Combining C and assembly in DSP applications

Eran Balaish, CEVA - December 10, 2007

As DSP processors become more powerful and compiler optimization techniques improve, the once common trend of writing DSP applications solely in assembly has withered away. Today, almost every DSP application is comprised of a combination of both C code and assembly code. In critical functions, where performance is of the essence, DSP engineers continue to use highly optimized assembly code. Less critical functions, however, are now written in C, allowing easier maintenance and better portability. This combination of C and assembly code requires special tools and methodologies in the tool box of every DSP engineer.

It is well known that assembly coding has the advantage of better performance, while C coding is considered much easier and faster to write. To understand why this is so, let's take a closer look at the advantages and disadvantages of assembly coding compared to C coding:

Assembly Coding Pros:

- Assembly code can take advantage of a processor's unique instructions as well as various specialized hardware resources. On the other hand, C code is generic, and must support various hardware platforms. Thus, it is difficult for C to support platform-specific code.
- The assembly programmer is usually very familiar with the application and can make assumptions that are unavailable to the compiler.
- The assembly programmer can use human creativity; the compiler, advanced as it may be, is merely an automatic program.

Assembly Coding Cons:

- The assembly programmer has to handle time-consuming machine-level issues such as register allocation and instruction scheduling. With C code, these issues are taken care of by the compiler.
- Assembly coding requires specialized knowledge of the DSP architecture and its instruction set, whereas C coding only requires knowledge of the C language—which is rather common.
- With assembly code, it is extremely difficult and time consuming to port applications from one platform to another. Porting is relatively easy for C applications.

Figure 1 demonstrates the utilization of dedicated hardware mechanisms for highly optimized assembly code. The C implementation on the left side creates a cyclic buffer p1 using modulo arithmetic. In the highly optimized assembly code on the right, an equivalent buffer is created using the Modulo Mechanism of the CEVA-TeakLite-III DSP Core. The Modulo Mechanism automatically performs the modulo arithmetic whenever there is an update to the buffer pointer (r0 in this case). This arithmetic occurs in the same cycle as the pointer update, so the assembly code is much more efficient than the C code, which would generate separate instructions for the modulo arithmetic.
Choosing the right mixture of C code and assembly code in DSP applications

The question is where to draw the line between C code and assembly code. The answer lies in the performance analysis provided by the profiler. (For more information about the profiler refer to Compiler optimization for DSP applications and DSP optimization strategies using simulators and profilers.) However, before using the profiler, the DSP engineer needs to define clear objectives for the application. Typically, the objectives are cycle count, code size and data size. Once these are defined, the application should first be written and built entirely in C. Only then should the profiler be used to analyze its performance.

In some rare cases, mostly in control applications, C level coding is sufficient. In most cases, the initial C level version of the application does not comply with one or more of the objectives. This usually means that assembly coding is required to some extent. There are many measures that can be taken in the C level to improve performance before resorting to assembly coding, but these measures are not in the scope of this article. (For tips on C-level optimization, see the DSP programmer's guide.) Assuming all C level measures have been exhausted and assembly coding has been initiated, it is highly recommended to save the original C code implementation. This eases debugging and also enables the return to the C implementation once the conditions are right (e.g., when moving to a more powerful platform).

The assembly portion of the code should be kept to a minimum. For this purpose, the performance results reported by the profiler should be analyzed and the critical functions of the application should be identified. Critical functions are those that consume the most execution time and therefore ought to be rewritten in assembly to meet performance objectives. Once the two or three most critical functions have been rewritten, it is time to take another performance measurement. If the application still does not meet its objectives, additional critical functions should be defined and rewritten in assembly. This process iterates until the performance objectives are met.

Compiler related considerations for assembly programmers

When writing assembly code which will later be combined with C code, the assembly programmer has to be aware of compiler conventions and assumptions. One of the important compiler conventions is the function calling convention, also known as the function argument passing convention. This convention describes how the compiler passes arguments when one function calls another. In order for an assembly function to be successfully called from a C function and vice versa, the assembly function must retrieve arguments and send arguments on the hardware resources defined by the function calling convention, which are usually registers or stack memory.
The assembly programmer must also know the compiler's register usage convention. This convention divides the hardware registers into callee-saved (or caller-used) and callee-used (or caller-saved) registers. The compiler assumes that callee-saved registers maintain their value across function calls. If assembly programmers want to use such registers, they must back them up first, and then restore their contents before returning to C code. In contrast, callee-used registers are not assumed to maintain their values across function calls. This means that assembly programmers can use these registers without a back up. However, they need to bear in mind that when their assembly functions call C functions, these registers can be overwritten by the callee.

Figure 2 shows an assembly code example taken from the CEVA-X1641 DSP Core FFT implementation. The highlighted `adds` instruction follows the CEVA-X1641 Compiler's calling convention of passing a pointer argument in the r0 address register. The blue `pushd` instructions back up the callee-saved registers used later in the function.

```assembly
SC.bitrev [en] r0        || LS0.pushd [2dw] r4,r5
SC.adds r0,#4*256,r4     || LS0.pushd [2dw] a8,a9
SC.mov 0x400000, s1      || LS0.pushd [2dw] a10,a11
SC.bitrev [en] r4         || LS0.pushd [2dw] a12,a13
SC.adds r1,#4*2,r2       || LS0.pushd [2dw] a14,a15
SQ.bkrep [ds1] #N/2-2    
|| LS0.ld [2dw] (r0)+s1, a0,a1
|| LS1.ld [2dw] (r4)+s1, a2,a3
SQ.mov #(N/4)/2-1, k11     
{                             
  LS0.ld [2dw] (r0)+s1, a0,a1
  || LS1.ld [2dw] (r4)+s1, a2,a3
  || A.M0.add a0,a2,a4
  || A.M1.add a1,a3,a5
  || A.L.sub a0,a2,a6
  || A.S.sub a1,a3,a7
  || SC.mov #2*256,s2

  LS0.st [2dw] a4,a5,(r1)+#16
  || LS1.st [2dw] a6,a7,(r2)+#16
  || SC.mov #4*2*complex_sizeof, s0
}
```

*Figure 2. A code snippet from the CEVA-X1641 hand-written assembly implementation of the FFT algorithm.*

In addition to calling convention and register usage convention—which are defined for every compiler—some compilers may have additional assumptions regarding hand written assembly code. These assumptions are often specific to the compiler and should be well documented by the compiler's provider. For example, some DSP architectures have memory access alignment limitations. Compilers for these DSPs usually assume that the stack pointer is aligned to a certain width (e.g., 32 bits). This allows the compiler to optimize read and write operations for the stack and use the full memory bandwidth of the machine. It also requires the assembly programmer to make sure the stack is aligned before calling C functions, or a misaligned access might occur.

Another example of compiler assumptions concerns the location of specific instructions in hand written assembly code. For example, the CEVA-X1641 Compiler assumes that a `mov acX, rN` instruction (move an accumulator to an address register) may never be used as the first instruction of an assembly function. This assumption enables better instruction scheduling when filling delay
slots of *call* instructions (call a function). Unique assumptions like this can usually be overridden using a dedicated compilation option.

**Common C language extensions for C and assembly connectivity**

Most compilers for embedded platforms, especially those intended for DSP programming, have a rich set of features for connectivity between C and assembly code. The vast majority of these features are not part of the standard C language and therefore are referred to as C language extensions. Listed below are some of the more useful features for DSP programming.

**Inline assembly.** This feature lets the programmer inject assembly instructions into C code. It is often used in low level C code such as device drivers when it is necessary to access machine resources directly. In most implementations of this feature, the compiler has limited information about the injected instructions and therefore makes worst-case assumptions about their characteristics. Such assumptions may interfere with many compiler optimizations. For instance, in architectures that support parallel execution for some instructions (but not for all), the compiler would not parallel injected instructions with other instructions as it might lead to illegal instruction packets.

**Binding a hardware register to a C variable.** When binding a hardware register to a C variable, the value of the variable in the C code reflects the value of the hardware register and vice versa. Whenever the C variable is read or written, the hardware register is read or written correspondingly. This feature is common in low level code. It is often combined with the assembly instruction inlining feature, allowing the inline assembly code to access C level variables. The example in Figure 3 below demonstrates a common combination of the inline assembly feature (colored orange) and the hardware register binding feature (colored purple).

```c
#include <asm-dsp.h>

int do_absub(int a, int b)
{
    register int s1 __asm__("a0") = a;
    register int s2 __asm__("a4") = b;
    register int s3 __asm__("a5");

    _dsp_asm("absub a0, a4, a5");

    return s3;
}
```

*Figure 3. A code snippet that combines inline assembly with hardware register binding.*

**Section attribute.** By default, the compiler allocates global C variables and functions to standard pre-defined memory sections. The section attribute allows the programmer to allocate them to unique user-defined memory sections. Later on in the linking stage, these sections can be mapped to specific memory addresses. This feature lets the programmer allocate C level elements to exact memory locations, which is crucial for DSP applications. **User-defined calling conventions; intrinsics**

**User-defined calling convention.** As discussed earlier, the compiler has a pre-defined calling convention which the assembly programmer must follow. In some cases, however, assembly functions can be better optimized with a different calling convention. For instance, suppose the compiler convention is to pass parameters in accumulators. A function that performs extensive address calculation is more efficient if it receives its parameters in address registers. For such scenarios, user-defined calling convention allows an alternative local calling convention for specific
functions. The feature relies on a dedicated syntax which is attached to the function’s prototype and informs the compiler of the modified calling convention.

**Compiler intrinsics.** "Compiler intrinsic" is a general name for built-in compiler functionality that can be triggered using dedicated macros or functions calls. For instance, the CEVA-X and CEVA-TeakLite-III compilers provide compiler intrinsics for ETSI/ITU basic DSP operations which are quite common in vocoders. For these operations, the compilers replace each basic operation with its equivalent highly optimized assembly sequence.

Compilers without such built-in support will have to call a user-defined function instead, leading to two major performance drawbacks: First, the user-defined function will generate function calls and returns, possibly inside a loop (as in Figure 4). This creates a tremendous amount of overhead. Second, the user-defined function will be compiled just like any other C code—meaning that the user-defined function is likely to have sub-optimal performance. On the other hand, compilers with intrinsic support already have the optimal implementation built in.

Figure 4 demonstrates how important this feature is. In figure 4, the C code on the left uses ETSI’s `mult_r` (multiply and round) basic operation. In return, the CEVA-TeakLite-III Compiler generates the efficient implementation on the right. The `mult_r` operation is colored purple both in the C code on the left and in the assembly code on the right.

![Figure 4. The CEVA-TeakLite-III Compiler built-in support for ETSI's basic operations.](Click to enlarge)

**Assembly intrinsics.** Assembly intrinsics are an advanced method for inlining assembly code into C code and they are elaborated below.

**Assembly intrinsics - writing assembly instructions as if they were C statements**

The inline assembly feature mentioned above has significant disadvantages:

1. It disrupts various compiler optimizations as the compiler is unaware of the contents of the inlined code and therefore uses worst case assumptions.
2. It might force the programmer to handle low level issues such as register allocation and instruction scheduling.

The assembly intrinsics feature lets the programmer inline assembly code without these disadvantages. From the programmer perspective, assembly intrinsics look just like C macros or functions. They receive C level variables and return C level output while representing a single assembly instruction. Since everything involved with this feature remains in C level, the programmer does not have to worry about register allocation, instruction scheduling and other low level issues. Instead of interfering with compiler optimizations, assembly intrinsics participate in optimizations as if they were regular compiler-generated assembly instructions. These characteristics make assembly intrinsics very powerful.

By using assembly intrinsics, the programmer can benefit from unique assembly instructions that are not likely to be generated by the compiler. Such assembly instructions are usually tailored for
specific algorithms. Using them in the appropriate location can have a dramatic effect on performance. For instance, the bitrev (bit reverse) instruction of the CEVA-X1641 is tailored for algorithms like FFT. Since it is unlikely that the compiler will identify a program as an FFT and use the bitrev instruction, the programmer can simply embed the bitrev assembly intrinsic into the C code.

Assembly intrinsics become even more powerful when combined with the programmer's deep knowledge of the application. Using this knowledge, the programmer can use precise sequences of assembly intrinsics in the performance-critical sections of the C application. By doing so, the programmer can ensure that the compiler produces assembly code that is just as efficient as if it was hand written.

Figure 5 demonstrates the use of assembly intrinsics with the CEVA-X1641 Compiler. The C function on the left uses the st (store, colored red) and msu (multiply and subtract, colored purple) assembly intrinsics. The st intrinsic participates in predication (colored blue) and delay slot filling (colored green). The msu intrinsic participates in loop unrolling (colored orange) and Quad-Mac (colored purple). The assembly intrinsics also benefit from all the machine dependent issues handled by the CEVA-X1641 Compiler such as register allocation, instruction scheduling and hardware unit allocation.

```c
#include <asm/inst.h>

int foo(int cond_p)
{
    int i, sum = 0;
    extern int inp1, inp2;

    if (cond_p)
    {
        st_acX_direct32_dw (inp1, 1234);
    }
    else
    {
        st_acX_direct32_dw (inp1, 4321);
    }

    for (i = 0; i < 1000; i++)
    {
        sum = mnu_aX1_aCY1_aZ (inp2, inp1, sum);
    }

    return sum;
}
```

Debugging applications with both C code and assembly code

Debugging assembly code is not a trivial task. It requires a deep knowledge of the architecture and machine-level issues such as latencies and memory alignment limitations. Having C code alongside the assembly code only makes things harder, as the programmer must now debug the connections between C code and assembly code as well.

The first step when debugging a mixed application is to isolate the problem. Assuming the C level implementation of the assembly code has been maintained—and assuming the C level implementation works correctly—it is relatively easy to swap out assembly functions for their C implementations and re-test the application. To pinpoint problems rapidly, the programmer can use an iterative process: At each step half of the suspected functions are switched to their C implementations, so that at each step the programmer is testing only half as many functions as in...
Once the problematic assembly function has been identified, it should be investigated both for standalone assembly issues and for C and assembly connectivity issues. Debugging standalone assembly issues is quite straightforward for assembly programmers, but C and assembly connectivity issues are somewhat puzzling. Unlike standalone assembly issues, C and assembly connectivity issues are not viewable when looking at the assembly function itself. To find these problems, the programmer must inspect compiler conventions such as calling convention and register usage convention.

The programmer must also check compiler assumptions such as the whereabouts of assembly instructions. (To repeat the example mentioned earlier, the CEVA-X1641 Compiler assumes that a `mov acX, rN` instruction may never be used as the first instruction of an assembly function.) To reduce debugging time, the programmer should verify that all compiler conventions and assumptions are followed when the assembly function is first implemented.

**H.264 video encoder and AMR-NB - real life examples**

At CEVA, the techniques and methodologies described here are used in a variety of applications, including video codecs, audio codecs, vocoders and device drivers. In all cases, the features described here provided a dramatic performance improvement.

The H.264 video encoder is a good case study. It is very demanding in terms of processing power (usually measured in MHz) and other resources, especially when compared to other types of codecs such as voice codecs. CEVA uses its high-end CEVA-X16xx DSP Core family and its MM2000 multimedia platform to provide the processing power required by this encoder.

The critical functions of this encoder were identified using advanced profiling techniques and then optimized. The optimization process of the encoder's critical functions was gradual. First, the functions were fully optimized in C using advanced features like assembly intrinsics. Then, the assembly code provided by the Compiler was further optimized in assembly level.

Figure 6 shows the performance improvement throughout the optimization process of a critical function of the encoder. Only the last optimization stage involved a full scale assembly coding. All other stages were based on C code with assembly intrinsics. The assembly intrinsics were mostly used for SIMD (Single Instruction Multiple Data) operations like `avg_acW_acX_acZ_4b`. This instruction performs byte averaging on eight input bytes, producing 4 byte results. Such SIMD operations are very useful for video codecs that perform a lot of calculations at the byte level. (For this reason, the CEVA-X16xx architecture offers broad support for byte-level SIMD operations.)
The **AMR-NB** (Adaptive Multi Rate - Narrow Band) is a voice codec widely used in wireless communication applications. CEVA has implemented this vocoder for all its DSP cores, but for the purpose of this article, we review only the CEVA-X1620 implementation. It is quite common to implement this vocoder completely in assembly, but using the various features described here, the C implementation and the CEVA-X1620 Compiler can achieve results that are competitive with assembly implementation. One of the most crucial features that boosts the performance of the CEVA-X1620 Compiler is the support for ETSI intrinsics.

Figure 7 shows the performance improvement in MCPS (Millions Cycles Per Second) throughout the optimization process of the entire AMR-NB application. Only the last optimization stage involved a full scale assembly coding. All other stages were based on C code with ETSI intrinsics, assembly intrinsics and so on.

In conclusion, the H.264 encoder and AMR-NB examples clearly show the performance superiority of assembly implementation. However, they also show that a pure assembly implementation should not necessarily be the first optimization method to use. With a high quality software development tool
chain offering various C and assembly features, DSP programmers can reach impressive performance results without implementing their entire application in assembly. As this article shows, writing a combination of C and assembly code is not a trivial exercise. However, the various features discussed here make it easier for the DSP engineer to handle this task.

About the author
Mr. Eran Balaish serves as Senior Compiler Project Manager. In his role, Eran is responsible for planning and managing all Compiler related research and development for CEVA. Since 2003 he has held several development and management positions in R&D, such as leading the development of the CEVA-X1641 Compiler and the CEVA-TeakLite-III Compiler. Eran holds a B.Sc in Electrical Engineering and Computer Science from Tel-Aviv University. He can be reached at eran.balaish@ceva-dsp.com

Related articles:
- Optimizing Compilers and Embedded DSP Software
- Get better DSP code from your compiler
- Programming and optimizing C code
- Tutorial: Programming High-Performance DSPs
- Back to the Basics: Tested and effective methods for speeding up DSP algorithms
- Guidelines for writing efficient C/C++ code