TMS320C62xx Programmer's Guide

Literature Number: SPRU198
January 1997

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This manual serves as a reference for programming TMS320C62xx digital signal processor (DSP) devices.

Chapter 1 provides a fundamental description of coding development and the ’C62xx architecture.

Chapters 2–4 discuss optimization methods for C and assembly code. These chapters provide a discussion of some theory behind these methods and give code examples. This information can help you to choose the most appropriate optimization techniques for your code.

Chapter 5 provides code examples that give you additional material use in preparing your own programs.

Related Documentation From Texas Instruments

The following books describe the TMS320C62xx devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

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

TMS320C62xx CPU and Instruction Set Reference Guide (literature number SPRU189) describes the 'C62xx CPU architecture, instruction set, pipeline, and interrupts for the TMS320C62xx digital signal processors.

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

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.

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

TMS320C62xx Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C62xx digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port, serial ports, direct memory access (DMA), clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6x Software Tools Getting Started Guide (literature number SPRU185) describes how to install the TMS320C6x assembly language tools, the C compiler, the simulator, and the C source debugger. Installation instructions for SunOS™, Solaris™, Windows™ 95, and Windows NT™ systems are given.

TMS320C6201 Digital Signal Processor Data Sheet (literature number SPRS051) describes the features of the TMS320C6xx and provides pinouts, electrical specifications, and timings for the device.

Trademarks

If You Need Assistance . . .

❑ World-Wide Web Sites

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

❑ North America, South America, Central America

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

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

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

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

❑ Asia-Pacific

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

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

❑ Japan

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

❑ Documentation

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

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

Email: comments@books.sc.ti.com

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

Contents

Figures

Tables

Examples

Chapter 1

Introduction

This chapter introduces some features of the ’C62xx microprocessor and discusses the basic process for creating code.

Topic **Page**

1.1 TMS320C62xx Architecture

The ’C62xx 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 ’C62xx DSPs are based on the ’C62xx CPU, which consists of:

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

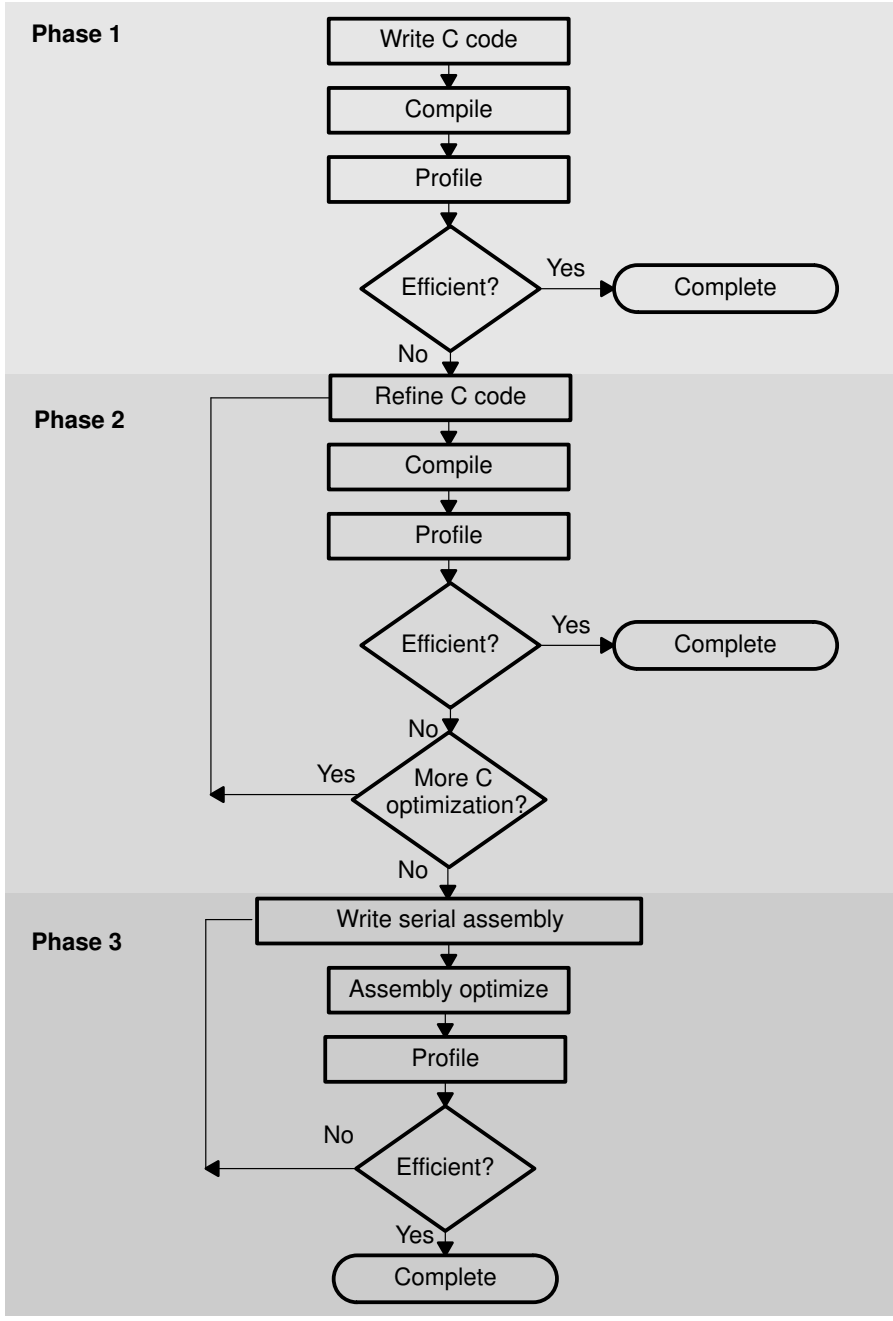
1.2 TMS320C62xx Pipeline

The ’C62xx 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

You can achieve the best performance from your 'C62xx code if you follow this flow when you are writing and debugging your code:



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

Phase	Goal
1	Begin by developing C code and using the ’C6x compiler. This requires no knowledge of the ’C62xx, however you could use the ’C6x profiling tools that are described in the *TMS320C6x C Source Debugger User’s Guide* to pinpoint any inefficient areas of your code.
2	Refine your C code using procedures such as compiler options, intrinsics, statements, data types, and code trans formations. You can realize substantial gains from the perfor mance of your C code by using these simple methods. Proceed to the next phase if you are still dissatisfied with the performance of your code.
3	Extract the inefficient areas from your C code and rewrite them in assembly optimizer source code.

Chapter 2

Optimizing C Code

The goal in this chapter is to maximize C performance by using compiler options, intrinsics, and code transformations. This chapter discusses various ways to optimize your C code by providing specific examples and discussing topics such as:

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

Topic **Page**

2.1 Analyzing C Code Performance

The following techniques can help you to analyze the performance of specific code regions:

- Use the *clock()* and *printf()* functions in C to time the performance of specific code regions. You can use the standalone loader (load6x) to run the code shown in Example 2–1.
- Use the profiler mode in the debugger as explained in the *TMS320C6x C Source Debugger User's Guide.*
- Use breakpoints, the clk register, and the runb command in the debugger as described in the *TMS320C6x C Source Debugger User's Guide.*

Most often the critical performance areas in your code are loops. The easiest way to optimize a loop is by extracting it into a separate file that can be transformed, recompiled, and run standalone. As you use the techniques described in this chapter to optimize your C code, you can then evaluate the results by running the code and looking at the instructions generated by the compiler.

Example 2–1. Using the clock() Function

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

short a[100], b[100], c[100];
void vecsum(short *a, short *b, short *c, int);

main()
{
    clock_t overhead, start, stop;

    /*************************************************************************/
    /* COMPUTE THE OVERHEAD OF CALLING CLOCK                                 */
    /*************************************************************************/
    start    = clock();
    stop     = clock();
    overhead = stop - start;

    /*************************************************************************/
    /* CALL AND TIME THE VECSUM ROUTINE.                                     */
    /*************************************************************************/
    start = clock();
    vecsum(a, b, c, 100);
    stop  = clock();

    printf("vecsum cycles: %d\n", stop - start - overhead);
}
```

2.2 Compiler Options

Table 2–1 defines the options mentioned in this chapter.

- Although –o3 is preferable, at a minimum use the –o option.
- Use the –pm (program level optimization) option for as much of your program as possible.

For a complete description of these and other options, see the *TMS320C6x Optimizing C Compiler User's Guide*.

Table 2–1. Subset of Compiler Options

Option	Description
–o	Enables software pipelining and other optimizations in the compiler.
–pm	Enables program level optimization.
–mt	Enables the compiler to make assumptions that allow it to be more aggressive with certain optimizations.
–ms	Ensures that redundant loops are not generated.
–k	Keeps the assembly file so that you can inspect it.

2.3 Tips on Data Types

The ’C6x compiler defines different sizes for each data type:

- short 16 bits
- int 32 bits
- long 40 bits

Based on the size of each data type, follow these guidelines:

- Avoid code that assumes that *int* and *long* types are the same size because the ’C6x compiler uses 40-bit operations for *long* values.
- Use the *short* data type for multiplication inputs whenever possible because this data type provides the most efficient use of the 16-bit multiplier in the ’C62xx.
- Use *int* or *unsigned int* types for loop counters.

2.4 Using Intrinsics

One way to optimize your C code is by using *intrinsics,* which are special functions that map directly to inlined 'C62xx instructions.

- Intrinsics are specified with a leading underscore and are accessed by calling them as you do a function.
- All instructions that are not easily expressed in C code are supported as intrinsics by the 'C6x compiler.

Traditionally, saturated addition can be expressed in C code only by writing a multicycle function, such as the one in Example 2–2. Example 2–3 shows code that uses the *_sadd()* intrinsic, which results in a single 'C62xx instruction.

Table 2–2 lists the 'C62xx intrinsics. For more information, see the *TMS320C6x Optimizing C Compiler User's Guide*.

Example 2–2. 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);
}
```

Example 2–3. Saturated Add With Intrinsics

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

Table 2–2. TMS320C6x C Compiler Intrinsics

C Compiler Intrinsic	Assembly Instruction	Description
int **_add2(**int *src1*, int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add will not affect the upper half add.
uint **_clr(**uint *src2,* uint *csta*, uint *cstb***);**	**CLR**	Clears specified field in src2. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.
int **_ext(**uint *src2,* uint *csta*, int *cstb***);**	**EXT**	Extracts 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 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 **_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.
int **_mpy(**int *src1,* int *src2***);** int **_mpyus(**uint *src1,* int *src2***);** int **_mpysu(**int *src1,* uint *src2***);** uint **_mpyu(**uint *src1,* uint *src2***);**	**MPY** **MPYUS** **MPYSU** **MPYU**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.
int **_mpyh(**int *src1,* int *src2***);** int **_mpyhus(**uint *src1,* int *src2***);** int **_mpyhsu(**int *src1,* uint *src2***);** uint **_mpyhu(**uint *src1,* uint *src2***);**	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.
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.

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

C Compiler Intrinsic	Assembly Instruction	Description
void **_nassert(**int**);**		Generates no code. Tells the optimizer that the expression declared with the assert function is true; this gives a hint to the optimizer as to what optimizations might be valid.
uint **_norm(**int *src2***);** uint **_lnorm(**long *src2***);**	**NORM**	Returns the first nonredundant sign bit of src2.
int **_sadd(**int *src1,* int *src2***);** long **_lsadd(**int *src1,* long *src2***):**	**SADD**	Adds src1 to src2 and saturates the result. Returns the result.
long **_sat(**int *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 initial src2 value. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.
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.
uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result.
int **_ssub(**int *src1,* int *src2***);** long **_lssub(**int *src1,* long *src2***):**	**SSUB**	Subtracts src2 from src1, saturates the result size, and returns the result.
uint **_subc(**uint *src1,* uint *src2***);**	**SUBC**	Conditionally subtracts divide step and returns the result.
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.

2.5 Memory Dependencies

To schedule instructions in parallel, the compiler must determine the relationships, or dependencies, between instructions. A *dependency* means that one instruction must occur before another. Because only independent instructions can execute in parallel, dependencies inhibit parallelism. The compiler uses these guidelines when scheduling instructions:

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

You can use the following techniques to help the compiler determine which instructions are independent:

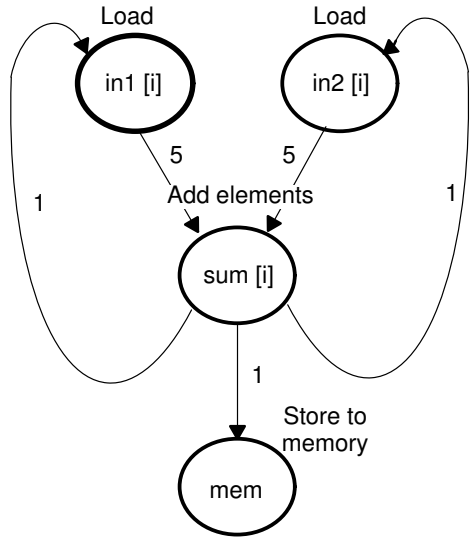
- Use the –pm (program-level optimization) option, which gives the compiler a global view of the whole program or module and allows it to be more aggressive ruling out dependencies.
- Use the *const* keyword to indicate which objects are not changed by a function.
- Use the –mt compiler, which allows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

Example 2–4 shows the C code for a basic vector sum. Figure 2–1 shows a dependency graph for the basic vector sum.

Example 2–4. 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 2–1. Dependency Graph for Vector Sum #1



The dependency graph in Figure 2–1 shows the following:

- 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 since they both point to the same memory location:

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

Although within a single iteration the reads to *in1* and *in2* finish before the store to *sum*, the 'C6x compiler is using software pipelining to execute multiple iterations in parallel and, therefore, must determine memory dependencies that exist across loop iterations.

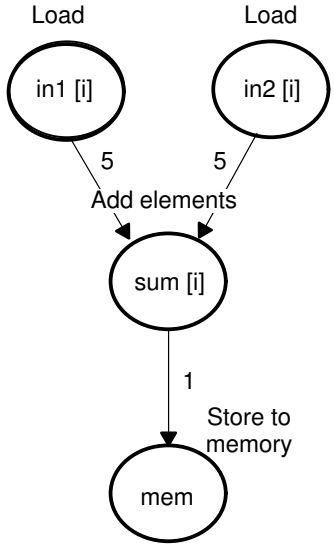
To help the compiler, you can qualify an object with the *const* keyword, which indicates that a variable or the memory referenced by a variable will not be changed by the function.

Example 2–5 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 2–2 shows the revised dependency graph for the code in the inner loop.

Example 2–5. 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 2–2. Dependency Graph for Vector Sum #2



Example 2–6 shows the output of the compiler for the vector sum. The compiler finds better schedules when dependency paths are eliminated between instructions. For this loop, the compiler found a software pipeline with a two-cycle kernel (compared with seven for the previous loop).

Example 2–6. Compiler Output for Vector Sum Code

```
L14:                    ; PIPE LOOP KERNEL

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

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

Another way to eliminate memory dependences is to use the –mt option, which allows the compiler to make assumptions that can eliminate memory dependency paths. For example, with this option enabled when compiling the code in Example 2–4, the compiler assumes that *in1* and *in2* do not alias memory pointed to by *sum*, and therefore, would eliminate memory dependencies among the instructions that access those variables.

2.6 Trip Count Issues

A *trip count* is the number of times that a loop executes, and the trip counter is the variable used to count each iteration. When the trip counter reaches a limit equal to the trip count, the loop terminates. The structure of a software pipeline requires the execution of a minimum number of loop iterations (a minimum trip count) in order to fill, or *prime*, the pipeline.

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

For example, the optimizer transforms the loop in Example 2–7(a) to something like the code in Example 2–7(b):

Example 2–7. Trip Counters

(a) Original code

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

(b) Optimized code

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

The minimum trip count for a software pipeline is determined by the number of iterations executing in parallel.

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

  In Example 2–5, the compiler cannot determine if the loop always executes more than the minimum trip count and, therefore, generates two versions of the loop:

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

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

Two techniques communicate trip-count information to the compiler:

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

  Example 2–8 shows the vector sum code with an *_nassert* statement.

Example 2–8. Vector Sum With const Keywords and _nassert

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

    _nassert(N >= 10);

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

}
```

The compiler does not generate code for the *_nassert()* function. *_nassert()* simply communicates information to the compiler that helps it determine information about the range of a variable. In Example 2–8, *_nassert()* asserts that N is always at least 10.

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

2.7 Using Word Access for Short Data

The ’C62xx 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 ’C62xx intrinsics to operate on the data. For example, rewriting the *vecsum()* function (Example 2–9)to use word accesses doubles the performance of the loop. See Section 4.2, *Loading Two Data Values* with LDW for more information.

Example 2–9. Vector Sum With const Keywords, _nassert, Word Reads

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

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

  _nassert(N >= 20);

  for (i = 0; i < (N >> 1); i++)
    i_sum[i] = _add2(i_in1[i], i_in2[i]);
}
```

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

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

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

If a *vecsum()* 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 2–10 provides a generic *vecsum()* function that handles all types of data.

Example 2–10. Vector Sum With const Keywords, _nassert, Word Reads, Generic Version

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

  _nassert(N >= 20);

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

    for (i = 0; i < (N >> 1); i++)
      i_sum[i] = _add2(i_in1[i], i_in2[i]);

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

Other intrinsics that are useful for reading short data as words are the multiply intrinsics. Example 2–11 is a dot product example that reads word-aligned short data and uses the *_mpy()* and *_mpyh()* intrinsics:

- The *_mpyh()* intrinsic uses the 'C62xx instruction MPYH, which multiplies the high 16 bits of two registers, giving a 32-bit result.
- Two sum variables are used (*sum1* and *sum2*).

  Using only one sum variable would inhibit parallelism by creating a dependency between the write from the first *sum* calculation to the read in the second *sum* calculation. Within a small loop body, avoid writing to the same variable since it can inhibit parallelism and create dependences.

Example 2–11. 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;
}
```

Example 2–12 show an FIR filter that uses word reads of *short* data and *_mpyXX()* intrinsics. Example 2–13 shows an optimized version of the Example 2–12. 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 2–12. 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 2–13. FIR—Optimized Form

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

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

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

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

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

2.8 Loop Unrolling

Another technique that can improve performance is unrolling the loop to increase 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 the resources of the ’C62xx architecture.

Example 2–6 (the output of the compiler for the vector sum code in Example 2–5) shows that the loop produces a new *sum[i]* every two cycles:

- Three memory operations are being performed: a load for both *in1[i]* and *in[2]* and a store for *sum[i]*.
- Because only two memory operations can execute per cycle, two cycles are necessary to perform three memory operations.

The performance of a software pipeline is limited by the number of resources that can execute in parallel. In its word-aligned form (Example 2–9), the vector sum loop delivers two results every two cycles since the two loads and the store are all operating on two 16-bit values at a time.

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 2–14 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 2–14 is not simple loop unrolling where the loop body is simply replicated. The additional instructions use memory pointers offset to point midway into the input arrays and assumes that the additional arrays are a multiple of four *shorts* in size.

Example 2–14. Vector Sum With const Keywords, _nassert, Word Reads, and Unrolled

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

  _nassert(N >= 20);

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

Software pipelining is performed 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 2–15, 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 2–15. 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. Since 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 2–16.

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.

Example 2–16. 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);
  }
}
```

2.9 What Disqualifies a Loop From Being Software Pipelined

In a sequence of nested loops, the innermost loop is the only one that can be software pipelined.

The following restrictions apply to the software pipelining of loops:

- Although a software pipeline loop can contain intrinsics, it cannot contain function calls.
- You may not have a conditional break (early exit) in the loop.
- The loop must have a loop counter that counts down and that terminates at 0. One reason that you run the optimizer with the –o2 or –o3 option is to convert as many loops as possible into downcounting loops.
- If the trip counter is modified within the body of the loop, it typically cannot be converted into a downcounting loop. For example, the following code is not software pipelined

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

- A conditionally incremented loop control variable is not software-pipelined. The following is an example that would not be pipelined.

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

- The code size is too large and requires more than the 32 registers in the ’C62xx.
- A register value is “live-too-long”. See section 4.8, *Live-too-Long Issues*.
- If the loop has complex condition code within the body that requires more than the five ’C62xx condition registers.

Chapter 3

Structure of Assembly Code

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

- Labels
- Conditions
- Instructions
- Units
- Operands
- Comments

Topic **Page**

3.1 Labels

Labels identify a line of code or a variable and represent memory addresses that contain either an instruction or data.

Figure 3–1 shows the position of the label in a line of assembly code. The colon following the label is optional.

Figure 3–1. Labels in Assembly Code

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

Labels must meet the following conditions:

- The initial character of a label must be 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.

3.2 Parallel Bars

Figure 3–2. Parallel Bars in Assembly Code

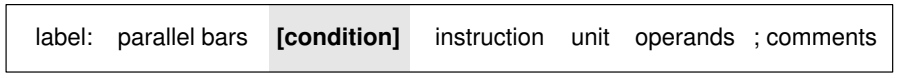
label:	**parallel bars**	[condition]	instruction	unit	operands	; comments

Instructions that execute in parallel with the previous instruction signify this with parallel bars (||). This field is left blank for instructions that do not execute in parallel with the previous instruction.

3.3 Conditions

Five registers in the ’C62xx are available for conditions: A1, A2, B0, B1, and B2. Figure 3–3 shows the position of a condition in a line of assembly code.

Figure 3–3. Conditions in Assembly Code



All ’C62xx 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 executes if ...
`[A1]`	A1 = 0
`[!A1]`	A1! = 0

3.4 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 list in Table 3–1.
- *Processor mnemonics* are the actual microprocessor instructions that execute at runtime and perform the operations in the program. Table 3–2 summarizes the ’C62xx mnemonics. Processor mnemonics must begin in column 2 or greater.

Figure 3–4 shows the position of the instruction in a line of assembly code.

Figure 3–4. Instructions in Assembly Code

Table 3–1. ’C62xx Directives

Directives	Description
.sect “name”	Creates section of information (data or code)
.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

Table 3–2. ’C62xx Mnemonics

Arithmetic	Multiply	Load/Store	Program Control	Bit Management	Logical	Pseudo/Other
ABS	MPY	LD	B	CLR	AND	IDLE
ADD	MPYH	MVK	B IRP	EXT	CMPEQ	MV
ADDA	MPYHL	MVKH	B NRP	LMBD	CMPGT	MVC
ADDK	MPYLH	ST		NORM	CMPLT	NOP
ADD2	SMPY	STP		SET	OR	ZERO
SADD					SHL	NEG
SAT					SHR	NOT
SSUB					SSHL	
SUB					XOR	
SUBA						
SUBC						
SUB2						

3.5 Units

The ’C62xx CPU contains eight functional units, which are shown in Figure 3–5.

Table 3–3. Functional Units and Descriptions

Functional Unit	Description
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations Left most 1, 0, bit counting for 32 bits Normalization count for 32 and 40 bits 32 bit logical operations
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40 bit shifts and 32-bit bit-field operations 32 bit logical operations, Branching Constant generation Register transfers to/from the control register file
.M unit (.M1, .M2)	16 x 16 bit multiplies
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation

Figure 3–5. ’C62xx Functional Units

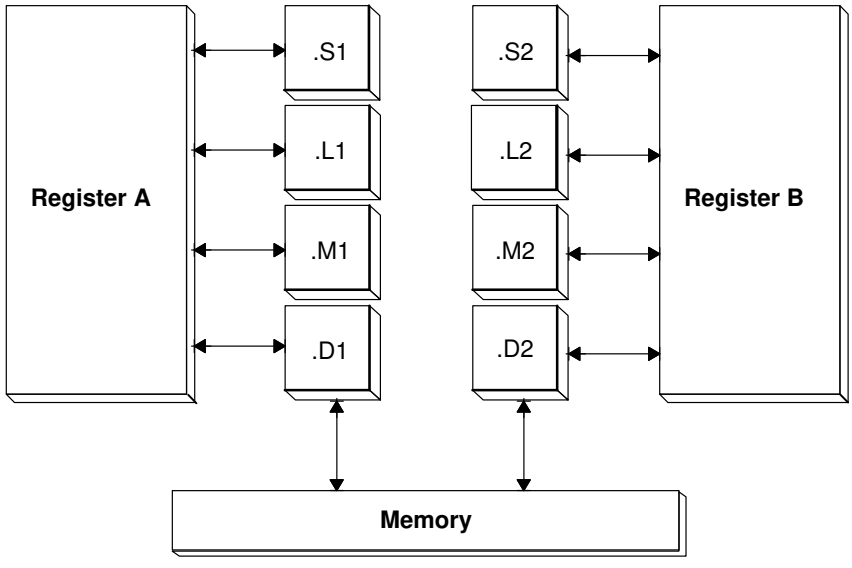
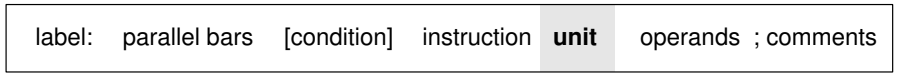


Figure 3–6 shows the position of the unit in a line of assembly code.

Figure 3–6. Units in the Assembly Code



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

3.6 Operands

The ’C62xx architecture requires that memory reads and writes move data between memory and a register. Figure 3–7 shows the position of the operands in a line of assembly code.

Figure 3–7. Operands in the Assembly Code

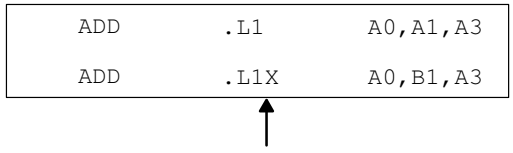
label:	parallel bars	[condition]	instruction	unit	**operands**	; comments

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.
- ❏ Destination operands must be on same side as one source.
- ❏ One source operand per execute packet can come from the other side.

  When an operand comes from the other register file, the unit includes an X as shown in Figure 3–8.

Figure 3–8. Operands in Instructions



The ’C62xx instructions use three types of operands to access data:

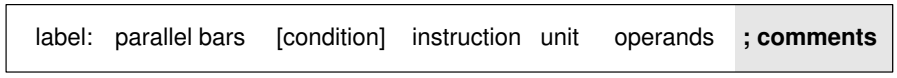
- ❏ *Register operands* indicate a register that contains the data.
- ❏ *Constant operands* specify the data within the assembly code.
- ❏ *Pointer operands* contain addresses of data values.

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

3.7 Comments

As with all programming languages, comments provide code documentation. Figure 3–9 shows the position of the unit in a line of assembly code.

Figure 3–9. Comments in Assembly Code



The following are guidelines for using comments in assembly code:

- ❏ May begin in any column when preceded by a semicolon (;)
- ❏ Must begin in first column when preceded by an asterisk (*)
- ❏ Not required, but recommended

Chapter 4

Optimizing Assembly Code

This chapter describes methods that help to develop more efficient assembly language programs. The primary purpose of this chapter is to help you understand the code produced by the assembly optimizer and to help you perform manual optimization.

Topic **Page**

4.1 Writing Parallel Code

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

4.1.1 Dot-Product C Code

The C code in Example 4–1 represents a dot-product algorithm that includes the following operations:

- ❏ Multiply each element in array *a* by the corresponding element in array *b*
- ❏ Accumulate each product in *sum*

Example 4–1. 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);
}
```

4.1.2 Translating C Code to ’C62xx Instructions

Example 4–2 shows the translation of the C code to ’C62xx instructions and illustrates the following decisions that affect the assignment of units:

- ❏ The load halfword (LDH) instructions increment through the *a* and *b* arrays. Each LDH does a post increment on the pointer and, therefore, on each iteration sets the pointer to the next halfword (16 bits) in the array. The load instructions must use a .D unit.
- ❏ All multiply (MPY) instructions must use a .M unit.
- ❏ The ADD instruction accumulates the total of the results from the multiply (MPY) instruction.
- ❏ The subtract (SUB) instruction decrements the loop counter.
- ❏ The branch (B) instruction is conditional on the loop counter, A1, and executes only until A1 is 0. Branch (B) instructions must use a .S unit.

Example 4–2. Translating C Code to ’C62xx Instructions

(a) 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);
}
```

(b) ’C62xx instructions

```
         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
```

4.1.3 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.

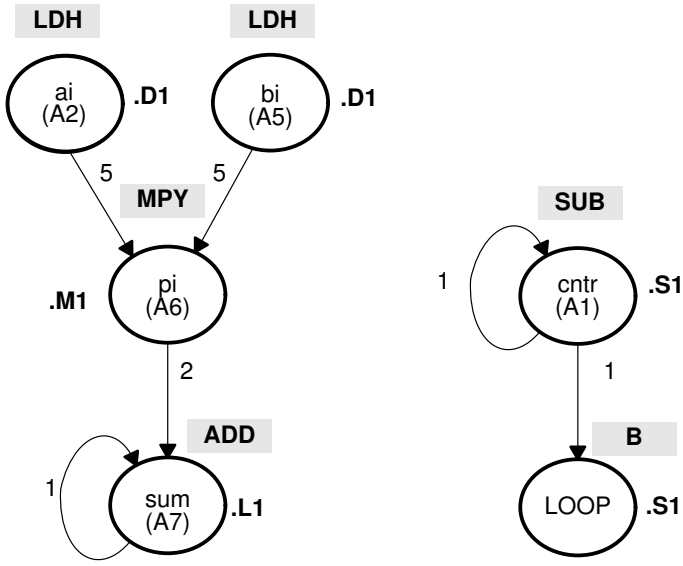
Figure 4–1 shows the dependency graph for the dot-product assembly instructions:

- The two LDH instructions, which write the values of *ai* and *bi*, are parents of the MPY instruction.
- The MPY instruction, which writes the product, *pi*, is the parent of the ADD instruction.
- 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.

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

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

Figure 4–1. Dependency Graph for Dot Product



Example 4–3. Serial Assembly

```
        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
```

4.1.4 Serial vs. Parallel Code

Example 4–4 shows an example of a dot-product loop written serially. The NOP instructions allow for the delay slots of the LDH, MPY, and branch 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 4–4. Dot-Product Serial Assembly

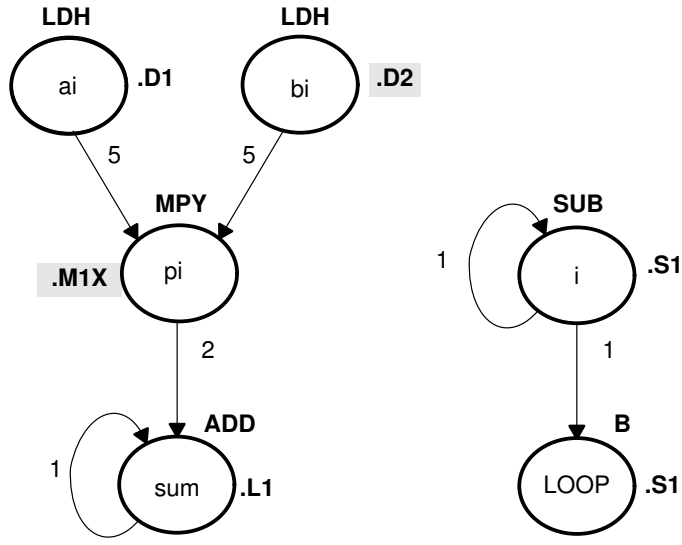
```
       MVK    .S1    100, A1       ; set up loop counter
       ZERO   .L1    A7            ; zero out accumulator
LOOP:
       LDH    .D1    *A4++,A2      ; load ai from memory
       LDH    .D1    *A3++,A5      ; load bi from memory
       NOP    4                    ; delay slots for LDH
       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
```

To help improve the performance of this loop, you can assign the functional units to execute the code in parallel, as shown in the dependency graph in Figure 4–2.

- Since 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 accomplish this, you can allocate the following functional units:
  - *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, the MPY uses the 1X cross path.
- 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 at the end of the code in Example 4–4. 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.

Executing this dot-product code in Example 4–5 requires eight cycles for each iteration plus one cycle to set up the loop counter and initialize the accumulator; 100 iterations require 801 cycles.

Figure 4–2. Dependency Graph for Parallel Assembly



Example 4–5. Dot-Product Parallel Assembly

```
       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
```

4.1.5 Comparing Performance of Serial and Parallel Code

Table 4–1 compares the performance of the serial code with the parallel code.

Table 4–1. Comparison of Serial and Parallel Code

Code Example	100 Iterations	Cycle Count
Example 4–4 Serial Assembly	2 + 100 × 16	1602
Example 4–5 Parallel Assembly	1 + 100 × 8	801

4.2 Loading Two Data Values With LDW

In writing the parallel code in section 4.1, you used an LDH instruction to read *a[i]*. Because *a[i]* and *a[i + 1]* are next to each other in memory, you can use the load word (LDW) instruction to read *a[i]* and *a[i + 1]* at the same time and to load both into a single 32-bit register.

4.2.1 Unrolled Dot-Product C Code

The C code in Example 4–6 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.

Example 4–6. Unrolled Dot-Product C Code

```
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);
}
```

4.2.2 Translating the Unrolled C Code to ’C62xx Instructions

Example 4–7 shows the ’C62xx instructions that execute the unrolled 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, which is the same as the one in Example 4–7, 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]*.

Note:

This is only true for the case when the ’C62xx 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]. Refer to the *TMS320C62xx Peripherals Reference Guide* for more information.

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

Example 4–7. List of Symbolic Dot-Product Instructions

```
       LDW    *aptr++,ai_i+1       ; load ai and ai+1 from memory
       LDW    *bptr++,bi_i+1       ; load bi and bi+1 from memory
       MPY    ai_i+1,bi_i+1,pi     ; ai * bi
       MPYH   ai_i+1,bi_i+1,pi+1   ; ai+1 * bi+1
       ADD    pi,sum0,sum0         ; sum0 += (ai * bi)
       ADD    pi+1,sum1,sum1       ; sum1 += (ai+1 * bi+1)
       SUB    cntr,1,cntr          ; decrement loop counter
[cntr]B       LOOP                 ; branch to loop
```

4.2.3 Drawing a Dependency Graph for the Unrolled Loop

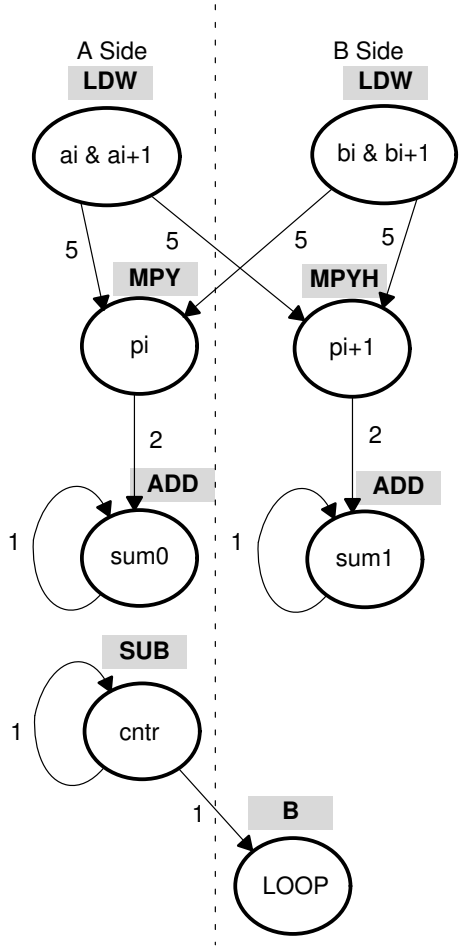
Figure 4–3 shows the dependency graph for the unrolled loop:

- The LDW instructions are parents of the MPY instructions.
- The MPY instructions are parents of the ADD instructions.

To split the graph between the A and B registers, follow these guidelines:

- Place an equal number of LDWs, MPYs, and ADDs on each side.
- To keep both sides even, place the remaining two instructions, branch and SUB, on opposite sides.

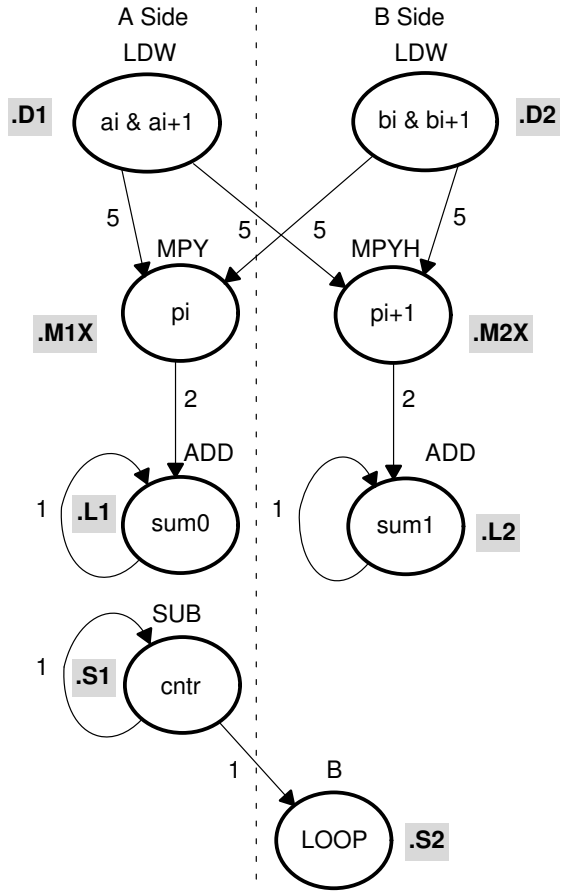
Figure 4–3. Dependency Graph of Dot Product With LDW



4.2.4 Allocating Resources

After splitting the dependency graph, you can allocate functional units and registers, as shown in the dependency graph in Figure 4–4 and the instructions in Example 4–8. The .M1X and .M2X represent a path in the dependency graph crossing from one side to the other.

Figure 4–4. Dependency Graph of Dot Product With LDW



Example 4–8. Dot Product Instructions With LDW

```
      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
```

4.2.5 Adding the Setup Code

Example 4–9 shows the assembly code for the unrolled loop, using LDW instructions instead of LDH instructions.

- 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.

Executing the dot-product code with the optimizations in Example 4–9 requires only 50 iterations, because you operate in parallel on both the even and odd array elements. With the setup code and the final ADD instruction, 100 iterations of this loop require a total of 402 cycles (1 + 8 $\times$ 50 + 1).

Example 4–9. Dot-Product Assembly With LDW

```
       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
```

4.2.6 Comparing Performance With Use of LDW

Table 4–2 compares the performance of the different versions of the dot-product code.

Table 4–2. Comparison With Use of LDW

Code Example	100 Iterations	Cycle Count
Example 4–4 Dot-Product Serial Assembly	2 + 100 × 16	1602
Example 4–5 Dot-Product Parallel Assembly	1 + 100 × 8	801
Example 4–9 Dot-Product Assembly With LDW	1 + (50× 8)+ 1	402

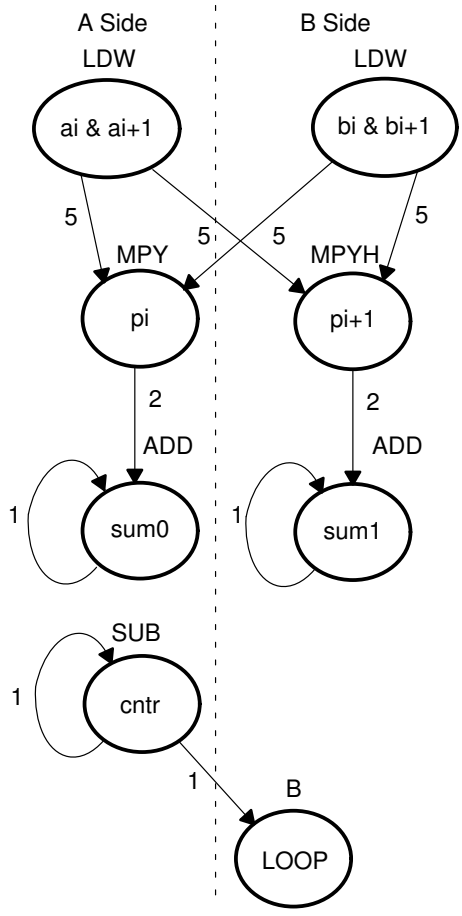
4.3 Software Pipelining

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations execute in parallel. The parallel resources on the ’C62xx make it possible to initiate a loop iteration before previous iterations finish. The goal of software pipelining is to start a new loop iteration as soon as possible

The dot-product code in Example 4–9 needed eight cycles for one iteration of the loop, five cycles for the LDWs, two cycles for the MPYs, and one cycle for the ADDs. After the first eight cycles, you then start and finish an iteration every cycle.

Figure 4–5 shows the dependency graph for the dot-product instructions. Example 4–10 shows the dot-product instructions from Example 4–8 with the SUB instruction now conditional on A1.

Figure 4–5. Dot-Product Instructions With Conditional SUB Instruction



Example 4–10. Dot Product Instructions With Functional Units

```
      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
```

4.3.1 Using the Modulo Iteration Interval Table

The *iteration interval* of a loop is the number of cycles between the initiations of successive iterations of that loop.

The *modulo iteration interval scheduling table* shows how a software-pipelined loop executes and keeps track of resources on a cycle-by-cycle basis to ensure that no resource is used twice on any given cycle.

Table 4–3 shows a modulo iteration interval table for the dot-product loop before software pipelining (Example 4–9):

- Each row represents a functional unit
- The columns indicate what is executing on a particular cycle.
- In this example, each unit is used only once every eight cycles:
  - 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 .L1 units are issued on cycles 7, 15, 23, 31, etc.
  - SUB on the .S1 units is issued on cycles 1, 9, 17, 25, etc.
  - B on the .S2 unit is issued on cycles 2, 10, 18, 24, etc.

Table 4–3. Dot-Product Modulo Iteration Interval Table

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					

4.3.2 Determining 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 4–5, 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 minimal iteration interval based on resources is 1.
- In the dot-product instructions, no data dependencies affect the minimum iteration interval.

4.3.3 Creating a Fully Pipelined Schedule

Having determined the modulo iteration interval is 1, you can initiate a new iteration every cycle. You can schedule LDW and MPY instructions on every cycle. Table 4–4 shows a fully pipelined schedule for the dot-product example:

- The right-most column is a single-cycle loop that contains the entire loop.
- Cycles 0–6 are loop setup code, or loop prologue.

Asterisks define which iteration of the loop the instruction is executing each cycle. For example, the right-most 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 branch 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* and *n + 7* executing in parallel. 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 less.

Table 4–4. Dot-Product Modulo Iteration Interval Table

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

4.3.4 Software Pipelined Dot Product

The following describes Example 4–11, which shows the assembly code for Table 4–4.

- The accumulators are initialized to 0 and the loop counter is set up in the first execute packet in parallel with the first LDW instructions.
- Asterisks in the comments indicate the iterations that execute like those in Table 4–4.
- The branch target is the execute packet defined by the label LOOP and multiple branch instructions are in the pipe.
  - The first branch is issued on cycle 2, but does not actually branch until the end of cycle 7 after five delay slots. The second branch is issued on cycle 3, but does not branch until cycle 8.
  - 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.

If the SUB instruction were not conditional on A1, you would have an infinite loop:

- As the loop executes five more times, the loop counter, if not conditional, would become a negative number on the next decrement.
- When A1 is negative, it is nonzero and, therefore, causes the condition on the branch to be true again.
- Making the SUB instruction conditional on A1 ensures that A1 stops decrementing when it reaches 0.

Executing the dot-product code with the software pipelining as shown in Example 4–11 requires a total of 58 cycles (7 + 50 + 1), which is a significant improvement over the 402 cycles required by the code in Example 4–9.

Example 4–11. Software Pipelined Dot Product

```
        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
```

4.3.5 Removing Extraneous Instructions

The code in Example 4–11 executes multiple iterations with the following events occurring in parallel:

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

In most cases, extra iterations are not a problem, except that when extraneous LDWs access unmapped memory, you can get unpredictable results. To remove the extraneous LDW and MPY instructions, you can add an epilogue that is included in the second part of Example 4–12 on page 4-23.

- To eliminate LDWs from the iterations 51 through 57, run the loop seven fewer times.
- The loop counter is 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.

Executing the dot-product code in Example 4–12 with no extraneous LDWs still requires a total of 58 cycles (7 + 43 + 7 + 1).

Example 4–12. Software Pipelined Dot Product 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

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
```

Example 4–12. Software Pipelined Dot Product With No Extraneous Loads (Continued)

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

       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
```

4.3.6 Priming the Loop

Although Example 4–12 executes as fast as possible, the code size could be smaller. To help reduce code size, you can use a technique called *priming the loop*. Assuming that you can handle extraneous LDWs, start with Example 4–11, which has no epilogue and, therefore, fewer instructions. (This technique could be used equally well on Example 4–12.)

To eliminate the prologue and, therefore, the extra LDW and MPY instructions, begin execution at the loop body (at the LOOP label). Eliminating the prologue has several implications:

- Two LDWs, two MPYs, and two ADDs occur in the first execution cycle of the loop.
- Since 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 4–13 shows the loop without the prologue but with four new instructions that zero out the inputs to the MPY and ADD instructions.

- Making the MPYs and ADDs use 0s before valid data is available ensures that the final accumulate values are unaffected.
- The loop counter is initialized to 57 to accommodate the seven extra cycles needed to prime the loop.

Executing the dot-product code in Example 4–13 is less efficient than the code in Example 4–11. Because the first LDWs are not issued until after seven cycles, the code in Example 4–13 requires a total of 65 cycles (7 + 57+ 1).

Example 4–13. Software Pipelined Dot Product — No Prologue or Epilogue

```
                                                                    ADDs
       MVK    .S1    57,A1         ; set up loop counter             (1)

  [A1] ADD    .S1    -1,A1,A1      ; decrement loop counter          (2)
||     ZERO   .L1    A7            ; zero out sum0 accumulator
||     ZERO   .L2    B7            ; zero out sum1 accumulator

  [A1] ADD    .S1    -1,A1,A1      ;* decrement loop counter         (3)
||[A1] B      .S2    LOOP          ; branch to loop
||     ZERO   .L1    A6            ; zero out add input
||     ZERO   .L2    B6            ; zero out add input

  [A1] ADD    .S1    -1,A1,A1      ;** decrement loop counter        (4)
||[A1] B      .S2    LOOP          ;* branch to loop
||     ZERO   .L1    A2            ; zero out mpy input
||     ZERO   .L2    B2            ; zero out mpy input

  [A1] ADD    .S1    -1,A1,A1      ;*** decrement loop counter       (5)
||[A1] B      .S2    LOOP          ;** branch to loop

  [A1] ADD    .S1    -1,A1,A1      ;**** decrement loop counter      (6)
||[A1] B      .S2    LOOP          ;*** branch to loop

  [A1] ADD    .S1    -1,A1,A1      ;***** decrement loop counter     (7)
||[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] ADD    .S1    -1,A1,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
```

4.3.7 Removing Extra SUB Instructions

Because you know that the loop count is at least 6, you can eliminate the extra SUB instructions as shown in Example 4–14.

- The first five branch instructions are made unconditional, since 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 4–13.)
- Based on the elimination of six SUB instructions, the loop counter is now 51 (57 – 6).

This code shows some improvement over Example 4–13. The loop in Example 4–14 requires 63 cycles (5 + 57 + 1).

Example 4–14. Software Pipelined Dot Product 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] ADD    .S1    -1,A1,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
```

4.3.8 Comparing Performance

Table 4–5 compares the performance of all versions of the dot-product code.

Table 4–5. Comparison of Dot-Product Code Examples

Code Example		100 Iterations	Cycle Count
Example 4–4	Serial Assembly	2 + 100 × 16	1602
Example 4–5	Parallel Assembly	1 + 100 × 8	801
Example 4–9	Dot-Product Assembly With LDW	1 + (50 × 8) + 1	402
Example 4–11	Software Pipelined Dot Product	7 + 50 + 1	58
Example 4–12	Software Pipelined Dot Product With No Extraneous Loads	7 + 43 + 7 + 1	58
Example 4–13	Software Pipelined Dot Product — No Prologue or Epilogue	7 + 57+ 1	65
Example 4–14	Software Pipelined Dot Product With Smallest Code Size	5 + 57 + 1	63

4.4 Modulo Scheduling of Multicycle Loops

Section 4.3 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. When you have multicycle loops, resource conflicts can affect modulo scheduling.

4.4.1 Weighted Vector Sum C Code

Example 4–15 shows the C code for a weighted vector sum.

Example 4–15. 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];
       }
}
```

4.4.2 Translating the Inner Loop to ’C62xx Instructions

Example 4–16 shows the symbolic instructions for the weighted vector sum instructions that execute the inner loop ($m \times ai$) in Example 4–15.

Example 4–16. List of Symbolic Weighted Vector Sum Instructions

```
       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
```

4.4.3 Determining the Minimum Iteration Interval

Example 4–16 includes three memory operations in the inner loop (two LDHs and the STH) that must each use a .D unit. Because only two .D units are available on any single cycle, this loop requires at least two cycles. Since no other resources are used more than twice, the minimum iteration interval for this loop is 2. Because memory operations are determining the minimum iteration interval, unrolling the loop and performing LDWs can help improve the performance.

4.4.4 Unrolling the Weighted Vector Sum C Code

Example 4–17 shows the C code for an unrolled version of the weighted vector sum.

Example 4–17. Weighted Vector Sum C Code

```
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];
      }
}
```

4.4.5 Translating Unrolled Inner Loop to 'C62xx Instructions

Example 4–18 shows a the weighted vector sum instructions that calculate *c[i]* and *c[i+1]* in Example 4–17.

- 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 4–18. List of Symbolic Weighted Vector Sum Instructions 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
```

4.4.6 Determining a New Minimum Iteration Interval

Use the following considerations to determine the minimum iteration interval for the assembly instructions in Example 4–18.

- 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.
- Since the remaining four instructions (two ADDs, AND, and SUB) can all go on a .L unit, the minimum iteration interval for this loop is the same as in Example 4–16.

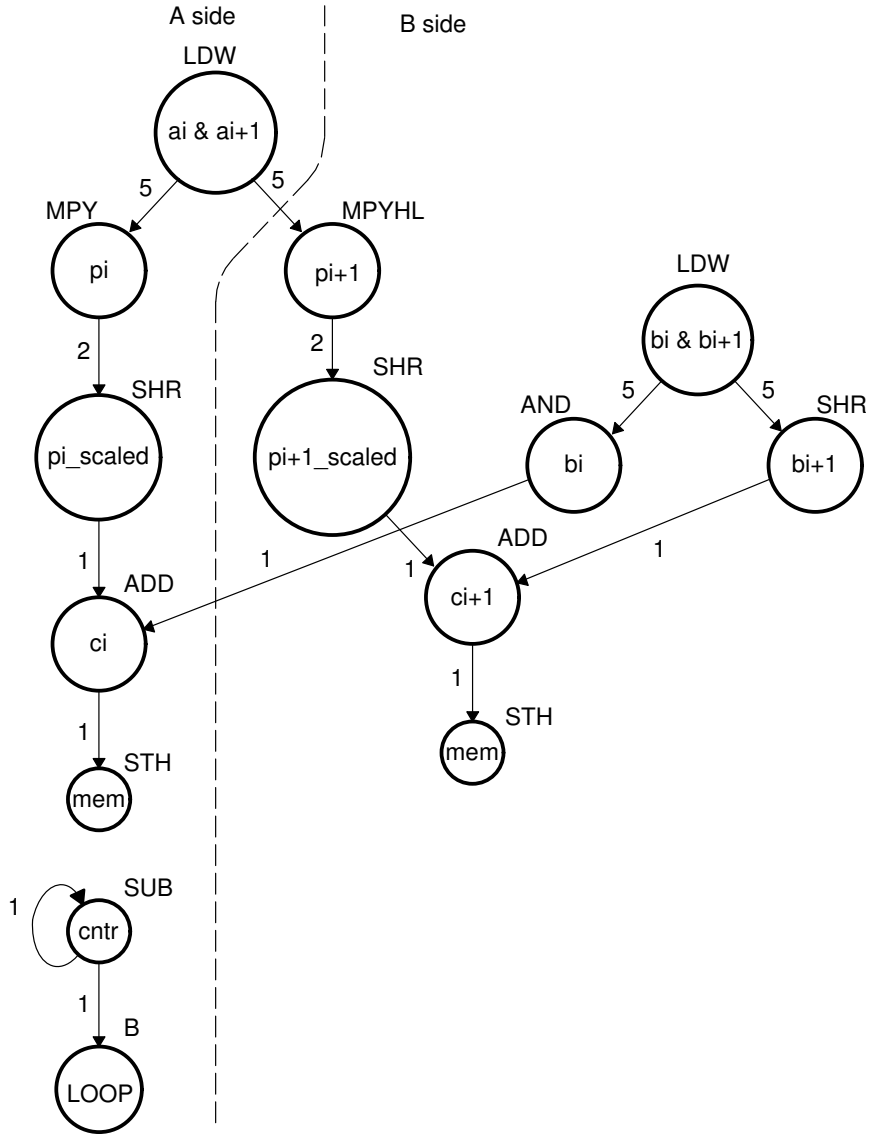
By using LDWs instead of LDHs, the program can do twice as much work in the same number of cycles.

4.4.7 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 4–6 shows the dependency graph divided evenly with a minimum interation interval of 2.

Figure 4–6. Dependency Graph of Weighted Vector Sum



4.4.8 Allocating Resources

Using the dependency graph, you can allocate functional units and registers as shown in Example 4–19. 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 4–19. List of Actual Weighted Vector Sum Instructions

```
       LDW    .D1    *A4++,A2      ; ai & ai+1
       LDW    .D2    *B4++,B2      ; bi & bi+1
       MPY    .M1X   A2,B6,A5      ; m * ai
       MPYHL  .M2X   A2,B6,B5      ; m * ai+1
       SHR    .S1    A5,15,A7      ; (m * ai) >> 15
       SHR    .S2    B5,15,B7      ; (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
```

4.4.9 Modulo Iteration Interval Scheduling

Table 4–6 provides a method to keep track of resources which 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 4–6 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 4–6 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 LDW uses the .D1 unit 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 the 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 4–6. Weighted Vector Sum Modulo Iteration Interval Table (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.

4.4.10 Resource Conflicts

Resources from one instruction cannot conflict with resources from any other instruction scheduled modulo iteration intervals away. In other words, for a two-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 4–7 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_s*, which is on .S2 in cycle 7.

Table 4–7. Modulo Iteration Interval Table 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.

4.4.11 Live Too Long

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, if in a two-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.

The ADD *ci* in Figure 4–6 demonstrates a *live-too-long problem*. The parents of ADD *ci* (AND *bi* and SHR *pi_s*) are scheduled on cycles 5 and 7, respectively.

- Since the SHR *pi_s* is scheduled on cycle 7, the earliest you can schedule ADD *ci* is cycle 8.
- In cycle 7, AND *bi** writes *bi* for the next iteration of the loop, which means if you schedule ADD *ci* on cycle 8, it reads the parent value of *bi* for the next iteration, which is incorrect. This situation illustrates the live-too-long problem.

4.4.12 Solving the Live-Too-Long Problem

The live-too-long problem in Table 4–7 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, Figure 4–7 and Table 4–8 show that AND *bi* has been moved to cycle 6 so that you can schedule ADD *ci* to read the correct value on cycle 8.

Figure 4–7. Dependency Graph of Weighted Vector Sum

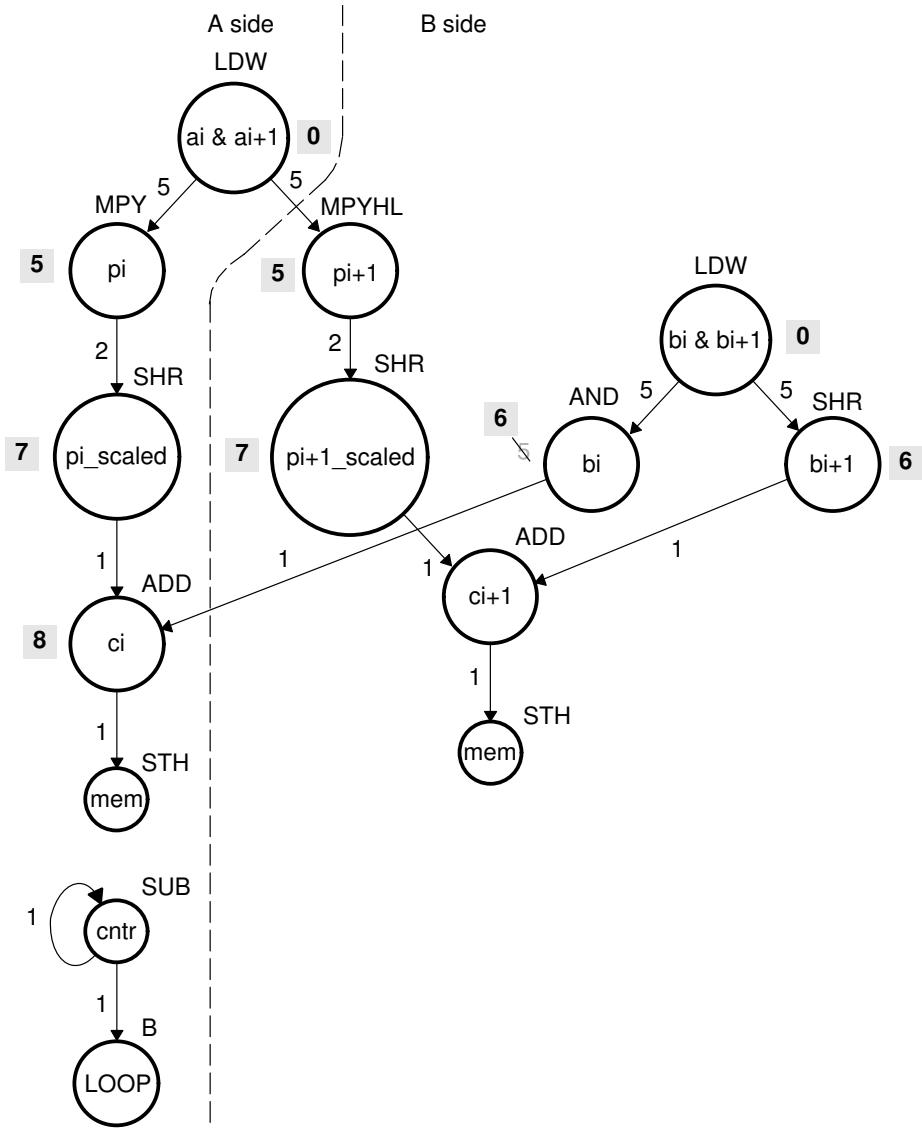


Table 4–8. Weighted Vector Sum Modulo Iteration Interval Table (2-Cycle loop)

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 current iteration of the loop iteration 0 through iteration 5; shading indicates changes in scheduling from Table 4–7.

4.4.13 Scheduling the Remaining Instructions

Figure 4–8 shows the dependency graph with additional changes. The final version of the loop, with all instructions scheduled correctly, is shown in Table 4–9.

- Table 4–9 shows the following additions:
  - B LOOP (.S1, cycle 6)
  - SUB (.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 4–9 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_s* (.S2) moved from cycle 7 to cycle 9.
  - MPYHL 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, since iterations n and n + 5 execute in parallel.

Figure 4–8. Dependency Graph for Scheduling ci+1 (Weighted Vector Sum)

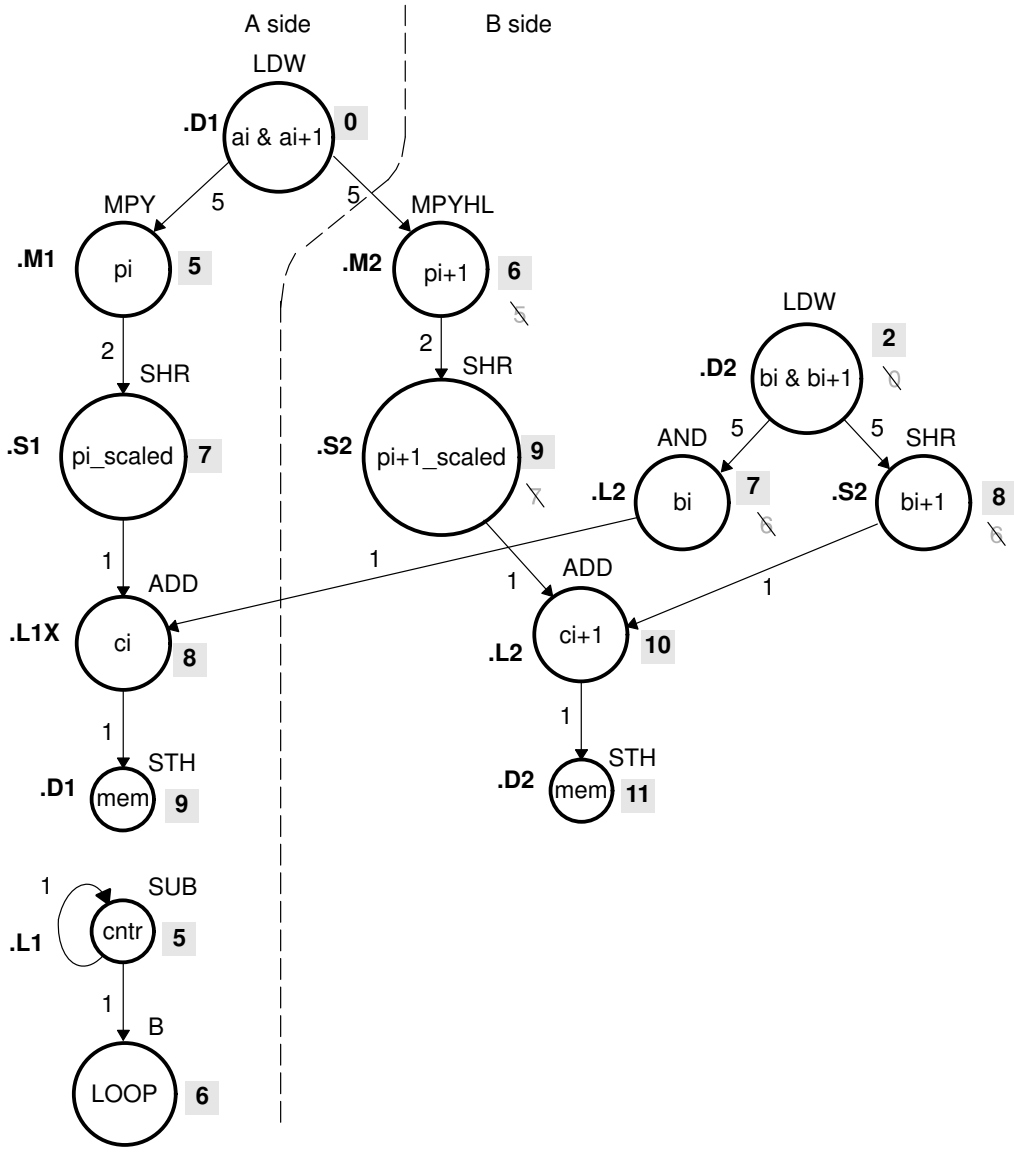


Table 4–9. Weighted Vector Sum Modulo Iteration Interval Table (2-Cycle loop)

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 current iteration of the loop iteration 0 through iteration 5; shaded cells indicate cycle 0.

4.4.14 Final Assembly

Example 4–20 lists the final assembly for the weighted vector sum.

- 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, schedule the final executions of ADD *ci+1* and STH *ci+1* after exiting the loop and execute the loop only 49 times.
- The mask for the AND instruction is created with MVK and MVKH in parallel with the loop prologue.
- The pointer to the odd elements in array *c* is also set up in parallel with the loop prologue.

Example 4–20. 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 4–20. 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
```

4.5 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, where output samples are used as input to the computation of the next output sample

4.5.1 IIR Filter C Code

Example 4–21 shows C code for a simple IIR filter.

- *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 4–21. 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;
       }
}
```

4.5.2 Symbolic 'C62xx Instructions (Inner Loop)

Example 4–22 shows symbolic 'C62xx instructions that execute the inner loop.

- *xptr* is not post incremented after loading *xi+1*, since *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 post incremented after storing *yi+1*, since *yi* of the next iteration is *yi+1* for the current iteration.

Example 4–22. List of Symbolic IIR Instructions

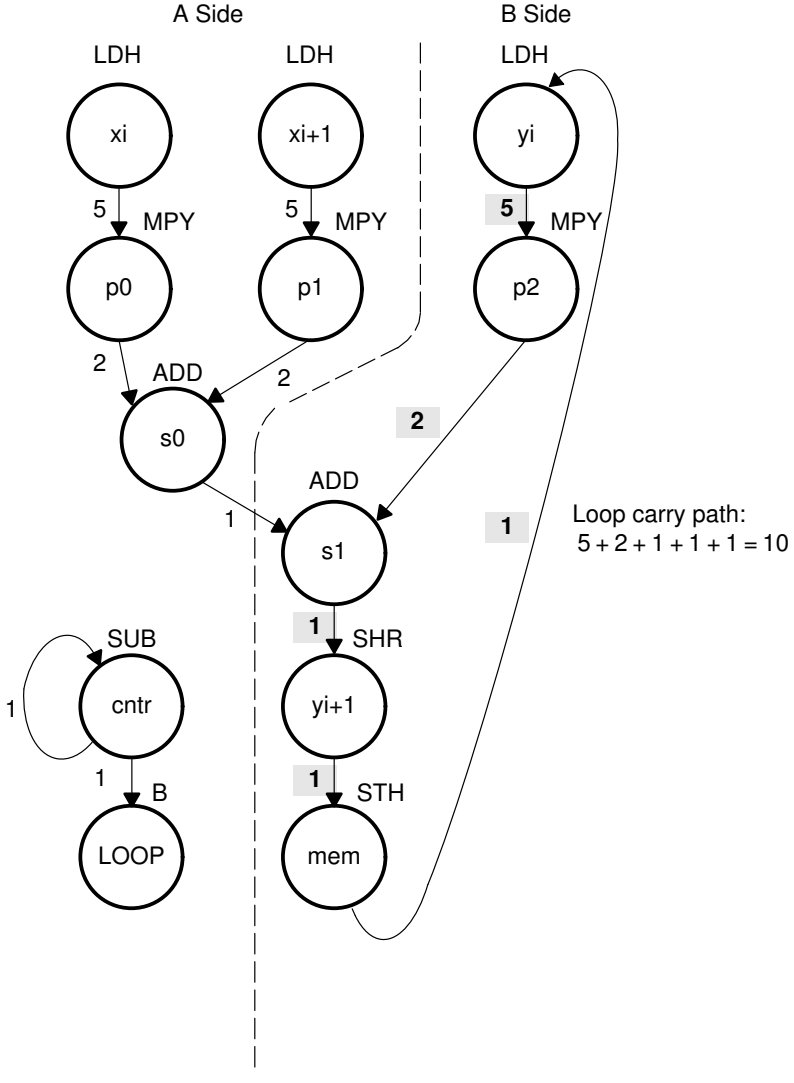
```
       LDH    *xptr++,xi    ; xi+1
       MPY    c1,xi,p0      ; c1 * xi
       LDH    *xptr,xi+1    ; xi+1
       MPY    c2,xi+1,p1    ; c2 * xi+1
       ADD    p0,p1,s0      ; c1 * xi + c2 * xi+1
       LDH    *yptr++,yi    ; yi
       MPY    c3,yi,p2      ; c3 * yi
       ADD    s0,p2,s1      ; 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
```

4.5.3 Dependency Graph

Figure 4–9 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 4–9. Dependency Graph Of IIR Filter



4.5.4 Minimum Iteration Interval

To determine the minimum iteration interval, you must consider both resources and data dependency constraints.

- Based on resources in Table 4–10, the minimum iteration interval is 2 because the total non-.M units on the A side is 5 and no other units are used more than twice.
- The IIR has a data dependency constraint defined by its loop carry path. Figure 4–9 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 must be on 8 and STH on on 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 4–10 shows how to improve the performance.

Table 4–10. 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

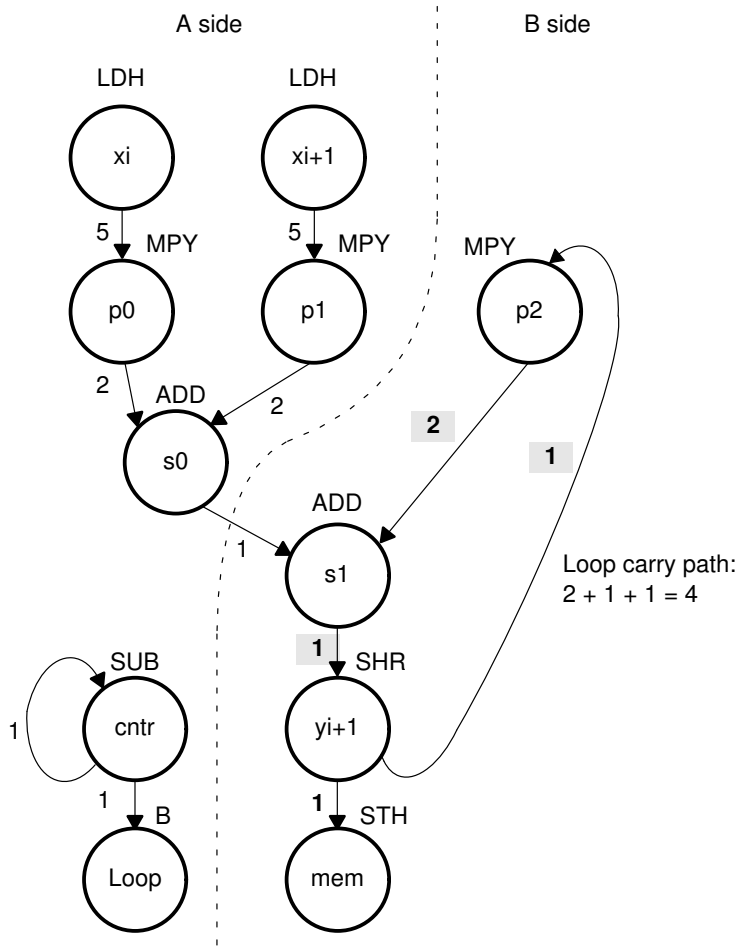
Unit(s)	Instructions	Total/Unit
.M2	MPY	1
.S2	SHR	1
.D2	STH	1
.L2 or .S2, .D2	ADD	1
Total non-.M units		3

4.5.5 New Dependency Graph

Figure 4–10 shows a new graph with a loop carry of 4 (2 +1 + 1).

- Since 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.
- With the LDH of *yi* no longer in the graph, you can issue the LDH of *y[0]* once outside the loop. Every iteration after that, the *y+1* values written by the SHR instruction are valid *y* inputs to the MPY instruction.

Figure 4–10. Dependency Graph of IIR Filter With Smaller Loop Carry



4.5.6 New Symbolic 'C62xx Instructions (Inner Loop)

In Example 4–23, you no longer have LDH *yi*. 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 4–23. List of Symbolic IIR Instructions 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
```

4.5.7 Allocating Resources

Example 4–24 lists the 'C62xx instructions with the functional units and registers that are used in the inner loop.

Example 4–24. List of Actual IIR Instructions

```
       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
```

4.5.8 Modulo Iteration Interval Scheduling

Table 4–11 shows the modulo iteration interval table for the IIR. The SHR on cycle 10 finishes in time for the MPY *p2* instruction from the next iteration to read its result on cycle 11.

Table 4–11. IIR Modulo Iteration Interval Table (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 current iteration of the loop iteration 0 through iteration 2.

4.5.9 Final Assembly

Example 4–25 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 4–25 requires 408 cycles: (4×100) + 8.

Example 4–25. 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
```

4.6 If-Then-Else Statements in a Loop

If-then-else statements in C cause certain instructions to execute when the *if* condition is true or other instructions to execute when it is false. One way to accomplish this on the 'C62xx is with conditional instructions. Since all 'C62xx instructions can be conditional on one of five general-purpose registers, conditional instructions can handle both the true and false cases of the if-then-else C statement.

4.6.1 If-Then-Else C Code

Example 4–26 contains a loop with an if-then-else statement. You either add *a[i]* to *sum* or subtract *a[i]* from *sum*.

Example 4–26. 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);
}
```

4.6.2 Branching vs. Conditional Instructions

Branching is one way to execute the if-then-else statement:

- Branch to the ADD when the *if* statement is true
- Branch to the SUB when the *if* statement is false

  Because each branch has five delay slots, this method requires additional cycles, and branching within the loop makes software pipelining almost impossible.

Conditionals avoid having to branch to the appropriate piece of code after checking whether the condition is true or false.

- 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.

4.6.3 ’C62xx Instructions (Inner Loop)

Example 4–27 lists the ’C62xx instructions needed to execute the C code in Example 4–26.

- If the result of the bitwise AND is nonzero, a 1 is written into *cond*.
- A conditional MVK performs the *!(!(cond))* C statement.
  - If the result of the AND is 0, *cond* remains at 0.
  - If the result of the AND is nonzero, *cond* is changed to 1.
- CMPEQ is used to create *if*.
- The ADD is conditional when *if* is nonzero (corresponds to *then*).
- The SUB is conditional on when *if* is 0 (corresponds to *else*).

Example 4–27. List of Symbolic If-Then-Else Instructions

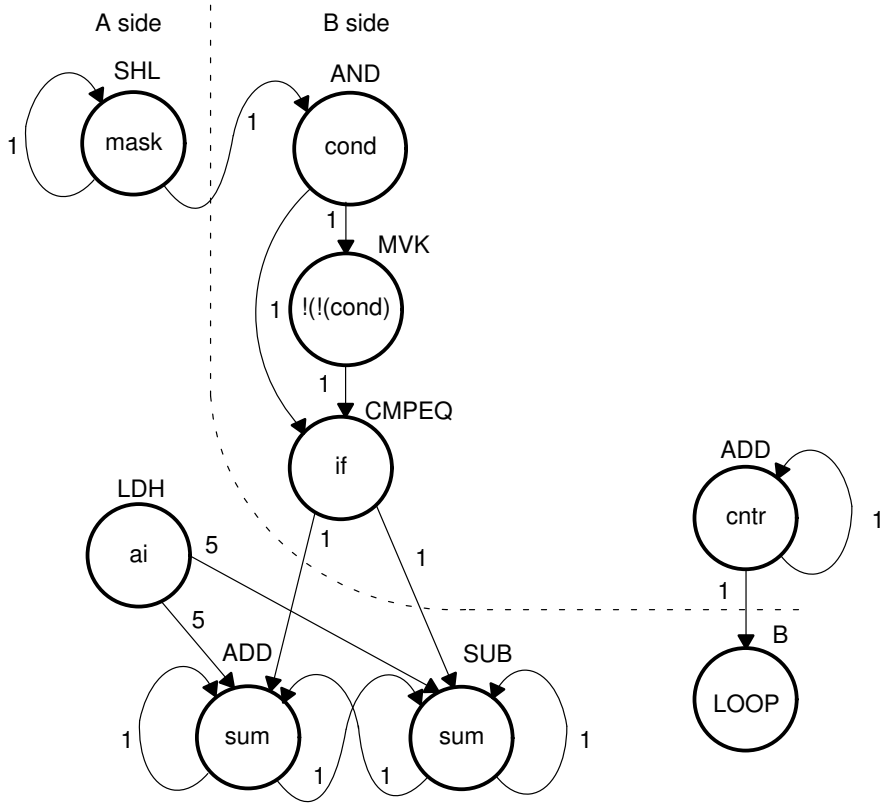
```
      AND    codeword,mask,cond  ; cond = codeword & mask
[cond]MVK    1,cond              ; !(!(cond))
      CMPEQ  theta,cond,if       ; (theta == !(!(cond)))
      LDH    *aptr++,ai          ; a[i]
[if]  ADD    sum,ai,sum          ; sum += a[i]
[!if] SUB    sum,ai,sum          ; sum -= a[i]
      SHL    mask,1,mask         ; mask = mask << 1;

[cntr]ADD    -1,cntr,cntr        ; decrement counter
[cntr]B      LOOP                ; for LOOP
```

4.6.4 Dependency Graph

Figure 4–11 shows the dependency graph for the if-then-else C code.

- Two nodes on the graph contain *sum*: one for the ADD and one for the SUB. Because some interations 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*, since both instructions read *ai*.
- CMPEQ *if* is also a parent to ADD *sum* and SUB *sum*, since both read *if* for the conditional execution.
- The result of SHL *mask* is read on the next iteration by the AND *cond* instruction.

Figure 4–11. Dependency Graph of If-Then-Else Code



4.6.5 Minimum Iteration Interval

With nine instructions, the minimum iteration interval is at least 2, since a maximum of eight instructions can be in parallel. Based on the way the dependency graph in Figure 4–11 is split, five instructions are on the A side and four are on the B side. Because none of the instructions are MPYs, all instructions must go on the .S, .D, or .L units, which means you have a total of six resources.

- LDH must be on a .D unit.
- SHL, B, and MVK must be on a .S unit.
- The ADDs and SUB can be on the .S, .L, or .D units.
- The AND can be on a .S or .L unit.

From Table 4–12, you can see that no one resource is used more than two times so that the minimum iteration interval is still 2.

The minimum iteration interval is also affected by the total number of instructions. Since 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. Since only four instructions are on the B side, the minimum iteration interval is still 2.

Table 4–12. 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

4.6.6 Allocating Resources

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 4–28 shows the ’C62xx instructions with the functional units.

Example 4–28. List of Actual If-Then-Else Instructions

```
      AND    .S2X   B4,A6,B2      ; cond = codeword & mask
 [B2] MVK    .S2    1,B2          ; !(!(cond))
      CMPEQ  .L2    B6,B2,B1      ; (theta == !(!(cond)))
      LDH    .D1    *A4++,A5      ; a[i]
[B1]  ADD    .L1    A7,A5,A7      ; sum += a[i]
[!B1] SUB    .D1    A7,A5,A7      ; sum -= a[i]
      SHL    .S1    A6,1,A6       ; mask = mask << 1;

 [B0] ADD    .L2    -1,B0,B0      ; decrement counter
 [B0] B      .S1    LOOP          ; for LOOP
```

4.6.7 Final Assembly With Software Pipelining

Example 4–29 shows the final assembly code after software pipelining. The performance of this loop is 70 cycles (2 × 32 + 6).

Example 4–29. If-Then-Else Assembly

```
       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
```

4.6.8 Performance Improvements

You can improve the performance of the code in Example 4–29 if you know that the loop count is at least 3:

- ❑ Remove the decrement counter instructions outside the loop.
- ❑ Put the MVK (for setting up the loop counter) in parallel with the first branch.
- ❑ The first two branches are now unconditional, since the loop count is at least 3 and you know that the first two branches must execute.

These two changes save two cycles at the beginning of the loop prologue. To account for the removal of the three decrement-loop-counter instructions, set the loop counter to 3 less than the actual number of times you want the loop to execute: in this case, 29 (32 – 3).

Example 4–30 shows the improved loop with a cycle count of 68 ($2 \times 32 + 4$). Table 4–13 compares the performance of Example 4–29 and Example 4–30.

Table 4–13. Comparison of If-Then-Else Code Examples

Code Example		Cycles	Cycle Count
Example 4–29	If-Then-Else Assembly	$(2 \times 32) + 6$	70
Example 4–30	If-Then-Else Assembly With Loop Count Greater Than 3	$(2 \times 32) + 4$	68

Example 4–30. If-Then-Else Assembly 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
```

4.7 Loop Unrolling

When resources are not fully utilized, you can help improve performance by unrolling the loop. In Example 4–31, 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-percent improvement in throughput: three cycles to do two iterations, rather than the four cycles required in Example 4–30.

4.7.1 Unrolled If-Then-Else C Code

Example 4–31 shows the unrolled version of Example 4–30.

Example 4–31. Unrolled If-Then-Else C Code

```
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);
}
```

4.7.2 ’C62xx Instructions (Inner Loop)

Example 4–32 shows the unrolled loop with 16 instructions and the possibility of achieving a loop with a minimum iteration interval of 3.

Example 4–32. List of Symbolic Unrolled If-Then-Else Instructions

```
        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
```

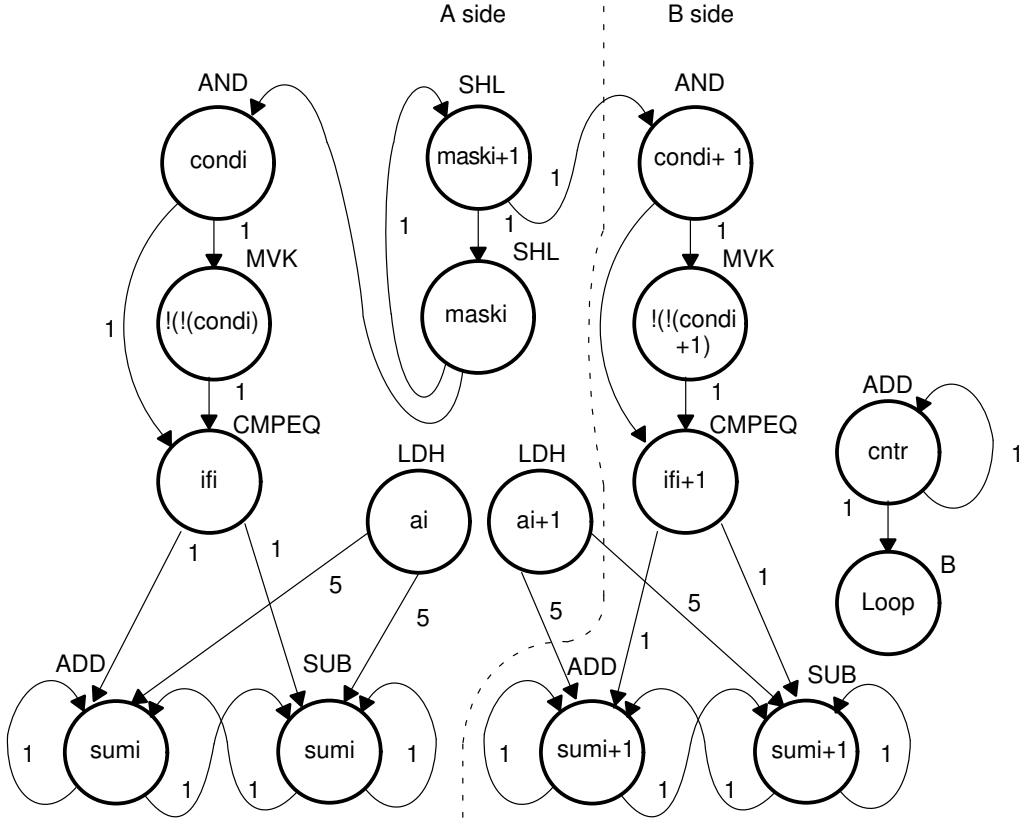
4.7.3 Dependency Graph

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 4–12 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 4–12. Dependency Graph of Unrolled If-Then-Else Code



4.7.4 Minimum Iteration Interval

With 16 instructions, the minimum iteration interval is at least 3 because a maximum of six instructions can be in parallel with the following allocation possibilities:

- LDH must be on a .D unit.
- SHL, B, and MVK must be on a .S unit.
- The ADDs and SUB can be on a .S, .L, or .D unit.
- The AND can be on a .S or .L unit.

From Table 4–14, 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. Since only seven non-.M instructions are on the B side, the minimum iteration interval is still 3.

Table 4–14. Resource Table for Unrolled If-Then-Else Code

(a) A side *(b) B side*

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

4.7.5 Allocating Resources

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 4–33 shows the ’C62xx instructions with the functional units.

Example 4–33. Unrolled If-Then-Else Instructions

```
      AND    .L1X   B4,A6,A2      ; condi = codeword & maski
[A2]  MVK    .S1    1,A2          ; !(!(condi))
      CMPEQ  .L1X   B6,A2,A1      ; (theta == !(!(condi)))
      LDH    .D1    *A4++,A5      ; a[i]
[A1]  ADD    .L1    A7,A5,A7      ; sum += a[i]
[!A1] SUB    .D1    A7,A5,A7      ; sum -= a[i]
      SHL    .S1    A6,1,A6       ; maski+1 = maski << 1;

      AND    .L2X   B4,A6,B2      ; condi+1 = codeword & maski+1
 [B2] MVK    .S2    1,B2          ; !(!(condi+1))
      CMPEQ  .L2    B6,B2,B1      ; (theta == !(!(condi+1)))
      LDH    .D1    *A4++,B5      ; a[i+1]
 [B1] ADD    .L2    B7,B5,B7      ; sum += a[i+1]
 [!B1]SUB    .D2    B7,B5,B7      ; sum -= a[i+1]
      SHL    .S1    A6,1,A6       ; maski = maski+1 << 1;

 [B0] ADD    .D2    -1,B0,B0      ; decrement counter
 [B0] B      .S2    LOOP          ; for LOOP
```

4.7.6 Final Assembly

Example 4–34 shows the final assembly code after software pipelining. The cycle count of this loop is now 53: (3×16) + 5.

Code Example		Cycles	Cycle Count
Example 4–29	If-Then-Else Assembly	(2 × 32) + 6	70
Example 4–30	If-Then-Else Assembly With Loop Count Greater Than 3	(2 × 32) + 4	68
Example 4–33	Unrolled If-Then-Else Instructions	(3 × 16) + 5	53

Example 4–34. Unrolled If-Then-Else Assembly

```
       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
```

4.8 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 4.4.10, *Resource Conflicts* on page 4-35, solved this problem simply by moving the parent to a later cycle.

4.8.1 C Code With Live-Too-Long Problem

Example 4–35 shows C code with a live-too-long problem that cannot be solved by rescheduling the parent instruction. A *split-join* path in the dependency graph in Figure 4–13 causes this live-too-long problem.

Example 4–35. 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);
}
```

4.8.2 ’C62xx Instructions (Inner Loop)

Example 4–36 shows the symbolic instructions that execute the loop in Example 4–35.

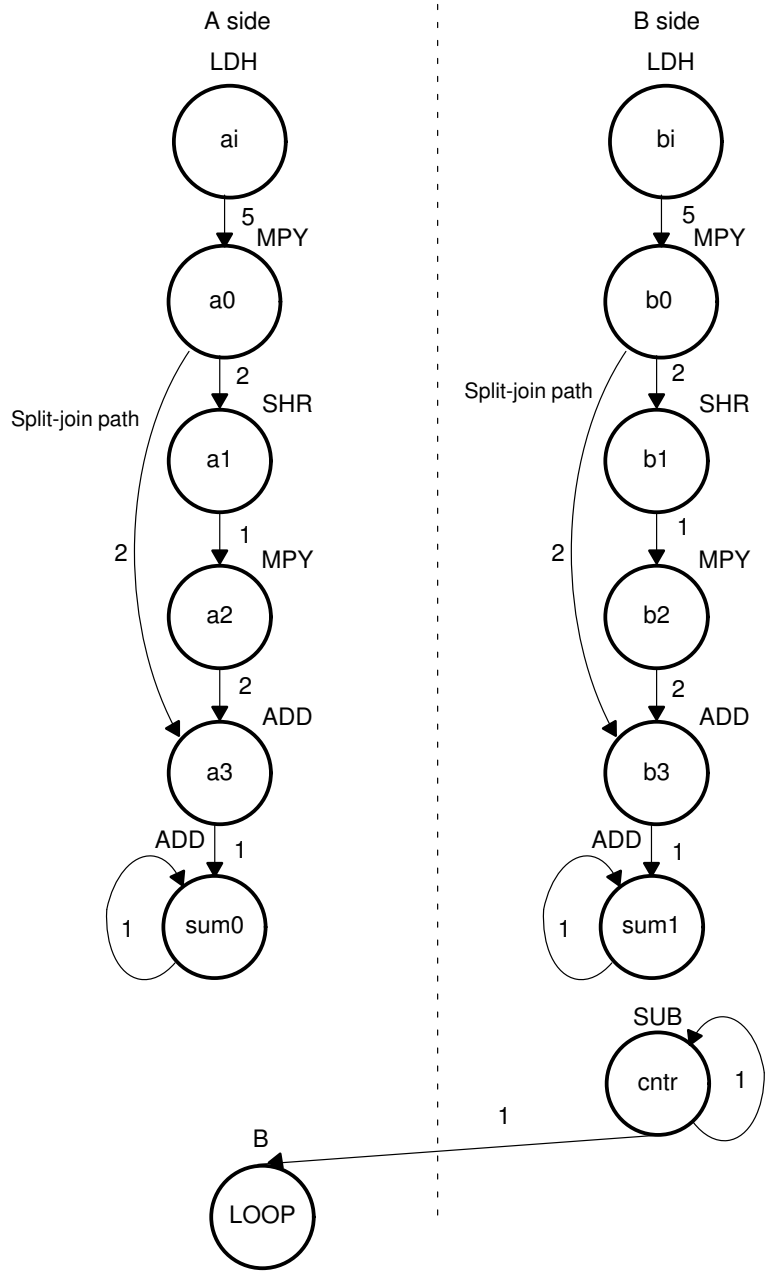
Example 4–36. List of Symbolic Live-Too-Long Instructions

```
      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
```

4.8.3 Dependency Graph

Figure 4–13 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 4–13. Live-Too-Long Code



4.8.4 Minimum Iteration Interval

Table 4–15 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 resources are 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, since three non-.M units can be used on a side during each cycle.

Table 4–15. Resource Table For Live–Too-Long Code

(a) A side *(b) B side*

Unit(s)	Instructions	Total/Unit	Unit(s)	Instructions	Total/Unit
.M1	MPY	1	.M2	MPY	1
.S1	B and SHR	2	.S2	SHR	1
.D1	LDH	1	.D2	LDH	1
.L1, .S1, or .D1	2 ADDs	2	.L2, .S2, or .D2	2 ADDs and SUB	3
Total non-.M units		5	Total non-.M units		5

4.8.5 Split-Join-Path Problems

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 4.5, *Loop Carry Paths*, on page 4-46. In this example, a *live-too-long* problem determines the minimum iteration interval.

In Figure 4–13, 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. Therefore, if the value must 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 resources.

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.

4.8.6 Inserting Moves

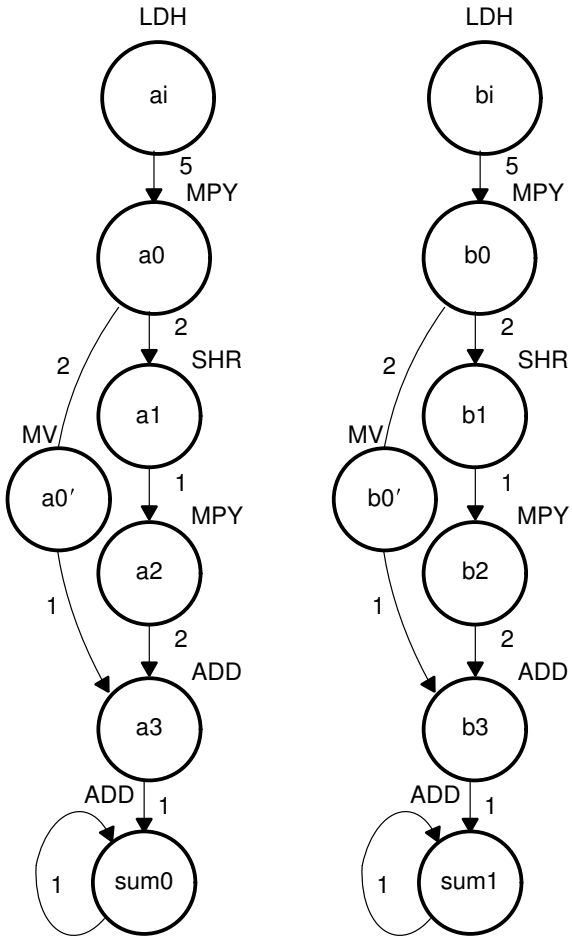
Another solution to the *live-too-long* problem is to break up the lifetime of *a0* and *b0* by inserting move instructions (MVs). The MV instruction breaks up the left path of the split-join path into two smaller pieces.

4.8.7 New Dependency Graph

Figure 4–14 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 as follows:

- If MV *a0'* is scheduled on cycle 8, then *a0* is live on cycles 7 and 8, and *a0'* is live on cycles 9 and 10.
- Since no values are live more than two cycles, the minimum iteration interval for this graph is 2.

Figure 4–14. New Dependency Graph of Live-Too-Long Code



4.8.8 Allocating Resources

Example 4–37 shows the ’C62xx instructions with the functional units assigned. The choice of units for the ADDs and SUB is flexible and represents one of a number of possibilites. 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 4–37. List of Actual Live-Too-Long Code Instructions

```
       LDH    .D1    *A4++,A0      ; load ai from memory
       LDH    .D2    *B4++,B0      ; load bi from memory
       MPY    .M1    A0,A6,A3      ; a0 = ai * c
       SHR    .S1    A3,15,A5      ; a1 = a0 >> 15
       MPY    .M1X   A5,B6,A7      ; a2 = a1 * d
       MV     .D1    A3,A2         ; save a0 across iterations
       ADD    .L1    A7,A2,A9      ; a3 = a2 + a0
       ADD    .L1    A1,A9,A1      ; sum0 += a3
       MPY    .M2X   B0,A6,B10     ; b0 = bi * c
       SHR    .S2    B10,15,B5     ; b1 = b0 >> 15
       MPY    .M2X   B5,A8,B7      ; b2 = b1 * e
       MV     .D2    B10,B8        ; save b0 across iterations
       ADD    .L2    B7,B8,B9      ; b3 = b2 + b0
       ADD    .L2    B1,B9,B1      ; sum1 += b3

 [B2]  SUB    .S2    B2,1,B2       ; decrement loop counter
 [B2]  B      .S1    LOOP          ; branch to loop
```

4.8.9 Final Assembly With Move Instructions

Example 4–38 shows the final assembly code after software pipelining. The performance of this loop is 212 cycles (2 $\times$100 + 11 + 1).

Example 4–38. Final Assembly 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 4–38. Final Assembly 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
```

4.9 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.

4.9.1 FIR C Code

Example 4–39 shows C code for a simple FIR filter. There are two memory reads (*x[i+j]* and *h[i]*) for each multiply. Because the 'C62xx can perform only two LDHs per cycle, it seems, at first glance, that only one multiply-accumulate per cycle is possible.

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 'C62xx 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 would have to load elements that are not word-aligned: 1 and 2, 3 and 4, etc.

Example 4–39. 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;
          }
}
```

4.9.2 Redundant Loads

In order to achieve two multiply-accumulates per cycle, you need to 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.

4.9.3 New FIR C Code

Example 4–40 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 4–40. 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;
          }
}
```

4.9.4 Symbolic 'C62xx Instructions (Inner Loop)

Example 4–41 shows the 'C62xx instructions that perform the innermost loop.

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.

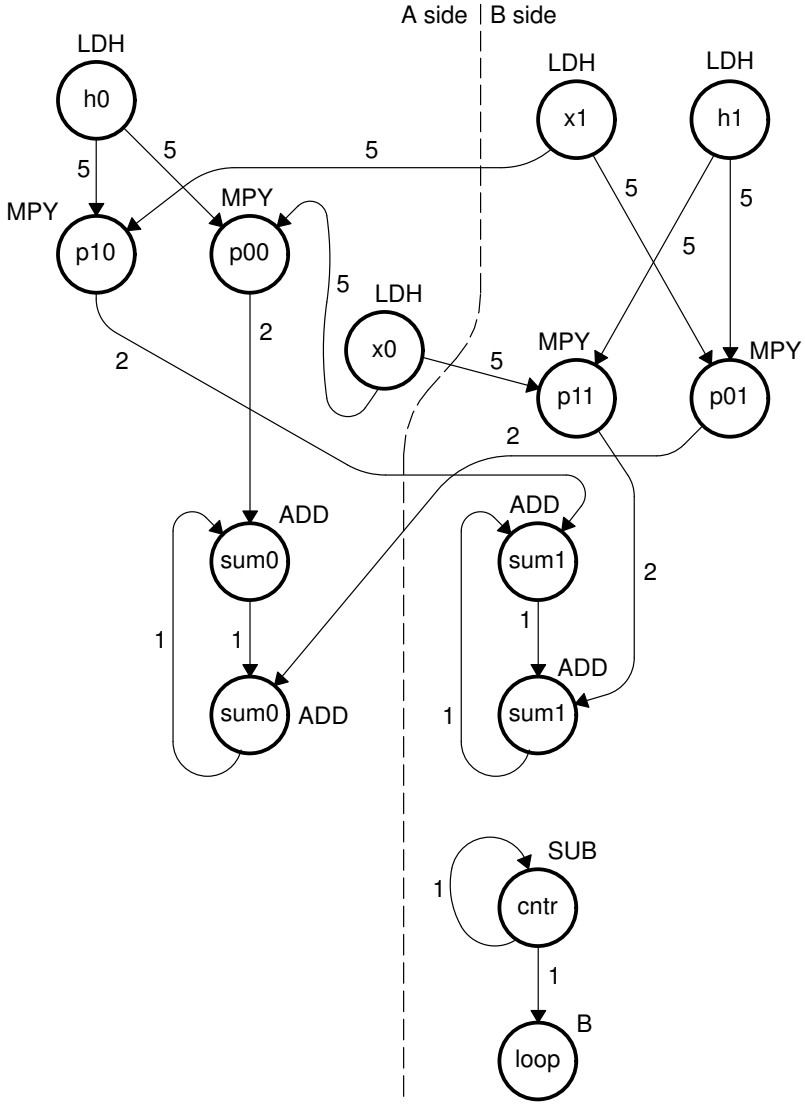
Example 4–41. List Of Symbolic FIR Instructions

```
          LDH     *Bx++[2],x1       ; x1 = x[j+i+1]
          LDH     *Ah++[2],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     *Ax++[2],x0       ; x0 = x[j+i+2]
          LDH     *Bx++[2],h1       ; h1 = h[i+1]
          MPY     x1,h1,p01         ; x1 * h1
          MPY     x0,h1,p11         ; x0 * h1
          ADD     p01,sum0,sum0     ; sum0 += x1 * h1
          ADD     p11,sum1,sum1     ; sum1 += x0 * h1

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

4.9.5 Dependency Graph

Figure 4–15 shows the dependency graph of the FIR filter.

Figure 4–15. Dependency Graph of FIR Filter With Redundant Load Elimination



4.9.6 Minimum Iteration Interval

Table 4–16 shows that the minimum iteration interval is 2. An iteration interval of 2 means that two multiply-accumulates are executing per cycle.

Table 4–16. Resource Table for FIR 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

4.9.7 ’C62xx Instructions (Inner Loop)

Example 4–42 shows the ’C62xx instructions with allocated resources.

Example 4–42. List of Actual FIR Instructions

```
        LDH     .D2     *B5++[2],B1     ; x1 = x[j+i+1]
        LDH     .D1     *A5++[2],A1     ; h0 = h[i]
        MPY     .M1     A0,A1,A7        ; x0 * h0
        MPY     .M1X    B1,A1,A8        ; x1 * h0
        ADD     .L1     A7,A9,A9        ; sum0 += x0 * h0
        ADD     .L2X    A8,B9,B9        ; sum1 += x1 * h0

        LDH     .D1     *A4++[2],A0     ; x0 = x[j+i+2]
        LDH     .D2     *B4++[2],B0     ; h1 = h[i+1]
        MPY     .M2     B1,B0,B7        ; x1 * h1
        MPY     .M2X    A0,B0,B8        ; x0 * h1
        ADD     .L1X    B7,A9,A9        ; sum0 += x1 * h1
        ADD     .L2     B8,B9,B9        ; sum1 += x0 * h1

[B2]    SUB     .S2     B2,1,B2         ; decrement loop counter
[B2]    B       .S2     LOOP            ; branch to loop
```

4.9.8 Final Assembly

Example 4–43 shows the final assembly for the FIR without redundant load instructions.

- At the end of the inner loop is a branch to OUTLOOP that executes the next outer loop.
- The outer loop counter is 50 because iterations j and j + 1 execute each time the inner loop is run.
- The inner loop counter is 16 because iterations i and i + 1 execute each inner loop iteration.

The cycle count for this nested loop is 2352 cycles: 50 (16 $\times$ 2 + 9 + 6) + 2. Fifteen cycles are overhead for each outer loop:

- Nine cycles execute the inner loop prologue.
- Six cycles execute the branch to the outer loop.

See Section 4.11, *Software Pipelining the Outer Loop*, for information on how to reduce this overhead.

Example 4–43. FIR 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]

          MV              A7,A7                                            (9)
||        MPY     .M2X    A0,B0,B8        ; x0 * h1
||        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 4–43 FIR 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)
```

4.10 Memory Banks

The internal memory of the ’C62xx family varies from device to device. See the *TMS320C62xx Peripherals Reference Guide* to determine the memory spaces in your particular device.

Most ’C62xx devices use an interleaved memory bank scheme, as shown in Figure 4–16. Each number in the diagram 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.

For devices that have more than one memory space (Figure 4–17), an access to Bank 0 in one space does not interfere with an access to Bank 0 in another memory space, and no pipeline stall occurs.

Figure 4–16. Four-Bank Interleaved Memory

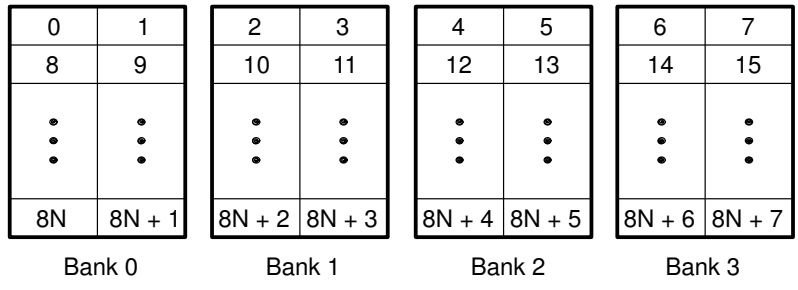
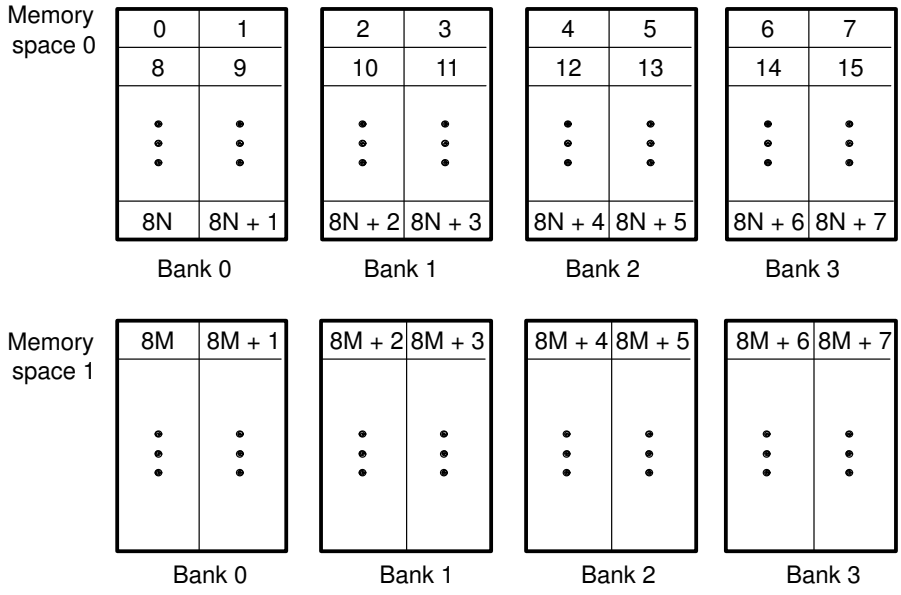


Figure 4–17. Four-Bank Interleaved Memory With Two Memory Spaces



4.10.1 FIR Inner Loop

Example 4–44 shows the inner loop from the final assembly in Example 4–43.

- 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 4–44 has no memory hits. However, if both *x[1]* and *h[0]* are on an even halfword in memory (Bank 0) and they are in the same memory space, every cycle incurs a memory pipeline stall and the loop runs at half the speed.

Example 4–44. Inner Loop of FIR

```
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 might have different alignments each time the function is called. One solution to this problem is to write a FIR filter that never has memory hits, regardless of the *x* and *h* array alignments.

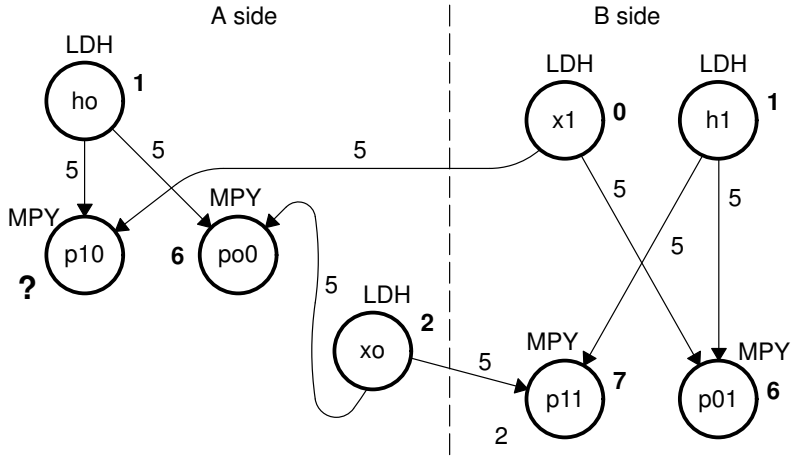
If accesses to the even and odd elements of an array (*h* or *x)* are scheduled on the same cycle, the accesses are always on adjacent memory banks. Thus, to write an FIR filter that cannot ever have any memory hits, even and odd elements of the same array must be scheduled on the same loop cycle.

In the case of the FIR, scheduling the even and odd elements of the same array on the same loop cycle cannot be done in a two-cycle loop, as shown in Figure 4–18, which bases a valid loop on the following constraints:

- *h0* and *h1* are on the same loop cycle.
- *x0* and *x1* are on the same loop cycle.
- MPY *p00* must be scheduled three or four cycles after LDH *x0*, since 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 4–18 shows one scenario that 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.

Figure 4–18. FIR With Even and Odd Elements of Each Array on Same Loop Cycle



Note: Numbers in bold represent the cycle the instruction is scheduled on.

4.10.2 Unrolled FIR C Code

The main limitation in solving the problem in Figure 4–18 is that you are trying to schedule a 2-cycle loop, which means that no values can be *live* more than two cycles. One solution is to increase the iteration interval to 3, but this decreases performance. Another way to solve this problem is to unroll the inner loop one more time and produce a 4-cycle loop.

Example 4–45, shows the FIR C code after unrolling the inner loop one more time. This solution adds to the flexibility of scheduling and allows you to write FIR code that never has memory hits, regardless of array alignment and memory space.

This solution is needed only when both the *h* and *x* arrays must reside in the same memory space; if each array resides in a separate memory space, the 2-cycle loop in Example 4–40 is sufficient.

Example 4–45. Unrolled FIR 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;
          }
}
```

4.10.3 Unrolled ’C62xx Instructions For the Inner Loop of the FIR

Example 4–46 shows a list of symbolic FIR instructions.

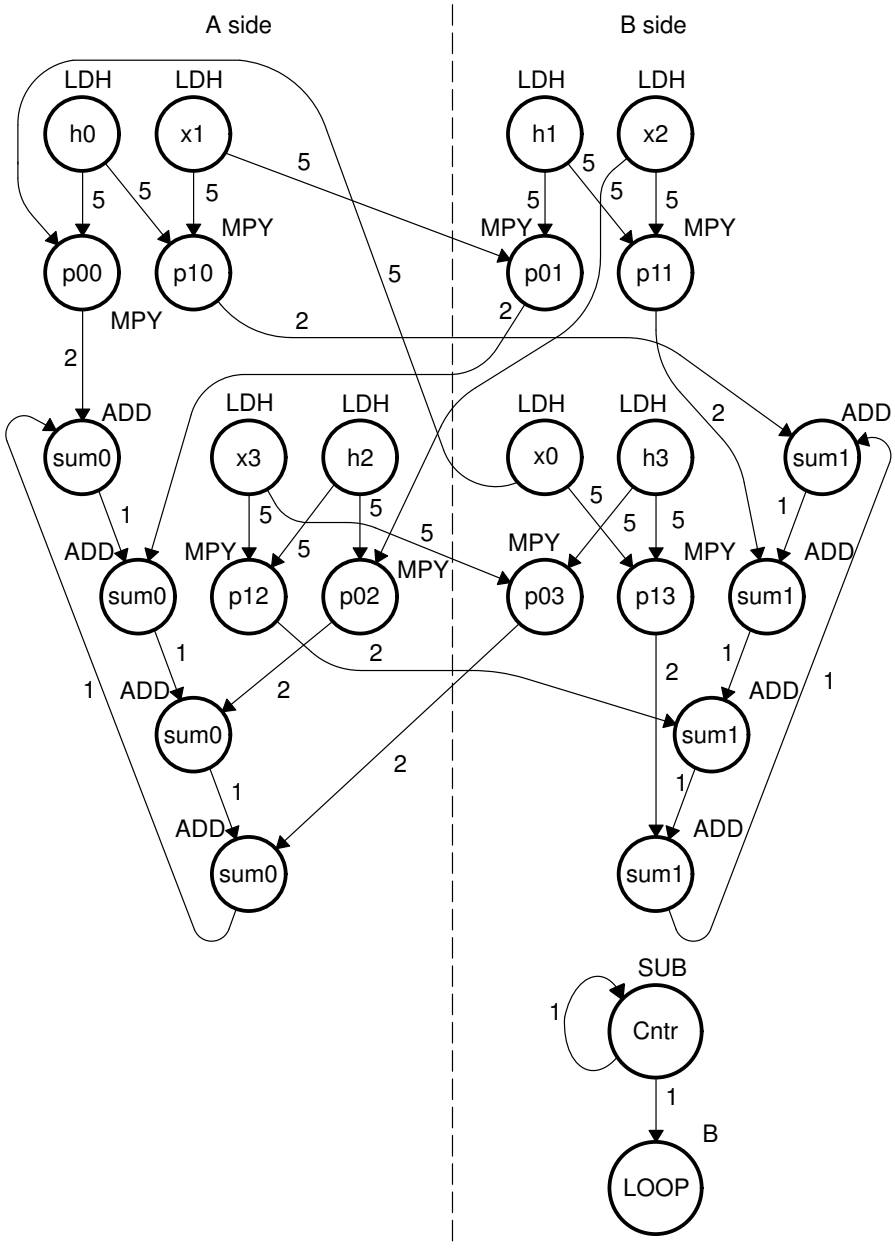
Example 4–46. List of Symbolic Unrolled FIR Instructions

```
         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
```

4.10.4 New Dependency Graph

Figure 4–19 shows the dependency graph of the FIR with no memory hits.

Figure 4–19. Dependency Graph of FIR With No Memory Hits



4.10.5 Unrolled Symbolic 'C62xx Instructions With Functional Units for Inner Loop

Example 4–47 shows the 'C62xx instructions with functional units assigned.

- Symbolic names now have an A or B in front of them to signify the register file where they reside.
- The actual register names are not chosen yet (as in previous examples) because they are allocated after scheduling the loop.
- Now there is a pointer to each array in both the A and B register files. Because each pointer references either the even elements or the odd elements, pointers are incremented by two halfwords instead of one as in Example 4–46.

Example 4–47. List of Symbolic Unrolled FIR Instructions

```
        LDH     .D1     *Ax++[2],Ax1            ; x1 = x[j+i+1]
        LDH     .D2     *Bh++[2],Ah0            ; h0 = h[i]
        MPY     .M1X    Bx0,Ah0,Ap0a            ; x0 * h0
        MPY     .M1     Ax1,Ah0,Ap1a            ; x1 * h0
        ADD     .L1     Ap0a,Asum0,Asum0        ; sum0 += x0 * h0
        ADD     .L2X    Ap1a,Bsum1,Bsum1        ; sum1 += x1 * h0

        LDH     .D2     *Bx++[2],Bx2            ; x2 = x[j+i+2]
        LDH     .D1     *Ah++[2],Bh1            ; h1 = h[i+1]
        MPY     .M2X    Ax1,Bh1,Bp0b            ; x1 * h1
        MPY     .M2     Bx2,Bh1,Bp1b            ; x2 * h1
        ADD     .L1X    Bp0b,Asum0,Asum0        ; sum0 += x1 * h1
        ADD     .L2     Bp1b,Bsum1,Bsum1        ; sum1 += x2 * h1

        LDH     .D1     *Ax++[2],Ax3            ; x3 = x[j+i+3]
        LDH     .D2     *Bh++[2],Ah2            ; h2 = h[i+2]
        MPY     .M1X    Bx2,Ah2,Ap0c            ; x2 * h2
        MPY     .M1     Ax3,Ah2,Ap1c            ; x3 * h2
        ADD     .L1     Ap0c,Asum0,Asum0        ; sum0 += x2 * h2
        ADD     .L2X    Ap1c,Bsum1,Bsum1        ; sum1 += x3 * h2

        LDH     .D2     *Bx++[2],Bx0            ; x0 = x[j+i+4]
        LDH     .D1     *Ah++[2],Bh3            ; h3 = h[i+3]
        MPY     .M2X    Ax3,Bh3,Bp0d            ; x3 * h3
        MPY     .M2     Bx0,Bh3,Bp1d            ; x0 * h3
        ADD     .L1X    Bp0d,Asum0,Asum0        ; sum0 += x3 * h3
        ADD     .L2     Bp1d,Bsum1,Bsum1        ; sum1 += x0 * h3


[Bcntr] SUB     .S2     Bcntr,1,Bcntr           ; decrement loop counter
[Bcntr] B       .S2     LOOP                    ; branch to loop
```

4.10.6 Register 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 all write to a unique register because the ’C62xx has only 32 registers. As a result, registers must share values that are not live on the same cycles in the loop.

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 *TMS320C6x Optimizing C Compiler User’s Guide* for more information.

4.10.7 Minimum Iteration Interval With No Memory Hits

Based on Table 4–17, the minimum iteration interval should be 4. An iteration interval of 4 means that two multiply/accumulates still execute per cycle.

Table 4–17. Resource Table for FIR 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

4.10.8 Final Assembly

Example 4–48 shows the final assembly to the FIR 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 & 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, i+3 are executing each inner loop iteration.

The cycle count for this nested loop is 50 (8 $\times$ 4 + 10 + 6) +2 = 2402 cycles. There is a rather large outer-loop overhead for executing the branch to the outer loop (6 cycles) and the inner loop prologue (10 cycles). Section 4.11 addresses how to reduce this overhead by software pipelining the outer loop.

Code Example	Cycles	Cycle Count
Example 4–43 FIR With Redundant Load Elimination	50 (16 $\times$ 2 + 9 + 6) + 2	2352
Example 4–48 FIR With Redundant Load Elimination and No Memory Hits	50 (8 $\times$ 4 + 10 + 6) + 2	2402

Example 4–48. FIR 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]                              (1)
||        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]                 (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]                   (3)
||        LDH     .D2     *B4++[2],A1       ; h0 = h[i]

          LDH     .D1     *A4++[2],A5       ; x3 = x[j+i+3]                 (4)
||        LDH     .D2     *B1++[2],B5       ; x0 = x[j+i+4]

          LDH     .D2     *B4++[2],A7       ; h2 = h[i+2]                   (5)
||        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]                (6)
||        LDH     .D1     *A4++[2],A0       ;* x1 = x[j+i+1]

          LDH     .D1     *A8++[2],B6       ;* h1 = h[i+1]                  (7)
||        LDH     .D2     *B4++[2],A1       ;* h0 = h[i]

          MPY     .M1X    B5,A1,A0          ; x0 * h0                       (8)
||        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                (9)
||        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               (10)
||        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 4–48 FIR 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          (1)
||        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                    (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)
```

4.11 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.

4.11.1 FIR C Code

Example 4–49 shows the unrolled C code for the FIR in Example 4–45, *Unrolled FIR C Code*, on page 4-89.

Example 4–49. Unrolled FIR 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;
          }
}
```

4.11.2 Making the Outer Loop Parallel With the Inner Loop Epilogue and Prologue

The final assembly in Example 4–48, *FIR With Redundant Load Elimination and No Memory Hits*, on page 4-95, contained 16 cycles of overhead to call the inner loop every time:

- ❑ Ten cycles for the loop prologue
- ❑ 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 prologue.
- ❑ Create an epilogue to the inner loop.
- ❑ Put some outer loop instructions in parallel with the inner-loop epilogue.

4.11.3 Final Assembly

Example 4–50 shows the final assembly for the FIR with a software pipelined outer loop. Below the inner loop (starting on page 4-100), each instruction is marked in the comments with an e, p, or o for instructions relating to epilogue, prologue, 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 prologue of the next inner loop and the outer loop instructions.

The improved cycle count for this loop is 2006 cycles: 50 ((7 $\times$ 4) + 6 + 6) + 6. The outer-loop overhead for this loop has been reduced from 16 to 8 (6 + 6 – 4); the –4 represents one iteration less for the inner-loop iteration (seven instead of eight).

Code Example	Cycles	Cycle Count
Example 4–43 FIR With Redundant Load Elimination	50 (16 $\times$ 2 + 9 + 6) + 2	2352
Example 4–48 FIR With Redundant Load Elimination and No Memory Hits	50 (8 $\times$ 4 + 10 + 6) + 2	2402
Example 4–50 FIR With Redundant Load Elimination and No Memory Hits With Outer Loop Software Pipelined	50 (7 $\times$ 4 + 6 + 6) + 6	2006

Example 4–50. FIR 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 4–50. FIR 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 4–50. FIR 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
```

4.12 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.

4.12.1 FIR C Code

Example 4–51 shows the unrolled FIR C code used in the previous example.

Example 4–51. Unrolled FIR 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;
          }
}
```

4.12.2 ’C62xx Instructions (Inner Loop)

Example 4–52 shows a list of symbolic FIR instructions.

Example 4–52. List of Symbolic Unrolled FIR Instructions

```
        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
```

4.12.3 'C62xx Instructions (Outer Loop)

Example 4–53 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 are conditional on the pointer reset counter, *prc*.
- The shifting and storing of the even and odd *y* elements is conditional on the store counter, *sc*.

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 4–53. List of Symbolic FIR Outer Loop Instructions

```
[sc]    SUB     sc,1,sc          ; dec store lp cntr
[!sc]   SHR     sum0,15,y0       ; (sum0 >> 15)
[!sc]   SHR     sum1,15,y1       ; (sum1 >> 15)
[!sc]   STH     y0,*y++[2]       ; y[j] = (sum0 >> 15)
[!sc]   STH     y1,*y++[2]       ; y[j+1] = (sum1 >> 15)
[!sc]   MVK     8,sc             ; reset store lp cntr

[prc]   SUB     prc,1,prc        ; dec pointer reset lp cntr
[!prc]  SUB     x,rstx,x         ; reset x ptr (A side)
[!prc]  SUB     x,rstx,x         ; reset x ptr (B side)
[!prc]  SUB     h,rsth,h         ; reset h ptr (A side)
[!prc]  SUB     h,rsth,h         ; reset h ptr (B side)
[!prc]  MVK     8,prc            ; reset pointer reset lp cntr
```

4.12.4 Unrolled FIR 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 again.

Example 4–54 shows the C code for the FIR, which operates on eight elements every inner loop. Two outer loops are also being processed together, as in Example 4–51.

Example 4–54. Unrolled FIR 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;
          }
}
```

4.12.5 ’C62xx Instructions (Inner Loop)

Example 4–55 shows the symbolic instructions that perform the inner and outer loops of the FIR:

- 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 *sc* 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 read 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 4–55. List of Symbolic FIR Instructions

```
         LDW      *h++[2],h01       ; h[i+0] & h[i+1]
         LDW      *h++[2],h23       ; h[i+2] & h[i+3]
         LDW      *h++[2],h45       ; h[i+4] & h[i+5]
         LDW      *h++[2],h67       ; h[i+6] & h[i+7]

         LDW      *x++[2],x01       ; x[j+i+0] & x[j+i+1]
         LDW      *x++[2],x23       ; x[j+i+2] & x[j+i+3]
         LDW      *x++[2],x45       ; x[j+i+4] & x[j+i+5]
         LDW      *x++[2],x67       ; x[j+i+6] & x[j+i+7]
         LDH      *x,x8             ; x[j+i+8]

[prc]    SUB      prc,1,prc         ; dec pointer reset lp cntr
[!prc]   SUB      x,rstx,x          ; reset x ptr (A side)
[!prc]   SUB      x,rstx,x          ; reset x ptr (B side)
[!prc]   SUB      h,rsth,h          ; reset h ptr (A side)
[!prc]   SUB      h,rsth,h          ; reset h ptr (B side)
[!prc]   MVK      8,prc             ; reset pointer reset lp cntr

[sc]     SUB      sc,1,sc           ; dec store lp cntr
[!sc]    SHR      sum0,15,y0        ; (sum0 >> 15)
[!sc]    SHR      sum1,15,y1        ; (sum1 >> 15)
[!sc]    STH      y0,*y++[2]        ; y[j] = (sum0 >> 15)
[!sc]    STH      y1,*y++[2]        ; y[j+1] = (sum1 >> 15)

         MPY      h01,x01,p00       ; p00 = h[i+0]*x[j+i+0]
[sc]     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

         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

         MPYLH    h01,x01,p10       ; p10 = h[i+0]*x[j+i+1]
[sc]     ADD      p10,sum17,p10     ; sum1(p10) = p10 + sum1
```

Example 4–55. List of Symbolic FIR Instructions (Continued)

```
         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

[!sc]    MVK      8,sc              ; reset store lp cntr

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

4.12.6 ’C62xx Instructions (Inner Loop and Outer Loop)

Example 4–56 shows the FIR instructions with functional units allocated. 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). Since 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 4–56. List of Actual FIR Instructions

```
        LDW     .D1     *Ah++[2],Ah01           ; h[i+0] & h[i+1]
        LDW     .D2     *Bh++[2],Bh23           ; h[i+2] & h[i+3]
        LDW     .D1     *Ah++[2],Ah45           ; h[i+4] & h[i+5]
        LDW     .D2     *Bh++[2],Bh67           ; h[i+6] & h[i+7]

        LDW     .D2     *Bx++[2],Ax01           ; x[j+i+0] & x[j+i+1]
        LDW     .D1     *Ax++[2],Bx23           ; x[j+i+2] & x[j+i+3]
        LDW     .D2     *Bx++[2],Ax45           ; x[j+i+4] & x[j+i+5]
        LDW     .D1     *Ax++[2],Bx67           ; x[j+i+6] & x[j+i+7]
        LDH     .D2     *Bx,Ax8                 ; x[j+i+8]

 [A2]   SUB     .S1     A2,1,A2                 ; dec store lp cntr
[!A2]   SHR     .S1     Asum07,15,Ay0           ; (Asum0 >> 15)
[!A2]   SHR     .S2     Bsum17,15,By1           ; (Bsum1 >> 15)
[!A2]   STH     .D1     Ay0,*Ay++[2]            ; y[j] = (Asum0 >> 15)
[!A2]   STH     .D2     By1,*By++[2]            ; y[j+1] = (Bsum1 >> 15)

        MPY     .M1     Ah01,Ax01,Ap00          ; p00 = h[i+0]*x[j+i+0]
[A2]    ADD     .L1     Ap00,Asum07,Ap00        ; sum0(p00) = p00 + sum0

        MPYH    .M1     Ah01,Ax01,Ap01          ; p01 = h[i+1]*x[j+i+1]
        ADD     .L1     Ap01,Ap00,Asum01        ; sum0 += p01

        MPY     .M2     Bh23,Bx23,Bp02          ; p02 = h[i+2]*x[j+i+2]
        ADD     .L1X    Bp02,Asum01,Asum02      ; sum0 += p02

        MPYH    .M2     Bh23,Bx23,Bp03          ; p03 = h[i+3]*x[j+i+3]
        ADD     .L1X    Bp03,Asum02,Asum03      ; sum0 += p03

        MPY     .M1     Ah45,Ax45,Ap04          ; p04 = h[i+4]*x[j+i+4]
        ADD     .L1     Ap04,Asum03,Asum04      ; sum0 += p04

        MPYH    .M1     Ah45,Ax45,Ap05          ; p05 = h[i+5]*x[j+i+5]
        ADD     .L1     Ap05,Asum04,Asum05      ; sum0 += p05

        MPY     .M2     Bh67,Bx67,Bp06          ; p06 = h[i+6]*x[j+i+6]
        ADD     .L1X    Bp06,Asum05,Asum06      ; sum0 += p06

        MPYH    .M2     Bh67,Bx67,Bp07          ; p07 = h[i+7]*x[j+i+7]
        ADD     .L1X    Bp07,Asum06,Asum07      ; sum0 += p07

        MV      .L2X    Ax01,Bx01               ; move to other reg file
        MPYLH   .M2X    Ah01,Bx01,Bp10          ; p10 = h[i+0]*x[j+i+1]
[A2]    ADD     .L2     Bp10,Bsum17,Bp10        ; sum1(p10) = p10 + sum1

        MPYHL   .M1X    Ah01,Bx23,Ap11          ; p11 = h[i+1]*x[j+i+2]
        ADD     .L2X    Ap11,Bp10,Bsum11        ; sum1 += p11

        MPYLH   .M2     Bh23,Bx23,Bp12          ; p12 = h[i+2]*x[j+i+3]
        ADD     .L2     Bp12,Bsum11,Bsum12      ; sum1 +=  p12
```

Example 4–56. List of Actual FIR Instructions (Continued)

```
        MPYHL    .M1X     Bh23,Ax45,Ap13          ; p13 = h[i+3]*x[j+i+4]
        ADD      .L2X     Ap13,Bsum12,Bsum13      ; sum1 += p13

        MPYLH    .M1      Ah45,Ax45,Ap14          ; p14 = h[i+4]*x[j+i+5]
        ADD      .L2X     Ap14,Bsum13,Bsum14      ; sum1 += p14

        MPYHL    .M2X     Ah45,Bx67,Bp15          ; p15 = h[i+5]*x[j+i+6]
        ADD      .S2      Bp15,Bsum14,Bsum15      ; sum1 += p15

        MPYLH    .M2      Bh67,Bx67,Bp16          ; p16 = h[i+6]*x[j+i+7]
        ADD      .L2      Bp16,Bsum15,Bsum16      ; sum1 += p16

        MPYHL    .M1X     Bh67,Ax8,Ap17           ; p17 = h[i+7]*x[j+i+8]
        ADD      .L2X     Ap17,Bsum16,Bsum17      ; sum1 += p17

 [!A2]  MVK      .S1      4,A2                    ; reset store lp cntr

 [A1]   SUB      .S1      A1,1,A1                 ; dec ptr reset lp cntr
 [!A1]  SUB      .S2      Bx,Brstx,Bx             ; reset x ptr
 [!A1]  SUB      .S1      Ax,Arstx,Ax             ; reset x ptr
 [!A1]  SUB      .S1      Ah,Arsth,Ah             ; reset h ptr
 [!A1]  SUB      .S2      Bh,Brsth,Bh             ; reset h ptr
 [!A1]  MVK      .S1      4,A1                    ; reset ptr rst lp cntr

 [B0]   SUB      .S2      B0,1,B0                 ; dec outer lp cntr
 [B0]   B        .S2      LOOP                    ; Branch outer loop
```

4.12.7 Minimum Iteration Interval

Based on Table 4–18, the minimum iteration interval is 8. An iteration interval of 8 means that two multiply-accumulates per cycle are still executing.

Table 4–18. Resource Table for FIR

(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

4.12.8 Final Assembly

Example 4–57 shows the final assembly for the FIR with the outer loop conditionally executing in parallel with the inner loop. The cycle count of this code is 1612: 50 (8 × 4 + 0) + 12. The overhead due to the outer loop has been completely eliminated.

Code Example		Cycles	Cycle Count
Example 4–40	FIR With Redundant Load Elimination	50 (16 × 2 + 9 + 6) + 2	2352
Example 4–48	FIR With Redundant Load Elimination and No Memory Hits	50 (8 × 4 + 10 + 6) + 2	2402
Example 4–50	FIR With Redundant Load Elimination and No Memory Hits With Outer Loop Software Pipelined	50 (7 × 4 + 6 + 6) + 6	2006
Example 4–53	FIR With Redundant Load Elimination and No Memory Hits With Outer Loop Conditionally Executed With Inner Loop	50 (8 × 4 + 0) + 12	1612

Example 4–57. Final Assembly for FIR

```
          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 4–57. Final Assembly for FIR (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 4–57. Final Assembly for FIR (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)
```

Chapter 5

Applications Programming

This chapter provides extensive code examples to supplement those found in Chapters 2–4.

Topic **Page**

5.1 Summary of Major Programming Methods

The key to implementing applications on the ’C62xx is to take advantage of the processor’s full speed. The main technique for achieving this goal involves unrolling software loops to reach the limits of the functional units while meeting the data dependency constraints.

In addition to loop unrolling, the following methods are helpful for improving performance:

- Rearranging the C code

  If you are implementing a system based on an existing C code, rearranging the tasks in the C code is a useful method to gain better performance.

- Avoiding memory bank hits

  Memory bank hits, especially those in the inner loop in a nested loop application, hurt the performance dramatically and must be avoided. Most of the memory bank hits, however, can be eliminated by allocating the relevant arrays properly. Some situations, like accessing a word and a half-word in the same cycle, can also create the chance of a memory bank hit and should also be avoided.

  If the system implementation is quite complicated, the program-memory size becomes an issue. To achieve a good balance between program-memory size and speed, you can implement the less critical portions with highly-compact assembly code that sacrifices on performance.

5.2 Implementation of GSM EFR Vocoder

This section presents the implementation of some representative pieces of the enhanced full rate (EFR) vocoder for the Global Systems for Mobile Communications (GSM).

Note: Copyright for C Code

European Telecommunications Standards Institute (ETSI) has the copyright to all the C code used in this section.

The following are the global constants/symbols defined in EFR:

```
#define Word16    short
#define Word32    int
#define MAX_32    0x7fffffffL
#define MIN_32    0x80000000L
#define MAX_16    0x7fff
#define MIN_16    0x8000
```

5.2.1 Implementation of the Multiply-Accumulate Loop

First, examine the most popular loop used in almost every fixed point vocoder, the multiply-accumulate (MAC) loop, shown in Example 5–1.

Example 5–1. Typical MAC Loop C Code

```
input:
         Word16  N;  (typical value of N is an even integer,
                     greater than or equal to 20)
         Word16  *x, *y;

result:
         Word32 sum;

C Code
-------------------------------------------------------------------------------
         sum=0;
         for(i=0;i<N;i++) sum=L_mac(sum,x[i],y[i]);
-------------------------------------------------------------------------------
where    L_mac(a,b,c) = _sadd(a,_smpy(b,c))
```

Example 5–2 shows a list of symbolic instructions for each iteration of the loop.

Example 5–2. Symbolic Instructions of the MAC Loop

```
LOOP:
       LDH    .D  *xptr++, xi      ; load x[i]
       LDH    .D  *yptr++, yi      ; load y[i]
       SMPY   .M  xi,yi,tmp        ; smpy(x[i],y[i])
       SADD   .L  sum,tmp,sum      ; sum=sadd(sum,smpy(x[i],y[i])
[cntr] SUB    .ALU   cntr,1,cntr   ; decrement the loop counter
[cntr] B      .S  LOOP             ; branch to the loop
```

In Example 5–2, *xptr* and *yptr* are the pointers for *x* and *y*, respectively. These instructions can easily fit into one execution packet, but two functional units are not used.

In general, unrolling the loop once as in the code in Example 5–3, does not give the same result as the code shown in Example 5–1, because of the ordering dependence of the saturated addition.

Example 5–3. MAC Loop C Code With Loop Unrolling

```
Word32 sum_e, sum_o;
       sum_e=0;
       sum_o=0;
       for(i=0;i<N;i+=2) {
           sum_e=L_mac(sum_e,x[i],y[i]);
           sum_o=L_mac(sum_o,x[i+1],y[i+1]);
       }
       sum=L_add(sum_o,sum_e);
-------------------------------------------------------------------------------
where  L_add(a,b)=_sadd(a,b)
```

Both approaches lead to the same result if *x[i]=y[i]* for every *i*, since *_smpy(x[i], x[i]* is always greater than or equal to 0. This special MAC loop is used to compute the energy of a particular signal segment. In this case, take the approach shown in Example 5–3, since it doubles the performance of Example 5–1. Example 5–4 shows the C code for this special MAC loop. Example 5–5 lists the symbolic instructions for this loop, with one loop unrolling.

Example 5–4. Special MAC Loop C Code

```
    sum=0;
    for(i=0;i<N;i++)
    sum = L_mac(sum,x[i],x[i]);

or
    sum_e=0;
    sum_o=0;
    for(i=0;i<N;i+=2) {
         sum_e=L_mac(sum_e,x[i],x[i]);
         sum_o=L_mac(sum_o,x[i+1],x[i+1]);
    }
    sum=L_add(sum_o,sum_e);
```

Example 5–5. Special MAC Loop Symbolic Instructions

```
LOOP:
       LDH    .D  *xptre++, xi            ; load x[i]
       SMPY   .M  xi,xi,tmp_e             ; smpy(x[i],x[i])
       SADD   .L  sum_e,tmp_e,sum_e       ; sum_e=sadd(sum_e,smpy(x[i],x[i])
       LDH    .D  *xptro++, xi+1          ; load x[i+1]
       SMPY   .M  xi+1,xi+1,tmp_o         ; smpy(x[i+1],x[i+1])
       SADD   .L  sum_o,tmp_o,sum_  o     ; sum_o=sadd(sum_o,smpy(x[i+1],x[i+1])
[cntr] SUB    .S  cntr,2,cntr             ; decrement the loop counter
[cntr] B      .S  LOOP                    ; branch to the loop
       SADD   .L  sum_e, sum_o, sum       ; sum=sadd(sum_o+sum_e)
```

In Example 5–5, *xptre* and *xptro* are the pointers for *x* and point to *x[0]* and *x[1]*, respectively, at the beginning. The eight instructions in the loop fit perfectly into one execution packet. Notice that this approach computes two MACs in one cycle. It doubles the performance of Example 5–2 for the general MAC loop.

The final assembly codes are shown in Example 5–6.

Example 5–6. Final Assembly Code for the Energy Computation MAC Loop

```
*******************************************************************************
**        Texas Instruments, Inc                                             **
**                                                                           **
**        MAC Loop -- Energy Computation                                     **
**                                                                           **
**        Compute two samples a time                                         **
**                                                                           **
**        Total cycles = (N/2+2)                                             **
**                                                                           **
**        Register Usage:          A              B                         **
**                                 3              3                         **
**                                                                           **
**        Notice that x[0] and x[1] will not be available till LOOP          **
**        is executed once. Therefore, sum_e and sum_o should be 0s          **
**        for the first three iterations. This is why A5, B5, A6,            **
**        and B6 should be set to 0s in the prelog.                          **
*******************************************************************************

                        ; A4 -- &x[0]
                        ; B4 -- N
                        ; A6 -- sum

          ADD    .L2X   A4,2,B4          ; &x[1]
||        SUB    .D2    B4,6,B1          ; loop counter
||        B      .S2    LOOP             ; branch to the loop
||        MVK    .S1    0,A6             ; initialize sum_e

          LDH    .D1    *A4++[2],A5      ; load x[0]
||        LDH    .D2    *B4++[2],B5      ; load x[1]
||        B      .S2    LOOP             ; branch to the loop
||        MV     .L2X   A6,B6            ; initialize sum_o

          LDH    .D1    *A4++[2],A5      ; load x[2]
||        LDH    .D2    *B4++[2],B5      ; load x[3]
||        B      .S1    LOOP             ; branch to the loop
||        MV     .L1    A6,A5            ; take care the initial three iterations
||        MV     .L2    B6,B5            ; take care the initial three iterations

          LDH    .D1    *A4++[2],A5      ; load x[4]
||        LDH    .D2    *B4++[2],B5      ; load x[5]
||        B      .S1    LOOP

          LDH    .D1    *A4++[2],A5      ; load x[6]
||        LDH    .D2    *B4++[2],B5      ; load x[7]
```

Example 5–6.Final Assembly Code for the Energy Computation MAC Loop(Continued)

```
LOOP:
       SMPY   .M1    A5,A5,A          ; smpy(x[i],x[i])
||     SMPY   .M2    B5,B5,B7         ; smpy(x[i+1],x[i+1])
||     SADD   .L1    A7,A6,A6         ; sum_e=sadd(sum_e,smpy(x[i],x[i]))
||     SADD   .L2    B7,B6,B6         ; sum_o=sadd(sum_o,smpy(x[i+1],x[i+1]))
||     LDH    .D1    *A4++[2],A5      ; load x[i]
||     LDH    .D2    *B4++[2],B5      ; load x[i+1]
||[B1] B      .S1    LOOP             ; branch to the loop
||[B1] SUB    .S2    B1,2,B1          ; decrement loop counter

       SADD   .L1X   A6,B6,A6         ; final result, sum = sum_e + sum_o
```

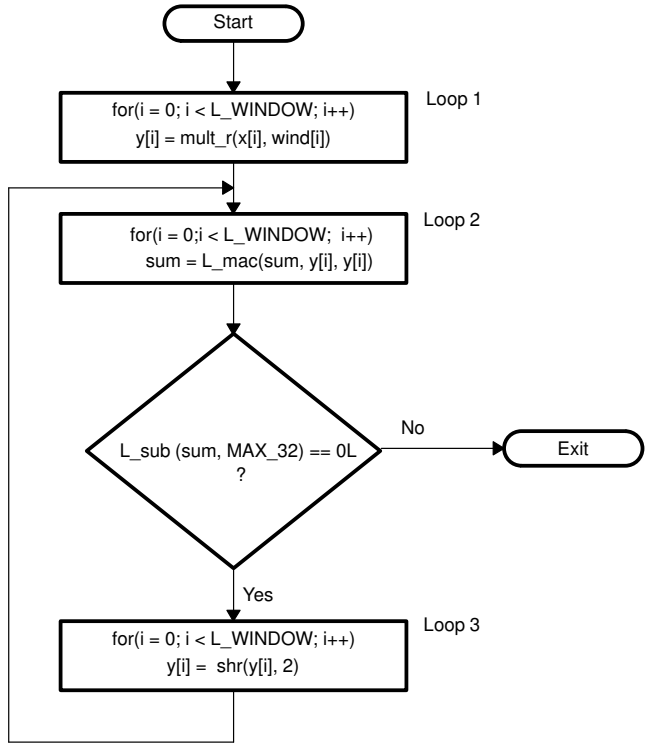
5.2.2 Implementation of the Windowing and Scaling Part of autocorr.c

autocorr.c is one of the top ten most computationally intensive modules for the EFR. The part used in Example 5–7 is used for windowing speech samples and for scaling down the windowed sample sequence if the input level is too high.

Example 5–7. C Code for Windowing and Scaling Part of autocorr.c

```
#define L_WINDOW      240

input:
        Word16 x[L_WINDOW], wind[L_WINDOW];

local variables/arrays:
        Word16 i;
        Word16 y[L_WINDOW];
        Word32 sum;
        Word16 overfl, overfl_shft;

Original C code:
-----------------------------------------------------------

        /* Windowing of signal */

for (i = 0; i < L_WINDOW; i++)
{
        y[i] = mult_r (x[i], wind[i]);
}

/* Compute r[0] and test for overflow */

overfl_shft = 0;

do
{
        overfl = 0;
        sum = 0L;

        for (i = 0; i < L_WINDOW; i++)
        {
                        sum = L_mac (sum, y[i], y[i]);
        }

        /* If overflow divide y[] by 4 */

        if (L_sub (sum, MAX_32) == 0L)
        {
                overfl_shft = add (overfl_shft, 4);
                overfl = 1;             /* Set the overflow flag */

                for (i = 0; i < L_WINDOW; i++)
                {
                        y[i] = shr (y[i], 2);
                }
          }
}
while (overfl != 0);
-----------------------------------------------------------
Where   mult_r(a,b) = _sadd(_smpy(a,b),0x8000L)>>16
        L_mac(a,b,c)= _sadd(a,_smpy(b,c))
        L_sub(a,b) = _ssub(a,b)
        add(a,b) = ((_sadd((a)<<16,((b)<<16)))>>16)
        shr(a,b) = ((b)<0 ? (_sshl((a),(-b+16))>>16):((a)>>(b)))
```

Figure 5–1. Flow Diagram for Example 5–7



5.2.2.1 Unrolling the Loop

Try the unrolling loop technique for each loop.

Example 5–8 is the list of symbolic instructions needed to execute one iteration of Loop 1. You can use any arithmetic logic unit (ALU) for the loop-counter update.

Example 5–8. Instructions to Execute One Iteration of Loop 1

```
LOOP:
       LDH     .D      *windptr++,windi          ;load wind[i]
       LDH     .D      *xptr++,xi                ;load x[i]
       SMPY    .M      windi,xi,windxi0          ;smpy(x[i],wind[i])
       SADD    .L      windxi0,0x8000,wind1xi1   ;sadd(smpy(x[i],wind[i]),0x8000L)
       SHR     .S      windxi1,16,yi             ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
       STH     .D      yi,*yptr++                ;store y[i]
[cntr] SUB     .ALU    cntr,1,cntr               ;decrement loop counter
[cntr] B       .S      LOOP                      ;branch to loop
```

In Example 5–8, *windptr*, *xptr*, and *yptr* are the pointers of *wind*, *x*, and *y*.

The unit used most often (three times) is the .D unit (or the .S unit). With properly partitioned resources, this is a two-cycle loop.

If you unroll the loop once and load both *x* and *wind* in words (in GSM EFR, both *x* and *wind* can be loaded in words if they are map-aligned with the word boundary), you can compute two ys with two cycles. The following is the new list of the instructions in a loop iteration.

Example 5–9. New Instructions for Loop 1

```
LOOP:
        LDW     .D  *windptr++,windi_windi+1          ;load wind[i] and wind[i+1]
        LDW     .D  *xptr++,xi_xi+1                   ;load x[i] and x[i+1]
        SMPY    .M  windi_windi+1,xi_xi+1,windxi0     ;smpy(x[i],wind[i])
        SMPYH   .M  windi_windi+1,xi_xi+1,windxi0+1   ;smpy(x[i+1],wind[i+1])
        SADD    .L  windxi0,0x8000,windxi1            ;sadd(smpy(x[i],wind[i]),0x8000L)
        SADD    .L  windxi0+1,0x8000,windxi1+1        ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
        SHR     .S  windxi1,16,yi                     ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
        SHR     .S  windxi1+1,16,yi+1                 ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
        STH     .D  yi,*yptre++[2]                    ;store y[i]
        STH     .D  yi+1,*yptro++[2]                  ;store y[i+1]
[cntr]  SUB     .S  cntr,2,cntr                       ;decrement loop counter
[cntr]  B       .S  LOOP                              ;branch to loop
```

In Example 5–9, *yptre* and *yptro* are the pointers for *y* and point to *y[0]* and *y[1]*, respectively, at the beginning.

Note:

Loop 2 is a special MAC loop, as described in subsection 5.2.1 on page 5-3. It can be implemented either as shown in Example 5–10 without loop unrolling or as in Example 5–11 with loop unrolling once.

Example 5–10. List of Instructions for Loop 2, No Loop Unrolling

```
        LDH     .D      *yptr++,yi          ;load y[i]
        SMPY    .M      yi,yi,yyi           ;smpy(y[i],y[i])
        SADD    .L      sum,yyi,sum         ;sadd(sum,smpy(y[i],y[i]))
 [cntr] SUB     .S      cntr,1,cntr         ;decrement loop counter
 [cntr] B       .S      LOOP                ;branch to loop
```

Example 5–11. Instructions for Loop 2 With Loop Unrolling

```
LOOP:
       LDH    .D     *yptre++,yi          ;load y[i]
       LDH    .D     *yptro++,yi+1        ;load y[i+1]
       SMPY   .M     yi,yi,yyi            ;smpy(y[i],y[i])
       SMPY   .M     yi+1,yi+1,yyi+1      ;smpy(y[i+1],y[i+1])
       SADD   .L     sum_e,yyi,sum_e      ;sadd(sum_e,smpy(y[i],y[i])
       SADD   .L     sum_o,yyi+1,sum_o    ;sadd(sum_o,smpy(y[i+1],y[i+1]))
[cntr] SUB    .S     cntr,2,cntr          ;decrement loop counter
[cntr] B      .S     LOOP                 ;branch to loop

       SADD   .L     sum_e,sum_o,sum      ;sum=sum_o+sum_e
```

Later, you will see both approaches used in this application.

Loop 3 is a single-cycle loop and you cannot speed it up by simply unrolling the loop. The instructions for each iteration are shown in Example 5–12.

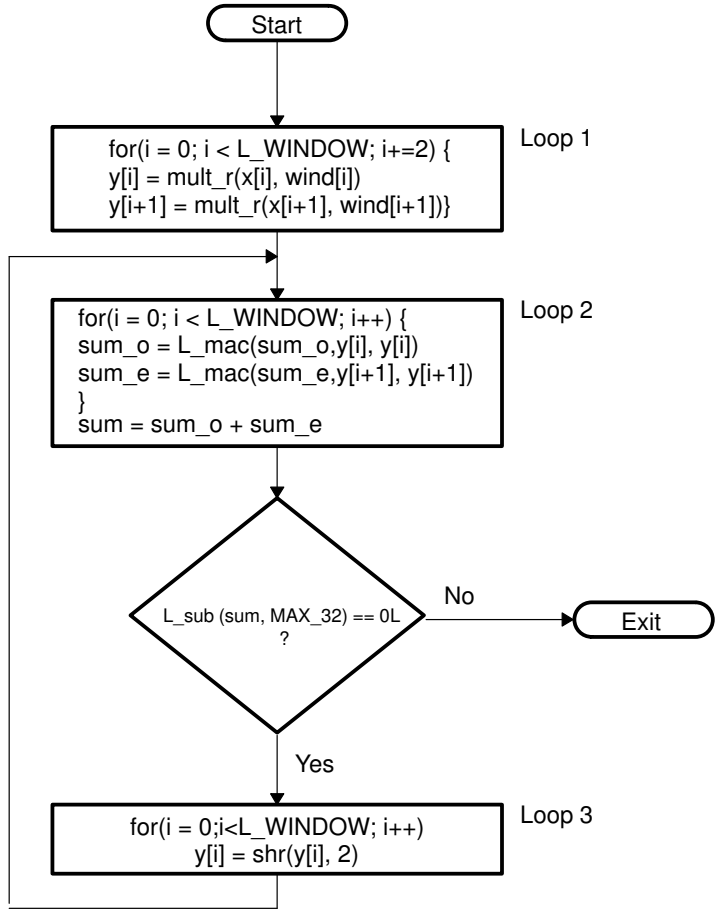
Example 5–12. Instructions for Loop 3

```
LOOP:
       LDH    .D     *yptrl++,yi          ;load y[i]
       SHR    .S     yi,2,yi0             ;shr(y[i],2)
       STH    .D     yi0,*yptrs++         ;store y[i]=shr(y[i],2)
[cntr] SUB    .L     cntr,1,cntr          ;decrement loop counter
[cntr] B      .S     LOOP                 ;branch to loop
```

In Example 5–12, *yptrl* and *yptrs* are the pointer of array *y* for loading and storing purposes, respectively.

The new flow diagram is shown in Figure 5–2.

Figure 5–2. Unrolling the Loop



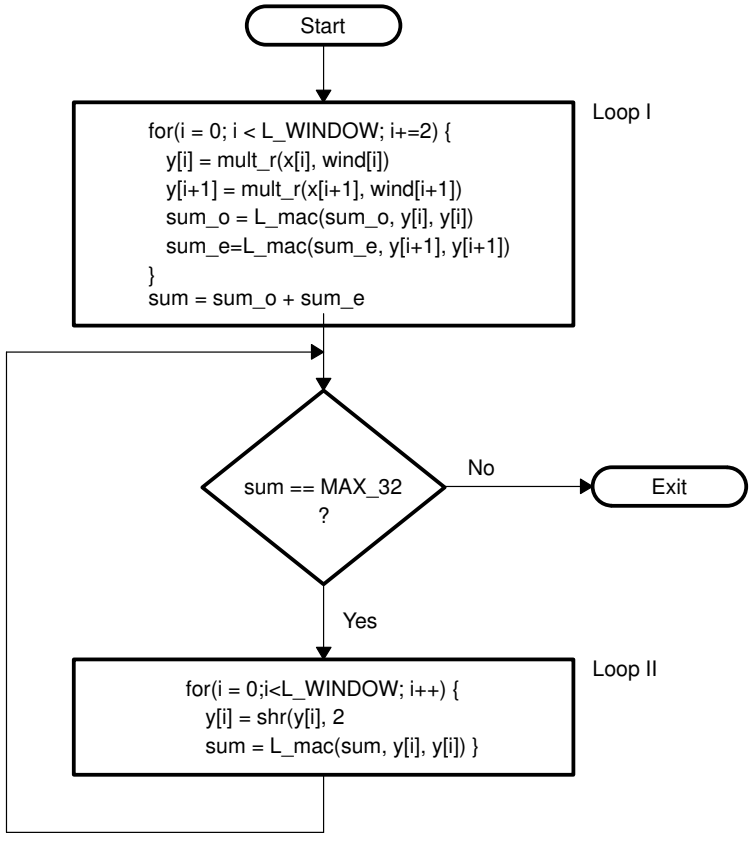
5.2.2.2 Rearrange the C Code

The first execution of Loop 2 can be combined with Loop 1 to form Loop I and its subsequent executions could be combined with Loop 3 to form Loop II.

One small change is the implementation of if L_sub(sum, MAX_32) == 0L as sum == MAX_32.

The new flow diagram with rearranged C code is shown in Figure 5–3.

Figure 5–3. Flow Diagram With Rearranged C Code



You could use Loop I as one of the two approaches as shown in Example 5–13.

Example 5–13. Instructions for Loop I

```
LOOP:
        LDW    .D  *windptr++,windi_windi+1          ;load wind[i] and wind[i+1]
        LDW    .D  *xptr++,xi_xi+1                   ;load x[i] and x[i+1]
        SMPY   .M  windi_windi+1,xi_xi+1,windxi0     ;smpy(x[i],wind[i])
        SMPYH  .M  windi_windi+1,xi_xi+1,windxi0+1   ;smpy(x[i+1],wind[i+1])
        SADD   .L  windxi0,0x8000,windxi1            ;sadd(smpy(x[i],wind[i]),0x8000L)
        SADD   .L  windxi0+1,0x8000,windxi1+1        ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
        SHR    .S  windxi1,16,yi                     ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
        SHR    .S  windxi1+1,16,yi+1                 ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
        SMPY   .M  yi,yi,yyi                         ;smpy(y[i],y[i])
        SMPY   .M  yi+1,yi+1,yyi+1                   ;smpy(y[i+1],y[i+1])
        SADD   .L  sum_e,yyi,sum_e                   ;sum_e=sadd(sum_e,smpy(y[i],y[i]))
        SADD   .L  sum_o,yyi+1,sum_o                 ;sum_o=sadd(sum_o,smpy(y[i+1],y[i+1])
        STD    .D  yi,*yptre++[2]                    ;store y[i]
        STD    .D  yi+1,*yptro++[2]                  ;store y[i+1]
[cntr]  SUB    .S  cntr,2,cntr                       ;decrement loop counter
[cntr]  B      .S  LOOP                              ;branch to loop
```

or as

```
 LOOP:
         LDW    .D  *windptr++,windi_windi+1          ;load wind[i] and wind[i+1]
         LDW    .D  *xptr++,xi_xi+1                   ;load x[i] and x[i+1]
         SMPY   .M  windi_windi+1,xi_xi+1,windxi0     ;smpy(x[i],wind[i])
         SMPYH  .M  windi_windi+1,xi_xi+1,windxi0+1   ;smpy(x[i+1],wind[i+1])
         SADD   .L  windxi0,0x8000,windxi1            ;sadd(smpy(x[i],wind[i]),0x8000L)
         SADD   .L  windxi0+1,0x8000,windxi1+1        ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)
         SHR    .S  windxi1,16,yi                     ;sadd(smpy(x[i],wind[i]),0x8000L)>>16
         SHR    .S  windxi1+1,16,yi+1                 ;sadd(smpy(x[i+1],wind[i+1]),0x8000L)>>16
         SMPYH  .M  windxi1,windxi1,yyi               ;smpy(y[i],y[i])
         SMPYH  .M  windxi+1,windxi+1,yyi+1           ;smpy(y[i+1],y[i+1])
         SADD   .L  sum_e,yyi,sum_e                   ;sum_e=sadd(sum_e,smpy(y[i],y[i]))
         SADD   .L  sum_o,yyi+1,sum_o                 ;sum_o=sadd(sum_o,smpy(y[i+1],y[i+1])
         STD    .D  yi,*yptre++[2]                    ;store y[i]
         STD    .D  yi+1,*yptro++[2]                  ;store y[i+1]
 [cntr]  SUB    .S  cntr,2,cntr                       ;decrement loop counter
 [cntr]  B      .S  LOOP                              ;branch to loop
```

The only difference between these two implementations is how to compute *sum_e* and *sum_o*. Using *sum_o* as an example, the former approach computes *sum_o* following the order of the original C code:

```
sum_o = _sadd(sum_o, _smpy(_sadd(_smpy(a,b),0x8000L)>>16,
        _sadd(_smpy(a,b),0x8000L)>>16)),
```

The latter computes sum_o in a slightly different way as:

```
sum_o = _sadd(sum_o, _smpyh(_sadd(_smpy(a,b),0x8000),
        _sadd(_smpy(a,b),0x8000))).
```

This provides the flexibility to better pack the instructions and reduces cycle count.

Loop I is a two-cycle loop. Loop II is still a single-cycle loop. Its instructions are shown in Example 5–14.

Example 5–14. Instructions for Loop II

```
LOOP:
       LDH    .D     *yptrl++,yi   ;load y[i]
       SHR    .S     yi,2,yi0      ;shr(y[i],2)
       SMPY   .M     yi0,yi0,yyi   ;smpy(shr(y[i],2),shr(y[i],2))
       SADD   .L     sum,yyi,sum   ;sum=sadd(sum,smpy(shr(y[i],2),shr(y[i],2)))
       STH    .D     yi0,*yptrs++  ;store y[i]=shr(y[i],2)
[cntr] SUB    .L     cntr,1,cntr   ;decrement loop counter
[cntr] B      .S     LOOP          ;branch to loop
```

5.2.2.3 Memory-Bank Hits

To schedule Loop I as a two-cycle loop:

- *x[i] + x[i + 1] << 16* and *wind[i] + wind[i + 1] << 16* must be loaded in the same cycle.
- *y[i]* and *y[i+1]* must be stored in the other cycle.

To avoid a memory-bank hit:

- Allocate *x* and *wind* in different memory spaces, if possible. For instance, allocate *wind[i]* in data ROM and *x* in data RAM.
- If no data ROM is available, allocate *x* and *wind* so they are offset from each other by one word.

There is no memory-bank hit problem when storing *y[i]* and *y[i +1]*. No memory-bank hits occur in Loop II, since the distance between the load and store is always six halfwords.

This part of autocorr.c C code is implemented in Example 5–15.

Example 5–15. Implemented C Code for autocorr.c

```
Word16 i;
Word16 y[L_WINDOW];
Word32 sum, sum_e, sum_o;
Word16 overfl, overfl_shft;

/* Windowing of signal */

sum_e=sum_o=0L;
for (i = 0; i < L_WINDOW; i+=2)
{
       y[i] = mult_r (x[i], wind[i]);
       y[i+1] = mult_r(x[i+1], window[i+1]);
       sum_e = L_mac(sum_e, y[i], y[i]);
       sum_o = L_mac(sum_o, y[i+1], y[i+1]);
}
sum=sum_e+sum_o;

/* Compute r[0] and test for overflow */

overfl_shft = 0;

do
{
       overfl = 0;
       /* If overflow divide y[] by 4 */

       if (sum == MAX_32)
       {
              overfl_shft = add (overfl_shft, 4);
              overfl = 1;                  /* Set the overflow flag */

              sum=0L;
              for (i = 0; i < L_WINDOW; i++)
              {
                     y[i] = shr (y[i], 2);
                     sum = L_mac(sum, y[i], y[i]);
              }
       }
}
while (overfl != 0);
```

5.2.2.4 Code-Size Reduction

Finally, consider the code-size reduction referred to in Figure 5–3 on page 5-13. Notice that Loop I is always executed and that Loop II is not executed, except for high-input levels. This means that cycle count is the most important factor for Loop I, while code size is more critical for Loop II.

5.2.2.5 Final Assembly Code

The final assembly code is presented in Example 5–16.

Example 5–16. Final Assembly Code for Windowing and Scaling of autocorr.c

```
*******************************************************************************
**      Texas Instruments, Inc                                                **
**                                                                            **
**      Implementation of The Windowing and Scaling Part of autocorr.c        **
**      In EFR                                                                **
**                                                                            **
**      Compute two samples a time                                            **
**                                                                            **
**      Total cycles = 257 (No Scaling)                                       **
**                   = 519 (One Scaling)                                      **
**                                                                            **
**      Register Usage:          A              B                             **
**                               11             9                             **
**                                                                            **
*******************************************************************************
       ; B4 -- &x[0]
       ; A4 -- &window[0]
       ; A6 -- &y[0]
       ; B8 -- L_WINDOW
       ; A0 -- sum and sum_e
       ; B0 -- sum_o
       ; B15 -- stack pointer


       ; notice that we use the latter approach in Example 5-13 to obtain the sum terms

       LDW    .D2    *B4++,B5      ; load x[0] & x[1]
||     LDW    .D1    *A4++,A5      ; load wind[0] & wind[1]
||     MVK    .S1    480,A6        ; reserve space for y[i]
||     SUB    .S2    B8,6,B1       ; LOOP I counter

       SUB    .L1X   B15,A6,A6     ; &y[0]
||     MVK    .S2    1,B7

       LDW    .D2    *B4++,B5      ; load x[2] & x[3]
||     LDW    .D1    *A4++,A5      ; load wind[2] & wind[3]

       SHL    .S2    B7,15,B7      ; 32768 or 0x8000L for rounding
||     MVK    .S1    -1,A10
||     ADD    .L2X   A6,2,B6       ; &y[1]
||     MV     .L1    A6,A3         ; &y[0]

       LDW    .D2    *B4++,B5      ; load x[4] & x[5]
||     LDW    .D1    *A4++,A5      ; load wind[4] & wind[5]
||     MVKLH  .S1    32767,A10     ; 7fffffff = MAX_32
||     MV     .L1X   B7,A7         ; 32768
```

Example 5–16. Final Assembly Code for Windowing and Scaling of autocorr.c (Continued)

```
       SMPYH   .M2X    B5,A5,B2        ; smpy(x[1],wind[1])
||     SMPY    .M1X    B5,A5,A2        ; smpy(x[0],wind[0])
||     B       .S2     LOOP I

       LDW     .D2     *B4++,B5        ; load x[6] & x[7]
||     LDW     .D1     *A4++,A5        ; load wind[6] & wind[7]
||     MVK     .S1     0,A0            ; sum_o = 0
||     MVK     .S2     0,B0            ; sum_e = 0

       SMPYH   .M2X    B5,A5,B2        ; smpy(x[3],wind[3])
||     SMPY    .M1X    B5,A5,A2        ; smpy(x[2],wind[2])
||     SADD    .L1     A2,A7,A2        ; sadd(smpy(x[1],wind[1]),0x8000L)
||     SADD    .L2     B2,B7,B2        ; sadd(smpy(x[0],wind[0]),0x8000L)
||     B       .S1     LOOP I

       LDW     .D2     *B4++,B5        ; load x[8] & x[9]
||     LDW     .D1     *A4++,A5        ; load wind[8] & wind[9]
||     SHR     .S1     A2,16,A9        ; y[1]=sadd(smpy(x[1],wind[1]),0x8000L)>>16
||     SHR     .S2     B2,16,B9        ; y[0]=sadd(smpy(x[0],wind[0])+0x8000L)>>16
||     SMPYH   .M1     A2,A2,A11       ; smpy(y[0],y[0])
||     SMPYH   .M2     B2,B2,B11       ; smpy(y[1],y[1])

LOOP I:

       STH     .D1     A9,*A6++[2]     ; store y[1]
||     STH     .D2     B9,*B6++[2]     ; store y[0]
||     SADD    .L1     A2,A7,A2        ; sadd(smpy(x[3],wind[3]),0x8000L)
||     SADD    .L2     B2,B7,B2        ; sadd(smpy(x[2],wind[2]),0x8000L)
||     SMPYH   .M2X    B5,A5,B2        ; smpy(x[5],wind[5])
||     SMPY    .M1X    B5,A5,A2        ; smpy(x[4],wind[4])
||[B1] SUB     .S2     B1,2,B1         ; decrement the loop counter
||[B1] B       .S1     LOOP I

       SADD    .L1     A0,A11,A0       ; sum_e += smpy(y[0],y[0])
||     SADD    .L2     B0,B11,B0       ; sum_o += smpy(y[1],y[1])
||     SMPYH   .M1     A2,A2,A11       ; smpy(y[2],y[2])
||     SMPYH   .M2     B2,B2,B11       ; smpy(y[3],y[3])
||     SHR     .S1     A2,16,A9        ; y[3]=sadd(smpy(x[3],wind[3]),0x8000L)>>16
||     SHR     .S2     B2,16,B9        ; y[2]=sadd(smpy(x[2],wind[2]),0x8000)>>16
||     LDW     .D2     *B4++,B5        ; load x[10] & x[11]
||     LDW     .D1     *A4++,A5        ; load wind[10] & wind[11]

       SADD    .L1X    A0,B0,A0        ; sum = sum_e + sum_o
||     MPY     .M2     B0,0,B0         ; overfl_shift = 0
                                       ; LOOP I completed
```

Example 5–16. Final Assembly Code for Windowing and Scaling of autocorr.c (Continued)

```
LTEST:

       CMPEQ  .L1    A0,A10,A1     ; if (sum == MAX_32)

 [!A1] B      .S1    FINISH        ; No, exit
||[A1] LDH    .D1    *A3,B5        ; load y[0]
||[A1] ADD    .L2X   A3,2,B9       ; &y[1]
||[A1] ADD    .D2    B0,4,B0       ; add (overfl_shift,4)

  [A1] LDH    .D2    *B9++,B5      ; load y[1]
||[A1] SUB    .S2    B8,7,B1       ; counter for LOOPII
||[A1] B      .S1    LOOP II
||[A1] MV     .L1    A3,A9         ; &y[0]
  [A1] LDH    .D2    *B9++,B5      ; load y[2]
||[A1] MVK    .S1    0,A0          ; sum = 0
||[A1] B      .S2    LOOP II

  [A1] LDH    .D2    *B9++,B5      ; load y[3]
||[A1] B      .S1    LOOP II
||[A1] MV     .L1    A0,A2         ; to take care of the initial condition

  [A1] LDH    .D2    *B9++,B5      ; load y[4]
||[A1] B      .S1    LOOP II

  [A1] LDH    .D2    *B9++,B5      ; load y[5]
||[A1] SHR    .S1X   B5,2,A5       ; y[0]=shr(y[0],2)
||[A1] B      .S2    LOOP II

LOOP II:

       LDH    .D2    *B9++,B5      ; load y[6]
||     SHR    .S1X   B5,2,A5       ; y[1] = shr(y[1],2)
||[B1] B      .S2    LOOP II       ; branch
||     STH    .D1    A5,*A9++      ; store y[0]
||[B1] ADD    .L2    B1,-1,B1      ; derement LOOPII counter
||     SMPY   .M1    A5,A5,A2      ; smpy(y[0],y[0])
||     SADD   .L1    A2,A0,A0      ; sum +=smpy(y[i],y[i])

       STH    .D1    A5,*A9++      ; store y[n-1]
||     SMPY   .M1    A5,A5,A2      ; smpy(y[n-1],y[n-1])
||     SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-3],y[n-3])
||     B      .S2    LTEST         ; branch back to LTEST

       SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-2],y[n-2])

       SADD   .L1    A2,A0,A0      ; sum +=smpy(y[n-1],y[n-1])

       NOP    3                    ; save the code size

FINISH:
```

If code size is not an issue, you can eliminate the last three NOPs by expanding the epilog of Loop II. This results in a 3-cycle count saving every time Loop II executes and two fetch packets (2 × 32 = 64 bytes) increase in code size.

5.2.3 Implementation of cor_h

cor_h is the second-most computationally intensive routine called to compute the matrix of autocorrelation, rr. The core part of cor_h is presented in Example 5–17, in the code search module.

Example 5–17. C Code cor_h

```
#define L_CODE  40

input:
        Word16 sign[L_CODE], h[L_CODE];


output:
        Word16 rr[L_CODE][L_CODE];



local variables/arrays:
        Word16 h2[L_CODE]; /* function of h, the impulse response of weighted
                              synthesis filter */
        Word16 dec, j, i, k;
        Word32 s;



Original C code
----------------------------------------------------------
        for (dec=1; dec<L_CODE; dec++)
        {
                s = 0;
                j = L_CODE-1;
                i = sub(j, dec);
                for (k=0; k<(L_CODE-dec); k++, i--, j--)
                {
                        s = L_mac(s, h2[k], h2[k+dec]);
                        rr[j][i] = mult(round(s), mult(sign[i],sign[j]));
                        rr[i][j] = rr[j][i];
                }
        }
----------------------------------------------------------
where sub(a,b) = _ssub(a<<16, b<<16)>>16
        L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        mult(a,b) = _smpy(a,b)>>16
and     round(a) = _sadd(a,0x8000L)>>16
```

The instructions to execute one iteration of the inner loop are listed in Example 5–18.

Example 5–18. Instructions to Execute One Inner Loop Iteration

```
INNERLOOP:
  LDH       .D      *h2ptr++,h2k           ;load h2[k]
  LDH       .D      *h2decptr++,h2deck     ;load h2[k+dec]
  SMPY      .M      h2k,h2deck,h2kk        ;smpy(h2[k],h2[k+dec])
  SADD      .L      s,h2kk,s               ;sadd(s,smpy(h2[k],h2[k+dec])
  SADD      .L      s,0x8000L,sround       ;round(s)<<16
  LDH       .D      *signiptr--,signi      ;load sign[i]
  LDH       .D      *signjptr--,signj      ;load sign[j]
  SMPY      .M      signi,signj,signij     ;smpy(sign[i],sign[j])=mult(sign[i],sign[j])<<16
  SMPYH     .M      signij,sround,rrji0    ;L_mult(round(s),mult(sign[i],sign[j]))
  SHR       .S      rrji0,16,rrji          ;rr[j][i]
  STH       .D      rrji,*rrjiptr--[41]    ;store rr[j] [i]
  STH       .D      rrji,*rrijptr--[41]    ;store rr[i] [j]
[icntr]  SUB.ALU    icntr,1,icntr          ;decrement inner loop counter
[icntr]  B  .S      INNERLOOP              ; branch to inner loop
```

In Example 5–18, *h2ptr* and *h2decptr* are the pointers for *h2*, pointing to *h2[k]* and *h2[k+dec]*. *signiptr* and *signjptr* are the pointers for *sign*, pointing to *sign[i]* and *sign[j]*. *rrjiptr* and *rrijptr* are the pointers for *rr*, pointing to *rr[i]*, *[i]* and *rr [i] [j]*, respectively.

There is no need to find *round(s)* and *mult(sign[i], sign[j])*, since you have the SMPYH instruction.

The .D unit (the unit used most often) is used six times in the inner loop. Ideally, these instructions can be arranged in three cycles. However, memory-bank hits occur with any combination of the load and/or store instruction.

Next, consider unrolling the inner loop once. The C code is shown in Example 5–19.

Example 5–19. C Code With Inner Loop Unrolling

```
for (dec=1; dec<L_CODE; dec++)
{
        s = 0;
        j = L_CODE-1;
        i = sub(j, dec);
        for (k=0; k<(L_CODE-dec); k+=2, i-=2, j-=2)
        {
                s = L_mac(s, h2[k], h2[k+dec]);
                rr[j][i] = mult(round(s), mult(sign[i],sign[j]));
                rr[i][j] = rr[j][i];
                s = L_mac(s, h2[k+1], h2[k+1+dec]);
                rr[j-1][i-1] = mult(round(s), mult(sign[i-1],sign[j-1]));
                rr[i-1][j-1] = rr[j-1][i-1];
        }
        if((dec&1)!=0) {
                s = L_mac(s,h2[L_CODE-dec-1],h2[L_CODE-1]);
                rr[dec][0] = mult(round(s),mult(sign[0],sign[dec]));
                rr[0][dec] = rr[dec][0];
        }
}
```

Notice that eight values must be loaded and four values must be stored in every iteration; however, *h2[k]* and *h2[k+1]* can be loaded in a word. The same is true for *sign[j]* and *sign[j–1]*. A total of six loadings are required. The inner loop instructions, then, are shown in Example 5–20.

Example 5–20. Inner Loop Instructions With Loop Unrolling

```
INNER LOOP:
  LD      .D   *h2ptr++,h2k_h2k+1           ;load h2[k] and h2[k+1]
  LDH     .D   *h2decptr++,h2deck           ;load h2[k+dec]
  SMPY    .M   h2k_h2k+1,h2deck,h2kk0       ;smpy(h2[k],h2[k+dec])
  SADD    .L   s,h2kk0,s                    ;sadd(s,smpy(h2[k],h2[k+dec])
  SADD    .L   s,0x8000L,sround             ;round(s)<<16
  LDH     .D   *signiptr--,signi            ;load sign[i]
  LDW     .D   *signjptr--,signj_signj-1    ;load sign[j]
  SMPYLH  .M   signi,signj_signj-1,signij0  ;smpy(sign[i],sign[j])
  SMPYH   .M   signij0,sround,rrji0         ;L_mult(round(s),mult(sign[i],sign[j]))
  SHR     .S   rrji0,16,rrji                ;rr[j][i]
  STH     .D   rrji,*rrjiptr--[82]          ;store rr[j] [i]
  STH     .D   rrji,*rrijptr--[82]          ;store rr[i] [j]
  LDH     .D   *h2decptr++,h2deck+1         ; load h2[k+1+dec]
  SMPYHL  .M   h2k_h2k+1,h2deck+1,h2kk1     ;smpy(h2[k+1],h2[k+1+dec])
  SADD    .L   s,h2kk1,s                    ;sadd(s,smpy(h2[k+1],h2[k+1+dec])
  SADD    .L   s,0x8000L,sround             ;round(s)<<16
  LDH     .D   *signiptr--,signi-1          ;load sign[i-1]
  SMPY    .M   signi-1,signj_signj-1,signij1;smpy(sign[i-1],sign[j-1])
  SMPYH   .M   signij1,sround,rrji1         ;L_mult(round(s),mult(sign[i-1],sign[j-1]))
  SHR     .S   rrji1,16,rrj1i1              ;rr[j-1][i-1]
  STH     .D   rrj1i1,*rrj1i1ptr--[82]      ;store rr[j-1] [i-1]
  STH     .D   rrj1i1,*rri1j1ptr--[82]      ;store rr[i-1] [j-1]
[icntr]SUB.ALUicntr,2,icntr                 ;decrement inner loop counter
[icntr]B  .S   INNERLOOP                    ; branch to loop
```

To avoid memory-bank hits:

- Load words *(h2[k], h2[k + 1])* and *(sign[i – 1], sign[i])* together and allocate *h2* and *sign* such that they are aligned with each other.
- Store *rr[j][i]* and *rr[j – 1][i – 1]* together and *rr[i][j]* and *rr[i – 1][j –1]* together.

There is a total of five loading/storing pairs, so that each iteration requires only five cycles. You gain speed by eliminating both the memory-bank hits, as well as by reducing the cycles required to complete each *rr*.

The final assembly code with reduced code size is shown in Example 5–21. Here, the primitive technique introduced in Chapter 4 is used to reduce the code size for both the prelog and epilog of the inner loop.

Example 5–21. Final Assembly Code With Reduced Code Size

```
*******************************************************************************
**     Texas Instruments, Inc                                                **
**                                                                           **
**     Implementation of cor_h in EFR                                        **
**                                                                           **
**     Compute four rrs at a time                                            **
**                                                                           **
**     Total cycles = 2533                                                   **
**                                                                           **
**     Register Usage:              A              B                         **
**                                                                           **
**                                 16              15                        **
**                                                                           **
*******************************************************************************

                                  ; A4 --- L_CODE
                                  ; B4 --- &h2[0]
                                  ; A6 --- &sign[0]
                                  ; B6 --- &rr[0][0]


       SUB    .L1    A4,1,A13      ; used to obtain &rr[i][j] and &rr[i-1][j-1]
||     ADDK   .S1    76,A6         ; &sign[L_CODE-2]
||     ADDK   .S2    3360,B6       ; &rr[L_CODE-1][L_CODE-2]+[82]=&rr[j][i]+[82]
||     SUB    .D1    A4,1,A2       ; outer loop counter

       MVK    .S2    0,B2          ; not doing the initial store
||     ADD    .L2    B4,2,B13      ; &h2[k+dec]
||     MVK    .S1    2,A11         ; used to increase/decrease the pointers
                                  ; for h2 and sign


OUTERLOOP:

       LDW    .D1    *A6,A10       ; load sign[j-1] & sign[j]
||     LDW    .D2    *B4,B12       ; load h2[k] & h2[k+1]
||     ADD    .L1X   B13,2,A3      ; &h2[k+dec+1]
||     SUB    .S1    A6,A11,A4     ; &sign[i-1]
||[A2] ADD    .L2X   A2,2,B0       ; define the inner loop counter
||     MPY    .M1    A13,A11,A3
||     MPY    .M2    B11,0,B11     ; initialize s

       LDH    .D2    *B13++[2],A7  ; load h2[k+dec]
||     LDH    .D1    *A3,B7        ; load h2[k+dec+1]
||     ADD    .L2X   A4,2,B9       ; &sign[i]
||     MV     .S2    B6,B14        ; &rr[j][i]+[82]
||     SUB    .L1    A6,4,A8       ; &sign[j-3]
||[B2] ADDK   .S1    -164,A14      ; from &rr[dec][0]+[82] to &rr[dec][0]
```

Example 5–21. Final Assembly Code With Reduced Code Size (Continued)

```
         LDH     .D2   *B9,A0         ; load sign[i]
||       LDH     .D1   *A4--[2],B5    ; load sign[i-1]
||       ADD     .L2   B4,4,B8        ; &h2[k+2]
||       ADD     .L1   A11,2,A11      ; update A11
||       ADDK    .S2   -82,B14        ; &rr[j][i]
||       MVK     .S1   3,A1           ; determine when the stores in the inner loop
                                      ; actually starts

 [B2]    STH     .D1   A12,*A14       ; store rr[dec][0]
||[B2]   ADDK    .S1   -164,A9        ; from &rr[0][dec]+[82] tp &rr[0][dec]
||       MV      .L1X  B6,A14         ; &rr[j][i]+[82]
||       SHR     .S2   B0,1,B0        ; inner loop counter
||       SUB     .L2X  B14,A3,B3      ; &rr[i-1][j-1]
||       SUB     .D2   B6,2,B6        ; &rr[j][i-1], for the next
                                      ; outer loop iteration
 [B0]    B       .S2   INNERLOOP
||[A2]   SUB     .S1   A2,1,A2        ; decrement outer loop counter
||[A2]   AND     .L2X  A2,1,B2        ; decide if the last store is needed
||[B2]   STH     .D1   A12,*A9        ; store rr[0][dec]
||       SUB     .L1   A14,A3,A9      ; &rr[i][j]+[82]
||[B0]   MV      .D2   B0,B1          ; counter for branching to outer loop

INNERLOOP:

         SHR     .S2   B9,16,B10      ; ## obtain rr[j-1][i-1]
||       SMPYH   .M1   A3,A0,A3       ; # smpyh(sadd(s,0x8000L),smpy(sign[i],sign[j]))
||       SADD    .L2X  B11,A15,B9     ; # sadd(s,0x8000L)

||       LDW     .D1   *A8--,A10      ; *load sign[j] & sign[j-1]
||       LDW     .D2   *B8++,B12      ; *load h2[k] & h2[k+1]
||[!B1]  B       .S1   OUTERLOOP      ; outer LOOP
||       ADD     .L1X  B13,2,A3       ; &h2[k+dec+1]

         LDH     .D1   *A3,B7         ; *h2[k+dec+1]
||       LDH     .D2   *B13++[2],A7   ; *h2[k+dec]
||       SMPYH   .M2   B9,B5,B9       ; # smpyh(sadd(s,0x8000L),smpy(sign[i-1],sign[j-1]))
||       SMPY    .M1X  A7,B12,A7      ; smpy(h2[k],h2[k+dec]
||       SUB     .S2   B1,1,B1        ; decrement the counter for branching to the outer loop
||[A1]   SUB     .L1   A1,1,A1        ; decrement the inner loop
||       ADD     .L2X  A4,2,B9        ; &sign[i]

         LDH     .D2   *B9,A0         ; *load sign[i]
||       LDH     .D1   *A4--[2],B5    ; *load sign[i-1]
||       SMPYHL  .M1   A10,A0,A0      ; smpy(sign[j],sign[i])
||       SMPYLH  .M2   B7,B12,B7      ; smpy(h2[k+1],h2[k+1+dec])
||[!A1]  ADDK    .S1   -164,A14       ; ## from &rr[j][i]+[82] to &rr[j][i]
||[!A1]  ADDK    .S2   -164,B14       ; ## from &rr[j-1][i-1]+[82] to &rr[j-1][i-1]
```

Example 5–21. Final Assembly Code With Reduced Code Size (Continued)

```
 [!A1]   STH    .D1    A12,*A14      ; ## store rr[j][i]
||[!A1]  STH    .D2    B10,*B14      ; ## store rr[j-1][i-1]
||       SADD   .L1X   B11,A7,A5     ; s = sadd(s,smpy(h2[k],h2[k+dec])
||       SMPY   .M2X   A10,B5,B5     ; smpy(sign[i-1],sign[j-1]
||[B0]   SUB    .L2    B0,1,B0       ; decrement inner loop counter
||[!A1]  ADDK   .S1    -164,A9       ; ## from &rr[i][j]+[82] to rr[i][j]
||[!A1]  ADDK   .S2    -164,B3       ; ## from &rr[i-1][j-1]+[82] to &rr[i-1][j-1]

[!A1]   STH    .D1    A12,*A9       ; ## store rr[i][j]
||[!A1]  STH    .D2    B10,*B3       ; ## store rr[j-1][i-1]
||       SHR    .S1    A3,16,A12     ; # obtain rr[j-1][i-1]
||       SADD   .L1    A5,A15,A3     ; sadd(s,0x8000L)
||       SADD   .L2X   A5,B7,B11     ; s = sadd(s,smpy(h2[k+1],h2[k+dec+1]
||[B0]   B      .S2    INNERLOOP     ; end of INNERLOOP

        ADD    .L2X   B4,A11,B13     ; &h2[k+dec]
||[!A2]  B      .S2    FINISH        ; exit

FINISH:

        NOP    5
```

The value of *s* is represented by both B11 and A5 to avoid two .L1 or two .L2 units occurring in the same execute packet. Due to the dependence on *s* as well as the removal of memory-bank hits, it takes 20 cycles for each iteration of the modified C code. The number of #s denotes that this instruction is not executed or that the result of this instruction is not useful until # number of iterations: each time the outer loop enters the inner loop.

The code size is 11 fetch packets (352 bytes). Without applying the primitive technique, the code size will be at least four fetch packets more than the code shown in Example 5–21.

You can squeeze the instruction

```
ADD    .L2X      B4,A11,B13   ; &h2[k+dec]
```

into the inner loop to save about 1.5% of the cycle counts, with a one fetch packet increase in program memory.

5.2.4 Implementation of the rrv Computation In search_10i40

Example 5–22 shows the implementation of the *rrv* computation in search_10i40.

Example 5–22. C Code of the rrv Computation In Search_10i40

```
#define L_CODE 40
#define STEP 5
#define _1_16 (Word16)(32768L/16)
#define _1_8  (Word16)(32768L/8)

input:
       Word16 rr[L_CODE][L_CODE], ipos[L_CODE];

local variables/arrays:
       Word16 rrv[L_CODE];
       Word16 i0,i1,i2,i3,i4,i5,i6,i7,i8,i9; /* defined on [0,L_CODE-1] */
       Word32 s;


           (The values of i0, i1, i2, i3, i4, i5, i6, and i7 were obtained before entering this loop.)

Original C code
-------------------------------------------------------
       for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
       {
              s = L_mult (rr[i9][i9], _1_16);
              s = L_mac (s, rr[i0][i9], _1_8);
              s = L_mac (s, rr[i1][i9], _1_8);
              s = L_mac (s, rr[i2][i9], _1_8);
              s = L_mac (s, rr[i3][i9], _1_8);
              s = L_mac (s, rr[i4][i9], _1_8);
              s = L_mac (s, rr[i5][i9], _1_8);
              s = L_mac (s, rr[i6][i9], _1_8);
              s = L_mac (s, rr[i7][i9], _1_8);
              rrv[i9] = round (s);
       }
-------------------------------------------------------
       where  L_mult(a,b) = _smpy(a,b)
              L_mac(a,b,c) = _sadd(a,_smpy(b,c))
       and    round(a) = _sadd(a,0x8000L)>>16
```

The instructions for one loop iteration are shown in Example 5–23.

Example 5–23. Instructions for One Loop Iteration

```
LOOP:
       LDH    .D     *rr9ptr++[205],rr99 ;load rr[i9][i9]
       SMPY   .M     rr99,_1_16,s        ;s=L_mult(rr[i9][i9],_1_16)
       LDH    .D     *rr0ptr++[5],rr09   ;load rr[i0][i9]
       SMPY   .M     rr09,_1_8,s0        ;L_mult(rr[i0][i9],_1_8)
       SADD   .L     s,s0,s              ;s=L_mac(s,rr[i0][i9],_1_8)
       LDH    .D     *rr1ptr++[5],rr19   ;load rr[i1][i9]
       SMPY   .M     rr19,_1_8,s1        ;L_mult(rr[i1][i9],_1_8)
       SADD   .L     s,s1,s              ;s=L_mac(s,rr[i1][i9],_1_8)
       LDH    .D     *rr2ptr++[5],rr29   ;load rr[i2][i9]
       SMPY   .M     rr29,_1_8,s2        ;L_mult(rr[i2][i9],_1_8)
       SADD   .L     s,s2,s              ;s=L_mac(s,rr[i2][i9],_1_8)
       LDH    .D     *rr3ptr++[5],rr39   ;load rr[i3][i9]
       SMPY   .M     rr39,_1_8,s3        ;L_mult(rr[i3][i9],_1_8)
       SADD   .L     s,s3,s              ;s=L_mac(s,rr[i3][i9],_1_8)
       LDH    .D     *rr4ptr++[5],rr49   ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,s4        ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     s,s4,s              ;s=L_mac(s,rr[i4][i9],_1_8)
       LDH    .D     *rr5ptr++[5],rr59   ;load rr[i5][i9]
       SMPY   .M     rr59,_1_8,s5        ;L_mult(rr[i5][i9],_1_8)
       SADD   .L     s,s5,s              ;s=L_mac(s,rr[i5][i9],_1_8)
       LDH    .D     *rr6ptr++[5],rr69   ;load rr[i6][i9]
       SMPY   .M     rr69,_1_8,s6        ;L_mult(rr[i6][i9],_1_8)
       SADD   .L     s,s6,s              ;s=L_mac(s,rr[i6][i9],_1_8)
       LDH    .D     *rr7ptr++[5],rr79   ;load rr[i7][i9]
       SMPY   .M     rr79,_1_8,s7        ;L_mult(rr[i7][i9],_1_8)
       SADD   .L     s,s7,s              ;s=L_mac(s,rr[i7][i9],_1_8)
       SADD   .L     s,0x8000L,sround    ;round(s)
       SHR    .S     sround,16,rrv9      ;rrv[i9]
       STH    .D     rrv9,*rrv9ptr++[5]  ;store rrv[i9]
[icntr]SUB    .ALU   icntr,1,icntr       ;decrement inner loop counter
[icntr]B      .S     INNERLOOP           ;branch to loop
```

In Example 5–23, *rrqptr*, *rroptr*, *rriptr*, *rraptr*, *rr3ptr*, *rr4ptr*, *rr5ptr*, *rr6ptr*, *rruptr*, and *rrvqptr* are the pointers for *rr[i9] [i9]*, *rr[i0] [i9]*, *rr[i1] [i9]*, *rr[i2] [i9]*, *rr[i3] [i9]*, *rr[i4] [i9]*, *44[i5] [i9]*, *rr[i6] [i9]*, *rr[i7] [i9]*, and *rrv[i9]*.

The .D unit (the unit used the most) is used ten times per iteration. Although these instructions can be arranged in five cycles, any combination of the load hits the same memory bank, since any two values loaded are exactly 40 half-words apart; it still takes ten cycles for one *rrv*.

Next, consider unrolling the inner loop once. The C code becomes as shown in Example 5–24.

Example 5–24. C Code for Unrolled Loop

```
for (i9 = ipos[9]; i9 < L_CODE; i9 += 2*STEP)
{
       s = L_mult (rr[i9][i9], _1_16);
       S = L_mult (rr[i9+5][i9+5], _1_16);
       s = L_mac (s, rr[i0][i9], _1_8);
       S = L_mac (S, rr[i0][i9+5], _1_8);
       s = L_mac (s, rr[i1][i9], _1_8);
       S = L_mac (S, rr[i1][i9+5], _1_8);
       s = L_mac (s, rr[i2][i9], _1_8);
       S = L_mac (S, rr[i2][i9+5], _1_8);
       s = L_mac (s, rr[i3][i9], _1_8);
       S = L_mac (S, rr[i3][i9+5], _1_8);
       s = L_mac (s, rr[i4][i9], _1_8);
       S = L_mac (S, rr[i4][i9+5], _1_8);
       s = L_mac (s, rr[i5][i9], _1_8);
       S = L_mac (S, rr[i5][i9+5], _1_8);
       s = L_mac (s, rr[i6][i9], _1_8);
       S = L_mac (S, rr[i6][i9+5], _1_8);
       s = L_mac (s, rr[i7][i9], _1_8);
       S = L_mac (S, rr[i7][i9+5], _1_8);
       rrv[i9] = round (s);
       rrv[i9+5] = round (S);
}
```

Example 5–25 shows the instructions for each iteration.

Example 5–25. Instructions for One Iteration of the Loop

```
LOOP:
       LDH    .D     *rr9ptr++[410],rr99    ;load rr[i9][i9]
       SMPY   .M     rr99,_1_16,s           ;s=L_mult(rr[i9][i9],_1_16)
       LDH    .D     *rr95ptr++[410],rr995  ;load rr[i9+5][i9+5]
       SMPY   .M     rr995,_1_16,S          ;S=L_mult(rr[i9+5][i9+5],_1_16)
       LDH    .D     *rr0ptr++[10],rr09     ;load rr[i0][i9]
       SMPY   .M     rr09,_1_8,s0           ;L_mult(rr[i0][i9],_1_8)
       SADD   .L     s,s0,s                 ;s=L_mac(s,rr[i0][i9],_1_8)
       LDH    .D     *rr05ptr++[10],rr095   ;load rr[i0][i9+5]
       SMPY   .M     rr095,_1_8,S0          ;L_mult(rr[i0][i9+5],_1_8)
       SADD   .L     S,S0,S                 ;S=L_mac(S,rr[i0][i9+5],_1_8)
       LDH    .D     *rr1ptr++[10],rr19     ;load rr[i1][i9]
       SMPY   .M     rr19,_1_8,s1           ;L_mult(rr[i1][i9],_1_8)
       SADD   .L     s,s1,s                 ;s=L_mac(s,rr[i1][i9],_1_8)
       LDH    .D     *rr15ptr++[10],rr195   ;load rr[i1][i9+5]
       SMPY   .M     rr195,_1_8,S1          ;L_mult(rr[i1][i9+5],_1_8)
       SADD   .L     S,S1,S                 ;S=L_mac(S,rr[i1][i9+5],_1_8)
       LDH    .D     *rr2ptr++[10],rr29     ;load rr[i2][i9]
       SMPY   .M     rr29,_1_8,s2           ;L_mult(rr[i2][i9],_1_8)
       SADD   .L     s,s2,s                 ;s=L_mac(s,rr[i2][i9],_1_8)
       LDH    .D     *rr2ptr++[10],rr295    ;load rr[i2][i9+5]
       SMPY   .M     rr295,_1_8,S2          ;L_mult(rr[i2][i9+5],_1_8)
       SADD   .L     S,S2,S                 ;S=L_mac(S,rr[i2][i9+5],_1_8)
       LDH    .D     *rr3ptr++[10],rr39     ;load rr[i3][i9]
       SMPY   .M     rr39,_1_8,s3           ;L_mult(rr[i3][i9],_1_8)
       SADD   .L     s,s3,s                 ;s=L_mac(s,rr[i3][i9],_1_8)
       LDH    .D     *rr3ptr++[10],rr395    ;load rr[i3][i9+5]
       SMPY   .M     rr395,_1_8,S3          ;L_mult(rr[i3][i9+5],_1_8)
       SADD   .L     S,S3,S                 ;S=L_mac(S,rr[i3][i9+5],_1_8)
       LDH    .D     *rr4ptr++[10],rr49     ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,s4           ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     s,s4,s                 ;s=L_mac(s,rr[i4][i9],_1_8)
       LDH    .D     *rr4ptr++[10],rr49     ;load rr[i4][i9]
       SMPY   .M     rr49,_1_8,s4           ;L_mult(rr[i4][i9],_1_8)
       SADD   .L     s,s4,s                 ;S=L_mac(S,rr[i4][i9],_1_8)
       LDH    .D     *rr5ptr++[10],rr59     ;load rr[i5][i9]
       SMPY   .M     rr59,_1_8,s5           ;L_mult(rr[i5][i9],_1_8)
       SADD   .L     s,s5,s                 ;s=L_mac(s,rr[i5][i9],_1_8)
       LDH    .D     *rr5ptr++[10],rr595    ;load rr[i5][i9+5]
       SMPY   .M     rr595,_1_8,S5          ;L_mult(rr[i5][i9+5],_1_8)
       SADD   .L     S,S5,S                 ;S=L_mac(S,rr[i5][i9+5],_1_8)
       LDH    .D     *rr6ptr++[10],rr69     ;load rr[i6][i9]
       SMPY   .M     rr69,_1_8,s6           ;L_mult(rr[i6][i9],_1_8)
       SADD   .L     s,s6,s                 ;s=L_mac(s,rr[i6][i9],_1_8)
       LDH    .D     *rr6ptr++[10],rr695    ;load rr[i6][i9+5]
       SMPY   .M     rr695,_1_8,S6          ;L_mult(rr[i6][i9+5],_1_8)
       SADD   .L     S,S6,S                 ;S=L_mac(S,rr[i6][i9+5],_1_8)
       LDH    .D     *rr7ptr++[10],rr79     ;load rr[i7][i9]
       SMPY   .M     rr79,_1_8,s7           ;L_mult(rr[i7][i9],_1_8)
```

Example 5–25. Instructions for One Loop Iteration (Continued)

```
        SADD    .L      s,s7,s                  ;s=L_mac(s,rr[i7][i9],_1_8)
        LDH     .D      *rr7ptr++[10],rr795     ;load rr[i7][i9+5]
        SMPY    .M      rr795,_1_8,S7           ;L_mult(rr[i7][i9+5],_1_8)
        SADD    .L      S,S7,S                  ;S=L_mac(S,rr[i7][i9+5],_1_8)
        SADD    .L      s,0x8000L,sround        ;round(s)
        SHR     .S      sround,16,rrv9          ;rrv[i9]
        STH     .D      rrv9,*rrv9ptr++[10]     ;store rrv[i9]
        SADD    .L      S,0x8000L,Sround        ;round(S)
        SHR     .S      Sround,16,rrv95         ;rrv[i9+5]
        STH     .D      rrv95,*rrv95ptr++[10]   ;store rrv[i9+5]
[icntr]SUB    .ALU    icntr,2,icntr           ;decrement inner loop counter
[icntr]B      .S      INNERLOOP               ;branch to loop
```

In Example 5–25, *rrqptr* and *rrqsptr* are the pointers for *rr[i9] [i9]* and *rr[i9+5] [i9+5]*. *rrixptr* and *rrixsptr* are the pointers for *rr[ix] [i9]* and *rr[ix] [i9+5]*, (where *ix* = i0, i1, …, i7). *rrvqptr* and *rrvqsptr* are the pointers for *rrv[i9]* and *rrv[i9+5]*, respectively. Again, .D is the unit used the most (twenty times per iteration).

Notice that any pairs of *rr[ix] [i9]*, *rr[iy] [i9+5]* never hit the same memory bank. The same is true for pairs *rrv[i9*], *rrv[iq+5]*, as well as for *rr9, rrv[i9+5]*. For ease of understanding:

- Load *rr[ix] [i9], rr[ix] [i9+5]* together
- Load *rr[i9] [i9], rr[i9+5] [i9+5]* together
- Store *rrv[i9], rrv[i9+5]* together

In this way, each iteration takes ten cycles, without any memory bank hits. You double the speed by unrolling the loop once.

The final assembly code is shown in Example 5–26.

Example 5–26. Final Assembly Code of rrr Computation

```
*******************************************************************************
**     Texas Instruments, Inc                                                **
**                                                                           **
**     Implementation of the rrv Computation in search_10i40 in EFR          **
**                                                                           **
**     Compute two rrvs a time                                               **
**                                                                           **
**     Total cycles = 55                                                     **
**                                                                           **
**     Register Usage:               A              B                        **
**                                                                           **
**                                   16             14                       **
**                                                                           **
*******************************************************************************
                            ; B4 --- i0
                            ; B5 --- i1
                            ; B6 --- i2
                            ; B7 --- i3
                            ; A8 --- i4
                            ; B9 --- i5
                            ; A10 -- i6
                            ; A11 -- i7
                            ; B3 --- i9
                            ; A15 --- &rr[0][0]
                            ; A0 --- &rrv[0]
                            ; B14 --- stack pointer

       MVK    .S1    410,A2              ; offset of rr[i9][i9]
||     MVK    .S2    410,B2              ; offset of rr[i9+5][i9+5]

       MVK    .S2    82,B0

       MPYU   .M2    B3,B0,B3            ; [i9][i9]
||     SHL    .S1X   B3,1,A13
||     SUB    .L2    B0,2,B0             ; 80
||     ADD    .S2X   A15,B2,B13          ; &rr[5][5]

       MPYU   .M2    B4,B0,B4            ; [i0][0]
||     ADD    .L2X   A15,10,B15          ; &rr[0][5]
||     MVK    .S1    80,A1

       MPYU   .M2    B5,B0,B5            ; [i1][0]
||     ADD    .L1X   B3,A15,A3           ; &rr[i9][i9]
||     ADD    .L2    B3,B13,B3           ; &rr[i9+5][i9+5]
||     ADD    .S1    A15,A13,A15         ; &rr[0][i9]
||     ADD    .S2X   B15,A13,B15         ; &rr[0][i9+5]
||     MPYU   .M1    A10,A1,A10          ; [i6][0]
```

Example 5–26. Final Assembly Code of rrr Computation (Continued)

```
        MPYU    .M2     B6,B0,B6            ; [i2][0]
||      MPYU    .M1     A11,A1,A11          ; [i7][0]
||      ADD     .L1X    B4,A15,A4           ; &rr[i0][i9]
||      ADD     .L2     B4,B15,B4           ; &rr[i0][i9+5]
||      LDH     .D1     *A3++[A2],A13       ; load rr[i9][i9]
||      LDH     .D2     *B3++[B2],B13       ; load rr[i9+5][i9+5]
||      ADD     .S1     A0,A13,A0           ; &rrv[i9]

        MPYU    .M2     B7,B0,B7            ; [i3][0]
||      MPYU    .M1     A8,A1,A8            ; [i4][0]
||      ADD     .L1X    B5,A15,A5           ; &rr[i1][i9]
||      ADD     .L2     B5,B15,B5           ; &rr[i1][i9+5]
||      ADD     .S1     A10,A15,A10         ; &rr[i6][9]
||      ADD     .S2X    A10,B15,B10         ; &rr[i6][i9+5]
||      LDH     .D1     *A4++[10],A13       ; load rr[i0][i9]
||      LDH     .D2     *B4++[10],B13       ; load rr[i0][i9+5]

        MPYU    .M2     B9,B0,B9            ; [i9][0]
||      ADD     .L1X    B6,A15,A6           ; &rr[i2][i9]
||      ADD     .L2     B6,B15,B6           ; &rr[i2][i9+5]
||      ADD     .S1     A11,A15,A11         ; &rr[i7][i9]
||      ADD     .S2X    A11,B15,B11         ; &rr[i7][i9+5]
||      LDH     .D1     *A5++[10],A13       ; load rr[i1][i9]
||      LDH     .D2     *B5++[10],B13       ; load rr[i1][i9+5]

        ADD     .L1X    B7,A15,A7           ; &rr[i3][i9]
||      ADD     .L2     B7,B15,B7           ; &rr[i3][i9+5]
||      ADD     .S1     A8,A15,A8           ; &rr[i4][i9]
||      ADD     .S2X    A8,B15,B8           ; &rr[i4][i9+5]
||      LDH     .D1     *A6++[10],A13       ; load rr[i2][i9]
||      LDH     .D2     *B6++[10],B13       ; load rr[i2][i9+5]

        ADD     .L1X    B9,A15,A9           ; &rr[i5][i9]
||      ADD     .L2     B9,B15,B9           ; &rr[i5][i9+5]
||      LDH     .D1     *A7++[10],A13       ; load rr[i3][i9]
||      LDH     .D2     *B7,B13             ; load rr[i3][i9+5]
||      MVK     .S2     2048,B7             ; _1_16

        LDH     .D1     *A8++[10],A13       ; load rr[i4][i9]
||      LDH     .D2     *B8++[10],B13       ; load rr[i4][i9+5]
||      SMPY    .M1X    A13,B7,A12          ; s=smpy(rr[i9][i9],_1_16)
||      SMPY    .M2     B13,B7,B12          ; S=smpy(rr[i9+5][i9+5],_1_16)
||      SHL     .S2     B7,1,B7             ; _1_8
||      ADD     .L2X    A0,10,B0            ; &rrv[i9+5]

        LDH     .D1     *A9++[10],A13       ; load rr[i5][i9]
||      LDH     .D2     *B9++[10],B13       ; load rr[i5][i9+5]
||      SMPY    .M1X    A13,B7,A15          ; s0=smpy(rr[i0][i9],_1_8)
||      SMPY    .M2     B13,B7,B15          ; S0=smpy(rr[i0][i9+5],_1_8)
```

Example 5–26. Final Assembly Code of rrr Computation (Continued)

```
        LDH     .D1     *A10++[10],A13          ; load rr[i6][i9]
||      LDH     .D2     *B10++[10],B13          ; load rr[i6][i9+5]
||      SMPY    .M1X    A13,B7,A15              ; s1=smpy(rr[i1][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S1=smpy(rr[i1][i9+5],_1_8)

        LDH     .D1     *A11++[10],A13          ; load rr[i7][i9]
||      LDH     .D2     *B11++[10],B13          ; load rr[i7][i9+5]
||      SMPY    .M1X    A13,B7,A15              ; s2=smpy(rr[i2][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S2=smpy(rr[i2][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12             ; s=sadd(s,s0)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S0)
||      MVK     .S1     3,A1                    ; loop counter

        SMPY    .M1X    A13,B7,A15              ; s3=smpy(rr[i3][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S3=smpy(rr[i3][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12             ; s=sadd(s,s1)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S1)
||      MVK     .S1     32767,A14

LOOP:
        SADD    .L1     A12,A15,A12             ; s=sadd(s,s2)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S2)
||      SMPY    .M1X    A13,B7,A15              ; s4=smpy(rr[i4][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S4=smpy(rr[i4][i9+5],_1_8)
||      LDH     .D1     *A3++[A2],A13           ;* load rr[i9][i9]
||      LDH     .D2     *B3++[B2],B13           ;* load rr[i9+5][i9+5]
||      ADD     .S1     A14,1,A14               ; 32768 for rounding

        SMPY    .M1X    A13,B7,A15              ; s5=smpy(rr[i5][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S5=smpy(rr[i5][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12             ; s=sadd(s,s3)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S3)
||      LDH     .D1     *A4++[10],A13           ;* load rr[i0][i9]
||      LDH     .D2     *B4,B13                 ;* load rr[i0][i9+5]

        SMPY    .M1X    A13,B7,A15              ; s6=smpy(rr[i6][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S6=smpy(rr[i6][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12             ; s=sadd(s,s4)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S4)
||      LDH     .D1     *A5++[10],A13           ;* load rr[i1][i9]
||      LDH     .D2     *B5++[10],B13           ;* load rr[i1][i9+5]

        SMPY    .M1X    A13,B7,A15              ; s7=smpy(rr[i7][i9],_1_8)
||      SMPY    .M2     B13,B7,B15              ; S7=smpy(rr[i7][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12             ; s=sadd(s,s5)
||      SADD    .L2     B12,B15,B12             ; S=sadd(S,S5)
||      LDH     .D1     *A6++[10],A13           ;* load rr[i2][i9]
||      LDH     .D2     *B6++[10],B13           ;* load rr[i2][i9+5]
||      ADD     .S2X    A7,10,B7                ; &rr[i3][i9+5]
```

Example 5–26. Final Assembly Code of rrr Computation (Continued)

```
        SADD    .L1     A12,A15,A12          ; s=sadd(s,s6)
||      SADD    .L2     B12,B15,B12          ; S=sadd(S,S6)
||      LDH     .D1     *A7++[10],A13        ;* load rr[i3][i9]
||      LDH     .D2     *B7,B13              ;* load rr[i3][i9+5]
||      MVK     .S2     2048,B7              ; _1_16
||[A1]  B       .S1     LOOP                 ; branch to the loop

        SADD    .L1     A12,A15,A12          ; s=sadd(s,s7)
||      SADD    .L2     B12,B15,B12          ; S=sadd(S,S7)
||      LDH     .D1     *A8++[10],A13        ;* load rr[i4][i9]
||      LDH     .D2     *B8++[10],B13        ;* load rr[i4][i9+5]
||      SMPY    .M1X    A13,B7,A12           ;* s=smpy(rr[i9][i9],_1_16)
||      SMPY    .M2     B13,B7,B12           ;* S=smpy(rr[i9+5][i9+5],_1_16)
||      SHL     .S2     B7,1,B7              ; _1_8
||[A1]  SUB     .S1     A1,1,A1              ; decrement loop counter

        SADD    .L1     A12,A14,A14          ; round(s)
||      SADD    .L2X    B12,A14,B4           ; round(S)
||      LDH     .D1     *A9++[10],A13        ;* load rr[i5][i9]
||      LDH     .D2     *B9++[10],B13        ;* load rr[i5][i9+5]
||      SMPY    .M1X    A13,B7,A15           ;* s0=smpy(rr[i0][i9],_1_8)
||      SMPY    .M2     B13,B7,B15           ;* S0=smpy(rr[i0][i9+5],_1_8)

        SHR     .S1     A14,16,A14           ; rrv[i9]
||      SHR     .S2     B4,16,B4             ; rrv[i9+5]
||      SMPY    .M1X    A13,B7,A15           ;* s1=smpy(rr[i1][i9],_1_8)
||      SMPY    .M2     B13,B7,B15           ;* S1=smpy(rr[i1][i9+5],_1_8)
||      LDH     .D1     *A10++[10],A13       ;* load rr[i6][i9]
||      LDH     .D2     *B10++[10],B13       ;* load rr[i6][i9+5]

        SMPY    .M1X    A13,B7,A15           ;* s2=smpy(rr[i2][i9],_1_8)
||      SMPY    .M2     B13,B7,B15           ;* S2=smpy(rr[i2][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12          ;* s=sadd(s,s0)
||      SADD    .L2     B12,B15,B12          ;* S=sadd(S,S0)
||      LDH     .D1     *A11++[10],A13       ;* load rr[i7][i9]
||      LDH     .D2     *B11++[10],B13       ;* load rr[i7][i9+5]

        STH     .D1     A14,*A0++[10]        ; store rrv[i9]
||      STH     .D2     B4,*B0++[10]         ; store rrv[i9+5]
||      SMPY    .M1X    A13,B7,A15           ;* s3=smpy(rr[i3][i9],_1_8)
||      SMPY    .M2     B13,B7,B15           ;* S3=smpy(rr[i3][i9+5],_1_8)
||      SADD    .L1     A12,A15,A12          ;* s=sadd(s,s1)
||      SADD    .L2     B12,B15,B12          ;* S=sadd(S,S1)
||      ADD     .S2X    A4,10,B4             ;* &rr[i0][i9+5]
||      MVK     .S1     32767,A14            ; end of LOOPX
```

Notice that because of the shortage of registers:

- *B7* serves as *_1_16*, *_1_8* and as the pointer for *rr[i3][i9+5]*
- *B4* is both the value of *rrv[i9+5]* and the pointer of *rr[i0][i9+5]*
- *A14* represents *0x8000L* as well as *rrv[i9]*

The last iteration of the loop could be expanded as the epilog of the loop to overlap with the prolog of the code following this part of the code.

5.2.5 Implementation of the Index Search In search_10i40

The index search in search_10i40 is the core of search_10i40. The C code is shown in Example 5–27.

Example 5–27. Index Search for search_10i40 C Code

```
#define L_CODE 40
#define STEP   5
#define _1_16  (Word16)(32768L/16)
#define _1_8   (Word16)(32768L/8)

input:
       Word16 rr[L_CODE][L_CODE], ipos[L_CODE], dn[L_CODE];

local variables/arrays:
       Word16 rrv[L_CODE];
       Word16 i0,i1,i2,i3,i4,i5,i6,i7,i8,i9; /* defined on [0,L_CODE-1] */
       Word16 ia,ib;
       Word16 ps,ps0,ps1,ps2,sq,sq2;
       Word16 alp,alp_16;
       Word32 s,alp0,alp1,alp2;

(Notice that the values of i0, i1, i2, i3, i4, i5, i6, i7 , ps0, and alp0 have
been obtained before entering this loop.)

Original C code
-------------------------------------------------------
       sq = -1;
       alp = 1;
       ps = 0;
       ia = ipos[8];
       ib = ipos[9];

       /* initialize 10 indices for i8 loop (see i2-i3 loop) */
       for (i8 = ipos[8]; i8 < L_CODE; i8 += STEP)
       {
           ps1 = add (ps0, dn[i8]);
```

Example 5–27. Index Search for search_10i40 C Code (Continued)

```
            alp1 = L_mac (alp0, rr[i8][i8], _1_128);
            alp1 = L_mac (alp1, rr[i0][i8], _1_64);
            alp1 = L_mac (alp1, rr[i1][i8], _1_64);
            alp1 = L_mac (alp1, rr[i2][i8], _1_64);
            alp1 = L_mac (alp1, rr[i3][i8], _1_64);
            alp1 = L_mac (alp1, rr[i4][i8], _1_64);
            alp1 = L_mac (alp1, rr[i5][i8], _1_64);
            alp1 = L_mac (alp1, rr[i6][i8], _1_64);
            alp1 = L_mac (alp1, rr[i7][i8], _1_64);

             /* initialize 3 indices for i9 inner loop (see i2-i3 loop) */

            for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
            {
               ps2 = add (ps1, dn[i9]);

               alp2 = L_mac (alp1, rrv[i9], _1_8);
               alp2 = L_mac (alp2, rr[i8][i9], _1_64);

               sq2 = mult (ps2, ps2);

               alp_16 = round (alp2);

               s = L_msu (L_mult (alp, sq2), sq, alp_16);

               if (s > 0)
       {
                    sq = sq2;
                    ps = ps2;
                    alp = alp_16;
                    ia = i8;
                    ib = i9;
            }
            }
       }
-------------------------------------------------------
  where add(a,b) = _sadd(a<<16,b<<16)>>16
        L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        mult(a,b) = _smpy(a<<16,b<<16)>>16
        L_mult(a,b)=_smpy(a,b)
        round(a) = _sadd(a,0x8000L)>>16
  and   L_msu(a,b,c)=_ssub(a,_smpy(b,c))
```

This is a typical example of the performance being limited by data dependency constraints. In this case, the dependency is on the values of *alp* and *sq*.

5.2.5.1 Rearranging the C Code

To avoid the unnecessary shift, *ps*, *ps1*, *ps2*, *alp*, *alp_16*, *sq*, and *sq2* are implemented as Word32 variables. The effected calculations are:

Original	Implemented as
`ps1 = add (ps0, dn[i8]);`	`ps1 = sadd(ps0, dn[i8]<<16);`
`ps2 = add (ps1, dn[i9]);`	`ps2 = sadd(ps1, dn[i9]<<16);`
`sq2 = mult (ps2, ps2);`	`sq2 = smpyh(ps2,ps2);`
`alp_16 = round(alp2);`	`alp_16 = sadd(alp2,0x8000L);`

There is no need to compute *s* explicitly. Instead of implementing the following:

```
s = L_msu (L_mult (alp, sq2), sq, alp_16);

if (s > 0)
{
   sq = sq2;
   ps = ps2;
   alp = alp_16;
   ia = i8;
   ib = i9;
   }
```

you can do the following to fulfil the same task:

```
if(_smpyh(alp,sq2) > _smpyh(sq,alp_16)) {
  sq = sq2;
  ps = ps2;
  alp = alp_16;
  ia = i8;
  ib = i9;
}.
```

5.2.5.2 Performance analysis

The instructions to execute one iteration of the inner loop are shown in Example 5–28.

Example 5–28. Inner Loop Instructions

```
INNERLOOP:
        LDH   .D     *dn9ptr++[5],dn9      ; load dn[i9]
        SHL   .S     dn9,16,dn9h           ; dn[i9] << 16
        SADD  .L     ps1,dn9h,ps2          ; ps2 = sadd(ps1, dn[i9] << 16)
        SMPYH .M     ps2,ps2,sq2           ; sq2 = smpyh(ps2,ps2)
        LDH   .D     *rrvptr++[5],rrv      ; load rrv[i9]
        SMPY  .M     rrv,_1_8,tmp1         ; smpy(rrv[i9], _1_8)
        SADD  .L     alp1,temp1,alp2       ; alp2=sadd(alp1,smpy(rrv[i9],_1_8))
        LDH   .D     *rr89prt++,rr89       ; load rr[i8][i9]
        SMPY  .M     rr89,_1_64,tmp2       ; smpy(rr[i8][i9],_1_64)
        SADD  .L     alp2,tmp2,alp2        ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
        SADD  .L     alp2,0x8000L,alp_16   ; alp_16=sadd(alp2,0x8000L)
        SMPYH .M     alp,sq2,tmp3          ; smpyh(alp,sq2)
        SMPYH .M     sq,alp_16,tmp4        ; smpyh(sq,alp_16)
        CMPGT .L     tmp3,tmp4,cndr        ; if(smpyh(alp,sq2) > smpyh(sq,alp_16))
 [cndr] MV    .ALU   sq2,sq
 [cndr] MV    .ALU   ps2,ps
 [cndr] MV    .ALU   alp_16,alp
 [cndr] MV    .ALU   i8,ia
 [cndr] MV    .ALU   i9,ib
 [icntr]SUB   .ALU   icntr,1,icntr
 [icntr]B     .S     INNERLOOP             ;branch to the loop
```

Since both *sq* and *alp* are carried over and required from one iteration to the next, their values should be put in registers for speed. You may notice that at least four cycles are required to compute a new *sq* and *alp*, and that the requirement on the functional units does not exceed four execution packets. Therefore, the inner loop can be effected in four cycles per iteration.

For the outer loop, any pair of *rr[ix][i8]*, *rr[iy][i8]*, (where *ix*, *iy* = i0, i1, ..., i7), will definitely hit the memory bank if they are read together. Therefore, they should be loaded in one cycle each.

5.2.5.3 Partitioning the Registers

The total number of registers required for this cycle, including the registers for the pointer of the arrays, loop counters, intermediate results, etc., exceeds the number of registers available. To partition the registers without losing speed, the strategies are:

- For the inner loop, store the results of *ps*, *ia*, and *ib*, whose values are not used in this code.
- For the outer loop, store the pointers of arrays starting at *rr[i5][i8]*, *rr[i6][i8]*, and *rr[i7][i8]*, whose values are needed last in the outer loop.

Assume that before entering this code, *&dn[0]*, *&ipos[0]*, *&rr[0][0]*, *&rrv[0][0]*, *i0*, *i1*, *i2*, *i3*, *i4*, *i5*, *i6*, *i7*, *ps0*, and *alp0* are known. Assume that the Word16 integers are stored in the stack in the order *i0*, *i1*, *i2*, *i3*, *i4*, *i5*, *i6*, *i7*, *ia*, and *ib*, and that a pointer *&local_16[0]*, pointing to *i0*, is also known. The Word32 integers and the pointers of the *rr* arrays are stored in the stack in the order *ps0*, *ps*, *alp0*, *alp1*, *&rr[i5][i8]*, *&rr[i6][i8]*, and *&rr[i7][i8]*. The pointer, *&local_32[0]*, pointing to ps0, is known as well.

The C code is shown in Example 5–29.

Example 5–29. The Index Search Modified C Code

```
 sq = -1;
alp = 1;
local_32[1] = 0;
local_16[8] = ipos[8];
local_16[9] = ipos[9];

/* initialize 10 indices for i8 loop (see i2-i3 loop) */

for (i8 = ipos[8]; i8 < L_CODE; i8 += STEP)
{
  ps1 = _sadd (local_32[0], dn[i8]<<16);

  local_32[3] = _sadd(local_32[2], _smpy(rr[i8][i8], _1_128));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i0][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i1][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i2][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i3][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i4][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i5][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i6][i8], _1_64));
  local_32[3] = _sadd(local_32[3], _smpy(rr[i7][i8], _1_64));

/* initialize 3 indices for i9 inner loop (see i2-i3 loop) */

for (i9 = ipos[9]; i9 < L_CODE; i9 += STEP)
{

  ps2 = _sadd(ps1, dn[i9]<<);

  alp2 = _sadd(local_32[3], _smpy(rrv[i9], _1_8));
  alp2 = _sadd(alp2, _smpy(rr[i8][i9], _1_64));

  sq2 = _smpyh(ps2, ps2);

  alp_16 = _sadd(alp2,0x8000L);
```

Example 5–29. The Index Search Modified C Code (Continued)

```
if (_smpyh(alp,sq2) > _smpyh(sq,alp_16))
   {

        sq = sq2;
        local_32[1]= ps2;
        alp = alp_16;
        local_16[8] = i8;
        local_16[9] = i9;
       }
   }
}
```

5.2.5.4 Final Assembly Code

The final code consists of the following steps:

Step 1: Load *i0*, *i1*, ... *i9*, *alp0*, and *ps0*; and initialize *sq*, *ia*, and ib. Part of the code overlaps that of the last iteration of the code in Section 5.2.3 on page 5-20.

Step 2: Obtain the pointer for the arrays started at *rr[i0][i8]*, *rr[i1][i8]*, ... *rr[i7][i8]*, *rr[i8][i9]*, *rrv[i9]*, *dn[i8]*, and *dn[i9]*.

Step 3: Load *rr[i0][i8]*, *rr[i1][i8]*, ... *rr[i7][i8]* and *dn[i8]*, compute the new *ps1* and *alp1*, update the pointers and store pointers *&rr[i5][i8]*, *&rr[i6][i8]*, and *&rr[i7][i8]*.

Step 4: Load *rr[i8][i9]*, *rrv[i9]*, and *dn[i9]*. Compute *alp2*, *ps2*, *alp_16*, *sq2* and perform a comparison. Update the parameters *ia*, *ib*, *alp*, *sq*, and *ps* based on the comparison result. Repeat this step eight times.

Step 5: Reload the values of *ps0* and *alp0*, and *&rr[i5][i8]*, *&rr[i6][i8]*, and *&rr[i7][i8]*. Verify that Step 3 has been repeated eight times. If not, go to Step 3. If yes, exit.

To avoid memory bank hits, arrays *rr* and *rrv* must not be aligned on the same word or half-word boundary. The same applies to arrays *rr* and *dn*. As you can see in the final assembly code shown in Example 5–30, there are several places that LDH (or STH) and LDW (or STW) occur in the same execution packet. They belong to one of the two categories; that is, always loading values from or storing values to the same memory locations, as in iterations like the following:

```
       LDW    .D1    *+A6[3],A11   ; load alp1
||[B2] STH    .D2    B13,*+B6[9]   ; store ib=i9
```

The following are used in the inner loop in different memory locations such as the outer loop:

```
 [B2]  STW    .D1    B11,*+A6[1]   ; store ps
||     LDH    .D2    *B10++[5],A5  ; load rr[i5][i8]
```

In the former case, the memory bank hits can be completely eliminated by allocating corresponding arrays in memory properly. Memory bank hits occur in every other iteration in the latter case, however. Although, in general, you should avoid writing such code,the performance of the prelog of the outer loop after the first iteration is limited by the .D unit in this case. You still save some cycle counts compared to not doing so.

Notice that you overlapped the last three iterations of the inner loop with part of the prelog of the outer loop to improve the performance.

Example 5–30. Final Assembly Code of the search_10i40 Index Search

```
*******************************************************************************
**     Texas Instruments, Inc                                                **
**                                                                           **
**     Implementation of The Index Search in search_10i40 in EFR             **
**                                                                           **
**     Total cycles = 400 (among the 400 cycles, 10 cycles are caused        **
**                         by memory bank hits                               **
**                                                                           **
**     Register Usage:      A             B                                  **
**                                                                           **
**                          15            15                                 **
**                                                                           **
*******************************************************************************

                          ; A13 --- &ipos[0] and alp
                          ; B6 --- &local_16[0]
                          ; A6 --- stack pointer, point to &local_32[0]
                          ; B8 --- &rr[0][0]
                          ; A4 --- &rrv[0]
                          ; B14 --- &dn[0]
                          ; B1 --- reserved for the counter of the
                          ;        outmost loop in search_10i40


       LDH    .D1    *+A13[8],A7              ; load i8 = ipos[8]

       LDH    .D1    *+A13[9],B13             ; load i9 = ipos[9]
||     LDH    .D2    *B6,A13                  ; load i0
||     MV     .S1X   B6,A5                    ; &local_v16[0]
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
        LDH     .D2     *+B6[2],B9              ; load i2
||      LDH     .D1     *+A5[1],A14             ; load i1
        MVK     .S1     0,A8                    ; could insert two .D
                                                ; units here for the store
                                                ; of rrv[i9+30] and rrv[i9+35]
                                                ; in the code which this piece
                                                ; immediately follows

        LDH     .D1     *+A5[4],A15             ; load i4
||      LDH     .D2     *+B6[3],B10             ; load i3
||      MVK     .S1     80,A0
||      MVK     .S2     80,B0

        STW     .D1     A8,*+A6[1]              ; ps=0
||      LDH     .D2     *+B6[5],B11             ; load i5
||      SHL     .S2X    A7,1,B10                ; [0][i8]
||      MPYU    .M1     A7,A0,A12               ; [i8][0]

        STH     .D1     A7,*+A5[8]              ; store ia=i8
||      STH     .D2     B13,*+B6[9]             ; store ib=i9
||      ADD     .L2     B8,B10,B2               ; &rr[0][i8]
||      MPYU    .M2X    A13,B0,B3               ; [i0][0]

        LDW     .D1     *A6,B15                 ; load ps0
||      LDH     .D2     *+B6[6],A1              ; load i6
||      ADD     .S1X    A12,B2,A12              ; &rr[i8][i8]
||      ADD     .S2     B14,B10,B7              ; &dn[i8]
||      ADD     .L2X    B8,A12,B8               ; &rr[i8][0]
||      MPYU    .M1     A14,A0,A14              ; [i1][0]
||      MPYU    .M2     B9,B0,B9                ; [i2][0]

        LDW     .D1     *+A6[2],A11             ; load alp0
||      LDH     .D2     *+B6[7],B5              ; load i7
||      ADD     .S2     B13,B13,B12             ; [0][i9]
||      ADD     .L2     B3,B2,B3                ; &rr[i0][i8]

        LDH     .D1     *A12,A5                 ; load rr[i8][i8]
||      LDH     .D2     *B7++[5],B12            ; load dn[i8]
||      ADD     .S2     B14,B12,B14             ; &dn[i9]
||      ADD     .L1X    A14,B2,A14              ; &rr[i1][i8]

        LDH     .D2     *B3++[5],A5             ; load rr[i0][i8]
||      ADD     .L2     B9,B2,B9                ; &rr[i2][i8]
||      MPYU    .M1X    B10,A0,A9               ; [i3][0]

        LDH     .D1     *A14++[5],A5            ; load rr[i1][i8]
||      ADD     .L1X    A4,B12,A4               ; &rrv[i9]
||      MPYU    .M1     A15,A0,A15              ; [i4][0]
||      MPYU    .M2     B11,B0,B11              ; [i5][0]
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
        LDH     .D2     *B9++[5],A5          ; load rr[i2][i8]
||      MVK     .S1     256,A0               ; A0=_1_128
||      ADD     L1X     A9,B2,A9             ; &rr[i3][i8]
||      MPYU    .M1     A1,A0,A1             ; [i6][0]

        LDH     .D1     *A9++[5],B12         ; load rr[i3][i8]
||      ADD     .D2     B11,B2,B10           ; &rr[i5][i8]
||      MVK     .S1     7,A2                 ; outer loop counter
||      MVK     .S      512,B0               ; B0=_1_64
||      ADD     .L1X    A15,B2,A15           ; &rr[i4][i8]
||      ADD     .L2     B8,B12,B4            ; &rr[i8][i9]
||      MPYU    .M2     B5,B0,B5             ; [i7][0]

        LDH     .D1     *A15++[5],A5         ; load rr[i4][i8]
||      SHL     .S1     A0,1,A0              ; _1_64

||      SHL     .S2     B12,16,B11           ; dn[i8] << 16
||      ADD     .L1X    A1,B2,A1             ; &rr[i6][i8]
||      SMPY    .M1     A5,A0,A8             ; smpy(rr[i8][i8],_1_128)

        LDH     .D2     *B10++[5],A5         ; load rr[i5][i8]
||      MVK     .S1     -1,A3                ; sq=-1
||      SMPY    .M1     A5,A0,A8             ; smpy(rr[i0][i8],_1_64)

        LDH     .D1     *A1++[5],B12         ; load rr[i6][i8]
||      ADD     .D2     B5,B2,B11            ; &rr[i7][i8]
||      SHL     .S1     A0,7,A13             ; alp=0x10000
||      SADD    .L1     A11,A8,A11           ; alp1=sadd(alp0,smpy(rr[i8][i8],_1_128))
||      SADD    .L2     B15,B11,B15          ; ps1
||      SMPY    .M1     A5,A0,A8             ; smpy(rr[i1][i8],_1_64)

        LDH     .D2     *B11++[5],A5         ; load rr[i7][i8]
||      SADD    .L1     A11,A8,A11           ; alp1=sadd(alp1,smpy(rr[i0][i8],_1_64))
||      SMPY    .M1     A5,A0,A8             ; smpy(rr[i2][i8],_1_64)

OUTERLOOP:

        LDH     .D1     *A4++[5],A5          ; load rrv[i9]
||      LDH     .D2     *B4++[5],B12         ; load rr[i8][i9]
||      SADD    .L1     A11,A8,A11           ; alp1=sadd(alp1,smpy(rr[i1][i8],_1_64))
||      SUB     .L2     B13,5,B13
||      SMPY    .M1X    B12,A0,A8            ; smpy(rr[i3][i8],_1_64)

        LDH     .D2     *B14++[5],B12        ; load dn[i9]
||      SADD    .L1     A11,A8,A11           ; alp1=sadd(alp1,smpy(rr[i2][i8],_1_64))
||      SMPY    .M1     A5,A0,A8             ; smpy(rr[i4][i8],_1_64)
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
        STW     .D1     B10,*+A6[4]           ; store &rr[i5][i8+5]
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i3][i8],_1_64))
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i5][i8],_1_64)

        STW     .D1     A1,*+A6[5]            ; store &rr[i6][i8+5]
||      SHL     .S1     A0,6,A10              ; 0x8000L
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i4][i8],_1_64))
||      SMPY    .M1X    B12,A0,A8             ; smpy(rr[i6][i8],_1_64)

        LDH     .D1     *A4++[5],A5           ;* load rrv[i9]
||      LDH     .D2     *B4++[5],B12          ;* load rr[i8][i9]
||      SHL     .S1     A0,3,A0               ; A0=_1_8
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smpy(rr[i5][i8],_1_64))
||      SMPY    .M1     A5,A0,A8              ; smpy(rr[i7][i8],_1_64)


        LDH     .D2     *B14++[5],B12         ;* load dn[i9]
||      SADD    .L1     A11,A8,A11            ; alp1=sadd(alp1,smyp(rr[i6][i8],_1_64))
||      SMPY    .M1     A5,A0,A5              ; smpy(rrv[i9],_1_8)
||      SMPY    .M2     B12,B0,B12            ; smpy(rr[i8][i9],_1_64)


        STW     .D      B11,*+A6[6]           ; store &rr[i7][i8+5]
||      SHL     .S2     B12,16,B1             ; dn[i9] << 16
||      SADD    .L1     A11,A8,A11            ; done alp1=sadd(alp1,smpy(rr[i7][i8],_1_64))


        STW     .D1     A11,*+A6[3]           ; store alp1
||      SADD    .L1     A11,A5,A5             ; alp2=sadd(alp1,smpy(rrv[i9],_1_8))
||      SADD    .L2     B11,B15,B5            ; ps2=sadd(ps1,dn[i9]<<16)


        LDH     .D1     *A4++[5],A5           ;** load rrv[i9]
||      LDH     .D2     *B4++[5],B12          ;** load rr[i8][i9]
||      B       .S2     INNERLOOP             ; branch to the innerloop
||      SADD    .L1X    A5,B12,A1             ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||      SMPYH   .M2     B5,B5,B8              ; sq2=smpyh(ps2,ps2)


        LDH     .D2     *B14++[5],B12         ;** load dn[i9]
||      MVK     .S1     4,A1                  ; innerloop counter
||      MVK     .S2     0,B2
||      SADD    .L1     A1,A10,A8             ; alp_16 = sacc(alp2, 0x8000L)
||      SMPY    .M1     A5,A0,A5              ;* smpy(rrv[i9],_1_8)
||      SMPY    .M2     B12,B0,B12            ;* smpy(rr[i8][i9],_1_64)
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
INNERLOOP:

        LDW     .D1     *+A6[3],A11         ; load alp1
||[B2]  STH     .D2     B13,*+B6[9]         ; store ib=i9
||      SHL     .S2     B12,16,B10          ;* dn[i9]<<16
||      ADD     .L2     B13,5,B13           ; i9=i9+STEP
||      SMPYH   .M1     A8,A3,A11           ; smpyh(alp_16,sq)
||      SMPYH   .M2X    B8,A13,B10          ; smpyh(alp,sq2)


 [B2]   STW     .D1     B11,*+A6[1]         ; store ps
||[B2]  STH     .D2     A7,*+B6[8]          ; store ia = i8
||      MV      .S2     B5,B11              ;
||      SADD    .L1     A11,A5,A5           ;*alp2=sadd(alp1,smpy(rrv[i9],_1_8))
||      SADD    .L2     B10,B15,B5          ;* ps2=sadd(ps1,dn[i9]<<16)


        LDH     .D1     *A4++[5],A5         ;*** load rrv[i9+10]
||      LDH     .D2     *B4++[5],B12        ;*** load rr[i8][i9+10]
||[A1]  SUB     .S1     A1,1,A1             ; decrement innerloop counter
||[A1]  B       .S2     INNERLOOP           ; branch to INNERLOOP
||      SADD    .L1X    A5,B12,A11          ;*alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||      CMPGT   .L2X    B10,A11,B2          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)
||      SMPYH   .M2     B5,B5,B8            ;* sq2=smpyh(ps2,ps2)


  [B2]  MV      .D1     A8,A13              ; alp=alp_16
||      LDH     .D2     *B14++[5],B12       ;*** load dn[i9+10]
||[B2]  MV      .S1X    B8,A3               ; sq=sq2
||      SADD    .L1     A11,A10,A8          ;* alp_16=sadd(alp2, 0x8000L)
||      SMPY    .M1     A5,A0,A5            ;*** A0 = _1_8
||      SMPY    .M2     B12,B0,B12          ;*** B0 = _1_64
                                            ; end of innerloop


 [B2]  STW     .D1     B11,*+A6[1]         ; store ps
||[B2] STH     .D2     A7,*+B6[8]          ; store ia = i8
||     SHL     .S2     B12,16,B10          ; dn[i9]<<16
||     MV      .L2     B5,B11              ; ps2
||     SMPYH   .M1     A8,A3,A11           ; smpyh(alp_16,sq)
||     SMPYH   .M2X    B8,A13,B10          ; smpyh(alp,sq2)


       LDW     .D1     *+A6[2],A11         ; load alp0
||[B2] STH     .D2     B13,*+B6[9]         ; store ib=i9
||     MV      .S2X    A6,B2               ; stack pointer
||     SADD    .L1     A11,A5,A5           ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||     SADD    .L2     B10,B15,B5          ; ps2=sadd(ps1,dn[i9]<<16)
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
         LDW     .D1     *+A6[5],A1          ; &rr[i6][i8]
||       LDW     .D2     *B2,B15             ; load ps0
||       MVK     .S1     205,A0
||       SADD    .L1X    A5,B12,A11          ; alp2=sadd(alp2,smpy(rr[i8][i9],_1_64))
||       CMPGT   .L2X    B10,A11,B2          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)
||       SMPYH   .M2     B5,B5,B8            ; sq2=smpyh(ps2,ps2)


         LDH     .D1     *++A12[A0],A5       ; load rr[i8][i8]
||       LDH     .D2     *B7++[5],B12        ; load dn[i8]
||[B2]   MV      .S1     A8,A13              ; alp=alp_16
||       ADDK    .S2     -90,B14             ; &dn[i9]
||[B2]   MV      .L1X    B8,A3               ; sq=sq2


         LDW     .D1     *+A6[4],B10         ; &rr[i5][i8]
||       LDH     .D2     *B3++[5],A5         ; load rr[i0][i8]
||       ADDK    .S1     -90,A4              ; &rrv[i9]
||       SADD    .L1     A11,A10,A8          ; alp_16=sadd(alp2, 0x8000L)
||       ADD     .L2     B13,5,B13


         LDH     .D1     *A14++[5],A5        ; load rr[i1][i8]
||[B2]   STH     .D2     B13,*+B6[9]         ; store ib=i9
||       SMPYH   .M1     A8,A3,A10           ; smpyh(alp_16,sq)
||       SMPYH   .M2X    B8,A13,B10          ; smpyh(alp,sq2)


         MVK     .S1     256,A0              ; _1_128
||       LDW     .D1     *+A6[6],B11         ; &rr[i7][i8]
||       LDH     .D2     *B9++[5],A5         ; load rr[i2][i8]
||[A2]   B       .S2     OUTERLOOP           ; branch to OUTERLOOP


         LDH     .D1     *A9++[5],B12        ; load rr[i3][i8]
||[B2]   STH     .D2     A7,*+B6[8]          ; store ia = i8
||       ADD     .S2     B13,5,B13           ; update i9
||       CMPGT   .L2X    B10,A10,B0          ; if smpyh(alp,sq2) > smpyh(alp_16,sq)


         LDH     .D1     *A15++[5],A5        ; load rr[i4][i8]
||[B0]   STH     .D2     B13,*+B6[9]         ; store ib=i9
||       SHL     .S1     A0,1,A0             ; _1_64
||       ADDK    .S2     -35,B13             ; update i9
||[B0]   MV      .L1     A8,A13              ; alp=alp_16
||       SMPY    .M1     A5,A0,A             ; smpy(rr[i8][i8],_1_128)
```

Example 5–30. Final Assembly Code of the search_10i40 Index Search (Continued)

```
 [B2]  STW    .D1B11,*+A6[1]            ; store ps
||     LDH    .D2    *B10++[5],A5       ; load rr[i5][i8]
||[B0] MV     .S1X   B8,A3              ; sq=sq2
||     SHL    .S2    B12,16,B11         ; dn[i8] << 16
||[A2] SUB    .L1    A2,1,A2            ; decrement OUTERLOOP counter
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i0][i8],_1_64)


       LDH    .D1    *A1++[5],B12       ; load rr[i6][i8]
||[B0] STH    .D2    A7,*+B6[8]         ; store ia = i8
||     ADDK   .S2    310,B4             ; &rr[i8][i9]
||     SADD   .L1    A11,A8,A11         ; alp1=sadd(alp0,smpy(rr[i8][i8],_1_128))
||     SADD   .L2    B15,B11,B15        ; ps1 = sadd(ps0,dn[i8]<<16)
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i1][i8],_1_64)


 [B0]  STW    .D1    B5,*+A6[1]         ; store ps
||     LDH    .D2    *B11++[5],A5       ; load rr[i7][i8]
||     ADD    .S1    A7,5,A7            ; update i8
||     SADD   .L1    A11,A8,A11         ; alp1=sadd(alp1,smpy(rr[i0][i8],_1_64))
||     MV     .L2X   A0,B0              ; _1_64
||     SMPY   .M1    A5,A0,A8           ; smpy(rr[i2][i8],_1_64)
```

5.2.6 Implementation of residu.c, the FIR Filter, In GSM EFR

Example 5–31 shows the FIR filter, residu.c, in GSM EFR.

Example 5–31. residu.c C Code

```
 #define Word16  short #define Word32  int
  Original C code
-------------------------------------------------------
/* m = LPC order == 10 */ #define m 10

void Residu (
    Word16 a[], /* (i)     : prediction coefficients                    */
    Word16 x[], /* (i)     : speech signal                              */
    Word16 y[], /* (o)     : residual signal                            */
    Word16 lg   /* (i)     : size of filtering                          */
)
{
    Word16 i, j;
    Word32 s;


    for (i = 0; i < lg; i++)
    {
        s = L_mult (x[i], a[0]);
        for (j = 1; j <= m; j++)
            s = L_mac (s, a[j], x[i - j]);
        s = L_shl (s, 3);
        y[i] = round (s);
    }
    return;
}
-------------------------------------------------------
  where L_mult(a,b) = _smpy(a,b)
        L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        L_shl(a,b) = (b>0) ? _sshl(a,b) : _shr(a,b)
        round(a) = _sadd(a,0x8000L)>>16
   and  lg = 40.
```

5.2.6.1 Rearranging the C Code

It is obvious that *L_shl(s,3)* can be implemented simply as *_sshl(s,3)*. Since array *a* has dimension m + 1 = 11 and the inner loop is always executed 10 times per outer loop iteration, you can completely unroll the inner loop to gain the speed by representing array *a* with registers. Since *a* is a short integer array, array *a* requires at most six registers for full representation. You can assign one register only for *a[0]* for two reasons: 1) *a[0]* is always a constant, 4096 and, 2) *_shr(0x8000L,3)* = 4096. You can change the order of rounding and left shift to save one register. (If not, you need another register for 0x8000L.) The C code after complete inner loop unrolling is shown in Example 5–32.

Example 5–32. residu.c C Code After Rearrangement Using Intrinsics

```
for (i = 0; i < lg; i++)
{
    s = _smpy(x[i], a[0]);
    s = _sadd(s,_smpy(a[1], x[i-1]));
    s = _sadd(s,_smpy(a[2], x[i-2]));
    s = _sadd(s,_smpy(a[3], x[i-3]));
    s = _sadd(s,_smpy(a[4], x[i-4]));
    s = _sadd(s,_smpy(a[5], x[i-5]));
    s = _sadd(s,_smpy(a[6], x[i-6]));
    s = _sadd(s,_smpy(a[7], x[i-7]));
    s = _sadd(s,_smpy(a[8], x[i-8]));
    s = _sadd(s,_smpy(a[9], x[i-9]));
    s = _sadd(s,_smpy(a[10], x[i-10]));
    s = _sadd(s, a[0]);
    s = _sshl(s,3);
    y[i] = _shr(s,16);
}
```

5.2.6.2 Performance Analysis

The performance is limited by .L unit for *_sadd* or by .M unit for *_smpy*, since both of these two units are used at least 11 times per iteration. In other words, it takes at least six cycles per iteration. You may chose to unroll the loop once to compute two *y*s per iteration for the following reasons:

1) To satisfy the ordering property of *_sadd*

2) To maximize speed. Eleven cycles are required to compute two *y*s, while six cycles are needed for one *y*.

The C code is is shown in Example 5–33.

Example 5–33. Implemented C Code for residu.c

```
for (i = 0; i < lg; i+=2)
{
    s0 = _smpy(x[i], a[0]);
    s1 = _smpy(x[i+1], a[0]);
    s0 = _sadd(s0,_smpy(a[1], x[i-1]));
    s1 = _sadd(s1,_smpy(a[1], x[i]));
    s0 = _sadd(s0,_smpy(a[2], x[i-2]));
    s1 = _sadd(s1,_smpy(a[2], x[i-1]));
    s0 = _sadd(s0,_smpy(a[3], x[i-3]));
    s1 = _sadd(s1,_smpy(a[3], x[i-2]));
    s0 = _sadd(s0,_smpy(a[4], x[i-4]));
    s1 = _sadd(s1,_smpy(a[4], x[i-3]));
    s0 = _sadd(s0,_smpy(a[5], x[i-5]));
    s1 = _sadd(s1,_smpy(a[5], x[i-4]));
    s0 = _sadd(s0,_smpy(a[6], x[i-6]));
    s1 = _sadd(s1,_smpy(a[6], x[i-5]));
    s0 = _sadd(s0,_smpy(a[7], x[i-7]));
    s1 = _sadd(s1,_smpy(a[7], x[i-6]));
    s0 = _sadd(s0,_smpy(a[8], x[i-8]));
    s1 = _sadd(s1,_smpy(a[8], x[i-7]));
    s0 = _sadd(s0,_smpy(a[9], x[i-9]));
    s1 = _sadd(s1,_smpy(a[9], x[i-8]));
    s0 = _sadd(s0,_smpy(a[10], x[i-10]));
    s1 = _sadd(s1,_smpy(a[10], x[i-9]));
    s0 = _sadd(s0, a[0]);
    s1 = _sadd(s1, a[0]);
    s0 = _sshl(s0,3);
    s1 = _sshl(s1,3);
    y[i] = _shr(s0,16);
    y[i+1] = _shr(s1,16);
}
```

5.2.6.3 Final Assembly Code for residu.c

The final assembly code is shown in Example 5–34.

Example 5–34. residu.c Final Assembly Code

```
*******************************************************************************
**                                                                           **
**      Implementation of residu.c EFR                                       **
**                                                                           **
**      Compute two ys at a time                                             **
**                                                                           **
**      Total cycles = (lg/2+1)*11+6                                         **
**                   = 237  (for lg = 40)                                    **
**      Register Usage:              A               B                       **
**                                   9               10                      **
**                                                                           **
*******************************************************************************
                                  ; A4 --- &a[0]
                                  ; B4 --- &x[0]
                                  ; A6 --- &y[0]
                                  ; B6 --- lg

        LDH     .D2     *B4++,B0          ; load a[0] = 4096

        LDW     .D1     *A4--,A3          ; load x[0] & x[1]
||      LDW     .D2     *B4++,B4          ; load a[1] & a[2]

        LDW     .D1     *A4--,A1          ; load x[-2] & x[-1]
||      LDW     .D2     *B4++,B1          ; load a[3] & a[4]

        LDW     .D2     *B4++,B5          ; load a[5] & a[6]

        LDW     .D2     *B4++,B6          ; load a[7] & a[8]
||      LDW     .D1     *A4--,A3          ; load x[-4] & x[-3]

        LDW     .D2     *B4++,B7          ; load a[9] & a[10]
||      MVK     .S1     1,A2              ; to take care of the first execution
||      MV      .L1X    B0,A0             ; a[0] = 4096
||      MV      .S2     B6,B2             ; loop counter, L_SUBFR/2

LOOP:
        SMPY    .M1     A3,A0,A8          ; smpy(x[0],a[0])
||      SMPYHL  .M2X    A3,B0,B8          ; smpy(x[1],a[0])
||      LDW     .D1     *A4--,A1          ; load x[-6] & x[-5]
||[!A2] SADD    .L1     A8,A9,A9          ; s0 = sadd(s0, smpy(x[-9],a[9]))
||[!A2] SADD    .L2     B8,B9,B9          ; s1 = sadd(s1, smpy(x[-8],a[9]))

        SMPYHL  .M1X    A1,B4,A8          ; smpy(x[-1],a[1])
||      SMPY    .M2X    A3,B4,B8          ; smpy(x[0],a[1])
||[!A2] SADD    .L1     A8,A9,A9          ; s0 = sadd(s0, smpy(x[-10],a[10]))
||[!A2] SADD    .L2     B8,B9,B9          ; s1 = sadd(s1, smpy(x[-9],a[10]))
```

Example 5–34. residu.c Final Assembly Code (Continued)

```
         SMPYLH   .M1X     A1,B4,A8          ; smpy(x[-2],a[2])
 ||      SMPYH    .M2X     A1,B4,B8          ; smpy(x[-1],a[2])
 ||      ADD      .S1      A8,0,A9           ; s0=smpy(x[0],a[0])
 ||      ADD      S2       B8,0,B9           ; s1=smpy(x[1],a[0])
 ||      LDW      .D1      *A4--,A3          ; load x[-8] & x[-7]
 ||[!A2] SADD     L1       A9,A0,A9          ; s0 = sadd(s0, 4096)
 ||[!A2] SADD     .L2      B9,B0,B9          ; s1 = sadd(s1, 4096)

         SMPYHL   .M1X     A3,B1,A8          ; smpy(x[-3],a[3])
 ||      SMPY     .M2X     A1,B1,B8          ; smpy(x[-2],a[3])
 ||      SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-1],a[1]))

 ||      SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[0],a[1]))
 ||[!A2] SSHL     .S1      A9,3,A7           ; s0 = L_shl(s0,3)
 ||[!A2] SSHL     .S2      B9,3,B1           ; s1 = L_shl(s1,3)

         SMPYLH   .M1X     A3,B1,A8          ; smpy(x[-4],a[4])
 ||      SMPYH    .M2X     A3,B1,B8          ; smpy(x[-3],a[4])
 ||      SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-2],a[2]))
 ||      SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-1],a[2]))
 ||      LDW      .D1      *A4++[6],A1       ; load x[-10] & x[-9] and update the
pointer
 ||[!A2] SHR      .S1      A7,16,A7          ; y[0] = shr(s0, 16)
 ||[!A2] SHR      .S2      B10,16,B10        ; y[1] = shr(s1, 16)
                                             ; to the new &x[0]

         SMPYHL   .M1X     A1,B5,A8          ; smpy(x[-5],a[5])
 ||      SMPY     .M2X     A3,B5,B8          ; smpy(x[-4],a[5])
 ||      SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-3],a[3]))
 ||      SADD     .L2      B8,B9,B           ; s1 = sadd(s1, smpy(x[-2],a[3]))
 ||[!A2] STH      .D1      A7,*A6++          ; store y[0]
 || [B2] SUB      .S2      B2,2,B            ; decrement loop counter
 || [B2] B        S1       LOOP              ; branch to the loop

         SMPYLH   .M1X     A1,B5,A8          ; smpy(x[-6],a[6])
 ||      SMPYH    .M2X     A1,B5,B8          ; smpy(x[-5],a[6])
 ||      SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-4],a[4]))
 ||      SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-3],a[4]))
 ||      LDW      .D1      *A4--,A3          ;* load x[0] & x[1] for the next iteration

         SMPYHL   .M1X     A3,B6,A8          ; smpy(x[-7],a[7])
 ||      SMPY     .M2X     A1,B6,B8          ; smpy(x[-6],a[7])
 ||      SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-5],a[5]))
 ||      SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-4],a[5]))
 ||      LDW      .D1      *A4--,A1          ;* load x[-1] & x[-2]
```

Example 5–34. residu.c Final Assembly Code (Continued)

```
         SMPYLH   .M1X     A3,B6,A8          ; smpy(x[-8],a[8])
||       SMPYH    .M2X     A3,B6,B8          ; smpy(x[-7],a[8])
||       SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-6],a[6]))
||       SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-5],a[6]))
|||[!A2] STH      .D1      B10,*A6++         ; store y[1]

         SMPYHL   .M1X     A1,B7,A8          ; smpy(x[-9],a[9])
||       SMPY     .M2X     A3,B7,B8          ; smpy(x[-8],a[9])
||       SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-7],a[7]))
||       SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-6],a[7]))
||[A2]   SUB      .S2      A2,1,A2
||       LDW      .D1      *A4--,A3          ;* load x[-3] & x[-4]


         SMPYLH   .M1X     A1,B7,A8          ; smpy(x[-10],a[10])
||       SMPYH    .M2X     A1,B7,B8          ; smpy(x[-9],a[10])
||       SADD     .L1      A8,A9,A9          ; s0 = sadd(s0, smpy(x[-8],a[8]))
||       SADD     .L2      B8,B9,B9          ; s1 = sadd(s1, smpy(x[-7],a[8]))
```

There is no memory bank hit within the loop. To avoid a memory bank hit within the prelog of the loop, arrays *a* and *x* must be allocated like the *a[1]* and *x[0]* relative offset. Some of the instructions in the loop cannot be executed in the first iteration. Register A2 indicates which instructions these are.

5.2.7 Implementation of the Lag Search In the Routine lag_max()

Routine *Lag_max()* performs an open-loop pitch search and computes the normalized correlation for the selected lag. This subsection illustrates the implementation of the lag search. The lag search C code is shown in Example 5–35.

Example 5–35. Lag Search C Code for lag_max()

```
#define Word16  short
#define Word32  int
#define MIN_32  0x80000000L
#define PIT_MAX 143
#define L_FRAME 160


input:
        Word16 scal_sig[PIT_MAX+L_FRAME]; (pointed at scal_sig[PIT_MAX] when passed)
        Word16 scal_fac; (not used in this part of the code)
        Word16 L_frame, lag_min, lag_max;


local variables:
        Word16 i, j, *p, *p1, p_max;
        Word32 t0, max;


return:
        Word16 p_max;


Original C code
-------------------------------------------------------
    max = MIN_32;


    for (i = lag_max; i >= lag_min; i--)
    {
        p = scal_sig;
        p1 = &scal_sig[-i];
        t0 = 0;


        for (j = 0; j < L_frame; j++, p++, p1++)
        {
            t0 = L_mac (t0, *p, *p1);
        }
        if (L_sub (t0, max) >= 0)
        {
            max = t0;
            p_max = i;
        }
    }
-------------------------------------------------------
where   L_mac(a,b,c) = _sadd(a,_smpy(b,c))
        L_sub(a,b) = _ssub(a,b)
        L_frame = L_FRAME/2 = 80
   and the search range (lag_min, lag_max) is (18,35), (36,71), or (72,143).
```

5.2.7.1 Rearranging The C Code and Unrolling The Loops

This algorithm is preferable to smaller lag candidates, because it performs a comparison with *if(L_sub(t0,max) >= 0)*. notice that the search starts from *lag_max*. Since we do not have single instruction for >= (or <=) comparison, you must change the search order to start from *lag_min* to compare with *if(t0 > max)*. *p_max* is initialized to *lag_min* to handle the extreme case that every *t0* within the search range equals *MIN_32*. The C code is modified as shown in Example 5–36.

Example 5–36. C Code With the Comparison Order Changed

```
max = MIN_32;
p_max = lag_min;
for (i = lag_min; i < lag_max; i++)
{
  p  = scal_sig;
  p1 = &scal_sig[-i];
  t0 = 0;

  for (j=0; j<L_frame; j++, *p++, *p1++) {
    t0 = L_mac(t0, *p, *p1);
  }
  if (t0 > max)
  {
    max = t0;
    p_max = i;
  }
}
```

Next, look at the inner loop, a general MAC loop. Since **p* does not always equal **p1*, it does not fall into the special case shown in subsection 5.2.1 on page 5-3. Therefore, the performance cannot be improved by simply unrolling the inner loop.

Now consider unrolling the outer loop once. The C code with outer loop unrolling is shown in Example 5–37. Notice that since the number of lags that needs to be searched within each search range is always even, such unrolling will not create an additional case to handle.

Example 5–37. C Code With Outer Loop Unrolling

```
Word32  t1;


  max = MIN_32;
  p_max = lag_min;
  for (i = lag_min; i < lag_max; i+=2)
  {
    p = scal_sig;
    p1 = scal_sig[-i];
    t0 = 0;
    t1 = 0;
    for (j=0; j<L_frame; j++, p++, p1++) {
      t1=_sadd(t1,_smpy(*p,*-p1)); (or t1=_sadd(t1,_smpy(scal_sig[j],scal_sig[-i-1+j]))
      t0=_sadd(t0,_smpy(*p,*p1));  (or t0=_sadd(t0,_smpy(scal_sig[j],scal_sig[-i+j]))
    }
    if (t0 > max)
    {
      max = t0;
      p_max = i;
    }
    if( t1 > max)
    {
      max = t1;
      p_max = i+1;
    }
  }


with intrinsics substitutes.
```

Notice that in the order of the comparisons, the smaller lag is always compared first.

The instructions required for one iteration of the inner loop are shown in Example 5–38.

Example 5–38. Inner Loop Instructions

```
INNERLOOP:
         LDH    .D    *p++, sigj              ; load scal_sig[j]
         LDH    .D    *-p1, scalij1           ; load scal_sig[-i-1+j]
         SMPY   .M    sigj,scalij1,tmp1       ; smpy(scal_sig[j],scal_sig[-i-1+j])
         SADD   .L    t1,tmp1,t1              ; t1=sadd(t1,smpy(scal_sig[j],scal_sig[-i-1+j])
         LDH    .D    *p1++,scalij            ; load scal_sig[-i+j]
         SMPY   .M    sigj,scalij,tmp0        ; smpy(scal_sig[j],scal_sig[-i+j])
         SADD   .L    t0,tmp0,t0              ; t0=sadd(t0,smpy(scal_sig[j],scal_sig[-i+j])
[icntr]  SUB    .S    icntr,1,icntr           ; decrement inner loop counter
[icntr]  B      .S    INNERLOOP               ; branch to inner loop
```

The .D unit is the unit used the most (three times). Therefore, the inner loop takes two cycles.

Now unroll the inner loop once. Notice the first iteration of *t1* and the last iteration of *t0* perform outside the inner loop. This avoids memory bank hits. The C code with inner loop unrolled is shown in Example 5–39.

Example 5–39. Search Code With Inner and Outer Loops Unrolled

```
Word32  t1;

  max = MIN_32;
  p_max = lag_min;
  for (i = lag_min; i < lag_max; i+=2)
  {
    p = scal_sig;
    p1 = scal_sig[-i];
    t0 = 0;
    t1=_sadd(t1,_smpy(*p,*-p1)); (or t1=_sadd(t1,_smpy(scal_sig[j],scal_sig[-i-1+j]))
for (j=0; j<(L_frame-1); j+=2, p+=2, p1+=2) {
    t0=_sadd(t0,_smpy(*p,*p1));  (or t0=_sadd(t0,_smpy(scal_sig[j],scal_sig[-i+j]))
    t1=_sadd(t1,_smpy(*+p,*p1)); (or t1=_sadd(t1,_smpy(scal_sig[j+1],scal_sig[-i+j]))
    t0=_sadd(t0,_smpy(*+p,*+p1));  (or t0=_sadd(t0,_smpy(scal_sig[j+1],scal_sig[-i+j+1]))
    t1=_sadd(t1,_smpy(*+p[2],*+p1)); (or t1=_sadd(t1,_smpy(scal_sig[j+2],scal_sig[-i+j+1]))
    }
   t0=_sadd(t0,_smpy(scal_sig[L_frame-1],scal_sig[-i+L_frame-1]));
   if (t0 > max)
   {
     max = t0;
     p_max = i;
   }
   if( t1 > max)
   {
     max = t1;
     p_max = i+1;
   }
 }
```

Although five values of *scal_sig*, *[scal_sig[j], scal_sig[j+1]*, *scal_sig[j+2]*, *scal_sig[–i+j]*, and *scal_sig[–i+j+1]* are required for each inner loop iteration, *scal_sig[j]* does not need to be loaded, since it was loaded in the previous iteration. This means only four loads are required per iteration. Example 5–40 gives the instructions for the modified inner loop.

Example 5–40. Inner Loop Instructions

```
        LDH   .D  *p++, sigj              ; load scal_sig[j]
        LDH   .D  *-p1, scalij1           ; load scal_sig[-i-1+j]
        SMPY  .M  sigj, scalij1,t1        ; t1=smpy(scal_sig[j],scal_sig[-i-1+j])

INNERLOOP:
        LDH   .D  *p1++, scalij           ; load scal_sig[-i+j]
        SMPY  .M  sigj,scalij,tmp0        ; smpy(scal_sig[j],scal_sig[-i+j])
        SADD  .L  t0,tmp0,t0              ; t0=sadd(t0,smpy(scal_sig[j],scal_sig[-i+j])
        LDH   .D  *p++, sigj+1            ; load scal_sig[j+1]
        SMPY  .M  sigj+1,scalij,tmp1      ; smpy(scal_sig[j+1],scal_sig[-i+j])
        SADD  .L  t1,tmp1,t1              ; t1=sadd(t1,smpy(scal_sig[j+1],scal_sig[-i+j])
        LDH   .D  *p1++,scalij+1          ; load scal_sig[-i+j+1]
        SMPY  .M  sigj+1,scalij+1,tmp0    ; smpy(scal_sig[j+1],scal_sig[-i+j+1])
        SADD  .L  t0,tmp0,t0              ; t0=sadd(t0,smpy(scal_sig[j+1],scal_sig[-i+j+1])
        LDH   .D  *p++, sigj+2            ; load scal_sig[j+2], the scal_sig[j] for the
                                          ; next iteration
        SMPY  .M  sigj+2,scalij+1,tmp1    ; smpy(scal_sig[j+2],scal_sig[-i+j+1])
        SADD  .L  t1,tmp1,t1              ; t1=sadd(t1,smpy(scal_sig[j+2],scal_sig[-i+j+1])
[icntr] SUB   .S  icntr,2,icnt            ; decrement inner loop counter
[icntr] B     .S  INNERLOOP               ; branch to inner loop
```

The inner loop uses two cycles. You double the performance, therefore, by unrolling both the outer loop and inner loop if no memory bank hits occur.

5.2.7.2 Avoiding Memory Bank Hits

Load *scal_sig[–i+j]* and *scal_sig[j+1]* together and *scal_sig[–i+j+1]* and *scal_sig[j+2]* together to avoid memory-bank hits.

5.2.7.3 The Final Assembly Code for Lag Search

The final assembly code for the lag search segment is shown in Example 5–41.

Example 5–41. Final Assembly Code for Lag Search in Lag_max()

```
*******************************************************************************
**                                                                           **
**      Implementation of residu.c EFR                                       **
**                                                                           **
**      Compare two lags a time                                              **
**                                                                           **
**      Total cycles = 7+(L_frame+6)*(lag_max-lag_min+1)/2                   **
**                                                                           **
**      Register Usage:         A              B                             **
**                              10             9                             **
**                                                                           **
*******************************************************************************
                        ; A4 --- &scal_sig
                        ; A6 --- lag_max
                        ; B6 --- lag_min

        SUBAH   .D1     A4,A6,A7        ; p1=&scal_sig[-LAG_MIN]
||      MVK     .S2     1,B2
||      SUB     .L1X    B6,A6,A1        ; the outer loop counter
||      MV      .L2X    A4,B7           ; p=&scal_sig[0]
||      MPY     .M2     B0,0,B0         ; initialize the comparison result
||      MPY     .M1     A2,0,A2         ; take care the initial iteration
||      MV      .S1     A6,A4           ; p_max = lag_min

        SHL     .S2     B2,31,B2        ; max=MIN_32=0x80000000L
||      LDH     .D1     *-A7[1],A5      ; scal_sig[-LAG_MIN-1]
||      LDH     .D2     *B7,B5          ; scal_sig[0]
||      ADD     .L1     A1,1,A1         ; make the counter to be an even number

OUTERLOOP:

        LDH     .D1     *A7,A5          ; scal_sig[-LAG_MIN]
||      LDH     .D2     *+B7[1],B6      ; scal_sig[1]
||[A2]  SADD    .L2     B10,B8,B10
||[A1]  MV      .S2     37,B1           ; inner loop counter
||      MPY     .M1     A3,0,A3
||      MPY     .M2     B8,0,B8
||      ADD     .S1     A7,2,A9         ; &scal_sig[-LAG_MIN+1]
||      SUB     .L1     A7,4,A7         ; update p1 = &scal_sig[-LAG_MIN-2]

        LDH     .D1     *A9++,A5        ; scal_sig[-LAG_MIN+1]
||      LDH     .D2     *+B7[2],B5      ; scal_sig[2]
||[B1]  B       .S      INNERLOOP       ; branch to the inner loop
||[A2]  CMPGT   .L2     B10,B2,B0       ; if(t0>max)

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+2]
||     LDH    .D2    *+B7[3],B6  ; scal_sig[3]
||[B0] MV     .L2    B10,B2      ; max = t0
||     MPY    .M1X   B1,1,A2     ; counter to branch to the outerloop
```

Example 5–41. Final Assembly Code for Lag Search in Lag_max() (Continued)

```
       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+3]
||     LDH    .D2    *+B7[4],B5  ; scal_sig[4]
||[B1] B      .S2    INNERLOOP   ; branch to the inner loop
||[A2] CMPGT  .L2X   A0,B2,B0    ; if(t1>max)
||[B0] SUB    .L1    A6,2,A4     ; p_max = i
||     ADD    .S1    A6,2,A6     ; update i
||     MPY    .M1    A0,0,A0     ; initialize t1=0
||     MPY    .M2    B10,0,B10   ; initialize t0=0

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+4]
||     LDH    .D2    *+B7[5],B6  ; scal_sig[5]
||     SMPY   .M1X   A5,B5,A3    ; _smpy(scal_sig[-LAG_MIN-1], scal_sig[0])
||[B0] MV     .L2X   A0,B2        ; max = t1
||[B0] SUB    .L1    A6,3,A4     ; p_max = i+1
||[A1] SUB    .S1    A1,2,A1     ; update inner loop counter
||     ADD    .S2    B7,12,B9    ; &scal_sig[1]

 INNERLOOP:

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+5]
||     LDH    .D2    *B9++,B5    ; scal_sig[6]
||     SMPY   .M1X   A5,B6,A3    ; _smpy(scal_sig[-LAG_MIN], scal_sig[1])
||     SMPY   .M2X   A5,B5,B8    ; _smpy(scal_sig[-LAG_MIN], scal_sig[0])
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||[B1] B      .S1    INNERLOOP   ; branch to inner loop
||[B1] SUB    .S2    B1,1,B1     ; decrement inner loop counter

       LDH    .D1    *A9++,A5    ; scal_sig[-LAG_MIN+6]
||     LDH    .D2    *B9++,B6    ; scal_sig[7]
||     SMPY   .M1X   A5,B5,A3    ; _smpy(scal_sig[-LAG_MIN+1], scal_sig[2])
||     SMPY   .M2X   A5,B6,B8    ; _smpy(scal_sig[-LAG_MIN+1], scal_sig[1])
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||     SUB    .S1    A2,1,A2     ; decrement the counter to branch to the outer loop
||[!A2]B      .S2    OUTERLOOP   ; branch to the outer loop

       LDH    .D1    *-A7[1],A5  ; scal_sig[-LAG_MIN-3]
||     LDH    .D2    *B7,B5      ; scal_sig[0]
||     SADD   .L1    A0,A3,A0    ; update t1
||     SADD   .L2    B10,B8,B10  ; update t0
||[!A1]B      .S1    FINISH      ; lag search is complete

FINISH:

       NO     5
```

Notice that all the epilogs and prelogs of the outer and inner loops are compressed to minimize the code size. *A2* is both the indicator for avoiding comparisons during the initial iteration of the outer loop and the counter for branching to the outer loop during inner loop executions.

Index

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Y