TMS320C6000
Optimizing C Compiler Tutorial

Literature Number: SPRU425A
August 2002

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of that third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Optimizing C Compiler Tutorial

This tutorial walks you through the code development flow, describes compiler feedback, and introduces you to compiler optimization techniques. It uses step-by-step instructions and code examples to show you how to use the software development tools in each phase of development.

Before you start this tutorial, you should install Code Composer Studio v1.2.

The sample code used in this tutorial is included on both the code generation tools and Code Composer Studio CD-ROMs. When you install the code generation tools, the sample code is installed in c:\ti\c6000\examples\cgtools\prog_gd\tutorial. Use the code in that directory to go through the examples in the tutorial.

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

Topic **Page**

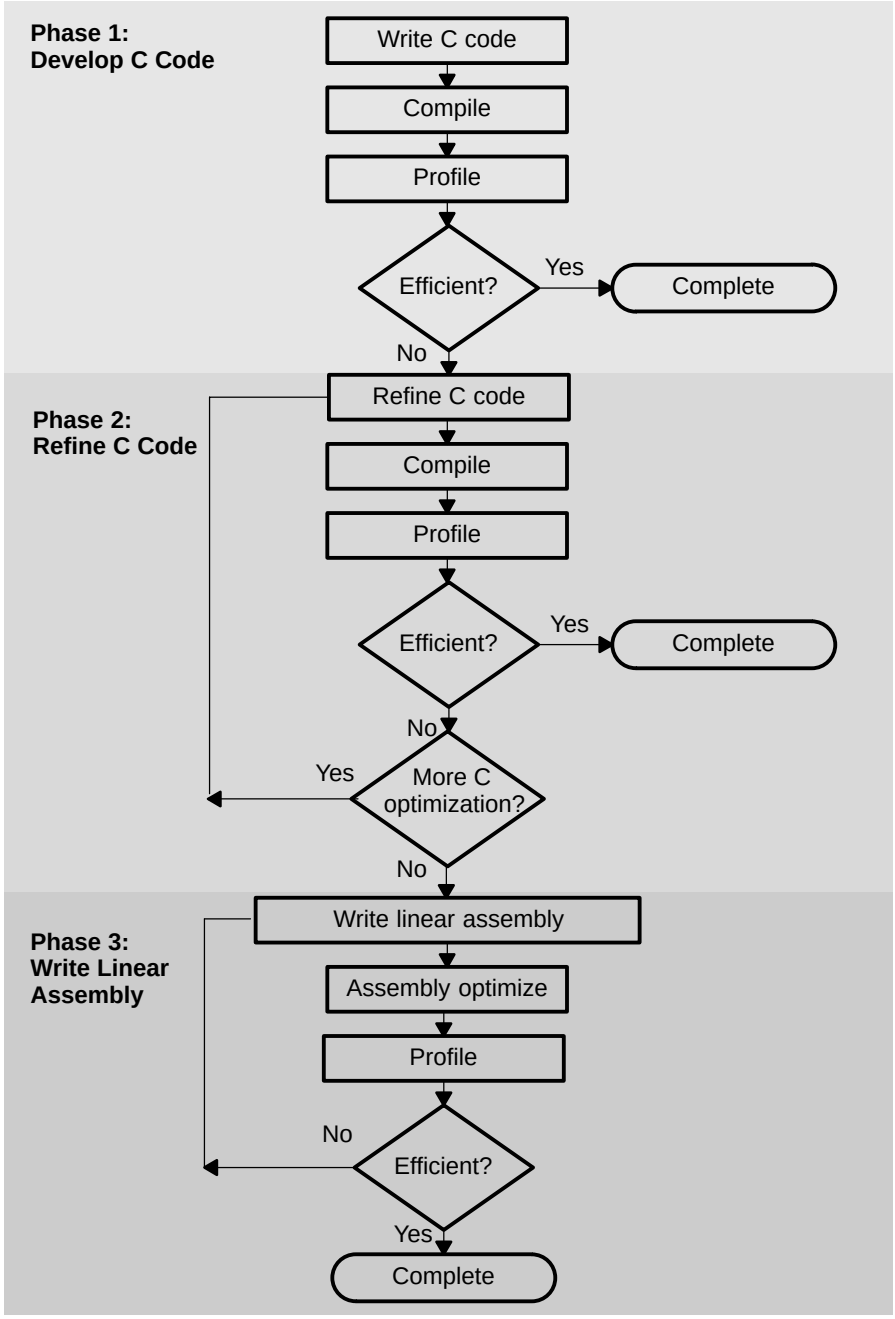
1 Code Development Flow To Increase Performance

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

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

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

You can achieve the best performance from your ’C6000 code if you follow this code development flow when you are writing and debugging your code:



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

Phase	Goal
1	You can develop your C code for phase 1 without any knowledge of the ’C6000. Use the ’C6000 profiling tools that are described in the *Code Composer Studio User’s Guide* to identify any inefficient areas that you might have in your C code. To improve the performance of your code, proceed to phase 2.
2	Use techniques described in this book to improve your C code. Use the ’C6000 profiling tools to check its performance. If your code is still not as efficient as you would like it to be, proceed to phase 3.
3	Extract the time-critical areas from your C code and rewrite the code in linear assembly. You can use the assembly optimizer to optimize this code.

Because most of the millions of instructions per second (MIPS) in DSP applications occur in tight loops, it is important for the ’C6000 code generation tools to make maximal use of all the hardware resources in important loops. Fortunately, loops inherently have more parallelism than non-looping code because there are multiple iterations of the same code executing with limited dependencies between each iteration. Through a technique called software pipelining, the ’C6000 code generation tools use the multiple resources of the VelociTI architecture efficiently and obtain very high performance.

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

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

Table 1. Code Development Steps

	Step	Description
Phase 1	1	Compile and profile native C/C++ code ❑ Validates original C/C++ code ❑ Determines which loops are most important in terms of MIPS requirements.
Phase 2	2	Add restrict qualifier, loop iteration count, memory bank, and data alignment information. ❑ Reduces potential pointer aliasing problems ❑ Allows loops with indeterminate iteration counts to execute epilogs ❑ Uses pragmas to pass count information to the compiler ❑ Uses memory bank pragmas and _nassert intrinsic to pass memory bank and alignment information to the compiler.
	3	Optimize C code using other ’C6000 intrinsics and other methods ❑ Facilitates use of certain ’C6000 instructions not easily represented in C. ❑ Optimizes data flow bandwidth (uses word access for short (’C62x, ’C64x, and ’C67x) data, and double word access for word (’C64x, and ’C67x) data).
Phase 3	4a	Write linear assembly ❑ Allows control in determining exact ’C6000 instructions to be used ❑ Provides flexibility of hand-coded assembly without worry of pipelining, parallelism, or register allocation. ❑ Can pass memory bank information to the tools ❑ Uses .trip directive to convey loop count information
	4b	Add partitioning information to the linear assembly ❑ Can improve partitioning of loops when necessary ❑ Can avoid bottlenecks of certain hardware resources

When you achieve the desired performance in your code, there is no need to move to the next step. Each of the steps in the development involve passing more information to the ’C6000 tools. Even at the final step, development time is greatly reduced from that of hand-coding, and the performance approaches the best that can be achieved by hand.

Internal benchmarking efforts at Texas Instruments have shown that most loops achieve maximal throughput after steps 1 and 2. For loops that do not, the C/C++ compiler offers a rich set of optimizations that can fine tune all from the high level C language. For the few loops that need even further optimizations, the assembly optimizer gives the programmer more flexibility than C/C++ can offer, works within the framework of C/C++, and is much like programming in higher level C. For more information on the assembly optimizer, see the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide* and the *TMS320C6000 Programmer’s Guide* (SPRU198).

In order to aid the development process, some feedback is enabled by default in the code generation tools. Example 1 shows output from the compiler and/or assembly optimizer of a particular loop. The -mw feedback option generates additional information not shown in Example 1, such as a single iteration view of the loop.

Example 1. Compiler and/or Assembly Optimizer Feedback

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

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

2 Writing C/C++ Code

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

2.1 Tips on Data Types

Give careful consideration to the data type size when writing your code. The ’C6000 compiler defines a size for each data type (signed and unsigned):

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

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

- Avoid code that assumes that int and long types are the same size, because the ’C6000 compiler uses long values for 40-bit operations.
- Use the short data type for fixed-point multiplication inputs whenever possible because this data type provides the most efficient use of the 16-bit multiplier in the ’C6000 (1 cycle for “short * short” versus 5 cycles for “int * int”).
- Use int or unsigned int types for loop counters, rather than short or unsigned short data types, to avoid unnecessary sign-extension instructions.
- When using floating-point instructions on a floating-point device such as the ’C6700, use the –mv6700 compiler switch so the code generated will use the device’s floating-point hardware instead of performing the task with fixed point hardware. For example, the RTS floating-point multiply will be used instead of the MPYSP instruction.
- When using the ’C6400 device, use the –mv6400 compiler switch so the code generated will use the device’s additional hardware and instructions.

2.2 Analyzing C Code Performance

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

- One of the preliminary measures of code is the time it takes the code to run. Use the clock( ) and printf( ) functions in C/C++ to time and display the performance of specific code regions. You can use the stand-alone simulator (load6x) to run the code for this purpose. Remember to subtract out the overhead of calling the clock( ) function.

- Use the profile mode of the stand-alone simulator. This can be done by compiling your code with the –mg option and executing load6x with the –g option. The profile results will be stored in a file with the .vaa extension. Refer to the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information.

- Enable the clock and use profile points and the RUN command in the Code Composer debugger to track the number of CPU clock cycles consumed by a particular section of code. Use "View Statistics" to view the number of cycles consumed.

- The critical performance areas in your code are most often loops. The easiest way to optimize a loop is by extracting it into a separate file that can be rewritten, recompiled, and run with the stand-alone simulator (load6x).

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

3 Compiling C/C++ Code

The ’C6000 compiler offers high-level language support by transforming your C/C++ code into more efficient assembly language source code. The compiler tools include a shell program (cl6x), which you use to compile, assembly optimize, assemble, and link programs in a single step. To invoke the compiler shell, enter:

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

For a complete description of the C/C++ compiler and the options discussed in this chapter, see the *TMS320C6000 Optimizing C/C++ Compiler User’s Guide (SPRU187)*.

3.1 Compiler Options

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

The options described in Table 2 are obsolete or intended for debugging, and could potentially decrease performance and increase code size. Avoid using these options with performance critical code.

Table 2. Compiler Options to Avoid on Performance Critical Code

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

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

Table 3. Compiler Options for Performance

Option	Description
–mh<*n*>§ –mhh	Allows speculative execution. The appropriate amount of padding must be available in data memory to insure correct execution. This is normally not a problem but must be adhered to.
–mi<*n*>§ –mii	Describes the interrupt threshold to the compiler. If you know that NO interrupts will occur in your code, the compiler can avoid enabling and disabling interrupts before and after software pipelined loops for a code size and performance improvement. In addition, there is potential for performance improvement where interrupt registers may be utilized in high register presure loops.(See the *TMS320C6000 Programmer's Guide* (SPRU198))
–mt§	Enables the compiler to use assumptions that allow it to be more aggressive with certain optimizations. When used on linear assembly files, it acts like a .no_mdep directive that has been defined for those linear assembly files. (See the *TMS320C6000 Programmer's Guide* (SPRU198))
–o3†	Represents the highest level of optimization available. Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. Various file level characteristics are also used to improve performance.
–op2§	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler. This improves variable analysis and allowed assumptions.
–pm‡	Combines source files to perform program–level optimization.

† Although –o3 is preferable, at a minimum use the –o option.
‡ Use the –pm option for as much of your program as possible.
§ These options imply assertions about your application.

Table 4. Compiler Options That Slightly Degrade Performance and Improve Code Size

Option	Description
–ms0 –ms1	Optimizes primarily for performance, and secondly for code size. Could be used on all but the most performance critical routines.
–oi0	Disables all automatic size–controlled inlining, (which is enabled by –o3). User specified inlining of functions is still allowed.

The options described in Table 5 are recommended for control code, and will result in smaller code size with minimal performance degradation.

Table 5. Compiler Options for Control Code

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

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

The options described in Table 6 provide information, but do not affect performance or code size.

Table 6. Compiler Options for Information

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

3.2 Memory Dependencies

To maximize the efficiency of your code, the 'C6000 compiler schedules as many instructions as possible in parallel. To schedule instructions in parallel, the compiler must determine the relationships, or dependencies, between instructions. Dependency means that one instruction must occur before another, for example, a variable must be loaded from memory before it can be used. Because only independent instructions can execute in parallel, dependencies inhibit parallelism.

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

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

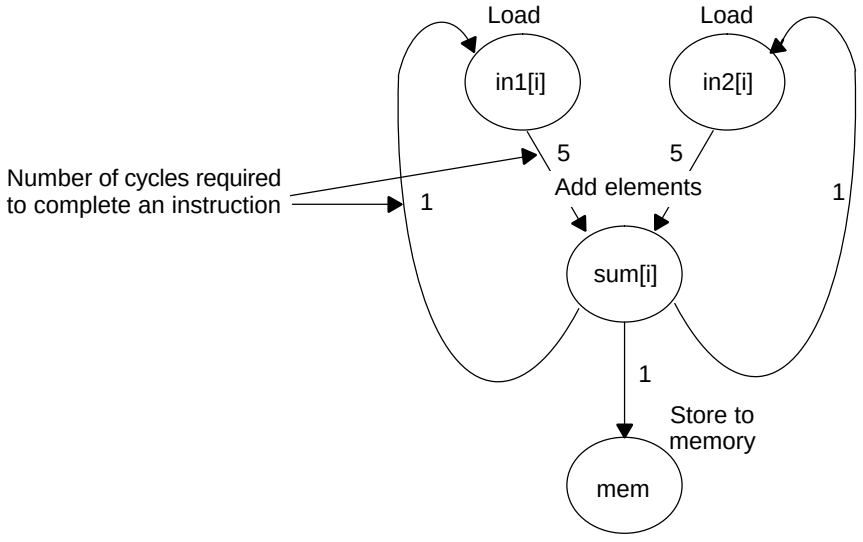
- Use the restrict keyword to indicate that a pointer is the only pointer that can point to a particular object in the scope in which the pointer is declared.
- Use the –pm (program-level optimization) option, which gives the compiler global access to the whole program or module and allows it to be more aggressive in ruling out dependencies.
- Use the –mt option, which allows the compiler to use assumptions that allow it to eliminate dependencies. Remember, using the –mt option on linear assembly code is equivalent to adding the .no_mdep directive to the linear assembly source file. Specific memory dependencies should be specified with the .mdep directive. For more information see section 4.4, *Assembly Optimizer Directives* in the *TMS320C6000 Optimizing C/C++ Compiler User's Guide*.

To illustrate the concept of memory dependencies, it is helpful to look at the algorithm code in a dependency graph. Example 2 shows the C code for a basic vector sum. Figure 1 shows the dependency graph for this basic vector sum. For more information, see *Drawing a Dependency Graph,* in the *TMS320C6000 Programmer's Guide* (SPRU198)

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



The dependency graph in Figure 1 shows that:

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

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

3.2.1 The Restrict Keyword

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

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

Example 3. Use of the Restrict Type Qualifier With Pointers

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

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

Example 4. Use of the Restrict Type Qualifier With Arrays

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

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

Do *not* use the const keyword with code such as listed in Example 5. By using the const keyword in Example 5, you are telling the compiler that it is legal to write to any location pointed to by *a* before reading the location pointed to by *b*. This can cause an incorrect program because both *a* and *b* point to the same object —*array*.

Example 5. Incorrect Use of the restrict Keyword

```
void func (short *a, short * restrict b)/*Bad!! */
{
       int i;
       for (i = 11; i < 44; i++) *(--a) = *(--b);
}
void main ()
{
       short array[] = {   1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                           11, 12, 13, 14, 15, 16, 17, 18,
                           19, 20, 21, 22, 23, 24, 25, 26,
                           27, 28, 29, 30, 31, 32, 33, 34,
                           35, 36, 37, 38, 39, 40, 41, 42,
                           43, 44};
       short *ptr1, *ptr2;

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

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

3.2.2 The –mt Option

Another way to eliminate memory dependencies is to use the –mt option, which allows the compiler to use assumptions that can eliminate memory dependency paths. For example, if you use the –mt option when compiling the code in Example 2, the compiler uses the assumption that that in1 and in2 do not alias memory pointed to by sum and, therefore, eliminates memory dependencies among the instructions that access those variables.

If your code does not follow the assumptions generated by the –mt option, you can get incorrect results. For more information on the –mt option refer to the *TMS320C6000 Optimizing Compiler User's Guide*.

3.3 Performing Program-Level Optimization (–pm Option)

You can specify program-level optimization by using the –pm option with the –o3 option. With program-level optimization, all your source files are compiled into one intermediate file giving the compiler complete program view during compilation. This creates significant advantage for determining pointer locatins passed into a function. Once the compiler determines two pointers do not access the same memory location, substantial improvements can be made in software pipelined loops. Because the compiler has access to the entire program, it performs several additional optimizations rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called, directly or indirectly, the compiler removes the function.

Also, using the –pm option can lead to better schedules for your loops. If the number of iterations of a loop is determined by a value passed into the function, and the compiler can determine what that value is from the caller, then the compiler will have more information about the minimum trip count of the loop leading to a better resulting schedule.

4 Understanding Feedback

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

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

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

1) Qualify the loop for software pipelining

2) Collect loop resource and dependency graph information

3) Software pipeline the loop

4.1 Stage 1: Qualify the Loop for Software Pipelining

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

Example 6. Stage 1 Feedback

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

- **Trip Count**. The number of iterations or trips through a loop.
- **Minimum Trip Count**. The minimum number of times the loop might execute given the amount of information available to the compiler.
- **Maximum Trip Count**. The maximum number of times the loop might execute given the amount of information available to the compiler.

- **Maximum Trip Count Factor**. The maximum number that will divide evenly into the trip count. Even though the exact value of the trip count is not deterministic, it may be known that the value is a multiple of 2, 4, etc..., which allows more agressive packed data and unrolling optimization.

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

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

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

- It cannot have too many instructions in the loop. Loops that are too big, typically require more registers than are available and require a longer compilation time.

- It cannot call another function from within the loop unless the called function is inlined. Any break in control flow makes it impossible to software pipeline as multiple iterations are executing in parallel.

If any of the conditions for software pipelining are not met, qualification of the pipeline will halt and a disqualification messages will appear. For more information about what disqualifies a loop from being software-pipelined, see the *TMS320C6000 Programmer's Guide* (SPRU198).

4.2 Stage 2: Collect Loop Resource and Dependency Graph Information

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

Example 7. Stage 2 Feedback

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

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

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

- **Unpartitioned resource bound across all resources**. The best case resource bound mii before the compiler has partitioned each instruction to the A or B side. In Example 7, the unpartitioned resource bound is 4 because the .S units are required for 8 cycles, and there are 2 .S units.

- **Partitioned resource bound across all resources**. The mii after the instructions are partitioned to the A and B sides. In Example 7, after partitioning, we can see that the A side .L, .S, and .D units are required for a total of 13 cycles, making the partitioned resource bound $\lceil 13/5 \rceil = 5$. For more information, see the description of **Bound** (.L .S .D .LS .LSD) later in this section.

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

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

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

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

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

  - **Logical ops** (.LS) represents the total number of instructions that can use either the .L or .S unit.

  - **Addition ops** (.LSD) represents the total number of instructions that can use either the .L or .S or .D unit.

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

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

  Where ceil represents the ceiling function. This means you always round up to the nearest integer. In Example 7, if the B side needs:

  3 .L unit only instructions

  4 .S unit only instructions

  1 logical .LS instruction

  you would need at least ⌈8/2⌉ cycles or 5 cycles to issue these.

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

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

  Where ceil represents the ceiling function. This means you always round up to the nearest integer. In Example 7, the A side needs:

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

  you would need at least ⌈13/3⌉ cycles or 5 cycles to issue these.

4.3 Stage 3: Software Pipeline the Loop

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

Example 8. Stage 3 Feedback

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

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

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

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

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

If the attempt was not successful, the compiler provides additional feedback to help explain why. In this case, the compiler cannot find a schedule at 11 cycles because register is live too long. For more information about live too long issues, see the *TMS320C6000 Programmer's Guide* (SPRU198).

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

The compiler then proceeds to:

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

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

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

- **Regs Live Always** refers to the number of registers needed for variables to be live every cycle in the loop. Data loaded into registers outside the loop and read inside the loop will fall into this category.
- **Max Regs Live** refers to the maximum number of variable live on any one cycle in the loop. If there are 33 variables live on one of the cycles inside the loop, a minimum of 33 registers is necessary and this will not be possible with the 32 registers available on the ’C62x and ’C67x cores. In addition, this is broken down between A and B side, so if there is uneven partitioning with 30 values and there are 17 on one side and 13 on the other, the same problem will exist. This situation does not apply to the 64 registers available on the ’C64x core.
- **Max Cond Regs Live** tells us if there are too many conditional values needed on a given cycle. The ’C62x and ’C67x cores have 2 A side and 3 B side condition registers available. The ’C64x core has 3 A side and 3 B side condition registers available.

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

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

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

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

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

```
Speculative Load Threshold : 12
```

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

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

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

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

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

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

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

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

This means that the loop must execute at lease twice to safely use the software pipelined version of the loop. If this value is less than the known minimum trip count, two versions of the loop will be generated. For more information on eliminating redundant loops, see the *TMS320C6000 Programmer's Guide* (SPRU198).

5 Feedback Solutions

5.1 Loop Disqualification Messages (Stage 1 Feedback)

Loop disqualification messages are generated as part of Stage 1 feedback when the compiler is qualifying the loop for software pipelining. For more information, see section 4.1, *Stage 1: Qualify the loop for Software Pipelining*, on page 19.

Bad Loop Structure

Description

This error is very rare and can stem from the following:

- An asm statement inserted in the C code innerloop.
- Parallel instructions being used as input to the Linear Assembly Optimizer.
- Complex control flow such as GOTO statements and breaks.

Solution

Remove any asm statements, complex control flow or parallel instructions as input to linear assembly.

Loop Contains a Call

Description

Sometimes the compiler may not be able to inline a function call that is in a loop. Because the compiler could not inline the function call, the loop could not be software pipelined.

Solution

If the caller and the callee are C or C++, use –pm and –op2. See the *TMS320C6000 Opimizing C/C++ Compiler User's Guide* for more information on the correct usage of –op2. Do not use –oi0, which disables automatic inlining.

Add the inline keyword to the callee's function definition.

Too Many Instructions

Loops that are too big typically will not schedule due to too many registers needed and cause a large compilation time in the compiler. The limit on the number of instructions is variable.

Solution

Use intrinsics in C code to select more efficient 'C6000 instructions.

Write code in linear assembly to pick exact 'C6000 instruction to be executed.

For more information...

See *Loop Unrolling* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Optimizing Assembly Code via Linear Assembly* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Software Pipelining Disabled

Software pipelining has been disabled by a command–line option. Pipelining will be turned off when using the –mu option, not using –o2/–o3, or using – ms2/–ms3.

Uninitialized Trip Counter

The trip counter may not have been set to an initial value.

Suppressed to Prevent Code Expansion

Software pipelining may be suppressed because of the –ms1 flag. When the –ms1 flag is used, software pipelining is disabled in less promising cases to reduce code size. To enable pipelining, use –ms0 or omit the –ms flag altogether.

Loop Carried Dependency Bound Too Large

If the loop has complex loop control, try –mh according to the recommendations in the *TMS320C6000 Optimizing C/C++ Compiler User's Guide.*

Cannot Identify Trip Counter

The loop control is too complex. Try to simplify the loop.

5.2 Pipeline Failure Messages

Pipeline Failure messages are generated as part of Stage 3 feedback when the compiler is trying to Software pipeline the loop. For more information, see section 4.3, *Stage 3: Software Pipeline the Loop*, on page 24.

Address Increment Too Large

Description

One thing the compiler does when software pipelining is to allow reordering of all loads and stores occurring from the same array or pointer. This allows for maximum flexibility in scheduling. Once a schedule is found, the compiler then goes back and adds the appropriate offsets and increment/decrements to each load and store. Sometimes, the loads and/or stores end up being offset too far from each other after reordering (the limit for standard load pointers is +/– 32) . If this happens, the best bet is to restructure the loop so that the pointers are closer together or rewrite the pointers to use register offsets that are precomputed.

Solution

Modify code so that the memory offsets are closer.

Cannot Allocate Machine Registers

Description

After software pipelining and finding a valid schedule, the compiler must allocate all values in the loop to specific machine registers (A0–A15 and B0–B15 for the ’C62x and ’C67x, or A0–A31 and B0–B31 for the ’C64x). Sometimes the loop schedule found simply requires more registers than the ’C6000 has available and thus software pipelining that particular ii is not possible. The analyzing feedback example shows:

```
ii = 12 Cannot allocate machine registers
Regs Live Always : 1/5 (A/B-side)
Max Regs Live : 14/19
Max Cond Regs Live : 1/0
```

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

Max Regs Live refers to the maximum number of variable live on any one cycle in the loop. If there are 33 variables live on one of the cycles inside the loop, a minimum of 33 registers is necessary and this will not be possible with the 32 registers available on the C62/C67 cores. 64 registers are available on the 'C64x core. In addition, this is broken down between A and B side, so if there is uneven partitioning with 30 values and there are 17 on one side and 13 on the other, the same problem will exist.

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

Solution

Try splitting the loop into two separate loops. Repartition if too many instructions on one side.

For loops with complex control, try the –mh option.

Use symbolic register names instead of machine registers (A0–A15 and B0–B15 for 'C62x and 'C67x, or A0–A31 and B0–B31 for 'C64x).

For More Information...

See *Loop Unrolling (in Assembly)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Loop Unrolling (in C)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

TMS320C6000 C/C++ Compiler User's Guide (SPRU187)

Cycle Count Too High. Not Profitable

Description

In rare cases, the iteration interval of a software pipelined loop is higher than a non-pipelined list scheduled loop. In this case, it is more efficient to execute the non-software pipelined version.

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

Unpartition/repartition the linear assembly source code.

Add const and restrict keywords where appropriate to reduce dependences.

For loops with complex control, try the –mh option.

Probably best modified by another technique (i.e. loop unrolling).

Modify the register and/or partition constraints in linear assembly.

For more information...

See *Loop Unrolling* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See the *TMS320C6000 C/C++ Compiler User's Guide* (SPRU187).

Did Not Find Schedule

Description

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

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

Unpartition/repartition the linear assembly source code.

Probably best modified by another technique (i.e. loop unrolling).

Modify the register and/or partition constraints in linear assembly.

For more information...

See *Loop Unrolling*, in the *TMS320C6000 Programmer's Guide* (SPRU198).

Iterations in Parallel > Max. Trip Count

Description

Not all loops can be profitably pipelined. Based on the available information on the largest possible trip count, the compiler estimates that it will always be more profitable to execute a non-pipelined version than to execute the pipelined version, given the schedule that it found at the current iteration interval.

Solution

Probably best optimized by another technique (i.e. unroll the loop completely).

For more information...

See *Loop Unrolling (in Assembly) and (In C)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Software Pipelining* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Speculative Threshold Exceeded

Description

It would be necessary to speculatively load beyond the threshold currently specified by the –mh option.

Solution

Increase the –mh threshold as recommended in the software pipeline feedback located in the assembly file.

Iterations in Parallel > Min. Trip Count

Description

Based on the available information on the minimum trip count, it is not always safe to execute the pipelined version of the loop. Normally, a redundant loop would be generated. However, in this case, redundant loop generation has been suppressed via the –ms0/–ms1 option.

Solution

Add MUST_ITERATE pragma or .trip to provide more information on the minimum trip count

If adding –mh or using a higher value of –mhn could help, try the following suggestions:

- ❏ Use –pm program level optimization to gather more trip count information.
- ❏ Use the MUST_ITERATE pragma or the .trip directive to provide minimum trip count information.

For more information...

See *Performing Program Level Optimization (–pm Option)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Communicating Trip Count Information to the Compiler* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *The .trip Directive* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Register is Live Too Long

Description

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

After this message, the compiler prints out a detailed description of which values are live to long:

```
ii = 11 Register is live too long
|72| -> |74|
|73| -> |75|
```

The numbers 72, 73, 74, and 75 correspond to line numbers and they can be mapped back to the offending instructions.

Solution

Use the –mx option for both C code and linear assembly.

Write linear assembly and insert MV instructions to split register lifetimes that are live–too–long.

For more information...

See *Split–Join–Path Problems* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Too Many Predicates Live on One Side

Description

The C6000 has predicate, or conditional, registers available for use with conditional instructions. There are 5 predicate registers on the 'C62x and 'C67x, and 6 predicate registers on the 'C64x. There are two or three on the A side and three on the B side. Sometimes the particular partition and schedule combination, requires more than these available registers.

Solution

Try splitting the loop into two separate loops.

If multiple conditionals are used in the loop, allocation of these conditionals is the reason for the failure. Try writing linear assembly and partition all instructions, writing to condition registers evenly between the A and B sides of the machine. For the ’C62x and ’C67x, if there is an uneven number, put more on the B side, since there are 3 condition registers on the B side and only 2 on the A side.

Too Many Reads of One Register

Description

The ’C62x and ’C67x cores can read the same register a maximum of 4 times per cycle. The ’C64x core can read the same register any number of times per cycle. If the schedule found happens to produce code where a single register is read more than 4 times in a given cycle, the schedule is invalidated. This problem is very rare and only occurs on the ’C67x due to some floating point instructions that have multiple cycle reads.

Solution

Split into multiple loops or reduce the complexity of the loop if possible.

Unpartition/repartition the linear assembly source code.

Probably best modified by another technique (i.e. loop unrolling).

Modify the register and/or partition constraints in linear assembly.

For more information...

See *Loop Unrolling (in Assembly)* in the *TMS320C6000 Programmer’s Guide* (SPRU198).

See *Loop Unrolling (in C)* in the *TMS320C6000 Programmer’s Guide* (SPRU198).

Trip var. Used in Loop – Can’t Adjust Trip Count

Description

If the loop counter (named trip counter because of the number of trips through a loop) is modified within the body of the loop, it typically cannot be converted into a downcounting loop (needed for software pipelining on the ’C6000). If possible, rewrite the loop to not modify the trip counter by adding a separate variable to be modified.

The fact that the loop counter is used in the loop is actually determined much earlier in the loop qualification stage of the compiler. Why did the compiler try

to schedule this anyway? The reason has to do with the –mh option. This option allows for extraneous loads and facilitates epilog removal. If the epilog was successfully removed, the loop counter can sometimes be altered in the loop and still allow software pipelining. Sometimes, this isn't possible after scheduling and thus the feedback shows up at this stage.

Solution

Replicate the trip count variable and use the copy inside the loop so that the trip counter and the loop reference separate variables.

Use the –mh option.

For more information...

See *What Disqualifies a Loop From Being Software Pipelined* in the *TMS320C6000 Programmer's Guide* (SPRU198).

5.3 Investigative Feedback

Investigative feedback is determined by examining the feedback generated using the –mw option. For more information, see section 4, *Understanding Feedback*, on page 19.

Loop Carried Dependency Bound is Much Larger Than Unpartitioned Resource Bound

Description

If the loop carried dependency bound is much larger than the unpartitioned resource bound, this can be an indicator that there is a potential memory alias disambiguation problem. This means that there are two pointers that may or may not point to the same location, and thus, the compiler must assume they might. This can cause a dependency (often between the load of one pointer and the store of another) that does not really exist. For software pipelined loops, this can greatly degrade performance.

Solution

Use –pm program level optimization to reduce memory pointer aliasing.

Add restrict declarations to all pointers passed to a function whose objects do not overlap.

Use –mt option to assume no memory pointer aliasing.

Use the .mdep and .no_mdep assembly optimizer directives.

If the loop control is complex, try the -mh option.

For More Information...

See section 3.3, *Performing Program–Level Optimization (–pm Option)*, on page 18.

See *The const Keyword* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *The restrict Keyword*, on page 15.

See *Memory Dependencies*, on page 13.

See *Memory Alias Disambiguation* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Assembly Optimizer Options and Directives* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Two Loops are Generated, One Not Software Pipelined

Description

If the trip count is too low, it is illegal to execute the software pipelined version of the loop. In this case, the compiler could not guarantee that the minimum trip count would be high enough to always safely execute the pipelined version. Hence, it generated a non-pipelined version as well. Code is generated, so that at run-time, the appropriate version of the loop will be executed.

Solution

Check the software pipeline loop information to see what the compiler knows about the trip count. If you have more precise information, provide it to the compiler using one of the following methods:

- ❏ Use the MUST_ITERATE pragma to specify loop count information in c code.
- ❏ Use the .trip directive to specify loop count information in linear assembly.

Alternatively, the compiler may be able to determine this information on its own when you compile the function and callers with –pm and –op2.

For More Information...

See *Communicating Trip Count Information to the Compiler*, in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *The .trip Directive*, in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Performing Program–Level Optimization (–pm Option)*, on page 18.

Uneven Resources

Description

If the number of resources to do a particular operation is odd, unrolling the loop is sometimes beneficial. If a loop requires 3 multiplies, then a minimum iteration interval of 2 cycles is required to execute this. If the loop was unrolled, 6 multiplies could be evenly partitioned across the A and B side, having a minimum ii of 3 cycles, giving improved performance.

Solution

Unroll the loop to make an even number of resources.

For More Information...

See *Loop Unrolling (in C)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Loop Unrolling (in Assembly)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

Larger Outer Loop Overhead in Nested Loop

Description

In cases where the inner loop count of a nested loop is relatively small, the time to execute the outer loop can start to become a large percentage of the total execution time. For cases where this significantly degrades overall loop performance, unrolling the inner loop may be desired.

Solution

Unroll the inner loop.

Make one loop with the outer loop instructions conditional on an inner loop counter

For More Information

See *Loop Unrolling (In C) (In Assembly)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Outer Loop Conditionally Executed With Inner Loop* in the *TMS320C6000 Programmer's Guide* (SPRU198).

There are Memory Bank Conflicts

Description

In cases where the compiler generates 2 memory accesses in one cycle and those accesses are either 8 bytes apart on a 'C620x device, 16 bytes apart on a 'C670x device, or 32 bytes apart on a 'C640x device, AND both accesses reside within the same memory block, a memory bank stall will occur. To avoid this degradation, memory bank conflicts can be completely avoided by either placing the two accesses in different memory blocks or by writing linear assembly and using the .mptr directive to control memory banks.

Solution

Write linear assembly and use the .mptr directive

Link different arrays in separate memory blocks

For More Information

See *The .mptr Directive* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Loop Unrolling (in Assembly)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Loop Unrolling (in C)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Memory Banks* in the *TMS320C6000 Programmer's Guide* (SPRU198).

T Address Paths Are Resource Bound

Description

T address paths defined the number of memory accesses that must be sent out on the address bus each loop iteration. If these are the resource bound for the loop, it is often possible to reduce the number of accesses by performing word accesses (LDW/STW) for any short accesses being performed.

Solution

Use word accesses for short arrays; declare int * (or use _nassert) and use mpy intrinsics to multiply upper and lower halves of registers

Try to employ redundant load elimination technique if possible

Use LDW/STW instructions for accesses to memory

For More Information...

See *Using Word Accesses for Short Data (C)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Redundant Load Elimination* in the *TMS320C6000 Programmer's Guide* (SPRU198).

See *Using Word Access for Short Data (Assembly)* in the *TMS320C6000 Programmer's Guide* (SPRU198).

6 Tutorial Introduction: Simple C Tuning

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

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

Example 9. Vector Summation of Two Weighted Vectors

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

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

6.1 Project Familiarization

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

Set Up Code Composer Studio for C62x Fast Simulator Little Endian

1) Click on Setup CCStudio to setup the target.

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

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

4) Click on the close button and exit setup.

5) Save the configuration on exit.

Load the Tutorial Workspace

1) Start Code Composer Studio.

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

   Browse to: ti\c6000\examples\cgtools\prog_gd\tutorial\C_tutorial.wks

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

Build tutor.out

From the menu bar, select Project –> Rebuild All

Load tutor.out

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

   Browse to: ti\c6000\examples\cgtools\prog_gd\tutorial\

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

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

Profile the C_tutorial project

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

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

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

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

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

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

You can see cycle counts of 414, 98, 78, and 54 for functions in tutor1–4, running on the C6xxx simulator. Each of these functions contains the same C code but has some minor differences related to the amount of information to which the compiler has access.

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

6.2 Getting Ready for Lesson 1

Compile and rerun the project

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

   All of the files are built with compiler options, –k –mg –mw –mhh –o3 –fr C:ti\c6000\examples\cgtools\prog_gd\tutorial\c_tutorial

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

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

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

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

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

   This clears the Profile Statistics window.

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

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

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

7 Lesson 1: Loop Carry Path From Memory Pointers

Open lesson_c.c

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

Example 10. lesson_c.c

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

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

Compile the project and analyze the feedback in lesson_c.asm

When you rebuilt the project in Getting Ready for Lesson 1, each file was compiled with –k –gp –mh –o3. Because option –k was used, a *.asm file for each *.c file is included in the rebuilt project.

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

2) Select lesson_c.asm and click Open.

   Each .asm file contains software pipelining information. You can see the results in Example 11, Feedback From lesson_c.asm:

Example 11. Feedback From lesson_c.asm

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

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

Q Where are the problems with this loop?

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

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

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

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

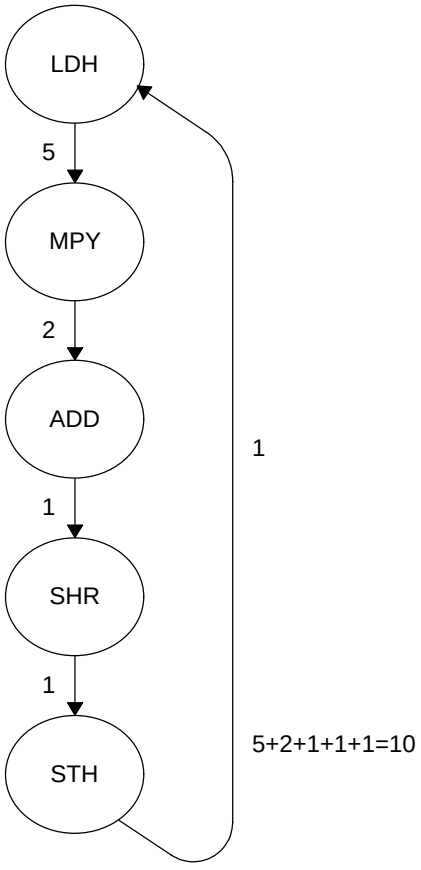
The ^ symbol is interspersed in the assembly output in the comments of each instruction in the loop carry path, and is visible in lesson_c.asm.

Example 12. lesson_c.asm

```
L2:     ; PIPED LOOP KERNEL

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

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

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

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

You can also use a dependency graph to analyze feedback, as in Figure 2.

Figure 2. Dependency Graph for Lesson_c.c



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

A If we look at the original C code in Example 10 (lesson_c.c), we see that the LDHs correspond to loading values from xptr and yptr, and the STH corresponds to storing values into w_sum array.

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

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

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

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

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

A The next example, lesson1_c provides the answer:

Open lesson1_c.c and lesson1_c.asm

Example 13. lesson1_c.c

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

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

}
```

The only change made in lesson1_c is the addition of the restrict type qualifier for xptr and yptr. Since we know that these are actually separate arrays in memory from w_sum, in function lesson1_c, we can declare that nothing else points to these objects. No other pointer in lesson1_c.c points to xptr and no other pointer in lesson1_c.c points to zptr. See the *TMS320C6000 Optimizing C/C++ Compiler User's Guide* for more information on the restrict type qualifier. Because of this declaration, the compiler knows that there are no possible dependency between xptr, yptr, and w_sum. Compiling this file creates feedback as shown in Example 14, lesson1_c.asm:

Example 14. lesson1_c.asm

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

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

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

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

Rebuild lesson_c.c using the –mt option

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

   The Build Options dialog window appears.

2) Select the Compiler tab.

3) In the Category box, select Advanced.

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

   The -mt option will appear in the options window.

5) Click OK to set the new options.

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

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

   If prompted, reload lesson_c.asm.

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

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

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

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

   You now see the following:

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

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

Table 7. Status Update: Tutorial example lesson_c lesson1_c

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

8 Lesson 2: Balancing Resources With Dual-Data Paths

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

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

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

Open lesson1_c.asm

Example 15. lesson1_c.asm

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

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

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

Q Is it possible to improve the balance of resources?

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

Q Why did the compiler not unroll the loop?

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

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

Open lesson2_c.c

Example 16. lesson2_c.c

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

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

}
```

In lesson2_c.c, no code is altered, only additional information is passed via the MUST_ITERATE pragma. We simply guarantee to the compiler that the trip count (in this case the trip count is N) is a multiple of two and that the trip count is greater than or equal to 20. The first argument for MUST_ITERATE is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument. For more information about the MUST_ITERATE pragma, see the *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187).

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

Open lesson2_c.asm and examine the feedback

Example 17. lesson2_c.asm

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

Notice the following things in the feedback:

Loop Unroll Multiple: 2x: This loop has been unrolled by a factor of two.

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

A Known Minimum Trip Count of 10: This is because we specified the count of the original loop to be greater than or equal to twenty and a multiple of two

and after unrolling, this is cut in half. Also, a new line, Known Maximum Trip Count, is displayed in the feedback. This represents the maximum signed integer value divided by two, or 3FFFFFFFh.

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

Q Is this the lower limit?

A Check out Lesson 3 to find out!

Table 8. Status Update: Tutorial example lesson_c lesson1_c lesson2_c

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

9 Lesson 3: Packed Data Optimization of Memory Bandwidth

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

Open lesson2_c.asm

Example 18. lesson2_c.asm

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

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

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

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

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

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

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

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

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

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

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

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

Open lesson3_c.c

Example 19. lesson3_c.c

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

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

    WORD_ALIGNED(xptr);
    WORD_ALIGNED(yptr);

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

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

Open lesson3_c.asm

Example 20. lesson3_c.asm

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

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

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

Table 9. Status Update: Tutorial example lesson_c lesson1_c lesson2_c lesson3_c

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

10 Lesson 4: Program Level Optimization

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

Q Is this necessary in all cases?

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

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

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

Enable program level optimization

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

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

View profile statistics

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

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

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

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

   The new profile statistics should appear in the Profile Statistics window, as in Example 21.

Example 21. Profile Statistics

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

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

Table 10. Status Update: Tutorial example lesson_c lesson1_c lesson2_c lesson3_c

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

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

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

11 Lesson 5: Writing Linear Assembly

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

Linear assembly is similar to regular 'C6000 assembly code in that you use 'C6000 instructions to write your code. With linear assembly, however, you do not need to specify all of the information that you need to specify in regular 'C6000 assembly code. With linear assembly code, you have the option of specifying the information or letting the assembly optimizer specify it for you. Here is the information that you do *not* need to specify in linear assembly code:

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

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

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

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

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

In order to follow this example in Code Composer Studio, you must open the ccs project , l_tutorial.pjt, located in c:\ti\tutorial\sim62xx\linear_asm. Build the program and look at the software pipeline information feedback in the generated assembly files.

Example 22. Using the iircas4 Function in C

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

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

    #pragma MUST_ITERATE(10);

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

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

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

Example 23 shows the assembly output from Example 22

Example 23. Software Pipelining Feedback From the iircas4 C Code

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

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

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

Notice that there are 5 cross path reads on the A side and only 3 on the B side. We would like 4 cross path reads on the A side and 4 cross path reads on the B side. This would allow us to schedule at an iteration interval (ii) of 4 instead of the current ii of 5. Example 24 shows how to rewrite the iircas4 ( ) function Using Linear Assembly.

Example 24. Rewriting the iircas4 ( ) Function in Linear Assembly

```
        .def    _iircas4_sa

_iircas4_sa:    .cproc  AI,C,BD,AY

        .no_mdep

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

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

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

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

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

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

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

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

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

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

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

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

        .endproc
```

The following example shows the software pipeline feedback from Example 24.

Example 25. Software Pipeline Feedback from Linear Assembly

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

Notice in Example 24 that each instruction is manually partitioned. From the software pipeline feedback information in Example 25, you can see that a software pipeline schedule is found at ii = 4. This is a result of rewriting the iircas4 ( ) function in linear assembly, as shown in Example 24.