Instruction Set Extensions for the Advanced Encryption Standard on a Multithreaded Software Defined Radio Platform

Christipher Jenkins*
Department of Electrical and Computer Enginering,
University of Wisconsin-Madison,
Madison, WI
E-mail: cdjenkins@wisc.edu
*Corresponding author

Suman Mamidi
Department of Electrical and Computer Enginering,
University of Wisconsin-Madison,
Madison, WI
E-mail: mamidi@cae.wisc.edu

Michael Schulte
Department of Electrical and Computer Enginering,
University of Wisconsin-Madison,
Madison, WI
E-mail: schulte@engr.wisc.edu

John Glossner
Sandbridge Technologies,
White Plains, NY
E-mail: jglossner@sandbridgetech.com

Abstract: Software-defined radio (SDR) is an emerging technology that facilitates having multiple wireless communication protocols on one device. Previous work has shown that current W-CDMA, GPS, GSM, and WiMAX applications can run on this class of device while consuming significant processing power. Next generation wireless networks require speeds in excess of 50Mbps. Some of the fastest AES software implementations only achieve 20Mbps on our reference platform. In order to have secure software-defined radio, the security processing gap must be addressed. This paper presents instruction set architecture (ISA) extensions for the Sandblaster DSP. The Sandblaster DSP is a multithreaded processor for SDR that issues multiple operations each cycle and supports vector operations.

Keywords: AES; multithreaded; security; software defined radio; instruction set extensions; hardware design.

Reference to this paper should be made as follows: C. Jenkins, S. Mamidi, M. Schulte, J. Glossner. (2007) 'Instruction Set Extensions for the Advanced Encryption Standard on a Multithreaded Software Defined Radio Platform', International Journal on Embedded Systems

Biographical notes: Christipher Jenkins is a Master's student in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His work focuses on security extensions for software-defined radios.

1 Introduction

Mobile devices have evolved from using basic embedded processors to multi-functional devices using dedicated digital signal processors (DSPs) and application specific integrated circuits (ASICs) to accomodate the growing demand for smaller and faster embedded appliances. Many of these devices, which primarily use dedicated DSPs and ASICs, require new chip designs to efficiently implement new wireless standards and specifications. The primary disadvantage of this methodology is the cost to design and manufacture new chips. Another proposed methodology is to move the wireless baseband processing from dedicated DSPs and ASICs into software on programmable domain-specific processors. In this scenario, software updates can add wireless standards, features, or improved algorithms without a new device being needed. A current approach to programmable communications systems is software defined radio (SDR).

SDRs, which use a combination of software and hardware to dynamically support multiple communications standards, have been widely recognized as one of the most important new technologies for wireless communication systems [16]. SDRs enable the efficient implementation of a diverse set of wireless communication systems by providing the ability to change communication protocols and dynamically update communications systems through over-the-air software downloads [14]. Thus, SDRs can easily adapt to emerging standards. As new standards become available, software updates can be used to enable new communication protocols without the need for new hardware. Although the flexibility and programmability of SDRs make them ideal for use in future wireless communication systems, several challenges must be addressed before SDRs can be widely used in systems that require high levels of security, confidentiality [3], and performance.

SDRs sometimes use multithreading to alleviate some of the potential performance challenges of baseband processing. Traditionally, a single thread runs on a processor for some amount of time before the operating system (OS) removes that thread to execute another. At any given time, only one thread executes on the processor. Research has shown that this execution style leads to ununsed functional units and large context switch overhead [27]. Functional units may go unused due to data dependecies or resource hazards. On the other hand, multithreading allows independent threads to issue during the same cycle or on alternate cycles without the need for context switching or additional processors. The unused functional units can be used by other threads, thus allowing higher throughput and utilization.

In current cellular systems, encryption is often done in software as the necessary data rates can be achieved on a programmable processor. As bit rates increase, security for wireless data presents a new set of challenges not commonly found in general purpose processors (GPPs). One of the most formidable challenges is the security processing requirements. For example, a typical secure communication channel provides authentication and key exchange (RSA, DSA, ECC) to setup a transfer with symmetric-key encryption (3DES, AES, RC4) and message integrity (SHA, MD4, MD5) during the bulk of the transfer. For example, a 10Mbps transfer, using 3DES and SHA, requires 653.1 millions of instructions per second (MIPS) on the StrongARM SA-1110 processor [23]. This is beyond the performance of many embedded processors [1, 18]. High-Speed Downlink Packet Access (HSDPA) [22] is a current high-end cellular protocol with data rates of up to 14.4Mbps. While there are dedicated solutions which can reach this speed, these solutions require dedicated silicon area that cannot be reused.

Next generation mobile networks are expected to exceed 50 Mbps [5]. Well below this data rate, the advanced encryption standard (AES) becomes a bottleneck in software, and an efficient solution must be developed to meet this growing demand. We chose to accelerate AES as it is the current government standard for encrypting confidential data [19]. Likewise, AES is the new standard for encryption in wireless devices using Wi-Fi Protected Access 2 (WPA2 or 802.11i) [20]. Hence, prevelance in multiple standards and government requirements makes AES a useful protocol to accelerate. While much work has been done to show that sufficient baseband processing capabilities are available in SDRs [25], instruction set architecture (ISA) support for secure SDR has not been a focus of previous research.

This paper prsents a hardware design and instruction set extensions for AES processing on a multithreaded SDR platform, the Sandbridge SB3010 System. Our design satisfies the microarchitecture constraints of our SDR platform. These constraints are: 1) the ability to operate efficiently in a multithreaded microarchitecture, 2) encryption and decryption have similar performance, 3) no hidden state is added to the programming model, 4) a throughput of at least 50Mbps at the highest cipher level, and 5) small area compared to the DSP core. All of these goals are achieved though an efficient AES processing solution for multithreaded SDRs. To the best of our knowledge, this is the first time instruction set extensions for AES on a multithreaded SDR platform have been proposed and analyzed.

The rest of the paper is organized as follows: Section 2 describes the AES algorithm and its processing requirements. Section 3 presents our proposed implementation for AES acceleration. Section 4 demonstrates the performance and design characteristics of our solution. Section 5 provides conclusions and discusses future work.

2 Background on AES

AES is the current government standard for encrypting electronic data [19]. AES performs encryption and decryption on 128-bit blocks partitioned into four 32-bit columns known as the state. The AES algorithm is composed of byte transformations that occur over several rounds. Each

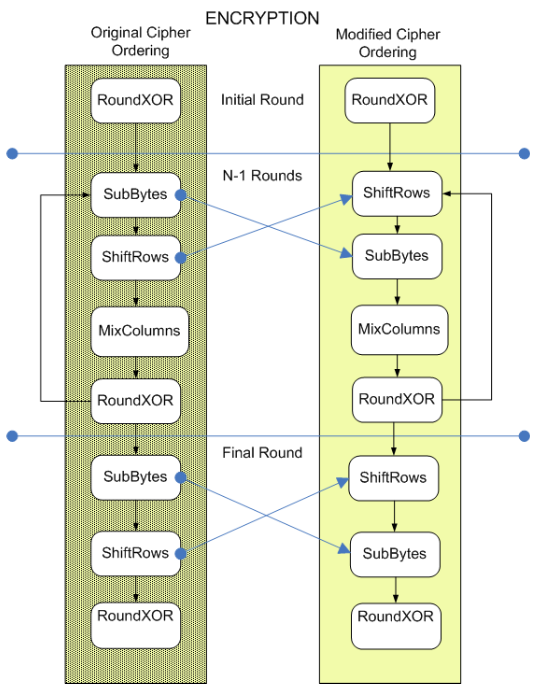

Figure 1: AES Encryption Algorithm

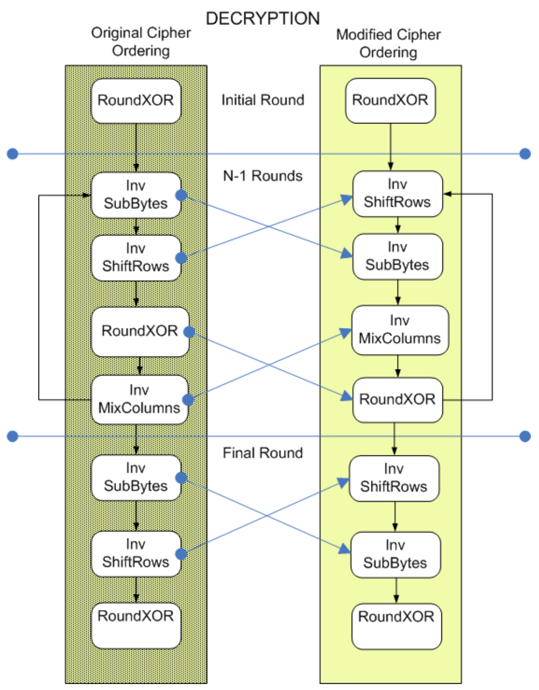

Figure 2: AES Decryption Algorithm

round uses a different key, known as a round key, that is extracted during a key expansion process, which usually occurs before the cipher takes place. The number of rounds depends on the key length, which can be 128 bits (10 rounds), 192 bits (12 rounds), or 256 (14 rounds) bits.

Regardless of the key length, the AES encryption and decryption algorithms follow the same structure as, shown in Figures 1 and 2. The initial round is simply an XOR of the plaintext data with the original key. The rest of the rounds are equivalent transformations with a slight variation during the final round. The four transformation that make up each round of the encryption core are: SubBytes, ShiftRows, MixColumns, and RoundXOR. For decryption they are: InvSubBytes, InvShiftRows, RoundXOR, and InvMixColumns. (Inv)ShiftRows rotates (left) right each corresponding row in the state by 0, 1, 2, or 3 bytes. (Inv)SubBytes does a byte substitution on each byte in the state based on a multiplicative inverse and (inverse) affine transform in the Galois field $GF(2^8)$. (Inv)MixColumns multiplies each column independently by a predefined polynomial over $GF(2^8)$. RoundXOR is the same for both encryption and decryption and is simply the bitwise XOR of the state and the round key. More details on the AES algorithm are given in [19].

3 Proposed AES Implementation

To speed up AES for high bandwidth use, we target data rates of 50Mbps and higher. One of our goals is to provide AES functional units suitable for use inside of a programmable SDR, as opposed to an ASIC or custom chip design. The main reason for low performance in the AES algorithm stems from the need for byte manipulations on processors with 16, 32, or 64-bit datapaths. Three of the four transformations in AES manipulate data at the byte level. Hence, compilers must insert additional instructions to perform byte manipulation. While SIMD support may help for some byte operations, some of the algorithmic manipulations such as (Inv)SubBytes, (Inv)ShiftRows, and (Inv)MixColumns cannot be easily vectorized and are not well supported on conventional embedded processors. For increased throughput, either specialized instructions or increased processor clock rate must be employed. Target peformance can be achieved by increasing the clock speed, but to keep low power, the clock speed is not increased beyond a certain threshold. Since opcode space is often an issue in embedded systems, we introduce just seven new instructions. These instructions are summarized in Table 1.

Name and Format	Description
roundXOR vrd, vra, vrb	Performs RoundXOR
shift_rows vrd, vra	ShiftRows
sbox_mix_xor vrd, vra, vrb	Performs SubBytes, MixColumns, and RoundXOR
sbox_xor vrd, vra, vrb	Performs SubBytes and RoundXOR
shift_rows_inv vrd, vra	Performs Inverse ShiftRows
sbox_mix_xor_inv vrd, vra, vrb	Performs InvSubBytes, InvMix-Columns, and RoundXOR
sbox_xor_inv vrd, vra, vrb	Performs InvSubBytes and RoundXOR

Table 1: Proposed ISA Extensions

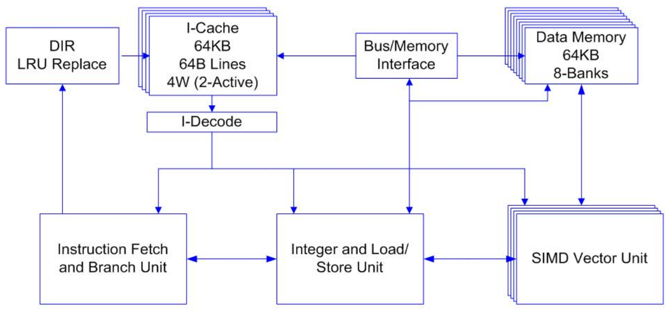

Figure 3: Sandblaster DSP Core

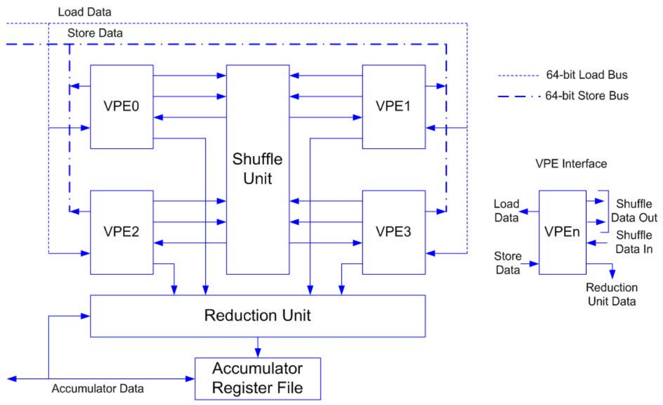

Figure 4: SIMD Vector Unit

3.1 SB3010 SDR Platform

The SB3010 SDR Platform contains four Sandblaster DSP cores, an ARM processor, and input and output peripherals found on many wireless handheld devices. Each core is multithreaded, runs eight threads simultaneously [15], and is paritioned into three units, as shown in Figure 3. These are an instruction fetch and branch unit, an integer and load/store unit, and a SIMD vector unit. Integer operations have up to two source operands and one destinatin operand, and vector operations have up to three source operands and one destination operand. Vector operations process sets of four vector elements simultaneously. To save on program memory, the SB3010 uses compound instructions which can issue up to three operations per cycle. To support eight hardware threads, the architecture uses a uniquire form of interleaved multithreading (IMT) known as token thread triggering (T^3) [15]. Each thread occupies one pipeline stage per cycle. Each cycle each thread advances forward until it has passed through all eight stages. Each thread appears to itself as running on a single-cycle processor. By using T^3 complex dependency hardware can be removed to reduce power and area.

The SIMD vector processing unit (VPU) conists of four vector processing elements (VPEs), a shuffle unit, a reduction unit, and an accumulator register file, as shown in Figure 4. The VPU simultaneously performs arithmetic and logic operations on 16-bit and 32-bit fixed-point data types in each VPE. It takes one cycle to load 16-bit data and two cycles to load 32-bit data. The VPU allows up to four execution cycles, which is hidden due to the T^3 microarchitecture of the Sandblaster DSP. Our AES ISA extensions are implemented in the vector processing unit. The Sandblaster toolchain uses an optimizing ANSI C compiler [25] to generate code for the DSP core. It provides a fast cycle count accurate simulator [25, 10], which uses the Sandbridge architecture Description Language (SaDL) to specify the underlying architecture. The compiler supports instrinsics, which are user-defined functions supported at the ISA level in the Sandblaster simulator, to model new instructions.

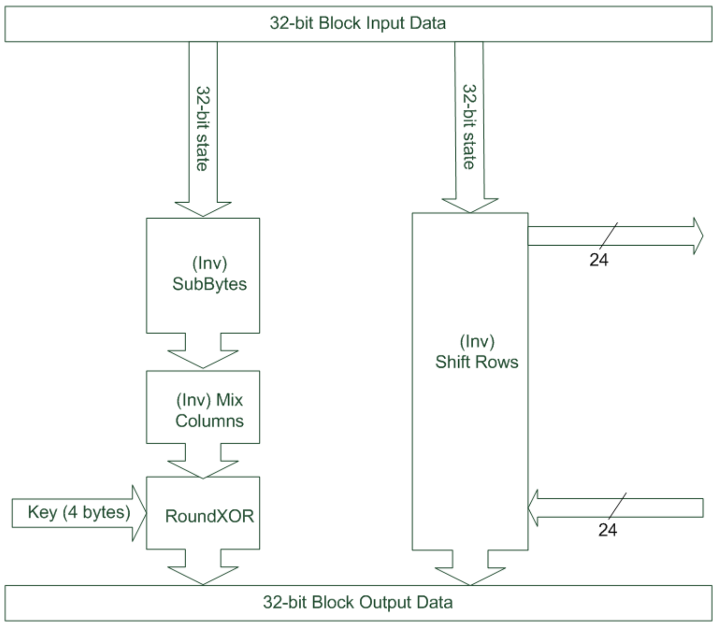

Figure 5: AES Hardware Design in each VPE

3.2 AES Module Design

As stated in the Section 1, five main constraints guided our design. To operate efficiently in a multithreaded microarchitecture, we provide AES instruction set extensions. ASIC designs add hidden state to the programming model, and we want full visibility how instructions affect the processor state. The AES algorithm allows some transformation re-ordering while maintaining correctness as shown in Figures 1 and 2. We used two manipulations which allowed for a more efficient design. First, we re-ordered

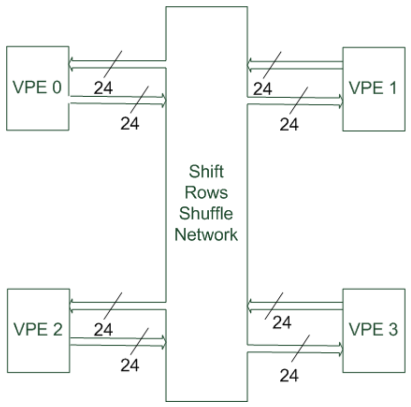

Figure 6: (Inv)ShiftRow shuffle network connecting VPEs

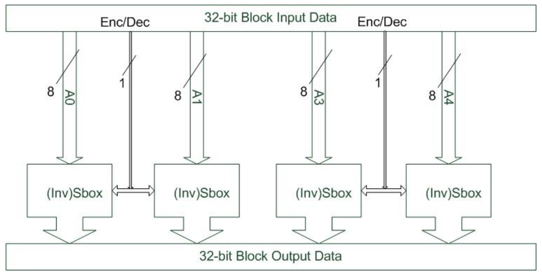

Figure 7: (Inv)SubBytes unit inside each VPE

(Inv)ShiftRows and (Inv)SubBytes transformations. This allows the main encryption loop to be divided into two parts (instructions), as shown in Figure 5. The first part, shown on the right, is simply the (Inv)ShiftRows function, which shifts out three bytes and shifts in the corresponding three bytes from the other VPEs as shown in Figure 6. The second part, shown on the left, implements the (Inv)SubBytes, (Inv)MixColumns, and RoundXOR functions. The state of AES, as mentioned before, is partitioned into four columns. The Sandblaster vector unit is well-suited for AES as its vector registers contain four 32-bit elements, which matches the size of each column of the AES state. Thus, each column is operated on independently for the the second part.

Each VPE contains four tables as shown in Figure 7. The tables contain both forward and inverse Sbox lookups. The Sbox lookups perform the (Inv)SubBytes transformation. Each byte of the VPE word is used an an index into the table, and a 1-bit selector chooses the output depending on if encryption or decryption is being performed. Each table contains 512 bytes, which corresponds to 8Kbytes (16 tables total) for the entire design. The MixColumns block uses the architectural design of (Inv)MixColumns as defined in [13].

Each VPE contains four MixColumns Compute boxes as shown in Figure 8. Each MixColumns Compute box performs a Galois field multiplication operation on each byte depending on the Enc/Dec bit. The XOR logic tree XORs the appropriate output of each MixColumns Compute box to produce the correct byte to complete the (Inv)MixColumns operation. Lastly, the resulting four bytes are finally XORed with the round key for that VPE.

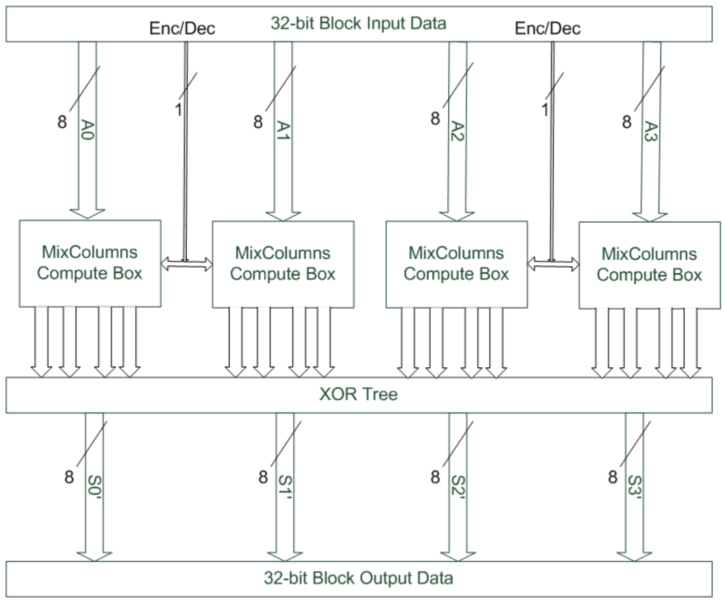

Figure 8: MixColumns unit inside each VPE

Our design depends on using the equivalent reverse cipher as defined in the NIST standard [19]. Hence, we do not use the same set of round keys for both encryption and decryption. However, by transforming the round keys and switching InvMixColumns and RoundXOR (Figure 2), encrpytion and decryption use the same design and require the same latency to process a block. In a pure software implementation with table lookups only for the sbox, decryption takes longer than encryption [11].

3.3 VEX Architecture and Simulator

For completeness, we also model a high-end DSP using the VLIW-Example (VEX) architecture and use this model to determine the performance of AES without instruction set support [7]. VEX is a 32-bit clustered VLIW processor based on the HP/ST Lx ST200 [6] family of embedded VLIW processors. A cluster is defined as a collection of registers and associated functional units in a tightly coupled configuration. The VEX architecture can be configured to contain multiple clusters, but for our design we model a high-end VLIW processor that has one cluster and can issue up to four operations per cycle. The simulator allows defining the number and type of funtional units, latencies, and other architecture parameters that are common to embedded environments. Scheduling of operations is done statically by the compiler and the execution strictly adheres to program order. Our VEX processor configuration parameters are shown in Table 2.

CPU Features	General Purpose Registers 64 Branch Units 4 Issue Width 4 Total Store Operations 1 Total Load Operations 1 Integer ALU Slots 2
L1 Cache Config	Instruction cache size 64 KB 4-way Line size 256 bytes Miss penalty 40 cycles Store Miss Penalty 56 cycles WB Miss Penalty 102 cycles
System specs	Prefetch units 1 Multiply slots 1 Memory slots 2 Prefetch enabled Stream buffer disabled

Table 2: VEX Processor Configuration Parameters

3.4 Methodology

Often, encryption and decryption are executed using a single thread. Using more than one thread gives an increase in throughput, but in most SDR systems, as many threads as possible should be available for baseband and multimedia processing needs. To test the potential of our design, we chose the StrongARM SA-1110 processor as our baseline [2]. The StrongARM SA-1110 is representative of current embedded cores which can perform AES encryption/decryption for most of today's cellular standards. Next, we ran C code[1] [9] on VEX [21]. Then, we ran the same C code on the Sandblaster DSP for three AES implementations; unrolled loops, conventional loops, and an optimized swap method.

For testing our ISA extensions, the C code for rounds is replaced by eqivalent intrinsics. No other code was modified, except for the swap optimization. The swap optimization removes a bottleneck from the original code. During the AES loop a copy method is employed to copy over data so pointer references do not have to change–allowing the same function to be called again without changing the argument order to the function. We used a property of XOR to swap pointer values instead of the expensive cost of copying data. The properties is as follows,

$p1$ and $p2$ are pointer values stored in x and y respectively

$x = x \oplus y = p1 \oplus p2;$

$y = y \oplus x = p2 \oplus p1 \oplus p2 = p1;$

$x = x \oplus y = p1 \oplus p2 \oplus p1 = p2;$

Thus, three XOR operations, swap the pointer values *p1* and *p2*.

Prior to running the above tests, we coded a model of the hardware in C to verify functional correctness. We also coded our own AES implementation in C using instrinsics. This was done to see how the compiler maps our intrinsics to determine if hand optimizations are needed after inserting intrinsics. For each test, we ran the algorithm over 256KB of data, and key generation occurred prior to encryption/decryption.

[1]Our configuration used a fast AES implementation with 4 lookup tables

Unit	Delay (ns)	Area (um^2)
(Inv)SubBytes	1.14	50816
(Inv)MixColumns	0.77	6698
RoundXOR	0.05	657
AES VPE Unit	2.28	61206
VPU (Inv)Shift Rows Shuffle Unit	0.05	1667
VPU Total (calculated)	2.28	246491

Table 3: Area and Delay Estimates

4 Results and Analysis

We modeled our AES hardware design using Verilog and synthesized it using Synopsys Design Compiler and LSI Logic's GFlx-p 0.11 micron standard cell library to obtain area and delay estimates, which are presented in Table 3. The (Inv)SubBytes and (Inv)MixColumns units correspond to the units shown in Figures 7 and 8, respectively. The AES VPE unit corresponds to the units shown in Figures 5. The (Inv)ShiftRows Shuffle Unit has only a minor impact on the overall area as it primarily performs routing and is not combinational. The AES entire VPE Unit, which implements the (Inv)SubBytes, (Inv)MixColumns, and RoundXOR transformations, has slightly more area than a 16-bit x 16-bit multiplier implemented in the same technology. The majority of the area is due to the (Inv)SubBytes lookup-tables. The Sandblaster DSP has a cycle time of 1.67ns and up to four execution cycles. Based on Table 3, our design will have to be pipelined, but only requires two stages and meets the latency requirements of the microarchitecture. The shuffle unit timing depends purely on routing delay and Table 3 shows that it does not cause any timing issues.

4.1 Performance

As shown in Figure 9, the result of our proposed design shows a dramatic improvement over previous optimized software implementations of AES across all key lengths. In this figure, ARM BC and ARM V2 are based on results for the StrongARM SA-1110 processor given in [2]. The VEX architecture uses the configuration shown in Table 2. The speedups, shown relative to a Strong-ARM SA-1110 processor, assume a single thread executing AES at a common clock rate. The C code on the Sandblaster DSP barely outperforms the ARM without ISA extensions. With our new instructions, speedup ranges from 3.75 to over 10 compared to the reference ARM BC code.

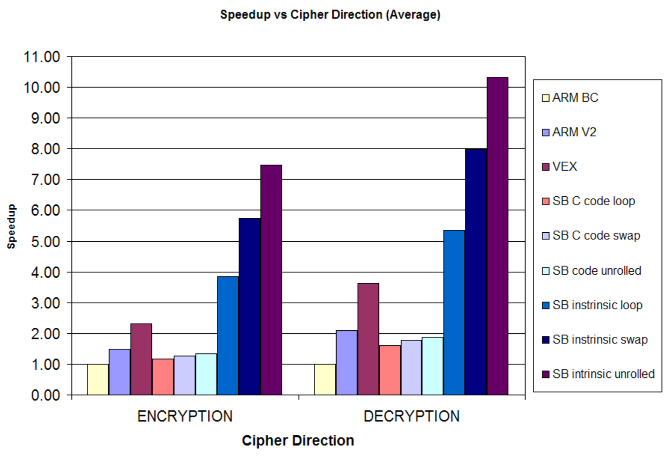

Figure 9: Average SpeedUp Across All Ciphers

Version	Cipher Direction	SpeedUp
SB C code loop	Encryption	3.33
SB C code loop	Decryption	3.33
SB C code swap	Encryption	4.52
SB C code swap	Decryption	4.52
SB C code unrolled	Encryption	5.54
SB C code unrolled	Decryption	5.52

Table 4: Average Speed Up Across All Ciphers

Table 4 shows the speedup due to ISA extensions for encryption and decryption solely on the SB3010. These results show the speedup after removing the round function and replacing it with our new instructions. The speedup takes into account setting up loop parameters (where needed), loading new key data each round, and all system-specific instructions for entering and exiting the main loop. Code using conventional loops ran the slowest as addititonal time is spent on branching and performaning copy operations. Using the optimized swap method improves performance by an average of 9.6% across all cipher lengths for our test C code from [9]. With our instructions the same optimization gains us an average performance improvement of 49.1%. To extract more performance, the target system needs to employ efficient coding techniques and compiler optimizations when using our extensions. When unrolling the entire loop, we obtained the best performance both running standard C code and the same code with our instructions. No swap optimization is used when unrolling a loop as copying pointer data is not needed. Table 5 shows the decrease in code size for each version of AES code with and without the new extensions. Even with unrolling the entire loop, using the new proposed instructions results in a smaller footprint than using the swap optimization with a loop for native C code.

As stated in Section 3.4, we created a light-weight version of AES C code using instrinsics to determine how efficiently the compiler maps intrinsics. Based on viewing the assembler output, we noticed that the compiler produces extra instructions. These extra instructions pro-

	Encryption		Decryption	
Version	C code	Intrinsics	C code	Intrinsics
SB C code loop	2008	760	2016	760
SB C code swap	1704	808	1704	808
SB C code unrolled	7768	1408	7800	1416

Table 5: Code Size in bytes for AES-128 on the SB3010

vided memory consistency (st a, ld a), but also reduced performance by 40 to 50%. After removing these memory consistent instructions, we verified the data output of our custom AES code to ensure the removal of these instructions did not affect the output. Therefore, to fully benefit from these new instructions, the compiler must be able to generate optimized code with these new instructions or some hand modifications to the compiler generated assembly code must be performed.

4.2 Comparison to Related Work–ASICs

Other approaches have been applied to the AES algorithm to achieve high performance. [12] presents an ASIC capable of achieving optical link throughput speeds of 30Gbps to 70Gbps. While the design is efficient, the area requirements for the design are not practical for use with typical software-defined radios. Likewise, key scheduling would mostly be moved to software as performance for key generation becomes less important as more data is encrypted.

[24] presents a small, area-efficient AES encryption engine. SDRs may be paired with ASICs, but typically the ASIC state is not visible to the processor meaning an additonal API needs to be supported by the compiler to allow access to the ASIC. Depending on the particular SDR, the ASIC design may not scale well or fit into the datapath of the DSP. The latency (7.62ns) of this design may also not be suitable for multithread DSPs.

4.3 Comparison to Related Work–ISA Extensions

The design in [8] provides RISC-style ISA extensions. While these extensions are similar to ours, there are some important differences. First, their table sizes range from 8K to 32K depending on the wordsize of the machine. 8K is feasible, but 32K in the datapath may not be practical for an embedded SDR processor. Our design uses 8K for an equivalent 128-bit word size processor. With their architecture, each round takes twelve instructions while ours takes only two.

[17] uses extended Sboxes to integrate the SubBytes and MixColumns operations into a single instruction for a RISC-like processor. Their pipeline decomposes the Sbox function into its components to support both encryption and decryption in the same pipeline. However, they require 36 instructions each round. This would equate to 36 cycles on our SDR processor if these extensions were put inside of the integer unit (the type of unit the extensions were designed for). This corresponds to approximately 360 cycles for encryption/decryption using a 128-bit key. Based

on the dymanics of our platform, this produces around 27Mbps which falls below our design goal. This paper states it can hit 640Mbps, but that is with 1-GHz, 2-way super scalar MIPS processor. The power required for 1-GHZ operation would no longer keep the SB3010 a low-power device.

[26] demonstrates various extensions to help accelerate AES processing. These extensions range from simple byte-level Sbox support to word-size hardare support for byte rotations, SubBytes, and (Inv)MixColumns transformations. These extensions are optimized for 32-bit processors. Likewise, some extensions work better depending on if the state is stored in column-oriented or row-oriented packed format. Our extensions are better suited for taking advantage of the parallel nature of AES. Also, [26] may require bytes to be placed in a specific manner to use some of the extensions efficiently. Our extensions do not require any specialized byte-level placement, except the state is placed in column-oriented packed format. Lastly, in [26] some instuctions require an intermediate state not visible to the programmer, while ours does not add any additional non-visible state.

5 Conclusions and Future Work

Most current cellular systems provide data rates of 2Mbps or less. Next generation cellular networks will have data rates of at least 50Mbps. Even efficient software implementations of AES cannot achieve the necessary throughput for encryption and decryption on embedded processors. Therefore, to achieve these data rates for consumer devices, a better solution must be developed. Our proposed ISA extensions provide the needed throughput while maintaining the current microarchitecture pipeline of the SB3010 platform.

Our next goal is to reduce the area of the sboxes as they make up most of the area. Work proposed in [17] suggests to use an inverse table as opposed to a sbox table. This approach still requires the same number of instructions, but may reduce the size of the lookup table from 512 bytes to 256 bytes while adding negligible area for (inverse)affine transform boxes. Also, since we have not used all of the execution stages available in our target architecture, a compact implemtation [4, 24] of the sbox may give us a smaller area, but have a latency that can still match the microarchitecture. Finally, we plan to investigate, if any, redudant instructions (st a, ld a) can be identified and removed from the three different coding styles (loops, no loops, swap). In the realm of embedded computing, hand-optimization is normal, so doing such optimizations to obtain more performance is quite practical. We believe that 100 Mbps and higher is possible by carefully specifying our instrinsics in a more compiler-friendly manner or hand-optimizing assembly code to get the added performance benefits.

REFERENCES

[1] ARM. Product backgrounder. Technical report, Jan 2005. http://www.arm.com/miscPDFs/3823.pdf.

[2] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. Efficient AES implementations for ARM based platforms. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 841–845, New York, NY, USA, 2004. ACM Press.

[3] A. Brawerman and J. A. Copeland. Towards a fraud-prevention framework for software defined mobile devices. *EURASIP Journal on Wireless Communications and Networking*, 5(3):401–412, August 2005.

[4] D. Canright. A very compact s-box for AES. *Cryptographic Hardware and Embedded Systems*, pages 441–455, 2005.

[5] Nortel R&D Community. Focus on broadband wireless access. *Nortel Technical Journal*, 2, Jul 2005. http://www.nortel.com/corporate/news/collateral/ntj2.pdf.

[6] P. Faraboschi, J. Fisher, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[7] Joseph A. Fischer, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, compilers, and Tools*. Elsevier, 2005.

[8] A. Fiskiran and R. Lee. Fast parallel table lookups to accelerate symmetric-key cryptography. *Proceedings of the International Conference on Information Technology: Coding and Computing*, pages 526–531, Apr 2005.

[9] Brian Gladman. AES and combined encryption/authentication modes. http://fp.gladman.plus.com/AES/.

[10] J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis. Sandbridge software tools. In *5th Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 3553, pages 269–278, Jul 2005.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings IEEE International Workshop on Workload Characterization*, pages 3–14, Washington, DC, USA, 2001.

[12] Alireza Hodjat and Ingrid Verbauwhede. Minimum area cost for a 30 to 70 gbits/s AES processor. *IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design*, 2004.

[13] Hua Li and Zac Firggstad. An efficient architecture for the AES MixColumns operation. *Department of Mathematics and Computer Science University of Lethbridge*.

[14] S. Mamidi, E. R. Blem, M. J. Schulte, J. Glossner, D. Iancu, A. Iancu, M. Moudgill, and S. Jinturkar. Instruction set extensions for software defined radio on a multithreaded processor. In *Proceedings of the ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 266–273, September 2005.

[15] S. Mamidi, E. R. Blem, M. J. Schulte, D. Iancu J. Glossner, A. Iancu, M. Moudgill, and S. Jinturkar. Instruction set extensions for software defined radio. *Accepted for publication in the Internatinal Journal of Embedded Systems*, 2007.

[16] M. Mehta, N. Drew, G. Vardoulias, N. Greco, and C. Niedermeie. Reconfigurable terminals: An overview of architectural solutions. *IEEE Communications Magazine*, 39:146–155, August 2001.

[17] K. Nadehara, M. Ikekawa, and I. Kuroda. Extended instructions for the AES cryptography and their efficient implementation. *IEEE Workshop on Signal Processing Systems*, pages 152–157, Oct 2004.

[18] University of Maryland. Technical report. http://www.cs.umd.edu/class/fall2001/cmsc411/proj01 /arm/armchip.html.

[19] National Institute of Standards and Technology. Announcing the advanced encryption standard. Technical report. http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[20] National Institute of Standards and Technology. Establishing wireless robust security networks: A guide to IEEE 802.11i. Technical report.

[21] Hewlett Packard. VEX toolchain. http://www.hpl.hp.com/downloads/vex/.

[22] Qualcomm. HSDPA for improved downlink data transfer. Technical report, Oct 2004. http://www.cdmatech.com/download_library/pdf/ hsdpa_downlink_wp_12–04.pdf.

[23] S. Ravi, A. Raghunathan, and N. Potlapally. Securing wireless data: System architecture challenges. In *International Symposium System Synthesis*, pages 195–200, October 2002.

[24] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with s-box optimization. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 239–254, London, UK, 2001. Springer-Verlag.

[25] M. J. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *Journal of VLSI Signal Processing*, 41, 2005.

[26] Stefan Tillich. Instruction set extensions for efficient AES implementation on 32-bit processors. *Cryptographic Hardware and Embedded Systems*, pages 270–284, 2006.

[27] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.