. .eescale Semiconductor
Application Note

AN1838
Rev. 1, 11/2004

Speed and Code Size Trade-Offs on StarCore™-Based DSPs

By Zvika Rozenshein, Dror Halahmi, Arnon Mordoh, and Yuval Ronen

This document discusses the architectural characteristics that prevent the full, simultaneous realization of both execution-speed and code-size goals. Examples are provided to show how size is traded for speed on signal processing kernels. Other examples show how code size can be minimized when speed requirements are not an issue.

When implementing communication protocols specified in C programs, especially on an architecture such as that of the StarCore SC140/SC1400 cores, developers can efficiently use C compilers for product development. Compilers add yet another dimension to the necessary trade-off decisions. This dimension is also covered in the report.

CONTENTS

1 Basics of the StarCore SC140/SC1400 Architecture

On the StarCore SC140 DSP architecture, there is a trade-off between optimizing code for maximum execution speed and optimizing code for maximum program code density. In previous-generation DSP architectures, optimizing code for maximal speed would often also yield favorable code size. On the SC140 core, however, these two optimization goals may result in two very different code implementations.

For any given application, one goal can rarely be pursued at the expense of the other. Instead, we must strike a balance that meets required speed goals and acceptable code size. We present a test case (GSM EFR) to demonstrate achievable speed and size points achieved by combining hand-coded assembly and compiled C code. The methods and results presented can be of use in making trade-off decisions and in selecting portions of an application to be coded in assembly (versus C) as well as portions to be optimized for speed (rather than for size). Based on this test case, code-size estimates for other applications can be made. The accuracy of such estimates depends on the degree of similarity between the application at hand and the test cases presented here. This section describes the architectural properties of the SC140 core that have an impact on the speed-versus-size trade-off.

The SC140 core has four data arithmetic units and two address generation units that can operate in parallel, providing instruction-level parallelism opportunities on embedded, power-efficient signal processors. The high degree of parallel computation comes at a cost, however: the larger the number of operations to be performed in parallel, the larger the number of machine words required to specify the operations for execution. At one extreme, with only one instruction to be computed in a given cycle, a single operation can be encoded efficiently in a single 16-bit word. At the other extreme, when all computational units are employed, up to eight 16-bit words may be required to encode these operations. **Example 1** illustrates a simple case where the encoding of two instructions specified for parallel execution consumes exactly the same number of words as the two instructions would if they executed sequentially.

Example 1. Instructions 1 and 2 Take 4 Bytes, Regardless of Parallelism

```
P:00000000    51 add d0,d1,d2                ; Instr. 1
              6D
P:00000002    18 move (r0)+,d0               ; Instr. 2
              50
P:00000004    51 add d0,d1,d2 & move (r0)+,d0  ; Instr 1,2 in parallel
              2D
              18
              50
```

However, the size of the instruction encoding may depend on the context in which the instruction is executed, as discussed in the following subsections.

1.1 Using Data Registers D8–D15 and Address Registers R8–R15

When one or more of data registers (D8–D15) and address registers (R8–R15) are used in an execution set (that is, the set of instructions specified to be executed in parallel), then the addition of one or two 16-bit Prefix words is necessary to encode the execution set.

Example 2. Instruction That Uses r8 Instead of r0 Is Larger by 4 Bytes

```
P:00000016    18 move (r0)+,d0      ; Instruction uses r0
              50
P:00000018    80 move (r8)+,d0      ; Instruction uses r8
              34
              00
              A0
              18
              50
```

Using these 16 registers greatly boosts performance, since the registers reduce the bottleneck created if all execution units process data using only 8 data registers and 8 address registers. The prefix words required by the use of these registers make it obvious that optimizing code to improve speed (by using the registers) comes at the expense of code size.

1.2 Specific Combinations of Instructions

Placing two instructions in a single execution set for parallel execution often results in an encoding that is the same size as the encoding for the instructions when they are placed sequentially. This property does not hold true for all combinations of instructions. For other combinations, prefix words are needed to place these instructions in the same execution set, as shown in **Example 3**.

Example 3. Instructions 1 and 2 Take 6 Bytes When Sequential and 8 Bytes When Parallelized

```
P:00000000 08  move var,d0              ; Instr. 1
           10
           00
           80
P:00000004 18  adda r0,r1               ; Instr. 2
           E9
P:00000006 C0  move var,d0 & adda r0,r1   ; Instr 1,2 in parallel
           96
           18
           E9
           08
           10
           00
           80
```

Parallelizing such instruction combinations naturally increases application performance, but the prefix words required to encode this parallelism expand the code size .

1.3 Predication

Predication enables the conditional execution of instructions without the use of branch instructions, which are costly because of their effects on the execution pipeline. Predication is used in SC140 code to specify conditions within the execution set. The entire execution set can execute conditionally based on a previously computed condition. Part of the execution set can execute conditionally, while the rest of the execution set executes regardless of the condition.

Example 4. Making an Instruction Conditional Increases Its Size by 2 Bytes

```
P:00000000 51 add d0,d1,d2
           6D
P:00000002 C2 ift add d0,d1,d2
           92
           51
           6D
```

Using predication in an execution set requires adding Prefix words to the execution set. In some cases, predicated code would be no larger than code that uses branches, but in some cases predication results in larger code size.

Note: The maximum number of prefix words in an execution does not exceed two (for a total of 32 bits added). This rule applies regardless of the number of preceding cases that may apply to the execution set.

2 Optimizing for Maximum Speed

Programs coded to use all SC140 units (four data arithmetic units and two address generation units) can achieve high execution speeds. The SC140 core can speed up execution by a factor of four or more. Speed-ups of over four times are possible because of the rich instruction set and a high degree of orthogonality in the programming model. **Table 1** presents speed-ups for various DSP kernels. The kernels are optimized for execution speed, minimizing the number of cycles required to perform the computation.

Table 1. DSP Kernel Speed-ups

Kernel	Parameters	Ratio of DSP56600 to StarCore SC140		StarCore SC140		DSP56600	
		Speed	Size	Cycles	Bytes	Cycles	Bytes
FIR	N=12, T=12	1:5.33	1:5.05	46	106	245	21
Complex FIR	N=12, T=12	1:3.38	1:1.39	204	46	689	33
Lattice FIR	N=12, T=12	1:5.92	1:3.09	92	102	545	33
Lattice IIR	N=12, T=10	1:6.92	1:5.71	84	240	581	42
Biquad IIR	N=12, B=3	1:3.35	1:3.39	91	122	305	36
FFT	256 points	1:6.76	1:2.83	1614	348	10907	123

The SC140 core can process these kernels approximately three to seven times faster than the DSP56600, but such high speeds increase the code size by 40– 600 percent. Notice that the degree of speed-up is a relatively weak predictor of code-size increase, as is demonstrated by the complex FIR and the Biquad IIR results. In practical applications, such kernels often consume a large percentage of execution time, but do not dominate the size of the application. Thus, typical speed-ups for full applications are lower than those achieved for the kernels, and typical code size increases are much lower than those in **Table 1**.

3 Optimizing for Minimal Code Size

When code is optimized to minimize its size, a combination of SC140 architectural properties result in code that is significantly smaller than is achievable on comparable DSPs. Among the properties that contribute to code-size efficiency are the basic 16-bit instruction word, the orthogonality of the instruction set, the wealth of addressing modes, and the ability to perform arithmetic operations in the address generation units. **Table 2** compares the code size obtained by compiling a set of code-size benchmarks for various DSPs. The code in these benchmarks is characterized by the following:

- *Data types*. All arithmetic is performed on integers. Fixed-point (saturating) and floating-point types are not used.
- *Control flow*. Most of the code enforces sequential computation. Chains of if-then-else statements are frequently used, while loops are used infrequently. Calls to subroutines are embedded in the code.
- *Memory accesses*. Most accesses are random. Relatively few of the accesses (compared to DSP kernels) are part of progressions through arrays. Accesses to memory are with a variety of data widths (8-, 16-, and 32-bit accesses) in addition to numerous bit-field operations.
- *Practicality*. Most of the benchmarks are real, functional applications.

These benchmarks are representative of sections in communication protocols that are not performance sensitive. These sections are sometimes referred to as control code.

Table 2. Code-Size Benchmarks

Benchmark	StarCore SC140	320C62XX	Ratio of SC140 to 320C62xx	DSP16000	Ratio of SC140 to DSP 16000	320C54x	Ratio of SC140 to 320C54x
	Bytes	Bytes		Bytes		Bytes	
auto	5926	8572	1:1.45	8438	1:1.42	6904	1:1.17
blit	452	1368	1:3.03	592	1:1.31	1032	1:2.28
compress	3154	5028	1:1.59	3670	1:1.16	3480	1:1.10
des	2188	4036	1:1.84	2342	1:1.07	2970	1:1.36
dhry21	1378	2000	1:1.45	1818	1:1.32	1492	1:1.08
engine	772	1264	1:1.64	1606	1:2.08	606	1:0.78
eval2	1914	3340	1:1.75	4548	1:2.38	1784	1:0.93
fir_int	356	564	1:1.58	748	1:2.10	386	1:1.08
g3fax	676	1208	1:1.79	794	1:1.17	790	1:1.17
jpeg	1890	2672	1:1.41	2264	1:1.20	1724	1:0.91
pocsag	1492	3040	1:2.04	1778	1:1.19	1766	1:1.18
summin	440	668	1:1.52	674	1:1.53	410	1:0.93
ucbqsort	958	1712	1:1.79	1656	1:1.73	1126	1:1.18
v42bis	2216	4604	1:2.08	3056	1:1.38	2488	1:1.12
Total	23812	40076	1:1.68	33984	1:1.43	26958	1:1.13

All benchmarks were compiled to minimize code size. As **Table 2** shows, the SC140 architecture implements control code at a code size that is significantly smaller than for other DSPs. DSP applications are rarely composed only of code that exhibits the characteristics in our benchmarks. Rather, they are a combination of such code with signal processing code that heavily relies on fixed-point arithmetic, loop structures, systematic progressions through memory arrays, and mostly 16- and 32-bit accesses to memory. Consequently, for a full DSP application, the ratios between the SC140 code size and the code size for implementing the same application on a DSP with a single arithmetic unit is lower than ratios presented here.

4 Optimizing Code for a Combined Goal

Applications are not often optimized for a single goal. Rather, they are optimized to meet specific performance (speed) goals while also minimizing the code size required to reach those goals. This section discusses approaches to optimizing applications for a combination of both speed and size goals.

4.1 Classifying the Application's Code Sections

In many DSP applications, 20 percent of the application code contributes as much as 80 percent to the overall execution time of the application, and the remaining 80 percent of the code consumes only 20 percent of the total execution time. This observation leads to the following approach to porting an application from its standard implementation in C to the StarCore SC140 architecture:

1. Profile the application. Identify the time-consuming performance-critical code sections.
2. Optimize the performance-critical sections for maximum speed. Refine the classification in Step 1, if needed.
3. Optimize the remaining, non-critical sections for minimal size.

In systems concerned only with performance, when code size does not present a real constraint, both sections of the code can be optimized for speed. The classification of code as performance-critical should therefore be based on both the application algorithm and the system requirements.

4.2 Optimizing Performance-Critical Code Sections

Focusing the development effort on the performance-critical sections of an application can yield very high performance. You can optimize the code in C or directly in SC140 assembly. Optimizing the code in C usually requires restructuring the C code to expose higher levels of parallelism than are expressed in the original C code. Based on the amount of effort this requires and on the level of optimization the compiler is able to extract from the C code, the developer may choose to optimize the code in assembly. The SC140 core has a simple execution pipeline and an orthogonal programming model so that programming in SC140 assembly code is a reasonably straightforward and practical undertaking.

When C code is optimized for maximum performance, the compiler attempts to use the full sets of data and address registers. It groups instructions into execution sets and uses predication to reduce costly branches. The compiler also applies techniques, such as software pipelining and loop unrolling, to reduce the cycle count. As the optimization of the performance-critical sections progresses, the classification of code sections can be fine-tuned.

4.3 Optimizing Non-Critical Sections

The contribution of the non-critical sections of the code to application speed is relatively small. However, since these sections comprise most of the code, it is beneficial to keep them as small as possible. When C code is optimized for minimal code size, the compiler uses only eight data registers and eight address registers and refrains from grouping instructions into execution sets if such grouping incurs prefix words. The compiler attempts to use predication only when the resulting code has no larger branches. All this is done in addition to applying architecture-independent optimization techniques that are widely used by compilers for embedded microcontrollers.

5 Test Case: GSM Enhanced Full Rate Voice Codec

The examples discussed so far deal with DSP kernels and code that is not performance-sensitive in isolation. This section presents speed and code-size trade-offs for the GSM enhanced full rate (EFR) speech codec. This test case applies the approaches presented thus far to show possible performance and size points that may be achieved when you are implementing such an application on the SC140 core.

5.1 Classifying EFR Code Sections

Based on a profiling of the application, a specific section of the EFR code was selected. For the purpose of the following presentation, the following terms are used:

- G1: the set of subroutines that are performance critical, contributing about 80 percent of the theoretical computational requirements
- G2: the remaining code of EFR

5.2 Optimization Approaches

A number of different optimization approaches are considered. These approaches differ in the methods used for developing the optimized code (programming in assembly versus programming in C) and in the goals set for optimizing each section. Some of the data points measured and presented in this section are not optimal in any sense. Rather, they provide a more complete picture of the capabilities at the time they were measured, or they represent theoretical minimums that indicate lower bounds on possible implementations.

5.3 EFR Data Points

Table 3 describes the various data points that have been measured for the GSM EFR.

Table 3. GSM EFT Data Points

Data Point	G1	G2	Description
All-C-SPD	C, optimized for speed	C, optimized for speed	StarCore SC140 compiler, version 0.95.
All-C-SPC	C, optimized for size	C, optimized for size	StarCore SC140 compiler, version 0.95.
All-C-SPC+	C, optimized for size	C, optimized for size	Code-size figures represent projected results based on additional optimizations to be implemented by the compiler. Performance figures are not available.
G2-SPD	Assembly, optimized for speed	C, optimized for speed	StarCore SC140 compiler, version 0.95.
G2-SPC	Assembly, optimized for speed	C, optimized for size	StarCore SC140 compiler, version 0.95.
G2-SPC+	Assembly, optimized for speed	C, optimized for size	Compiled code size figures are projected based on additional optimizations to be implemented by the compiler. Performance figures are not available.
G2-proj	Assembly, optimized for speed	Assembly, optimized for size	The figures presented are based on the estimation technique presented in Appendix A, “Projecting SC140 Code Size from DSP56600 Code,” which allows projecting the performance and size of StarCore SC140 code based on the dynamic measurement of optimized DSP56600 code for the same application.
All-proj	Assembly, optimized for size	Assembly, optimized for size	The figures presented are based on the estimation technique presented in Appendix A, “Projecting SC140 Code Size from DSP56600 Code,”which allows projecting the performance and size of StarCore SC140 code based on the dynamic measurement of optimized DSP56600 code for the same application. These figures represent the theoretical minimal size of EFR for the StarCore SC140. Note that this theoretical minimum goes hand in hand with performance that is equivalent to that of the DSP56600.
DSP56600	EFR fully coded in assembly, and optimized for both speed and size, for DSP56600. Provided for reference.		

Table 4 presents the measurements obtained for these data points.

Table 4. Performance Measurements

Data Point	Code Size (Bytes)	Size Ratio of DSP56600 to StarCore SC140	Performance (MCPS)	Performance Ratio of DSP56000 to StarCore SC140
All-C-SPD	44492	1:1.73	15.5	1:1.15
All-C-SPC	35884	1:1.39	27.5	1:0.65
All-C-SPC+	30204	1:1.17	—	—
G2-SPD	45776	1:1.78	7.17	1:2.49
G2-SPC	41676	1:1.62	8.73	1:2.04
G2-SPC+	36644	1:1.42	—	—
G2-proj	30784	1:1.19	6.84	1:2.61
All-proj	22937	1:0.89	17.84	1:1.00
DSP56600	25788	—	17.84	—

The results shown in **Table 4** were measured on a single-channel implementation of EFR. A multi-channel implementation is expected to require slightly larger program code and require slightly more MCPS. The results of the EFR test case are graphically presented in **Figure 1**.

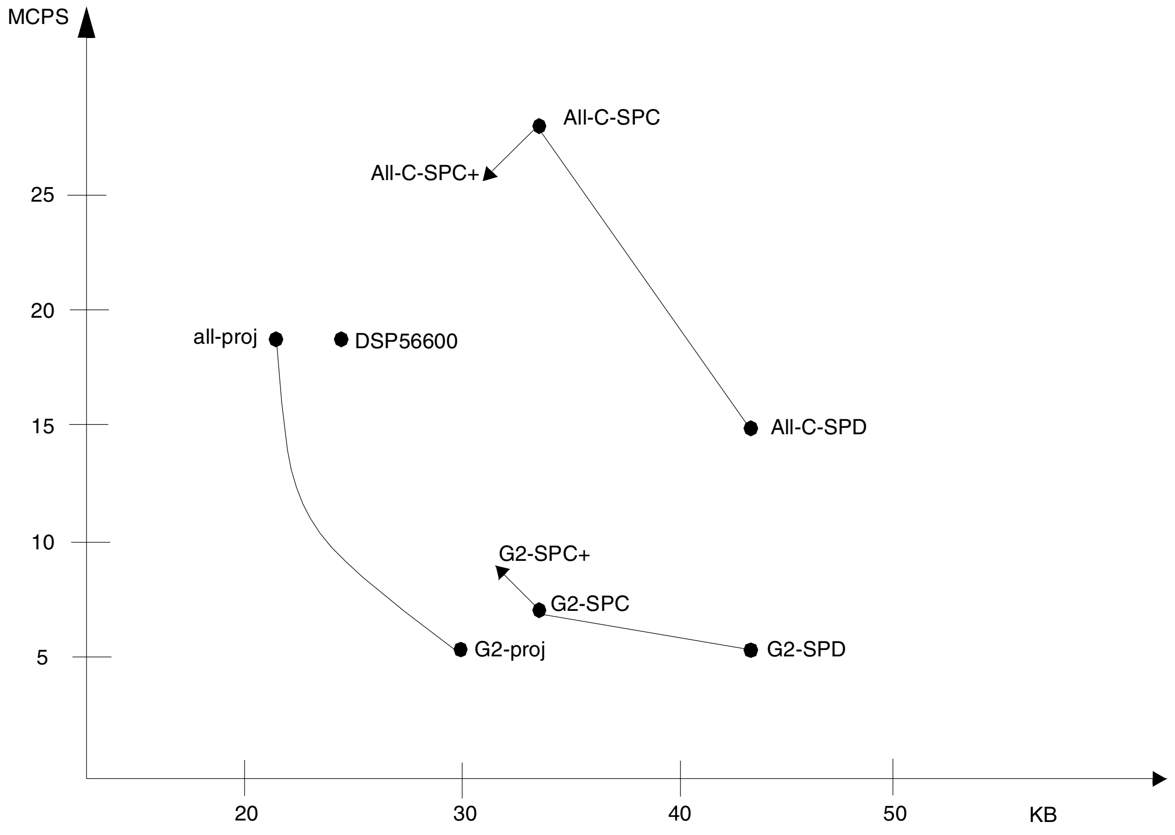

Figure 1. EFR Test Results

5.4 Conclusions

The data points All-proj and G2-proj represent the theoretical minimums for the StarCore SC140. Additional optimizations and improvements planned for the StarCore SC140 compiler are geared toward attaining compiled sizes that are as close as possible to these theoretical minimums. The various data points on the chart delineate the fact that a range of performance and size points are possible. To be able to estimate the required code size for an implementation, you should first set specific performance goals. Once the performance goals are set, it is possible to define splits between compiled code and assembled code, and between code optimized for speed and code optimized for size.

NOTES:

NOTES:

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

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

AN1838
Rev. 1
11/2004