Application Report
SPRABF2–October 2011

Introduction to TMS320C6000 DSP Optimization

Paul Yin

ABSTRACT

The TMS320C6000™ Digital Signal Processors (DSPs) have many architectural advantages that make them ideal for computation-intensive real-time applications. However, to fully leverage the architectural features that C6000™ processors offer, code optimization may be required. First, this document reviews five key concepts in understanding the C6000 DSP architecture and optimization. Then, it highlights the five most effective code optimization techniques that developers can apply to a range of applications to meet their performance targets.

Contents

List of Figures

List of Tables

1 Introduction

The C6000 DSPs have many architectural advantages that make them ideal for computation-intensive real-time applications. Figure 1 shows that the family consists of fixed point DSP cores such as C64x™ and C64x+™, early floating/fixed point DSP cores such as C67x™ and C67x+, and the newest floating/fixed point DSP cores such as C674x and C66x. All the DSP cores share the same high-level architecture, with hardware enhancements made and additional instructions supported on the newer devices. Moreover, applications developed on older devices are 100% upward compatible.

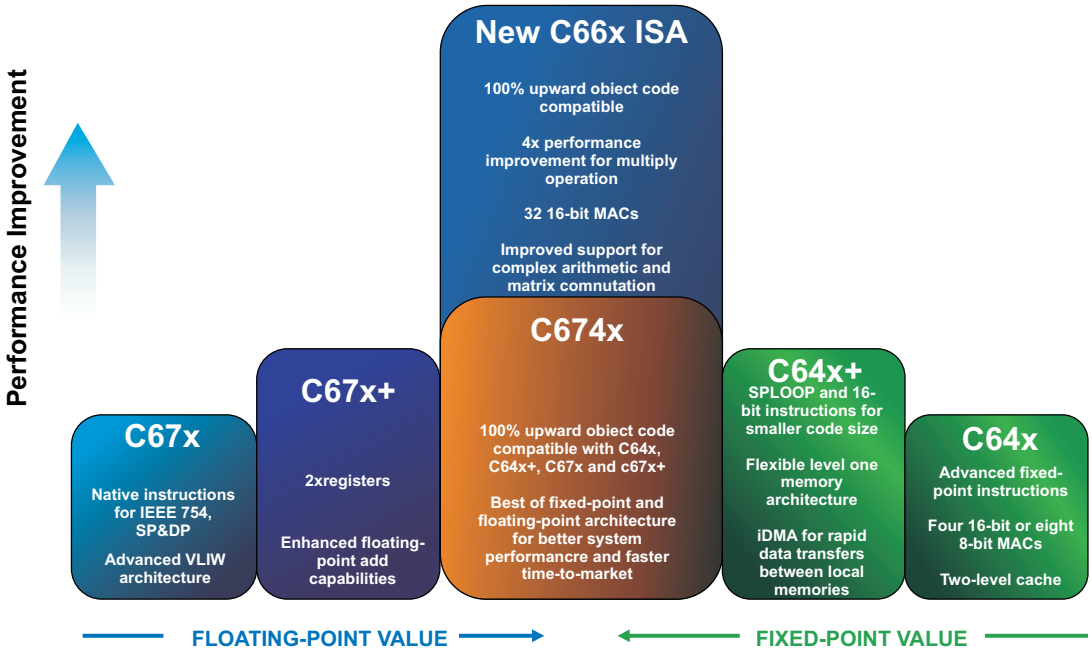

Figure 1. The C6000 DSP Cores

The C674x core is a combination of the C64x+ and C67x+ cores. The latest generation C66x core is an enhanced version of the C674x core. The C674x core is capable to operate at 1.25 GHz, and perform 10,000 million complex multiply-and-accumulate operations; the C66x core is capable to operate at 1.25 GHz, and perform 40,000 million complex multiply-and-accumulate operations. However, to fully utilize the *horsepower* that is offered, optimization may be needed.

For application development and optimization, first time developers will likely be concerned with the following two questions:

- How easy is it to program the C6000 DSP?
- How easy is it to optimize applications on the C6000 DSP?

The answer to these questions varies depending on the level of optimization that is required.

The C6000 code generation tools support the following programming languages: ANSI C (C89), ISO C++ (C++98), C6000 DSP assembly, and C6000 linear assembly (a TI developed, simplified assembly language that is similar to the standard C6000 DSP assembly language). Figure 2 shows that the effort needed for development and optimization increases significantly from C/C++ to DSP assembly. On the other hand, there may be only marginal performance improvement. The highly optimized C/C++ compiler that comes with the C6000 code generation tools is capable of generating assembly code that is very close in performance to the most optimized hand-written assembly code. Thus, for most applications, programming and optimizing in C/C++ code is efficient enough to meet the performance requirements of the application. Therefore, it is recommended that most applications be developed in C/C++.

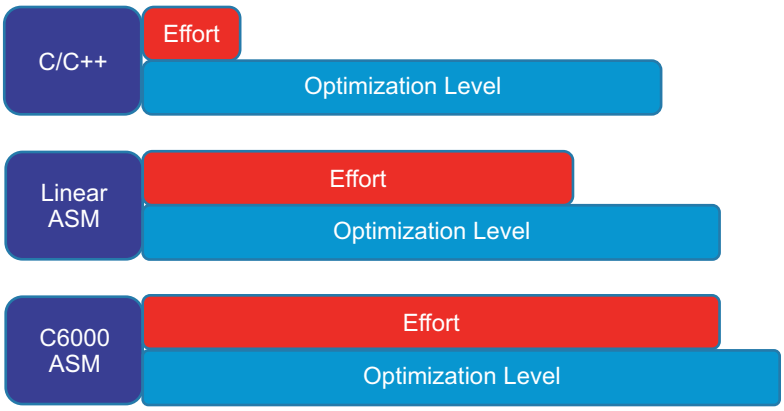

Figure 2. Optimization Level vs. Effort Required

Now, to answer the first question, programming in C/C++ on the C6000 DSP is as easy as programming on any other processor. Furthermore, for existing applications that are already developed in C/C++ language, the effort to port the code to the C6000 DSP is minimal.

For the second question, optimizing C/C++ code on the C6000 DSP is straightforward.

The primary purpose of this document is to elaborate on optimizing C code. Section 2 explains the five key concepts needed for first time developers to quickly understand the C6000 DSP and C6000 DSP optimization. Section 3 elaborates on the five most effective yet simple optimization techniques, which are often sufficient for developers to meet their performance target. The techniques covered in Section 3 are also effective in optimizing C++ code, with minor syntactic differences. Additional considerations on optimizing C++ code are beyond the scope of this document.

The document is intended to provide first time DSP developers a simple path into C6000 DSP optimization, but it is not meant to be a comprehensive programming guide. For advanced information on the C6000 architecture, see references [3] and [4]. For additional optimization techniques and reference on programming in linear or DSP assembly, see references [1] and [5].

2 Understanding the C6000 DSP: Five Key Concepts

The architecture that gives the C6000 DSP its industry leading performance consists of a DSP core designed for parallel processing and a DSP pipeline designed for high throughput. Therefore, an optimized DSP application should utilize the DSP core and pipeline as fully as possible.

Section 2 is structured as follows:

- Section 2.1 introduces the C6000 DSP core that is designed for parallel processing.
- Section 2.2 introduces the C6000 DSP pipeline that is designed for high throughput.
- Section 2.3 introduces the concept of software pipelining, which is an instruction scheduling method to ensure the full utilization of the pipeline.
- Section 2.4 introduces the concept of compiler optimization, a simple method that discusses how programmers can guide the C6000 C compiler through the optimization process by providing more information about the application.
- Section 2.5 introduces the intrinsic operations supported by the C6000 C compiler and the intrinsic-based, highly optimized software libraries that support the C6000 DSPs. Moreover, this section introduces the concept of inlined functions.

2.1 Concept I: C6000 Core

2.1.1 A Simplified Load/Store Architecture

Figure 3 illustrates a simplified load/store architecture, which consists of a processing unit that handles the execution of all instructions, and many registers that hold data as operands or results for these instructions. All the instructions to be executed are fetched from the memory to the processing unit sequentially (the address of the instruction is stored in the program counter register). Data can be loaded from the memory to the registers via a *load* instruction; and register contents can be stored back to the memory via a *store* instruction.

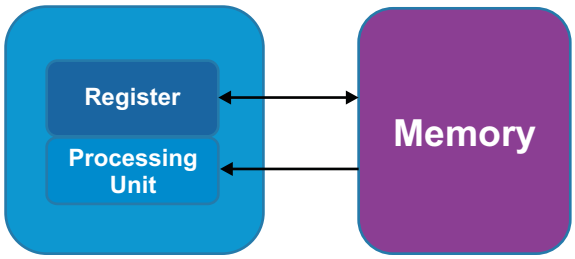

Figure 3. A Simplified Load/Store Architecture

2.1.2 Performing a Multiply-and-Accumulate Function

One of the most common operations in digital signal processing is the multiply-and-accumulate operation, which is used in algorithms like an FIR filter or FFT. Example 1 shows a multiply-and-accumulate loop written in C language.

Example 1. A 16-Bit Multiply-and-Accumulate Loop Example in C

```
for (i=0; i < count; i++)
{
       prod = m[i] * n[i];
       sum += prod;
}
```

Example 2 shows an alternative expression of the loop in pseudo assembly language.

Example 2. The Multiply-and-Accumulate Loop Example in Pseudo Assembly

```
/*
1. Pointer_m & pointer_n represent registers used to hold the address of the data in memory.
2. Data_m, data_n, data_product, data_sum, and data_count represent registers used to hold
   actual data.
3. The branch instruction (with a condition enabled) tells the CPU to jump back to the top of the
   loop (loop_top) if the value in the data_count register is not zero.
*/

loop_top:             load       *pointer_m++, data_m
                      load       *pointer_n++, data_n
                      multiply   data_m, data_n, data_prod
                      add        data_prod, data_sum, data_sum
                      subtract   data_count, 1, data_count
if (data_count != 0)  branch     loop_top
```

Assuming the code in Example 2 is executed on the simple CPU core mentioned in Section 2.1.1, during each loop iteration the processing unit needs to complete six instructions, one instruction at a time. To complete the entire loop, it will take this simple CPU 6*count instruction cycles. In this section, an instruction cycle is defined as the time it takes the CPU to complete 1 instruction.

2.1.3 The C6000 DSP Core

It can be seen that the instructions in Example 2 require different hardware logic to execute. When the instructions are executed on the simple CPU in Figure 3 one at a time, only a portion of the logic is in use on a given cycle and other logic is idle. One way to improve the performance is to partition the processing unit in Figure 3 into smaller specialized functional units that can work simultaneously. Each functional unit handles a subset of the instructions. This way, without significantly increasing the size of the hardware logic, the CPU core is capable to handle multiple instructions at the same time.

The C6000 DSP core, as shown in Figure 4, is designed based on this principle

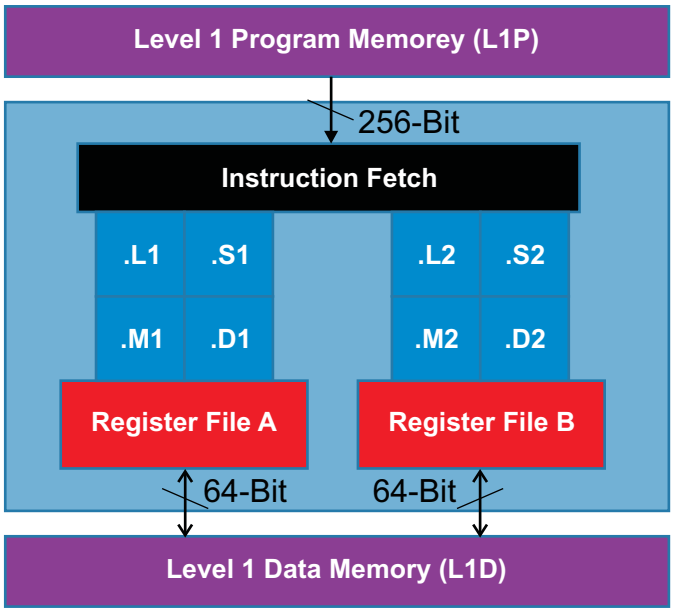

Figure 4. High Level C6000 DSP Architecture

At a high level, all the C6000 DSPs share the same architecture. As shown in Figure 4, there are:

- 8 parallel functional units that can execute instructions in parallel with no interdependency (.D1, .D2, .S1, .S2, .M1, .M2, .L1, and .L2)
  - .D (.D1 and .D2) handles data load and store operations
  - .S (.S1 and .S2) handles shift, branch, or compare operations
  - .M (.M1 and .M2) handles multiply operations
  - .L (.L2 and .L2) handles logic and arithmetic operations
  - All of these functional units, besides the main function they support, also support additional instructions. For details on each functional unit, see reference [3].
- 32 32-bit registers for each side of functional units
  - A0-A31 for A side (.D1, .S1, .M1, and .L1)
  - B0-B31 for B side (.D2, .S2, .M2, and .L2)
- Separated program and data memory (L1P and L1D) running at the same rate as the C6000 core
  - This enables fetching of program and data simultaneously.
- 256-bit internal program bus that allows the C6000 core to fetch 8 32-bit instructions from L1P every cycle.

- 2 64-bit internal data buses that allows both .D1 and .D2 to fetch data from L1D every cycle

2.1.4 Performance of the C6000 DSP Core

The C6000 C compiler is capable of generating the following assembly code, as shown in Example 3, from the C code in Example 1. The understanding of the syntax is beyond the scope of this document. For advanced information, see reference [3].

Example 3. The Multiply-and-Accumulate Loop Example in C6000 Assembly

```
     //Instructions   //Function Unit        //Registers
Loop:    ldw                .D1               *A4++,A5
 ||      ldw                .D2               *B4++,B5
 ||[B0]  sub                .S2               B0,1,B0
 ||[B0]  B                  .S1               loop
 ||      mpy                .M1               A5,B5,A6
 ||      mpyh               .M2               A5,B5,B6
 ||      add                .L1               A7,A6,A7
 ||      add                .L2               B7,B6,B7
```

The code in Example 3 has been optimized by the C compiler to fully utilize the C6000 core. Instead of performing one multiply-and-accumulate, it actually performs two multiply-and-accumulate operations. When the code is executed on the C6000 CPU, the C6000 CPU core can process all 8 instructions in parallel (parallel instructions are indicated by the "||" symbol in front of the assembly op-codes), resulting in all 8 instructions executing in 1 instruction cycle. Because two multiply-and-accumulate operations are done every loop iteration, it only takes the C6000 CPU count/2 instruction cycles to complete the loop.

2.2 Concept II: C6000 Pipeline

The simple CPU in Section 2.1 is assumed to be able to fetch and execute a given instruction in one instruction cycle, and the CPU can start processing the next instruction upon the completion of the previous one. Similarly, each functional unit in the C6000 CPU can fetch and execute a given instruction in one instruction cycle, and the functional unit starts processing the next instruction upon the completion of the previous one.

There is one limitation of this model that needs to be addressed.

The processing of one instruction is not a one-step operation. The three steps are: the fetch stage, the decode stage, and the execute stage.

The CPU needs to:

1. Obtain the instruction from the memory.
2. Analyze the instruction so it knows what to do.
3. Perform the operation.

Each stage requires its own logic to complete its functionality. In the previous model, because the CPU only processes 1 instruction at a time, during the fetch stage, the decode and execute logic are left idle; during the decode stage, the fetch and execute logic are left idle; and during the execute stage, the fetch and decode logic are left idle.

To maximize performance, all of the hardware support that is available should be utilized. In this case, how can you utilize the idle logic to process more instructions and maximize the CPU performance?

The answer is pipelining multiple instructions.

On the C6000 DSP, in addition to the highly parallel architecture, the C6000 pipeline is also designed to maximize the performance of the C6000 DSPs.

2.2.1 Introduction to Pipelining

Figure 5, in which each stage is assumed to take exactly 1 clock cycle, illustrates the concept and advantage of pipelining.

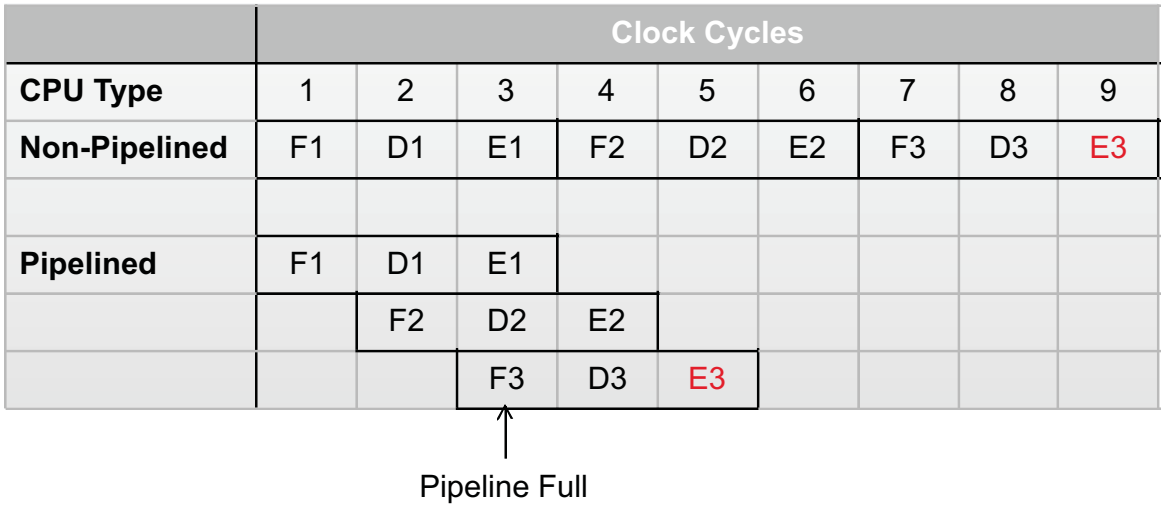

Figure 5. A Simple Pipeline

If each instruction can only be processed when the previous instruction is completed, the CPU can only process one instruction every 3 cycles. The concept of pipelining is basically to fetch the second instruction as soon as the fetching of the first is completed; and fetch the third instruction while the first instruction has entered the execute stage and the second instruction has entered the decode stage. With this approach, even though the processing time of each individual instruction still takes 3 CPU cycles, it only takes the CPU n + 2 cycles to process n instructions. Thus, the effective throughput of the pipeline is 1 cycle per instruction.

By applying pipelining, without increasing the clock speed or implementing additional functional units, the throughput of the CPU has been tripled.

2.2.2 The C6000 Pipeline

In reality, on a CPU with a clock rate that is on the order of hundreds of MHz, each of the three stages takes multiple cycles to complete. Therefore, to further improve the throughput of the CPU, the fetch, decode, and execute stages have been divided into sub-stages on the C6000 architecture; and each sub-stage takes 1 CPU cycle to complete. The stages are shown as follows:

- 4-stage fetch
  - PG: Program address generate (update program counter register)
  - PS: Program address send (to memory)
  - PW: Program (memory) access ready wait
  - PR: Program fetch packet receive (fetch packet = eight 32-bit instructions)
- 2-stage decode (DP and DC)
  - DP: Instruction dispatch (or assign, to the functional units)
  - DC: Instruction decode

- Up to 10-stage, variable duration execute
  - E1 – E10, where E1 is the first sub stage in the execute stage
  - Different instructions might require different number of stages to complete its execution

The operation of the C6000 pipeline is illustrated in Figure 6. The pipeline operation is most efficient when all the sequential instructions proceed to the next logical stage every clock cycle. For more detailed information on the C6000 pipeline, see reference [3].

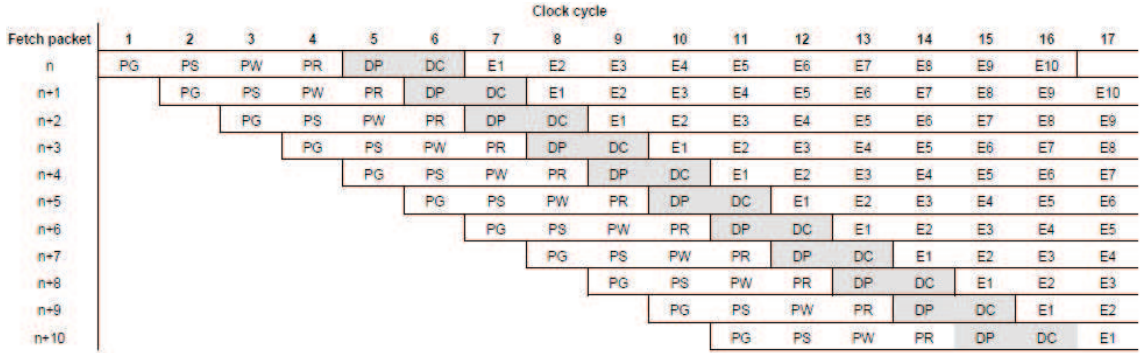

Figure 6. High Level C6000 DSP Pipeline

2.2.3 Delay Slots and the Need for Software Pipelining

In the following two cases, the pipeline will not be able to maintain the optimal mode of operation:

- The current instruction is dependent on the result of the previous instruction and it takes more than 1 cycle for the result to be ready.
  - Example 1: The result of a load takes 5 cycles to come back. Therefore, after issuing the load (into E1), the CPU has to wait for 4 cycles before it can issue the next instruction that uses the value loaded from the memory.
  - Example 2: The result of a single-precision floating-point multiplication takes 4 cycles to come back. Therefore, after issuing the multiplication, the CPU has to wait for 3 cycles before it can issue the next instruction.
- A branch is performed.
  - The CPU will not know where the next instruction is until it executes the branch instruction, I.e., the branch target does not enter PG until branch is in E1. As a result, the CPU has to wait for 5 more cycles before the branched-to instruction enters the execute stage.

The cycles that the CPU has to wait on are the delay slots of the instructions. Therefore, a load has four delay slots, a single-precision floating-point multiplication has three delay slots, and a branch has five delay slots. In loop operations, these delay slots can cause significant performance degradation.

To fill the delay slots and fully utilize the pipeline, a software scheduling technique can be used. Moreover, enhancements are made to the C6000 architecture to support this technique. The software scheduling technique to fill the delay slots is called software pipelining, and the hardware enhancement is called the SPLOOP buffer.

These topics are covered in Section 2.3.

2.3 Concept III: Software Pipelining

2.3.1 Software Pipeline

Software pipelining is a technique to schedule instructions from a loop so that multiple iterations of the loop can execute in parallel and the pipeline can be utilized as fully as possible. The effectiveness of software pipelining can be demonstrated by Example 4 and Figure 7.

Example 4. Adding 15 Values in Memory

```
Solution 1:   Traditional scheduling: repeat the following code 15 times
              Load                    *pointer_value++, data_value
              add                     data_value, data_sum, data_sum
Solution 2:   Software pipelining

Note: The branch instruction is omitted to simplify the example.
Note: Because a load instruction has 4 delay slots, the add instruction must be executed 5 cycles
after its corresponding load (if load had 0 delay slot, an add would be executed 1 cycle after the
load).
```

Solution 1				Solution 2			
Clock	.D1	.L1		Clock	.D1	.L1	
1	Load1			1	Load1		Prolog
2				2	Load2		
3				3	Load3		
4				4	Load4		
5				5	Load5		
6		Add1		6	Load6	Add1	Kernel
7	Load2			7	Load7	Add2	
8				8	Load8	Add3	
9				9	Load9	Add4	
10				10	Load10	Add5	
11				11	Load11	Add6	
12		Add2		12	Load12	Add7	
13	Load3			13	Load13	Add8	
14				14	Load14	Add9	
15				15	Load15	Add10	
16				16		Add11	Epilog
17				17		Add12	
18		Add3		18		Add13	
19	Load4			19		Add14	
20				20		Add15	

Figure 7. Traditional Scheduling vs. Software Pipelining

In solution 1, a non-optimized scheduling method is applied. It takes the CPU 6 clock cycles to complete each load-and-add. For the entire operation, it takes the CPU 90 clock cycles to complete. There is no parallelism applied at all, and the hardware pipeline is unutilized most of the time.

In solution 2, software pipelining is applied. Immediately after executing load1, load2 is executed, and one more load is executed every clock cycle afterwards. On the other hand, add1 is indeed executed on the 6th clock cycle, and one more add is executed every clock cycle in parallel with a load from 5 iterations before. From a code execution point of view, similar to what is in solution 1, an add is always executed 5 cycles after its corresponding load, but it only takes the CPU 20 clock cycles to complete the entire operation.

The trick that enables such a performance improvement happens mainly in the section in green, where a load from one iteration of the loop and an add from another iteration of the loop are executed in parallel. This section in green, in which the pipeline is *fully* utilized, is called the kernel of the software pipelined loop. The section in red, which happens immediately before the kernel, is called the prolog, and the section in yellow, which happens immediately after the kernel, is called the epilog.

NOTE: It is important to know that the instructions of the loop code in solution 1 (i.e., *load (n)* and then *add (n)*) and solution 2 (i.e., *load (n+5)* || *add (n)*) are identical. However, instructions in solution 2 are from different iterations of the code, and can be executed in parallel, reducing CPU cycle count.

As shown in Figure 7, software pipelining is very effective. However, implementing software pipelining manually, even in the simplest case, can be a very difficult and error-prone task. For a program developed in C, this task is much easier. When the compiler option, *–o3*, is used, the C6000 C compiler is able to automatically perform software pipelining on the loop code. For more information on the compiler options, see Section 2.4.2 and Section 3.1.

2.3.2 Software Pipelined Loop (SPLOOP) Buffer

One major drawback of software pipelining is that it increases the size of the generated assembly file and can affect interruptibility. The SPLOOP buffer is implemented on the C64x+, C674x, and C66x cores to address these issues.

The SPLOOP buffer stores a single scheduled iteration of the loop in a specialized buffer, along with the correct timing dependencies. The hardware selectively overlays copies of the single scheduled iteration in a software pipelined manner to construct an optimized execution of the loop.

In most cases, the C compiler can automatically utilize the SPLOOP buffer while performing software pipelining. However, the SPLOOP buffer cannot be used to handle loops that exceed 14 execute packets (An execute packet consists of up to eight instructions that can be executed in parallel). For complex loops, such as nested loops, conditional branches inside loops, and function calls inside loops, the effectiveness of the compiler may be compromised. When the situation becomes too complex, the compiler might not be able to pipeline at all. Therefore, it is recommended to avoid implementing long or complex loops in C.

For more information on SPLOOP buffer, see references [1] and [3].

2.4 Concept IV: Compiler Optimization

The concept of compiler optimization is using the C compiler to generate assembly source files that utilize the C6000 core functional units and the C6000 DSP pipeline as fully as possible. The process of compiler optimization is generally sufficient for most applications to reach their performance targets.

2.4.1 Introduction to the C6000 C Compiler

As mentioned before, it is recommended to develop the application in C for its simplicity to develop and update, and use the C6000 code generation tools to generate highly optimized assembly source files and final executable. The C Compiler in the C6000 code generation tools serve the purpose to both generate and optimize the C6000 assembly files. The compiler supports a vast amount of configurable options, including those related to optimization and debugging. For the entire list of the configurable options, see reference [2].

2.4.2 Guiding the Compiler

In general, the compiler is capable of generating highly optimized code. However, in many cases, the compiler, in order to assure the correct operation of the code, must take a conservative approach. For such cases, additional information and instructions can be provided by the developer to guide the compiler to maximally optimize the code.

Common methods to guide the compiler during optimization are:

- Compiler options that control optimization
  - These options control how the compiler processes all the input source files.
  - For example, the *--opt_level* option controls how much optimization the compiler should perform on the source code. This topic is further discussed in Section 3.1.
- Keywords
  - The C6000 C compiler supports several ANSI C keywords and C6000 special keywords. These keywords, appended before the function or object they aim to qualify, tell the compiler how to treat these functions or objects.
  - For example, the *restrict* keyword represents an assurance that, within the scope of the pointer declaration, the object pointed to can be accessed only by that pointer. This topic is further discussed in Section 3.2.
- Pragma directives.
  - Pragma directives tell the compiler how to treat a specific function, object, or section of code. Many pragmas can help the compiler further optimize the source code.
  - For example, the *MUST_ITERATE* pragma, when placed right before a loop operation, tells the compiler the minimum iteration count of the loop, the maximum iteration count, and the divisibility of the cycle count. This topic is further discussed in Section 3.3.

2.4.3 Understanding Compiler Feedback

The methods in Section 2.4.2 can be very effective in guiding the compiler to generate highly optimized code. But how does the programmer know when to provide such guidance and what kind of guidance the compiler needs, so that the highest performance can be obtained?

The answer is compiler feedback, a way of telling the programmer what the compiler has done, including information such as the obstacle that it has encountered and the optimization result.

In most DSP applications, the CPU spends the majority of its time handling various loop operations. During the optimization process the most important part is loop optimization. On the C6000 DSP, loop performance is greatly affected by how well the compiler can software pipeline. Therefore, the most critical portion of the compiler feedback is the loop feedback.

The compiler goes through a 3-stage process when optimizing a loop. These three stages are:

1. Qualify the loop for software pipelining.
2. Collect loop resource and dependency information.
3. Software pipeline the loop.

A simplified loop feedback is provided in Example 5 (actual source code not included as it is not needed for understanding the feedback).

Example 5. Software Pipeline Information Feedback

```
;* SOFTWARE PIPELINE INFORMATION

// Stage 1: Loop Qualification (Section 2.4.3.1)

;* Loop1 : LOOP
;* Known Minimum Trip Count            : 1
;* Known Max Trip Count Factor         : 1

// Stage 2: Dependency and Resource Information Collection (Section 2.4.3.2)

;* Loop Carried Dependency Bound (^)   : 3
;* 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*

// Stage 3: Software Pipeline (Section 2.4.3.3)

;* Searching for software pipeline schedule at …
;*        ii = 4 Schedule found
;* done
;*
;* Minimum safe trip count             : 2
;*
:* --------------------------------------------------------------------------*
```

NOTE: In order to view the feedback, the developer must use the -k option, which retains the assembly output from the compiler. Additional feedback can be generated with the -mw option.

2.4.3.1 Stage 1 – Loop Qualification

For most loop code, the number of loop iterations, or the trip count, can only be determined during run time. To optimize such loop code as much as possible, the compiler needs information such as the minimum (possible) trip count, the maximum (possible) trip count, and the maximum trip count factor. The maximum trip count factor is the largest integer that can evenly divide any possible trip count. In the loop qualification stage, the compiler tries to collect such information.

As shown in Example 5, the minimum trip count determined is 1, the maximum trip count cannot be determined (this is generally the case because the upper limit is often unknown), and the maximum trip factor determined is 1. This information indicates that the compiler assumes that the loop can iterate any number of times. Therefore, it has to take a more conservative approach during optimization. However, more information can be provided with the *MUST_ITERATE* pragma that is mentioned in Section 2.4.2. For example, if the programmer knows the loop will iterate at least 20 times, or if the loop count is always a factor of 2, then the compiler might be able to take a more aggressive approach in performing software pipelining or to invoke other techniques such as loop unrolling. Loop unrolling will be explained in detail in Section 3.3.2.

2.4.3.2 Stage 2 – Dependency and Resource Information

The main goal in loop optimization is to minimize the iteration interval (ii). The iteration interval is the number of cycles it takes the CPU to complete one iteration of the parallel representation of the loop code. The overall cycle count of the software pipelined loop can be approximated with ii * number_of_iterations. The value of ii is bounded below by two factors: the *loop carried dependency bound* and the *partitioned resource bound*. The compiler determines these two values in stage 2.

Since both the *loop carried dependency bound* and *partitioned resource bound* directly impact ii, the smaller they are, as determined by the compiler, the smaller ii can be. However, due to the lack of explicit information and the high complexity of the code, the compiler sometimes determines a larger-than-actual bound value. In this case, the stage 2 feedback can be very useful in helping the programmer realize the potential optimization steps.

2.4.3.2.1 Loop Carried Dependency Bound

The *loop carried dependency bound* is defined as the distance of the largest *loop carry path*; and *a loop carry path* occurs when one iteration of a loop produces a value that is used in a future iteration.

The *loop carried dependency bound* can be visualized with a dependency graph, which is a graphical representation of the data dependency between each instruction. Example 6 shows a simple loop code, and Figure 8 shows the simplified, corresponding dependency graph (i.e., branch related code removed).

Example 6. A Simple Summation Code

```
void simple_sum(short *sum, short *in1, unsigned int N)
{
     int i;
     for (i = 0; i < N; i++)
     {
          sum[i] = in1[i] + 1;
     }
}
```

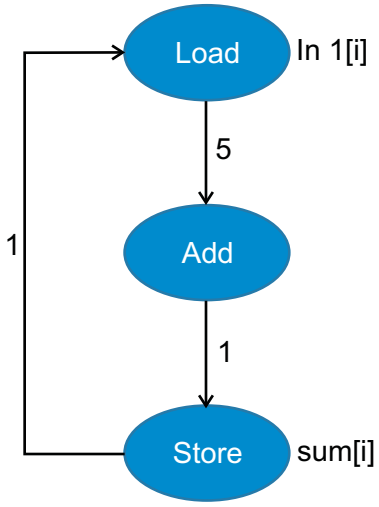

Figure 8. Simple Dependency Graph

As shown in Figure 8, each arrow connects two instructions (or data objects), with the one that is pointed to by the arrow depending on the result of the other. The number on the arrow represents after how many cycles the next instruction can be executed. One arrow that is particularly critical is the one pointing from store to load, which completes a loop carried path. If that arrow wasn't there, there would be no loop carried path in this dependency graph.

To the compiler, the location of the data pointed to by *sum and *in1 are unknown; it has to prepare for all possible outcomes during compilation. The worst case scenario happens when sum = in1 +4. In this case, as the dependency graph shows, load (i+1) depends on store (i), which in turn depends on add (i), which depends on load (i). In other words, iteration (i+1) is dependent on iteration (i), and it cannot be started until 7 cycles after iteration (i) is started. Therefore, the *loop carried dependency bound* for the simple_sum loop is 7.

However, if sum and in1 will never point to the same memory buffer, there should not be any loop carried dependency. In this case, additional techniques, as shown in Section 3.1.2 and Section 3.2, can be used to minimize the *loop carried dependency bound*.

2.4.3.2.2 Partitioned Resource Bound

For a given loop, the number of times each functional unit is used during each iteration also affects ii. For example, if 8 multiply instructions need to be completed every iteration, each .M unit needs to complete 4 multiply instructions per iteration (since there are two .M units). Therefore, the next iteration, regardless of whether software pipelining is used or not, will not be able to start until at least 4 cycles later.

The *partitioned resource bound* is defined as the maximum number of cycles any functional unit is used in a single iteration, after all the instructions are assigned to individual functional units.

If the resource partition information indicates an unbalanced partition between the A side and B side functional units, additional techniques, such as loop unrolling, can be used to balance the usage of the two halves of the core. These techniques are explained in Section 3.3.2.

If the feedback indicates that one type of functional unit is used much more heavily than all other types, the programmer can consider using alternative expressions that utilize different functional units. For example, if the .M units are overused, instead of performing b = a * 2, perform b = a + a to use the .L units, or b = a << 2 to use the .S units.

2.4.3.3 Stage 3 – Searching for Software Pipeline Schedule

In stage 3, based on the knowledge it has gained in the previous stages, the compiler tries to optimize the loop by finding the minimal value of ii that it can use to schedule a software pipelined loop. The compiler starts with an ii value that is equal to the larger of the two bounds (the *loop carried dependency bound* and the *partitioned resource bound*). If a software pipelined schedule can be created with this ii, then the optimization task is completed; if not, the compiler will increase the value of ii by 1, and try again. The process on how the compiler schedules a loop based on a particular value of ii is beyond the scope of this document.

NOTE: The following compiler feedback, *ii = 4 Schedule found with 5 iterations in parallel*, means the following:

- The code to be optimized was divided into five stages
- The parallel representation combines the five stages in parallel, with each stage for a different iteration
- The parallel representation can be completed in 4 cycles

In the loop feedback, the compiler can also include error or warning messages, such as “Software Pipelining Disabled (this can very often be fixed by using the aforementioned compiler option *--opt_level=3*, or simply *-o3*)”, to indicate a failure and the cause of the failure. Based on these messages, the programmer should either provide additional information via means mentioned in Section 2.4.2, or modify the program. For detailed information on the warning messages and suggested ways to fix them, see reference [1].

For more detailed information on compiler feedback, also see reference [1].

2.5 Concept V: Explicit Code Optimization

Compiler optimization can be a very effective and time saving approach. However, there are cases when the compiler alone is not sufficient, such as if the code makes function calls in a loop or the developer is implementing complex, hard-to-implement operations. In such cases, three additional C6000 software resources can be used: intrinsic operations, optimized C6000 DSP libraries, and C inline functions.

2.5.1 Intrinsic Operations and DSP Libraries

The C6000 DSP functional units support many complicated operations efficiently. For example, a shuffle operation, which separates the even and odd bits of a 32-bit value into two different variables, can be completed in 2 cycles on the C6000 DSP. However, such an operation will take lines of code to implement in C. Even after compiler optimization, the C version of the bit extraction operation will still take many cycles.

To help the programmer further improve performance in situations like this, intrinsic operations are introduced. Intrinsic operations are function-like statements, specified by a leading underscore, to perform operations that are cumbersome or inexpressible in C code. For example, the bit manipulation operation previously mentioned can be accomplished by calling the *_shfl* intrinsic operation. The intrinsic operations are not function calls (though they have the appearance of function calls), so no branching is needed. Instead, the use of intrinsic is a way to tell the compiler to issue one or more particular instructions of the C6000 instruction set. For a list of all the intrinsic operations supported by the compiler, see reference [2].

As a developer, you do not always need to implement a complicated function using intrinsic operations. For many standard math functions and signal processing functions, TI provides highly optimized DSP software libraries, such as FastRTS and DSPLIB. The library APIs can be used either as is to reduce development and optimization time or as an example on how to use intrinsic operations to create highly optimized functions. For example, the *Inverse FFT with Mixed Radix* function (DSPF_sp_ifftSPxSP), available in DSPLIB, implements many of the techniques that are highlighted in this document. For a list of all the C6000 DSP software libraries, visit http://processors.wiki.ti.com/index.php/Software_libraries.

2.5.2 Inline Functions

As mentioned before in Section 2.2.3, a function call inside a loop might prevent the compiler from being able to perform software pipelining and loop optimization. When an inline function is called, the C compiler automatically inserts the function at the point of the call. This action is called inline function expansion, and is very advantageous especially in short functions for the following reasons:

- It saves the overhead of a function call
- The optimizer can optimize the function in context with the surrounding code.
- Most importantly, the optimizer can perform loop optimization.

However, one potential drawback is that the size of the code could be increased due to the direct insertion of the called function (especially if that function is inlined in multiple locations). The programmer should try to avoid making inline functions too long or calling a given function too often in different points of the source code. For more information on inline functions, see reference [2].

3 C6000 Optimization: Five Key Techniques

Figure 9 illustrates a typical code optimization flow.

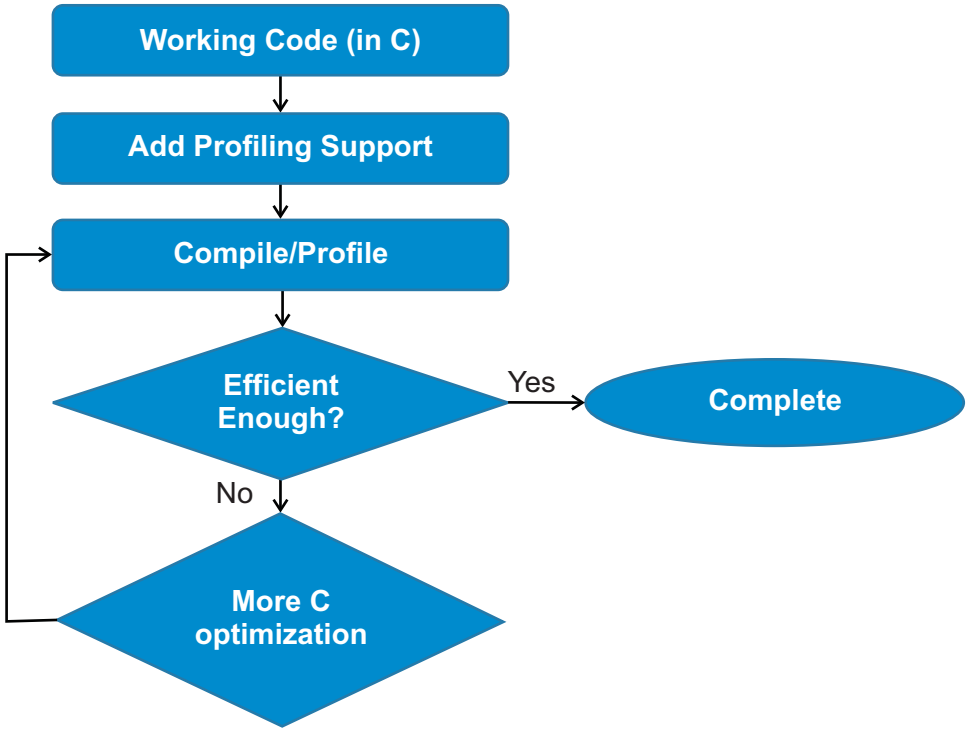

Figure 9. Code Optimization Flow

Before starting the optimization process, profiling support needs to be added to the application to measure code performance. Refer to Appendix B on common ways to add profiling support.

After adding profiling support, one can begin the compilation process. The command below shows a simple compilation process initiated on the command prompt window.

```
cl6x  dsp1.c dsp2.c -z –l linker.cmd –l dsp3.lib –o DSP.out
```

The compilation, development, and optimization process can also be done on TI's Code Composer Studio™ integrated development environment. For more information on Code Composer Studio, visit http://www.ti.com/CCstudio. Code Composer Studio also provides profiling support. For more information, visit http://processors.wiki.ti.com/index.php/Profiler.

If the overall cycle count meets the target, there is no need to optimize the code. If more optimization is needed, the following five techniques are most effective in helping a developer meet the performance target. Section 3.1 through Section 3.5 cover each of these techniques in detail.

1. Use –o3 and consider –mt for optimization; use –k and consider –mw for compiler feedback
2. Apply the *restrict* keyword to minimize *loop carried dependency bound*
3. Use the MUST_ITERATE and UNROLL pragmas to optimize pipeline usage
4. Choose the smallest applicable data type and ensure proper data alignment to help compiler invoke Single Instruction Multiple Data (SIMD) operations
5. Use intrinsic operations and TI libraries in case major code modification is needed

3.1 Technique I: Choosing Proper Compiler Options

```
Key Point
Use –o3 and consider –mt for optimization; use –k and consider –mw for compiler feedback
Example
cl6x –o3 –mt –k –mw dsp1.c dsp2.c -z –l linker.cmd –l dsp3.lib –o DSP.out
```

Compiler options are parameters that can be passed to the compiler when it is invoked, and these options control the operation of the compiler. This section covers the most effective options during optimization.

3.1.1 The --opt_level=3 (-o3) option

The --opt_level option is the single most important option for optimizing an application. It requires a value as its parameter, which can be 0, 1, 2, or 3. When --opt_level=0 (or –o0 in short) is used, no optimization is performed at all. The greater the value is, the higher the optimization level is. When --opt_level=3 (or –o3) is used, the compiler will try to perform all the optimization techniques it is capable of carrying out (such as software pipelining) based on the knowledge that is available to it. Other techniques discussed in this section can be viewed as different ways to supply additional information about the application to the compiler.

NOTE: One option that can greatly impact the effectiveness of the –o3 option is the –g option. The –g option enables full symbolic debugging support, which is a great tool during the development / debugging stage. However, it causes less parallelism and produces extra code, which can greatly impact the performance of the code, and should be avoided in production code.

For a code example demonstrating the performance improvement by using the –o3 option, see Section A.1.

3.1.2 The –mt option

In general, the compiler, due to the lack of knowledge about the implicit assumption of the code, has to prepare for the worst case. This sacrifice becomes very significant in the case when there are multiple pointers in a loop. The compiler has to prepare for the case when all the pointers refer to the same memory location, and this assumption can lead to a high *loop carried dependency bound* (see Section 2.4.3.2), which in turn has a great negative impact on the performance of the optimized loop.

The –mt option tells the compiler that, for any function in the entire application, the pointer-based parameters of this function will never point to the same address. This option can greatly enhance the performance of the code.

NOTE: This option has its limitations. It only applies to the pointer-parameters of the functions, not the local pointers. More importantly, this option can only be used if there are no overlapping pointer-parameters in any functions in the entire application. If used incorrectly, it could cause the code to behave incorrectly.

For a code example demonstrating the performance improvement by using the –mt option, see Section A.2.

3.1.3 The -k and –mw option

The –k and –mw options are not directly performance oriented. However, these options enable the compiler to generate detailed feedback information while having no impact on the performance of the code. The programmer can use the feedback generated to detect what may be tuned to achieve higher performance.

The –k option retains the compiler assembly output. The –mw option provide additional software pipelining information, including the single scheduled iteration of the optimized loop.

There are many other options that can further enhance the performance of the compiler. Moreover, there are many other options that control the non-performance-oriented operations of the compiler. For more information, see reference [2].

3.2 Technique II: Applying the Restrict Keyword

Key Point
Apply the restrict keyword to minimize loop carried dependency bound

As mentioned in Section 2.4.3, the *loop carried dependency bound* should be minimized in order to minimize the overall iteration interval of a loop. The –mt option mentioned in Section 3.1.2 is one viable option. The *restrict* keyword is a manual and local, but more powerful, alternative to the –mt compiler option.

The *restrict* keyword can be used to inform the compiler that the pointers in a given function will never point to the same memory buffer. This can avoid the unnecessary high loop carried dependency that the compiler might conclude. As demonstrated in Example 7, the *restrict* keyword can be applied to any pointers, regardless of whether it is a parameter, a local variable, or an element of a referenced data object. Moreover, because the *restrict* keyword is only effective in the function it is in, it can be used whenever needed (unlike the –mt compiler flag which has an effect on the entire application).

Example 7. Applying the Restrict Keyword

```
void myFunction(int * restrict input1, struct *s)
{
       int * restrict output;
       int * restrict input2 = s->pointer;
}
```

Recall that the code in Example 6 had a *loop carried dependency bound* of 7. After applying the *restrict* keyword to the pointer parameters, as shown in Example 8, the *loop carried dependency bound* is reduced to 0.

Example 8. Simple Summation Code With the Restrict Keyword Applied

```
void simple_sum(short * restrict sum, short * restrict in1, unsigned int N)
{
        int i;
        for (i = 0; i < N; i++)
        {
                    sum[i] = in1[i] + 1;
        }
}
```

Generally, if the compiler feedback shows that the *loop carried dependency bound* is higher than expected, consider applying the *restrict* keyword.

For a code example demonstrating the performance improvement by using the *restrict* keyword, see Section A.2.

3.3 Technique III: Passing Information via Pragma

Key Point
Use the MUST_ITERATE and UNROLL pragmas to optimize pipeline usage for loops

The loop performance can be further optimized by using various pragma directives. The most effective two pragmas are MUST_ITERATE and UNROLL.

3.3.1 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of the loop count, and it has the following syntax, and it must be inserted immediately above the loop code.

```
#pragma MUST_ITERATE(lower_bound, upper_bound, factor)
```

The lower_bound defines the minimum possible total iterations of the loop, the upper_bound defines the maximum possible total iterations of the loop, and the factor tells the compiler that the total iteration is always an integer multiple of this number. The lower_bound and upper_bound must be divisible and at least as big as the factor. Any of the three parameters can be omitted if unknown, but if any of the value is known, it is recommended to provide this information to the compiler. For example, if the upper_bound is unknown but the other two are 10 and 2, respectively, one should apply the pragma as follows:

```
#pragma MUST_ITERATE(10, , 2)
```

The lower_bound and the factor can be extremely helpful to the compiler during optimization.

Without the minimum iteration count, the compiler has to take into consideration that sometime the loop will iterate once, or not at all. Most of the time, the minimum safe trip count for the software pipeline is larger than 1, which means that the compiler has to assume for the case when the software pipelining fails and generate extra code. This will increase the code size and negatively impact the performance. Therefore, it will be very helpful to provide the lower_bound value to the compiler. If the exact lower bound cannot be determined, at least see if the *minimum safe trip count* (reported by the compiler) can be assured.

By providing the factor value, we are giving the compiler the freedom to unroll the loop if it finds it profitable. When the compiler unrolls the loop, for example, by a factor of 2 the following line will appear in the compiler feedback.

```
;* Loop Unroll Multiple : 2x
```

NOTE: In general, a factor of 2 or 4 can lead to significant performance increase.

For a code example demonstrating the performance improvement by using the MUST_ITERATE pragma, see Section A.3.

3.3.2 Loop Unrolling and the UNROLL Pragma

In many cases, the resource partition or functional unit utilization becomes unbalanced. For example, in Example 9, the .D unit on the A side is used twice every iteration, and the .D unit on the B side is only used once. This indicates that the .D unit on the B side is left *open* for 1 cycle in each iteration of the loop. In this case, the *partitioned resource bound* is affected by this unbalanced partition. Ideally, it would be more efficient if both units are used 1.5 cycles per iteration or 3 cycles per 2 iterations.

Example 9. A Loop Code With Unbalanced Resource Partition

```
void Loop(int * restrict output, int * restrict input1, int * restrict input2, int n)
{
     int i;
     for (i=0; i<n; i++)
     {
          output[i] = input1[i] + input2[i];
     }
}

//An excerpt from its compiler feedback

;* Partitioned Resource Bound(*) : 2
;* Resource Partition:
;*            A-side   B-side
;* .L units   0        0
;* .S units   0        1
;* .D units   2*       1
;* .M units   0        0
```

3.3.2.1 Manual Unroll

As shown in Example 10, loop unrolling can be accomplished manually. The number of iterations is halved, and two of the previous single loop implementation is carried out in the new loop.

Example 10. Manually Unrolling a Loop

```
void Loop(int * restrict output, int * restrict input1, int * restrict input2, int n)
{
     int i;
     for (i=0; i<n; i+=2)
     {
          output[i] = input1[i] + input2[i];
          output[i+1] = input1[i+1] + input2[i+1];
     }
}

//An excerpt from its compiler feedback

;* Partitioned Resource Bound(*) : 3
;* Resource Partition:
;*              A-side    B-side
;* .L units     0         0
;* .S units     1         0
;* .D units     3*        3*
;* .M units     0         0
```

The .D units are used 3 times on each side per iteration, but the number of iterations is halved. This, in theory, can lead to a 25% reduction in overall cycle count.

Although effective, the manual unroll process can be tedious for most loops. Generally, it is recommended that the C6000 compiler and pragmas be used to unroll loops.

3.3.2.2 Compiler Unroll

As mentioned in Section 3.3.1, the MUST_ITERATE pragma gives the compiler the flexibility to automatically unroll when it is deemed profitable, and the compiler often can perform this in an ideal way.

However, sometimes when the code gets more complicated, it is not always easy for the compiler to perform this action. The UNROLL pragma can be used to instruct the compiler to unroll exactly what the programmer wants it to. The syntax of the UNROLL pragma is as follows, and it should be used immediately before the loop it is trying to unroll.

```
#pragma UNROLL(factor)
```

Before using UNROLL, one must also ensure the loop count is indeed divisible by the factor. As demonstrated in Example 11, to let the compiler know this so it will not generate additional code to handle exceptions, the MUST_ITERATE pragma should also be used.

Example 11. Using the UNROLL Pragma

```
void Loop(int * restrict output, int * restrict input1, int * restrict input2, int n)
{
     int i;

     #pragma MUST_ITERATE(lower_bound, ,2)
     #pragma UNROLL(2)

     for (i=0; i<n; i++)
     {
     // n & lower_bound >= the minimum safe trip count, and divisible by 2

                 output[i] = input1[i] + input2[i];
     }
}
```

In general, if the resource partition portion of the compiler feedback shows a significant imbalance, consider using the UNROLL pragma. It is recommended that the programmer experiment with different factor value when using the UNROLL pragma, and pick the factor that yield the best performance.

For a code example demonstrating the performance improvement by using the UNROLL pragma, see Section A.3.

It is important to keep in mind that loop unrolling does increase the size of the loop. If the size of the loop is too big, it hurts the compiler's ability to perform software pipelining and may disallow the use of the SPLOOP buffer. Hence, if one of the following four messages is displayed in the compiler feedback, consider splitting the loop into smaller loops or reduce the complexity of the loop so the unrolled version can be software pipelined.

- Disqualified loop: Too many instructions
- Cannot Allocate Machine Registers
- Cycle Count Too High. Not Profitable
- Did Not Find Schedule

3.3.2.3 Loop Unrolling and SIMD

Another advantage of loop unrolling is that it gives the compiler more opportunities to perform SIMD instructions. The word SIMD stands for single instruction multiple data, which describes a state when the DSP core can operate on multiple data during every supported operation. This can be very beneficial especially during data load and store.

3.4 Technique IV: Enabling SIMD Instructions

Key Point
Choose the smallest applicable data type and ensure proper data alignment to help compiler invoke SIMD operations

Loop unrolling enables SIMD instructions to be used. However, if the data size is too big, it limits the effectiveness of SIMD instructions.

Two of the most frequently used SIMD instructions are the LDDW and STDW instructions, which perform aligned 64-bit load and 64-bit store, respectively. Assuming all the data used are actually 16-bit data, it would be ideal if the LDDW and STDW instructions can operate on 4 elements every time. However, if the data is declared as 32-bit type, LDDW and STDW can only operate on 2 elements every time. Therefore, to fully utilize SIMD instructions, it is important to choose the smallest data size that works for the data.

Besides LDDW and STDW, the C6000 core supports many other SIMD instructions. When the proper data type is chosen and loop unrolling performed, the compiler will generally utilize SIMD instructions automatically.

For a code example demonstrating the performance improvement by using proper data type, see Section A.4. For detailed descriptions on the C6000 instructions, see reference [3].

NOTE: The LDDW and STDW instructions require that data be aligned on 8-byte boundary. If data is not aligned, the non-aligned version of these instructions (LDNDW and STNDW) will be used instead. Even though the C64x+, C674x, and C66x cores can handle non-aligned data access as efficiently as aligned access, aligned data access can lead to better instruction parallelism when two or more data access are needed. Data alignment can be forced by using the DATA_ALIGN pragma; and the alignment information can be passed to the compiler via the _nassert intrinsic operation. For detail on the DATA_ALIGN pragma, see reference [2][2]. The _nassert intrinsic operation is covered in Section 3.5.1.

3.5 Technique V: Using Intrinsic Operations and Optimized Libraries

Key Point
Use intrinsic operations and TI libraries in case major code modification is needed

The four techniques mentioned in previous sections require minor changes to the implementation of the program. In general, they are sufficient to meet the performance target. In case further optimization is needed, a programmer can modify the implementation of the program by using C6000 intrinsic operations and C6000 software libraries.

3.5.1 Intrinsic Operations

C6000 intrinsic operations can help greatly optimize the code as it allows the programmer to directly invoke C6000 assembly operations that are difficult to express in the C language. C6000 DSP supports many intrinsic operations, and these operations can be grouped as follows:

- Intrinsics that support standard operation, but more efficient (SIMD)
  - Intrinsics that support faster memory access
    - i.e., _amemd8_const. This intrinsic operation allows aligned loads and stores of 8 bytes to memory. Syntax as follows:

```
double & _amemd8 (void *ptr);
```

- Intrinsics that support faster data operation (C64x / C64x+ / C674x / C66x only)
  - i.e., _mpyu4. For each unsigned 8-bit quantity in src1 and src2, the CPU performs an 8-bit by 8-bit multiply. The four 16-bit results are packed into a 64-bit result. Syntax as follows:

```
double _mpyu4 (unsigned src1, unsigned src2);
```

- Intrinsics that support non-standard operations that are cumbersome to write in C
  - i.e., _norm. Returns the number of bits up to the first non-redundant sign bit of the parameter. Syntax as follows:

```
unsigned _norm (int src2);
```

  - i.e., _shfl (C64x / C64x+ / C674x / C66x only). The lower 16 bits of src2 are placed in the even bit positions, and the upper 16 bits of src are placed in the odd bit positions. Syntax as follows:

```
unsigned _shfl (unsigned src2);
```

- nassert: a special intrinsic operation that generates no code. Instead it tells the optimizer that the expression declared with the assert function is true, which gives a hint to the optimizer as to what optimizations might be valid. Syntax and example as follows:

```
Syntax: void _nassert(int);
Example: _nassert((int) y % 8 == 0);
```

- The above example indicates that a pointer y is aligned to 8-byte boundary. This can be used to hint the compiler that SIMD instructions such as LDDW and STDW should be executed.

For a list of all the intrinsic operations supported by the compiler, see reference [2].

For a code example demonstrating the performance improvement by using the intrinsic operations see Section A.5.

3.5.2 3.5.2 TI DSP Libraries

It is not always easy to implement a function or algorithm using intrinsic operations. For many standard math functions and typical digital signal processing functions, such as audio and video processing functions, TI provides highly optimized DSP software libraries for the C6000 family of devices. These library APIs can be used either as is to reduce development and optimization time or as an example or baseline framework on how to use intrinsic operations to create highly optimized functions. For many of the libraries, the source files are provided to allow customization.

For a list of all the C6000 DSP software libraries, visit this topic at:

http://processors.wiki.ti.com/index.php/Software_libraries

NOTE: Many standard C functions such as standard input/output (I/O) functions and math operations, even though handy, can negatively impact performance, especially when used in loops. For example, a printf function can take several hundred cycles to complete. Therefore, standard I/O functions should be avoided as much as possible, especially from loops. When DSP/BIOS is used, use LOG_printf instead when this functionality is absolutely needed. When math or other functions are needed, use the APIs provided in the C6000 DSP library (if available) instead of the one supported by the standard C library.

3.6 Technique Summary

This section discusses the most effective techniques to optimize a given DSP program. The following steps, in descending order of priority, should be taken:

1. Use –o3 and consider –mt for optimization; use –k and consider –mw for compiler feedback
2. Apply the *restrict* keyword to minimize *loop carried dependency bound* for loops
3. Use the MUST_ITERATE and UNROLL pragmas to optimize pipeline usage
4. Choose the smallest applicable data type and ensure proper data alignment to help compiler invoke SIMD operations
5. Use intrinsic operations and TI libraries in case major code modification is needed

The above techniques can be applied one at a time. If the performance target is met, there is no need to go further down the list as it requires extra effort.

During original code development, it is recommended that techniques 2 to 4 be applied, when applicable, during the original code development phase, because the original author has the most accurate knowledge on the implicit assumptions of the program.

For advanced reading on the optimization techniques, see references [1] and [5].

4 Summary

This document serves the purpose to help first time C6000 DSP developers quickly understand the C6000 DSP architecture and C6000 DSP optimization. The document lists the five key concepts that are crucial to the understanding of the C6000 DSP and highlights the five most effective techniques that can be used during the optimization stage.

For advanced readings on all the concepts and techniques explained in the document, refer to the documents and links listed in Section 5.

5 References

1. *TMS320C6000 Programmer's Guide* (SPRU198)
2. *TMS320C6000 Optimizing Compiler v 7.3 User's Guide* (SPRU187)
3. *TMS320C674x DSP CPU and Instruction Set Reference Guide* (SPRUFE8)
4. *TMS320C674x DSP Megamodule Reference Guide* (SPRUFK5)
5. *Hand-Tuning Loops and Control Code on the TMS320C6000* (SPRA666)
6. *DSP/BIOS Timers and Benchmarking Tips* (SPRA829)
7. List of all C6000 DSP software libraries: http://processors.wiki.ti.com/index.php/Software_libraries
8. TI Code Composer Studio product folder: http://www.ti.com/CCstudio
9. Introduction to CCSv4 profiler: http://processors.wiki.ti.com/index.php/Profiler
10. Additional optimization techniques and considerations: http://processors.wiki.ti.com/index.php/C6000_Optimization_Techniques_Considerations

Appendix A Code Examples

Testing environment as follows:

- Compiler: TI Code Generation Tools ver. 7.2.0
- Platform: C674x CPU Cycle Accurate Simulator, Little Endian
  - Also use option *–mv = 6740* so the compiler can use all the C674x specific features.
- IDE: CCS 5.0.1
- Profiling Method: Reading the 64-bit counter on C6000 DSP
  - Also use option *-D* = "*_TMS320C6740*" to use the registers defined in *c6x.h*.

For more examples and performance data, see the *C6000 Profiling Examples* at the following URL: http://processors.wiki.ti.com/index.php/C6000_Profiling_Examples.

A.1 Example on the Compiler Options

The function in Example 12 performs a dot product operation. It is implemented in *dotp.c*, and called in *main.c*.

Example 12. Code Exmaple for A.1

```
int dotp(short * m, short * n, int count)
{
        int i;
         int sum = 0;
        for (i=0; i < count; i++) //count = 256
        {
                        sum = sum + m[i] * n[i];
        }
        return(sum);
}
```

Table 1. Cycle Count for A.1

Technique	No Option	-g	-o3
CPU Cycle Count	2837	7720	288

A.2 Example on the Restrict Keyword

The function in Example 13 performs a weighted vector sum operation. It is implemented in *wvs.c*, and called in *main.c*.

Example 13. Code Exmaple for A.2

```
//void wvs(short * xptr, short * yptr, short * zptr, short * w_sum, int N)
void wvs(short * restrict xptr, short * restrict yptr, short * restrict zptr, short * restrict 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++) //N = 40
       {
                    w_vec1 =  xptr[i] * w1;
                    w_vec2 =  yptr[N-1-i] * w2;
                    w_sum[i] = (w_vec1 + w_vec2) >> 15;
       }
}
```

Table 2. Cycle Count for A.2

Technique	-o3 w/o restrict	-o3 w/ restrict	-o3 –mt
CPU Cycle Count	507	117	117
Loop Carried Dependency Bound	12	0	0
ii	12	2	2

A.3 Example on the Pragma

The function in Example 14 performs a dot product operation. It is implemented in *dotp.c*, and called in *main.c*.

Example 14. Code Exmaple for A.3

```
int dotp(int * m, int * n, int count)
{
       int i;
       int sum = 0;

       _nassert((int) m % 8 == 0);
       _nassert((int) n % 8 == 0);
       #pragma MUST_ITERATE(256, 256,4)
       #pragma UNROLL(4)

       for (i=0; i < count; i++) //count = 256
       {
                    sum = sum + m[i] * n[i];
       }
       return(sum);
}
```

Table 3. Cycle Count for A.3

Technique	-o3	-o3 + MUST_ITERATE	-o3 + MUST_ITERATE + UNROLL	-o3 + MUST_ITERATE + UNROLL + _nassert
CPU Cycle Count	279	279	285	157
Unroll Factor	N/A	N/A	4x	4x
ii	1	1	4	2

A.4 Example on Using Proper Data Type

The function in Example 15 performs a dot product operation. It is implemented in *dotp.c*, and called in *main.c*.

Example 15. Code Exmaple for A.4

```
//int dotp(int * m, int * n, int count)
int dotp(short * m, short * n, int count)
{
       int i;
       int sum = 0;

       _nassert((int) m % 8 == 0);
       _nassert((int) n % 8 == 0);
       #pragma MUST_ITERATE(256, 256,4)
       #pragma UNROLL(4)

       for (i=0; i < count; i++) //count = 256
       {
                     sum = sum + m[i] * n[i];
       }
       return(sum);
}
```

Table 4. Cycle Count for A.4

Technique	int type element	short type element
CPU Cycle Count	157	91
Unroll Factor	4x	4x
ii	2	1

A.5 Example on the Intrinsic Operations

This function in Example 16 performs a saturated add operation. It is implemented in *sadd.c*, and called in *main.c*. The intrinsic operation, *_sadd (var1, var2)*, can be used to accomplish the same thing.

Example 16. Code Exmaple for A.5

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

Table 5. Cycle Count for A.5

Technique	sadd() w/ -o3	_sadd() intrinsic operation
CPU Cycle Count	18	2

Appendix B Profiling Support

B.1 Directly Accessing the C6000 Counter

The easiest way to perform profiling is to directly access the C6000 counter in the function that is to be profiled. The 64-bit counter is a free-running timer that advances every CPU clock during normal operation. The code in Example 17 can be used to perform profiling.

Example 17. Profiling Example

```
// Required header file // Both files come with the code generation tools package
#include <stdint.h> // defines uint64_t
#include <c6x.h> // defines _itoll, TSCH, TSCL

// In the variable declaration portion of the code:
uint64_t start_time, end_time, overhead, cyclecount;

// In the initialization portion of the code:
TSCL       = 0;                    //enable TSC
start_time = _itoll(TSCH, TSCL);
end_time   = _itoll(TSCH, TSCL);
overhead   = end_time-start_time; //Calculating the overhead of the method.

// Code to be profiled
start_time = _itoll(TSCH, TSCL);
function_or_code_here();
end_time   = _itoll(TSCH, TSCL);
cyclecount = end_time-start_time-overhead;

printf("The code section took: %lld CPU cycles\n", cyclecount);
```

B.2 Utilizing C6000 Compiler Function Hook

The C6000 compiler supports entry hook and exit hook functions, which can be handy during profiling. An entry hook is a routine that is called upon entry to each function in the program; and an exit hook is a routine that is called upon exit of each function. Applications for hooks include debugging, trace, profiling, and stack overflow checking. Example 18 shows a simplified entry/exit hook function implementation.

Example 18. Entry/Exit Hook Function Example

```
void entry_hook()
{
start_time = _itoll(TSCH, TSCL);
}

void exit_hook()
{
end_time   = _itoll(TSCH, TSCL);
}
```

To enable entry hook support, use compiler option *--entry_hook*; to enable exit hook support, use *--exit_hook*. For additional details on hook function support, see reference [2].

B.3 Other Profiling Support

If the C6000 DSP/BIOS is used, the profiling support provided in the DSP/BIOS can be used. For instructions on how to enable profiling under BIOS, see reference [6].

If Code Composer Studio is used, the profiling support provided by Code Composer Studio can be used. For more information, visit http://processors.wiki.ti.com/index.php/Profiler.

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 the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI 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. Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
	TI E2E Community Home Page		e2e.ti.com

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265