PROGRAMMING THE SANDBRIDGE MULTITHREADED PROCESSOR

Sanjay Jinturkar, John Glossner, Erdem Hokenek, and Mayan Moudgill
Sandbridge Technologies, Inc.
White Plains, NY
914-287-8500
glossner@sandbridgetech.com

Abstract

Programmer productivity is a major concern in the development of complex DSP and SDR applications. As most classical DSPs are programmed in assembly language, it takes a large software effort to develop an application. For modern speech coders it may take up to nine months or more before the application performance is known. Then, an intensive period of design verification ensues. This extended period of development and verification can be minimized if a user-friendly software tool chain capable for generating efficient code for the DSP processor [4] for applications written in C were to be available. Sandbridge Technologies software tool chain is a very user-friendly tool chain, capable of generating highly efficient object code for out of the box C code, and simulating the code on an ultra fast simulation environment. This tool chain provides significant advantages in software productivity.

1 Introduction

Sandbridge Technologies has developed a software tool chain to help the development communications applications on the Sandblaster Platform. The Sandblaster™ platform consists of the Sandblaster DSP processor[4], which does the base band processing. The software tool chain is primarily dedicated towards generating and simulating efficient code for this processor. The basic philosophy behind the tools is that the user should program in a higher-level language such as C not need to use any assembly language code.

The tool chain consists of an Integrated Devp. Environment, ANSI C compiler, functional simulator, and a real time operating system.

2 Paper layout

The paper has been divided into three separate sections. Section 3 and its subsections describe the various software tools available in the tool chain. Section 4 describes the coding guidelines that can be used to generate correct and efficient ANSI C code. Section 6 summarizes the presented issues.

.

3 Software Tool Chain

3.1 Integrated Dev Environment

The Sandbridge Technologies Integrated Development Environment (IDE) provides an easy to use graphical user interface to all the software tools. The IDE is based on the Open source Netbeans integrated development environment [9] The IDE is the graphical front end to the C compiler, assembler, simulator and the debugger. The IDE provides the ability to create, edit, build, execute and debug an application. In addition, it provides the ability to mount a file system, access CVS, access the web and communicate with the Sandblaster hardware board. A snapshot of a typical user session is displayed in Figure 1.

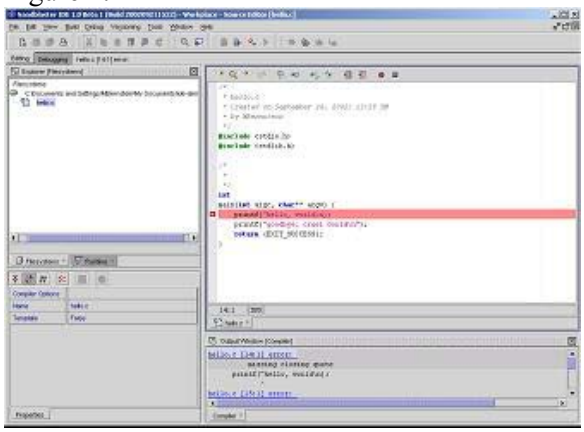

Figure 1. Snapshot of the IDE

As mentioned earlier, the IDE is based on the open source netbeans project. This approach enables it to take advantage of any publicly available netbeans modules that a user may prefer. This model is very advantageous, as a user does not have to rely

on proprietary development methodology or code for extending it.

3.2 Optimizing ANSI C Compiler

There are a number of issues that must be addressed in designing a DSP compiler. First, there is a fundamental mismatch between DSP data types and C language constructs. A basic data-type in DSPs is a saturating fractional fixed-point representation. C language constructs, however, define integer modulo arithmetic. This forces the programmer to explicitly program saturation operations. Saturation arithmetic is analogous to a stereo dial. As you increase the volume it always gets louder until it reaches the limit of the system precision. When this limit is reached the value still remains at the maximum value. If the volume control worked like C language modulo arithmetic the volume would immediately return to "0" after overflowing the precision limit and no sound would be heard. A DSP compiler must deconstruct and analyze the C code for the semantics of the operations represented and generate the underlying fixed point operations.

As DSP C compilers have difficulty generating efficient code, language extensions have been introduced to high-level languages. Typical additions may include special type support for 16-bit data types (Q15 formats), saturation types, multiple memory spaces, and SIMD parallel execution support. These additions often imply a special compiler, and the code may not be emulated easily on multiple platforms. Sandbridge Technologies has built a new best-in-class optimizing ANSI C compiler for the Sandblaster DSP, which doesn't rely on any extensions. This compiler applies a number of high performance compiler optimizations, which enable the generation of very efficient assembly code and obviate the need to write assembly code on this processor. In addition to applying a number of well know scalar and loop optimizations, the compiler applies DSP optimizations and supercomputer-class vector optimizations.

ANSI C does not provide language features to program saturated DSP computations. Therefore, a programmer has to write emulation C code to perform the same operation. The assembly code generated for this emulation C code is very inefficient. Therefore, DSP compilers typically use mechanisms called intrinsics to substitute the emulation C code with equivalent assembly code. However, this requires the user/compiler vendor to specify a predefined mapping between the snippets of assembly code and the emulation C code. Unfortunately, this forces the user to understand the details of underlying processor's assembly language, the details of the compiler's operation and makes the code non-portable and difficult to maintain. This approach is used by compilers on a number of well-known DSPs, such as TI C64x[6], TI 54x [5], StarCore [8]

However, the Sandbridge compiler does not use this approach. Sandbridge has developed proprietary semantic analysis techniques, which eliminates the need for intrinsics. A programmer writes C code in a processor independent manner - such as for a micro controller - focusing primarily on the function to be implemented. If saturated DSP operations are required, then the programmer writes the saturation emulation code in standard modulo C arithmetic. The compiler converts the C code into a dependence flow graph, analyses the range of the arithmetic operations (specifically the sign bit) in the emulation code, propagates it across code segments, determines if it is a saturating or non-saturating operation and emits the correct assembly code. The semantic analysis does not rely on a coding style or patterns in the C source code. This makes the approach very general and applicable to any piece of C code. This technique has a significant software productivity gains over intrinsic functions and does not force the compiler writers to become DSP assembly language programmers.

Another important technique used by the compiler is the exploitation of SIMD instructions. The Sandbridge architecture uses SIMD instructions to implement vector operations. The compiler performs high performance inner and outer loop vector optimizations that use SIMD instructions to exploit the data level parallelism inherent in signal processing applications. These optimizations include vector load, store and multiply-add-reduce-saturate. In conjunction with loop optimizations, these provide very efficient and tight loops that can provide as many as 16 RISC operations in a single cycle. It is important to note that though saturation operations are non associative (i.e. the order of computation is important), they do take advantage of the SIMD instructions. This is because the compiler was designed in conjunction with the processor and

special hardware support allows the compiler to safely vectorize such non-associative operations.

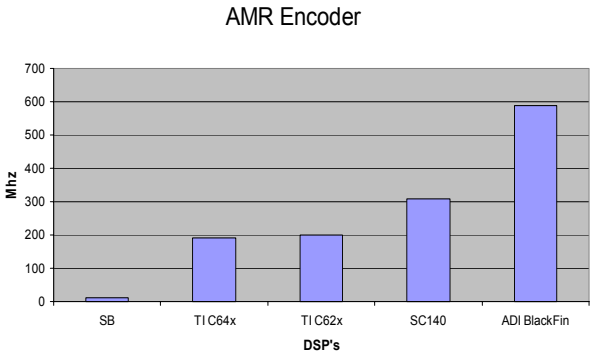

Figure 2. Out-of-the-box AMR ETSI Encoder C code results

We have discussed the compiler technology in the previous sections. This technology enables the compiler to produce very efficient assembly code on out of box C code. Figure 2 shows the results of compilers for state-of-the-art DSPs on out-of-the-box AMR ETSI C code. The x-axis shows the DSP vendor and the y-axis shows the number of MHz required to compute frames of speech in real-time. In all cases, the highest optimization level that produced the logically correct code was used. The AMR code is completely unmodified and no special include files are used. Without using any compiler techniques such as intrinsics or special typedefs, the Sandbridge compiler is able to achieve real-time operation on the Sandblaster™ core at hand-coded assembly language performance levels. Note that it is completely compiled from high-level language.

Since other solutions are not able to automatically generate DSP operations, they have to use proprietary intrinsics to improve their performance. With intrinsic libraries the results for most DSPs are near the Sandbridge results. However, as mentioned earlier, these intrinsics make the code non portable, dependent on the names of the emulation C routines, and harder to maintain. The Sandbridge solution does not suffer from these disadvantages.

3.3 Simulation Environment

Efficient compilation is just one aspect of software productivity. Prior to having hardware, algorithm designers should have access to fast simulation technology. Sandbridge recognizes this fact and has provided an ultra fast cycle counting accurate simulator, which improves the programmer productivity. The simulator uses architecture description of the underlying DSP and provides close to accurate cycle counts, but does not model the external memories or peripherals. However, the information provided by it is sufficient to develop the first executable version of an application.

The simulator is based on Just-in-Time code generation technology, which has been developed in house. This technique is different than the interpretive techniques used in other DSP simulators. In the interpretive technique, the simulator models the target architecture, may mimic the implementation pipeline, and has data structures to reflect the machine resources such as registers. The simulator contains a main driver loop, which performs the *fetch, decode, data read, execute and write back* operations for *each* instruction in the target executable code. Note that these actions are performed *every time* the instruction is executed. In addition, numerous conditional statements have to be executed within the main driver loop as all combination of opcodes and operands have to be accounted for.

The above approach has drawbacks. Actions such as instruction fetch, decode, and operand fetch are repeated *for each* execution of the target instruction. The instruction decode is implemented with a number of conditional statements, which adds more overhead. In addition, the execution of the target instruction requires the update of a number of data structures that mimic the target resources as registers in the simulator.

Our simulator uses the Just-in-Time translation technique. In this technique, the simulator takes advantage of the any apriori knowledge of the target executable and converts the target assembly code to host assembly code before executing any piece of code. Using this approach, the simulator generates host machine code for instruction fetch, decode and operand reads at the beginning of program execution (called the translation phase). The host instructions are then executed at the end of the translation phase. This approach eliminates the overhead of repetitive target instruction fetch, decode and operand read in the interpretive simulation model.

This technique provides very good simulation times. Figure 3 shows the post-compilation

simulation performance of the same AMR encoder for a number of DSP processors. All programs were executed on the same 1GHz laptop Pentium computer. The Sandbridge simulator is capable of simulating 25.6 Million instructions per second. This is more than two orders of magnitude faster than the nearest competitor and allows real-time execution of GSM speech coding on the Sandblaster simulator running on 1 GHz Pentium. To further elaborate, while some DSPs cannot even execute the out-of-box code in real-time on their native processor, Sandbridge achieves multiple real-time channels on a simulation model of processor.

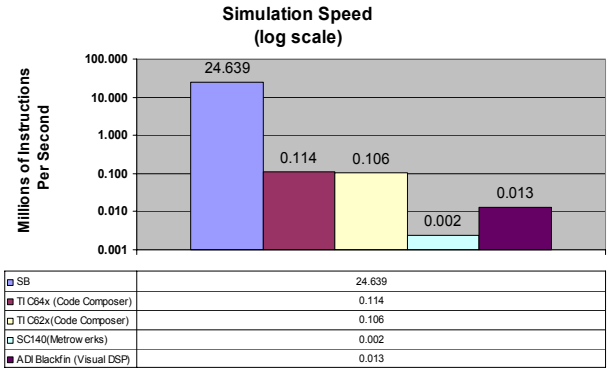

Figure 3. Simulation speed of ETSI AMR Encoder

3.4 Multithreading and RTOS

The Sandblaster DSP is a multi-threaded processor. A programming interface to the resources (multiple threads, peripherals, memories etc.) on the processor is provided via POSIX threads. Keeping with the philosophy of having standard, user friendly and efficient programming model, this interface is based on ANSI C and is commonly used in multi threaded/multi processor environment.

A multi-threaded application has a number of advantages over a single threaded application. – For instance, one thread may be waiting for data from a peripheral while the other can perform some important computation. Thus, the latency of access from the peripheral is hidden.

A thread can be considered a lightweight process. The overhead of creating and deleting a thread is minimal compared to that of a process. A thread possesses an independent flow of control and maintains its own stack pointer, registers, run time data, scheduling characteristics, and a set of pending and blocked signals. However, threads can share process id, address space, working directories, file descriptors, etc. This enables them to access each other's memory locations, access files etc. Obviously, multiple threads can belong to a process.

The first implementation of the Sandblaster processor provides access to a fixed number of threads. These threads are called hardware threads (HT). However, an application (written in C) can contain an infinite number of POSIX threads (called software threads or ST), which are scheduled on the HT's by the underlying RTOS.

The Sandblaster programming interface provides the ability to create, destroy, and join the threads. The underlying operating system supports the ability to prioritize & schedule ST's on HT's, access the peripherals such as A/D & D/A, access various memories etc. The RTOS has been kept lightweight to minimize the overhead.

An example of the use of the programming interface is provided below:

```
#include <stdio.h>
#include <pthread.h>
void *kid (void *arg){
    /* print in the kid thread */
    printf("hello world from kid\n");
    pthread_exit(arg);
       return 0;
}


int main(void){
    pthread_t k; void *v;
    /* Create a new thread (kid) */
    pthread_create(&k, NULL, kid, NULL);

    /* print in the main thread */
    printf("hello world from main\n");
    /* wait for the kid thread */
    pthread_join(k, &v);
    return 0;
}
```

4 SANDBRIDGE CODING GUIDELINES

The Sandbridge tool chain is very robust and efficient because its implementation has followed the coding guidelines (described below) for improving the robustness and efficiency. The guidelines are fairly generic and allow the generation of easy to understand, consistent and bug free ANSI C code. These guidelines are also useful while writing applications code. The guidelines are divided into

safety and efficiency issues. Some of the guidelines are listed below

4.1 GUIDELINES FOR SAFE CODE

These guidelines provide a way to detect common problems such as array bounds checking, range of values assumed by function arguments, common errors in using macros etc.

4.1.1 Function Arguments

All function arguments shall be checked, by asserts, for legal values. This shall immediately after the variable declarations. For example:

```
int
Foo(
    int * p,
    int   n
    )
{
    …
    assert( p != 0 ); /* p must point to
some value */
    assert( n > 0 );
    assert( n < 10); /* n must be in
range [1..9] */
    …
}
```

The function is expected to handle all cases that pass these initial tests.

4.1.2 Macros

All macro arguments shall be terminated by a _. The macro shall be enclosed in parentheses. All uses of a macro argument shall be enclosed in parentheses. Thus:

```
#define max(x_,y_)  ((x_)>(y_)?(x_):(y_))
```

4.1.3 Structure pointer dereferencing:

There shall be no dereferencing of structure pointers inside the C code. Thus the following are forbidden:

```
*str_1.fld_1 = 3;
x = str_1->fld_1;
```

Instead, the following template is followed for defining structures that are to be referenced via pointers. The macros are called accessor macros.

```
#define RVAL(x_) ((void)0,(x_))
typedef struct str_1 str_1; /* typedefs
appear separately */

struct str_1 {
type_1    field_1;
    …
type_n    field_n;
};

#define deref_str_1(s_)      (s_) /* hook
to add debug code*/

#define              x_field_1_str_1(s_)
(deref_str_1(s_)->field_1)

    …
#define              x_field_n_str_1(s_)
(deref_str_1(s_)->field_n)

#define   field_1_str_1(s_)         RVAL(
x_field_1_str_1(s_))
    …
#define       field_n_str_1(s_)       RVAL(
x_field_n_str_1(s_))
```

The x_ macros are used whenever a field is to be set, and the other whenever it is to be read. The fields of the structure should be accessed exclusively using these macros.

4.1.4 Arrays and bounds:

All array references shall be bounds checked in debug mode. The following macro shall be used:

```
#define            bounds(i_,            n_)
    (assert(((unsigned)(i_))<(n_)), (i_))
```

This macro checks that i is >= 0 but < N, and then returns i. This can be used to ensure that an array reference is within bounds using the following idiom:

```
int a[SIZE];
…
… a[bounds(i,SIZE)]…
```

Note that this requires that the size of an array be identifiable at every point that the array is referenced. If an array is passed to a function, then its size must also be a parameter to that function.

```
int
foo(
    int array[],
    int size
    )
{
…
… array[bounds(i,size)]…
…
}
```

If a structure contains an array or a pointer to the array, then it must provide accessors to access individual elements of the array. These accessors must include the bounds check for that array.

```
struct str_1 {
int arrays[SIZE];
int nptrs;
int *ptrs;
};
```

```
…
#define                    x_array_str_1(s_,i_)
(arrays_str_1(s_)[bounds(i_, SIZE)])
#define                     x_ptr_str_1(s_,i_)
(ptrs_str_1(s_)[bounds(i_, \
nptrs_str_1(s_))])
        …
    };
```

4.2 Guidelines for Efficient Code

4.2.1 Minimize usage of malloc()

Use of malloc() adds the overhead of a library call. If possible, one should use statically declared global or stack allocated arrays and structures.

4.2.2 Pass arrays explicitly

It is preferable to pass array arguments explicitly. They should not be enclosed in structures, as this makes compile time analysis hard.

4.2.3 Use arrays of shorts

The basic data type on Sandblaster DSP is a short. Therefore, best code is produced when arrays of shorts instead of arrays of ints or longs are used.

4.2.4 Iteration variable

The iteration variable inside a loop should be of type `int` instead of `short` or `long`. This minimizes the type conversions.

4.2.5 Avoid unrolling loops

It is not necessary to unroll loops manually. Instead, the compiler can do this more efficiently. Unrolling loops manually can actually degrade performance as it interferes with the compilers ability to analyze the code.

4.2.6 Do not convert array accesses to pointers

It is not necessary to convert array accesses into pointer iterators. For example, the code here

```
For (i  = 0; i < N; i++) {
   …A[i]…
}
```

is just as efficient as:

```
pA = A;
For (i = 0; i < N; i++) {
   … *pA …
   pA++;
}
```

while being more readable.

5 CONCLUSIONS

The Sandblaster software tool chain is a very user-friendly tool chain, capable of generating highly efficient object code for out of the box C code. The compiler applies state of the art optimizations and the simulator provides an ultra fast simulation environment. This provides fast turn around times for code development. The tool chain, in conjunction with the software coding guidelines, provides significant advantages in software productivity.

6 BIBLIOGRAPHY

[1] M. Saghir, P. Chow, and C. G. Lee. **Towards Better DSP Architecture and Compilers**. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 658-664, October, 1994.

[2] European Telecommunications Standards Institute, Digital cellular telecommunications system, **ANSI-C code for GSM AMR speech codec** (version 7.1.0), July, 1999.

[3] K.W. Leary and W. Waddington, **"DSP/C: A Standard High Level Language for DSP and Numeric Processing"**, *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, IEEE, 1990, pp. 1065-1068.

[4] Sandbridge Technologies, "Sandblaster DSP Overview", www.sandbridgetech.com

[5] Texas Instruments, "*TMS320C54x DSP Reference Set. Volume 1: CPU and Peripherals"*, TI Report number SPRU131E, June, 1998.

[6] Texas Instruments, "*TMS320C6000 CPU and Instruction Set Reference guide, Oct 2000* .

[7] Lucent Technologies, *"DSP 1611/17/18/27/28/29 Digital Signal Processor Information Manual"*, January, 1998.

[8] StarCore, "*SC140 DSP Core*", Nov 2001.

[9] www.netbeans.org, "Netbeans IDE".