Freescale Semiconductor
Application Note

Document Number: AN3357
Rev 2, 7/2008

Tuning C Code for StarCore®-Based Digital Signal Processors

This document describes how to tune existing C code for best performance on Freescale StarCore®-based DSPs (including SC140, SC140e, SC1400, and SC3400 core-based processors). The document presents tips to help the programmer quickly optimize code for the StarCore target platforms. The goal is to assist programmers moving C code from another platform to StarCore or tuning existing StarCore code to obtain the best performance.

Contents

1 StarCore Architecture Overview

This overview of the StarCore architecture outlines key architectural features and then considers functional resources, supported data types, and registers. Table 1 maps the StarCore cores to the Freescale DSPs.

Table 1. Freescale DSPs and StarCore Cores

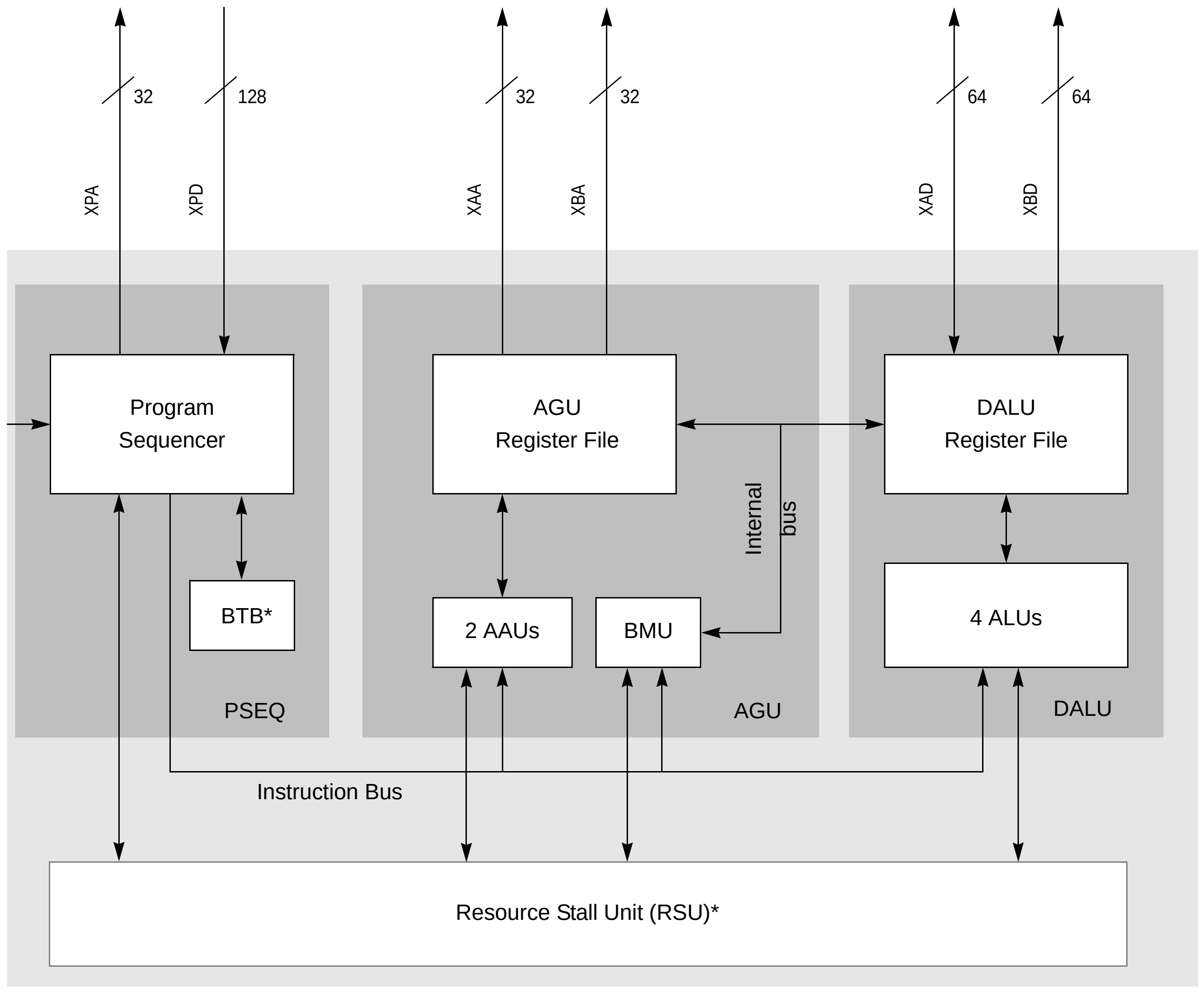
DSP Core	Products
SC140	MSC8101/MSC8103/MSC8122/MSC8126
SC140e	MXC
SC1400	MSC71xx
SC3400	MSC8144/MSC8144E/MSC8144EC
Note: The SC140 and SC1400 cores are architecturally identical. Some SC140 cores are of an older architectural revision (V1), which differs slightly from newer SC140 and SC1400 cores (V2). The SC140e core has some architectural improvements over the SC140 core.	

1.1 Architectural Features

The key features of all StarCore cores are as follows:

- Fixed-point DSP.
- Highly parallel 6-issue VLIW machine.
- Data ALU consisting of:
  - Four independent orthogonal ALUs, each with a MAC unit and a bit field unit.
  - Sixteen 40-bit registers (d[0–15]).
- Address generation unit consisting of:
  - Two identical address arithmetic units (AAUs).
  - One bit mask unit (BMU).
  - 27 registers including eight 32-bit address registers (r[0–7]) and eight registers that are either additional address registers (r[8–15]) or used as base registers for modulo addressing (referred to as b[0–7]).
- Program sequencer (PSEQ): resource stall unit (RSU) (SC3400 only). Implements the interlocked pipeline.
- Branch target buffer (BTB) (SC3400 only). Implements the branch prediction mechanism.
- High data bandwidth. Two 64-bit data buses and a 128-bit program bus. Each bus operates independently.
- 32-bit unified program and data byte-addressable address space.
- Variable length execution set (VLES) software model, which allows the trade-off between efficient code size for control code and high-performance DSP code
- Five-stage pipeline (SC140/SC140e/SC1400) or a 12-stage fully interlocked pipeline with branch prediction and speculative execution (SC3400)
- Four zero cycle-overhead hardware loops (supporting up to four nested hardware loops). [1]

1. On the SC3400 core, there is some training overhead associated with the hardware loops



* Present on SC3400 only

Figure 1. StarCore Cores

1.2 Instruction Support

Several data types are natively supported:

- Signed integer (byte, word, and long)
- Unsigned integer (byte, word, and long)
- Signed fractional

The table below shows the arithmetic operations by instruction type which can be performed by the StarCore cores. The DALU supports both fractional and integer operations. The AGU can also be used for 32-bit integer operations.

Table 2. Operations Supported on Each Functional Unit

Instruction Type	Operations (SC140/SC140e/SC1400)	Operations (SC3400)
DALU	40-bit add/subtract with saturation 32-bit add/subtract without saturation 16-bit fractional multiply with saturation 16-bit multiply without saturation 40-bit logical operations (inc. sine/zero extension) 40-bit shift operations 40-bit comparisons	40-bit add/subtract with saturation 32-bit add/subtract without saturation 16-bit fractional multiply with saturation 16-bit multiply without saturation 40-bit logical operations (inc. sine/zero extension) 40-bit shift operations 40-bit comparisons 8-bit application specific arithmetic 16-bit SIMD operations
AGU	32-bit add/subtract without saturation 32-bit shifting (limited functionality) 32-bit comparisons	32-bit add/subtract without saturation 32-bit shifting (limited functionality) 32-bit comparisons

1.3 Fractional Versus Integer Arithmetic

The programmer must understand the basic differences between fractional and integer operations and the instructions used in both cases. The StarCore cores support fractional and integer operations simultaneously in parallel. The type of arithmetic is specified by the instruction; there are instructions for fractional operations and integer operations. There are also move instructions to load and store the data for both data types. The fractional arithmetic instructions perform a left shift by 1 bit after a multiply operation and also saturate at the relevant point (either 32- or 40-bits depending on the configuration of the core). The code in Figure 2 compares instruction sequences for fractional operations (at the left) and integer operations (at the right).

Fractional Operations

```
move.f (r0)+,d0

move.f (r1)+,d1

mac d0,d1,d2

moves.f d2,(r3)
```

Integer Operations

```
move.w (r0)+,d0

move.w (r1)+,d1

imac d0,d1,d2

move.w d2,(r3)
```

Figure 2. Instruction Sequences for Fractional and Integer Operations

1.4 Arithmetic Operations on the AGU

Arithmetic operations can be performed on the AGU. The address arithmetic units (AAU) are used to add, subtract, and shift pointers but the compiler also uses them for general-purpose integer operations when necessary.

2 Using the Tools

Use of the latest Freescale CodeWarrior™ tools for StarCore is assumed. These tools enable the programmer to build code for StarCore-based products, connect to development boards, and use software simulators.

2.1 Specifying the Target Architecture

To build the best code, specify the correct target architecture in the integrated development environment (IDE). Code built for the SC140 and SC1400 cores runs unmodified on the SC140e core. Code built for the SC140, SC140e, or SC1400 core runs unmodified on the SC3400 core. However, building for the correct target produces faster and smaller code, in comparison to simply running legacy code because the compiler is familiar with the architectural features.

2.2 Enabling Optimizations

Optimizations are disabled by default when no optimization level is specified and either new project stationary is created or code is built on the command line. This code is designed for reference only. It is inefficient and should not be used in production code. The levels of optimization available to the programmer are from zero to three, with three producing the most optimized code. In addition, there is an option to build for size, which can be combined with any optimization level. In practice, two optimization levels are most often used: O3 (optimize fully for speed) and O3Os (optimize for size). In a typical application, critical code is optimized for speed and the bulk of the code may be optimized for size. Global or cross-file optimizations result in full visibility into all the functions, enabling much better optimizations for speed and size. The disadvantage is that since the optimizer can remove function boundaries and eliminate variables, the code becomes difficult to debug. Note that optimizations can be applied at the project, module, and function level and require the use of pragmas or an application configuration file.

Table 3. Available Optimization Levels

Setting	Description
O0	Optimizations disabled. Outputs unoptimized assembly code.
O1	Only target-independent optimizations. Outputs unoptimized assembly code.
O2	Target independent and target-specific optimizations. Outputs non-linear assembly code.
O3	Target independent and target-specific optimizations, with global register allocation. Outputs non-linear assembly code. Recommended for speed-critical parts of application.
Os	Performs space optimizations for the indicated optimization level. Outputs assembly code which is small. Recommended to use in combination with O3 for size-critical parts of application.
Og	Global (cross file) optimization.

2.3 Using the Profiler

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 the critical areas. The function profiler works in the IDE and also with the command line simulator. Table 4 shows a sample function profile output, which has been formatted for clarity.

Table 4. Sample Profiler Output

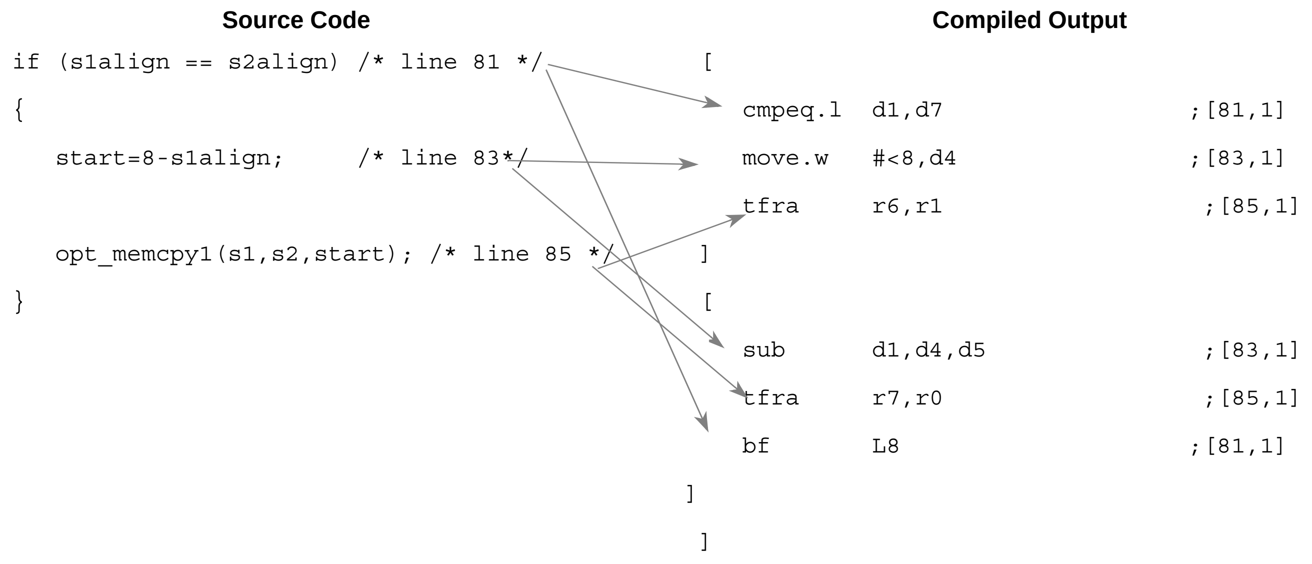
Module	Function	PC	No. of Calls	Stack Size	Percentage	Total Cycles	Min Cycles	Max Cycles	Mean Cycles
fr_long_term_asm	_FrGsmCalculationOfTheLTPParametersmaxCC	0x00005030	2080	16	13.98	2303712	1104	1113	1107
fr_structures	_FrGsmShortTermSynthesisFiltering	0x000059a0	2080	0	11.69	1927034	303	2764	926
fr_structures	_FrGsmShortTermAnalysisFiltering	0x00005a40	2080	0	10.12	1667640	260	2400	801

2.4 Analyzing Compiled Code

The programmer should routinely examine compiler-generated assembly code to get information useful in modifying the C source and making further improvements. You can examine the code in the IDE or use the `--keep` command line option. The compiler-generated assembly files are labeled with the `.sl` file extension to avoid confusion with hand-generated assembly files, which have the `.asm` extension.

2.4.1 Correlating Generated Assembly with C Source

In order to correlate generated assembly with C source, use the line number of the C source code as shown in the comments which follow each assembly instruction. The line number is the first of the two numbers in bracket. In this example, three lines of C source generated two VLES with three instructions in each.



Source Code

```
if (s1align == s2align) /* line 81 */
{
   start=8-s1align;      /* line 83*/

   opt_memcpy1(s1,s2,start); /* line 85 */
}
```

Compiled Output

```
[
   cmpeq.l   d1,d7          ;[81,1]
   move.w    #<8,d4         ;[83,1]
   tfra      r6,r1          ;[85,1]
]
[
   sub       d1,d4,d5       ;[83,1]
   tfra      r7,r0          ;[85,1]
   bf        L8             ;[81,1]
]
]
```

Figure 3. Source Code Versus Compiled Output

2.4.2 Parallelism and VLES

Parallelism is statically encoded by the compiler or the assembly programmer. A grouping of instructions in parallel is called a variable-length execution set (VLES). There is no scheduling in hardware. Note that the square brackets [] denote instructions that operate in parallel.

```
move.4f d4:d5:d6:d7,(r1)+
[ mac d4,d8,d12                ; these four MAC instructions
  mac d5,d9,d13                ; execute in parallel
  mac d6,d10,d14
  mac d7,d11,d15
]
```

Figure 4. Example of Parallel Code

3 Data Types and Their Use

Table 5 shows how the StarCore compiler implements C data types and how the data types are stored in registers. Floating-point types are not natively supported, and fractional types are enabled by using intrinsics, as discussed in Section 3.6, "Fractional Intrinsics." Data must be aligned to its access width in memory; misaligned accesses are not supported.

Table 5. Data Types in Registers and Memory

C Data Type	Size (Bits)	Dn Register Size	Rn Register Size	Minimum Alignment in Memory
char	8	40	32	8
unsigned char	8	40	32	8
short	16	40	32	16
unsigned short	16	40	32	16
int	32	40	32	32
unsigned int	32	40	32	32
long int	32	40	32	32
unsigned long int	32	40	32	32
float, double, long double	32	40	32	32
pointer	32	-	32	32
fractional short	16	40	-	16
fractional long	32	40	-	32

3.1 Short Versus Int

Avoid using short (16-bit) data types when possible. Instead, use int (32-bit), except for data to be multiplied. Also, use integers for array indices. This may seem counter-intuitive, but the shorter datatype is often less efficient. In C, a 16-bit integer (short) has wrap-around behavior (for example, 32766, 32767, -32768, -32767, …) To support this, a short must be sign-extended over the upper 24 bits of a data register before it can be mixed with higher-precision types. In addition, function arguments and return values must be sign extended due to limited visibility across function boundaries. There is a slight overhead for sign extension in such operations. Figure 5 illustrates the sign-extension process. Here, a **sxt.w** instruction is needed so that the results of the operation performed on type short can be mixed with other types across the function boundary.

Short Addition

```
short func (short x) {
  return (x+1);
}
```

Generated Assembly Code

```
[ inc d0          ; increment
  rtsd            ; function return
]
 sxt.w d0,d0      ; sign extension
```

Figure 5. C Source for an Addition of Type Short, Generated Assembly Code

3.2 32-bit Multiplication

Avoid using 32-bit multiplication where possible because it is not natively supported. The multiplier on each ALU is 16-bit. Use of a 32-bit multiplication generates the following (or similar) 4-VLES sequence, which takes four cycles to execute.

32-Bit Multiplication

```
int a,b,c;      // 32-bit types
long long d;    // 64-bit type
d=a*b;
```

Generated Assembly Code

```
[  impysu   d1,d3,d2
   impyuu   d1,d3,d4   ]
   imacus   d1,d3,d2
   aslw     d2,d2
   iadd     d4,d2
```

Figure 6. C Source for 32-Bit Multiplication; Generated Assembly Code

3.3 Floating-Point Operations

StarCore cores are fixed-point devices. Any floating-point operation is not natively supported and consequently is slow. However, the compiler supports the float and double data types, and functional support is provided for floating-point operations through library routines.

3.4 Division Operations

Avoid division when possible. Division is implemented through a run-time library call and consequently consumes a significant number of cycles. If possible, use a right shift, which is supported by instructions.

3.5 64-Bit Data

The cores do not natively support 64-bit data types (long long and double double), so they should be avoided where possible. The compiler supports them through library routines, if support is enabled with a switch. When used, the 64-bit data types typically reside in memory or split across a pair of registers.

3.6 Fractional Intrinsics

Fractional data types are not natively supported in C. The StarCore instruction sets include both integer and fractional instructions. When fractional data is used, it is important that the compiler knows so that fractional instructions are used. If the compiler uses integer instructions, unintended errors can occur and saturation does not occur (integer instructions do not support saturation). For fractional data, fractional intrinsics must be used to communicate to the compiler that the data is fractional. The compiler also supports two special types, fractional short and fractional long/int, which can only be used with intrinsics.

No Intrinsics

```
short SimpleFir0( short *x,
short *y) {

int i;
short ret;
ret = 0;
for(i=0;i<16;i++)
  ret += x[i]*y[i];
return(ret);
}
```

Corresponding Assembly Code

```
[  clr       d0
   doensh3   #<16
]  nop
   LOOPSTART3
[  sxt.w     d0,d0
   move.w    (r0)+,d4
   move.w    (r1)+,d3  ]
   imac      d3,d4,d0
   LOOPEND3
 [ sxt.w     d0,d0
   rts ]
```

Figure 7. Example Without Intrinsics and Corresponding Assembly Code

Note in Figure 8 that when intrinsics are used, the compiler uses different instructions. The data is loaded using fractional loads (**move.f**) and uses fractional arithmetic instructions (**mac**). The **mac** instruction performs a left shift after the multiply and saturates after the addition, if necessary.

Intrinsics Used

```
#include <prototype.h>


 short SimpleFir1( short *x,
 short *y) {

 int i;
 int ret;
 ret = 0;
 for(i=0;i<16;i++)
   ret=L_mac(ret,x[i],y[i]);
 return(extract_h(ret));
 }
```

Corresponding Assembly Code

```
[  clr       d0
   doensh3   #15
   move.f    (r0)+,d4 ]
   move.f    (r1)+,d5
   LOOPSTART3
[  mac       d4,d5,d0
   move.f    (r1)+,d5
   move.f    (r0)+,d4
]
  LOOPEND3
[  mac       d4,d5,d0
   rtsd  ]
   asrw      d0,d0
```

Figure 8. Example Using intrinsics and Corresponding Assembly Code

3.7 Memory Contention

When data is placed in memory, be aware of how the data is accessed. The StarCore cores have two data buses, each 64 bits wide. Each bus can issue an access each clock cycle. Therefore, a lot of data can be brought into the core and saved from the core, resulting in a high amount of parallelism. However, there is a penalty if the two data buses issue transactions that conflict in memory. Data should be separated appropriately to avoid this contention. The scenarios that cause contention are device-dependent because memory bank configuration and interleaving differs from device to device. For details, see Section 7, “References.”

3.8 Cache Accesses

In the caches, place data that is used together next to each other in memory so that the prefetching the caches is more likely to obtain the data before it is accessed. For more information, see Section 7, “References.”

3.9 Data Alignment to Enable Multiple Register Moves

By default, the compiler aligns variables in memory to their access width. For example, an array of short (16-bit) data is aligned to 16 bits. The StarCore cores support loading of multiple data values across the 64-bit data buses, which can be necessary to keep the ALUs busy. However, to leverage multiple data moves, the data must be aligned to a higher alignment. For example, to load two 16-bit values at once, the data must be aligned to 32 bits.

Many compiler optimizations require multiple register moves because there is so much data to move to keep all the functional units busy. For the compiler to be able to use multiple register moves, the following must be true:

- The data must be aligned to the combined access width.
- The compiler must be informed of this alignment (for example, across a function boundary).

These requirements are met using the alignment pragma.

- Step 1: Align the data

  ```
  Word16 gInAry [NO_INPUTS];
  #pragma align gInAry 4
  ```

- Step 2: Indicate to the compiler that any pointers pointing to that data are aligned. This is especially important when pointers are passed into functions. In this case, place the pragma inside the function itself. In Figure 9, the pragma indicates to the compiler that inputPtr points to an array that is aligned to 4 bytes.

  ```
  void DcOffsetRemovalOpt (Word16 * inputPtr) {
  #pragma align * inputPtr 4
  ... }
  ```

Function Using Aligned Data

```
Word16 gOutAry [NO_INPUTS];
#pragma align gOutAry 4
Word16 gInAryRef [ ] = {
    WORD16(0.7948),
    WORD16(0.9568),
    WORD16(0.5226),
    ... }
void DcOffsetRemovalOpt (Word16 * inputPtr) {
#pragma align * inputPtr 8
for (i=0;i<NO_INPUTS;i++) {
    inputPtr[i]=(L_sub(inputPtr[i],temp));
    }
...
}
```

Resulting Code

```
LOOPSTART3
[
   sub      d2,d4,d8
   sub      d2,d5,d9
   sub      d2,d6,d10
   sub      d2,d7,d11
   move.4w  d8:d9:d10:d11,(r0)+
   move.4w  (r1)+,d4:d5:d6:d7
]
   LOOPEND3
```

Figure 9. Function Using Aligned Data and Resulting Code

In addition, the compiler can be instructed that all that all pointer function parameters are 8 bytes aligned, removing the need to place pragmas inside every function. This is accomplished by passing the following to the compiler shell (scc) -Xcfe "-fl auto_align8". When using this option, the programmer must ensure that the pointers do point to memory locations which are aligned to 8 bytes and must still communicate alignment for other pointers which are not function parameters.

3.10 Indexed Arrays Instead of Pointers

In general, pointer notation is more complex for the compiler to resolve than array notation. When possible, use array notation, especially in cases that use complex offsets. In Figure 10, the arrays are referenced by an index 'j':

Pointer	Array
`accA = L_mac(accA, iRefPtr+j, iInPtr+j);`	`accA = L_mac(accA, iRefPtr[j], iInPtr[j]);`

Figure 10. Example of Pointer Use and Array Use

An additional benefit is that for manual unrolling or partial summation, the compiler can more easily figure out adjacent array values as candidates for multiple register moves:

```
accA = L_mac(accA, iRefPtr[j], iInPtr[j]);
accB = L_mac(accB, iRefPtr[j+1], iInPtr[j+1]);
```

3.11 Pointer Aliasing

When pointers are used at the same piece of code, ensure that they cannot point to the same memory location (alias). When the compiler knows 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: using the restrict keyword or by informing the compiler that no pointers alias anywhere in the program.

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.

```
void foo (short * a,short * b,int N) {          doen3     d4
int i;                                          FALIGN
  for (i=0;i<N;i++) {                           LOOPSTART3
     b[i]=shr(a[i],2);                          move.w    (r0)+,d4
  }                                             asrr      #<2,d4
  return;                                       move.w    d4,(r1)+
}                                               LOOPEND3
```

Figure 11. Example Using Two arrays and Corresponding Assembly Code

Arrays with restrict Keyword

```
void foo (short * restrict a,short * restrict
b,int N)

  int i;


  for (i=0;i<N;i++) {

     b[i]=shr(a[i],2);

  }


  return;

}
```

Corresponding Assembly Code

```
move.w    (r0)+,d4

 asrr      #<2,d4

 doensh3   d2

 FALIGN

   LOOPSTART3

[    move.w    d4,(r1)+      ; parallel

     move.w    (r0)+,d4      ; accesses

]

   asrr      #<2,d4

   LOOPEND3

   move.w    d4,(r1)
```

Figure 12. Two Arrays with restrict Keyword and Corresponding Assembly Code

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 ensure that no pointers alias when using this option.

4 Functions

This section presents tips for dealing with functions, as follows:

- *Inline small functions*. The compiler normally inlines small functions, but the programmer can force inlining of functions. For small functions, the save, restore, and parameter passing overhead can be significant relative to the number of cycles of the function itself. Therefore, inlining is beneficial. Also, inlining functions decreases the chance of an instruction cache miss because the function is sequential to the former caller function and is likely to be prefetched. Note that inlining functions increases the size of the code. Pragma inline forces every call of the function to be inlined.

  ```
  int foo () {
      #pragma inline
  ... }
  ```

  Similarly, there is a pragma noinline to disable inlining of a particular function. In addition, the compiler allows the programmer to specify inlining of a function on a case-by-case basis. This allows trade-off between speed benefit of inlining and code size increase. The following lines force inlining of the next call of the function foo. They are placed just before the call.

  ```
  #pragma inline_call foo
  foo ();
  ```

- *Calling conventions*. Change the calling convention for functions with many arguments. The calling conventions typically pass the first two arguments (scalar or pointer) in registers and the remaining arguments on the stack. For functions with many arguments that are called frequently, the calling conventions may be inefficient. Custom calling conventions that override the defaults can be specified for any function through an application configuration file and pragmas.
- *Optimization level*. Change the optimization level on a function basis when appropriate. Typically, some functions are optimized for speed (-O3) while others are optimized for size (-O3-Os). This can be applied at the function level by pragmas or an application configuration file. The opt_level pragma is used, for example:

  ```
  foo ( ) {
  #pragma opt_level "O3"
  ... }
  ```

  The pragma can also be applied at the file (module) level by placing it at the top of a file, outside any functions.
- *Take advantage of instruction caches*. Functions should be aligned so that they fall on cache boundaries. This can be accomplished using the align pragma. Note that the align pragma is placed after the function.

  ```
  int foo () { ... }
      #pragma align foo 256
  ```

  You can set a minimum alignment for all functions with the following option to the compiler: "-Xicode --min_func_align=256" For details, see Section 7, "References."

- *Sequentially-called functions*. Functions called sequentially should be placed in sequential areas of memory so that cache prefetching is likely to bring the code into the cache before it is needed. This is accomplished through the use of linker command files.

5 Loops

This section presents tips for dealing with loops, as follows:

- *Hardware loop mapping*. The StarCore architecture supports four hardware loops. Hardware loops are faster than normal software loops (decrement counter and branch) because they have less overhead. Hardware loops use loop registers that start with a count equal to the number of iterations of the loop, decrease by 1 each iteration (step size of –1), and finish when the loop counter is zero. The programmer should be familiar with hardware loop structures in the output assembly. Note the LOOPSTART and LOOPEND markings, which are assembler directives marking the start and end of the loop body, respectively.

```
   doensh3   #<5
   move.w    #<1,d0
   LOOPSTART3
[  iadd      d2,d1
   iadd      d0,d4
   add       #<2,d0
   add       #<2,d2    ]
   LOOPEND3
```

Figure 13. Example Hardware Loop

  In C, the programmer can write a loop with either an increasing or decreasing loop counter, any step size, and any final bound. The compiler transforms C loops into hardware loops. However, certain structures may prevent the compiler from generating a hardware loop.

- *Function calls in loops*. A function call in the loop body prevents the compiler from generating a hardware loop, significantly degrading performance.[1] Therefore, function calls in loops should be avoided.

```
for (i=0;i<10;i++)          FALIGN
   foo();                   L3
                                jsr      _foo
                                deceq    d3      ; loop counter
                                bf       <L3
```

Figure 14. Loop Containing a Function Call Resulting in Generation of a Software Loop

1. However, in the SC3400, branch prediction is used for change-of-flow instructions and hardware loops. This is an aspect of performance that can be traded off.

- *Loop conditions*. Keep loop conditions simple. A dynamic loop end count or a complicated loop end count that cannot be resolved prevents the compiler from generating a hardware loop.
- *Loop step*. Ensure that the loop step is a power of 2 or is equal to 3, 5, or 7. Otherwise, the optimizer cannot map the loop into a hardware loop.
- *Loop index and bounds*. Ensure that the loop index and bounds are of type short. In certain circumstances, if the loop index or bounds is not of type short or unsigned short, the compiler may not be able to map the loop into a hardware loop.
- *Loop count information*. The loop count pragma should be used to specify more information about the iterations of a loop when that information is available. The compiler can use this information to better optimize the code. This is an important pragma and can enable the compiler to do other optimizations. The following can be specified:
  - Minimum number of iterations. When specified as non-zero, the compiler can remove the code that checks for zero loop count. This improves performance.
  - Maximum number of iterations.
  - Modulo and remainder. Used for unrolling information. It is possible to set a modulo and an optional remainder. If the loop always executes a multiple of 2 or 4 times, the compiler can use this information to unroll correctly. In the following example, the loop is specified to execute a minimum of four times, a maximum of 40 times, and always with multiple of 2. No remainder is specified. Note that the pragma is placed immediately inside the loop body.

    ```
    for(j=0; j<refSize; j+=2) {
      #pragma loop_count (4,40,2)
    ... }
    ```
- *Loop-carried dependencies*. Avoid loop-carried dependencies, which occur when values from the current iteration of the loop cannot be completed without knowing values from a previous iteration. The following example illustrates a loop-carried dependency.

  ```
  for (i=0; i<10; i++)
      c[i]=c[i-1]+a[i];
  ```
- *Combine loops*. If two loops with similar characteristics execute sequentially, they are good candidates to be combined into one loop to provide more available parallelism to the compiler.
- *Inner loop*. Do as much work as possible in the inner loop, which increases the available parallelism for the compiler.

6 SC3400-Specific Recommendations

Because the SC3400 architecture significantly differs from previous StarCore architectures, additional techniques exist to assist in the optimization process, as follows:

- *Small hardware loops*. Force unrolling of small hardware loops. Hardware loops use branch prediction. There is some training associated with shorter loops. In general, the shorter the loop, the higher the penalty. Therefore, there is an incentive to unroll loops by whatever means possible. The programmer should ensure that the code is written so that unrolling is possible. The compiler provides the option to make shorter loops longer to reduce the training overhead. This option is disabled by default but can be enabled with the following option:

  -Xllt “-unroll 0|1|2|3)” (0 for disabled, 1 for speed versus size compromise, 2 for speed, 3 for blind application of the optimization)
- *Loop index optimization*. Reuse of indices can cause thrashing of the branch target buffer (BTB) entries. This includes the scenario when two non-nested loops are contained within one outer loop. The compiler has an option to enable optimization of hardware loop indices so minimize branch target buffer (BTB) stalls:

  -Xllt “-loop_renumbering<0|1>” (0 for disabled, 1 for enabled). Enabled by default.
- *Change-of-flow destination*. Ensure that the change-of-flow destination is inside the same 15-bit (32 Kbyte) aligned block as its source. If this is not the case, that instruction cannot use the branch prediction mechanism. This can be avoided by the linking and placement of the code.
- *Reads and writes to overlapping addresses*. Due to the pipeline in the SC3400, a read that occurs after a write in code can actually execute before that write. If the address of the read and the write overlap, there is a stall in the write queue. The compiler adds an option to reduce reads and writes to overlapping addresses and the write queue stalls associated with them:

  -xllt -3x00_raw1
- *SC3000 intrinsics*. Use SC3000 intrinsics for application-specific and SIMD instructions. Various intrinsics exist to support the application-specific and SIMD instructions. The compiler cannot take advantage of these instructions, so using intrinsics is necessary.

7 References

The following documents are available either on the Freescale web site listed on the back cover of this manual or through your local sales distributor.

1. *StarCore C Compiler Reference Manual*.
2. *SC3400 Core Platform Cache Optimization in the MSC8144* (AN3356)
3. Reference Manuals for specific devices (MSC8144, MSC8126, MSC8122, and so on)
4. SC140 and SC1400 DSP core reference manuals.
5. *SC3400 DSP Core Reference Manual.*
6. *SC3400 Programmer's Guide.*
7. *StarCore Linker Reference Manual*

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: AN3357
Rev 2
7/2007