SOFTWARE CENTRIC APPROACH TO DEVELOPING WIRELESS APPLICATIONS

Sanjay Jinturkar, Vaidyanathan Ramadurai, Sanyogita Shamsunder, Mayan Moudgill, John Glossner

Sandbridge Technologies, 1 North Lexington Avenue, White Plains, NY 10601
sjinturkar@sandbridgetech.com

ABSTRACT

There is an increasing focus on implementing complex wireless applications in software. Software implementation enables re-configurability and lowers the development costs. Such an implementation is frequently done in a higher level language such as C. The implementation process has two major components - A well architected C code, followed by efficient compilation and execution of it on the target platform. In this paper, we focus on the software implementation of 802.11 physical layer using the Sandblaster tools [2] for the Sandblaster multithreaded processor [1].

1. INTRODUCTION

The Sandblaster multithreaded processor is an excellent platform for development of wireless and multimedia applications. Physical layers of number of 2G & 3G wireless applications have been developed on this platform. The development has been done using the Sandblaster software tools[2], which facilitate development in C and obviate the need to write any assembly language.

In this paper, we discuss the implementation of the physical layer for 802.11 transmitter [5] on the Sandblaster processor. The techniques described here can be generalized towards the development of other wireless applications.

2. SANDBLASTER MULTITHREADED PROCESSOR

Sandbridge Technologies has developed the Sandblaster architecture for a convergence device. As handsets are converging to multimedia multi-protocol systems, the Sandblaster architecture supports the data types necessary for convergence devices including RISC control code, DSP, and Java.

As shown in Figure 1, the design includes a unique combination of modern techniques such as a SIMD Vector/DSP unit, a parallel reduction unit, and a RISC-based integer unit. Each processor core provides support for concurrent execution for up to eight threads of execution. All states may be saved from each individual thread and no special software support is required for interrupt processing. The machine is partitioned into a RISC-based control unit that fetches instructions from a set-associative instruction cache. Instruction space is conserved through the use of compounded instructions that are grouped into packets for execution.

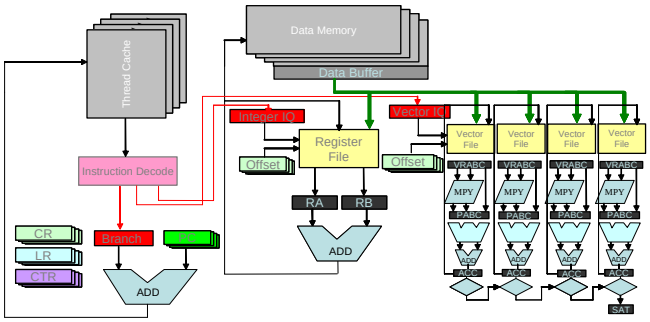

Figure 1: Sandblaster Multithreaded Processor

The memory subsystem has been designed carefully to minimize power dissipation. The pipeline design in combination with the memory design ensures that all memories are single ported and yet the processor can sustain nearly 4 taps per cycle for a filter (the theoretical maximum) in every thread unit simultaneously. A RISC-based execution unit, depicted in the center of Figure 1, assists with control processing.

Physical layer processing often consists of control structures with compute-intensive inner loops. A base band processor must deal with both integer and fractional data types. For the control code, a 16 entry, 32-bit register file per thread unit provides for very efficient control processing. Common integer data types are typically stored in the register file. This allows for branch bounds to be computed and addresses to be efficiently generated.

Intensive DSP physical layer processing is performed in the SIMD/Vector unit depicted on the right side of Figure 1. Each cycle, a 4x16-bit vector may be loaded into the register file while two vectors are being multiplied,

saturated, reduced (e.g. summed), and saturated again. The branch bound may also be computed and the instruction looped on itself until the entire vector is processed. This may be specified in as little as 64-bits. This compares very favorably to VLIW implementations.

To enable physical layer processing in software, the processor supports many levels of parallelism. Thread-level parallelism is supported by providing hardware support for up to 8 independent programs to be simultaneously active on a single Sandblaster core. This minimizes the latency in physical layer processing. Since many algorithms have stringent requirements on response time, multithreading is an integral technique in reducing latencies.

In addition to thread-level parallelism, the processor also supports data-level parallelism through the use of a Vector unit. In the inner kernel of signal processing or base band routines, the computations appear as vector operations of moderate length. Filters, FFTs, convolutions, etc., all can be specified in this manner. Efficient, low power support for data level parallelism effectively accelerates inner loop signal processing.

3. SANDBLASTER TOOLS

Sandbridge Technologies has developed a software tool chain to facilitate the development of communications applications on the Sandblaster Platform. The software tool chain is primarily dedicated towards generating and simulating efficient code for the Sandblaster processor. The basic philosophy behind the tools is that the user should program in a higher-level language such as C and not need to use any assembly language code. The tool chain consists of an Integrated Development Environment, ANSI C compiler, functional simulator, and a real time operating system.

The 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 [4]. 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 the IDE is shown in Figure 2.

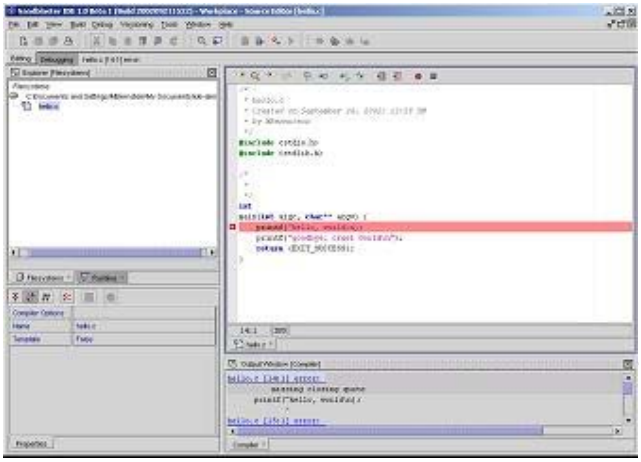

Figure 2: Integrated Development Environment

The software tool set also consists of an optimizing ANSI C compiler. This C compiler performs a number of high performance optimizations, including the generation of saturation instructions from out of the box C code. This obviates the need to write assembly language code or to use machine specific intrinsics for saturation code. The compiler has proprietary semantic analysis techniques, which eliminate the need for intrinsics. A programmer writes C code in a processor independent manner and the compiler converts the C code into a dependence flow graph, analyzes 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.

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.

Sandbridge has also developed 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.

The tool set also consists of an operating system kernel which provides access to the resources (multiple threads, peripherals, memories etc.) on the processor. This is provided via POSIX threads interface. Keeping with the philosophy of having a standard, user-friendly and efficient programming model, this interface is based on ANSI C and is commonly used in multi-threaded/multi processor environment.
All the above tools and interfaces have been used in the development of the WLAN physical layer.

4. WLAN TRANSMITTER

The 802.11 transmitter is responsible for converting data bits into streams of I & Q samples. These samples are modulated, spread, over-sampled and filtered before being converted to analog signals and transmitted into the air. Figure 3 shows the major blocks of the transmitter.

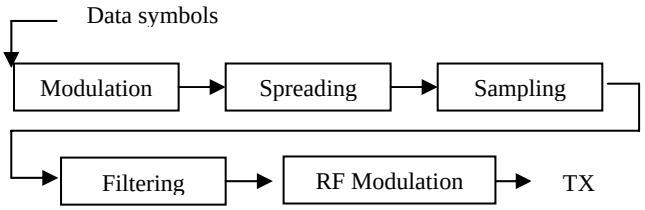

Figure3 Block diagram of a WLAN transmitter

The differential modulator converts a stream of bits into symbols by using either Differential Binary Phase Shift Keying (DBPSK for 1 Mbps) or Differential Quadrature Phase Shift Keying (DQPSK, for 2 Mbps). The spreader converts the symbols into chips by multiplying the I & Q symbols by a 11 bit Barker code. Each chip is then over sampled by a factor of 2. The over-sampling is followed by a pulse shaping filter, which limits the chips to a certain frequency range. These chips are then processed by the D/A and the analog front end before being transmitted over the air [5].

If the above blocks had been implemented in hardware, there would be a separate block for each of the above functionality. However, when they are implemented in software, these artificial boundaries can be eliminated, and blocks can be combined together to eliminate redundant computations. This approach requires a careful analysis of the entire processing chain to:

- Identify blocks that can be merged and eliminate redundant memory accesses between the blocks.
- Identify blocks where computations can be performed a priori, and the precomputed values used at execution time;
- Use coding guidelines that will maximize the parallelism when the code is executed on the target hardware/simulator.

We have used the above techniques to implement the transmitter and receiver on the Sandblaster platform.

4.1 Eliminating Redundant Blocks

In a hardware implementation, modulation, spreading, over sampling and filtering are functionally separate blocks. However, on a signal processor with efficient software tools, there is no reason for them to exist as separate functions, as the visibility to all the blocks can improve the quality of optimizations applied to the blocks by the compiler. Therefore, the compiler performs function inlining optimization[6] on all the blocks and eliminates redundancies between them

4.2 Precompute Values

The computations performed by multiple blocks described above can be done ahead of time, and stored in a table. The table can be dereferenced at execution time. This eliminates the need to generate the same information for every input symbol. This is possible because the I & Q values are a combination of a known pattern of input bits provided by the WLAN Medium Access Layer (MAC) to the physical layer.

As an example, let us assume that a set of bits {1, 0, 1, 1,1,1,0...} are being passed by the MAC layer to the physical layer. In the 1 Mbps case, each symbol consists of one bit. Each symbol is mapped to a pair of I & Q, where I & Q are 1 or -1, depending on the previous symbol (as the modulation scheme being used is differential modulation).

In the above example, if the first symbol 1 is mapped to (1, -1), then the second symbol 0 is also mapped to (1, -1). This is because there is no phase shift when a 0 is encountered. However, when the next symbol is 1, then the corresponding output symbol is rotated by 180 and the I & Q values are (-1, 1). Figure 4 shows the pre-computed information, which is de-referenced at execution time.

Previous State (I,Q)	Modulator Output	
	Input bit = 0	Input bit = 1
1, 1	1,1	-1,-1
1, -1	1,-1,	-1,1
-1,1	-1,1	1,-1
-1,-1	-1,-1	1,1

Figure 4: Table lookup for modulation

This approach can be extended to the spreading step of the chain. In 802.11, each input bit is multiplied with a 11 bit Barker sequence {1,-1,1,1,-1,1,1,1,-1,-1,-1}[5]. The values can again be pre-computed and stored in a table. The spread symbols are shown in Figure 5.

Spreader Input Symbol (I, Q)	Spreader Output Sequence
1, 1	1-111-1111-1-1-1,1-111-1111-1-1-1
1, -1	1-111-1111-1-1-1,-11-1-11-1-1-1111
-1,1	-11-1-11-1-1111,1-111-111-1-1-1
-1,-1	-11-1-11-1-1-1111,-11-1-11-1-1-1111

Figure 5: Table lookup for Spreading

If the values were to be computed at run time, then this would require 22 multiplications. However, when the values are pre-computed, a single dereference from the memory is sufficient.

The next step in the chain is over-sampling, where each chip is replicated M times, where M is the over-sample factor. This can again be done statically.

Sequence before over sampling	Sequence after Over sampling (by 2)
1-111-111-1-1-1,1-111-111-1-1-1	11-1-11111-1-1111111-1-1-1-1-1-1,11-1-11111-1-1111111-1-1-1-1-1-1

Figure 6: Table lookup for Over sampling

The final computation step is the pulse shaping filter, where the over sampled chips are multiplied with a coefficient and summed to get the filtered samples. Given that there is a finite set of values that are being input to the filter, the output of the filter can be predetermined and stored in the lookup table. Table 7 shows the sample output from the filter for a known set of input sequence. Note that in this example, the filter length (memory) is less than 2 symbols. If the filter is longer than 22 samples, then the previous 2 states are needed. This also results in a larger table.

Sequence before filtering	Sequence after filtering
1-111-111-1-1-1,1-111-111-1-1-1	512 455 512…..

Figure 7: Table lookup for filtering

Clearly, instead of repeated computations, only repeated memory referencing is done. The complexity can be further reduced by combining all the tables into a single table, where by the multiple table lookups can be minimized to a single lookup. For instance for a previous state (1, 1) and input 0, the entries in the combined table in Figure 8.

Note that this table has eight entries, and they can be accessed using a three bit index which is generated from the current & previous input bits to the modulator This methodology can easily be extended to the 2, 5.5. and 11 Mbps data rates.

(Previous state, Input sample)	Sequence after filtering
(1,1,0)	512 455 512…..
(1,1,1)	512,…
(1,-1,0)	…
(1,-1,1)	…
(-1,1,0)	…
(-1,1,1)	…
(-1,-1,0)	…
(-1,-1,1)	…

Figure 8: A combined table lookup

4.3 Using Coding Guidelines

In the above section, we described the table lookup mechanism that reduced computation at execution time. The generation of the table lookups can be optimized by adhering to C coding guidelines [2]. These guidelines enable the combination of multiple tables into a single table, and their subsequent vectorization.

5. SUMMARY

In this paper, we have described the software implementation of the WLAN transmitter in C using the Sandblaster tools. Table lookup techniques enable apriori computation of various blocks of the transmitter, and thus minimize the load on the processor. The table lookup techniques can be extended to the development of other physical layers also.

6. REFERENCES

[1] Sandbridge Technologies, “Sandblaster DSP Overview”, www.sandbridgetech.com

[2] Sanjay Jinturkar , John Glossner, Mayan Moudgill, Erdem Hokenek, “Programming the Sandblaster Multithreaded Processor”, GSPx 2003.

[3] Sanjay Jinturkar , John Glossner, Vladimir Kotlyar, Erdem Hokenek, “The Sandblaster Automatic Multithreaded Vectorizing Compiler”, GSPx 2004.

[4] www.netbeans.org, “Netbeans IDE”.

[5] IEEE 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications, IEEE 802.11 Std. (August 1999)

[6] David Bacon et. al, “Compiler Transformations for High Performance Computing” Tech report UCB-CSD-93-781, Univ. Of California, Berkeley, Nov 1993.

[7] Jin Lu, Mayan Moudgill “Fast transmitter based on table lookup, US Patent application # 20040096011