Instruction Set Extensions for Software Defined Radio on a Multithreaded Processor

Suman Mamidi, Emily R. Blem, Michael J. Schulte
University of Wisconsin-Madison
1415 Engineering Dr.
Madison, WI 53706, USA
mamidi@cae.wisc.edu

John Glossner, Daniel Iancu, Andrei Iancu, Mayan Moudgill, Sanjay Jinturkar
Sandbridge Technologies
1 North Lexington Avenue, 10th Floor
White Plains, NY 10601, USA
jglossner@sandbridgetech.com

ABSTRACT

Software defined radios, which provide a programmable solution for implementing the physical layer processing of multiple communication standards, are widely recognized as one of the most important new technologies for wireless communication systems. Emerging communication standards, however, require tremendous processing capabilities to perform high-bandwidth physical-layer processing in real time. In this paper, we present instruction set extensions for several important communication algorithms including convolutional encoding, Viterbi decoding, turbo decoding, and Reed-Solomon encoding and decoding. The performance benefits of these extensions are evaluated using a supercomputer class vectorizing compiler and the Sandblaster low-power multithreaded processor for software defined radio. The proposed instruction set extensions provide significant performance improvements, while maintaining a high degree of programmability.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Real-time and embedded systems*

General Terms

Design and Performance

Keywords

Instruction set extensions, digital signal processor, multithreading, software defined radio, forward error correction, Reed-Solomon coding, Viterbi decoding, turbo decoding, and convolutional encoding.

1. INTRODUCTION

Traditional wireless communication systems have typically been implemented using custom hardware solutions [7]. Chip rate, symbol rate, and bit rate co-processors are often coordinated by programmable digital signal processors (DSPs), but the DSPs do not typically participate in physical layer processing. Even when supporting a single communication system, the hardware development cycle is onerous and often requires multiple chip redesigns late in the certification process. When multiple communication systems must simultaneously be supported, silicon area and design validation are major inhibitors to commercial success.

Software Defined Radios (SDRs), which provide a programmable platform for implementing the physical layer processing of multiple communication standards, are widely recognized as one of the most important new technologies for wireless communication systems [13]. A SDR platform capable of dynamically reconfiguring itself to implement multiple communication standards enables elegant reuse of silicon area and dramatically reduces time to market through software modifications, instead of time-consuming hardware redesigns. SDRs also allow wireless devices to be reconfigured through software updates to implement emerging wireless communication standards, thereby decreasing product development time.

Although SDRs offer many benefits, they require extremely high-performance processors to implement real-time physical-layer processing for high-bandwidth communication systems. For example, as noted in [6], physical layer processing for a WCDMA uplink receiver requires more than 5.5 billion operations per second. Consequently, innovative architectural techniques are needed to provide high-performance, low-power programmable processors for SDR.

Sandbridge technologies has developed the SB3010, a SDR baseband processor that performs physical layer processing in software for a variety of communication systems including 2Mbps WCDMA, 11Mbps IEEE802.11b, GSM/GPRS, and GPS [14]. The SB3010 consists of four multithreaded Sandblaster Processor cores, an ARM9 applications processor, on-chip instruction caches and data memories, a programmable RF interface, and several peripheral interfaces. Each Sandblaster Processor core provides over two billion multiply-accumulate (MAC) operations per second and features powerful compound instructions, Single Instruction Multiple Data (SIMD) vector operations, and simultaneous execution of up to eight hardware threads.

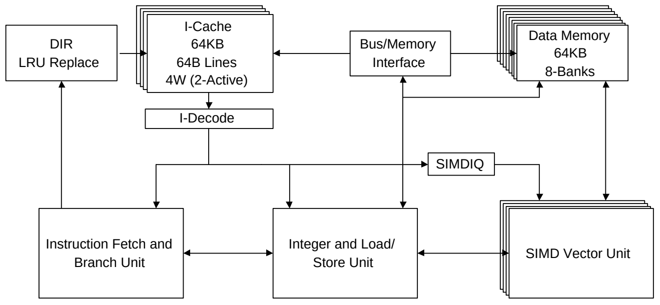

Figure 1: Sandblaster processor.

```
L0: lvu      %vr0, %r3, 8
||  vmulreds %ac0, %vr0,%vr0,%ac0
||  loop     %lc0, L0
```

Figure 2: A 64-bit compound instruction.

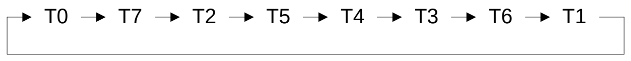

Figure 3: Token Triggered Threading.

In this paper, we present instruction set extensions for software defined radio and evaluate the performance benefits of these extensions using a supercomputer-class vectorizing compiler and the Sandblaster multithreaded processor. These instruction set extensions significantly improve the performance of compute-intensive communication algorithms including convolutional encoding, Viterbi decoding, turbo decoding, and Reed-Solomon encoding and decoding. Due to the importance of these algorithms in SDR, they are required on radios that implement the Department of Defense's Joint Tactical Radio System Software Communication Architecture (JTRS SCA) with Specialized Hardware Components [10]. Improving the performance of these algorithms enables simultaneous processing of multiple communication algorithms at very high data rates.

This paper is organized as follows: Section 2 gives an overview of the Sandblaster Multithreaded Processor. Section 3 describes our methodology for determining and evaluating new operations. Section 4 presents the algorithms, instruction set extensions, and potential hardware implementations. Section 5 summarizes the speedups due to the proposed instruction set extensions, along with area and delay estimates for potential hardware implementations. Section 6 discusses related work on instruction set extensions for wireless communication systems. Section 7 summarizes the paper. The main contributions of this paper are (a) the introduction of several instruction set extensions and hardware designs for software defined radio, and (b) the evaluation of the performance benefits from the instruction set extensions on a compound-instruction, multithreaded processor with SIMD vector operations.

2. SANDBLASTER PROCESSOR

The Sandblaster Processor utilizes a unique combination of techniques including hardware support for multiple threads, compound instructions, SIMD vector operations, and instruction set support for Java code [14]. The Sandblaster Processor provides substantial parallelism and throughput for high-performance SDR, while maintaining fast interrupt response, high-level language programmability, and low power dissipation. Figure 1 shows a block diagram of the Sandblaster Processor. The processor is partitioned into three units; an instruction fetch and branch unit, an integer and load/store unit, and a SIMD vector unit.

The Sandblaster Processor conserves program memory through the use of powerful 64-bit compound instructions that issue up to three compound operations each cycle. To simplify instruction decoding and reduce hardware requirements, certain operations are not specifiable within the same compound instruction. Figure 2 shows a 64-bit compound instruction with three compound operations. This instruction implements the inner loop of a vector sum-of-squares computation. The first compound operation, `lvu`, loads the vector register, `vr0`, with four 16-bit elements and updates the address pointer, `r3`, to the next element. The `vmulreds` operation reads four 16-bit elements from `vr0`, multiplies each element by itself, saturates each product, adds all four saturated products plus a 40-bit accumulator register, `ac0`, with saturation after each addition, and stores the result back in `ac0`. The `loop` operation decrements the loop count and branches to L0 if the result is not zero. Since our proposed instruction set extensions are implemented in the SIMD vector unit, they can be part of compound instructions that also include branch, integer, or load/store operations.

The Sandblaster processor uses a unique form of interleaved multithreading, called Token Triggered Threading (T^3), which is illustrated in Figure 3. With T^3, all threads can be simultaneously executing instructions, but only one thread may issue an instruction on a cycle boundary [14]. This constraint is also imposed on round-robin threading. What distinguishes T^3 is that each clock cycle a token indicates the subsequent thread that is to issue an instruction. Thread ordering may be sequential (e.g. round robin), even/odd, or based on other communication patterns. Compared to Simultaneous Multithreading (SMT) [4], T^3 has much less hardware complexity and power dissipation since the method for selecting threads is simplified, only a single compound instruction issues each clock cycle, and dependency checking hardware is eliminated. Compared to traditional interleaved multithreading, it provides higher performance through compound instructions, SIMD vector operations, and greater flexibility in scheduling threads. The current implementation of the Sandblaster Processor supports up to eight simultaneous threads of execution per processor core.

Figure 4 shows a block diagram of the SIMD vector processing unit (VPU), which consists of four vector processing elements (VPEs), a shuffle unit, a reduction unit, and an accumulator register file. The four VPEs perform arithmetic and logic operations in SIMD fashion on 16-bit, 32-bit, and 40-bit fixed-point data types. High-speed 64-bit data busses allow each VPE to load or store 16 bits of data each cycle in SIMD fashion. Support for SIMD execution reduces code size, as well as power consumption from fetching and decoding instructions, since multiple sets of data elements are processed with a single instruction.

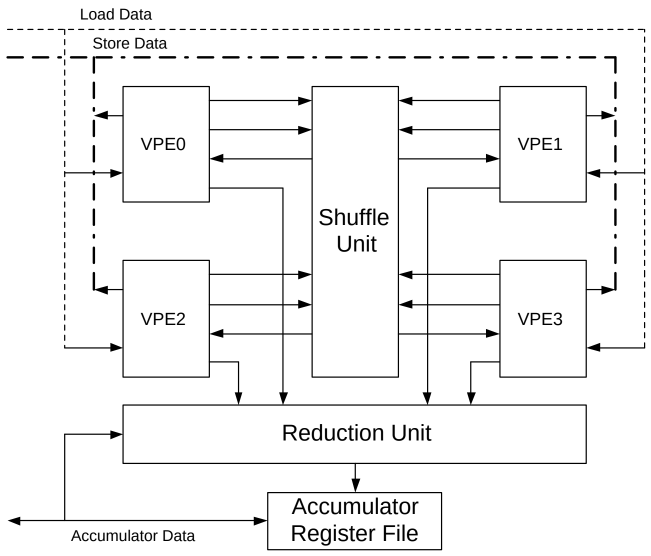

Figure 4: SIMD vector processing unit

Most vector operations have eight pipeline stages once they are fetched; Instruction Decode, Register Read, Execute1, Execute2, Execute3, Execute4, Transfer, and Write Back. Since there are eight cycles between when consecutive instructions issue from the same thread, results from one instruction in a thread are guaranteed to be written back to the register file by the time the next instruction from the same thread is ready to read them. Thus, the long pipeline latency of the VPEs is effectively hidden, and no data dependency checking or bypass hardware is needed. Consequently, our instruction set extensions can have up to four execution stages without significantly impacting the processor pipeline. This allows the extensions to be fairly complex since their latency is hidden by the multithreaded pipeline.

3. DESIGN METHODOLOGY

Our basic methodolgy for selecting and analyzing instruction set extensions for software defined radio consists of the following steps:

1. Select representative benchmark applications for software defined radio. In this paper, we focus on Forward Error Correction (FEC) algorithms due to their widespread use in wireless communcication systems and their high computational requirements. For example, as noted in [6], a Viterbi decoder for an 802.11a wireless LAN receiver with 64-State Quadrature Amplitude Modulation requires over 6.7 billion operations per second. Since FEC coding is performed on all transmitted and received data, multiple threads may be utilized for FEC coding.
2. Profile each application to determine portions of the code that use the most cycles. Each application, which is written in C, is compiled with full compiler optimizations including vectorization, loop unrolling, software pipelining, code motion, function inlining, and peephole optimizations [18]. Portions of the code that use the most cycles are then identified using the Sandbridge Software Tools [8].
3. Determine suitable operations for speeding up portions of the code that use the most cycles. In some cases, it is useful to introduce multiple operations to implement a particular algorithm. Consistent with existing Sandblaster vector operations, our new operations have up to three vector source operands and one vector destination operand. As described in Section 2, the new operations can have up to four execution cycle without significantly impacting the processor pipeline. The new operations are designed to be flexible enough to support different implementations of the target applications.
4. Rewrite each application with the new operations provided as intrinsics and rerun the simulations to determine the new cycle count for each application. This process is simplified by the Sandblaster architecture Description Language (SaDL) [8]. SaDL provides an abstraction of the Sandblaster architecture and implementation into a single database, which guides the generation and optimization of the Sandbridge tool chain. With this approach, the application uses the original C code with minor replacements of portions of the C code with intrinsics for the new operations. The compiler then optimizes and schedules the entire application, which lets the new operations be included in compound instructions and undergo the same optimizations as other operations. Using our technique of semantic analysis described in [18], the Sandblaster compiler can be extended to automatically generate the new operations from the original C code, without the need to rewrite the code to use intrinsics.
5. Implement potential hardware designs for the new operations in Verilog. Each design is simulated to ensure correct performance and then synthesized using Synopsys Design Compiler and LSI Logic's glfxp 0.11 micron CMOS standard cell library to obtain area and delay estimates.

4. ALGORITHMS, INSTRUCTION SET EXTENSIONS, AND HARDWARE DESIGNS

In this section, we (a) briefly describe various Forward Error Correction (FEC) algorithms that are frequently used in wireless communcation systems, (b) provide instruction set extensions to speedup these algorithms, and (c) suggest hardware designs to implement the proposed instruction set extensions. The FEC encoding algorithms add redundancy to the transmitted signal to support error detection and correction by an FEC decoder at the receiver. Further details on FEC algorithms are given in [16].

4.1 Convolutional Encoder

Convolutional encoding, an FEC encoding technique used in GSM/GPRS and other communication standards, is well suited for trasmitting data over noisy channels [3]. Convolutional encoders are often implemented in hardware using a shift register and combinational logic that performs modulo-2 additions (i.e., exclusive-or (XOR) operations).

Convolutional encoders have three main parameters; Rate, Constraint Length, and Taps. The serial convolutional encoder shown in Figure 5 is called a 1/2 Rate, 3,7,5 Encoder. The 1/2 Rate indicates that the encoder produces two output bits for each input bit. In this example, the encoder

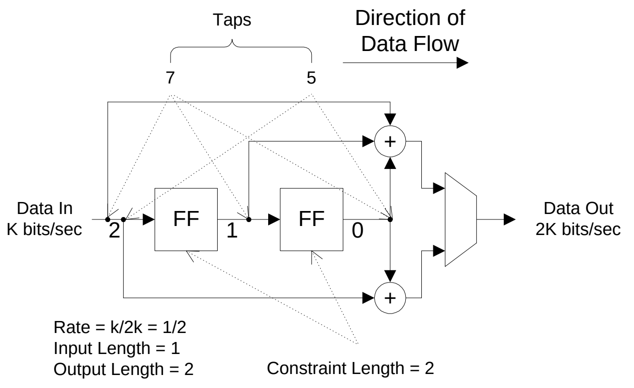

Figure 5: Sample convolutional encoder.

has an Input Length of 1 and an Ouput Length of 2. The Constraint Length for this encoder is 3, since the encoder examines three bits at a time. The Taps, 7 and 5, represent the code generator polynomials, which when read in binary (111_2 and 101_2) correspond to the shift register connections to the upper and the lower module-2 adders, respectively. Although the convolutional encoder shown in Figure 5 has small hardware requirements its serial implementation and lack of programmability make it ill-suited for software defined radio.

A convolutional encoder is fully-programmable if all the parameters defined above are programmable. Traditional software implementations of fully programmable convolutional encoders are computationally expensive since they have several shifts, logic operations, and conditional branches. Our instruction set exentensions provide two new operations for performing fully-programmable convolutional encoding; update_shifter and convolve.

The update_shifter operation is a SIMD vector operation that corresponds to right shifting a specified number of data bits into a shift register, where both the number of data bits shifted in and the size of the shift register can vary. This operation uses three input operands, Data In, (Constraint Length, Input Length)[1] and Current State, to produce one output operand, Next State. The update_shifter operation right shifts Data In of length Input Length into Current State of length Constraint Length to produce the Next State.

Figure 6 shows a hardware design that implements the update_shifter operations, when Input Length and Constraint Length can vary from one to eight bits. The design consists of three barrel shifters. The first aligns the Current State based on the Constraint Length. The second inserts the Data In bits based on the Input Length. The third right shifts based on the Constraint Length to produce the Next State. Larger maximum Input Lengths and Constraint Lengths can be supported by increasing the size of the barrel shifters.

The convolve operation is a SIMD vector operation that corresponds to taking the appropriate bits from the shift register and XORing them to generate the encoded output bits. It takes two input operands, Current State and Taps, and generates one output operand, Data Out. The Taps

[1]To limit the number of input operands to three, Constraint Length and Input Length, which do not vary when implementing a particular encoder, are stored in the same register.

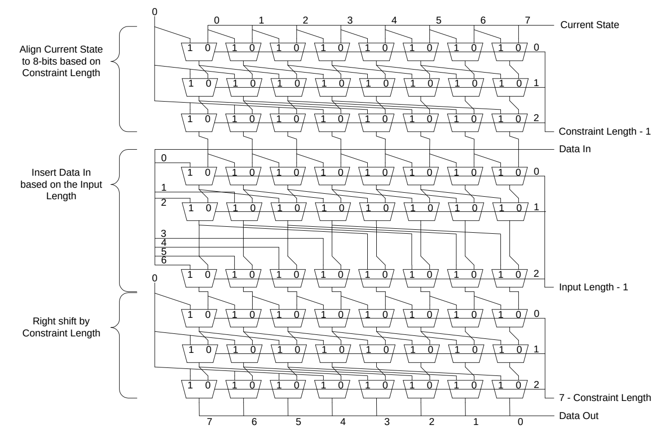
Align Current State to 8-bits based on Constraint Length
Insert Data In based on the Input Length
Right shift by Constraint Length
Current State
Constraint Length - 1
Data In
Input Length - 1
7 - Constraint Length
Data Out

Figure 6: Hardware for update_shifter operations.

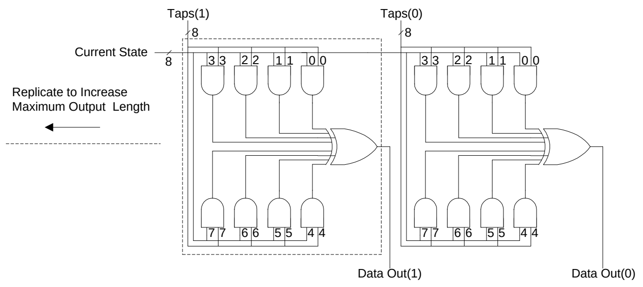

Figure 7: Hardware for convolve operations.

operand determines which bits of the Current State connect to the XOR gates that produce Data Out. Its length is equal to the Constraint Length times the Output Length.

Figure 7 shows a hardware design that implements the convolve operation when the maximum Output Length is two and the maximum Constraint Length is eight. The design consists of Constraint Length AND gates and one XOR gate per Data Out bit, where the inputs to each XOR gate are controlled by bits from the Tap operand. If bit n from the Tap operand is one, then the corresponding bit from Current State influences the output of the XOR gate. The hardware can support larger maximum Output Lengths by replicating the block shown in the dotted box and can support larger maximum Constraint Lengths by increasing the number of AND gates and the fan-in of each XOR gate.

4.2 Viterbi Decoder

Convolutional encoders often work in conjunction with Viterbi decoders, which have fixed decoding time and are well suited for hardware implementation [9]. The main steps for Viterbi decoding are shown in Figure 8. The text in parentheses in Figure 8 shows the percentage of the total execution time spent in each step in the original software implementation on the Sandblaster Processor. The Add-Compare-Select (ACS) function accounts for rougly 87% of the overall execution time for the Viterbi decoder.

Three new operations are introduced to speed up the ACS function; acs_select_metric, acs_set_flag, and select_state. The acs_select_metric operation is a vector operation that takes three input operands, MetricIn, PathMetric1, and PathMetric2, and produces one ouput operand, MetricOut. Each

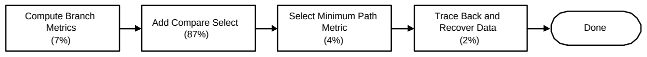
Compute Branch Metrics (7%)
Add Compare Select (87%)
Select Minimum Path Metric (4%)
Trace Back and Recover Data (2%)
Done

Figure 8: Main steps for Viterbi decoding.

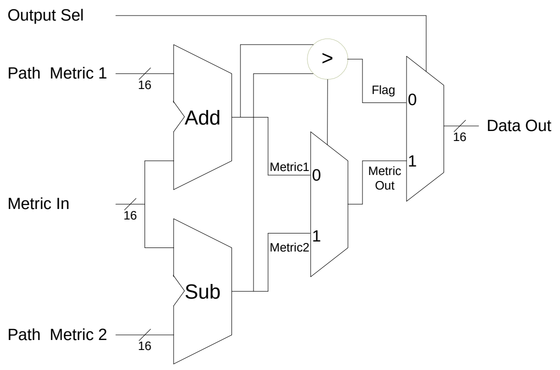

Figure 9: Hardware for add-compare-select operations.

VPE performs the following operation:

```
Metric1 = PathMetric1 - MetricIn;
Metric2 = PathMetric2 - MetrinIn;
if (Metric1 > Metric2)
   MetricOut = Metric1;
else
   MetricOut = Metric2;
```

The acs_set_flag operation is similar to the acs_select_metric operation, except that rather than an assignment to MetricOut, it returns a Flag operand that indicates the result of the comparison. Each VPE performs the following operation:

```
Metric1 = PathMetric1 - MetricIn;
Metric2 = PathMetric2 - MetricIn;
if (Metric1 > Metric2)
   Flag = 0;
else
   Flag = 1;
```

The Flag operand set by the acs_set_flag operation is read by the vector operation select_state, which uses three input operands, Flag, State1, and State2, to produce one output operand, StateOut. Each VPE performs the following operation:

```
if (Flag == 0)
   StateOut= State1;
else
   StateOut = State2;
```

Figure 9 shows hardware for the acs_select_metric and acs_set_flag operations. The input operands, MetricIn, PathMetric1, and PathMetric2, are read from the vector register file. The processor's control logic sets the Output Sel signal based on the operation being performed.

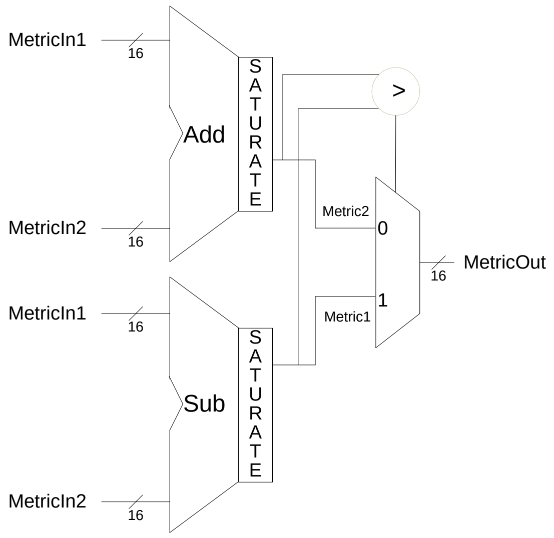

Figure 10: Hardware for add-saturate-compare-select operations.

4.3 Turbo Decoder

Due to turbo coding's outstanding error correction capabilities, it is widely used in third generation wireless communication systems. Turbo decoding consists of two steps; forward recursion and backward recursion [2]. During forward recursion, branch metrics are computed and forward metrics are updated for each trellis transition. The decoder then performs backward recursion to generate soft decisions. Turbo decoding is very compute intensive and about 70% of the execution time of the original code is spent performing add-saturate-compare-select (ASCS) operations.

To facilitate turbo decoding, we add one new operation, ascs_turbo. This is a vector operation that takes in two 16-bit input operands, MetricIn1 and MetricIn2, and computes a 16-bit output operand MetricOut. Each VPE performs the following operation:

```
Metric1 = Saturate(MetricIn1 - MetricIn2)
Metric2 = Saturate(MetricIn1 + MetricIn2)
if (Metric1 > Metric2)
   MetricOut = Metric1
else
   MetricOut = Metric2
```

As shown in Figure 10, the hardware design to implement this operation is straight forward. The adder and the subtractor saturate their outputs to either -2^{15} or $+2^{15}-1$ based on the sign of the result. The greater of Metric1 or Metric2 is selected as MetricOut. Due to similarities between the the acs_select_metric and acs_set_flag operations for Viterbi coding and the ascs_turbo operation for turbo decoding, all three operations can share hardware.

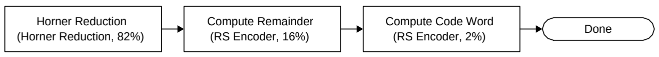

Figure 11: Main steps for Reed-Solomon encoder.

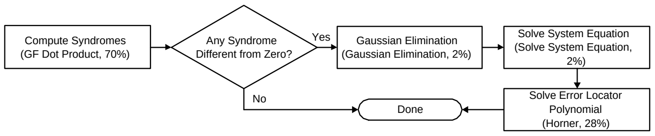
Compute Syndromes
(GF Dot Product, 70%)
Any Syndrome Different from Zero?
Yes
Gaussian Elimination
(Gaussian Elimination, 2%)
Solve System Equation
(Solve System Equation, 2%)
Solve Error Locator Polynomial
(Horner, 28%)
No
Done

Figure 12: Main steps for Reed-Solomon decoder.

4.4 Reed-Solomon Encoding/Decoding

Reed-Solomon coding is widely used in communication standards such as Digital Video Broadcast (DVB) and the IEEE802.16 WirelessMAN Standard. Reed-Solomon codes are referred to as RS(N,K) codes with M-bit symbols. This means the encoder takes K data symbols of M bits each and adds $N-K$ parity symbols to make an N symbol codeword. On the receiving end, the Reed-Solomon decoder can correct up to T symbols that contain errors in the codewords, where

$$T = \frac{(N-K)}{2} \tag{1}$$

Reed-Solomon codes are particularly well suited to correcting burst errors, in which a continuous sequence of bits is received with errors [5], [11].

The steps for Reed-Solomon encoding and decoding are shown in Figures 11 and 12, respectively. The text in the parenthesis indicates the function name that implements the particular step and the percentage of the total execution time spent by each function in the original software implementation on the Sandblaster Processor. For the decoder, the percentages assume a worst case scenario in which at least one error is present in the received codeword, which results in non-zero syndromes. In the more common case, in which there are no errors, 98% of the execution time is spent on computing the syndromes. Most of the time spent in Reed-Solomon encoding and decoding involves vector operations (e.g., dot products) in the Galois field GF(2^m). In GF(2^m), addition is equivalent to the bitwise XOR of two m-bit numbers.

Our instruction set extensions provide vector operations for Galois field (GF) arithmetic, which is fundamental to Reed-Solomon encoding and decoding. We investigate the performance benefits of four GF arithmetic vector operations; gfmul, gfmac, gfmul2, and gfmac2 [19]. Table 1 summarizes these operations and their functionality. In this table, $A \otimes B$ denotes GF multiplication (GFMUL) of A with B to generate a 15-bit intermediate product, C. The 15-bit intermediate product is reduced using an irreducible polynomial, P, of length, $L+1$, to generate an 8-bit product, D. The operation $(A \otimes B) \oplus Acc$ denotes a GF multipliy-accumulate (GFMAC) operation, in which Acc is added to $A \otimes B$ using GF addition. The Galois field operations are implemented by adding parallel GFMUL or GFMAC units to the SIMD Vector Processing Unit. The gfmul/gfmac instructions use one GFMUL/GFMAC unit per VPE and the gfmul2/gfmac2 instructions use two GFMUL/GFMAC units per VPE.

Figure 13 shows a hardware design for implementing the gfmul/gfmac instructions in each VPE. The GF Multiplier consists of an 8x8 array of AND gates which generate the partial products, followed by a tree of 64 XOR gates, which sum the partial products using Galois field addition to produce a 15-bit intermediate product. The 15-bit Left Shifter and 8-bit Right Shifter allow the length of the Galois Field to vary from one to eight bits. The GF Polynomail Reduction Unit reduces the intermediate product from 15 bits to 8 bits. It consists of seven stages, where each stage has eight parallel AND gates followed by eight parallel XOR gates. The 8-bit GF Adder consists of eight parallel XOR gates. The A, B, and Acc input operands come from the vector register file. The values for P and L, which do not change for a particular implementation of Reed-Solomon coding are loaded into a special purpose register at the start of the algorihm. Mac is set by the processor's control logic based on the operation being performed. The GFMUL unit is identical to the GFMAC unit, except it does not contain the GF Adder or 2-to-1 Multiplexor.

Table 1: Proposed Galois field operations

Instruction	Parallel Operations	Operations
gfmul	4	$Data\ Out = A \otimes B$
gfmac	4	$Data\ Out = (A \otimes B) \oplus Acc$
gfmul2	8	$Data\ Out = A \otimes B$
gfmac2	8	$Data\ Out = (A \otimes B) \oplus Acc$

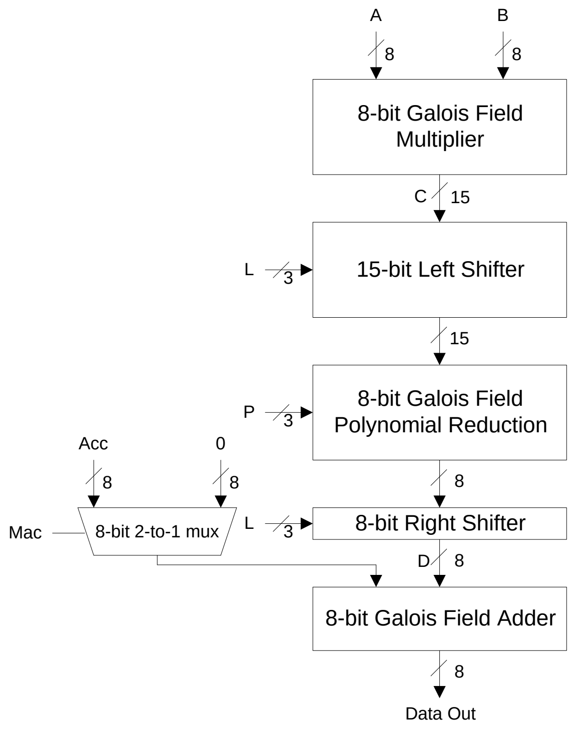

Figure 13: Galois field multiply-accumulate unit.

5. EXPERIMENTAL RESULTS

In this section, we discuss the speedup of the applications due to the proposed instruction set extensions and the bandwidth provided by each FEC algorithm when run on a single Sandblaster Processor core. We also provide estimates of the area and delay for the hardware designs that

Table 2: Speedup from Galois field operations

	gfmul	gfmac	gfmul2	gfmac2
RS Decoder	8.4	12.5	11.5	12.7
RS Encoder	1.25	1.25	1.25	1.25

implement these extensions, using the design methodology presented in Section 3.

The original implementation of convolutional encoding on the Sandblaster Processor spends 65,061 cycles encoding a 512-bit packet with a Constraint Length of 5. This translates to a throughput of 4.7 Mbits/sec at 600MHz. The implementation using the instruction set extensions spends 4,500 cycles to encode the same packet, resulting in a throughput of 68MBits/sec at 600MHz and a speedup of 14.5. Using instruction set extensions usually results in reduction of code size. In the case of the convolutional encoder, the code that uses instruction set extensions is 74.6% the size of the original implementation that does not use the instruction set extensions. The hardware design for the update_shifter operation has an area of $5,836\mu^2$ per VPE with a worst case delay of $0.34ns$. The design for the convolve operation has only $802\mu^2$ per VPE with a worst case delay of 0.18ns.

The benchmark for the Viterbi decoder implements a soft decision Viterbi decoder with an input packet of 344 6-bit values, each of which represents a pair of encoded bits (i.e. the input bit stream is produced by a 1/2 rate convolutional encoder which generates a pair of output bits for each input bit). The original implementation used 139,629 cycles, while the implementation with instruction set extensions used 39,000 cycles for a decoding throughput in excess of 16Mbits/sec at 600MHz and a speedup of 4.7. In terms of code size, the implementation that uses the instruction set extensions is 20% larger than the original implementation due to additional instructions needed to unroll loops and support vectorization. The three instruction set extensions have a combined area of $10,900\mu^2$ per VPE. The worst case delay is 0.74ns, which is seen on the acs_select_metric path.

The original turbo decoder used 479,989,032 cycles to decode 2,896 16-bit data samples. Adding the ascs_turbo operation speeds up turbo decoding by a factor of 2.35 and decreases the number of cycles used to 204,250,651. This results in a output throughput of over 8.5MBits/sec at 600MHz. The code size is reduced to 72% of the original implementation. The hardware that implements ascs_turbo uses $7,757\mu^2$ for each VPE with a worst delay of 0.79ns.

The Reed-Solomon encoder and decoder are designed for the DVB-T standard, in which the message size is 118 bytes, the codeword size is 204 bytes, and up to eight errors can be corrected [19]. In Table 2, we give the speedup of a Reed-Solomon decoder and encoder for each of the proposed Galois field operations. The designs with gfmac/gfmac2 operations can also implment gfmul/gfmul2 operations, while the designs with gfmul/gfmul2 operations perform gfmac/gfmac2 operations using two operations; a gfmul/gfmul2 operation followed by an XOR operation. With gfmac2, we see a 12.7 speedup in the Reed-Solomon decoder, which reduces the total number of cycles required to decode a 204-byte packet from 88,303 cycles to 6,953 cycles. With the instruction set extensions, the Sandblaster Processor performs worst case Reed-Solomon decoding at over 150Mbits/second at 600MHz. The extensions also result in an enocoder that is just 3% of the size of the original encoder and a decoder that is just 7% of the size of the original decoder. This large reduction in code size occurs because the original implementation uses a 2^{16}-word by 8-bit table to implement Galois field multiplication. Each GFMUL unit has an area of $10,688\mu^2$ and each GFMAC unit has an area of $12,463\mu^2$. The worst case delay for the GFMUL unit is 1.22ns and the worst case delay for the GFMAC unit is 1.32ns.

Table 3 summarizes the proposed instruction set extensions by giving the speedup, total area, and worst case delay for each operation. For reference, the area of a 16-bit by 16-bit multiplier implemented in the same technology and optimized for delay has an area of $24,789\mu^2$ and a worst case delay of 2.2 ns. The proposed instruction set extensions will not increase the processor's cycle time, which is currently 1.67ns, since vector instructions have four execute stages.

6. RELATED WORK

This section describes related work on instruction set extensions for wireless communications. Early research on hardware designs for convolutional encoders includes work in [3], which gives pipelined and parallel hardware implementations for convolutional encoders. More recently, DSP manufacturers including Texas Instruments and Freescale Semiconductor provide bit interleaving and bit de-interleaving instructions to facilitate convolutional encoding [17].

Viterbi decoders have high computatinal complexity and often require hardware support for real-time processing. This can be done either by a hardware accelerator, where part or all of the Viterbi decoder is performed by an on-chip co-processor [20] or by instruction set extensions. Tensilica provides instruction set extensions to implement the Viterbi butterfly for their configurable Xtensa processor and [12] presents hardware support to perform the add-compare-select operation.

Since the standardization of Turbo-codes in 1994, the complexity of their implementation has sparked many research studies. [1] maps the process of turbo decoding to a reconfigurable processor called XiRisc. Several commercial DSP companies provide hardware support for turbo decoding through on-chip hardware accelerators [15].

For high-speed Reed-Solomon encoding and decoding, several hardware implementations are available. Most common hardware implementations are based on Linear Feedback Shift Registers (LPSR) [21]. The authors of [5] propose instruction set extensions for the configurable Xtensa processor that facilitates software implementation on a single-issue processor to achieve high-speed Reed-Solomon decoding.

Unlike related work in this area, this paper presents instruction set extensions and hardware designs on a compound-instruction, multithreaded processor with SIMD vector operations for several FEC algorithms.

7. SUMMARY

In this paper, we analyze important FEC coding algorithms used for software defined radio. We propose new SIMD vector operations for these algorithms to improve their overall performance. We target the new operations to the Sandblaster Processor, find the speedups for the applications due to the new operations, and suggest hardware to implement these operations. Our analysis shows that the proposed instruction set extensions provide significant speedups with modest area requirements.

Table 3: Summary of operation, speedup, area and delay

Application	Operation	Speedup	Total Area (μ^2/VPE)	Delay (ns)
Convolutional Encoder	update_shifter	14.5	6,638	0.34
	convolve			
Viterbi Decoder	acs_select_metric	4.7	10,900	0.74
	acs_set_flag			
	select_state			
Turbo Decoder	ascs_turbo	2.3	7,757	0.79
Reed-Solomon Decoder	gfmul	8.4	10,688	1.22
	gfmul2	11.5	21,376	1.22
	gfmac	12.5	12,463	1.32
	gfmac2	12.7	24,926	1.32

8. REFERENCES

[1] A. La Rosa, L. Lavagno, and C. Passerone. Implementation of a UMTS Turbo Decoder on a Dynamically Reconfigurable Platform. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):100–106, January 2005.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo Codes. In *1993 International Conference on Communications*, pages 1064–1070, 1993.

[3] D. Haccoun and P. Lavoie and Y. Savaria. New Architectures for Fast Convolutional Encoders and Threshold Decoders. *IEEE Journal on Selected Areas in Communications*, 6(3):547–557, April 1998.

[4] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture*, pages 392–403, June 1995.

[5] H. M. Ji. An Optimized Processor for Fast Reed-Solomon Encoding and Decoding. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 3097–3100, 2002.

[6] Y.-H. Huang, H.-P. Ma, M.-L. Liou, and T.-D. Chiueh. A 1.1 G MAC/s Sub-Word-Parallel Digital Signal Processor for Wireless Communication Applications. *IEEE Journal of Solid-State Circuits*, 39(1):169–183, January 2004.

[7] J. Glossner, D. Iancu, J. Lu, E. Hokenek, and M. Moudgill. A Software Defined Communications Baseband Design. *IEEE Communications Magazine*, 41(1):120–128, January 2004.

[8] J. Glossner, S. Dorward, S. Jinturkar, M. Moudgill, E. Hokenek, M. Schulte, and S. Vassiliadis. Sandbridge Software Tools. In *Lecture Notes in Computer Science - Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 3553, July 2005.

[9] J. Hagenauer and P. Hoeher. A Viterbi Algorithm with Soft-decision Outputs and its Applications. In *Global Telecommunications Conference*, pages 1680–1686, 1989.

[10] Joint Tactical Radio System (JTRS) Joint Program Office. *Specialized Hardware Supplement to the Software Communication Architecture (SCA) Specification*, August 2004.

[11] L. Song, K. K. Parhi, I. Kuroda, and T. Nishitani. Hardware/software Codesign of Finite Field Datapath for Low-energy Reed-Solomon Codecs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):160–172, 2000.

[12] M. Hosemann, R. Habendorf, and G. P Fettweis. Hardware-Software Codesign of a 14.4 MBit - 64 state - Viterbi Decoder for an Application-Specific Digital Signal Processor. In *IEEE Workshop on Signal Processing Systems*, pages 45–50, August 2003.

[13] M. Mehta, N. Drew, G. Vardoulias, N. Greco, and C. Niedermeier. Reconfigurable Terminals: An Overview of Architectural Solutions. *IEEE Communications Magazine*, 39(8):146–155, August 2001.

[14] M. Schulte, J. Glossner, S. Mamidi, M. Moudgill, and S. Vassiliadis. A Low-Power Multithreaded Processor for Baseband Communication Systems. *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation, Lecture Notes in Computer Science*, 3133:393–402, July 2004.

[15] J. N. Popovic. Decoding Convolutional and Turbo Codes in 3G Wireless. Available from http://www.ti.com/, 2005.

[16] R. E. Blahut. *Algebraic Codes for Data Transmission.* Cambridge University Press, 2003.

[17] W. Rouwet. Building a Convolutional Encoder Using RCF Technology. Available from http://www.freescale.com/, 2004.

[18] S. Jinturkar, J. Glossner, V. Kotlyar, and M. Moudgill. The Sandblaster Automatic Multithreaded Vectorizing Compiler. In *2004 Global Signal Processing Expo and International Signal Processing Conference*, September 2004.

[19] S. Mamidi, M. Schulte, D. Iancu, A. Iancu, and J. Glossner. Instruction Set Extensions for Reed-Solomon Encoding and Decoding. In *IEEE 16th International Conference on Application-specific Systems, Architectures and Processors*, pages 364–369, 2005.

[20] Texas Instruments. TMS320C6418 Fixed-Point Digital Signal Processor Data Manual. Available from http://www.ti.com/, 2004.

[21] Y. Katayama and S. Morioka. One-Shot Reed-Solomon Decoder. In *Annual Conference on Information Science and Systems*, pages 700–705, 1999.