 and LDH h1 are on the same loop cycle.
- ❑ LDH x0 and LDH x1 are on the same loop cycle.
- ❑ MPY p00 must be scheduled three or four cycles after LDH x0, because it must read x0 from the previous iteration of LDH x0.
- ❑ All MPYs must be five or six cycles after their LDH parents.
- ❑ No MPYs on the same side (A or B) can be on the same loop cycle.

Figure 6–24. Dependency Graph of FIR Filter (With Even and Odd Elements of Each Array on Same Loop Cycle)



Note: Numbers in bold represent the cycle the instruction is scheduled on.

The scenario in Figure 6–24 *almost* works. All nodes satisfy the above constraints except MPY p10. Because one parent is on cycle 1 (LDH h0) and another on cycle 0 (LDH x1), the only cycle for MPY p10 is cycle 6. However, another MPY on the A side is also scheduled on cycle 6 (MPY p00). Other combinations of cycles for this graph produce similar results.

6.12.2 Unrolled FIR Filter C Code

The main limitation in solving the problem in Figure 6–24 is in scheduling a 2-cycle loop, which means that no value can be live more than two cycles. Increasing the iteration interval to 3 decreases performance. A better solution is to unroll the inner loop one more time and produce a 4-cycle loop.

Example 6–66 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 6–66. 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;
         }
}
```

6.12.3 Translating C Code to Linear Assembly

Example 6–67 shows the linear assembly for the unrolled inner loop of the FIR filter C code.

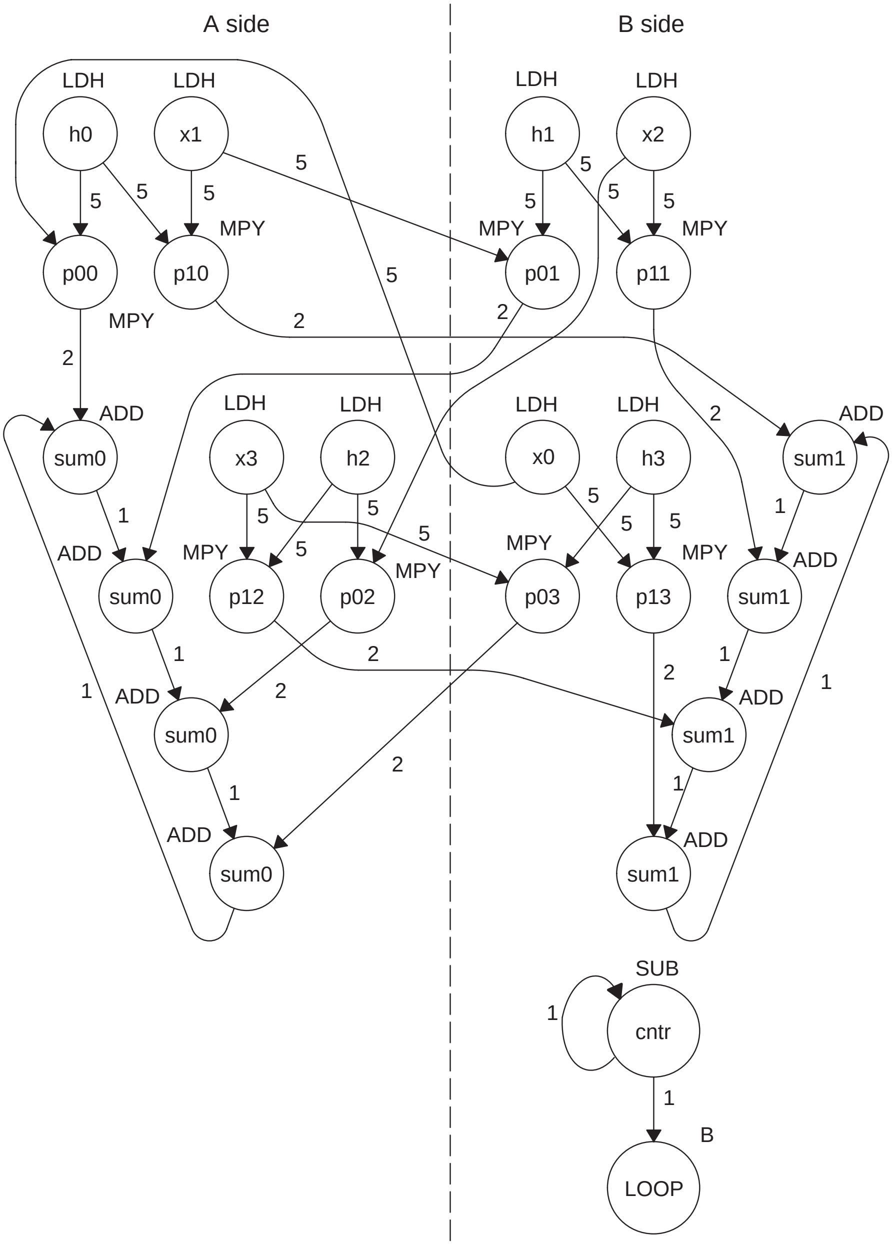
Example 6–67. Linear Assembly for Unrolled FIR Inner Loop

```
        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
```

6.12.4 Drawing a Dependency Graph

Figure 6–25 shows the dependency graph of the FIR filter with no memory hits.

Figure 6–25. Dependency Graph of FIR Filter (With No Memory Hits)



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

Example 6–68 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 6–68. 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 6–68. 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++[2],h0       ; h0 = h[i]
         MPY      .M1X     x0,h0,p00        ; x0 * h0
         MPY      .M1      x1,h0,p10        ; x1 * h0
         ADD      .L1      p00,sum0,sum0    ; sum0 += x0 * h0
         ADD      .L2X     p10,sum1,sum1    ; sum1 += x1 * h0

         LDH      .D2      *x++[2],x2       ; x2 = x[j+i+2]
         LDH      .D2      *h_1++[2],h1     ; h1 = h[i+1]
         MPY      .M2X     x1,h1,p01        ; x1 * h1
         MPY      .M2      x2,h1,p11        ; x2 * h1
         ADD      .L1X     p01,sum0,sum0    ; sum0 += x1 * h1
         ADD      .L2      p11,sum1,sum1    ; sum1 += x2 * h1

         LDH      .D1      *x_1++[2],x3     ; x3 = x[j+i+3]
         LDH      .D1      *h++[2],h2       ; h2 = h[i+2]
         MPY      .M1X     x2,h2,p02        ; x2 * h2
         MPY      .M1      x3,h2,p12        ; x3 * h2
         ADD      .L1      p02,sum0,sum0    ; sum0 += x2 * h2
         ADD      .L2X     p12,sum1,sum1    ; sum1 += x3 * h2

         LDH      .D2      *x++[2],x0       ; x0 = x[j+i+4]
         LDH      .D2      *h_1++[2],h3     ; h3 = h[i+3]
         MPY      .M2X     x3,h3,p03        ; x3 * h3
         MPY      .M2      x0,h3,p13        ; x0 * h3
         ADD      .L1X     p03,sum0,sum0    ; sum0 += x3 * h3
         ADD      .L2      p13,sum1,sum1    ; sum1 += x0 * h3


[ctr]    SUB      .S2      ctr,1,ctr        ; decrement loop counter
[ctr]    B        .S2      LOOP             ; branch to loop

         SHR      sum0,15,sum0              ; sum0 >> 15
         SHR      sum1,15,sum1              ; sum1 >> 15
         STH      sum0,*y++                 ; y[j] = sum0 >> 15
         STH      sum1,*y++                 ; y[j+1] = sum1 >> 15
         SUB      x,rstx,x                  ; reset x pointer to x[j]
         SUB      h_1,rsth,h_1              ; reset h pointer to h[0]
 [octr]  B        OUTLOOP                   ; branch to outer loop

         .endproc
```

6.12.6 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 ’C62x and ’C67x have only 32 registers. This would also work on the ’C64x which has 64 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 *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* for more information.

6.12.7 Determining the Minimum Iteration Interval

Based on Table 6–24, the minimum iteration interval for the FIR filter with no memory hits should be 4. An iteration interval of 4 means that two multiply/accumulates still execute per cycle.

Table 6–24. Resource Table for FIR Filter Code

(a) A side *(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

6.12.8 Final Assembly

Example 6–69 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.

6.12.9 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 6.13 addresses how to reduce this overhead by software pipelining the outer loop.

Table 6–25. Comparison of FIR Filter Code

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

Example 6–69. 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 6–69. 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
```

6.13 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.

6.13.1 Unrolled FIR Filter C Code

Example 6–70 shows the FIR filter C code after unrolling the inner loop (identical to Example 6–66 on page 6-122).

Example 6–70. 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;
         }
}
```

6.13.2 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 6–69 on page 6-129) contained 16 cycles of overhead to call the inner loop every time: ten cycles for the loop prolog and six cycles for the outer loop instructions and branching to the outer loop.

Most of this overhead can be reduced as follows:

- ❑ Put the outer loop and branch instructions in parallel with the prolog.
- ❑ Create an epilog to the inner loop.
- ❑ Put some outer loop instructions in parallel with the inner-loop epilog.

6.13.3 Final Assembly

Example 6–71 shows the final assembly for the FIR filter with a software-pipelined outer loop. Below the inner loop (starting on page 6-134), 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 6–71. 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 6–71. 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 6–71. 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
```

6.13.4 Comparing Performance

The improved cycle count for this loop is 2006 cycles: 50 ((7 × 4) + 6 + 6) + 6. The outer-loop overhead for this loop has been reduced from 16 to 8 (6 + 6 – 4); the –4 represents one iteration less for the inner-loop iteration (seven instead of eight).

Table 6–26. Comparison of FIR Filter Code

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

6.14 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.

6.14.1 Unrolled FIR Filter C Code

Example 6–72 shows the same unrolled FIR filter C code that used in the previous example.

Example 6–72. 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;
         }
}
```

6.14.2 Translating C Code to Linear Assembly (Inner Loop)

Example 6–73 shows a list of linear assembly for the inner loop of the FIR filter C code (identical to Example 6–67 on page 6-123).

Example 6–73. 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
```

6.14.3 Translating C Code to Linear Assembly (Outer Loop)

Example 6–74 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 6–74. 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
```

6.14.4 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 6–75 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 6–72 on page 6-136.

Example 6–75. 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;
        }
}
```

6.14.5 Translating C Code to Linear Assembly (Inner Loop)

Example 6–76 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 6–76. 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

         MPY      h01,x01,p00       ; p00 = h[i+0]*x[j+i+0]
[sctr]   ADD      p00,sum07,p00     ; sum0(p00) = p00 + sum0

         MPYH     h01,x01,p01       ; p01 = h[i+1]*x[j+i+1]
         ADD      p01,p00,sum01     ; sum0 += p01
```

Example 6–76. 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
```

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

Example 6–77 shows the linear assembly with functional units assigned. (As in Example 6–68 on page 6-125, symbolic names now have an A or B in front of them to signify the register file where they reside.) Although this allocation is one of many possibilities, one goal is to keep the 1X and 2X paths to a minimum. Even with this goal, you have five 2X paths and seven 1X paths.

One requirement that was assumed when the functional units were chosen was that all the sum0 values reside on the same side (A in this case) and all the sum1 values reside on the other side (B). Because you are scheduling eight accumulates for both sum0 and sum1 in an 8-cycle loop, each ADD must be scheduled immediately following the previous ADD. Therefore, it is undesirable for any sum0 ADDs to use the same functional units as sum1 ADDs.

One MV instruction was added to get x01 on the B side for the MPYLH p10 instruction.

Example 6–77. 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 6–77. 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

         MPY      .M1      h01,x01,p00        ; p00 = h[i+0]*x[j+i+0]
[sctr]   ADD      .L1      p00,sum07,p00      ; sum0(p00) = p00 + sum0

         MPYH     .M1      h01,x01,p01        ; p01 = h[i+1]*x[j+i+1]
         ADD      .L1      p01,p00,sum01      ; sum0 += p01

         MPY      .M2      h23,x23,p02        ; p02 = h[i+2]*x[j+i+2]
         ADD      .L1X     p02,sum01,sum02    ; sum0 += p02

         MPYH     .M2      h23,x23,p03        ; p03 = h[i+3]*x[j+i+3]
         ADD      .L1X     p03,sum02,sum03    ; sum0 += p03

         MPY      .M1      h45,x45,p04        ; p04 = h[i+4]*x[j+i+4]
         ADD      .L1      p04,sum03,sum04    ; sum0 += p04

         MPYH     .M1      h45,x45,p05        ; p05 = h[i+5]*x[j+i+5]
         ADD      .L1      p05,sum04,sum05    ; sum0 += p05
```

Example 6–77. 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
```

6.14.7 Determining the Minimum Iteration Interval

Based on Table 6–27, the minimum iteration interval is 8. An iteration interval of 8 means that two multiply-accumulates per cycle are still executing.

Table 6–27. Resource Table for FIR Filter Code

(a) A side *(b) B side*

Unit(s)	Total/Unit	Unit(s)	Total/Unit
.M1	8	.M2	8
.S1	7	.S2	6
.D1	5	.D2	6
.L1	8	.L2	8
Total non-.M units	20	Total non-.M units	20
1X paths	7	2X paths	5

6.14.8 Final Assembly

Example 6–78 shows the final assembly for the FIR filter with the outer loop conditionally executing in parallel with the inner loop.

Example 6–78. 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

         MPYLH    .M2X     A8,B8,B4          ; p10 = h[i+0]*x[j+i+1]
||[!A1]  SUB      .S2      B2,B5,B2          ; reset h ptr
||       MPYHL    .M1X     A8,B9,A14         ; p11 = h[i+1]*x[j+i+2]

         MPY      .M1      A8,A10,A7         ; p00 = h[i+0]*x[j+i+0]
||       MPYLH    .M2      B7,B9,B13         ; p12 = h[i+2]*x[j+i+3]
||[A2]   SUB      .S1      A2,1,A2           ; dec store lp cntr
||       ZERO     .L2      B11               ; zero out initial accumulator

  [!A2]  SHR      .S2      B11,15,B11        ; (Bsum1 >> 15)
||       MPY      .M2      B7,B9,B9          ; p02 = h[i+2]*x[j+i+2]
||       MPYH     .M1      A8,A10,A10        ; p01 = h[i+1]*x[j+i+1]
||[A2]   ADD      .L2      B4,B11,B4         ; sum1(p10) = p10 + sum1
||       LDW      .D1      *A4++[2],B9       ;* x[j+i+2] & x[j+i+3]
||       LDW      .D2      *B1++[2],A10      ;* x[j+i+0] & x[j+i+1]
||       ZERO     .L1      A10               ; zero out initial accumulator
```

Example 6–78. 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]
||       MPYLH    .M1      A9,A11,A10        ; p14 = h[i+4]*x[j+i+5]
||       ADD      .L2      B13,B7,B7         ; sum1 +=  p12
||       LDW      .D2      *B1++[2],A11      ;* x[j+i+4] & x[j+i+5]
||       LDW      .D1      *A4++[2],B10      ;* x[j+i+6] & x[j+i+7]
||[A1]   SUB      .S1      A1,1,A1           ;* dec pointer reset lp cntr

  [B0]   B        .S2      LOOP              ; Branch outer loop
||       MPY      .M1      A9,A11,A11        ; p04 = h[i+4]*x[j+i+4]
||       ADD      .L1X     B9,A13,A13        ; sum0 += p02
||       MPYLH    .M2      B8,B10,B13        ; p16 = h[i+6]*x[j+i+7]
||       ADD      .L2X     A10,B7,B7         ; sum1 += p13
||       LDW      .D1      *A0++[2],A9       ;* h[i+4] & h[i+5]
||       LDW      .D2      *B2++[2],B8       ;* h[i+6] & h[i+7]
||[!A1]  SUB      .S1      A4,A3,A4          ;* reset x ptr

         MPY      .M2      B8,B10,B11        ; p06 = h[i+6]*x[j+i+6]
||       MPYH     .M1      A9,A11,A11        ; p05 = h[i+5]*x[j+i+5]
||       ADD      .L1X     B13,A13,A9        ; sum0 += p03
||       ADD      .L2X     A10,B7,B7         ; sum1 += p14
||[!A1]  SUB      .S2      B1,B14,B1         ;* reset x ptr
||[!A1]  SUB      .S1      A0,A5,A0          ;* reset h ptr
||       LDH      .D2      *B1,A8            ;* x[j+i+8]

  [!A2]  MVK      .S1      4,A2              ; reset store lp cntr
||       MPYH     .M2      B8,B10,B13        ; p07 = h[i+7]*x[j+i+7]
||       ADD      .L1      A11,A9,A9         ; sum0 += p04
||       MPYHL    .M1X     B8,A8,A9          ; p17 = h[i+7]*x[j+i+8]
||       ADD      .S2      B12,B7,B10        ; sum1 += p15
||[!A2]  STH      .D2      B11,*B6++[2]      ; y[j+1] = (Bsum1 >> 15)
||[!A2]  STH      .D1      A12,*A6++[2]      ; y[j] = (Asum0 >> 15)
||       ADD      .L2X     A10,0,B8          ;* move to other reg file

         ADD      .L1      A11,A9,A12        ; sum0 += p05
||       ADD      .L2      B13,B10,B8        ; sum1 += p16
||       MPYLH    .M2X     A8,B8,B4          ;* p10 = h[i+0]*x[j+i+1]
||[!A1]  MVK      .S1      4,A1              ;* reset pointer reset lp cntr
||[!A1]  SUB      .S2      B2,B5,B2          ;* reset h ptr
||       MPYHL    .M1X     A8,B9,A14         ;* p11 = h[i+1]*x[j+i+2]
```

Example 6–78. 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)
```

6.14.9 Comparing Performance

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

Table 6–28. Comparison of FIR Filter Code

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

Chapter 7

Interrupts

This chapter describes interrupts from a software-programming point of view. A description of single and multiple register assignment is included, followed by code generation of interruptible code and finally, descriptions of interrupt subroutines.

Topic **Page**

7.1 Overview of Interrupts

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 ’C6000 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 *TMS320C6000 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 ’C6000 code generation tools to satisfy different requirements
❑ Interrupt subroutines

7.2 Single Assignment vs. Multiple Assignment

Register allocation on the ’C6000 can be classified as either single assignment or multiple assignment. Single assignment code is interruptible; multiple assignment is not interruptible. This section discusses the differences between each and explains why only single assignment is interruptible.

Example 7–1 shows multiple assignment code. The term multiple assignment means that a particular register has been assigned with more than one value (in this case 2 values). On cycle 4, at the beginning of the ADD instruction, register A1 is assigned to two different values. One value, written by the SUB instruction on cycle 1, already resides in the register. The second value is called an *in-flight* value and is assigned by the LDW instruction on cycle 2. Because the LDW instruction does not actually write a value into register A1 until the end of cycle 6, the assignment is considered in-flight.

In-flight operations cause code to be uninterruptible due to unpredictability. Take, for example, the case where an interrupt is taken on cycle 3. At this point, all instructions which have begun execution are allowed to complete and no new instructions execute. So, 3 cycles after the interrupt is taken on cycle 3, the LDW instruction writes to A1. After the interrupt service routine has been processed, program execution continues on cycle 4 with the ADD instruction. In this case, the ADD reads register A1 and will be reading the result of the LDW, whereas normally the result of the SUB should be read. This unpredictability means that in order to ensure correct operation, multiple assignment code should not be interrupted and is thus, considered uninterruptible.

Example 7–1. Code With Multiple Assignment of A1

```
cycle
   1      SUB    .S1    A4,A5,A1  ; writes to A1 in single cycle
   2      LDW    .D1    *A0,A1    ; writes to A1 after 4 delay slots
   3      NOP
   4      ADD    .L1    A1,A2,A3  ; uses old A1 (result of SUB)
   5–6    NOP           2
   7      MPY    .M1    A1,A4,A5  ; uses new A1 (result of LDW)
```

Example 7–2 shows the same code with a new register allocation to produce single assignment code. Now the LDW assigns a value to register A6 instead of A1. Now, regardless of whether an interrupt is taken or not, A1 maintains the value written by the SUB instruction because LDW now writes to A6. Because there are no in-flight registers that are read before an in-flight instruction completes, this code is interruptible.

Example 7–2. Code Using Single Assignment

```
cycle
   1  SUB    .S1    A4,A5,A1  ; writes to A1 in single cycle
   2  LDW    .D1    *A0,A6    ; writes to A1 after 4 delay slots
   3  NOP
   4  ADD    .L1    A1,A2,A3  ; uses old A1 (result of SUB)
   5–6NOP           2
   7  MPY    .M1    A6,A4,A5  ; uses new A1 (result of LDW)
```

Both examples involve exactly the same schedule of instructions. The only difference is the register allocation. The single assignment register allocation, as shown in Example 7–2, can result in higher register pressure (Example 7–2 uses one more register than Example 7–1).

The next section describes how to generate interruptible and non-interruptible code with the ’C6000 code generation tools.

7.3 Interruptible Loops

Even if code employs single assignment, it may not be interruptible in a loop. Because the delay slots of all branch operations are protected from interrupts in hardware, all interrupts remain pending as long as the CPU has a pending branch. Since the branch instruction on the ’C6000 has 5 delay slots, loops smaller than 6 cycles always have a pending branch. For this reason, all loops smaller than 6 cycles are uninterruptible.

There are two options for making a loop with an iteration interval less than 6 interruptible.

1) Simply slow down the loop and force an iteration interval of 6 cycles. This is not always desirable since there will be a performance degradation.

2) Unroll the loop until an iteration interval of 6 or greater is achieved. This ensures at least the same performance level and in some cases can improve performance (see section 6.9, *Loop Unrolling* and section 7.4.4, *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 ’C6000 code generation tools.

7.4 Interruptible Code Generation

The ’C6000 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 7.4.4.)

- ❑ 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 *TMS320C6000 Optimizing C/C++ Compiler User’s Guide*.

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

At this level, 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);
```

7.4.2 Level 1 – Specified Code Interruptible at All Times

At this level, the compiler employs single assignment everywhere and never produces a loop of less than 6 cycles. The command line option –mi1 can be used for an entire module and the following pragma can be used to force this level on a particular function:

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, 1);
```

7.4.3 Level 2 – Specified Code Interruptible Within Threshold Cycles

The compiler will disable interrupts around loops if the specified threshold number is not exceeded. In other words, the user can specify a threshold, or maximum interrupt delay, that allows the compiler to use multiple assignment in loops that do not exceed this threshold. The code outside of loops can have interrupts disabled and also use multiple assignment as long as the threshold of uninterruptible cycles is not exceeded. If the compiler cannot determine the loop count of a loop, then it assumes the threshold is exceeded and will generate an interruptible loop.

The command line option –mi (threshold) can be used for an entire module and the following pragma can be used to specify a threshold for a particular function.

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);
```

7.4.4 Getting the Most Performance Out of Interruptible Code

As stated in Chapter 4 and Chapter 7, the .trip directive and the MUST_ITERATE pragma 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 7–3.

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 7–3. Dot Product With _nassert Guaranteeing Minimum Trip Count

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

With the MUST_ITERATE pragma, 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 7–4 adds this information to the _nassert statement as follows:

Example 7–4. Dot Product With _nassert Guaranteeing Trip Count Range

```
int dot_prod(short *a, short *b, int n)
{
   int i, sum = 0;
   #pragma MUST_ITERATE (20,50);
   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 7–5 shows where the factor option in .trip and using the third argument of the MUST_ITERATE pragma are useful. For more information, see section 3.4.3.4, *Loop Unrolling*.

Example 7–5. Dot Product With MUST_ITERATE Pragma Guaranteeing Trip Count Range and Factor of 2

```
int dot_prod(short *a, short *b, int n)
{
   int i, sum = 0;
   #pragma MUST_ITERATE (20,50,2);
   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 7–6 shows a dot product loop that will execute a factor of 4 between 16 and 48 times.

Example 7–6. Dot Product With MUST_ITERATE Pragma Guaranteeing Trip Count Range and Factor of 4

```
int dot_prod(short *a, short *b, int n)
{
   int i, sum = 0;
   #pragma MUST_ITERATE (16,48,4);
   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.

7.5 Interrupt Subroutines

The interrupt subroutine (ISR) is simply the routine, or function, that is called by an interrupt. The 'C6000 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/C++ compiler, but must be done manually when writing hand-coded assembly.

7.5.1 ISR with the C/C++ Compiler

The C/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/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/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 for 'C62x and 'C67x, and A0–A31, B0–B31 for 'C64x) 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/C++ functions

❑ A function cannot return a value and thus, must be declared void.

See the section on *Register Conventions* in the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information on standard function calling conventions.

7.5.2 ISR with Hand-Coded Assembly

When writing an ISR by hand, it is necessary to handle the same tasks the C/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/C++ compiler uses B15.)
- If another C routine is called from the ISR (with an assembly branch instruction to the _c_func_name label) then all registers must be saved to the stack on entry.
- A B IRP instruction must be used to return from the routine. If this is the NMI ISR, a B NRP must be used instead.
- An NOP 4 is required after the last LDW in this case to ensure that B0 is restored before returning from the interrupt.

Example 7–7. 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
```

7.5.3 Nested Interrupts

Sometimes it is desirable to allow higher priority interrupts to interrupt lower priority ISRs. To allow nested interrupts to occur, you must first save the IRP, IER, and CSR to a register which is not being used or to or some other memory location (usually the stack). Once these have been saved, you can reenable the appropriate interrupts. This involves resetting the GIE bit and then doing any necessary modifications to the IER, providing only certain interrupts are allowed to interrupt the particular ISR. On return from the ISR, the original values of the IRP, IER, and CSR must be restored.

Example 7–8. 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
   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
   MVC    A0,CSR       ; restore original CSR
                       ; to complete before branch is taken
```

Chapter 8

’C64x Programming Considerations

This chapter covers material specific to the TMS320C64x series of DSPs. It builds on the material presented elsewhere in this book, whith additional information specific to the VelociTI.2 extensions that the ’C64x provides.

Before reading this chapter, familiarize yourself with the programming concepts presented earlier for the entire C6000 family, as these concepts also apply to the ’C64x.

The sample code that is used in this chapter is included on the Code Generation Tools and Code Composer Studio 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.

Topic **Page**

8.1 Overview of ’C64x Architectural Enhancements

The ’C64x is a fixed-point digital signal processor (DSP) and is the first DSP to add VelociTI.2 extensions to the existing high-performance VelociTI architecture. VelociTI.2 extensions provide the following features:

❑ Greater scheduling flexibility for existing instructions

❑ Greater memory bandwidth with double-word load and store instructions

❑ Support for packed 8-bit and 16-bit data types

❑ Support for non-aligned memory accesses

❑ Special purpose instructions for communications-oriented applications

8.1.1 Improved Scheduling Flexibility

The ’C64x improves scheduling flexibility using three different methods. First, it makes several existing instructions available on a larger number of units. Second, it adds cross-path access to the D-unit so that arithmetic and logical operations which use a cross-path may be scheduled there. Finally, it removes a number of scheduling restrictions associated with 40-bit operations, allowing more flexible scheduling of high-precision code.

8.1.2 Greater Memory Bandwidth

The ’C64x provides double-word load and store instructions (LDDW and STDW) which can access 64 bits of data at a time. Up to two double-word load or store instructions can be issued every cycle. This provides a peak bandwith of 128 bits per cycle to on-chip memory.

8.1.3 Support for Packed Data Types

The ’C64x builds on the ’C62x’s existing support for packed data types by improving support for packed signed 16-bit data and adding new support for packed unsigned 8-bit data. Packed data types are supported using new pack/unpack, logical, arithmetic and multiply instructions for manipulating packed data.

Packed data types store multiple pieces of data within a single 32-bit register. Pack and unpack instructions provide a method for reordering this packed data, and for converting between packed formats. Shift and merge instructions (SHLMB and SHRMB) also provide a means for reordering packed 8-bit data.

New arithmetic instructions include standard addition, subtraction, and comparison, as well as advanced operations such as minimum, maximum, and average. New packed multiply instructions provide support for both standard multiplies, as well as rounded multiplies and dot products. With packed data types, a single instruction can operate on two 16-bit quantities or four 8-bit quantities at once.

8.1.4 Non-aligned Memory Accesses

In order to capitalize on its memory and processing bandwidth, the ’C64x provides support for non-aligned memory accesses. Non-aligned memory accesses provide a method for accessing packed data types without the restrictions imposed by 32-bit or 64-bit alignment boundaries. The ’C64x can access up to 64 bits per cycle at any byte boundary with non-aligned load and store instructions (LDNW, LDNDW, STNW, and STNDW).

8.1.5 Additional Specialized Instructions

The ’C64x also provides a number of new bit-manipulation and other specialized instructions for improving performance on bit-oriented algorithms. These instructions are designed to improve performance on error correction, encryption, and other bit-level algorithms. Instructions in this category include BITC4, BITR, ROTL, SHFL, and DEAL. See the *TMS320C6000 CPU and Instruction Set User’s Guide* for more details on these and related instructions.

8.2 Packed-Data Processing on the ’C64x

8.2.1 Introduction to Packed Data Processing Techniques

Packed-data processing is a type of processing where a single instruction applies the same operation to multiple independent pieces of data. For example, the ADD2 instruction performs two independent 16-bit additions between two pairs of 16-bit values. This produces a pair of 16-bit results. In this case, a single instruction, ADD2, is operating on multiple sets of data, the two independent pairs of addends.

Packed-data processing is a powerful method for exploiting the inherent parallelism in signal processing and other calculation-intensive code, while retaining dense code. Many signal processing functions apply the same sets of operations to many elements of data. Generally, these operations are independent of each other. Packed-data processing allows the programmer to capitalize on this by operating on multiple pieces of data with a single compact stream of instructions. This saves code size and dramatically boosts performance.

The ’C64x provides a rich family of instructions which are designed to work with packed-data processing. At the core of this paradigm are packed data types, which are designed to store multiple data elements in a single machine register. Packed-data processing instructions are built around these data types to provide a flexible, powerful, programming environment.

Note:

Although ’C6000 family supports both big-endian and little-endian operation, the examples and figures in this section will focus on little endian operation only. The packed-data processing extensions that the ’C64x provides will operate in either big- or little-endian mode, and will perform identically on values stored in the register file. However, accesses to memory behave differently in big-endian mode.

8.2.2 Packed Data Types

Packed data types are the cornerstone of ’C64x packed-data processing support. Each packed data type packs multiple elements into a single 32-bit general purpose register. Table 8–1 below lists the packed data types that the ’C64x supports. The greatest amount of support is provided for unsigned 8-bit and signed 16-bit values.

Table 8–1. Packed data types

Element Size	Signed/Unsigned	Elements in 32-bit word	Element type	Level of support
8 bits	unsigned	4	unsigned char	high
16 bits	signed	2	short	high
8 bits	signed	4	char	limited
16 bits	unsigned	2	unsigned short	limited

8.2.3 Storing Multiple Elements in a Single Register

Packed data types can be visualized as 8-bit or 16-bit partitions inside the larger 32-bit register. These partitions are merely logical partitions. As with all ’C64x instructions, instructions which operate on packed data types operate directly on the 64 general purpose registers in the register file. There are no special packed data registers. How data in a register is interpreted is determined entirely by the instruction that is using the data. Figure 8–1 and Figure 8–2 illustrate how four bytes and two half-words are packed into a single word.

Figure 8–1. Four Bytes Packed Into a Single General Purpose Register.

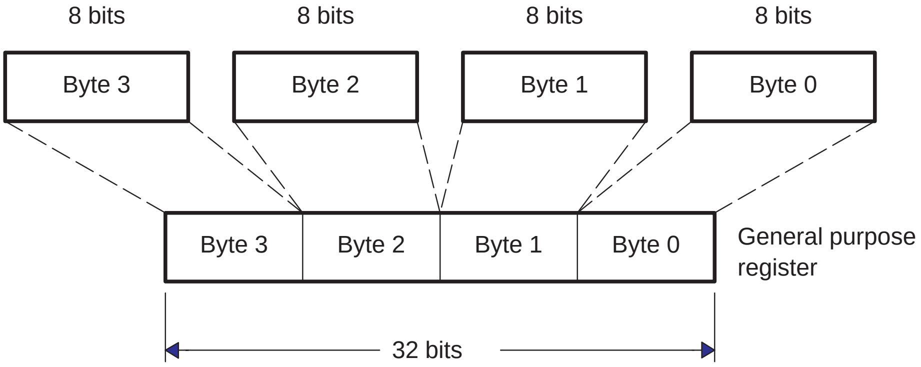
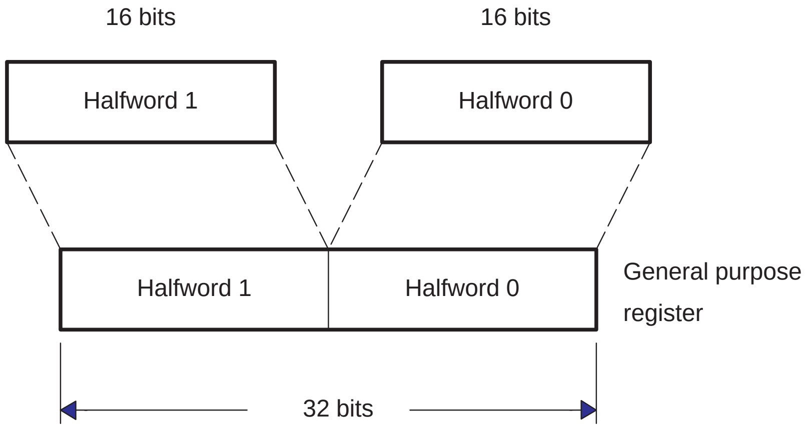


Figure 8–2. Two Half–Words Packed Into a Single General Purpose Register.



Notice that there is no distinction between signed or unsigned data made in Figure 8–1 and Figure 8–2. This is due to the fact that signed and unsigned data are packed identically within the register. This allows the instructions which are not sensitive to sign bits (such as adds and subtracts) to operate on signed and unsigned data equally. The distinction between signed and unsigned comes into play primarily when performing multiplication, comparing elements, and unpacking data (since either sign or zero extension must be performed).

Table 8–2 provides a summary of the operations that the ’C64x provides on packed data types, and whether signed or unsigned types are supported. Instructions which were not specifically designed for packed data types can also be used to move, store, unpack, or repack data.

Table 8–2. Supported Operations on Packed Data Types

Operation	Support for 8-bit		Support for 16-bit		Notes
	Signed	**Unsigned**	**Signed**	**Unsigned**	
ADD/SUB	Yes	Yes	Yes	Yes	
Saturated ADD		Yes	Yes	*	
Booleans	Yes	Yes	Yes	Yes	Uses generic boolean instructions
Shifts			Yes	Yes	Right-shift only
Multiply	*	Yes	Yes	*	
Dot Product	*	Yes	Yes	*	
Max/Min/ Compare		Yes	Yes		CMPEQ works with signed or unsigned
Pack	Yes	Yes	Yes	Yes	
Unpack		Yes	Yes	Yes	See Table 8–4 for 16-bit unpacks

* = Only ‘signed-by-unsigned’ support in these categories.

8.2.4 Packing and Unpacking Data

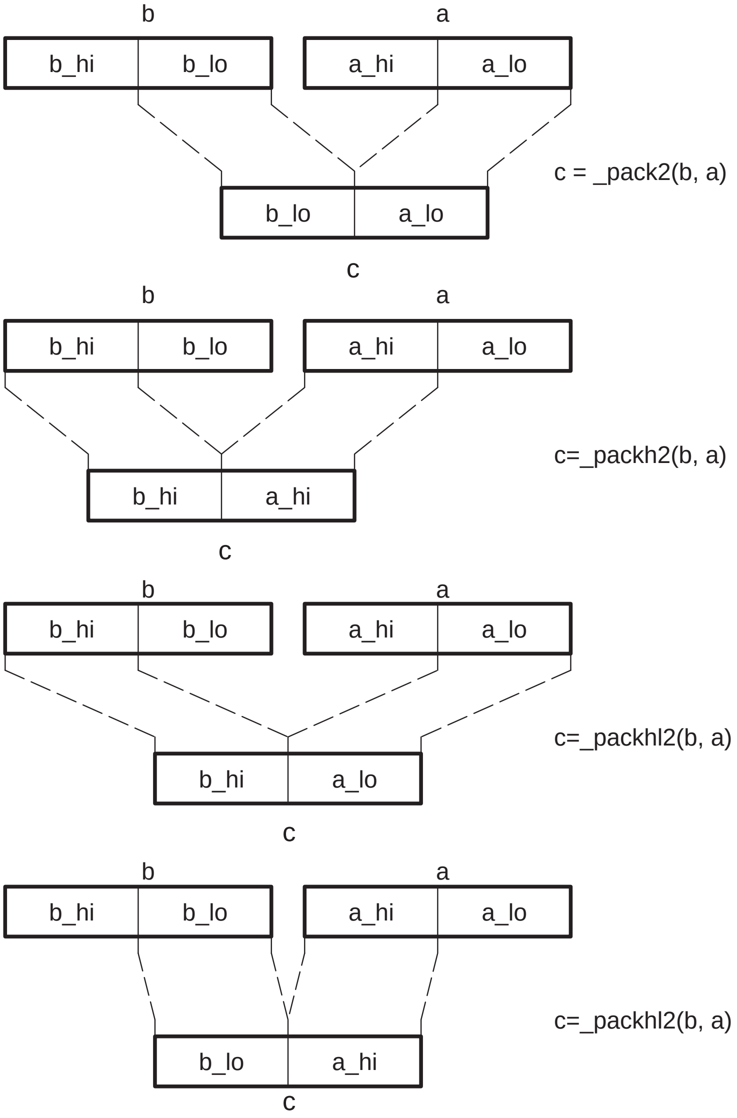
The ’C64x provides a family of packing and unpacking instructions which are used for converting between various packed and non-packed data types, as well as for manipulating elements within a packed type. Table 8–4 lists the available packing instructions and uses.

Table 8–3. Instructions for Manipulating Packed Data Types

Mnemonic	Intrinsic	Typical Uses With Packed Data
PACK2	_pack2	Packing 16-bit portions of 32-bit quantities.
PACKH2	_packh2	Rearranging packed 16-bit quantities.
PACKHL2	_packhl2	Rearranging pairs of 16-bit quantities.
PACKLH2	_packlh2	
SPACK2	_spack2	Saturating 32-bit quantities down to signed 16-bit values, packing them together.
SHR	(n/a)	Unpacking 16-bit values into 32-bit values
SHRU	(n/a)	
EXT	_ext	
EXTU	_extu	
PACKH4	_packh4	Unpacking 16-bit intermediate results into 8-bit final results.
PACKL4	_packl4	De-interleaving packed 8-bit values.
UNPKHU4	_unpkhu4	Unpacking unsigned 8-bit data into 16-bits.
UNPKLU4	_unpklu4	Preparing 8-bit data to be interleaved.
SPACKU4	_spacku4	Saturating 16-bit quantities down to unsigned 8-bit values, packing them together.
SHLMB	_shlmb	Rearranging packed 8-bit quantities
SHRMB	_shrmb	
SWAP4	_swap4	
ROTL	_rotl	

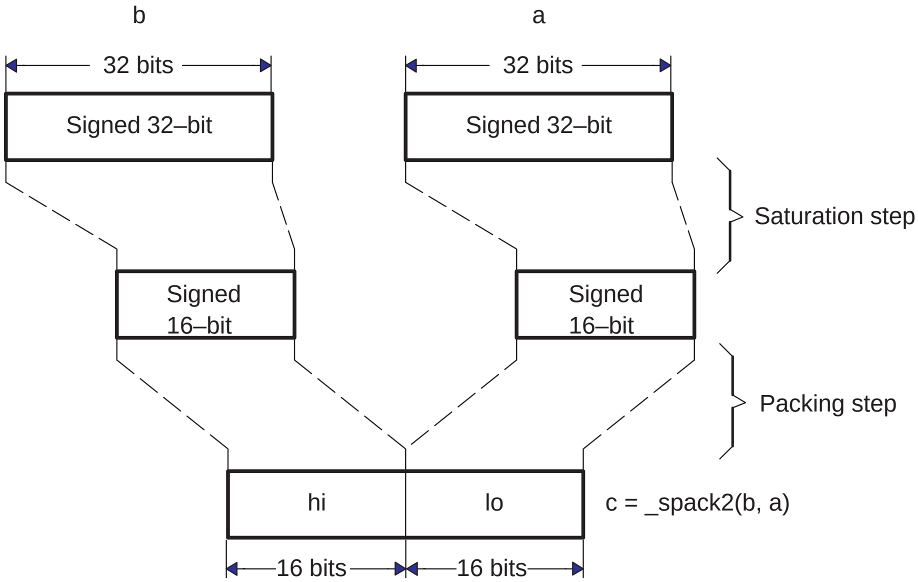
The _packXX2 group of intrinsics work by extracting selected half-words from two 32-bit registers, returning the results packed into a 32-bit word. This is primarily useful for manipulating packed 16-bit data, although they may be used for manipulating pairs of packed 8-bit quantites. Figure 8–3 illustrates the four _packXX2() intrinsics, _pack2(), _packlh2(), _packhl2(), and _packh2(). (The *l* and the *h* in the name refers to which half of each 32-bit input is being copied to the output, similar to how the _mpyXX() intrisics are named.)

Figure 8–3. Graphical Representation of _packXX2 Intrinsics



The saturating pack intrinsic, _spack2, is closely related to the _pack2 intrinsic. The main difference is that the saturating pack first saturates the signed 32-bit source values to signed 16-bit quantities, and then packs these results into a single 32-bit word. This makes it easier to support applications which generate some intermediate 32-bit results, but need a signed 16-bit result at the end. Figure 8–4 shows _spack2’s operation graphically.

Figure 8–4. Graphical Representation of _spack2



Notice that there are no special unpack operations for 16-bit data. Instead, the normal 32-bit right-shifts and extract operations can be used to unpack 16-bit elements into 32-bit quantities. Table 8–4 describes how to unpack signed and unsigned 16-bit quantities.

Table 8–4. Unpacking Packed 16-bit Quantities to 32-bit Values

Type	Position	C code	Assembly code
Signed 16-bit	Upper half	`dst = ((signed)src) >> 16;`	`SHR  src, 16, dst`
	Lower half	`dst = _ext(src, 16, 16);`	`EXT  src, 16,16, dst`
Unsigned 16-bit	Upper half	`dst = ((unsigned)src)>>16;`	`SHRU src, 16, dst`
	Lower half	`dst = _ext (src, 16, 16);`	`EXTU src, 16,16, dst`

For 8-bit data, the 'C64x provides the _packX4, _spacku4, and _unpkX4 intrinsics for packing and unpacking data. The operation of these intrinsics is illustrated in Figure 8–5 and Figure 8–6. These intrinsics are geared around converting between 8-bit and 16-bit packed data types. To convert between 32-bit and 8-bit values, an intermediate step at 16-bit precision is required.

Figure 8–5. Graphical Representation of 8–bit Packs (_packX4 and _spacku4)

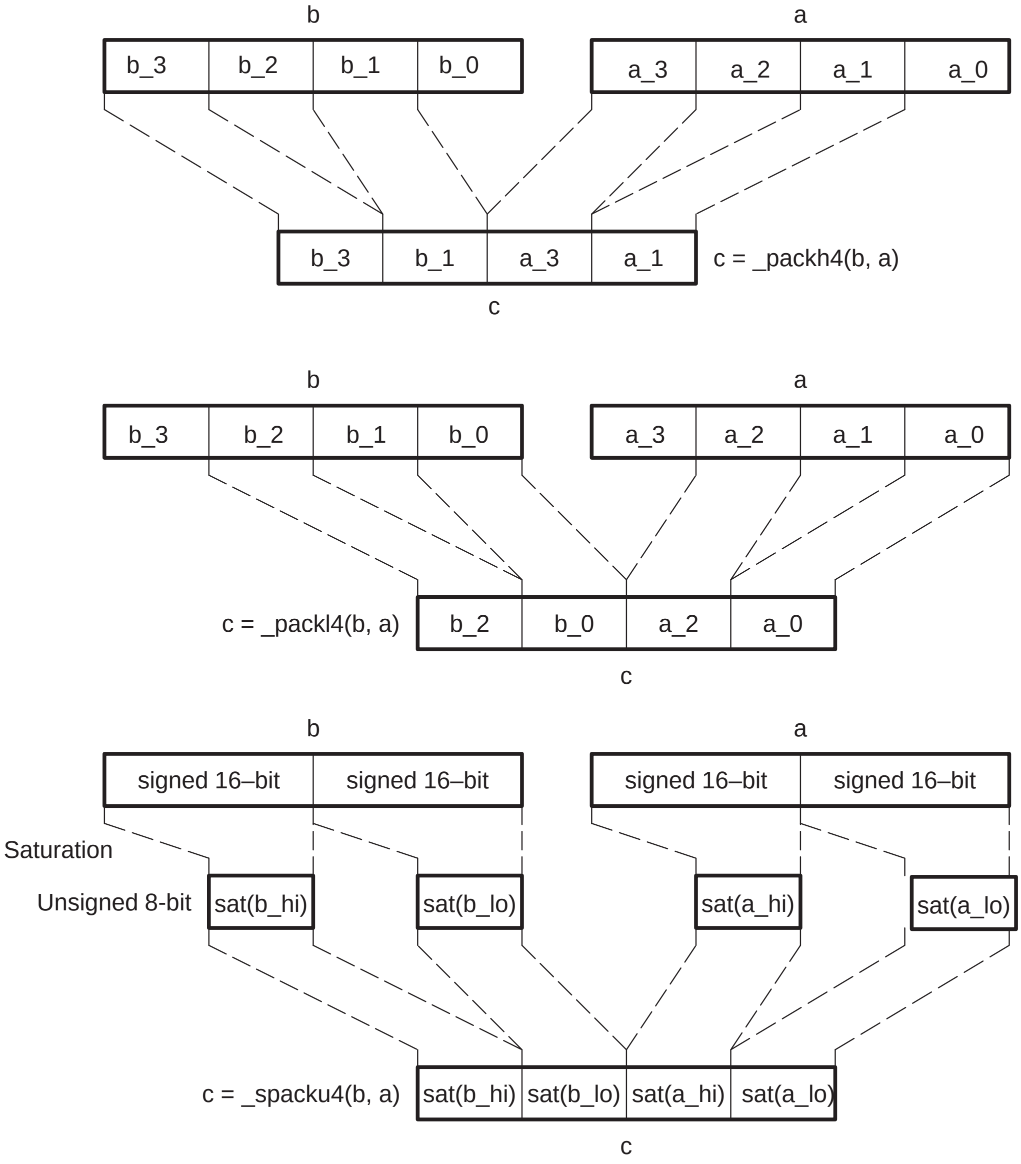
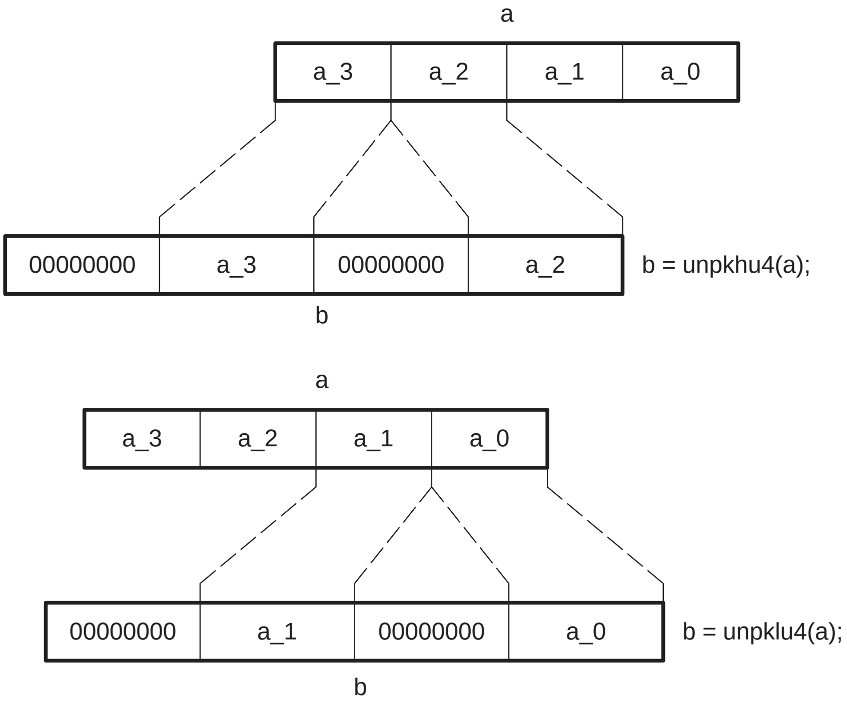
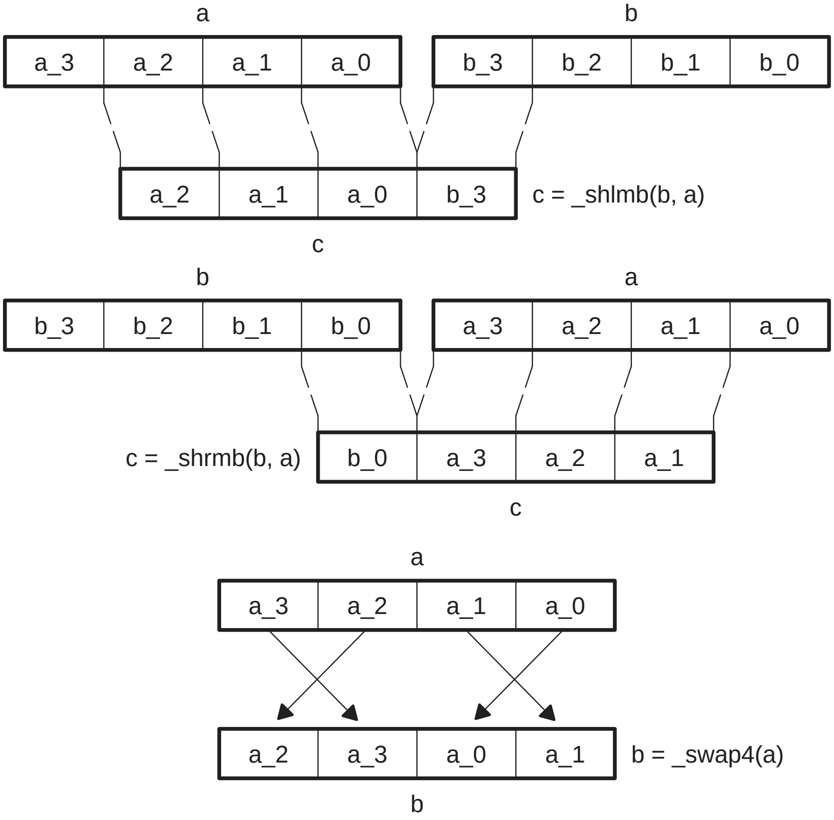


Figure 8–6. Graphical Representation of 8–bit Unpacks (_unpkXu4)



The ’C64x also provides a handful of additional byte-manipulating operations that have proven useful in various algorithms. These operations are neither packs nor unpacks, but rather shuffle bytes within a word. Uses include convolution-oriented algorithms, complex number arithmetic, and so on. Operations in this category include the intrinsics _shlmb, _shrmb, _swap4, and _rotl. The first three in this list are illustrated in Figure 8–7.

Figure 8–7. Graphical Representation of (_shlmb, _shrmb, and _swap4)



8.2.5 Optimizing for Packed Data Processing

The ’C64x supports two basic forms of packed-data optimization, namely vectorization and macro operations.

Vectorization works by applying the exact same simple operations to several elements of data simultaneously. Kernels such as vector sum and vector multiply, shown in Example 8–1 and Example 8–2, exemplify this sort of computation.

Example 8–1. Vector Sum

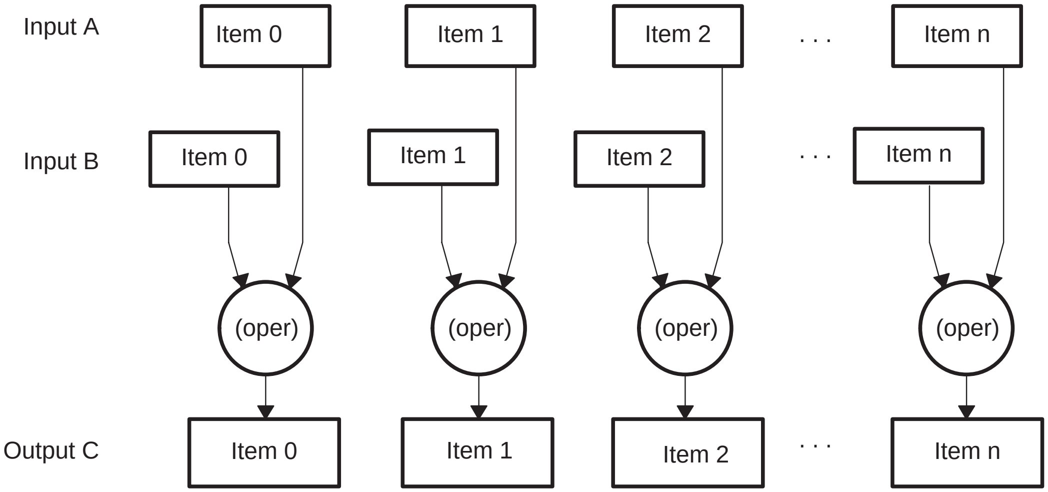
```
void vec_sum(const short *restrict a, const short *restrict b,
       short *restrict c, int len)
{
   int i;

   for (i = 0; i < len; i++)
      c[i] = b[i] + a[i];
}
```

Example 8–2. Vector Multiply

```
void vec_mpy(const short *restrict a, const short *restrict b,
       short *restrict c, int len, int shift)
{
   int i;

   for (i = 0; i < len; i++)
      c[i] = (b[i] * a[i]) >> shift;
}
```

This type of code is refered to as vector code because each of the input arrays is a vector of elements, and the same operation is being applied to each element. Pure vector code has no computation between adjacent elements when calculating results. Also, input and output arrays tend to have the same number of elements. Figure 8–8 illustrates the genenal form of a simple vector operation that operates on inputs from arrays A and B, producing an output, C (such as our Vector Sum and Vector Multiply kernels above perform).

Figure 8–8. Graphical Representation of a Simple Vector Operation



Although pure vector algorithms exist, most applications do not consist purely of vector operations as simple as the one shown above. More commonly, an algorithm will have portions which behave as a vector algorithm, and portions which do not. These portions of the code are addressed by another packed-data processing techniques.

The second form of packed data optimization involves combining multiple operations on packed data into a single, larger operation referred to here as a macro operation. This can be very similar to vectorization, but the key difference is that there is significant mathematical interaction between adjacent elements. Simple examples include dot product operations and complex multiplies, as shown in Example 8–3 and Example 8–4.

Example 8–3. Dot Product

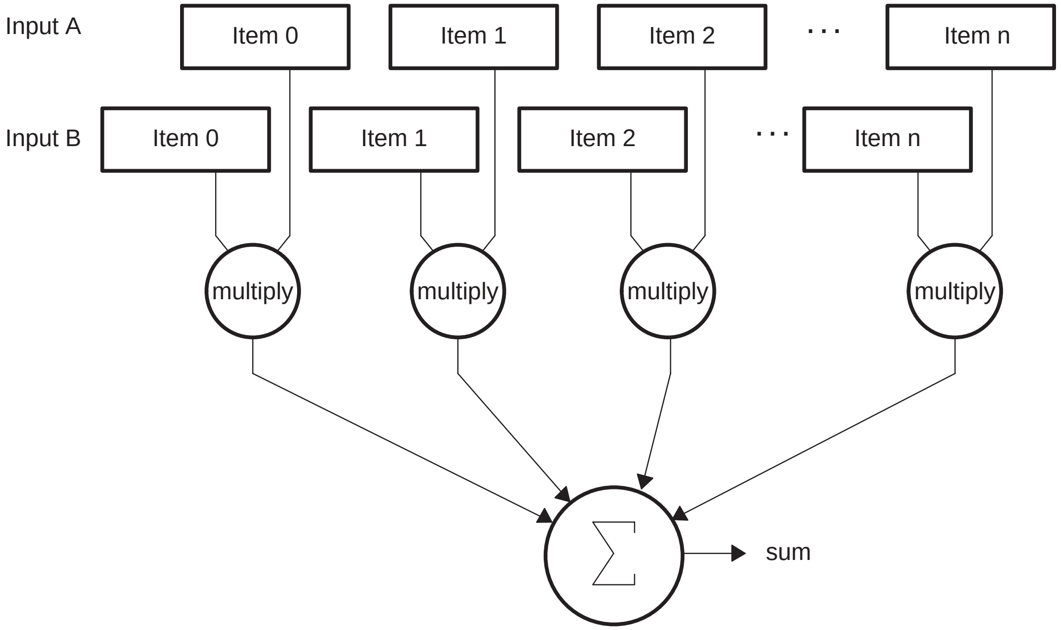
```
int dot_prod(const short *restrict a, const short *restrict b, int len)
{
   int i;
   int sum = 0;

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

   return sum;
}
```

Example 8–4. Vector Complex Multiply

```
void vec_cx_mpy(const short *restrict a, const short *restrict b,
          short *restrict c)
{
   int j;

   for (i = j = 0; i < len; i++, j += 2)
      {
      /* Real components are at even offsets, and imaginary components
      are at odd offsets within each array. */
      c[j+0] = (a[j+0] * b[j+0] – a[j+1] * b[j+1]) >> 16;
      c[j+1] = (a[j+0] * b[j+1] + a[j+1] * b[j+0]) >> 16;
      }
}
```

The data flow for the dot product is shown in Figure 8–9. Notice how this is similar to the vector sum in how the array elements are brought in, but different in how the final result is tabulated.

Figure 8–9. Graphical Representation of Dot Product

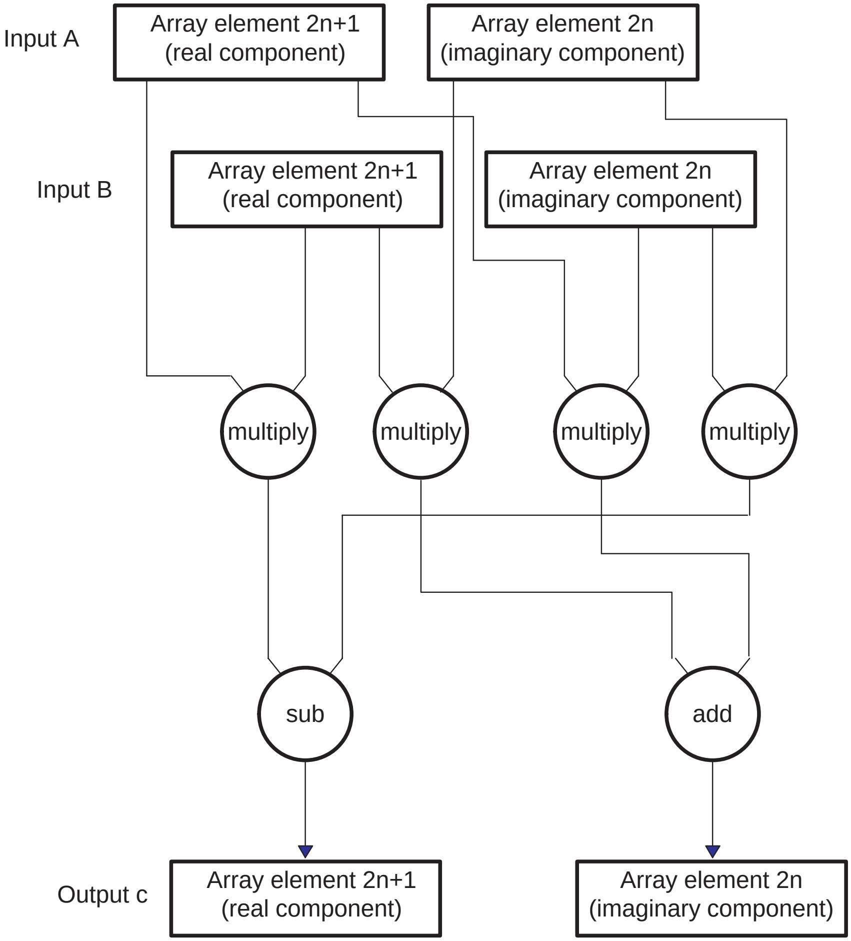


As you can see, this does not fit the pure vector model presented in Example 8–3. The Vector Complex Multiply also does not fit the pure vector model, but for different reasons.

Mathematically, the vector complex multiply is a pure vector operation performed on vectors of complex numbers, as its name implies. However, it is not, in implementation, because neither the language type nor the hardware itself directly supports a complex multiply as a single operation.

The complex multiply is built up from a number of real multiplies, with the complex numbers stored in arrays of interleaved real and imaginary values. As a result, the code requires a mix of vector and non–vector approaches to be optimized. Figure 8–10 illustrates the operations that are performed on a single iteration of the loop. As you can see, there is a lot going on in this loop.

Figure 8–10. Graphical Representation of a Single Iteration of Vector Complex Multiply.


Input A
Array element 2n+1 (real component)
Array element 2n (imaginary component)
Input B
Array element 2n+1 (real component)
Array element 2n (imaginary component)
multiply
multiply
multiply
multiply
sub
add
Output c
Array element 2n+1 (real component)
Array element 2n (imaginary component)

The following sections revisit these basic kernels and illustrate how single instruction multiple data optimizations apply to each of these.

8.2.6 Vectorizing With Packed Data Processing

The most basic packed data optimization is to use wide memory accesses, in other words, word and double-word loads and stores, to access narrow data such as byte or half-word data. This is a simple form of vectorization, as described above, applied only to the array accesses.

Widening memory accesses generally serves as a starting point for other vector and packed data operations. This is due to the fact that the wide memory accesses tend to impose a packed data flow on the rest of the code around them. This type of optimization is said to work from the outside in, as loads and stores typically occur near the very beginning and end of the loop body. The following examples use this outside-in approach to perform packed data optimization techniques on the example kernels.

Note:

The following examples assume that the compiler has not performed any packed data optimizations. The most recent release of the ’C6000 Code Generation Tools will apply many of the optimizations described in this chapter automatically when presented with sufficient information about the code.

8.2.6.1 Vectorizing the Vector Sum

Consider the vector sum kernel presented in Example 8–1. In its default form, it reads one half–word from the a[ ] array, one half-word from the b[ ] array, adds them, and writes a single half–word result to the c[ ] array. This results in a 2-cycle loop that moves 48 bits per iteration. When you consider that the ’C64x can read or write 128 bits every cycle, it becomes clear that this is very inefficient.

One simple optimization is to replace the half-word accesses with double-word accesses to read and write array elements in groups of four. When doing this, array elements are read into the register file with four elements packed into a register pair. The array elements are packed with, two elements in each register, across two registers. Each register contains data in the packed 16-bit data type illustrated in Figure 8–2.

For the moment, assume that the arrays are double-word aligned, as shown in Example 8–5. For more information about methods for working with arrays that are not specially aligned, see section 8.2.8. The first thing to note is that the ’C6000 Code Generation Tools lack a 64-bit integral type. This is not a problem, however, as the tools allow the use of *double*, and the intrinsics _lo(), _hi(), _itod() to access integer data with double-word loads. To account for the fact that the loop is operating on multiple elements at a time, the loop counter must be modified to count by fours instead of by single elements.

The type-cast tells the compiler to treat the array as an array of type *double*. This causes LDDW and STDW instructions to be issued for the array accesses. The _lo() and _hi() intrinsics break apart a 64-bit *double* into its lower and upper 32-bit halves. Each of these halves contain two 16-bit values packed in a 32-bit word. To store the results, the _itod() intrinsics assemble 32-bit words back into 64-bit doubles to be stored. Figure 8–11 and Figure 8–12 show these processes graphically.

The adds themselves have not been addressed, so for now, the add is replaced with a comment.

Example 8–5. Vectorization: Using LDDW and STDW in Vector Sum

```
void vec_sum(const short *restrict a, const short *restrict b,
             short *restrict c, int len)
{
   int i;
   unsigned a_hi, a_lo;
   unsigned b_hi, b_lo;
   unsigned c_hi, c_lo;

   for (i = 0; i < len; i += 4)
   {
      a_hi = _hi(*(const double *) &a[i]);
      a_lo = _lo(*(const double *) &a[i]);

      b_hi = _hi(*(const double *) &b[i]);
      b_lo = _lo(*(const double *) &b[i]);

      /*  ...somehow, the ADD occurs here,
      with results in c_hi, c_lo... */

      *(double *) &c[i] = _itod(c_hi, c_lo);
   }
}
```

Figure 8–11.Array Access in Vector Sum by LDDW

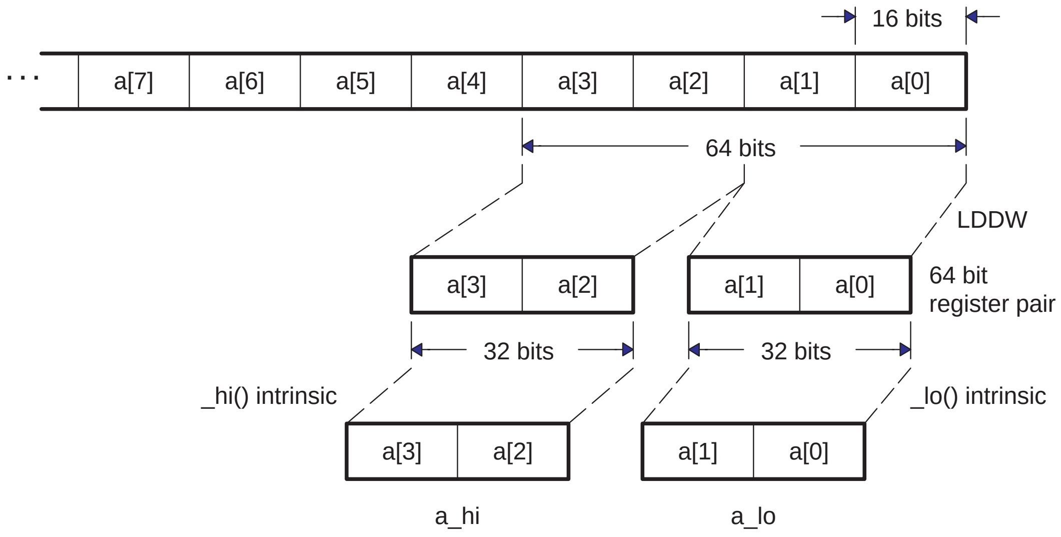
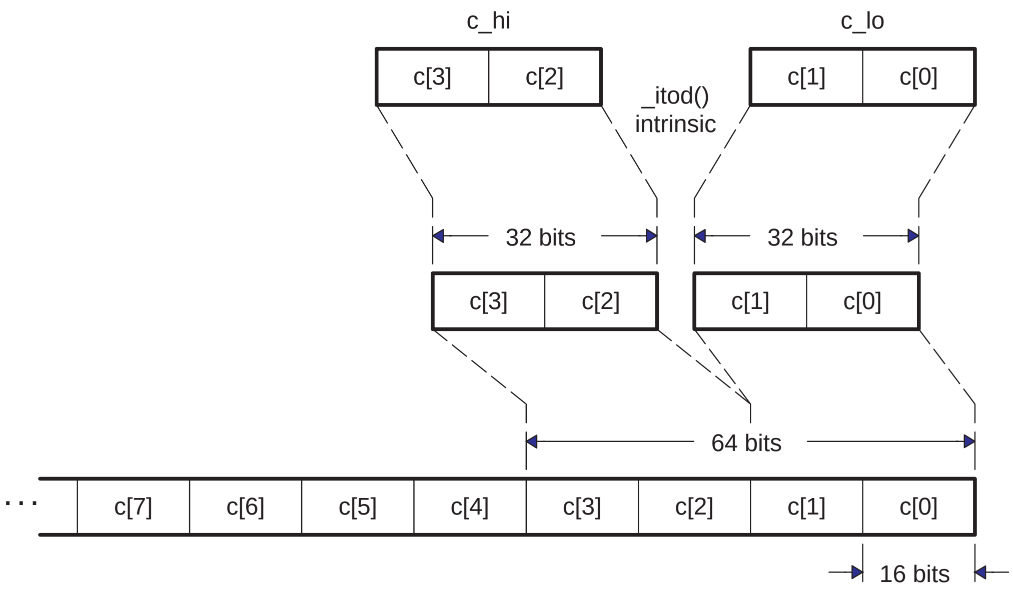


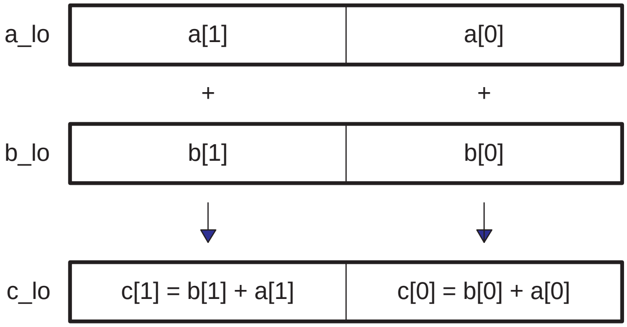
Figure 8–12. Array Access in Vector Sum by STDW



This code now efficiently reads and writes large amounts of data. The next step is to find a method to quickly add them. The _add2() intrinsic provides just that: It adds corresponding packed elements in two different words, producing two packed sums. It provides exactly what is needed, a vector addition. Figure 8–13 illustrates.

Figure 8–13. Vector Addition

c_lo = _add2(b_lo, a_lo);



So, putting in _add2() to perform the additions provides the complete code shown in Example 8–6.

Example 8–6. Vector Addition (Complete)

```
void vec_sum(const short *restrict a, const short *restrict b,
   short *restrict c, int len)
{
   int i;
   unsigned a3_a2, a1_a0;
   unsigned b3_b2, b1_b0;
   unsigned c3_c2, c1_c0;

   for (i = 0; i < len; i += 4)
   {
      a3_a2 = _hi(*(const double *) &a[i]);
      a1_a0 = _lo(*(const double *) &a[i]);

      b3_b2 = _hi(*(const double *) &b[i]);
      b1_b0 = _lo(*(const double *) &b[i]);

      c3_c2 = _add2(b3_b2, a3_a2);
      c1_c0 = _add2(b1_b0, a1_a0);

      *(double *) &c[i] = _itod(c3_c2, c1_c0);
   }
}
```

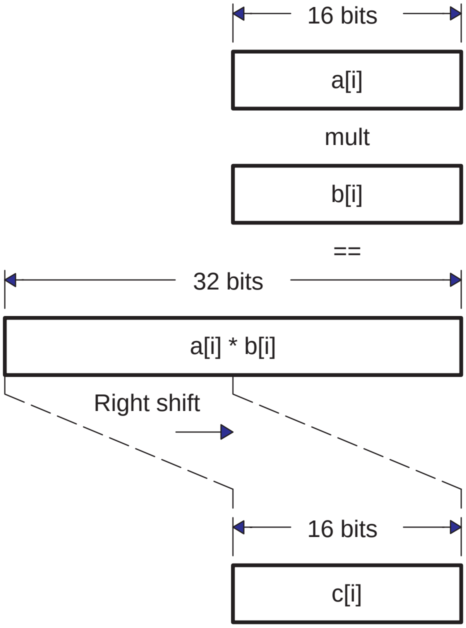
At this point, the vector sum is fully vectorized, and can be optimized further using other traditional techniques such as loop unrolling and software pipelining. These and other optimizations are described in detail throughout Chapter 7.

8.2.6.2 Vectorizing the Vector Multiply

The vector multiply shown in Figure 8–8 is similar to the vector sum, in that the algorithm is a pure vector algorithm. One major difference, is the fact that the intermediate values change precision. In the context of vectorization, this changes the format the data is stored in, but it does not inhibit the ability to vectorize the code.

The basic operation of vector multiply is to take two 16-bit elements, multiply them together to produce a 32-bit product, right-shift the 32-bit product to produce a 16-bit result, and then to store this result. The entire process for a single iteration is shown graphically in Figure 8–14.

Figure 8–14. Graphical Representation of a Single Iteration of Vector Multiply.



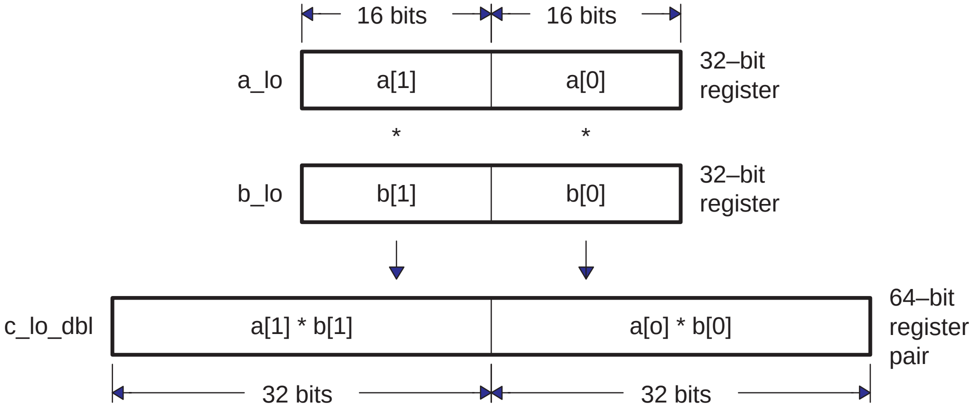
Notice that the values are still loaded and stored as 16-bit quantities. Therefore, you should use the same basic flow as the vector sum. Example 8–7 shows this starting point. Figure 8–11 and Figure 8–12 also apply to this example to illustrate how data is being accessed.

Example 8–7. Using LDDW and STDW in Vector Multiply

```
void vec_mpy(const short *restrict a, const short *restrict b,
             short *restrict c, int len, int shift)
{
   int i;
   unsigned a_hi, a_lo;
   unsigned b_hi, b_lo;
   unsigned c_hi, c_lo;

   for (i = 0; i < len; i += 4)
   {
      a_hi = _hi(*(const double *) &a[i]);
      a_lo = _lo(*(const double *) &a[i]);

      b_hi = _hi(*(const double *) &b[i]);
      b_lo = _lo(*(const double *) &b[i]);

      /*  ...somehow, the Multiply and Shift occur here,
      with results in c_hi, c_lo... */

      *(double *) &c[i] = _itod(c_hi, c_lo);
   }
}
```

The next step is to perform the multiplication. The ’C64x intrinsic, _mpy2(), performs two 16 × 16 multiplies, providing two 32-bit results packed in a 64-bit double. This provides the multiplication. The _lo() and _hi() intrinsics allow separation of the two separate 32-bit products. Figure 8–15 illustrates how _mpy2() works.

Figure 8–15. Packed 16×16 Multiplies Using _mpy2



Once the 32-bit products are obtained, use standard 32-bit shifts to shift these to their final precision. However, this will leave the results in two separate 32-bit registers.

The ’C64x provides the _pack family intrinsics to convert the 32-bit results into 16-bit results. The _packXX2() intrinsics, described in section 8.2.4, extract two 16-bit values from two 32-bit registers, returning the results in a single 32-bit register. This allows efficient conversion of the 32-bit intermediate results to a packed 16-bit format.

In this case, after the right-shift, the affected bits will be in the lower half of the 32-bit registers. Use the _pack2() intrinsic to convert the 32-bit intermediate values back to packed 16-bit results so they can be stored. The resulting C code is shown in Example 8–8.

Example 8–8. Using _mpy2() and _pack2() to Perform the Vector Multiply

```
void vec_mpy1(const short *restrict a, const short *restrict b,
      short *restrict c, int len, int shift)
{
   int i;
   unsigned a_hi, a_lo;                 /* Packed 16–bit values        */
   unsigned b_hi, b_lo;                 /* Packed 16–bit values        */
   double   c_hi_dbl, c_lo_dbl;         /* 32–bit prod in 64–bit pairs */
   int      c_hi3, c_hi2, c_lo1, c_lo0; /* Separate 32–bit products    */
   unsigned c_hi, c_lo;                 /* Packed 16–bit values        */

   for (i = 0; i < len; i += 4)
      {
      a_hi = _hi(*(const double *) &a[i]);
      a_lo = _lo(*(const double *) &a[i]);

      b_hi = _hi(*(const double *) &b[i]);
      b_lo = _lo(*(const double *) &b[i]);

      /* Multiply elements together, producing four products */
      c_hi_dbl = _mpy2(a_hi, b_hi);
      c_lo_dbl = _mpy2(a_lo, b_lo);

      /* Shift each of the four products right by our shift amount */
      c_hi3 = _hi(c_hi_dbl) >> shift;
      c_hi2 = _lo(c_hi_dbl) >> shift;
      c_lo1 = _hi(c_lo_dbl) >> shift;
      c_lo0 = _lo(c_lo_dbl) >> shift;

      /* Pack the results back together into packed 16–bit format */
      c_hi  = _pack2(c_hi3, c_hi2);
      c_lo  = _pack2(c_lo1, c_lo0);

      /* Store the results. */
      *(double *) &c[i] = _itod(c_hi, c_lo);
      }
}
```

This code works, but it is heavily bottlenecked on shifts. One way to eliminate this bottleneck is to use the packed 16-bit shift intrinsic, _shr2(). This can be done without losing precision, under the following conditions:

❑ If the shift amount is known to be greater than or equal to 16, use _packh2() instead of _pack2() before the shift. If the shift amount is exactly 16, eliminate the shift. The _packh2 effectively performs part of the shift, shifting right by 16, so that the job can be finished with a _shr2() intrinsic. Figure 8–17 illustrates how this works.

❑ If the shift amount is less than 16, only use the _shr2() intrinsic if the 32-bit products can be safely truncated to 16 bits first without losing significant digits. In this case, use the _pack2() intrinsic, but the bits above bit 15 are lost in the product. This is safe only if those bits are redundant (sign bits). Figure 8–17 illustrates this case.

Figure 8–16. Fine Tuning Vector Multiply (shift > 16)

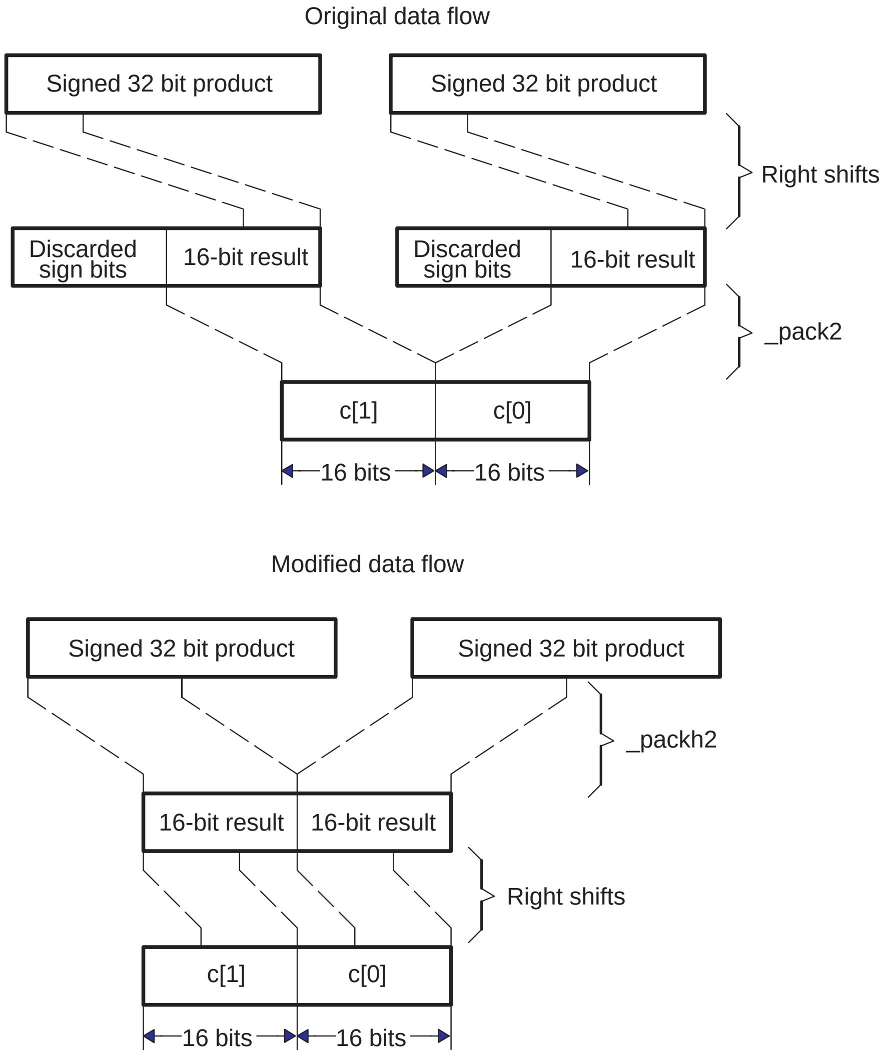
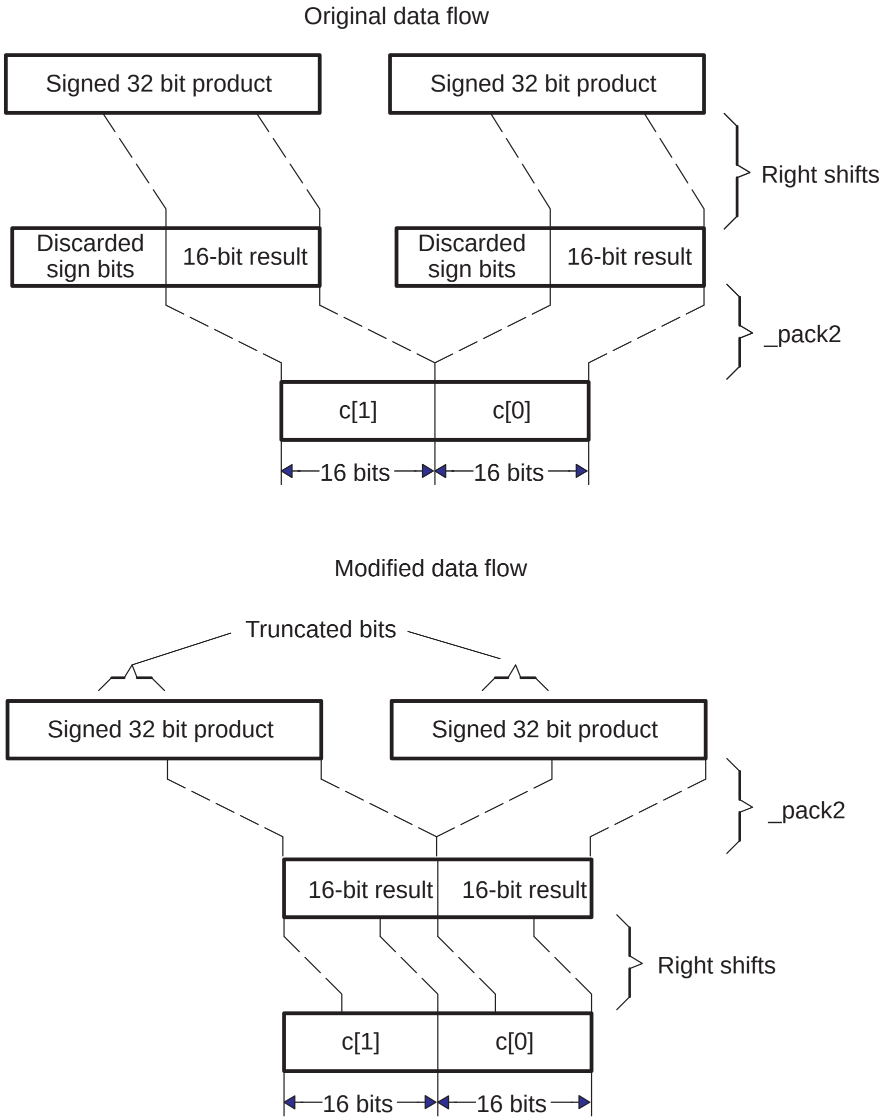


Figure 8–17. Fine Tuning Vector Multiply (shift < 16)



Whether or not the 16-bit shift version is used, consider the vector multiply to be fully optimized from a packed data processing standpoint. It can be further optimized using the more general techniques such as loop-unrolling and software pipelining that are discussed in Chapter 6.

8.2.7 Combining Multiple Operations in a Single Instruction

The Dot Product and Vector Complex Multiply examples that were presented earlier were both examples of kernels that benefit from *macro operations*, that is, instructions which perform more than a simple operation.

The ’C64x provides a number of instructions which combine common operations together. These instructions reduce the overall instruction count in the code, thereby reducing codesize and increasing code density. They also tend to simplify programming. Some of the more commonly used macro operations are listed in Table 8–5.

Table 8–5. Intrinsics Which Combine Multiple Operations in one Instruction

Intrinsic	Instruction	Operations combined
_dotp2	DOTP2	Performs two 16x16 multiplies and adds the products together.
_dotpn2	DOTPN2	Performs two 16x16 multiplies and subtracts the second product from the first.
_dotprsu2	DOTPRSU2	Performs two 16x16 multiplies, adds products together, and shifts/rounds the sum.
_dotpnrsu2	DOTPNRSU2	Performs two 16x16 multiplies, subtracts the 2nd product from the 1st, and shifts/rounds the difference.
_dotpu4 _dotpsu4	DOTPU4 DOTPSU4	Performs four 8x8 multiplies and adds products together.
_max2 _min2	MAX2 MIN2	Compares two pairs of numbers, and selects the larger/smaller in each pair.
_maxu4 _minu4	MAXU4 MINU4	Compares four pairs of numbers, and selects the larger/smaller in each pair.
_avg2	AVG2	Performs two 16-bit additions, followed by a right shift by 1 with round.
_avgu4	AVGU4	Performs four 8-bit additions, followed a right shift by 1 with round.
_subabs4	SUBABS4	Finds the absolute value of the between four pairs of 8-bit numbers.

As you can see, these macro operations can replace a number of separate instructions rather easily. For instance, each _dotp2 eliminates an add, and each _dotpu4 eliminates *three* adds. The following sections describe how to write the Dot Product and Vector Complex Multiply examples to take advantage of these.

8.2.7.1 Combining Operations in the Dot Product Kernel

The Dot Product kernel, presented in Example 8–3, is one which benefits both from vectorization as well as macro operations. First, apply the vectorization optimization as presented earlier, and then look at combining operations to further improve the code.

Vectorization can be performed on the array reads and multiplies that are this kernel, as described in section 8.2.3. The result of those steps is the intermediate code shown in Example 8–9.

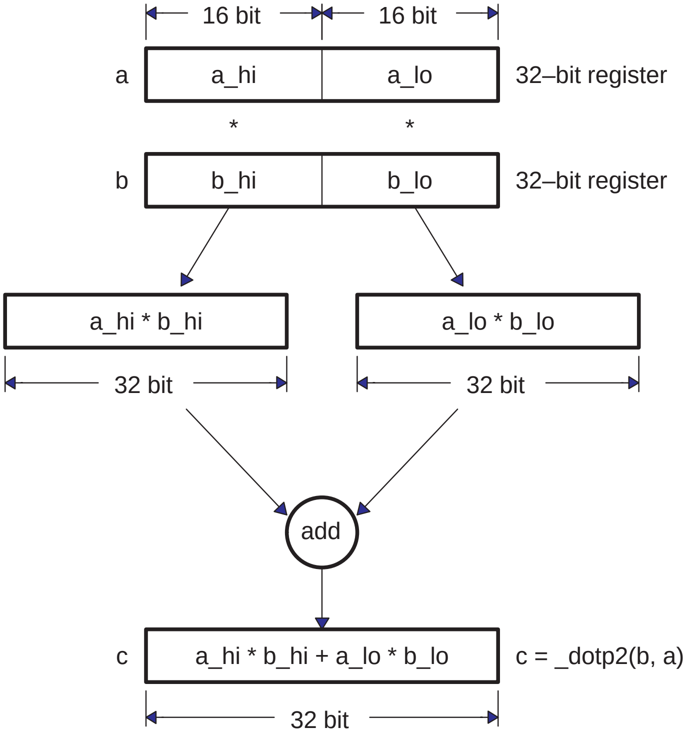
Example 8–9. Vectorized Form of the Dot Product Kernel

```
int dot_prod(const short *restrict a, const short *restrict b,
              short *restrict c, int len)
{
   int i;
   int      sum = 0;                  /* 32-bit accumulation         */
   unsigned a3_a2, a1_a0;             /* Packed 16-bit values        */
   unsigned b3_b2, b1_b0;             /* Packed 16-bit values        */
   double   c3_c2_dbl, c1_c0_dbl;     /* 32-bit prod in 64-bit pairs */
   int      c3, c2, c1, c0;           /* Separate 32-bit products    */
   unsigned c3_c2, c1_c0;             /* Packed 16-bit values        */

   for (i = 0; i < len; i += 4)
   {
      a3_a2 = _hi(*(const double *) &a[i]);
      a1_a0 = _lo(*(const double *) &a[i]);

      b3_b2 = _hi(*(const double *) &b[i]);
      b1_b0 = _lo(*(const double *) &b[i]);

      /* Multiply elements together, producing four products */
      c3_c2_dbl = _mpy2(a3_a2, b3_b2);
      c1_c0_dbl = _mpy2(a1_a0, b1_b0);

      /* Add each of the four products to the accumulation. */
      sum += _hi(c3_c2_dbl);
      sum += _lo(c3_c2_dbl);
      sum += _hi(c1_c0_dbl);
      sum += _lo(c1_c0_dbl);
   }

      return sum;
}
```

While this code is fully vectorized, it still can be improved. The kernel itself is performing two LDDWs, two MPY2, four ADDs, and one Branch. Because of the large number of ADDs, the loop cannot fit in a single cycle, and so the ’C64x datapath is not used efficiently.

The way to improve this is to combine some of the multiplies with some of the adds. The ’C64x family of _dotp intrinsics provides the answer here. Figure 8–18 illustrates how the _dotp2 intrinsic operates. Other _dotp intrinsics operate similarly.

Figure 8–18. Graphical Representation of the _dotp2 Intrinsic c = _dotp2(b, a)



This operation exactly maps to the operation the dot product kernel performs. The modified version of the kernel absorbs two of the four ADDs into _dotp intrinsics. The result is shown as Example 8–11. Notice that the variable *c* has been eliminated by summing the results of the _dotp intrinsic directly.

Example 8–10. Vectorized Form of the Dot Product Kernel

```
int dot_prod(const short *restrict a, const short *restrict b,
              short *restrict c, int len)
{
   int i;
   int      sum = 0;                  /* 32–bit accumulation          */
   unsigned a3_a2, a1_a0;             /* Packed 16–bit values         */
   unsigned b3_b2, b1_b0;             /* Packed 16–bit values         */

   for (i = 0; i < len; i += 4)
   {
      a3_a2 = _hi(*(const double *) &a[i]);
      a1_a0 = _lo(*(const double *) &a[i]);

      b3_b2 = _hi(*(const double *) &b[i]);
      b1_b0 = _lo(*(const double *) &b[i]);

      /* Perform dot–products on pairs of elements, totalling the
             results in the accumulator. */
      sum += _dotp2(a3_a2, b3_b2);
      sum += _dotp2(a1_a0, b1_b0);
   }

         return sum;
}
```

At this point, the code takes full advantange of the new features that the 'C64x provides. In the particular case of this kernel, no further optimization should be necessary. The tools produce an optimal single cycle loop, using the compiler version that was available at the time this book was written.

Example 8–11. Final Assembly Code for Dot–Product Kernel's Inner Loop

```
L2:
   [ B0]   SUB     .L2     B0,1,B0           ;
|| [!B0]   ADD     .S2     B8,B7,B7          ; |10|
|| [!B0]   ADD     .L1     A7,A6,A6          ; |10|
||         DOTP2   .M2X    B5,A5,B8          ; @@@@|10|
||         DOTP2   .M1X    B4,A4,A7          ; @@@@|10|
|| [ A0]   BDEC    .S1     L2,A0             ; @@@@@
||         LDDW    .D1T1   *A3++,A5:A4       ; @@@@@@@@@|10|
||         LDDW    .D2T2   *B6++,B5:B4       ; @@@@@@@@@|10|
```

8.2.7.2 Combining Operations in the Vector Complex Multiply Kernel

The Vector Complex Multiply kernel that was originally shown in Example 8–4 can be optimized with a technique similar to the one that used with the Dot Product kernel in Section 8.2.4.1. First, the loads and stores are vectorized in order to bring data in more efficiently. Next, operations are combined together into intrinsics to make full use of the machine.

Example 8–12 illustrates the vectorization step. For details, consult the earlier examples, such as the Vector Sum. The complex multiplication step itself has not yet been optimized at all.

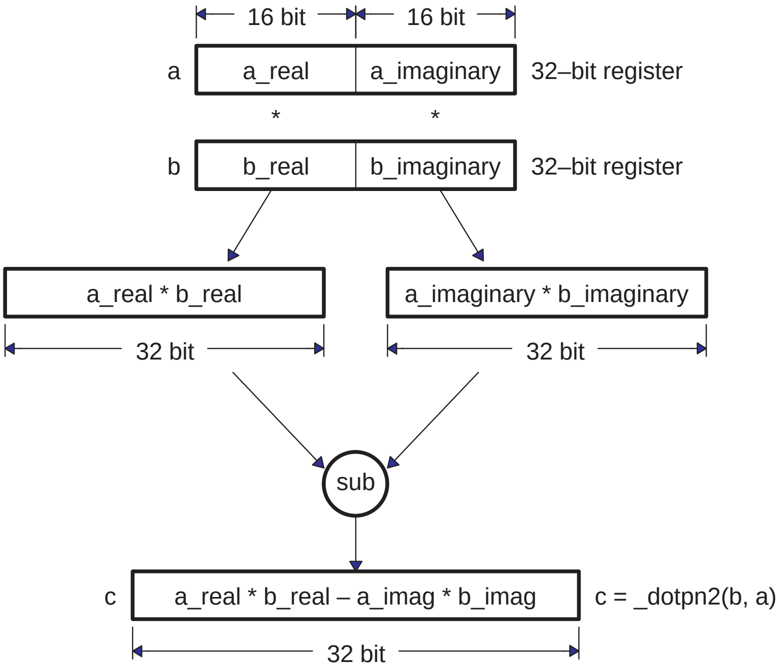
Example 8–12. Vectorized form of the Vector Complex Multiply Kernel

```
void vec_cx_mpy(const short *restrict a, const short *restrict b,
                    short *restrict c, int len, int shift)
    {
        int i;
        unsigned a3_a2, a1_a0;             /* Packed 16–bit values        */
        unsigned b3_b2, b1_b0;             /* Packed 16–bit values        */
        short    a3, a2, a1, a0;           /* Separate 16–bit elements    */
        short    b3, b2, b1, b0;           /* Separate 16–bit elements    */
        short    c3, c2, c1, c0;           /* Separate 16–bit results     */
        unsigned c3_c2, c1_c0;             /* Packed 16–bit values        */

        for (i = 0; i < len; i += 4)
        {
            /* Load two complex numbers from the a[] array.               */
            /* The complex values loaded are represented as ’a3 + a2 * j’ */
            /* and ’a1 + a0 * j’.  That is, the real components are a3    */
            /* and a1, and the imaginary components are a2 and a0.        */
            a3_a2 = _hi(*(const double *) &a[i]);
            a1_a0 = _lo(*(const double *) &a[i]);

            /* Load two complex numbers from the b[] array.               */
            b3_b2 = _hi(*(const double *) &b[i]);
            b1_b0 = _lo(*(const double *) &b[i]);

            /* Separate the 16–bit coefficients so that the complex      */
            /* multiply may be performed.  This portion needs further    */
            /* optimization.                                              */
            a3 = ((signed) a3_a2) >> 16;
            a2 = _ext(a3_a2, 16, 16);
            a1 = ((signed) a1_a0) >> 16;
            a0 = _ext(a1_a0, 16, 16);

            b3 = ((signed) a3_a2) >> 16;
            b2 = _ext(a3_a2, 16, 16);
            b1 = ((signed) a1_a0) >> 16;
            b0 = _ext(a1_a0, 16, 16);

            /* Perform the complex multiplies using 16x16 multiplies.    */
            c3 = (b3 * a2 + b2 * a3) >> 16;
            c2 = (b3 * a3 – b2 * a2) >> 16;

            c1 = (b1 * a0 + b0 * a1) >> 16;
            c0 = (b1 * a1 – b0 * a0) >> 16;

            /* Pack the 16–bit results into 32–bit words.                 */
            c3_c2 = _pack2(c3, c2);
            c1_c0 = _pack2(c1, c0);

            /* Store the results. */
            *(double *) &c[i] = _itod(c3_c2, c1_c0);
        }
    }
```

Example 8–12 still performs the complex multiply as a series of discrete steps once the individual elements are loaded. The next optimization step is to combine some of the multiplies and adds/subtracts into _dotp and _dotpn intrinsics in a similar manner to the Dot Product example presented earlier.

The real component of each result is calculated by taking the difference between the product of the real components of both input and the imaginary components of both inputs. Because the real and imaginary components for each input array are laid out the same, the _dotpn intrinsic can be used to calculate the real component of the output. Figure 8–19 shows how this flow would work.

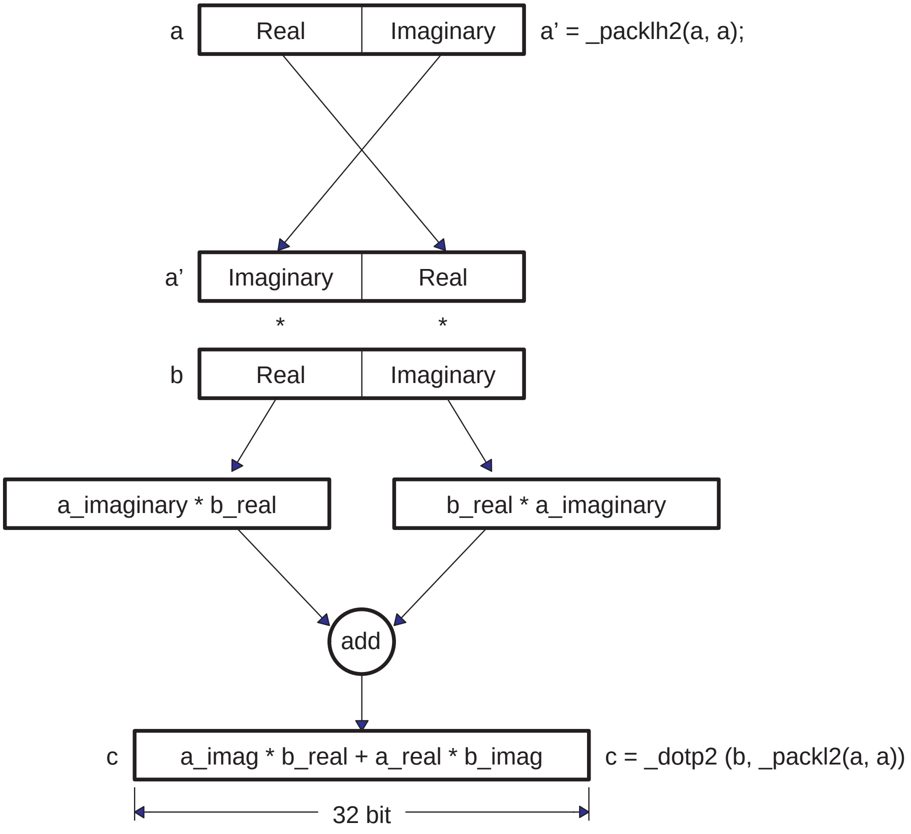
Figure 8–19. The _dotpn2 Intrinsic Performing Real Portion of Complex Multiply.



The calculation for the result’s imaginary component provides a different problem. As with the real component, the result is calculated from two products that are added together. A problem arises, though, because it is necessary to multiply the real component of one input with the imaginary component of the other input, and vice versa. None of the ’C64x intrinsics provide that operation directly given the way the data is currently packed.

The solution is to reorder the halfwords from one of the inputs, so that the imaginary component is in the upper halfword and the real component is in the lower halfword. This is accomplished by using the _packlh2 intrinsic to reorder the halves of the word. Once the half–words are reordered on one of the inputs, the _dotp intrinsic provides the appropriate combination of multiplies with an add to provide the imaginary component of the output.

Figure 8–20. _packlh2 and _dotp2 Working Together.



Once both the real and imaginary components of the result are calculated, it is necessary to convert the 32-bit results to 16-bit results and store them. In the original code, the 32-bit results were shifted right by 16 to convert them to 16-bit results. These results were then packed together with _pack2 for storing. Our final optimization replaces this shift and pack with a single _packh2. Example 8–13 shows the result of these optimizations.

Example 8–13. Vectorized form of the Vector Complex Multiply

```
void vec_cx_mpy(const short *restrict a, const short *restrict b,
                short *restrict c, int len, int shift)
{
    int i;
    unsigned a3_a2, a1_a0;             /* Packed 16–bit values        */
    unsigned b3_b2, b1_b0;             /* Packed 16–bit values        */
    int      c3,c2, c1,c0;             /* Separate 32–bit results     */
    unsigned c3_c2, c1_c0;             /* Packed 16–bit values        */

    for (i = 0; i < len; i += 4)
    {
        /* Load two complex numbers from the a[] array.               */
        /* The complex values loaded are represented as ’a3 + a2 * j’ */
        /* and ’a1 + a0 * j’.  That is, the real components are a3    */
        /* and a1, and the imaginary components are a2 and a0.        */
        a3_a2 = _hi(*(const double *) &a[i]);
        a1_a0 = _lo(*(const double *) &a[i]);

        /* Load two complex numbers from the b[] array.               */
        b3_b2 = _hi(*(const double *) &b[i]);
        b1_b0 = _lo(*(const double *) &b[i]);

        /* Perform the complex multiplies using _dotp2/_dotpn2.      */
        c3 = _dotpn2(b3_b2, a3_a2);                  /* Real         */
        c2 = _dotp2 (b3_b2, _packlh2(a3_a2, a3_a2)); /* Imaginary    */

        c1 = _dotpn2(b1_b0, a1_a0);                  /* Real         */
        c0 = _dotp2 (b1_b0, _packlh2(a1_a0, a1_a0)); /* Imaginary    */

        /* Pack the 16–bit results from the upper halves of the      */
        /* 32–bit results into 32–bit words.                         */
        c3_c2 = _packh2(c3, c2);
        c1_c0 = _packh2(c1, c0);

        /* Store the results. */
        *(double *) &c[i] = _itod(c3_c2, c1_c0);
    }
}
```

As with the earlier examples, this kernel now takes full advantage of the packed data processing features that the ’C64x provides. More general optimizations can be performed as described in Chapter 7 to further optimize this code.

8.2.8 Non-Aligned Memory Accesses

In addition to traditional aligned memory access methods, the ’C64x also provides intrinsics for non-aligned memory accesses. Aligned memory accesses are restricted to an alignment boundary that is determined by the amount of data being accessed. For instance, a 64-bit load must read the data from a location at a 64-bit boundary. Non-aligned access intrinsics relax this restriction, and can access data at any byte boundary.

There are a number of tradeoffs between aligned and non-aligned access methods. Table 8–6 lists the differences between both methods.

Table 8–6. Comparison Between Aligned and Non–Aligned Memory Accesses

Aligned	**Non–Aligned**
Data must be aligned on a boundary equal to its width.	Data may be aligned on any byte boundary.
Can read or write bytes, half-words, words, and double-words.	Can only read or write words and double-words.
Up to two accesses may be issued per cycle, for a peak bandwidth of 128 bits/cycle.	Only one non-aligned access may be issued per cycle, for a peak bandwidth of 64 bits/cycle.
Bank conflicts may occur.	No bank conflict possible, because no other memory access may occur in parallel.

Because the ’C64x can only issue one non-aligned memory access per cycle, programs should focus on using aligned memory accesses whenever possible. However, certain classes of algorithms are difficult or impossible to fit into this mold when applying packed-data optimizations. For example, convolution-style algorithms such as filters fall in this category, particularly when the outer loop cannot be unrolled to process multiple outputs at one time.

8.2.8.1 Using Non-Aligned Memory Access Intrinsics

Non-aligned memory accesses are generated using the _mem4() and _memd8() intrinsics. These intrinsics generate a non-aligned reference which may be read or written to, much like an array reference. Example 8–14 below illustrates reading and writing via these intrinsics.

Example 8–14. Non–aligned Memory Access With _mem4 and _memd8

```
char   a[1000];  /* Sample array */
    double d;

    /* Store four bytes at a[9] through a[12] */
    _mem4((void*) &a[9]) = 0x12345678;

    /* Load eight bytes from a[115] through a[122] */
    d = _memd8((void*) &a[115]);
```

It is easy to modify code to use non-aligned accesses. Example 8–15 below shows the Vector Sum from Example 8–6 rewritten to use non-aligned memory accesses. As with ordinary array references, the compiler will optimize away the redundant references.

Example 8–15. Vector Sum Modified to use Non–Aligned Memory Accesses

```
void vec_sum(const short *restrict a, const short *restrict b,
                 short *restrict c, int len)
    {
        int i;
        unsigned a3_a2, a1_a0;
        unsigned b3_b2, b1_b0;
        unsigned c3_c2, c1_c0;

        for (i = 0; i < len; i += 4)
        {
            a3_a2 = _hi(_memd8((void*) &a[i]));
            a1_a0 = _lo(_memd8((void*) &a[i]));

            b3_b2 = _hi(_memd8((void*) &b[i]));
            b1_b0 = _lo(_memd8((void*) &b[i]));

            c3_c2 = _add2(b3_b2, a3_a2);
            c1_c0 = _add2(b1_b0, a1_a0);

            _memd8((void*) &c[i]) = _itod(c3_c2, c1_c0);
        }
    }
```

8.2.8.2 When to Use Non-Aligned Memory Accesses

As noted earlier, the ’C64x can provide 128 bits/cycle bandwidth with aligned memory accesses, and 64 bits/cycle bandwidth with non-aligned memory accesses. Therefore, it is important to use non–aligned memory accesses in places where they provide a true benefit over aligned memory accesses. Generally, non-aligned memory accesses are a win in places where they allow a routine to be vectorized, where aligned memory accesses could not. These places can be broken down into several cases:

❑ Generic routines which cannot impose alignment,

❑ Single sample algorithms which update their input or output pointers by only one sample

❑ Nested loop algorithms where outer loop cannot be unrolled, and

❑ Routines which have an irregular memory access pattern, or whose access pattern is data-dependent and not known until run time.

An example of a generic routine which cannot impose alignment on routines that call it would be a library function such as *memcpy* or *strcmp*. Single-sample algorithms include adaptive filters which preclude processing multiple outputs at once. Nested loop algorithms include 2-D convolution and motion estimation. Data-dependent acccess algorithms include motion compensation, which must read image blocks from aribitrary locations in the source image.

In each of these cases, it is extremely difficult to transform the problem into one which uses aligned memory accesses while still vectorizing the code. Often, the result with aligned memory accesses is worse than if the code were not optimized for packed data processing at all. So, for these cases, non-aligned memory accesses are a win.

In contrast, non-aligned memory accesses should not be used in more general cases where they are not specifically needed. Rather, the program should be structured to best take advantage of aligned memory accesseswith a packed data processing flow. The following checklist should help.

❑ Use *signed short* or *unsigned char* data types for arrays where possible. These are the types for which the ’C64x provides the greatest support.

❑ Place arrays on double-word boundaries, using #pragma DATA_ALIGN. This allows the program to use LDDW and STDW to access the array, providing up to 128 bit/cycle bandwidth for accesing the array.

❑ Round loop counts, numbers of samples, and so on to multiples of 4 or 8 where possible. This allows the inner loop to be unrolled more readily to take advantage of packed data processing.

❑ In nested loop algorithms, unroll outer loops to process multiple output samples at once. This allows packed data processing techniques to be applied to elements that are indexed by the outer loop.

8.2.9 Performing Conditional Operations with Packed Data

The ’C64x provides a set of operations that are intended to provide conditional data flow in code that operates on packed data. These operations make it possible to avoid breaking the packed data flow with unpacking code and traditional ’if’ statements.

Common conditional operations, such as *maximum*, *minimum* and *absolute value* are addressed directly with their own specialized intrinsics. In addition to these specific operations, more generalized compare and select operations can be constructed using the packed compare intrinsics, _cmpXX2 and _cmpXX4, in conjunction with the expand intrinsics, _xpnd2 and _xpnd4.

The packed compare intrinsics compare packed data elements, producing a small bitfield which describes the results of the independent comparisons. For _cmpeq2, _cmpgt2, and _cmplt2, the intrinsic returns a two bit field containing the results of the two separate comparisons. For _cmpeq4, _cmpgtu4, and _cmpltu4, the intrinsic returns a four bit field containing the results of the four separate comparisons. In both sets of intrinsics, a 1 bit signifies that the tested condition is true, and a 0 signifies that it is false. Figure 8–21 and Figure 8–22 illustrate how these compare intrinsics work.

Figure 8–21. Graphical Illustration of _cmpXX2 Intrinsics

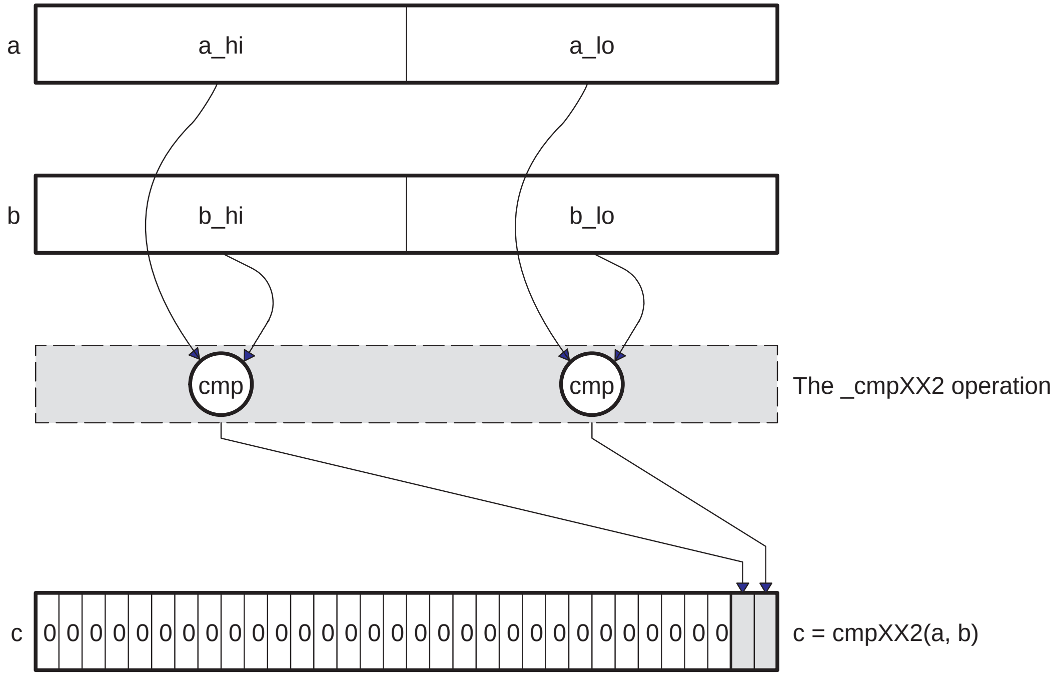
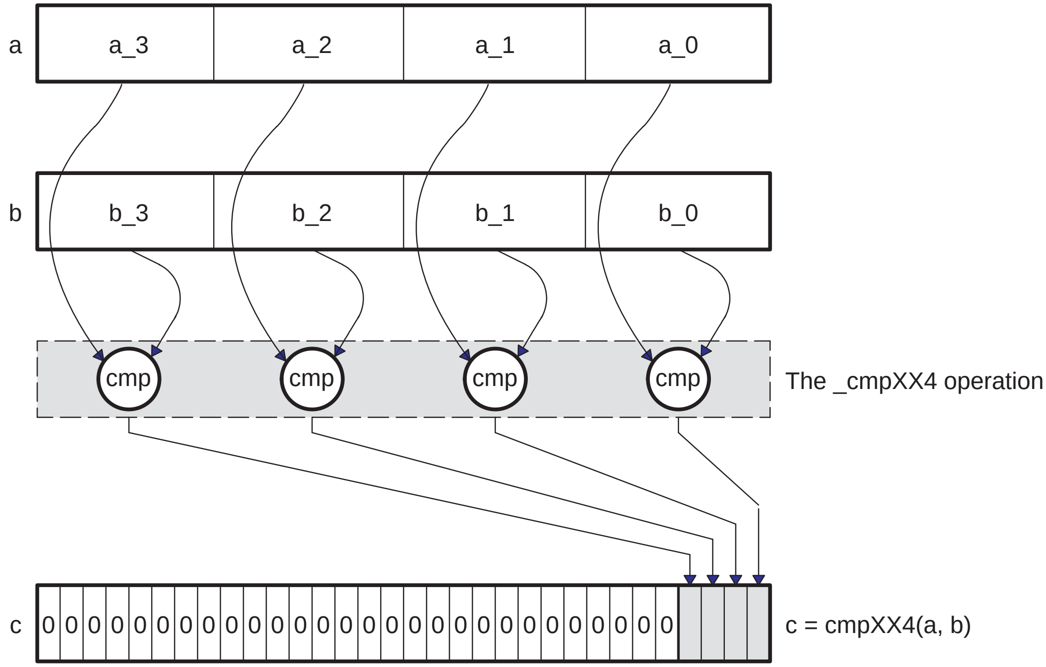


Figure 8–22. Graphical Illustration of _cmpXX4 Intrinsics



The expand intrinsics work from a bitfield such as the bitfield returned by the compare intrinsics. The _xpnd2 and _xpnd4 intrinsics expand the lower 2 or 4 bits of a word to fill the entire 32-bit word of the result. The _xpnd2 intrinsic expands the lower two bits of the input to two half–words, whereas _xpnd4 expands the lower four bits to four bytes. The expanded output is suitable for use as a mask, for instance, for selecting values based on the result of a comparison. Figure 8–23 and Figure 8–24 illustrate.

Figure 8–23. Graphical Illustration of _xpnd2 Intrinsic

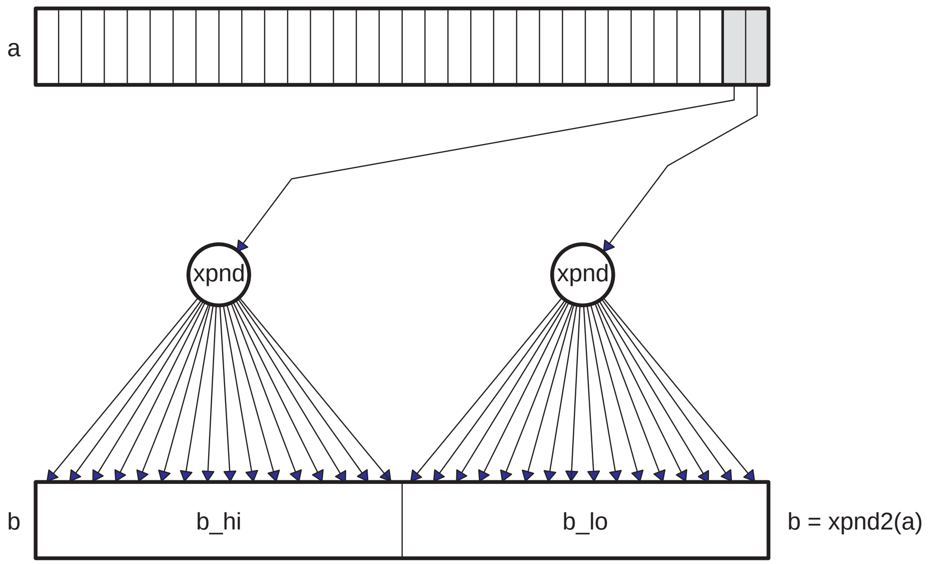
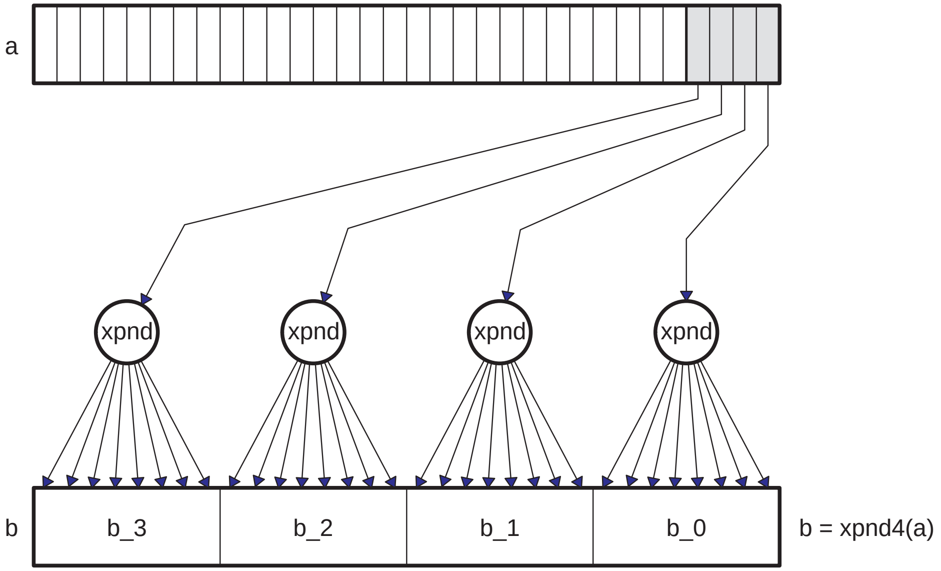


Figure 8–24. Graphical Illustration of _xpnd4 Intrinsic



Example 8–16 illustrates an example that can benefit from the packed compare and expand intrinsics in action. The *Clear Below Threshold* kernel scans an image of 8-bit unsigned pixels, and sets all pixels that are below a certain threshold to 0.

Example 8–16. Clear Below Threshold Kernel

```
void clear_below_thresh(unsigned char *restrict image, int count,
                           unsigned char threshold)
   {
       int i;

       for (i = 0; i < count; i++)
       {
           if (image[i] <= threshold)
               image[i] = 0;
       }
   }
```

Vectorization techniques are applied to the code (as described in Section 8.2), giving the result shown in Example 8–17. The _cmpgtu4() intrinsic compares against the threshold values, and the _xpnd4() intrinsic generates a mask for setting pixels to 0. Note that the new code has the restriction that the input image must be double-word aligned, and must contain a multiple of 8 pixels. These restrictions are reasonable as common image sizes have a multiple of 8 pixels.

Example 8–17. Clear Below Threshold Kernel, Using _cmpgtu4 and _xpnd4 Intrinsics

```
void clear_below_thresh(unsigned char *restrict image, int count,
                        unsigned char threshold)
{
    int i;
    unsigned t3_t2_t1_t0;               /* Threshold (replicated)   */
    unsigned p7_p6_p5_p4, p3_p2_p1_p0;  /* Pixels                   */
    unsigned c7_c6_c5_c4, c3_c2_c1_c0;  /* Comparison results       */
    unsigned x7_x6_x5_x4, x3_x2_x1_x0;  /* Expanded masks           */

    /* Replicate the threshold value four times in a single word */
    temp        = _pack2(threshold, threshold);
    t3_t2_t1_t0 = _packl4(temp, temp);

    for (i = 0; i < count; i += 8)
    {
        /* Load 8 pixels from input image (one double–word).        */
        p7_p6_p5_p4 = _hi(*(double*) &image[i]);
        p3_p2_p1_p0 = _lo(*(double*) &image[i]);

        /* Compare each of the pixels to the threshold.             */
        c7_c6_c5_c4 = _cmpgtu4(p7_p6_p5_p4, t3_t2_t1_t0);
        c3_c2_c1_c0 = _cmpgtu4(p3_p2_p1_p0, t3_t2_t1_t0);

        /* Expand the comparison results to generate a bitmask.     */
        x7_x6_x5_x4 = _xpnd4(c7_c6_c5_c4);
        x3_x2_x1_x0 = _xpnd4(c3_c2_c1_c0);

        /* Apply mask to the pixels.  Pixels that were less than or */
        /* equal to the threshold will be forced to 0 because the   */
        /* corresponding mask bits will be all 0s. The pixels that  */
        /* were greater will not be modified, because their mask    */
        /* bits will be all 1s.                                     */
        p7_p6_p5_p4 = p7_p6_p5_p4 & x7_x6_x5_x4;
        p3_p2_p1_p0 = p3_p2_p1_p0 & x3_x2_x1_x0;

        /* Store the thresholded pixels back to the image.          */
        *(double*) &image[i] = _itod(p7_p6_p5_p4, p3_p2_p1_p0);
    }
}
```

8.3 Linear Assembly Considerations

The ’C64x supports linear assembly programming via the C6000 Assembly Optimizer. The operation of the Assembly Optimizer is described in detail in the Optimizing C/C++ Compiler User’s Guide. This section covers ’C64x specific aspects of linear assembly programming.

8.3.1 Using BDEC and BPOS in Linear Assembly

The ’C64x provides two new instructions, BDEC and BPOS, which are designed to reduce codesize in loops, as well as to reduce pressure on predication registers. The BDEC instruction combines a decrement, test, and branch into a single instruction. BPOS is similar, although it does not decrement the register. For both, these steps are performed in the following sequence.

- Test the loop register to see if it is negative. If it is negative, no further action occurs. The branch is not taken and the loop counter is not updated.
- If the loop counter was not initially negative, decrement the loop counter and write the new value back to the register file. (This step does not occur for BPOS .)
- If the loop counter was not initially negative, issue the branch. Code will begin executing at the branch’s destination after the branch’s delay slots. From linear assembly, the branch appears to occur immediately, since linear assembly programming hides delay slots from the programmer.

This sequence of events causes BDEC to behave somewhat differently than a separate decrement and predicated branch. First, the decision to branch occurs *before* the decrement. Second, the decision to branch is based on whether the number is *negative*, rather than whether the number is *zero*. Together, these effects require the programmer to adjust the loop counter in advance of a loop.

Consider Example 8–18. In this C code, the loop iterates for `count` iterations, adding 1 to iters each iteration. After the loop, iters contains the number of times the loop iterated.

Example 8–18. Loop Trip Count in C

```
    int count_loop_iterations(int count)
    {
        int iters, i;

        iters = 0;

        for (i = count; i > 0; i––)
            iters++;

        return iters;
    }
```

Without BDEC and BPOS, this loop would be written as shown in Example 8–19 below. This example uses branches to test whether the loop iterates at all, as well as to perform the loop iteration itself. This loop iterates exactly the number of times specified by the argument ’count’.

Example 8–19. Loop Trip Count in Linear Assembly without BDEC

```
        .global _count_loop_iterations
_count_loop_iterations .cproc count
        .reg    i, iters, flag

        ZERO    iters               ; Initialize our return value to 0.

        CMPLT   count,  1,  flag
[flag]  B       does_not_iterate    ; Do not iterate if count

        MV      count,  i           ; i = count
loop:   .trip   1                   ; This loop is guaranteed to iterate at
                                    ; least once.

        ADD     iters,  1,  iters   ; iters++
        SUB     i,      1,  i       ; i––
 [i]    B       loop                ; while (i > 0);

does_not_iterate:

        .return iters                   ; Return our number of iterations.
        .endproc
```

Using BDEC , the loop is written similarly. However, the loop counter needs to be adjusted, since BDEC terminates the loop after the loop counter becomes negative. Example 8–20 illustrates using BDEC to conditionally execute the loop, as well as to iterate the loop. In the typical case, the loop count needs to be decreased by 2 before the loop. The SUB and BDEC before the loop perform this update to the loop counter.

Example 8–20. Loop Trip Count Using BDEC

```
        .global _count_loop_iterations
_count_loop_iterations .cproc count
        .reg    i, iters

        ZERO    iters               ; Initialize our return value to 0.

        SUB     count,  1,  i       ; i = count – 1;
        BDEC    loop,   i           ; Do not iterate if count < 1.

does_not_iterate:
        .return iters               ; Loop does not iterate, just return 0.

loop:   .trip   1                   ; This loop is guaranteed to iterate at
                                    ; least once.

        ADD     iters,  1,  iters   ; iters++
        BDEC    loop,   i           ; while (i–– >= 0);


        .return iters               ; Return our number of iterations.
        .endproc
```

Another approach to using BDEC is to allow the loop to execute extra iterations, and then compensate for these iterations after the loop. This is particularly effective in cases where the cost of the conditional flow before the loop is greater than the cost of executing the body of the loop, as in the example above. Example 8–21 shows one way to apply this modification.

Example 8–21. Loop Tip Count Using BDEC With Extra Loop Iterations

```
        .global _count_loop_iterations
_count_loop_iterations .cproc count
        .reg    i, iters

        MVK     –1,     iters       ; Loop executes exactly 1 extra iteration,
                                    ; so start with the iteration count == –1.

        SUB     count,  1,  i       ; Force ”count==0” to iterate exactly once.

loop:   .trip   1                   ; This loop is guaranteed to iterate at
                                    ; least once.

        ADD     iters,  1,  iters   ; iters++
        BDEC    loop,   i           ; while (i–– >= 0);

        .return iters               ; Return our number of iterations.
        .endproc
```

8.3.1.1 Function Calls and ADDKPC in Linear Assembly

The ’C64x provides a new instruction, ADDKPC , which is designed to reduce codesize when making function calls. This new instruction is not directly accessible from Linear Assembly. However, Linear Assembly provides the function call directive, .call, and this directive makes use of ADDKPC. The .call directive is explained in detail in the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide*.

Example 8–22 illustrates a simple use of the .call directive. The Assembly Optimizer issues an ADDKPC as part of the function call sequence for this .call, as shown in the compiler output in Example 8–23.

Example 8–22. Using the .call Directive in Linear Assembly

```
       .data
hello   .string ”Hello World”, 0

        .text
        .global _puts
        .global _main

_main   .cproc
        .reg    pointer

loop:
        MVKL    hello,  pointer     ; Generate a 32–bit pointer to the
        MVKH    hello,  pointer     ; phrase ”Hello World”.

        .call   _puts(pointer)      ; Print the string ”Hello World”.

        B       loop                ; Keep printing it.

        .endproc
```

Example 8–23. Compiler Output Using ADDKPC

```
loop:
;    .call   _puts(pointer)    ; Print the string ”Hello World”.
      B       .S1   _puts      ; |15|
      MVKL    .S1   hello,A4   ; |12|  Generate a 32–bit pointer to the
      ADDKPC  .S2   RL0,B3,2   ; |15|
      MVKH    .S1   hello,A4   ; |13|  phrase ”Hello World”.
RL0:  ; CALL OCCURS            ; |15|
```

8.3.1.2 Using .mptr and .mdep With Linear Assembly on the ’C64x

The Assembly Optimizer supports the .mptr and .mdep directives on the ’C64x. These directives allow the programmer to specify the memory access pattern for loads and stores, as well as which loads and stores are dependent on each other. Section 6.2, Assembly Optimizer Options and Directives, describes these directives in detail. This section describes the minor differences in the behavior of the .mptr directive on ’C64x vs. other C6000 family members.

Most ’C64x implementations will have different memory bank structure than existing ’C62x implementations in order to support the wider memory accesses that the ’C64x provides. Refer to the *TMS320C6000 Peripherals Reference Guide* for specific information on the part that you are using.

Additionally, the ’C64x’s non-aligned memory accesses do not cause bank conflicts. This is due to the fact that no other memory access can execute in parallel with a non-aligned memory access. As a result, the.mptr directive has no effect on non-aligned load and store instructions.

Appendix A

FeedbackSolutions

This appendix is provided as a quick reference to techniques that can be used to optimize loops, and in most cases, refers you to specific sections within this book for more detail.

Topic **Page**

A.1 Loop Disqualification Messages

Bad Loop Structure

Description

This error is very rare and can stem from the following:

❑ An asm statement inserted in the C code innerloop.

❑ Parallel instructions being used as input to the Linear Assembly Optimizer.

❑ Complex control flow such as GOTO statements and breaks.

Solution

Remove any asm statements, complex control flow or parallel instructions as input to linear assembly.

Loop Contains a Call

Description

Sometimes the compiler may not be able to inline a function call that is in a loop. Because the compiler could not inline the function call, the loop could not be software pipelined.

Solution

If the caller and the callee are C or C++, use –pm and –op2. See the *TMS320C6000 Opimizing C/C++ Compiler User's Guide* for more information on the correct usage of –op2. Do not use –oi0, which disables automatic inlining.

Add the inline keyword to the callee's function definition.

Too Many Instructions

Loops that are too big typically will not schedule due to too many registers needed and cause a large compilation time in the compiler. The limit on the number of instructions is variable.

Solution

Use intrinsics in C code to select more efficient ’C6000 instructions.

Write code in linear assembly to pick exact ’C6000 instruction to be executed.

For more information...

See section 3.4.1, *Using Intrinsics*, on page 3-18.

See Chapter 7, *Optimizing Assembly Code via Linear Assembly*.

Software Pipelining Disabled

Software pipelining has been disabled by a command–line option. Pipelining will be turned off when using the –mu option, not using –o2/–o3, or using – ms2/–ms3.

Uninitialized Trip Counter

The trip counter may not have been set to an initial value.

Suppressed to Prevent Code Expansion

Software pipelining may be suppressed because of the –ms1 flag. When the –ms1 flag is used, software pipelining is disabled in less promising cases to reduce code size. To enable pipelining, use –ms0 or omit the –ms flag altogether.

Loop Carried Dependency Bound Too Large

If the loop has complex loop control, try –mh according to the recommendations in the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide.*

Cannot Identify Trip Counter

The loop control is too complex. Try to simplify the loop.

A.2 Pipeline Failure Messages

Address Increment Too Large

Description

One thing the compiler does when software pipelining is to allow reordering of all loads and stores occurring from the same array or pointer. This allows for maximum flexibility in scheduling. Once a schedule is found, the compiler then goes back and adds the appropriate offsets and increment/decrements to each load and store. Sometimes, the loads and/or stores end up being offset too far from each other after reordering (the limit for standard load pointers is +/– 32) . If this happens, the best bet is to restructure the loop so that the pointers are closer together or rewrite the pointers to use register offsets that are precomputed.

Solution

Modify code so that the memory offsets are closer.

Cannot Allocate Machine Registers

Description

After software pipelining and finding a valid schedule, the compiler must allocate all values in the loop to specific machine registers (A0–A15 and B0–B15 for the ’C62x and ’C67x, or A0–A31 and B0–B31 for the ’C64x). Sometimes the loop schedule found simply requires more registers than the ’C6000 has available and thus software pipelining that particular ii is not possible. The analyzing feedback example shows:

```
ii = 12 Cannot allocate machine registers
Regs Live Always : 1/5 (A/B–side)
Max Regs Live : 14/19
Max Cond Regs Live : 1/0
```

Regs Live Always refers to the number of registers needed for variables live every cycle in the loop. Data loaded into registers outside the loop and read inside the loop will fall into this category.

Max Regs Live refers to the maximum number of variable live on any one cycle in the loop. If there are 33 variables live on one of the cycles inside the loop, a minimum of 33 registers is necessary and this will not be possible with the 32 registers available on the C62/C67 cores. 64 registers are available on the ’C64x core. In addition, this is broken down between A and B side, so if there is uneven partitioning with 30 values and there are 17 on one side and 13 on the other, the same problem will exist.

Max Cond Regs Live tells us if there are too many conditional values needed on a given cycle. The ’C62x/’C67x cores have 2 A side and 3 B side condition registers available. The ’C64x core has 3 A side and 3 B side condition registers available.

Solution

Try splitting the loop into two separate loops. Repartition if too many instructions on one side.

For loops with complex control, try the –mh option.

Use symbolic register names instead of machine registers (A0–A15 and B0–B15 for ’C62x and ’C67x, or A0–A31 and B0–B31 for ’C64x).

For More Information...

See section 6.9, *Loop Unrolling (in Assembly)*, on page 6-94.

See section 3.4.3.4, *Loop Unrolling (in C)*, on page 3-45.

TMS320C6000 C/C++ Compiler User’s Guide

Cycle Count Too High. Not Profitable

Description

In rare cases, the iteration interval of a software pipelined loop is higher than a non-pipelined list scheduled loop. In this case, it is more efficient to execute the non-software pipelined version.

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

Unpartition/repartition the linear assembly source code.

Add const and restrict keywords where appropriate to reduce dependences.

For loops with complex control, try the –mh option.

Probably best modified by another technique (i.e. loop unrolling).

Modify the register and/or partition constraints in linear assembly.

For more information...

See section 6.9, *Loop Unrolling*, on page 6-94.

TMS320C6000 C/C++ Compiler User’s Guide

Did Not Find Schedule

Description

Sometimes, due to a complex loop or schedule, the compiler simply cannot find a valid software pipeline schedule at a particular iteration interval.

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

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.

For more information...

See section 6.9, *Loop Unrolling*, on page 6-94.

Iterations in Parallel > Max. Trip Count

Description

Not all loops can be profitably pipelined. Based on the available information on the largest possible trip count, the compiler estimates that it will always be more profitable to execute a non-pipelined version than to execute the pipelined version, given the schedule that it found at the current iteration interval.

Solution

Probably best optimized by another technique (i.e. unroll the loop completely).

For more information...

See section 6.9, *Loop Unrolling (in Assembly)*, on page 6-94.

See section 3.4.3.4, *Loop Unrolling (in C)*, on page 3-45.

See section 3.4.3, *Software Pipelining*, on page 3-41.

Speculative Threshold Exceeded

Description

It would be necessary to speculatively load beyond the threshold currently specified by the –mh option.

Solution

Increase the –mh threshold as recommended in the software pipeline feedback located in the assembly file.

Iterations in Parallel > Min. Trip Count

Description

Based on the available information on the minimum trip count, it is not always safe to execute the pipelined version of the loop. Normally, a redundant loop would be generated. However, in this case, redundant loop generation has been suppressed via the –ms0/–ms1 option.

Solution

Add MUST_ITERATE pragma or .trip to provide more information on the minimum trip count

If adding –mh or using a higher value of –mhn could help, try the following suggestions:

- ❑ Use –pm program level optimization to gather more trip count information.
- ❑ Use the MUST_ITERATE pragma or the .trip directive to provide minimum trip count information.

For more information...

See section 3.2.2.3, *Performing Program Level Optimization (–pm Option)*, on page 3-14.

See section 3.4.3.3, *Communicating Trip Count Information to the Compiler*, on page 3-44.

See section 6.2.5, *The .trip Directive*, on page 6-8.

Register is Live Too Long

Description

Sometimes the compiler finds a valid software pipeline schedule but one or more of the values is live too long. Lifetime of a register is determined by the cycle a value is written into it and by the last cycle this value is read by another instruction. By definition, a variable can never be live longer than the ii of the loop, because the next iteration of the loop will overwrite that value before it is read.

After this message, the compiler prints out a detailed description of which values are live to long:

```
ii = 11 Register is live too long
|72| –> |74|
|73| –> |75|
```

The numbers 72, 73, 74, and 75 correspond to line numbers and they can be mapped back to the offending instructions.

Solution

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.

For more information...

See section 6.10.4.1, *Split–Join–Path Problems*, on page 6-104.

Too Many Predicates Live on One Side

Description

The C6000 has predicate, or conditional, registers available for use with conditional instructions. There are 5 predicate registers on the ’C62x and ’C67x, and 6 predicate registers on the ’C64x. There are two or three on the A side and three on the B side. Sometimes the particular partition and schedule combination, requires more than these available registers.

Solution

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. For the ’C62x and ’C67x, 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.

Too Many Reads of One Register

Description

The ’C62x and ’C67x cores can read the same register a maximum of 4 times per cycle. The ’C64x core can read the same register any number of times per cycle. If the schedule found happens to produce code where a single register is read more than 4 times in a given cycle, the schedule is invalidated. This problem is very rare and only occurs on the ’C67x due to some floating point instructions that have multiple cycle reads.

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

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.

For more information...

See section 6.9, *Loop Unrolling (in Assembly)*, on page 6-94.

See section 3.4.3.4, *Loop Unrolling (in C)*, on page 3-45.

Trip var. Used in Loop – Can't Adjust Trip Count

Description

If the loop counter (named trip counter because of the number of trips through a loop) is modified within the body of the loop, it typically cannot be converted into a downcounting loop (needed for software pipelining on the 'C6000). If possible, rewrite the loop to not modify the trip counter by adding a separate variable to be modified.

The fact that the loop counter is used in the loop is actually determined much earlier in the loop qualification stage of the compiler. Why did the compiler try to schedule this anyway? The reason has to do with the –mh option. This option allows for extraneous loads and facilitates epilog removal. If the epilog was successfully removed, the loop counter can sometimes be altered in the loop and still allow software pipelining. Sometimes, this isn't possible after scheduling and thus the feedback shows up at this stage.

Solution

Replicate the trip count variable and use the copy inside the loop so that the trip counter and the loop reference separate variables.

Use the –mh option.

For more information...

See section 3.4.3.6, *What Disqualifies a Loop From Being Software Pipelined*, on page 3-50.

A.3 Investigative Feedback

Loop Carried Dependency Bound is Much Larger Than Unpartitioned Resource Bound

Description

If the loop carried dependency bound is much larger than the unpartitioned resource bound, this can be an indicator that there is a potential memory alias disambiguation problem. This means that there are two pointers that may or may not point to the same location, and thus, the compiler must assume they might. This can cause a dependency (often between the load of one pointer and the store of another) that does not really exist. For software pipelined loops, this can greatly degrade performance.

Solution

Use –pm program level optimization to reduce memory pointer aliasing.

Add restrict declarations to all pointers passed to a function whose objects do not overlap.

Use –mt option to assume no memory pointer aliasing.

Use the .mdep and .no_mdep assembly optimizer directives.

If the loop control is complex, try the -mh option.

For More Information...

See section 3.2.2.3, *Performing Program–Level Optimization (–pm Option)*, on page 3-14.

See section 3.2.2.1, *The const Keyword*, on page 3-9.

See section 3.2.2.2, *The restrict Keyword*, on page 3-13.

See section 3.2.2, *Memory Dependencies*, on page 3-7.

See Appendix B, *Memory Alias Disambiguation*, on page B-1.

See section 6.2, *Assembly Optimizer Options and Directives*, on page 6-4.

Two Loops are Generated, One Not Software Pipelined

Description

If the trip count is too low, it is illegal to execute the software pipelined version of the loop. In this case, the compiler could not guarantee that the minimum trip count would be high enough to always safely execute the pipelined version. Hence, it generated a non-pipelined version as well. Code is generated, so that at run-time, the appropriate version of the loop will be executed.

Solution

Check the software pipeline loop information to see what the compiler knows about the trip count. If you have more precise information, provide it to the compiler using one of the following methods:

❑ Use the MUST_ITERATE pragma to specify loop count information in c code.

❑ Use the .trip directive to specify loop count information in linear assembly.

Alternatively, the compiler may be able to determine this information on its own when you compile the function and callers with –pm and –op2.

For More Information...

See section 3.4.3.3, *Communicating Trip Count Information to the Compiler*, on page 3-44.

See section 6.2.5, *The .trip Directive*, on page 6-8.

See section 3.2.2.3, *Performing Program–Level Optimization (–pm Option)*, on page 3-14.

Uneven Resources

Description

If the number of resources to do a particular operation is odd, unrolling the loop is sometimes beneficial. If a loop requires 3 multiplies, then a minimum iteration interval of 2 cycles is required to execute this. If the loop was unrolled, 6 multiplies could be evenly partitioned across the A and B side, having a minimum ii of 3 cycles, giving improved performance.

Solution

Unroll the loop to make an even number of resources.

For More Information...

See section 6.9, *Loop Unrolling (in Assembly)*, on page 6-94.

See section 3.4.3.4, *Loop Unrolling (in C)*, on page 3-45.

Larger Outer Loop Overhead in Nested Loop

Description

In cases where the inner loop count of a nested loop is relatively small, the time to execute the outer loop can start to become a large percentage of the total execution time. For cases where this significantly degrades overall loop performance, unrolling the inner loop may be desired.

Solution

Unroll the inner loop.

Make one loop with the outer loop instructions conditional on an inner loop counter

For More Information

See Chapter 6, *Loop Unrolling (In C) (In Assembly)*, on page 6-118.

See section 6.14, *Outer Loop Conditionally Executed With Inner Loop*, on page 6-136.

There are Memory Bank Conflicts

Description

In cases where the compiler generates 2 memory accesses in one cycle and those accesses are either 8 bytes apart on a ’C620x device, 16 bytes apart on a ’C670x device, or 32 bytes apart on a ’C640x device, AND both accesses reside within the same memory block, a memory bank stall will occur. To avoid this degradation, memory bank conflicts can be completely avoided by either placing the two accesses in different memory blocks or by writing linear assembly and using the .mptr directive to control memory banks.

Solution

Write linear assembly and use the .mptr directive

Link different arrays in separate memory blocks

For More Information

See section 6.2.4, *The .mptr Directive*, on page 6-5.

See section 6.9, *Loop Unrolling (in Assembly)*, on page 6-94.

See section 3.4.3.4, *Loop Unrolling (in C)*, on page 3-45.

See section 6.12, *Memory Banks*, on page 6-118

T Address Paths Are Resource Bound

Description

T address paths defined the number of memory accesses that must be sent out on the address bus each loop iteration. If these are the resource bound for the loop, it is often possible to reduce the number of accesses by performing word accesses (LDW/STW) for any short accesses being performed.

Solution

Use word accesses for short arrays; declare int * (or use _nassert) and use mpy intrinsics to multiply upper and lower halves of registers

Try to employ redundant load elimination technique if possible

Use LDW/STW instructions for accesses to memory

For More Information...

See section 3.4.2, *Using Word Accesses for Short Data (C)*, on page 3-27.

See section 6.11, *Redundant Load Elimination*, on page 6-110.

See section 6.4, *Using Word Access for Short Data (Assembly)*, on page 6-19.

Appendix B

Memory Alias Disambiguation

This appendix is a tutorial and practical treatment on the problem of memory alias disambiguation on the ’C6000. If you write ’C6000 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/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

B.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/C++ compiler handles memory alias disambiguation, and the differences between memory alias disambiguation and memory bank conflict detection.

Note that this appendix describes the ’C6000 code generation tools for release 2.10 or greater.

B.2 Background

B.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 B–1. Dependence Table

		Instruction 2 Reference	
		Read	**Write**
Instruction 1 Reference	**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:    STW    A3,*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:    STW    A3,*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.

B.2.2 Dependence Graphs

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

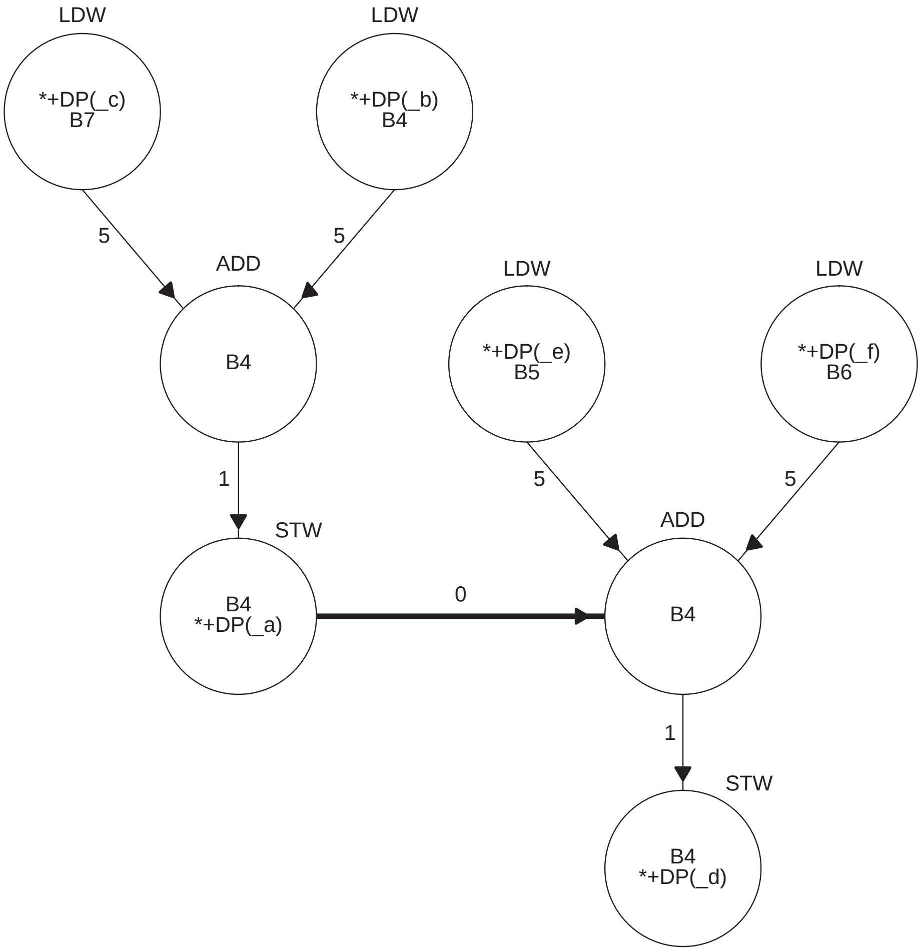
From the C fragment:

```
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|
ADD   B7,B4,B4     ; |5|
STW   B4,*+DP(_a)  ; |5|
;----------------------------------------------------------
; 6 | d = e + f;
;----------------------------------------------------------
LDW   *+DP(_e),B5  ; |6|
LDW   *+DP(_f),B6  ; |6|
ADD   B6,B5,B4     ; |6|
STW   B4,*+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 ’C6000 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.

B.2.3 Data Dependence in Loops

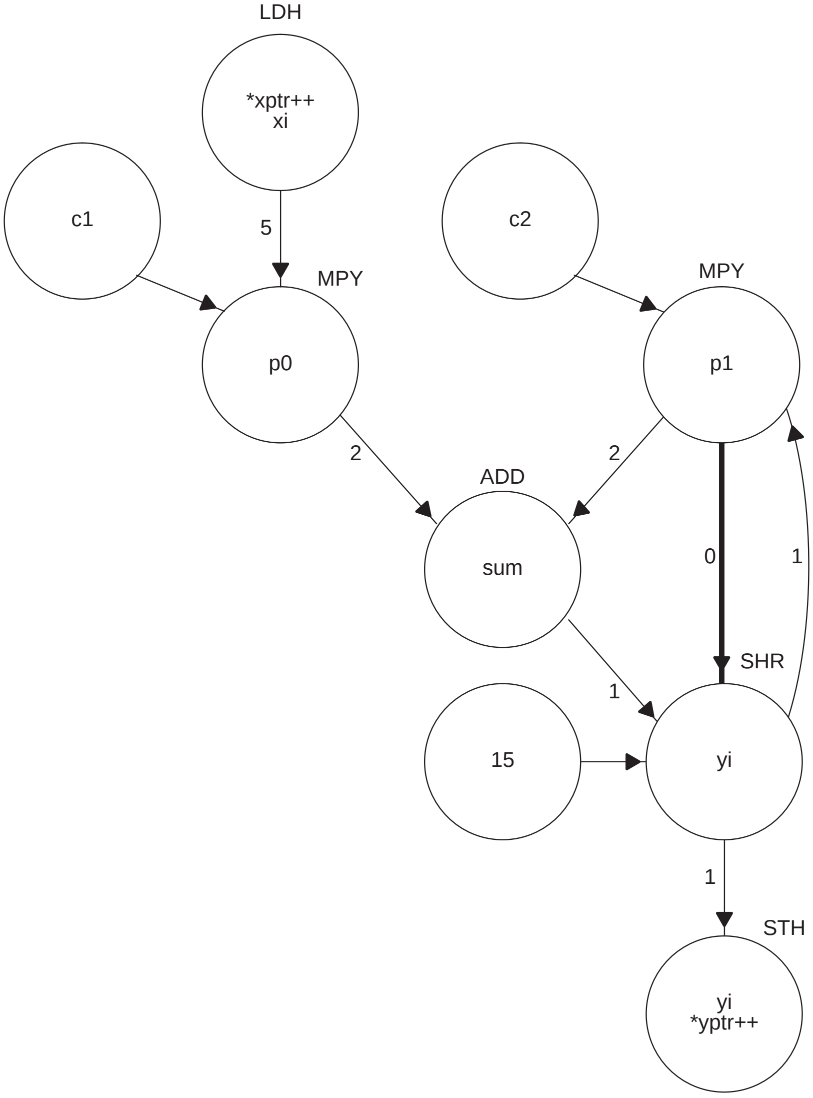
So far, we have only looked at relationships between instructions in a simple straight-line block of code. Considering dependence between instructions in a loop requires some extensions to those concepts.

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
   MPY   c1,xi,p0
   MPY   c2,yi,p1  ; reads yi from prior iteration
   ADD   p0,p1,sum
   SHR   sum,15,yi ; writes yi for next iteration
   STH   yi,*yptr++
; decrement and branch to loop
```

Here is the dependence graph:



Consider the instructions which reference `yi`. Note the flow dependence, carried to the next iteration by the loop, on yi from the `SHR` to `MPY`, because the `SHR` writes a value to `yi` which `MPY` reads. Also note the anti-dependence, not carried by the loop, on `yi` from the `MPY` to the `SHR`, because the `MPY` must read `yi` 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 ’C6000 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/C++ compiler, on the other hand, performs very sophisticated dependence analysis on nested loops.

B.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.

B.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:   STW   A3,*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.

B.3 Tools Solution

B.3.1 Overview of the Assembly Optimizer Solution

In a few cases, 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. This is the recommended approach.

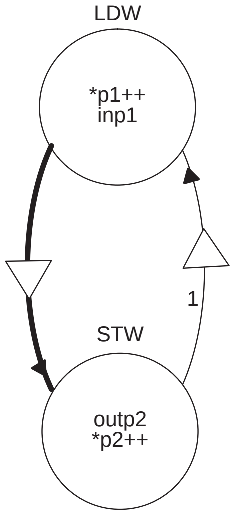
B.3.2 Default Presumption is Pessimistic

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:   STWoutp2,*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?

B.3.3 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.

B.3.4 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”
...
STW   outp2, *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.

B.4 Examples of Memory Alias Disambiguation

B.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.

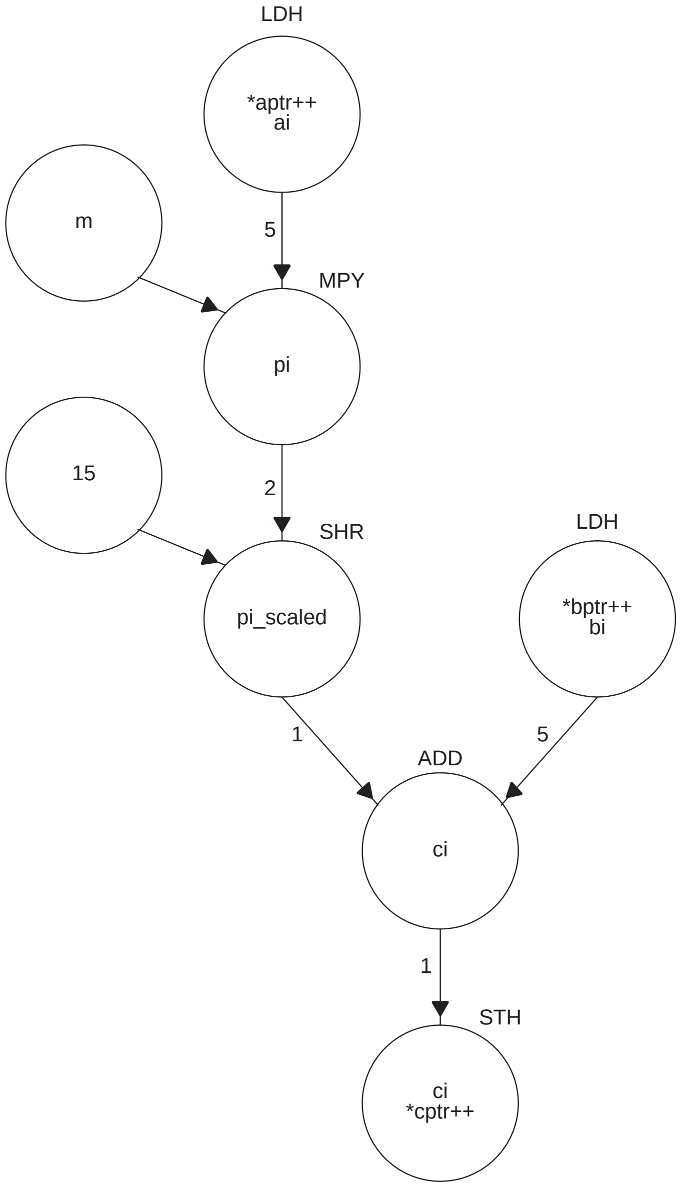
This linear assembly function is adapted from the weighted vector sum example. A typical call to this function could look like:

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

Here is the source:

```
wvs:  .cproc aptr, bptr, cptr, m
      .reg   cntr, ai, bi, pi, pi_scaled, ci
      MVK    100,cntr
      .no_mdep       ; presume no memory aliases
loop: .trip  100
      LDH    *aptr++,ai
      LDH    *bptr++,bi
      MPY    m,ai,pi
      SHR    pi,15,pi_scaled
      ADD    pi_scaled,bi,ci
      STH    ci,*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.

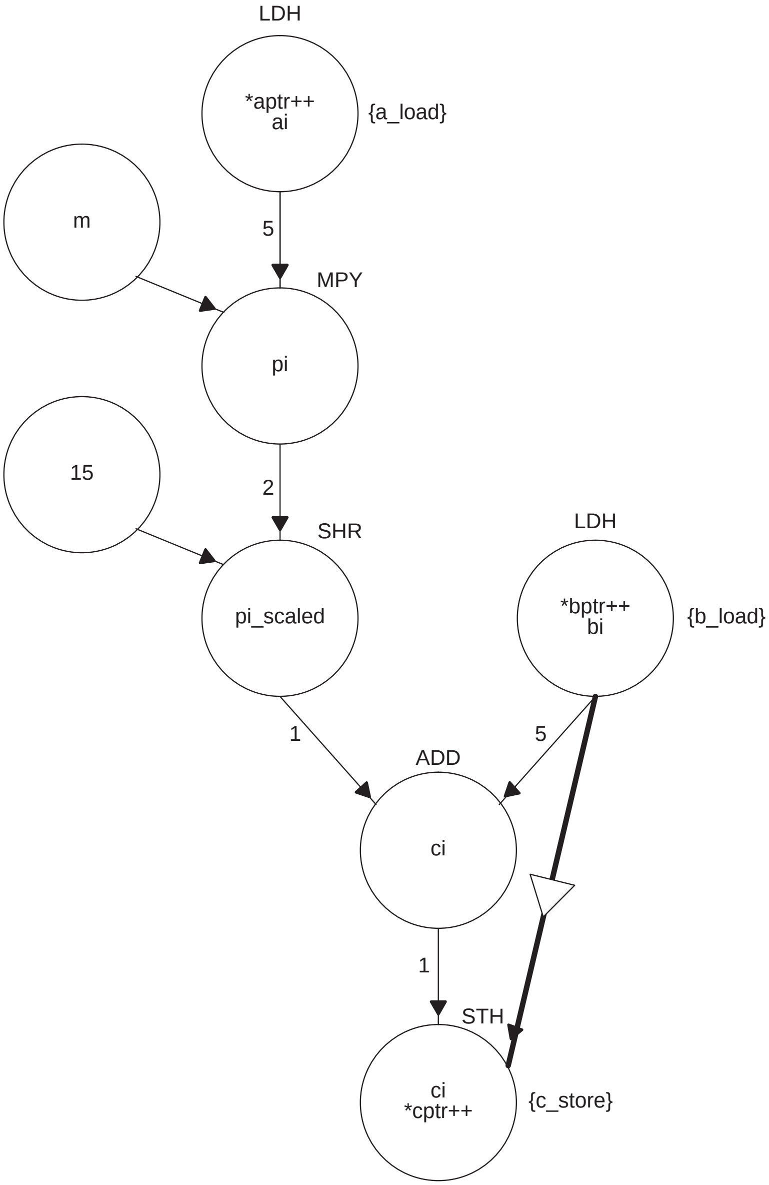
Suppose you know some calls to wvs pass the same array as the b input array and the c output array, just to save some space:

```
.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
      MPY   m,ai,pi
      SHR   pi,15,pi_scaled
      ADD   pi_scaled,bi,ci
      STH   ci,*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 `.mdep` 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:

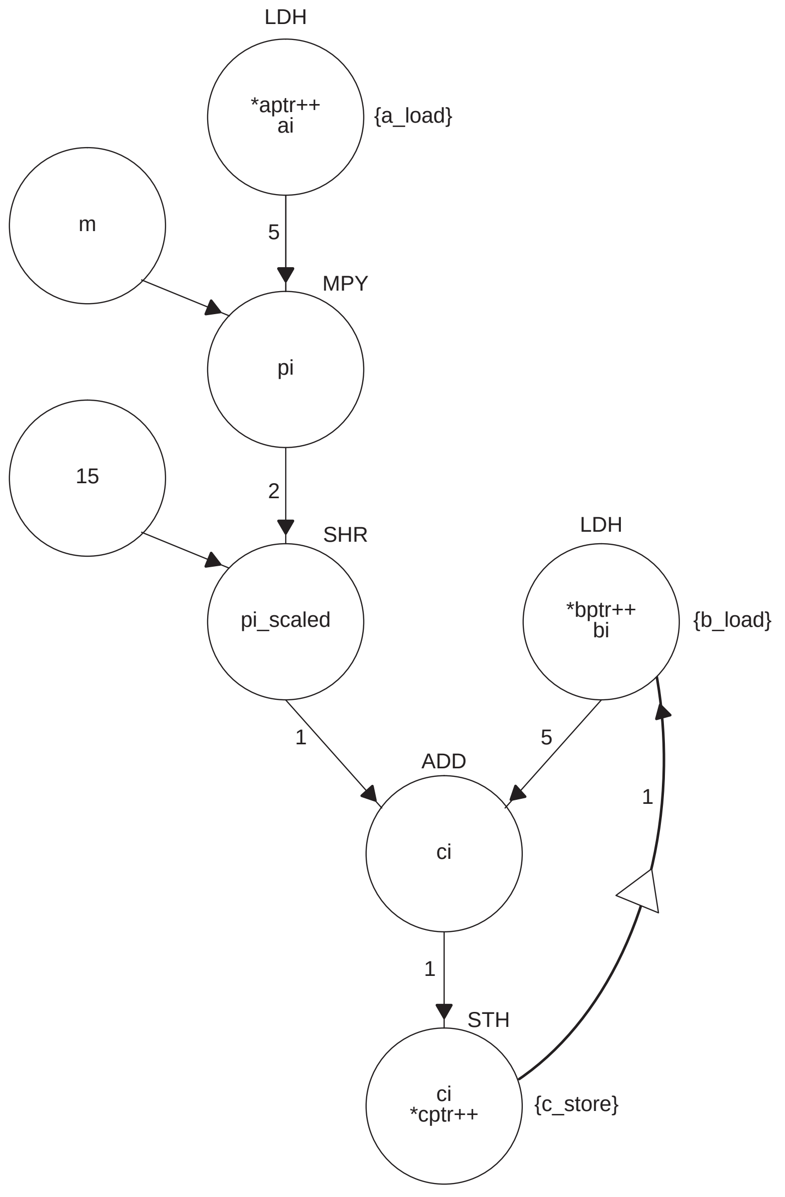
```
ADD   b,2,c       ; c points to b[1]
.call wvs(a, b, c, m)
```

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 iteration, 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.

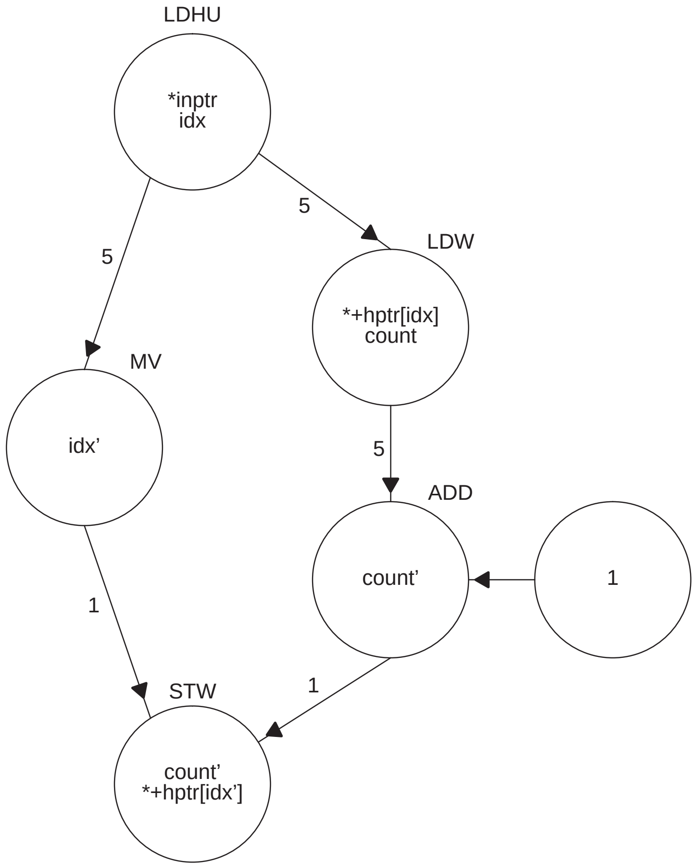
B.4.2 Handling Indexed Addressing Mode

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
      ADD   count,1,count
      STW   count,*+hptr[idx]
[len] SUB   len,1,len
[len] B  loop
      .endproc
```

Here is the dependence graph:

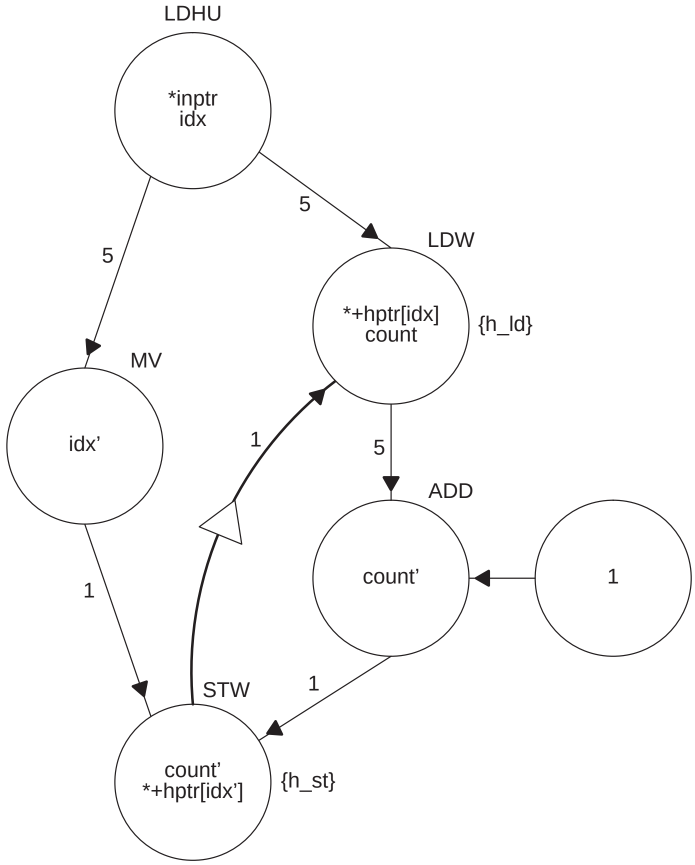


Note the assembly optimizer splits the lifetime of the `idx` register by adding the `MV` instruction; the new variable is shown as `idx'`. The lifetime of `count` is similarly split at the `ADD` instruction.

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 `.mdep:`

```
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.

B.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
mcbsp0_dxr: .cproc
      MVKL  MCBSP0_BASE,B4     ; load base of McBSP0 regs
      MVKH  MCBSP0_BASE,B4
      ...
      STW   B5,*+B4(0x10)   ; init XCR for transfer
      STW   B6,*+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
STW    B5,*+B4(0x10) {wrt_xcr}
STW    B6,*+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.

B.5 C/C++ Compiler and Alias Disambiguation

The C/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 *TMS320C6000 Opimizing C/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/C++ code.

The command line options:

```
cl6x –pm –o3
```

have several effects, of which improved alias disambiguation is 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.

Correct use of the `const` and `restrict` keywords helps the alias disambiguation problem. The `const` keyword tells the compiler that the data object will not be modified, 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. The `restrict` keyword tells the compiler that within the scope of a particular pointer, only the object pointed to can be accessed by that pointer.

B.6 Memory Alias Disambiguation versus Memory Bank Conflict Detection

If a ’C6000 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 `.mptr` for the purpose of solving this problem. See the *TMS320C6000 Optimizing C/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 `.mdep` directives without any regard to your `.mptr` directives, and vice versa.

B.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 a few 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/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.

Index