Freescale Semiconductor
Application Note

Document Number: AN3674
Rev 0, 04/2010

C Code Optimization Examples for the StarCore® SC3850 Core

This application note describes essential techniques to use when optimizing C code targeted for the SC3850 core using CodeWarrior® build tools. The document covers the use of key features aimed at providing necessary information for the compiler to take advantage of StarCore® features, as well as generate code that effectively uses information particular to the application being developed. More specifically, the note details the use of directives, keywords, data types, pragmas, and intrinsics in the application code. A DSP kernel is optimized step by step to show the whole development flow. The document assumes some experience with the C language, StarCore DSPs, and the CodeWarrior IDE.

Contents

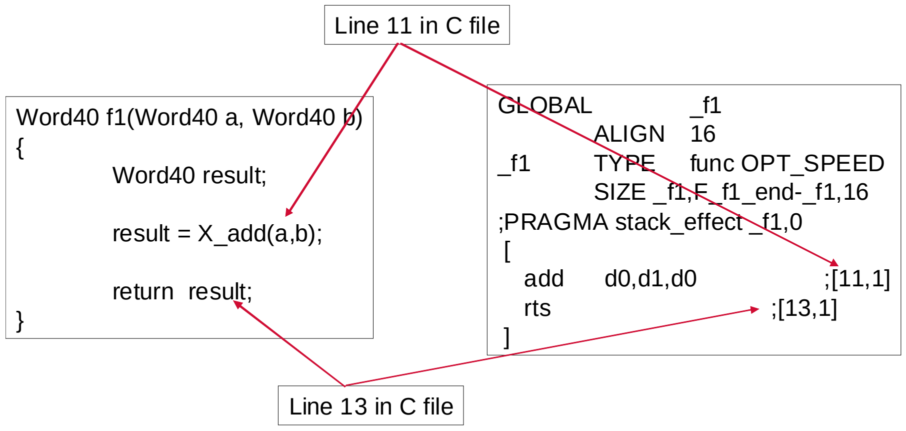
1 Introduction

To effectively port and optimize C code for an embedded application, the programmer must become intimately familiar with the features of the target processor that will eventually execute the application. Knowing the functional units available and the parallelism they offer, the data types and sizes that are supported, the paths to data and instruction memory will allow the developer to write C code that is tuned for that particular architecture. For example, we like to use dual MACs in SC3850 in some applications such as vector dot product. Dual MACs require that multiple data samples are preloaded in operating registers. To this end, we may need to takes advantage of the full core data bandwidth (2 64-bit data buses) and use proper intrinsics to move data into or from memory, resulting in efficient code.

The application software developer uses the software development tools to write application software for that particular architecture. The compiler maps the high-level C code to the target platform, preserving the defined behavior of the application. Understanding how the development tools generate code and how to use them effectively is very important to writing code that will achieve the desired results. Although the compiler will generate code tuned to a particular architecture, the developer generally must provide information to assist the compiler in generating optimal code.

A general procedure of writing and optimizing C code is given below. This application note will cover the first two items.

1. Start with C code.
   - Compile with debug mode and no optimization to verify the functionality first.
   - Enable optimizations with level O1, O2, O3, Os, and Og based on application requirements.
   - If performance is satisfactory, stop.
2. Modify C code.
   - Use compiler keywords, intrinsics, pragmas, word-wide memory access, loop unrolling optimization skills to provide information to the compiler and improve the performance based on compiler feedback.
   - The programmer should get in the habit of examining compiler generated assembly code. This procedure is useful because understanding the generated code gives the programmer information which can be used to modify the C source and make further improvements. It can be enabled in the IDE or with the command line option --keep. The compiler generated assembly files are labeled with the ".sl" file extension to avoid them being confused with hand-generated assembly files which have the ".asm" extension. Note that it is easy to correlate generated assembly with C source as the line number of the C source code for a particular assembly instruction is shown in comments. An example is given in Figure 1 to illustrate how to correlate C code with the compiler generated assembly code.



```
Word40 f1(Word40 a, Word40 b)
{
        Word40 result;

        result = X_add(a,b);

        return  result;
}
```

```
GLOBAL          _f1
        ALIGN   16
_f1     TYPE    func OPT_SPEED
        SIZE _f1,F_f1_end-_f1,16
;PRAGMA stack_effect _f1,0
 [
   add     d0,d1,d0              ;[11,1]
   rts                           ;[13,1]
 ]
```

Figure 1. C Source Code and Compiler Generated Assembly Code

3. Write ASM code
   — Identify functions or sections that need to be further optimized using profiling tools.
   — Write these sections in asm.
   — Writing starcore ASM is relatively easy by the nature of the close & protected pipeline of the Starcore.

As noted in the introductory paragraph, this document addresses the following:

- Essential techniques to use when optimizing C code targeted for the SC3850 core
- Use of key CodeWarrior features for the following:
  — provide the compiler with the information necessary to take advantage of StarCore features.
  — generate code that effectively uses information particular to the application being developed.
- Use of the following application code elements:
  — directives
  — keywords
  — data types
  — pragmas

The goal of this document is to guide users how to optimize C code through examples. A general flow of optimization is proposed and a DSP function is given to demonstrate how to follow the flow in real applications. Users are also encouraged to read Reference [7] for C code optimization skills. This document assumes that the user is familiar with the following:

- Elementary skills in C programming language
- Basic experience on CodeWarrior Integrated Development Environment for StarCore DSPs

2 Optimization Target

The optimization target is the SC3850 DSP core. It is very important to know what resources are available in the core. Figure 2 shows the SC3850 block diagram. The challenge facing DSP programmers is to use all the resources available in these advanced architectures effectively. Ideally, the design should maximize the use of both buses (for example, 64 bits being read and 64 bits being written) and all 6 operational units (4 ALUs and 2 AAU or BMU) simultaneously.

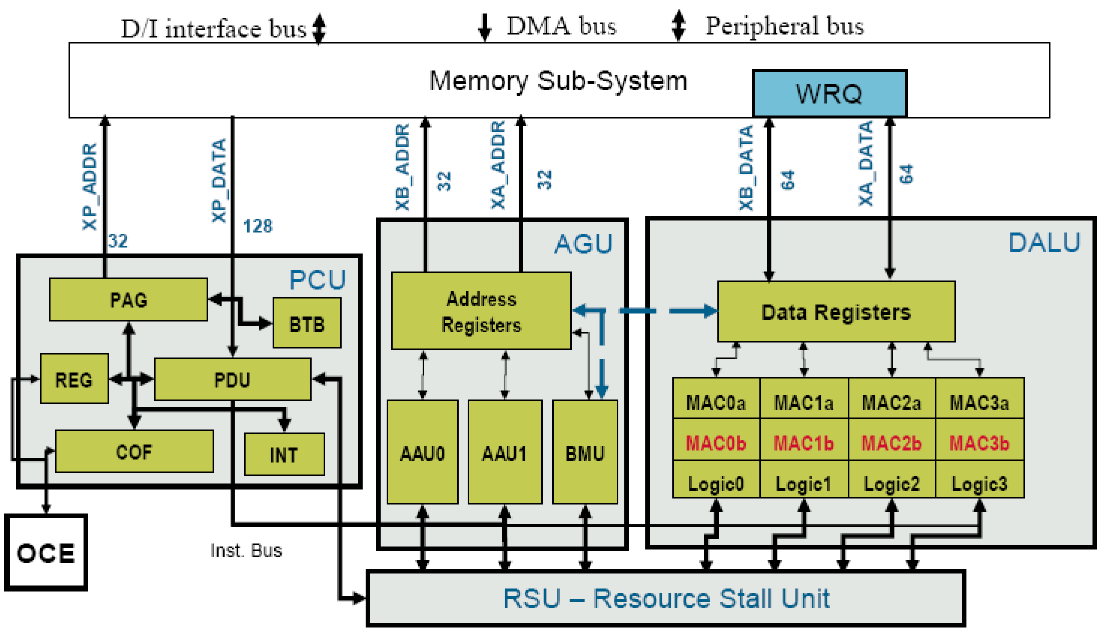


Figure 2. StarCore SC3850 Block Diagram

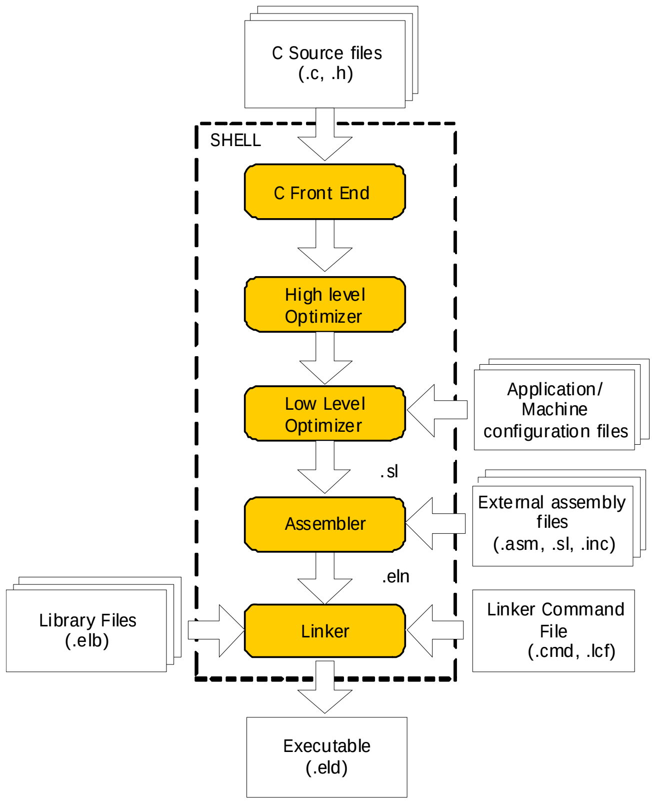
In general, the optimization techniques, such as parallelism, packed data moving, and loop unrolling discussed in this application note, apply to other DSP processors. However, implementation of these techniques is target dependent because individual DSPs can have their own unique architecture. In addition, different intrinsics are used to access the hardware units in the different processors. Therefore, the code optimized for a SC3850 DSP core target might not be optimized for another DSP processor.

3 Optimization Tool: Compiler

C code programming is easier for developers compared with assembly code and thus accelerates your time to market. The CodeWarrior StarCore compiler works as an interface between the hardware SC3850 and the programers. To fully use the resources of the hardware, the programers need knowledge on how the compiler works.

3.1 Compilation Stages

The job of the compiler is to maintain functionality of the application, support special functionality provided by the target and the application. Figure 3 illustrates the StarCore compilation stages.



Stage Descriptions:

- *C Front End (CFE*). Identifies C source files by their file extension, preprocesses the source files, converts the files to Intermediate Representation (IR) files, and then passes these converted files to the high-level optimizer.
- *High level optimizer*. Performs target-independent optimizations: strength reduction (loop transformations), function in-lining, common sub-expression elimination, loop invariant code, constant folding and propagation, jump to jump elimination, and dead storage/assignment elimination (remove redundant variables and value assignments). For detailed information and examples, please see reference [1]. This stage does not consider the DSP architecture structure (registers, functional units, and so on).
- *Low level optimizer*. Carries out target-specific optimizations: instruction scheduling, register allocation, software pipelining, condition execution and predication, speculative execution, post-increment detection, and so on.
- *Assembler*. Takes the optimized assembly files, together with any specified external assembly files, outputting these files to the linker.
- *Linker*. Combines the assembly object files, extracts any required object modules from the library, and produces the executable application. Linker also performs optimizations: dead code stripping (removing unused functions and duplicate code).

Figure 3. StarCore Compilation Stages

3.2 Optimization Levels and Options

For each compilation, you may specify only one optimization level. Each level is a balance between code density and speed. At each higher level compilation takes longer, but code execution is faster. Please refer to Reference [1] for more information.

- Level 0 (no optimization). Disables all optimizations and produces unoptimized assembly code. This level is usually used for code debugging.
- Level 1 (target-independent optimizations). This produces linear assembly code.
- Level 2 (target-independent optimizations, plus target-specific optimizations). This yields parallelized assembly code.
- Level 3 (target-independent optimizations and target-specific optimizations, plus global-algorithm register allocation by the low-level optimizer). The generated code usually is faster than Level-2 code.

With Optimization level 1, 2, and 3, you may add two supplemental optimizations:

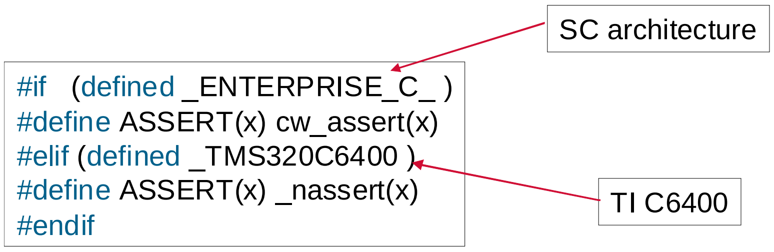
- *Space optimization*. The optimizer favors program size over the level you specify. Programs or modules that have been optimized for space require less memory but may sacrifice program execution speed.
- *Cross-file optimization*. The optimizer applies the specified level across all application files at the same time. This yields the most efficient program code. Cross-file optimization is a complex process, and, therefore, significantly increases compilation time. The disadvantage is that since the optimizer can remove function boundaries and eliminate variables, the code becomes difficult to read and debug. Global optimization may not always be desired because we like to make compiler generated code easy to understand. Developers usually apply cross-file optimization at the end of the cycle, after compiling and optimizing source files individually or in groups. The compiler default setting is *no cross-file optimization*.

4 Optimization Techniques

The more information the compiler knows about the application, the more efficient code it can generate. In this section, directives, keywords, and pragmas optimization techniques will be discussed to feed more information to the compiler about the application.

4.1 Use cw_assert for Loop Optimization and Data Alignment

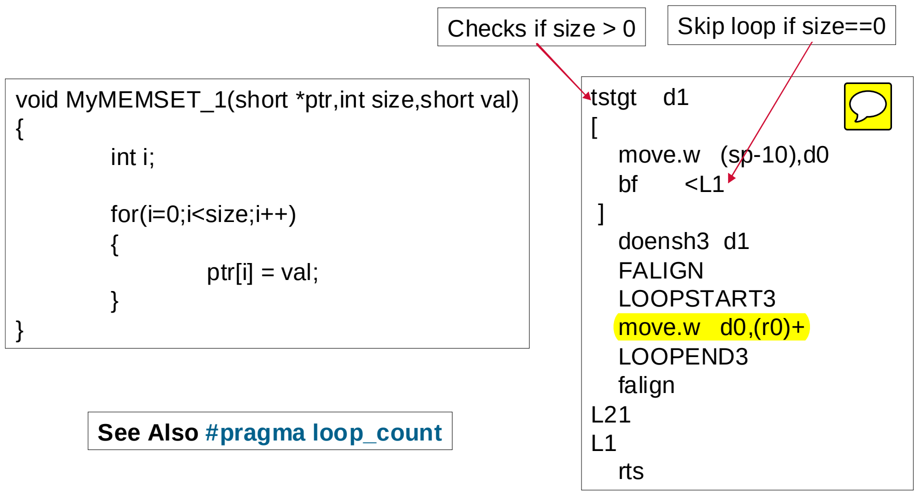
Tell the compiler all about variables to allow the compiler to apply more optimization techniques and generate efficient code. For this purpose, use the **cw_assert** directive. In most cases, there is a pragma that can substitute for **cw_assert**. If you want to keep your code more flexible for another platforms, use **cw_assert**. For example, there is a similar directive **_nassert** in the TI Code Composer Studio. To make the code portable, we can define a macro ASSERT as **cw_assert** if the code is to be run on StarCore 3850, or as **_nassert** if the code is to be run on the TI C6400 core, as shown in Figure 4.



```
#if   (defined _ENTERPRISE_C_ )
#define ASSERT(x) cw_assert(x)
#elif (defined _TMS320C6400 )
#define ASSERT(x) _nassert(x)
#endif
```

Figure 4. Flexible Use of cw_assert

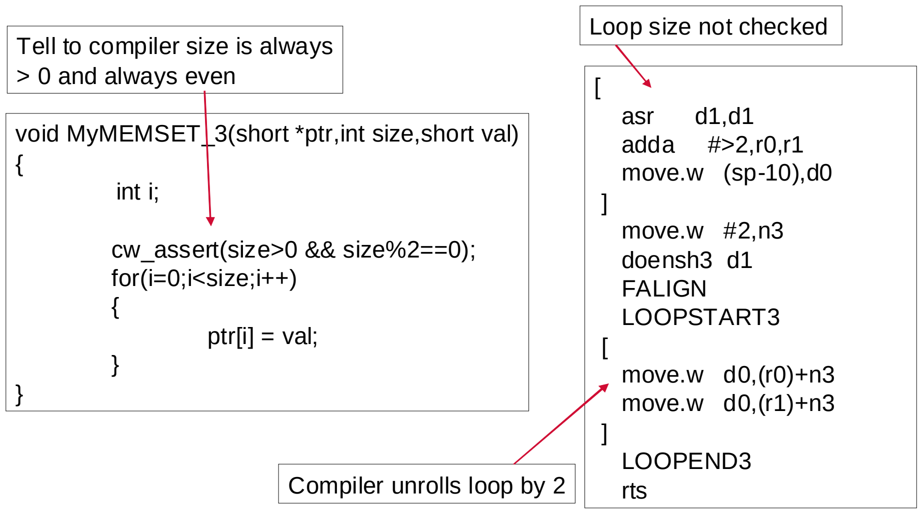
Directive **cw_assert** can be used for loop optimization. An example of using **cw_assert** for loop optimization is shown in Figure 5 and Figure 6. Note that, in this specific example, the number of iterations is half of the original code when using cw_assert.



```
void MyMEMSET_1(short *ptr,int size,short val)
{
        int i;

        for(i=0;i<size;i++)
        {
                ptr[i] = val;
        }
}
```

```
tstgt    d1
[
   move.w   (sp-10),d0
   bf       <L1
 ]
   doensh3  d1
   FALIGN
   LOOPSTART3
   move.w   d0,(r0)+
   LOOPEND3
   falign
L21
L1
   rts
```

Figure 5. Loop Optimization: Original Code



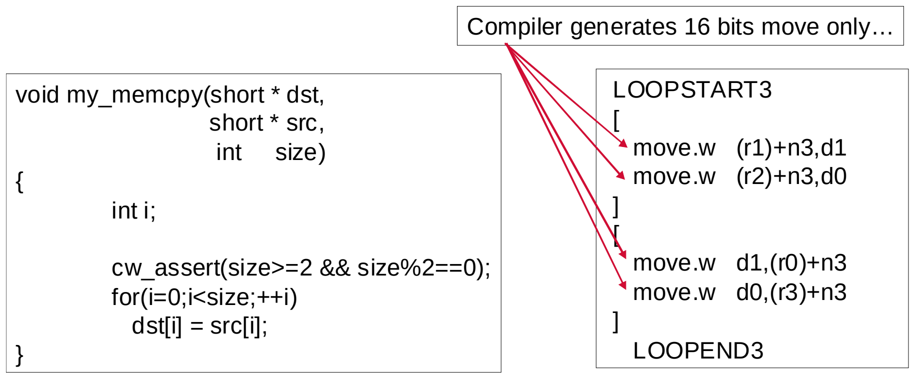
```
void MyMEMSET_3(short *ptr,int size,short val)
{
        int i;

        cw_assert(size>0 && size%2==0);
        for(i=0;i<size;i++)
        {
                ptr[i] = val;
        }
}
```

```
[
   asr      d1,d1
   adda     #>2,r0,r1
   move.w   (sp-10),d0
 ]
   move.w   #2,n3
   doensh3  d1
   FALIGN
   LOOPSTART3
 [
   move.w   d0,(r0)+n3
   move.w   d0,(r1)+n3
 ]
   LOOPEND3
   rts
```

Tell to compiler size is always > 0 and always even
Loop size not checked
Compiler unrolls loop by 2

Figure 6. Loop Optimization: Code with cw_assert

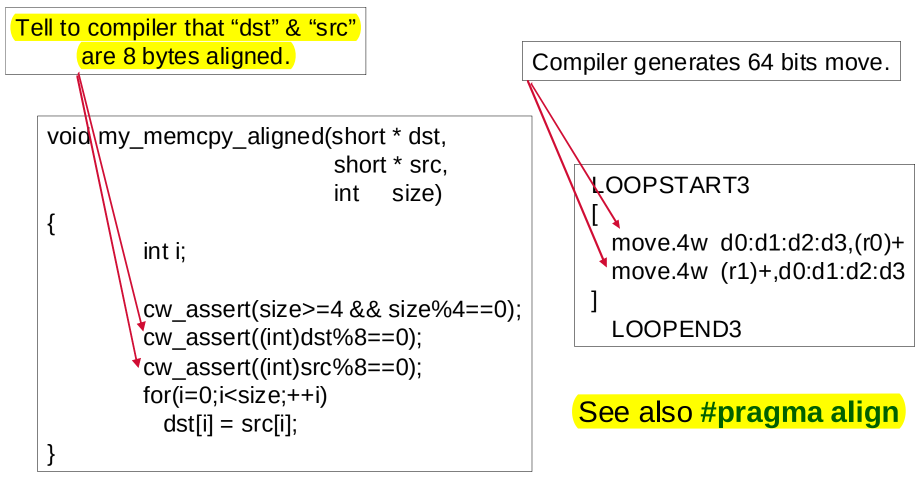
Directive **cw_assert** can be used for data alignment to move packed data, as shown in Figure 7 and Figure 8. Note that the data must be aligned to use packed moves.



```
void my_memcpy(short * dst,
               short * src,
               int     size)
{
        int i;

        cw_assert(size>=2 && size%2==0);
        for(i=0;i<size;++i)
          dst[i] = src[i];
}
```

```
LOOPSTART3
[
  move.w   (r1)+n3,d1
  move.w   (r2)+n3,d0
]
[
  move.w   d1,(r0)+n3
  move.w   d0,(r3)+n3
]
  LOOPEND3
```

Figure 7. Packed Data Move: Original Code


Tell to compiler that "dst" & "src" are 8 bytes aligned.
Compiler generates 64 bits move.
See also #pragma align

```
void my_memcpy_aligned(short * dst,
                       short * src,
                       int     size)
{
        int i;

        cw_assert(size>=4 && size%4==0);
        cw_assert((int)dst%8==0);
        cw_assert((int)src%8==0);
        for(i=0;i<size;++i)
          dst[i] = src[i];
}
```

```
LOOPSTART3
[
  move.4w  d0:d1:d2:d3,(r0)+
  move.4w  (r1)+,d0:d1:d2:d3
]
  LOOPEND3
```

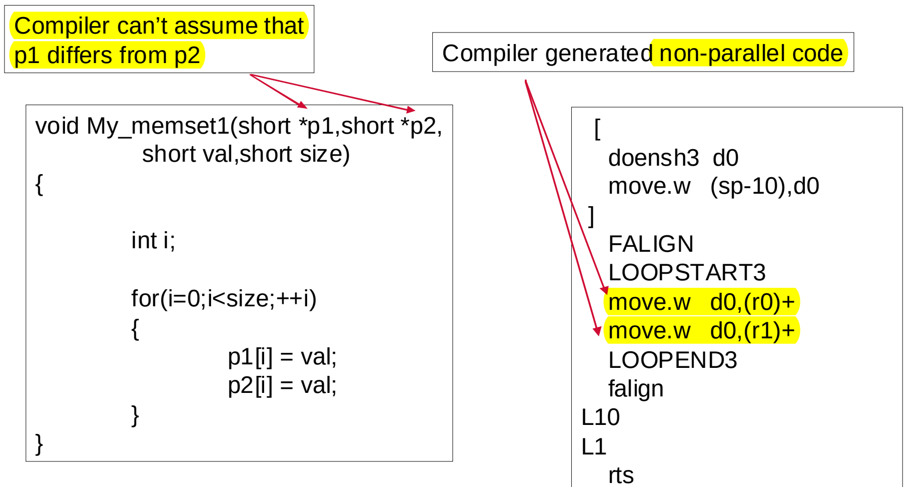
Figure 8. Packed Data Move: Code with cw_assert

4.2 Use restrict to Ensure Non-Aliased Pointers

When pointers are used at the same piece of code, make sure that they cannot point to the same memory location (alias). When the compiler knows that the pointers do not alias, it can put accesses to memory pointed to by those pointers in parallel, greatly improving performance. Communicate this to the compiler by one of two methods:

- Use the **restrict** keyword
- Inform the compiler that no pointers in the program alias anywhere

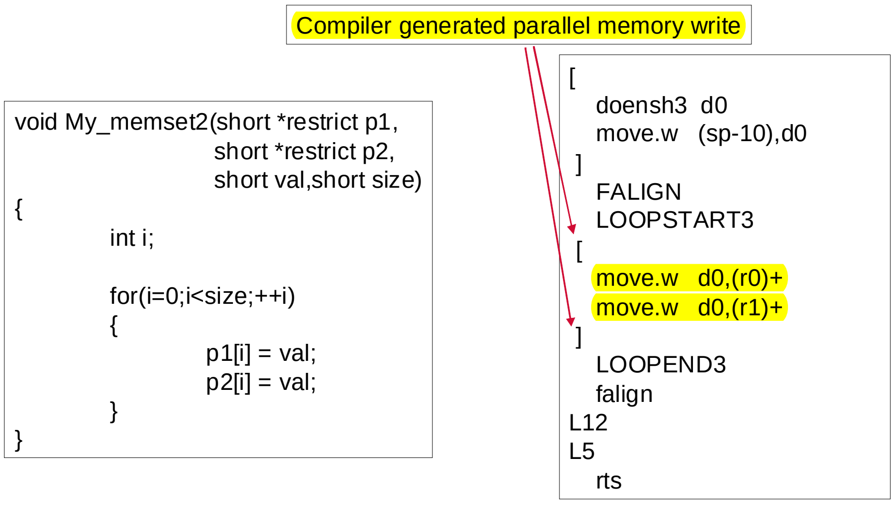
The **restrict** keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by the programmer that within the scope of the pointer declaration, the object pointed to can be accessed only by that pointer. A violation of this guarantee can produce undefined results. Example C code and the compiler generated assembly code are shown in Figure 9 and Figure 10 to illustrate how to use keyword **restrict** to avoid pointer alias.


Compiler can't assume that p1 differs from p2
Compiler generated non-parallel code

```
void My_memset1(short *p1,short *p2,
                short val,short size)
{

        int i;

        for(i=0;i<size;++i)
        {
                p1[i] = val;
                p2[i] = val;
        }
}
```

```
 [
   doensh3  d0
   move.w   (sp-10),d0
 ]
   FALIGN
   LOOPSTART3
   move.w   d0,(r0)+
   move.w   d0,(r1)+
   LOOPEND3
   falign
L10
L1
   rts
```

Figure 9. Possible Aliased Pointers



```
void My_memset2(short *restrict p1,
                short *restrict p2,
                short val,short size)
{
        int i;

        for(i=0;i<size;++i)
        {
                p1[i] = val;
                p2[i] = val;
        }
}
```

```
[
   doensh3  d0
   move.w   (sp-10),d0
 ]
   FALIGN
   LOOPSTART3
 [
   move.w   d0,(r0)+
   move.w   d0,(r1)+
 ]
   LOOPEND3
   falign
L12
L5
   rts
```

Figure 10. No Aliased Pointers using Restrict

Alternatively, if no pointers alias anywhere in the program, then the global option auto restrict can be used. It is applied by passing the following to the compiler shell (scc):

```
-Xcfe "-fl auto_restrict"
```

The programmer must make sure that no pointers alias when using this option. Compiler flag `-Xcfe "-fl auto restrict"` tells the compiler that all pointers are restricted. Using this option can be dangerous; any function implemented with aliased pointers generates a runtime error. Another compiler flag `-Xcfe "-flag=auto_restrict_locals"` tells the compilers that all local pointers are restricted. It may be dangerous to use for the same reason.

4.3 Use const for Constant Propagation and Dead Code Elimination

const is an ANSI keyword. It allows the compiler to perform some well-known optimizations, such as constant propagation and dead code elimination.

4.3.1 Constant Propagation

The compiler cannot assume that an initialized global variable is a constant. There is a possibility that the global variable changed from another file. Thus, the global variable must be loaded before performing an operation on it, as shown in Figure 11. We can use const to tell the compiler that an initialized global variable is a constant to perform constant propagation optimization, as shown in Figure 12.

```
short val = 11;/*GLOBAL VARIABLE*/

short f1(short a)
{
        return a*val;
}
```

```
move.w    _val,d1
 [
    impy      d0,d1,d0
    rtsd
 ]
    sxt.w     d0,d0
```

'val' must be loaded before performing an operation.

Figure 11. Constant Propagation: Original Code

```
const short val_c = 11; /*GLOBAL VARIABLE
*/

short f2(short a)
{
       return a*val_c;
}
```

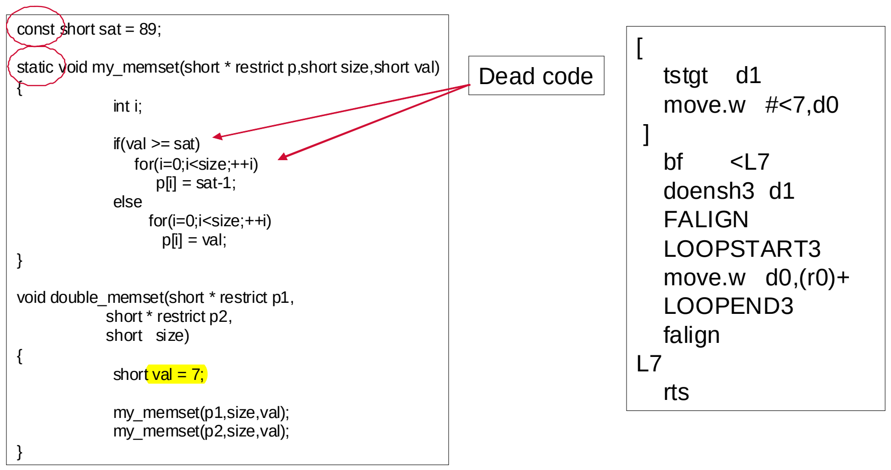
```
[
    impy.w    #11,d0
    rtsd
 ]
    sxt.w     d0,d0
```

Constant propagation performed

Figure 12. Constant Propagation: Code Using Const Keyword

4.3.2 Dead Code Elimination

In Figure 13, the **static** function declaration indicates that `my_memset()` is local to this file and cannot be called from another file. Since `my_memset()` is a static function, the compiler can assume “val” is always equal to 7 (since it cannot be called from another file). Then, the dead code can be removed to reduce the code size. Global optimization produces the same result, but global optimization might not always be desired.

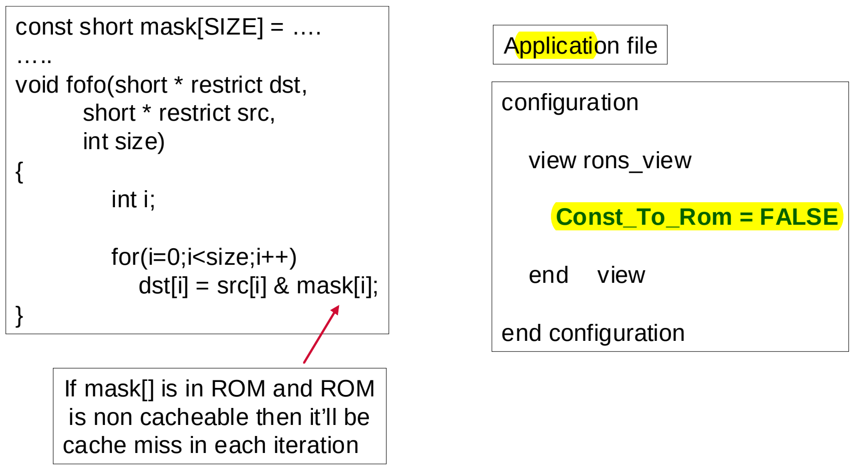


```
const short sat = 89;

static void my_memset(short * restrict p,short size,short val)
{
                int i;

                if(val >= sat)
                    for(i=0;i<size;++i)
                        p[i] = sat-1;
                else
                       for(i=0;i<size;++i)
                         p[i] = val;
}

void double_memset(short * restrict p1,
                  short * restrict p2,
                  short   size)
{
                short val = 7;

                my_memset(p1,size,val);
                my_memset(p2,size,val);
}
```

```
[
    tstgt    d1
    move.w   #<7,d0
 ]
    bf       <L7
    doensh3  d1
    FALIGN
    LOOPSTART3
    move.w   d0,(r0)+
    LOOPEND3
    falign
L7
    rts
```

Figure 13. Dead Code Elimination

All data marked as constant is placed in ROM section by default. In some old versions of the default linker command file, the ROM section is defined as non-cacheable. If constant data is frequently used, the desire is to make constant data cacheable. To do this, change the linker command by using `Const_To_Rom=FALSE` in the application file, as shown in Figure 14. As a result, the compiler places constants in some cacheable section instead of ROM to make cache hits possible.



```
const short mask[SIZE] = ….
…..
void fofo(short * restrict dst,
          short * restrict src,
          int size)
{
              int i;

              for(i=0;i<size;i++)
                 dst[i] = src[i] & mask[i];
}
```

If mask[] is in ROM and ROM is non cacheable then it'll be cache miss in each iteration

```
configuration

   view rons_view

         Const_To_Rom = FALSE

   end   view

end configuration
```

Figure 14. Make Constant Data Cacheable

4.4 Pragmas

Pragmas provide the compiler additional information about how to process certain C statements and hence help enhance the optimization process. The general syntax of using pragmas is:

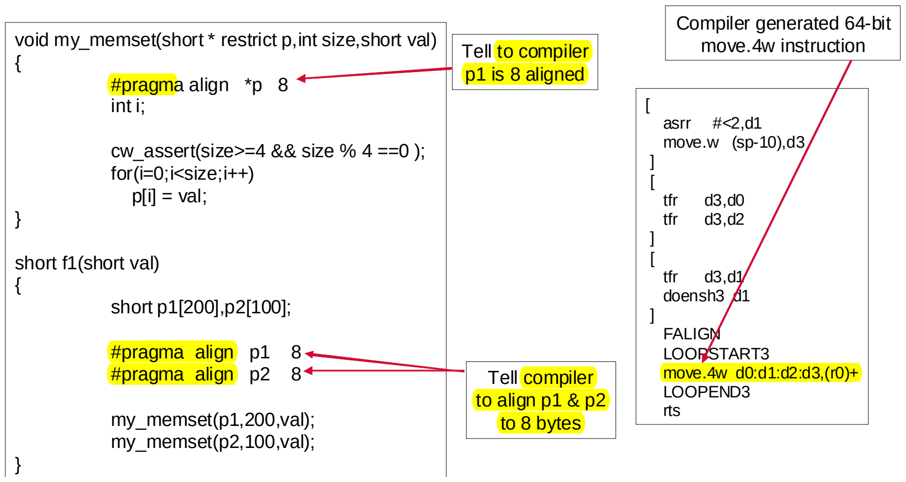
```
#pragma pragma_name [argument(s)]
```

Pragmas apply only to certain contexts as follows:

- *Function pragmas*. Appear only in the scope of the function, after the opening "{"
- *Statement pragmas*. Must be placed immediately before the relevant statement
- *Variable pragmas*. Must follow the variable definition

4.4.1 Use #pragma align for Data and Function Alignment

#pragma align can be used to tell the compiler to put array/structure at aligned memory address. It is also used to tell the compiler that a pointer is pointing to a block that is already aligned. An example is shown in Figure 15.



```
void my_memset(short * restrict p,int size,short val)
{
        #pragma align  *p  8
        int i;

        cw_assert(size>=4 && size % 4 ==0 );
        for(i=0;i<size;i++)
            p[i] = val;
}

short f1(short val)
{
        short p1[200],p2[100];

        #pragma align  p1  8
        #pragma align  p2  8

        my_memset(p1,200,val);
        my_memset(p2,100,val);
}
```

```
[
    asrr    #<2,d1
    move.w  (sp-10),d3
]
[
    tfr     d3,d0
    tfr     d3,d2
]
[
    tfr     d3,d1
    doensh3 d1
]
    FALIGN
    LOOPSTART3
    move.4w d0:d1:d2:d3,(r0)+
    LOOPEND3
    rts
```

Tell to compiler p1 is 8 aligned
Tell compiler to align p1 & p2 to 8 bytes
Compiler generated 64-bit move.4w instruction

Figure 15. Multiple Data Move using #pragma align

Functions should be aligned so that they fall on cache boundaries. **#pragma align** can also be used to align a frequently called function to a cache line to enable instruction cache optimization, as shown in Figure 16. Since L1 instruction cache had 256-byte cache line and L2 unified cache has 64-bye cache line, max (256 bytes) is chosen here and works for both cache levels.

```
void func99(int val)
{
    #pragma align func99 256

    printf("Hi %d\n",val);

    //Do something
}
```

Figure 16. Instruction Cache Optimization

Note that there is a command line switch order for compiler to align ALL functions, such as `-Xicode --min_func_align=256`. Other command line switches related to alignment are listed below.

- `-Xcfe "-fl auto_align8"`. Allows you to specify that all pointer function parameters are 8-byte aligned.
- `-Xcfe --min_all_align=<min>`. Sets the minimum alignment for all objects.
- `-Xcfe --min_array_align=<min>`. Sets the minimum alignment for all arrays.
- `-Xcfe --min_buffer_align=<min>`. Sets the minimum alignment for all buffers (array of characters).
- `--min_struct_align<n>`. Sets minimum structure alignment to specified number of bytes. Default <n> value is 4.

4.4.2 Use #pragma loop_count for Loop Optimization

#pragma loop_count provides information to compiler about loop counter. It must be inserted after the start of the loop. The syntax of using **#pragma loop_count** is listed below. Note that not all fields are mandatory.

```
#pragma loop_count(min_iter, max_iter[,{modulo},[remainder]])
```

— `min_iter`. Minimum number of iterations. The compiler can use this value to remove loop bypass tests.

— `max_iter`. Maximum number of iterations. The compiler can use this value to assess the induction variable range.

— `modulo` and `remainder` pair. The compiler uses these values to unroll loops with dynamic loop count.

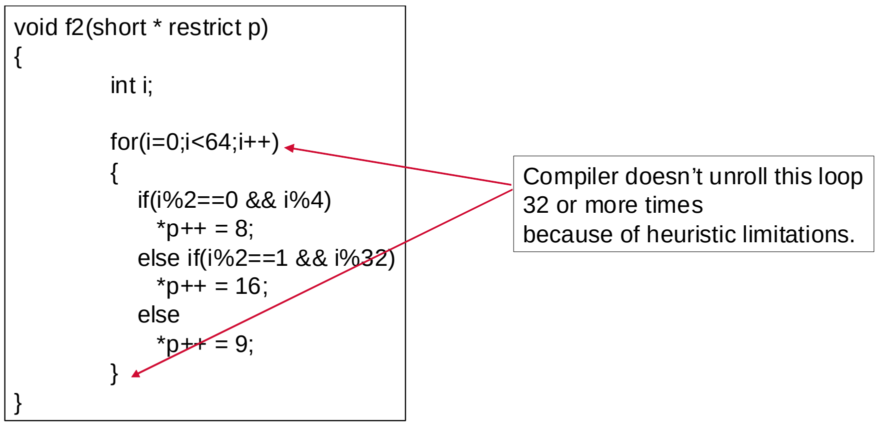
Example 1. #pragma loop_count

Iterate the loop at least once, at most 16 times; the loop count is a multiple of 2:

```
#pragma loop_count (1,16,2,0)
```

4.4.3 Use #pragma loop_unroll for Loop Optimization

#pragma loop_unroll can be used to tell the compiler to unroll the loop. Figure 17 shows some example code. The compiler uses a heuristic approach to determine how to process loops by trading off speed for code size. In this example, compiler optimization heuristics prevents loop unrolling.

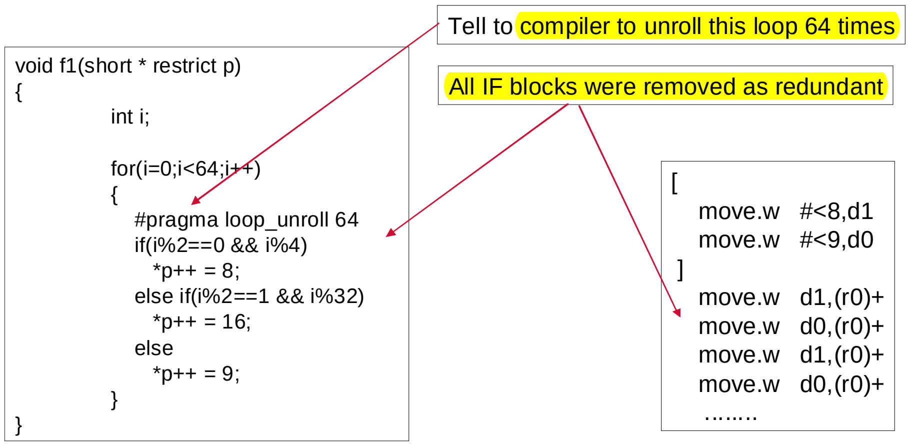


```
void f2(short * restrict p)
{
        int i;

        for(i=0;i<64;i++)
        {
          if(i%2==0 && i%4)
            *p++ = 8;
          else if(i%2==1 && i%32)
            *p++ = 16;
          else
            *p++ = 9;
        }
}
```

Compiler doesn’t unroll this loop
32 or more times
because of heuristic limitations.

Figure 17. Loop without Unrolling

We can use **#pragma loop_unroll** to force the compiler to unroll the loop by 64 times. The resulting assembly code is shown in Figure 18. We can see a significant performance improvement by using loop unrolling because unrolling the loop 64 times eliminates the need for if-else statements.



```
void f1(short * restrict p)
{
        int i;

        for(i=0;i<64;i++)
        {
          #pragma loop_unroll 64
          if(i%2==0 && i%4)
            *p++ = 8;
          else if(i%2==1 && i%32)
            *p++ = 16;
          else
            *p++ = 9;
        }
}
```

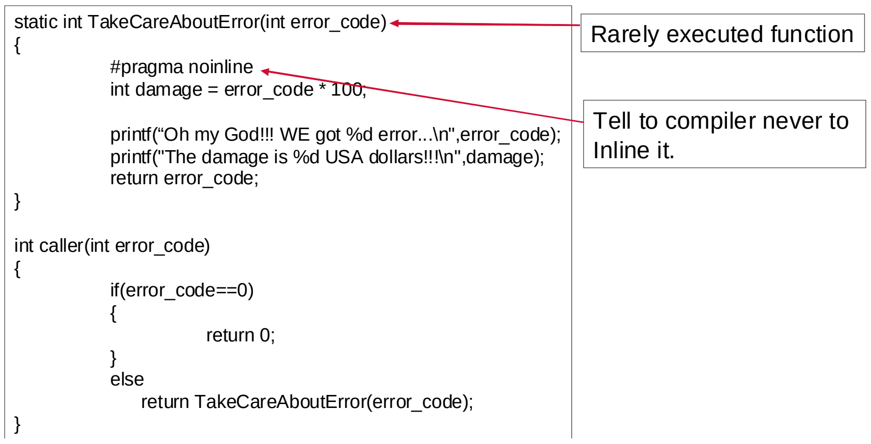
```
[
   move.w   #<8,d1
   move.w   #<9,d0
]
   move.w   d1,(r0)+
   move.w   d0,(r0)+
   move.w   d1,(r0)+
   move.w   d0,(r0)+
   ……..
```

Tell to compiler to unroll this loop 64 times
All IF blocks were removed as redundant

Figure 18. Loop Unrolled Code

4.4.4 Pragma noinline

#pragma noinline directs the compiler not to inline a function. It can be used to profile a function and perform ICache optimization. If a function is rarely executed, we can reduce code size of the caller function by not inlining the called function, Figure 19 shows how to use **#pragma noinline**.

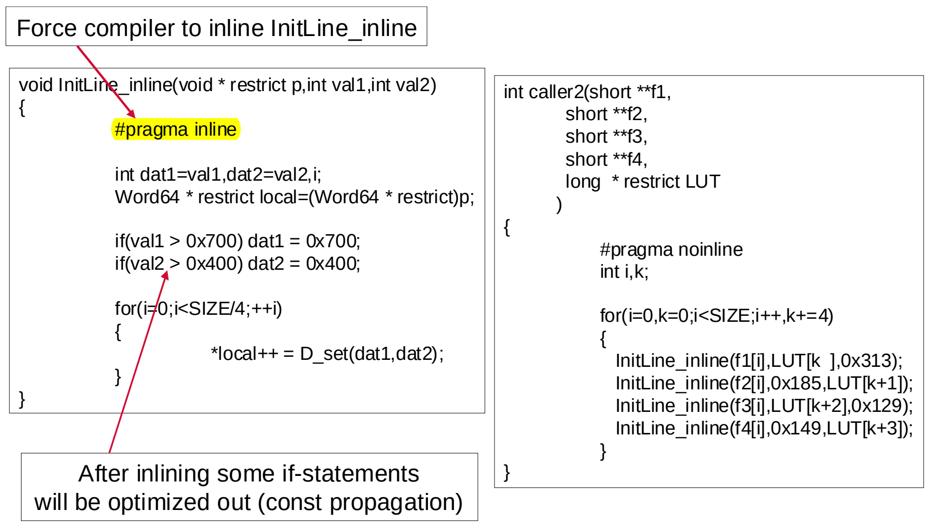

Rarely executed function
Tell to compiler never to Inline it.

```
static int TakeCareAboutError(int error_code)
{
            #pragma noinline
            int damage = error_code * 100;

            printf("Oh my God!!! WE got %d error...\n",error_code);
            printf("The damage is %d USA dollars!!!\n",damage);
            return error_code;
}

int caller(int error_code)
{
            if(error_code==0)
            {
                        return 0;
            }
            else
                return TakeCareAboutError(error_code);
}
```

Figure 19. Use of #pragma noinline

4.4.5 #pragma inline

#pragma inline directs the compiler to inline a function to save the function call overhead due to the cost of increased code size. Figure 20 shows how to use **#pragma inline**.


Force compiler to inline InitLine_inline
After inlining some if-statements will be optimized out (const propagation)

```
void InitLine_inline(void * restrict p,int val1,int val2)
{
            #pragma inline

            int dat1=val1,dat2=val2,i;
            Word64 * restrict local=(Word64 * restrict)p;

            if(val1 > 0x700) dat1 = 0x700;
            if(val2 > 0x400) dat2 = 0x400;

            for(i=0;i<SIZE/4;++i)
            {
                        *local++ = D_set(dat1,dat2);
            }
}
```

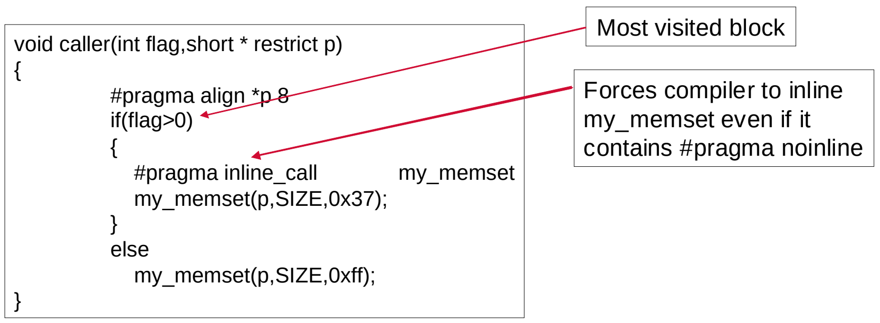
```
int caller2(short **f1,
        short **f2,
        short **f3,
        short **f4,
        long  * restrict LUT
       )
{
            #pragma noinline
            int i,k;

            for(i=0,k=0;i<SIZE;i++,k+=4)
            {
              InitLine_inline(f1[i],LUT[k  ],0x313);
              InitLine_inline(f2[i],0x185,LUT[k+1]);
              InitLine_inline(f3[i],LUT[k+2],0x129);
              InitLine_inline(f4[i],0x149,LUT[k+3]);
            }
}
```

Figure 20. Use of #pragma inline

In the example code, there is a large function called 10 times that would not be inlined by the compiler. Forcing the inlining of the function produced a 2x improvement on the callee function because of constant propagation optimizations (constants passed into called function allowed module runtime functions to be replaced with constant assignments).

4.4.6 #pragma inline_call

#pragma inline_call directs the compiler to inline the next call of the specified function. It must be placed just before that function call. It has no effect on a function call made through a pointer. Figure 21 shows how to use **#pragma inline_call**.



```
void caller(int flag,short * restrict p)
{
        #pragma align *p 8
        if(flag>0)
        {
            #pragma inline_call          my_memset
            my_memset(p,SIZE,0x37);
        }
        else
            my_memset(p,SIZE,0xff);
}
```

Most visited block
Forces compiler to inline my_memset even if it contains #pragma noinline

Figure 21. Use of #pragma Inline_call

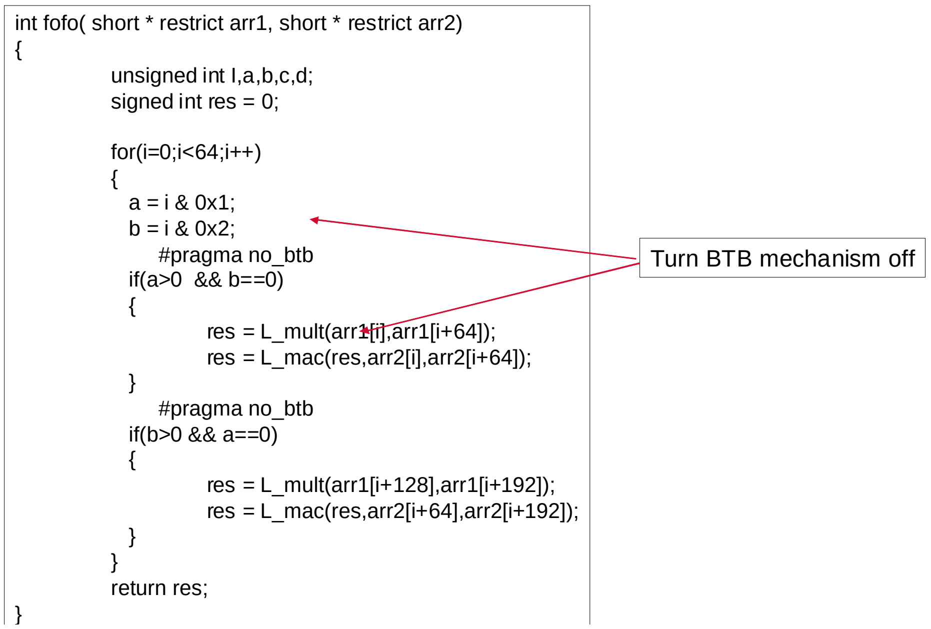
4.4.7 Use #pragma opt_level to Specify Optimization Level

#pragma opt_level controls level of code optimization, at either the function or module level. Valid optimization level values are O0, O1, O2, O3, Os, and O3s. Some examples of using **#pragma opt_level** are listed below.

- Use for debugging purposes by compiling the whole project for speed and a file/function with O0.
- Reduce the size of the whole application by compiling rarely visited functions with size optimization (O3s).
- Use for ICache optimization. Provide better ICache mapping opportunity for frequently called functions.

4.4.8 Use #pragma no_btb to Turn off Branch Target Buffer in the SC3850

In some applications, we may want to turn off the BTB (branch target buffer). For instance, Figure 22 shows the representative example code. Every loop iteration a different if-clause is true, which negates the performance improvement of the BTB. In this case, we may want to turn BTB off.



```
int fofo( short * restrict arr1, short * restrict arr2)
{
        unsigned int I,a,b,c,d;
        signed int res = 0;

        for(i=0;i<64;i++)
        {
          a = i & 0x1;
          b = i & 0x2;
             #pragma no_btb
          if(a>0  && b==0)
          {
                res = L_mult(arr1[i],arr1[i+64]);
                res = L_mac(res,arr2[i],arr2[i+64]);
          }
             #pragma no_btb
          if(b>0 && a==0)
          {
                res = L_mult(arr1[i+128],arr1[i+192]);
                res = L_mac(res,arr2[i+64],arr2[i+192]);
          }
        }
        return res;
}
```

Figure 22. Turn off BTB using #pragma no_btb

5 An Example Using a DSP Kernel

This section of the application note demonstrates how to optimize a DSP kernel by using the skills introduced in previous sections. The example walks you through the code development flow introduced in Section 1, “Introduction” and shows how to improve the code based on the feedback of the compiler. This example was created with the CodeWarrior StarCore Compiler, v23.7.1 (3x50). It is possible to use other compilers, but the output may differ.

This example develops a DSP kernel called **complex_mult**. This routine takes two input vectors and calculates one output vector. The elements in the output vector are the product of the corresponding elements in the input vectors. The inputs and output are 16-bit short data. Note that fractional multiplication is used in the DSP kernel. The prototype of the kernel is defined as:

```
function:  int complex_mult(short* coef, short* input, short* result, int n);
parameters:
                coef: pointer to input vector 1,  [0, 2, 4, 6..] real,
                                                  [1, 3, 5, 7..] imaginary,
                input: pointer to input vector 2, [0, 2, 4, 6..] real,
                                                  [1, 3, 5, 7..] imaginary,
                result: pointer to result vector  [0, 2, 4, 6..] real,
                                                  [1, 3, 5, 7..] imaginary,
                n:  number of elements vectors.
```

We will follow the procedure introduced in Section 1, “Introduction” to develop this kernel.

5.1 Start With Natural C Code Without Optimization

Fractional data types are not natively supported in natural C. For fractional arithmetic, fractional intrinsics must be used to communicate to the compiler that the data is fractional. Therefore, we have an issue here: how to calculate fractional multiplication because fractional arithmetic is not supported in natural C. Note that the difference between fractional and integer multiplications is the location of the decimal point. Thus, integer multiplication can be used for fractional multiplication with a proper bit shift. In this implementation, 32-bit intermediate results are calculated first by integer multiplication instructions. Then, the 32-bit data is shifted to the right by 15 bits and cast to 16-bit short data for output. It is straightforward to write the kernel in natural C. The C code of the kernel is listed as below. We can see from the code that most computation happens inside the loop, and, thus, we will focus on the loop for code optimization.

```
int complex_mult_nat(short* coef,
                 short* input,
                 short* result,
                 int n)
{
   int i, real, imag;

   for(i=0;i<2*n;i+=2)
   {
      real = (input[i]*coef[i]) - (input[i+1]*coef[i+1]);
      imag = (input[i]*coef[i+1]) + (input[i+1]*coef[i]);
      result[i] = (real >> 15);
      result[i+1] = (imag >> 15);
   }

return 0;
}
```

Figure 23. Natural C Code of the complex_mult Kernel

Compile the natural implementation with debug mode and verify the functionality. We can try different optimization levels on the code. Typically, -O3 is used for speed optimization and -Os is used for size optimization. Use --keep option to produce .sl file, which shows the compiler generated assembly code. Figure 24 shows the assembly code compiled with -O3 optimization level.

```
LOOPSTART3
[
   move.w    (r1)+n3,d0                          ;[209,1]
   move.w    (r0)+n3,d4                          ;[209,1]
]
[
   impy      d4,d0,d2                            ;[209,1]
   move.w    (r5)+n3,d1                          ;[209,1]
   move.w    (r4)+n3,d3                          ;[209,1]
]
[
   imac      -d3,d1,d2                           ;[209,1]
   impy      d4,d1,d1                            ;[210,1]
]
[
   imac      d3,d0,d1                            ;[210,1]
   asrr      #<15,d2                             ;[211,1]
]
[
   asrr      #<15,d1                             ;[212,1]
   move.w    d2,(r2)+n3                          ;[211,1]
]
   move.w    d1,(r3)+n3                    ;[0,1]
   LOOPEND3
```

Figure 24. Compiler Generated Assembly Code for Natural C Implementation

Six instruction sets are used inside the loop to produce one output sample (real and imaginary parts), which is 6 cycles per sample. Obviously, there is big room for optimization because only a small portion of the resource in SC3850 is used. Recall that there are 2 64-bit data buses and 6 operational units (4 ALUs and 2 AAU or BMU) in SC3850.

5.2 Optimizing C Code

Before optimizing the C code, you need to know what the optimization limit of the code is. In other words, what is the best cycle count we can achieve? Normally, the number of MACs and the data bus width limit the best performance. For the **complex_mult** kernel, the bottleneck is the data bus width. The SC3850 can access 128 bits per cycle. To calculate one complex output. we need to load four 16-bit data sets for 2 input elements (real and imaginary) and store two 16-bit data sets for 1 output. That requires 96 bits. Thus, the best performance is ~0.75 (96/128) cycle per sample. If the number of elements **n**=1024, the best cycle count is around 750. The performance of the natural C code developed in Section 5.1, "Start With Natural C Code Without Optimization" is not efficient compared with our goal of 0.75 cycles per sample.

To achieve the best performance, we will put some optimization restrictions on the kernel:

- The input length must be a multiple of 4.
- The input and output data are stored on double-word aligned boundaries.
- Vector pointers are not aliased

5.2.1 Apply restrict, pragma and cw_assert

The first step is to use the **restrict** and **cw_assert** keywords and **pragma** to provide some useful information to the compiler for optimization, as shown in Figure 25. Keyword **restrict** tells that the input pointers are not aliased and hence improve the performance by parallel data reading and writing. Keyword **cw_assert** tells the compiler that the loop count is greater than 0 to remove the loop count checking overhead. It aslo indicates that the loop count is multiple of 2. **Pragma align** inform the compiler that the input and output data buffers are 8 byte aligned to enable packed data access.

```
int complex_mult_natural_C_opt1(short* restrict coef,
                                short* restrict input,
                                short* restrict result,
                                int n)
{
#pragma align *coef 8
#pragma align *input 8
#pragma align *result 8

   int i, real, imag;

   cw_assert(n>0 && n%2==0);
   for(i=0;i<2*n;i+=2)
   {
      real = (input[i]*coef[i]) - (input[i+1]*coef[i+1]);
      imag = (input[i]*coef[i+1]) + (input[i+1]*coef[i]);
      result[i] = (real >> 15);
      result[i+1] = (imag >> 15);
   }
return 0;
}
```

Figure 25. C Code Optimization with restrict, pragma, and cw_assert

These keywords and pragma help the compiler understand more about the application and perform the corresponding required optimization techniques. In addition, it is straightforward to add them and this is why we want to use them in the first step. The compiler generated assembly code is shown in Figure 26. The cycle count is reduced down to 3 cycles per sample. We can see that **move.2w** is used instead of **move.w**, which means that 32-bit data buses are used. But it is not good enough. It is desired to fully use 64-bit data buses in SC3850. Also single MAC instructions **imac** and **impy** are used to calculate the output. We can improve the performance significantly by using dual MAC instructions avaiable in SC3850.

```
LOOPSTART3
[
   asrr     #<15,d4                         ;[234,1] 3%=1
   asrr     #<15,d5                         ;[235,1] 3%=1
   move.2w  (r1)+,d0:d1                     ;[232,1] 0%=0
   move.2w  (r0)+,d2:d3                     ;[232,1] 0%=0
]
[
   impy     d2,d0,d4                        ;[232,1] 1%=0
   impy     d2,d1,d5                        ;[233,1] 1%=0
   move.2w  d4:d5,(r2)+                     ;[0,1] 4%=1
]
[
   imac     -d3,d1,d4                       ;[232,1] 2%=0
   imac     d3,d0,d5                        ;[233,1] 2%=0
]
   LOOPEND3
```

Figure 26. Compiler Generated Assembly Code after Using restrict, pragma and cw_assert

5.2.2 Use Intrinsics and Packed Data Access

To use dual MAC instructions, we need to load two data samples in one register and apply SIMD instructions. Many useful assembly instructions, including SIMD instructions, are available to C programmers through intrinsics. The intrinsics are mapped directly to the corresponding assembly instructions by the compiler and thus, they are powerful in C code optimization. However, programmers need to learn which intrinsics are available before using them. In addition, the intrinsics are hardware dependent, which means they can not be used on a processors with a different architecture and instruction set.

In this kernel, **L_mpyre** and **L_mpyim** intrinsics are used to calculate the outputs. They are mapped to SIMD instructions **mpyre** and **mpyim**. Note that these instructions perform fractional multiplication, and thus shifting is not required to produce fractional output data. The following code shows how to use the intrinsics.

```
Word32 L_mpyre(Vector_Type32 src_vect1, Vector_Type32 src_vect2) - Complex fractional multiply -
real portion. Computes: (src1.H * src2.H) - (src1.L * src1.L), using 32-bit saturation mode
Word32 L_mpyim(Vector_Type32 src_vect1, Vector_Type32 src_vect2) - Complex fractional multiply -
imaginary portion. Computes: (src1.L * src2.H) + (src1.H * src1.L), using 32-bit saturation
mode
```

To use **L_mpyre** and **L_mpyim** for complex multiply, we need to take packed complex numbers as input (that is, 16-bit real part in the high end (H) and 16-bit imaginary part in the low end (L) of the same register). In this kernel, 32-bit pointers are defined to load real and imaginary parts of data samples into H and L portions of the same resigters. The loop is unrolled by a factor of 2 to use four dual MAC instructions in one instruction set, as shown in Figure 27. In addition, the **writer_4f** intrinsic stores two output samples (64 bits) with one cycle.

```
int *restrict coef_int = (int * restrict)coef;
int *restrict input_int = (int * restrict)input;
int *restrict result_int1 = (int * restrict)result;

   int i;
   int tempI1,tempQ1,tempI2,tempQ2;

   cw_assert(n>0 && n%2==0);
   for(i=0;i<n;i=i+2)
   {
    tempI1 = L_mpyre(input_int[i], coef_int[i]);
    tempQ1 = L_mpyim(input_int[i], coef_int[i]);
    tempI2 = L_mpyre(input_int[i+1], coef_int[i+1]);
    tempQ2 = L_mpyim(input_int[i+1], coef_int[i+1]);
    writer_4f((short*)&result_int1[i], tempI1, tempQ1, tempI2, tempQ2);
   }
```

Figure 27. C Code Optimization With Intrinsics and Packed Data Access

We can see from Figure 28 that 3 cycles are in the loop to produce 2 output samples, which is 1.5 cycles per sample. In this implementation, the two 64-bit data buses are not fully used. We can improve the performance if we can increase the usage of the data buses.

```
LOOPSTART3
[
    mover.4f  d0:d1:d2:d3,(r2)          ;[0,1]  4%=1
    move.2l   (r1)+,d2:d3               ;[255,1]  1%=0
]
[
    iadd      #<8,d4                    ;[259,1]  0%=0
    move.2l   (r0)+,d0:d1               ;[255,1]  1%=0
    move.l    d4,r2                     ;[259,1]  0%=0
]
[
    mpyre     d2,d0,d0                  ;[255,1]  2%=0VLIW circle
    mpyim     d2,d0,d1                  ;[256,1]  2%=0
    mpyre     d3,d1,d2                  ;[257,1]  2%=0
    mpyim     d3,d1,d3                  ;[258,1]  2%=0
]
    LOOPEND3
```

Figure 28. Compiler Generated Assembly Code After Using Intrinsics and Packed Data Access

5.2.3 Optimization with 4x loop unrolling.

To use the memory access bandwidth fully, the loop is unrolled by a factor of 4. The optimized C code is shown in Figure 29 and the compiler generated assembly code is shown in Figure 30. We can see that every instruction set uses 2 64-bit data buses and thus the optimized code achieve the best possible performance, 0.75 cycles per sample.

```
cw_assert(n>0 && n%4==0);
for(i=0;i<n;i=i+4)
{
    tempI1 = L_mpyre(input_int[i], coef_int[i]);
    tempQ1 = L_mpyim(input_int[i], coef_int[i]);
    tempI2 = L_mpyre(input_int[i+1], coef_int[i+1]);
    tempQ2 = L_mpyim(input_int[i+1], coef_int[i+1]);
    tempI3 = L_mpyre(input_int[i+2], coef_int[i+2]);
    tempQ3 = L_mpyim(input_int[i+2], coef_int[i+2]);
    tempI4 = L_mpyre(input_int[i+3], coef_int[i+3]);
    tempQ4 = L_mpyim(input_int[i+3], coef_int[i+3]);

    writer_4f((short*)&result_int1[i], tempI1, tempQ1, tempI2, tempQ2);
    writer_4f((short*)&result_int1[i+2], tempI3, tempQ3, tempI4, tempQ4);
}
```

Figure 29. C Code with 4x Loop Unrolling Optimization

```
LOOPSTART3
[
    mpyre    d5,d1,d2                      ;[290,1]  3%=1
    mpyim    d5,d1,d3                      ;[291,1]  3%=1
    mpyim    d4,d0,d1                      ;[289,1]  3%=1
    mpyre    d4,d0,d0                      ;[288,1]  3%=1
    move.2l  (r1)+n3,d6:d7                 ;[284,1]  0%=0
    move.2l  (r0)+n3,d4:d5                 ;[284,1]  0%=0
]
[
    mpyre    d6,d4,d0                      ;[284,1]  1%=0
    mpyim    d6,d4,d1                      ;[285,1]  1%=0
    mpyre    d7,d5,d2                      ;[286,1]  1%=0
    mpyim    d7,d5,d3                      ;[287,1]  1%=0
    mover.4f d0:d1:d2:d3,(r3)+n3           ;[294,1]  4%=1
    move.2l  (r4)+n3,d4:d5                 ;[288,1]  1%=0
]
[
    mover.4f d0:d1:d2:d3,(r2)+n3           ;[293,1]  2%=0
    move.2l  (r5)+n3,d0:d1                 ;[288,1]  2%=0
]
    LOOPEND3
```

Figure 30. Compiler Generated Assembly Code after 4x Loop Unrolling

6 Profiling

The CodeWarrior tools have a function profiler that shows how many cycles each function takes to execute. This is a valuable tool and should be used to find critical areas in the application. Programmers can check the cycle count after each step of optimization to find out how much performance has been gained. The profiler works in the CodeWarrior IDE or with the command line simulator. For detailed information please refer to References [5] and [6] on how to profile SC3850.

7 References

The following documents provide detailed information about SC3850 programming and optimization:

1. *StarCore C Compiler user Guide*. (Note: contains intrinsics information)
2. *SC3850 DSP Core Reference Manual.*
3. *C Compiler User Guide (version 23.06) for StarCore CodeWarrior Development Studio.*
4. *StarCore Linker Reference Manual.*
5. *MSC8156 SC3850 DSP Subsystem Reference Manual.*
6. *CodeWarrior Development Studio for StarCore Profiling and Analysis Tools Users Guide*
7. *Tuning C Code for StarCore-Based Digital Signal Processors.*

NOTE

Most of the documents in this list are only available with a signed non-disclosure agreement. Please contact your local Freescale sales office or representative for details and to obtain copies of these documents.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3674
Rev 0
04/2010