Generating Efficient Code with TMS320 DSPs: Style Guidelines

APPLICATION REPORT: SPRA366

Karen Baldwin
Rosemarie Piedra
Semiconductor Sales & Marketing

Digital Signal Processing Solutions
25 July 1997

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

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

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

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

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

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

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

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Abstract **7**
Product Support **8**
World Wide Web 8
General Guidelines **9**
Variable Declaration **10**
Local vs. Globals 10
Initialization of Variables **16**
Memory Alignment Requirements and Stack Management **18**
Accessing Memory-mapped Registers **23**
Looping **24**
TMS320 Loop Implementation - Analysis 24
Initial Conditional Branch 27
Control Code and Switch Statements **29**
Functions **30**
Math Operations **32**
q15 arithmetic/MACs 33
Acknowledgments **35**
Appendix A. Summary of Guidelines **36**
Appendix B. Instructions Used by the C54x Compiler **38**
Appendix C. Instructions Used by the C5x/2xxCompiler **39**
Appendix D. Instructions Used by the C3x/4x Compiler **40**
Appendix E. A Dot Product Example: C54x Study Case **41**

Tables

Table 1 Data Type Size (in bits) across TMS320 Compilers 15
Table 2 Loop Combinations 26
Table 3 Guideline Usability by Type and Version 36
Table 4 Instructions Used by the C54x Compiler 38
Table 5 Instructions Used by the C5x/2xx Compiler 39
Table 6 Instructions Used by the C3x/4x Compiler 40

Generating Efficient Code with TMS320 DSPs: Style Guidelines

Abstract

This report presents C-coding style guidelines to improve the efficiency of the Texas Instruments (TI™)TMS320 C2x/C2xx/C5x/C54x/C3x C-compilers, indicating what to avoid or what to promote when coding a TMS320 in C. For development time savings, apply these guidelines before deciding to re-write a time-critical portion in assembly.

To illustrate some of the guidelines a case study (vector dot product) is presented in Appendix E.

NOTE:

TI code generation tools have been designed to achieve the best optimization possible for the entire application, not for specific kernels. Since the tools look at the entire code, not selected pieces, you may see inefficiencies in a certain kernel of code that reflect efficient code generation in another section of code.

This application note assumes that you are using the latest releases of the TMS320 compilers (C3x/4x version 5.0, C2xx/c5x version 6.65 Beta, C54x version 1.2). Any effect of future compiler releases on the guidelines presented here will be documented in future releases of this document, but such an effect is not foreseen.

Product Support

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

General Guidelines

Before looking at TMS320 specific coding style guidelines, let's mention some general C guidelines to follow:

- ***TIP:* (All) Avoid removing registers for C-compiler usage (-r option).** Register removal is costly because it removes valuable resources for the compiler and produces overall code generation quality degradation. Let the compiler allocate register variables. Remove registers only for time-critical interrupts for which that is the only option left for speed-up improvement.

- ***TIP:* (All) To selectively optimize functions place them into separate files.** This will allow you to compile the files individually

- ***TIP:* (All) Use the least possible volatile variables.** Compilers by default assume that they are the only entity reading and writing to data. To avoid code removal, one option is to declare the variable as volatile however be aware that a volatile declaration might impact negatively the efficiency of the code generation. For example, if you make volatile the variable getting a partial sum, the compiler will not generate optimal code because it cannot place a volatile variable into a register. Also, volatile declaration will prevent inlining of the function where the variable is declared or used.

- ***TIP:* (All) For best optimization, use program-level optimization (-pm option) in conjunction with file level optimization (-oe option).** TMS320 compilers offer a whole program-level compilation (-pm) that when used with file level optimization yields the best overall code for the complete application. For this to take effect all the source code need to be passed in one single command line (i.e. cl*xx* -p -n *.c). By viewing all the files before generating code for each, the compiler gains valuable information on how the different code blocks interact and optimize it accordingly. The only drawback is an increased compilation time that may not be a concern during the last stages of the software development process.

Variable Declaration

Local vs. Globals

- **(C2x/C5x/2xx) Prefer global over local variables or use the -oe option**
- **(C54x) Prefer local over global variables**
- **(C3x) No special preference (assume preference for locals as a default)**

In general, without looking at any specific processor architecture, local variables tend to be more C friendly. When handling locals a compiler can usually assign registers to function-local variables whether they are declared "register" or not. On the contrary, if you declare a variable as global or static, a compiler can only try to "cache" their values in registers over relatively small portions of code. This will cause extra "stores back to memory" when the compiler detects that an intervening function (for example function *f2* in the code below) might potentially modify the global variable (*variable a*). Another point in favor of locals is good software engineering: globals have more dedicated memory use and functions will not be recursive.

```
int a;
void f1() {
for (i=0;i<n;i++) a ++;
f2();
a = a+3;

}
```

However, sometimes due to the specific processor architecture, globals might be preferred over locals. Let's analyze this point across TMS320 devices:

C2x/C2xx/C5x: In general, these use global variables instead of local variables. The reason is that the compiler uses more efficient direct addressing mode when accessing globals/statics, but uses indirect addressing for accessing local variables. The exception is when using the -oe optimizer option. Selecting this option tells the optimizer/compiler that the code is not called by any interrupt service routines and is non-recursive. Under these circumstances the compiler is free to treat all local variables as statics, allocating space for them in data memory. It can then use the faster direct addressing mode and optimize usage of data page pointer since it can guarantee that the variables are defined on the same page.

C3x: C3x is efficient in both *stack-relative addressing* (used for local accesses) and *direct-memory addressing* (used for global accesses). The exception is when the global variables exceeds 64k words in which case compilation under large-memory model is used to force DP register initialization at every access (big memory model), potentially doubling the code size. To prevent this case, prefer locals over globals.

C54x: C54x favors the use of local over global variables because of the C54x stack addressing mode.

Local variables can be accessed either by *stack-based addressing* (if the local variable is located in the first 128 16-bit words of local frame space) or by *indirect addressing using AR7-local frame pointer* (if the local variable is located after the first 128 16-bit words of local frame space). The advantage of stack addressing is that it doesn't add an extra word to the instruction to specify the variable address.

Global variables, on the other hand, are accessed via *dmad addressing*. This adds an extra word (the variable address) to the instruction accessing the variable making the instruction a 2-word instruction as a minimum. Note that even when the local frame exceeds 128 words, the use of local variables will provide the same performance as using globals. Even though the local variables are being accessed via indirect addressing with long constant indexing, it requires the same number of words as dmad addressing used for globals.

Local pointers should also be preferred over global pointers. The following example illustrates this point. Using **global pointers can produce larger code when global pointers are modified**. An operation like global_pointer++ is considered an operation with a side effect that must be resolved before the next "sequence point" (i.e. the next ; or ")") . This forces the immediate update of the global pointer variable in memory. A typical case to avoid is the usage of global pointers for MAC (multiply-and-accumulate)-style instructions. Notice the savings in code size by simply using local pointers in the following examples:

Example 1 C54x Sample Code With Global Ptrs

```
unsigned *a,*b;
unsigned int i; unsigned int sum = 0;

unsigned int operands_global() {
for (i=0; i<= 10; i++) {
sum += *a++ * *b++;
}
return sum;
}
```

The code shown below requires 11 extra words and 15 more cycles to execute, than is generated when local pointers are used.

```
000000 771A    STM #10,BRC
000001 000A
000002 F272    RPTBD L3-1
000003 0013'
000004 4A11    PSHM AR1
000005 E800    LD #0,A 31
000006 7211    MVDM *(_a),AR1
000007 0000-
000008 F495    nop
000009 1191    LD *AR1+,B
00000a 7311    MVMD AR1,*(_a)
00000b 0000-
00000c 7211    MVDM *(_b),AR1
00000d 0001-
00000e F495    nop
00000f 3091    LD *AR1+,T
000010 28F8    MAC *(BL), A
000011 000B
000012 7311    MVMD AR1,*(_b)
000013 0001-
000014 L3:
```

Example 2 C54x Sample Code With Local Pointers (better)

```
unsigned int operands_local(unsigned *a, unsigned *b) {

unsigned int i; unsigned int sum = 0;

for (i=0; i<= 10; i++){
sum += *a++ * *b++;
}
return sum;
}
```

The code using local pointers does not require update and store of the pointer variables resulting in smaller/faster code for this loop.

```
000000 8812    STLM A,AR2
000001 F495    nop
000002 7101    MVDK *SP(1),*(AR3)
000003 0013
000004 E800    LD #0,A 27
000005 EC0A    RPT #10
000006 B098    MAC *AR3+, *AR2+, A, A
000007 F495    nop
000008 F495    nop
000009 L3:
```

TIP: **(All) Declare globals in file where they are used the most, or compile using -pm -oe options.** In general, when using -o3 option file level optimization, this would allow the compiler to optimize the use of globals by allocating them to registers across functions inside the same file. In the specific case of the C2xx/c5x compiler, there is an extra benefit. Because the C2xx/C5x compiler initializes the DP to the beginning of the global variables of the file, it knows if the variable is in a different 128-page or not. This minimizes the need to set the DP register within the code generated for the file. When compiling with -pm (whole program mode) and -oe (no code is called by interrupt service routine nor are there any director or indirect recursive calls) this optimization translates into optimal usage of LDPK to load data page pointer.

***TIP:* (All) Allocate most often used elements of an structure, array or bit-fields in the first element, the lowest address or LSB respectively.**

C3x: In the C3x, *arrays and pointers* are accessed via indirect addressing. By following this recommendation, the compiler will be allowed to use C3x instructions that support indirect addressing with a 5 or 8-bit immediate displacement. This avoids the extra math required to manipulate the value of ARn or the usage of IRn registers. For *global structures* accessed via pointers the above recommendation also holds true. However this is not the case for global structures accessed directly (not via pointers) for which the linker itself determines the direct offset of the element (@label+offset) inside the structure that will always be valid and efficient (except for big-memory model). Also by allocating *bit-fields* to the lowest LSBits, the compiler can use C3x OR and AND instructions with short immediate operands to efficiently mask LSbits

***C2x/C2xx/C5x/C54x*:** Because of the lack of offset addressing in these processors, it is better to allocate your most often used data to the first element of an array or an structure. This avoids the use of additional instructions to calculate the correct address to access the element.

***TIP:* (ALL) Prefer unsigned variables over signed.** The unnecessary use of int is many times a common inefficient coding practice. However if you know that a variable will never be less than zero, there is no reason to use a signed integer. An unsigned variable will give you a larger dynamic range (16-bit vs 15-bit in signed integers) and it will provide more information about the variable to the compiler.

***TIP:* (C2x/ C2xx/C5x/C54x) Group together math operations involving the same data type.** The C2xx/C5x/C54x compilers sets/resets the SXM bit as required to guarantee correct operation and type casting. For this reason try to avoid the continuous switching of data types in math operations. The SXM bit is set to 1 (signed-extension enabled) in boot.asm. This is irrelevant for the C3x compiler because the C3x offers specific instructions to handle unsigned arithmetic.

***TIP:* (ALL) Pay attention to data type significance and optimize code accordingly.** The more information you pass to the compiler about the variables the better the code the compiler will produce. The following table lists the data type size in bits across different TMS320 processors. As you can see data type size is not the same across TMS320s, therefore portability issues might arise.

Table 1 Data Type Size (in bits) across TMS320 Compilers

	Char (8 significant bits)	short (16 significant bits)	Int (16 significant bits)	long (32 significant bits)	Float (32 significant bits)	double	long double
C2x/C2xx C5x/C54x	16	16	16	32	32 (IEEE)	32 (IEEE)	32(IEEE)
C3x	32	32	32	32	32 (TI float)	32 (TI float)	40 (TI float)
C6x/C67x	8	16	32	40	32 (IEEE)	64 (IEEE)	64(IEEE)

The correct understanding of the number of significant bits each data type carries can avoid inefficient code generation. Use long only when the full 32-bits are required. *Data type casting should only be used when absolutely required because it might cost you cycles*. The following is a C2xx/C5x/C54x example in which, due to wrong casting, the long-multiply RTS function is invoked when in reality only a regular MPY is need. It's also worthwhile to notice that the use of long data types in the C54x is more efficient than in the C2xx/C5x because the C54x offers special instructions that deal with double-word instructions.

Example 3 RTS Function Invoked when Regular MPY is Needed

```
char b,c;            /* 8-bit significant data */
int a;               /* 16-bit significant data */
long y;              /* 32-bit significant data */
y = (long) a*b*c ;   /* larger code size because of the casting of a to
                        long (32-bits), call L$$MPY */
y = (long) (a* ((int)b*c)); /* smaller code size because everything is
                       kept within the accumulator dynamic range */
```

Initialization of Variables

TIP: **(ALL) Initialize global vars with constants at load time.** Overall, initialization of variables with constant values are costly. In the C2xx/C5x/C54x , the storage of a constant value in a variable adds 1 extra word to the store instruction (ST) regardless of the size of the constant. In the C3x, an extra cycle will be required to store the constant value to a temporary register (this will not be the case in the C4x if the constant is short enough). For this reason, it's suggested to initialize variables at load time and use -cr option to avoid DSP memory consumption by the .cinit section

TIP: **(C54x) When initializing different variables with the same constant, rearrange your code.** If you want to initialize multiple variables with the same constant, the following re-arranging of code helps to improve code generation. Notice that event though the two pieces of code are not semantically identical, the overall result is the same but with different code being generated. In the C54x constant initialization of variables is expensive. The original code produces an store immediate (2-word instruction) for each variable initialization. The suggested code makes the compiler load the constant in the accumulator and produce successive stores into the different variables.

Example 4 C54x Sample Code with Constant

```
unsigned ag1, ag2;
main() {
ag2 = 3;
ag1 = 3;
}
```

The code below uses long constant, taking one extra word per assignment.

```
000000 76F8 ST #3,*(_ag2)
000001 0001-
000002 0003
000003 76F8 ST #3,*(_ag1)
000004 0000-
000005 0003
000006 FC00 RET
```

Example 5 C54x Sample Code with Assign Expression (better)

```
unsigned ag1, ag2;
main() {
ag1 = ag2 = 3;
}
```

This code uses store from accumulator, saving an extra word per assignment

```
000000 E803 LD #3,A
000001 80F8 STL A,*(_ag2)
000002 0001-
000003 80F8 STL A,*(_ag1)
000004 0000-
000005 FC00 RET
```

TIP: **(ALL) Use memcopy when copying an array variable into another.** The RTS function, memcopy has been optimized across TMS320 compilers. Memcopy is declared as "inline" in the string.h (except in case of the C6x compiler) The usage of memcopy should be restricted to copying arrays. Structure copying via memcopy will not generate better code than regular structure1 = structure2 assignment.

Memory Alignment Requirements and Stack Management

TIP: **(C54x) Group all like data declarations together, listing 16 bit data first.** To ensure consistent treatment of all 32 bit data, the C54x compiler pads memory when necessary to cause alignment of all 32 bit quantities on an even address boundary. This is necessary because double word operands are fetched based on address boundaries. If the double word fetch is from an odd address boundary, then the words are fetched LSW-MSW. If they are fetched from an even address boundary they are interpreted as MSW-LSW. Maintaining alignment of 32 bit quantities guarantees the compiler that all 32 bit data is interpreted in the same way.

To avoid wasted space due to 32 bit data alignment requirements, group all like data declarations together, listing 16 bit data first. This is especially true when defining local symbols in a function definition. For global symbols the compiler may rearrange the declarations to group for minimum space requirements. There may still be some memory padding, but the difference will not be as noticeable as in the case of local symbol declarations. This is because the compiler does not rearrange the order of local symbols. They are allocated space on the stack in the order in which they are defined. For this reason, special care should be taken in deciding the order in which local symbols are defined.

Example 6 Original code (no optimal local declaration)

```
func() {
int jk;             /* 1 word  */
long a;             /* 2 words */
int qa;             /* 1 word  */
long jd;            /* 2 words */
int xc;             /* 1 words */
unsigned long c;    /* 2 words */
int xb;             /* 1 word  */
long xyz;           /* 2 words */
                 /* Total symbol size 12 words */
```

In this example, declaration for 16 bit data and 32 bit data is interspersed without regard to alignment requirements. When reserving stack space for the above declarations the compiler generates the following FRAME instruction:

```
FRAME               #-17
```

The compiler uses an extra 5 spaces to allow for alignment.

Example 7 Suggested Code (rearranging declarations)

```
func() {
int jk;                 /* 1 word */
int qa;                 /* 1 word */
int xc;                 /* 1 word */
int xb;                 /* 1 word */
long a;                 /* 2 words */
long jd;                /* 2 words */
unsigned long c;        /* 2 words */
long xyz;               /* 2 words */
                  /* Total symbol size 12 words */
```

Resulting in the following FRAME instruction to reserve space for local symbols:

```
FRAME                   #-13
```

In this instance only one space is "wasted" to assure alignment of the first 32 bit symbol. All others are assumed to be aligned on correct boundary thereafter. This results in a savings of 4 words.

The compiler will also align structures on an even address boundary when that structure contains any 32 bit data. So the same consideration should be applied to the order in which these are declared. In addition, it is possible to take advantage of structure alignment in deciding in which order to declare structure elements. Because the structure is already aligned on an even address boundary, to avoid padding within the structure for alignment of 32 bit data, declare these first and group like-sized data together. For example, compare the size requirements specified by the compiler for the following C54x declarations:

Example 8 Size Requirements of C54x Declarations

```
typedef struct _sample1 {
unsigned long dum_a;     /* 2 words */
int dum_b;               /* 1 word */
int dum_c;               /* 1 word */
 int dum_d;              /* 1 word */
unsigned long dum_e;     /* 2 words */
int dum_f;               /* 1 word */
 unsigned long dum_h;    /* 2 words */
} SAMPLE1_STRUC; /* Total size 10 words */
SAMPLE1_STRUC x1;
```

The compiler generates following .bss directives for the above declarations:

```
.bss _x1,12,0,1  <== Reserves 12 words
```

```
typedef struct _sample2 {
int dum_b;              /* 1 word */
 unsigned long dum_a;   /* 2 words */
 int dum_c;             /* 1 word */
unsigned long dum_e;    /* 2 words */
 int dum_f;             /* 1 word */
unsigned long dum_h;    /* 2 words */
int dum_d;              /* 1 word */
} SAMPLE2_STRUC; /* Total size 10 words */
SAMPLE2_STRUC x2;
```

The compiler generates following .bss directives for the above declarations:

```
.bss _x2,14,0,1      <== Reserves 14 words
```

```
typedef struct _sample3 {
unsigned long dum_a;          /* 2 words */
unsigned long dum_e;          /* 2 words */
unsigned long dum_h;          /* 2 words */
 int dum_b;                   /* 1 word */
int dum_c;                    /* 1 word */
 int dum_d;                   /* 1 word */
 int dum_f;                   /* 1 word */
} SAMPLE3_STRUC; /* Total size 10 words */
SAMPLE3_STRUC x3;
```

The compiler generates following .bss directives for the above declarations:

```
..bss _x3,10,0,1     <== Reserves 10 words
```

***TIP:* (C54x) Use the .align linker directive to guarantee stack alignment on an even address.** As a consequence of maintaining alignment for 32 bit data, the compiler needs to make sure that the stack is initially aligned on an even address boundary and seeks to maintain that alignment on entrance to any defined function. Therefore it adjusts the initial stack address in the C environment initialization routine ,c_int00 (contained in boot.asm (boot.obj in RTS library), to align it on an even address boundary. If the stack address is not aligned on an even boundary the address is adjusted to the proper alignment. To avoid wasted space due to padding of the starting address for the stack, it is best to align the stack on an even address boundary when linking. The linker "align" keyword may be used to accomplish this. For example:

```
SECTIONS
{
.stack : { align(2) }
}
```

The compiler uses the following rules for establishing the size of the local FRAME for a given function:

- The number of words required for all local symbol declarations (including padding for alignment when necessary).
- The number of words required to store intermediate results that could not otherwise be maintained in registers.
- The number of words required to pass the argument list for the longest argument string among all functions called by the current function.
- Extra word to store value of frame pointer if the size of the local variable space exceed 127 words. (This limitation is based on the fact that the compiler uses stack relative addressing to access local variables. If the size of the local frame exceed 127, then the compiler can no longer use stack relative addressing because the offset will exceed the limit of 127. In this case the compiler will use ARn addressing and will preserve the current ARn value in a temporary location when performing nested function calls.)
- Padding to ensure stack is always aligned on even address boundary when entering this and subsequent functions.

How the compiler reallocates space on the stack? On entering any function, the compiler will first push the contents of any save on entry registers that it may have used for performing calculations or storing intermediate results. It then establishes space for the local function frame, by using the FRAME instruction to adjust the current stack pointer. The order in which space is used within the local frame is:

- space for compiler temporaries
- space for local variables
- space for argument block (arguments passed to functions called within this function)
- return address (for subsequently called functions)
- space to save local frame

Accessing Memory-mapped Registers

TIP: **(C2x/ C2xx, C5x,C54x) Prefer C- macros or "asm" statements versus pointers to access memory-mapped registers.** Using pointers to access memory-mapped registers forces the compiler to create extra space to store the address and extra cycles to load ARn for addressing. Using macros saves one cycle and two words of memory (one in data space for storing the address and one in program memory for nop instruction) due to the capabilities for storage of immediate operands. This can be seen in the following C54x example:

Using volatile pointers: (worse)

```
volatile unsigned *SPC0 = (unsigned *)0x22;*SPC0 = 0x0000;
```

Generates:

```
MVDM *(_SPC0),AR1
nop
ST #0,*AR1
```

===> 5 words

Using macro-defined pointers: (better)

```
#define SPC0 (volatile unsigned *)0x22
*SPC0 = 0x0000;
```

Generates:

```
STM #34,AR1
ST #0,*AR1
```

===> 4 words

Using asm statement: (best)

```
extern volatile unsigned SPC0; asm("_SPC0 .set 0x22");
SPC0 = 0x0000;
```

Generates:

```
ST #0,*(_SPC0)
```

The reference to _SPC0 is resolved correctly at assembly time.

===>3 words

C3x: C pointers is an efficient method to access memory-mapped registers due to the well-supported ARn indirect addressing mode.

Looping

Looping is one of the most common operations in DSPs. Some general suggestions before looking into TMS320 specific C-coding style guidelines for loops.

- ***The usage of -o3*** option in TMS320 compilers achieves time-efficient code generation for loops by enabling loop unrolling and delayed instructions. However this will increase your loop code size. If code size is a major concern, use the ***-ms option*** together with the -o3 option to disable loop unrolling and delayed instructions but still keep the other optimizations that the -o3 offers.
- In TMS320 compilers **up or down-loops don't affect code generation efficiency**. The compiler will automatically convert all the up-count loops to down-count loops to facilitate the usage of repeat instructions and branch conditionals.
- ***Avoid function calls or control statements inside critical loops:*** Even when a function call is controlled by an IF inside a loop, the fact that it *might be* called inhibits useful code optimization. Also, remember that the more deeply nested a loop is the less efficient loop mechanism would implement. Avoid deeply nested loops.
- ***Split-up loops comprised of two unrelated operations***: This is specially true if the loop split could become repeat single loops.

TMS320 Loop Implementation - Analysis

FOR loops can be implemented by a TMS320 compiler via repeat instructions or conditional branches. Ideally a FOR loop should be reduced to a simple RPT instruction (repeat block or repeat single). However many times this is not the case and the inefficiencies may be partially caused by the code style itself.. Let's illustrate this point with the following C54x dot-product example in Appendix E (Code 1).

Option -o3 to optimize loops was used , but we still end up with non-optimal code. Two inefficiencies are noted: *no repeat single instruction is being generated and also an initial conditional branch precedes the loop implementation*. Following, an analysis of why this happens is presented**.**

No generation of repeat instruction : In the dot-product case study, the C54x compiler generates a repeat block even when potentially could generate a repeat single. Typically generation of repeat blocks in TMS320 compilers is easier than generation of repeat singles. TMS320 code generation tools always generate "intermediate repeat blocks" first and then try to replace repeat blocks with repeat singles. This replacement process involves pattern matching techniques that attempt to locate where the RC (repeat counter) register is loaded so that the appropriate operand is used for the counter in the repeat single instruction. This pattern matching is easier to implement in load/store architectures like the C3x (just search for an "LDI xx,RC" . In the C54x, this search is more difficult because it has more instructions that could potentially initialize the RC register. Based on this, it's advised to write FOR loops the simplest way possible. One solution is presented in the following guideline:

TIP: **(All) For the upper limit of a FOR loop, use a constant or a variable with a "const" attribute. If you have to use a regular variable, try function inlining to achieve equivalent results.** As mentioned before, the use of a constant value (either a number, #define or a variable with a const attribute) for the upper limit in a FOR loop facilitates the generation of repeat instructions. This is specially true for repeat singles because the value for the repeat counter can be determined at compile time.

Basically, the more friendly loop construct for RPT singles is : for (i=**constant**; i<= **constant**; i++)

If you want to **maintain the FOR upper limit as a variable** (for example if you want to maintain the dot product as a function), you could **make the loop an inlined-function** and passing a constant as a parameter (functionally equivalent to a const). This is illustrated in Appendix E for a C54x dot product. Also, you could try making the FOR upper limit a global variable. The pattern matching techniques described above work better on global variables (global variables patterns are easier to recognize because they have unique labels)..

So far we have been just analyzing very simple loops . The following table illustrates some other possible combinations:

Table 2 Loop Combinations

Sample Code	RPTS (repeat single)	RPTB (repeat block)
for (i=**constant**; i<= **constant**; i++)	Yes,C3x/C4x, C54x	Yes, C3x/C4x, C5x, C54x
	No, C2x/C2xx/C5x	No, C2x/C2xx (Note 1)
for(i=**constant**; i<= **constant**; i+=**constant)**	Yes, C3x/C4x, C54x<if loop code doesn't depend on i or if the compiler is able to remove the code dependence on i>	Yes, C3x/C4x, C5x, C54x
		No, C2x/C2xx (Note 1)
	No, C2x/C2xx/C5x	
for (i=0; i<= **global_var**; i++)	Yes, C54x, C3x/C4x	Yes, C3x/C4x, C5x, C54x
	No, C2x/C2xx/C5x	No, C2x/C2xx (Note 1)
for(i=0; i<= **local_var**; I++)	No, C2xx/C5x/C54x	Yes, C3x/C4x, C5x, C54x
	Yes, C3x/C4x	No, C2x/C2xx (Note 1)
for(i=**non_zero_constant**; i<= var; i++)	No, C2xx/C5x/C54x	Yes, C3x/C4x, C5x, C54x
	Yes, C3x/C4x	No, C2x/C2xx (Note 1)
for(i=**var**; I<= **var**; i++)	No, C2xx/C5x/C54x	Yes, C3x/C4x, C5x, C54x
	Yes, C3x/C4x	No, C2x/C2xx (Note 1)

Note 1. C2x and C2xx devices lack of a repeat block instruction.

TIP: **(C3x) Use signed integer types in FOR upper limit and iteration counter.** In the C3x case the RC register is a signed register (in the C2xx/C5x/C54x is unsigned). If you use unsigned variables for FOR loops, the compiler will not be able to produce a RPTB because the unsigned dynamic range(16 bits) might exceed the signed dynamic range(15 bits) . The compiler can't prove that it will never exceed the highest positive value.

One it's recommended to use <= instead of < because the compiler can load the block repeat counter automatically without an additional subtract by one being required. This doesn't apply when using a constant as the upper limit because the compiler is smart enough to produce a repeat instruction with one counter less.

Initial Conditional Branch

The generation of the initial conditional branch is due to the way the FOR loop is written. Given the information that the code provides (data type for variable n is signed int) there is no way that the compiler can guarantee that the FOR loop will execute at least once ... therefore the compiler has to add a conditional branch to check if n equals 0 to bypass the loop. The solution? modify your code around to guarantee that to the compiler as explained in the following code generation tip.

***TIP:* (ALL) Select the correct data type of your FOR loop control variables to guarantee the loop will execute at least once.** You can remove the conditional check for a no loop by:

- Using constant upper limits (guideline given above to produce RPT instructions). Notice that that guideline also solves our other inefficiency problem: the condition branch to check if the loop will execute at least once because by handling constants the compiler knows in advance how many times the loop should execute.
- Manipulating the variable data type and the loop end-condition to check. For example let's analyze how you can achieve it in a simple loop of the type for (i=0; i<n;i++) :

TIP: (C2x/C2xx/C5x/C54x) Use unsigned variables for the upper limit (n) and use <= instead of < This guarantees that the loop will be repeated at least once. To illustrate this recommendation, compare the following pieces of code that at first look to be similar:

```
FOR (i=0;i<n;i++) : (original code)
```

if n is signed (a regular int), the compiler cannot make any assumptions on the value of n. Therefore it will generate extra code to bypass the loop when required.

```
FOR (i=0;i<n;i++) : (one step toward the solution)
```

if n is unsigned the compiler knows that n>=0. Because i=0 , the loop may not repeat at least once, therefore extra code is still required to bypass the loop is required increasing code size.

```
FOR (i=0;i<=n1;i++) : (suggested code: n1 = n-1)
```

if n1 is unsigned the compiler knows that n1>=0. Because i=0 , the loop will be repeated at least once, therefore no extra code to bypass the loop is required.

(C3x) No clean solution. In the case of the C3x, we cannot apply the same suggestion given above because the usage of unsigned variables will prevent the generation of repeat blocks. Fortunately, the cycle overhead of an extra branch outside the loop is in most cases minimum.

Control Code and Switch Statements

Code generation for switch and if-then-else statements is highly dependent on the how dense the compare operations are and on the compare capabilities of the device architecture itself. If-then-else statement always use a branch and compare method. In the case of the switch statement, the TMS320 compilers may use one of the following 3 methods to implement it:

- look-up tables (that store the switch labels)
- substract operations on the switch variable selector (check first the smallest selection value and keep substracting to check every path)
- compare and branch

TMS320 compilers will determine the most appropriate method according to how dense the code is.. For highly dense compare code, using switches typically produce better code than an if-then-else implementation.

TIP: **(ALL) For switch statements, assign the smallest selection value to your most commonly used path. For if-then-else statements, place the more common path at the start of the if-then-else chain.** Regardless of the method the compiler uses for switch code generation (see discussion above), assigning the smallest selection value to your most commonly used path will give you overall the best code. This becomes significant when the compiler uses substract operations on the switch variable selector to determine which path to follow. In this case, the checking starts with the smallest selection value. Therefore, you will save instruction cycles if you assign the most probable path to the smallest selection value. Even in the case of the compiler using another switch generation method, following this suggestion will not produce worse code.

Functions

TIP: **(ALL) Use "static inline"or use -pm -oe options performing whole program compilation.** When a function is called only by other functions in the same file, make the **function static**. Likewise, if a global variable is only accessed from functions in the same file, declare the **variable static**. These declarations are particularly helpful to the compiler at optimization level -o3 because if the instruction is small enough, it helps to exploit the in-lining full potential. It's a good idea to organize source files in such a way that minor functions and variables are grouped with the functions that use them and can therefore be declared static.

Another compiler feature that positively affects code generation efficiency is function inlining. Inlining saves the function call overhead and allows the compiler to optimize the function body within the context from which it was called. For example when the function contains a FOR loop, this facilitates the use of RPTBD because there is more code around it that the compiler can take advantage of.

The compiler provides the following options associated with inlining:

- **-o3:** inlines any small-enough function regardless if it's declared as inlined or not.. What is small? the compiler has a set threshold level for the function size that you can change to your own <value> with the -oi<value> option. <value> is given in an unit size that is only meaningful to the compiler. You can find out the size of your functions by using the -on1 option.

  NOTE:
  Do not declare or use volatile variables in a function to be inlined as this will prohibit inlining by the compiler.

- **-x2:** inlines only the functions declared with the inline attribute.

There are 2 types of inlining: static and normal. Static inlining specifies that the function is to be expanded inline and that no code is generated for the function declaration itself. In normal inlining, the function will get inlined but the compiler will also produce a function definition because it assumes that the function can be called from another file. If the function is only used within the file context declare the function static inline.

A similar effect to function inlining can be achieved by implementing functions as macros. C macros will always produce "function inling" regardless the optimization level that you use. On the other hand, with macros you have no protection against duplicated macro name (avoid this by using a cryptic function name for example _$$_myfunction). Another drawback of macros is that they make C-level source debugging difficult. This is because macros are expanded by the C preprocessor and so their definition is not carried through to the code generation process.

Math Operations

TIP: **(ALL) If your code contains a MAC-style operation, make the variable accumulating the result and the MAC operands local variables.** MAC (multiply and accumulate) -type operations are widely used in common DSP algorithms such in dot products, correlations, convolutions, and filtering. The C54x and the C3x/4x compilers are capable of producing optimal code for those algorithms. The C2xx/C5x compiler is not capable to generate MAC-type instructions. This is due to the fact that a C2xx/C5x MAC requires one of the operands to be in program memory. By default the compiler assumes that all variables reside in data memory.

Typical MAC operation:

```
for (i=0;i<N;i++) result += *p1++ * *p2++;
```

The usage of local variables will facilitate allocation of variable to registers (or to an accumulator) (i.e. result, p1, p2 should be local variables). If for example "result" is required to be global, use a temporary local variable and update "result" outside the loop. Also if using pointers, use local pointers instead of global pointers because the modification of global pointers (i.e. *p1++), compliance with ANSI C might force the intermediate update of the pointer variable p1 inside the loop creating unnecessary code (see variable declaration section)

Remember to combine this recommendation with the recommendations for LOOPS to produce the most efficient code for MAC operations. Appendix E presents a case study illustrating the type of C-coding style guidelines to apply to optimize a C54x dot-product .

q15 arithmetic/MACs

TMS320 compilers don't offer direct support for fractional data types (i.e. Q15,..). One solution is to use integer types as a replacement to Q formats as follows:

tms320.h file

```
#ifdef _c5x /* includes c2x/c2xx/c5x/c54x */
typedef short q15;
typedef long q30;
#elif _c6x
typedef short q15;
typedef int q30;
#endif
```

The following examples illustrate basic q15 math operations using the C54x compiler:

```
/* q15 arithmetic/accumulation examples */
#define N 100
extern int dotp(int *x, int *y, int n);
main(){
int i;
int sum;
int *x, *y, *z, *w;
int n = N;

/* CASE 1: typical Q15*Q15=Q30 multiply */

*w = ((long)*x * (long)*y)>>15;
/* Method 1: good: ansi compliant q15 *q15=q30 and store in z the
upper 16MSbits */
dummy(w);

*w = (int) (*x * *y)>>15;
/* Method 2: generates the same code due to a non-ansi compliant
feature of TMS320 compilers. Prefer method 1 */
dummy(w);

*w = ((long) (*x * *y)) >>15;
/* Method 3: generates the same code due to a non-ansi compliant
feature of TMS320 compilers. Prefer method 1 */
dummy(w);
```

```
/* CASE 2: typical Qxx accumulation */

*z = dotp(x,y,n);

dummy(z);
}

static inline int dotp (int *x,int *y,int n) {

int sum=0;
int i;
long longsum;

#if 0
for (i=0;i<n;i++) /* good: int accumulation: RPT MAC in version 1.2
*/
sum += (*x++ * *y++);
#endif

#if 0
for (i=0;i<n;i++) /* q15 accumulation: RPTB (MPY,add/shift) */
sum += (*x++ * *y++)>>15;
#endif

for (i=0;i<n;i++) /* q30 accumulation : might not be as code
efficient but more precise: */
longsum += (long) (*x++ * *y++);
sum = (int)(longsum >>15); /* q15 storage */

return sum;

}
```

Acknowledgments

Special thanks to George Mock , Chris Vick and Chris Wolf for their valuable inputs during the development and review process of this application report. Also, we acknowledge the contribution of previous related work by Alex Tessarolo, Mark Paley and David Bartley.

Appendix A. Summary of Guidelines

Table 3 Guideline Usability by Type and Version

	C2xx/C5x (version xx)	C54x (version 1.2)	C3x (version 5.0)
1. General Guidelines			
Avoid removing registers for C-compiler usage (-r option)	yes	yes	yes
To selectively optimize functions - Place into separate files	yes	yes	yes
Use the least possible volatile variables	yes	yes	yes
For best optimization , use program level optimization (-pm option) in conjunction with file level optimization (-oe option)	yes	yes	yes
2. Variable declaration			
<See also Loops section for specific recommendations for variables associated with loops>			
Local vs. Globals variables - preference	global	local	(NR) but somewhat toward locals
Declare globals in file where they are used the most	yes	yes	yes
Allocate most often used elements of an structure, array or bit-fields in the first element, the lowest address or LSB respectively	yes	yes	yes
Prefer unsigned variables over signed.	yes	yes	yes
Group together math operations involving the same data type.	yes	yes	no
Pay attention to data type significance and optimize code accordingly	yes	yes	yes
3. Initialization of variables			
Initialize global vars with constants at load time	yes	yes	yes
When initializing different variables with the same constant, rearrange your code	yes	yes	yes
Use memcopy when copying an array variable into another	yes	yes	yes
4. Memory alignment and Stack management			
Group all like data declarations together, listing 16 bit data first.	yes	(NR)	(NR)
Use the .align linker directive to guarantee stack alignment on an even address	yes	(NR)	(NR)
5. Accessing memory-mapped registers			
Prefer C- macros or "asm" statements versus pointers to access memory-mapped registers.	yes	yes	(NR)
NR = irrelevant			

	C2xx/C5x (version xx)	C54x (version 1.2)	C3x (version 5.0)
6. Loops			
Split-up loops comprised of two unrelated operations:	yes	yes	yes
Avoid function calls inside critical loops	yes	yes	yes
Select the type of your FOR loop control variables to guarantee the loop will execute at least once.	yes	yes	yes
For the upper limit of a FOR loop, use a constant or a variable with a "const" attribute. If you have to use a regular variable, try function inlining	yes	yes	yes
Use signed integer types in FOR upper limit and iteration counters	no	no	yes
7. Control functions			
For switch statements, assign the smallest selection value to your most commonly used path For if-then-else statements, place the more common path at the start of the if-then-else chain	yes	yes	yes
8. Functions			
Use "static inline"	yes	yes	yes
9. Math Operations			
If your code contains a MAC-style operation, make the variable accumulating the result and the MAC operands local variables	yes	yes	yes
NR = irrelevant			

Appendix B. Instructions Used by the C54x Compiler

Table 4 Instructions Used by the C54x Compiler

ABS	ADD	ADDM
AND	ANDM	B
BACC	BANZ	BC
BITF	CALA	CALL
CMPL	CMPM	CMPR
DADD	DLD	DRSUB
DST	DSUB	FCALA
FCALL	FRAME	FRET
FRETE	LD	LDM
LDU	MAC	MAR
MPY	MPYA	MPYU
MVDD	MVDK	MVDM
MVMM	NEG	OR
ORM	POPM	PORTR
PORTW	PSHM	READA
RET	RETE	RETF
RPT	RPTB	RSBX
SFTA	SFTL	SSBX
ST	STH	STL
STLM	STM	SUB
XOR	XORM	

Appendix C. Instructions Used by the C5x/2xxCompiler

Table 5 Instructions Used by the C5x/2xx Compiler

ABS	ACC	ACCL
ADD	ADDB	ADDH
ADDK	ADDS	ADLK
ADRK	ADRK	AND
ANDB	ANDK	APAC
APL	B	BACC
BANZ	BIT	BLDD
BLKD	BNV	BSAR
CALA	CALL	CMPL
IN	LAC	LACB
LACK	LACT	LALK
LAMM	LAR	LARK
LDPK	LMMR	LRLK
LT	MAR	MPY
MPYK	MPYU	NEG
NOP	OPL	OR
ORB	ORK	OUT
PAC	PSHD	RET
RPTB	RPTK	SACB
SACH	SACL	SAMM
SAR	SATH	SATL
SBB	SBLK	SBRK
SFL	SFR	SPAC
SPH	SPL	SPL
SPLK	SUB	SUB
SUBH	SUBK	SUBK
SUBS	TBLR	XOR
XORB	XORK	XPL
ZAC	ZALH	ZALS

Appendix D. Instructions Used by the C3x/4x Compiler

Table 6 Instructions Used by the C3x/4x Compiler

absf	absi	addf
addf3	addi	addi3
and	and3	andn
andn3	ash	ash3
b	bu	call
cmpf	cmpf3	cmpi
cmpi3	dbu	fix
float	frieee	lbu
ldf	ldfge	ldflt
ldi	ldige	ldile
ldilt	ldp	load
lsh	lsh3	mb
mb0	mh0	mh1
mpyf	mpyf3	mpyi
mpyi3	negf	negi
nop	not	or
or3	pop	popf
push	pushf	rcpf
reti	rets	rnd
rol	ror	rpts
stf	sti	stik
subf	subf3	subi
subi3	subrf	subri
toieee	tstb	tstb3
xor	xor3	

Appendix E. A Dot Product Example: C54x Study Case

C code	Corresponding Assembly Code (-o3 option)

/* CODE 1: asm code have initial branch conditional and no MAC generation */
 	
#define N 1000
int x[N],y[N];
int sum;
 	
main(){
int i;
int n;
 	
for (i = 0; i < n; i++) sum += x[i] * y[i];
 	
}
	Main:
	SSBX SXM
	LD *(AL),A
	BC L4,ALEQ
	; branch occurs
	SUB #1,A,A
	STLM A,BRC
	STM #_x,AR2
	RPTBD L4-1
	STM #_y,AR3
	; loop starts
	L3:
	MPY *AR3+,*AR2+,A
	ADD *(_sum),A
	STL A,*(_sum)
	; loop ends
	L4:
	RET

/*CODE 2: by making the variable accumulating the result a local a MAC is generated but still have the conditional branch and a RPTB */
#define N 1000
int x[N],y[N];
int sum;
 	
main()
{
int i;
int n;
int sum_local;
for (i = 0; i < n; i++)
sum_local += x[i] * y[i];
sum = sum_local;
 	
}
	_main:
	
	SSBX SXM
	LD *(AL),A
	BC L4,ALEQ
	; branch occurs
	SUB #1,A,A
	STLM A,BRC
	STM #_x,AR2
	RPTBD L4-1
	STM #_y,AR3
	; loop starts
	L3:
	MAC *AR3+, *AR2+, A, A
	nop
	nop
	; loop ends
	L4:
	RETD
	STL A,*(_sum)
	; return occurs

C code | **Corresponding Assembly Code (-o3 option)**

C code:

```
/* CODE 3: change the upper limit
to a constant to force RPT single.
Notice that the initial branch
conditional also went away */

#define N 1000
int x[N],y[N];
int sum;

main()
{
int i;
int n;
int sum_local;
for (i = 0; i < N; i++)
sum_local += x[i] * y[i];
sum = sum_local;

}
```

Corresponding Assembly Code (-o3 option):

```
_main:
STM #_x,AR3
STM #_y,AR2
RPT #999
; loop starts
L2:
MAC *AR2+, *AR3+, A, A
nop
nop
; loop ends
L3:
RETD
STL A,*(_sum)
; return occurs
```

C code:

```
/* CODE 4: this will also be
possible by making the loop an
inlined function */


#define N 1000
int x[N],y[N];
int sum;
int n;

main()
{
sum = dotp(x,y,N);
}
inline int dotp (int x[], int y[],
int n)
{
int i;
int sum_local;
for (i = 0; i < n; i++)
sum_local += x[i] * y[i];
return (sum_local);
}
```

Corresponding Assembly Code (-o3 option):

```
_main:
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>
ENTERING dotp()
STM #_x,AR3
STM #_y,AR2
RPT #999
; loop start
L2:
MAC *AR2+, *AR3+, A, A
nop
nop
; loop ends
L3:
;
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<<<<<<<<<<<<<
LEAVING dotp()
RETD
STL A,*(_sum)
; return occurs
```