Design and Implementation of Turbo Decoders for Software Defined Radio

Yuan Lin, Scott Mahlke, Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan at Ann Arbor
{linyz, mahlke, tnm}@umich.edu

Chaitali Chakrabarti
Department of Electrical Engineering
Arizona State University
chaitali@asu.edu

Alastair Reid, Krisztián Flautner
ARM, Ltd.
Cambridge, United Kingdom
{alastair.reid, krisztian.flautner}@arm.com

ABSTRACT

Software Defined Radio(SDR) is an emerging paradigm for wireless terminals, in which the physical layer of communication protocols is implemented in software rather than by ASICs. Many of the current and next generation wireless protocols include Turbo coding because of its superior performance. However, Turbo decoding is computationally intensive, and its low power implementations have typically been in ASICs. This paper presents a case study of algorithm-architecture co-design of Turbo decoder for SDR. We present a programmable DSP architecture for SDR that includes a set of architectural features to accelerate Turbo decoder computations. We then present a parallel window scheduling for MAX-Log-MAP component decoder that matches well with the DSP architecture. Finally, we present a software implementation of Turbo decoder for W-CDMA on the DSP architecture and show that it achieves 2Mbps decoding throughput.

1. INTRODUCTION

Software Defined Radio (SDR) promises to revolutionize the communication industry by delivering low-cost, flexible software solutions for wireless communication protocols. Turbo coding is an error correction algorithm that has been included in many current and next generation wireless protocols due to its superior Bit Error Ratio(BER) performance. However, its implementation in wireless systems is challenging because of the high computation and low power requirements. While current commercial implementations use non-programmable ASICs, it is highly desirable to design a platform that is flexible enough to support Turbo decoder as well as other DSP algorithms in software, while still maintaining the energy efficiency of algorithm-specific ASICs. This paper presents a Turbo decoder implementation on a SDR platform that jointly considers the processor architecture and the characteristics of the decoding algorithm.

Turbo Decoder Overview. Turbo decoder consists of two component SISO decoders with interleavers between them as shown in Figure 1. The observed input sequence, y, is split into two streams and fed into the two component decoders. Both component decoders receive the systematic input y_s, and their respective parity inputs y_{p1} and y_{p2}. In each iteration, data first goes through the de-interleaver, and is decoded by the first component decoder. The result is then fed into the interleaver, and decoded by the second component decoder, the result of which is fed back into the
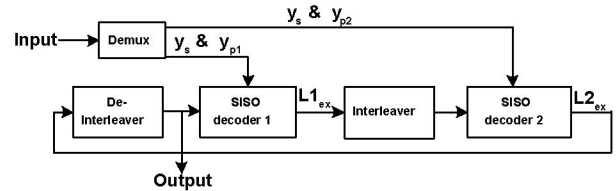

Figure 1: Block Diagram of a Turbo Decoder

de-interleaver. The extrinsic outputs from the two SISO decoders are labeled $L1_{EX}$ and $L2_{EX}$. This iterative process is repeated several times until the stopping criteria condition has been satisfied.

SISO Decoders. SISO decoders based on the MAP algorithm have superior performance. In our study, we implemented the MAX-Log-MAP algorithm. This is an approximation of the MAP algorithm that operates in the log-domain, allowing multiplications in the original MAP algorithms to be implemented by additions. A complete implementation study on every type of MAP algorithm is beyond the scope of this paper. However, the techniques explained in this paper can be applied in the implementation of other MAP algorithms.

Let s_k be the trellis state values at time k, then the likelihood values at time k, L_k, is defined by:

$$L_k = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \qquad (1)$$

The first term, $\alpha_{k-1}(s_{k-1})$, is the *alpha* metric that calculates the probability of the current state based on the input values before time k. The second term, $\gamma_k(s_{k-1}, s_k)$, is the branch metric that calculates the probability of the current state transition. The third term is the *beta* metric that calculates the probability of the current state given the future input values after time k. Alpha and beta calculations are defined recursively as shown below:

$$\alpha_k(s_k) = max(\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)) \qquad (2)$$
$$\beta_k(s_k) = max(\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})) \qquad (3)$$

As shown, the alpha computation is forward recursive and beta computation is backward recursive. Let s^1 and s^0 be the 1-branch and 0-branch trellis state transitions. The soft output value, log-likelihood calculation (LLC) at time k, is defined by subtracting the maximum likelihood values of the 1-branch state transitions from the maximum likelihood
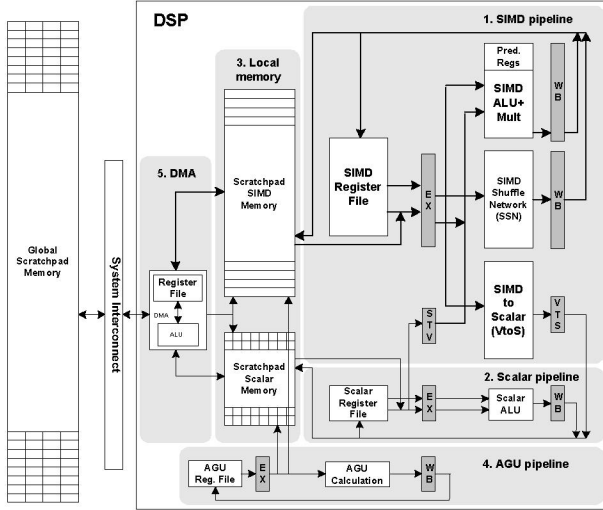
Figure 2: DSP Architecture for SDR including a processing engine with three pipelines and 4 register banks (SIMD 32x16bit, SIMD 32x1bit, scalar 16bit, AGU 16bit) and a DMA engine with one pipeline and 1 register bank (32bit)

values of 0-branch state transitions.

$$LLC_k = \max_{s^1}(L_k) - \max_{s^0}(L_k) \qquad (4)$$

Related Work. There have been numerous studies on parallelizing MAX-Log-MAP for ASICs [9], [13]. Although these studies provide interesting insights into high performance Turbo decoder design, most of these techniques cannot be applied to SDR, as we will show in Section 3. The existing software implementations can be separated into two groups. The first group includes implementations on mainstream DSPs, such as TI's C6X that achieves throughput of 286Kbps [2], Motorola's Starcore that achieves throughput of 1.8Mbps [6], and ST-Microelectronics' ST120 that achieves throughput of 540Kbps [10]. The second group includes ASIC and programmable FPGA accelerators for RISC processors. These include the XiRisc processor implementation with a throughput of 270Kbps [12] and Tensilica's Xtensa-based microprocessor with a throughput of 1.48Mbps [3]. Our processor achieves a comparable throughput of 2Mbps for W-CDMA. However, a detailed comparison with prior solutions is difficult because of lack of implementation details.

The rest of the paper is organized as follows. In Section 2, we introduce our SIMD-based (Single Instruction Multiple Data) high-performance DSP processor, and highlight a set of DSP architectural features to accelerate Turbo decoder computation. In Section 3, we present a parallel window MAX-Log-MAP scheduling scheme that best fits our DSP architecture. In Section 4, we describe the software implementation and optimizations for Turbo decoding in W-CDMA. Our results show that our DSP implemented in 90nm technology is able to achieve 2Mbps throughput while consuming sub-watt power.

2. DSP ARCHITECTURAL SUPPORT

In this section, we present our SIMD-based DSP architecture, and highlight the key architectural features that result

in an efficient software implementation of the Turbo decoder. In our previous work, we have implemented the complete W-CDMA protocol in C, and analyzed the protocol's algorithm characteristics [7]. Since our DSP architecture is designed to support many wireless protocol algorithms, the architectural features presented in this section are not specifically built for Turbo decoding, but rather for a large class of DSP algorithms.

Figure 2 shows the overall DSP architectural diagram. It consists of 5 major components: 1) a SIMD (Single Instruction, Multiple Data) pipeline for supporting Turbo decoder's vector operations; 2) a scalar pipeline for sequential operations; 3) two local scratchpad memories for the SIMD pipeline and the scalar pipeline; 4) an AGU pipeline for providing the addresses for local memory access; and 5) a programmable DMA unit to transfer data between memories and interface with the outside system. The SIMD pipeline, scalar pipeline and the AGU pipeline execute in VLIW-styled lock-step, controlled with one program counter (PC). The DMA unit has its own PC, its main purpose is to perform memory transfers and data rearrangement. It is also the only unit that can initiate memory access with the global scratchpad memory. A detailed description of this architecture can be found in [8].

2.1 SIMD Pipeline

The SIMD pipeline performs operations on 32 16-bit wide elements, in parallel. The register file has a 2 read port, 1 write port memory structure. It can only read and write data on the 512-bit SIMD bitwidth boundary (32 elements of 16-bits). The SIMD pipeline supports fixed-point DSP arithmetic and logic operations, such as saturated computations and multiply-and-accumulate (MAC) operations.

32-Wide SIMD Computation. Traditionally, embedded DSP processors fall under one of two categories: SIMD and VLIW. The SIMD approach separates the register file into clusters, reducing the complexity of access ports. However, in most commercial solutions, SIMD width is conservatively limited between 4 to 8, due to data array alignment difficulties in general purpose DSP computations [5]. VLIW architectures support ILP(Instruction Level Parallelism) very well, as memory operations can be done concurrently with multiple data computations. However, they are not very well suited for vector DLP (Data Level Parallelism), as each data computation requires its own instruction. The majority of computation in Turbo decoder is the trellis state update that can be expressed as vector operations with constant vector length. These types of constant-sized vector operations are best implemented with SIMD architectures. Furthermore, because the Turbo decoder has well-defined data shuffling patterns (trellis butterfly) and regular vector width (the number of trellis states is always a power of 2), we can scale up the SIMD width without worrying about data-alignment constraints. For the W-CDMA Turbo decoder, a 400MHz, 32-wide SIMD pipeline is used. In our previous work [8], we found this configuration to have the lowest power consumption among all of the SIMD width and processor frequency combinations that can meet the real-time computational requirements of W-CDMA.

Predicated Add/Subtract Operations. In branch metric calculations, it is useful to execute addition and subtraction operations concurrently on the different elements in an SIMD vector. Our SIMD pipeline has been augmented
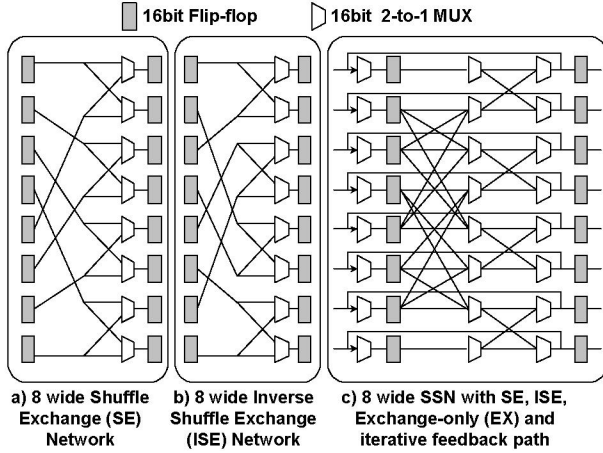
16bit Flip-flop   **16bit 2-to-1 MUX**

a) 8 wide Shuffle
Exchange (SE)
Network

b) 8 wide Inverse
Shuffle Exchange
(ISE) Network

c) 8 wide SSN with SE, ISE,
Exchange-only (EX) and
iterative feedback path

Figure 3: 8-wide SIMD Shuffle Network(SSN)

to support this operation, in which a bit-vector is used to determine whether each element has to perform an addition or subtraction.

SIMD Shuffle Network (SSN). In the Turbo decoder, SIMD data shuffling is needed to implement the trellis butterfly. This is supported on many modern DSPs through strided memory access, which requires multi-port memory structures that are power hungry [11]. Because of the power restrictions of SDR architectures, we limit the processor's memory to be single-ported with no support for strided access. Instead, the data shuffling operations are supported through the SSN (see Section 4). Figure 3c shows a simplified 8-wide version of the network (In the architecture, the SSN has the same width as the SIMD width). The SSN consists of a shuffle exchange (SE) network (Figure 3a), an inverse shuffle exchange (ISE) network (Figure 3b), Exchange Only (EX), and an iterative feedback path for iterative shuffling.

2.2 Scalar Support

In addition to SIMD computations, Turbo decoders also require scalar operations. Therefore, our processor includes a 16-bit scalar pipeline that executes in lock-step with the SIMD pipeline, with additional scalar-to-SIMD and SIMD-to-scalar operations to transfer values between the two pipelines.

Scalar-to-SIMD operations. The SIMD operation can take one of its source operands from the scalar pipeline. This feature is useful in implementing trellis computations. It is done through the STV (Scalar-To-Vector) registers, shown in the SIMD pipeline portion of Figure 2. The STV contains 4 16-bit registers, which only the scalar pipeline can write, and only the SIMD pipeline can read. The SIMD pipeline can read 1, 2, or all 4 STV register values and replicate them into 32-element SIMD vectors.

SIMD-to-Scalar operations. SIMD-to-Scalar operations transfer values from the SIMD pipeline into the scalar pipeline. This is done through the VTS (Vector-To-Scalar) registers, shown in Figure 2. There are two different SIMD reduction operations that are needed for the MAX-Log-MAP decoding: 1) find the maximum or minimum values of an SIMD vector, and 2) transferring one element of the SIMD vector into the scalar pipeline.
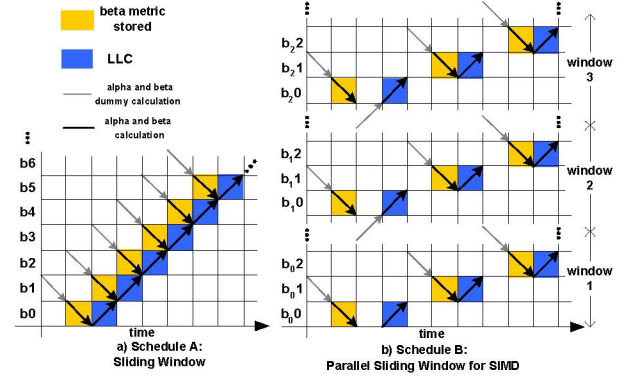
2.3 Programmable DMA

Figure 4: Parallel MAX-Log-MAP Scheduling

The DMA controller is responsible for transferring data between memories. It is the only component in the processor that can access the SIMD, scalar and global memories. Traditional DMA controllers perform copies from one memory region to another, where regions are either contiguous or have a simple strided access patterns. It is usually implemented as a slave device, controlled through a set of DMA registers and synchronization instructions that are executed on the master processor. In our processor, the DMA is also implemented as a slave device controlled by the scalar pipeline. However, it has the capability to execute its own instructions on its internal register file and ALU, similar to the scalar pipeline. This gives the DMA the ability to access the memory in a wide variety of application-specific patterns without assistance from the master processor. This ability allows inherently scalar components of the Turbo decoder, such as the interleaver, to be implemented efficiently on the DMA.

3. PARALLEL SISO DECODER DESIGN

MAX-Log-MAP decoder can be parallelized by dividing the decoding block into smaller sub-blocks, and performing alpha-beta-LLC computations on each sub-block independently. To account for the potential BER performance degradation, additional dummy calculations have to be performed before the alpha and beta computations in each sub-block. Figure 4a shows one possible schedule with the alpha, beta, and dummy beta calculations. For simplicity, the length of the dummy calculations in Figure 4 is the same as the number of alpha and beta calculations.

There have been many studies analyzing the trade-offs between different sliding-window and parallel-window scheduling algorithms [9] [13] [1]. Most of these studies assume ASIC architectures with one or more dedicated alpha and beta processors that can execute concurrently. For a software implementation, concurrent execution requires the alpha and beta calculations be expressed as two independent threads. If they are implemented as a single thread, we would have to rely on the compiler to discover independent instructions that can execute in parallel. However, SIMD processors cannot process multiple threads or multiple instructions at the same time. For instance, if the schedule in Figure 4a is implemented on our DSP processor, the alpha and beta calculations would be serialized. The parallel sliding window schedule, shown in Figure 4b, is better suited for a software implementation on SIMD-based processors. In

24

this schedule, one Turbo decoding block is broken up into N parallel windows. These multiple windows are processed concurrently with the same instruction sequence. Within each window, alpha, beta, and dummy calculations are computed sequentially. Compared to the schedule in Figure 4a, this schedule requires N-1 extra dummy alpha calculations to initialize the starting alpha metric for all of the windows except the first one. The number of parallel windows, N, is determined by the processor's SIMD width, W, and the trellis width, S. For W-CDMA Turbo coding, the trellis size is 8, and thus for the 32-wide SIMD processor, $\frac{W}{S} = 4$ windows can be processed in parallel.

4. TURBO DECODER IMPLEMENTATION

In this section, we present a case study of a software Turbo decoder implementation for W-CDMA: rate $\frac{1}{3}$, K=4 RSC encoder with block interleaving.

4.1 Trellis Computation Implementation

The majority of the Turbo SISO decoder operations are spent on trellis state updates. In this section, we present an efficient implementation of the trellis computation using the architectural features mentioned in Section 2.

Trellis computation. Figure 5a shows the two types of trellis computation for an 8-state trellis. The dark and light edges correspond to 0- and 1-branch transitions. Figure 5b shows the SIMD implementation of the alpha trellis computation. Beta trellis computation is not shown; it follows the same sequence of operations. Trellis computation can be divided into two steps, branch-metric calculation (BMC) and add-compare-select calculation (ACS). In the BMC stage, the inputs are loaded as scalar values from the scalar local memory. The scalar value is then duplicated into a vector using the STV registers. The input vector, In, is correlated with constant metric values, m, to calculate the branch metric values for 0-branch and 1-branch. The correlation function, shown in the figure as M, is defined as $b[i] = In[0] * m[i][0] + In[1] * m[i][1], i : 0 - 7$. Since the metric values $m[i]$ are either 1 or -1, we can use predicated add/subtract instructions, where m is stored as a predicate bitvector.

In the ACS step, the trellis state vector, s_t, adds both 0-branch and 1-branch metrics, and compares each pair of values to select the next trellis state vector s_{t+1}. The SSN network is used to rearrange the vectors between SIMD operations. The rearrangement step and the assembly code are shown in Figure 5c. Before this compare-and-select step, we first interleave the 0-branch and 1-branch metric values (not shown in the figure). Then two SIMD permutation operations are performed using the SSN. The first permutation operation (op1) takes one cycle, using the ISE (Inverse Shuffle Exchange) pattern. The second permutation operation (op2) takes two cycles, with one additional EX (Exchange only) permutation. Finally, a SIMD compare-and-select operation (op3) is performed to choose the next trellis state values.

Parallel Window Trellis Implementation. In W-CDMA, the trellis state size is 8. Since the SIMD processor width is 32, we can process four windows in parallel using the schedule shown in Figure 4b. Figure 6 shows an example of two 4-state trellis computations stacked together onto a 8-wide SIMD processor. The two trellis states are represented by the s and t vectors. The SSN's permutation patterns are
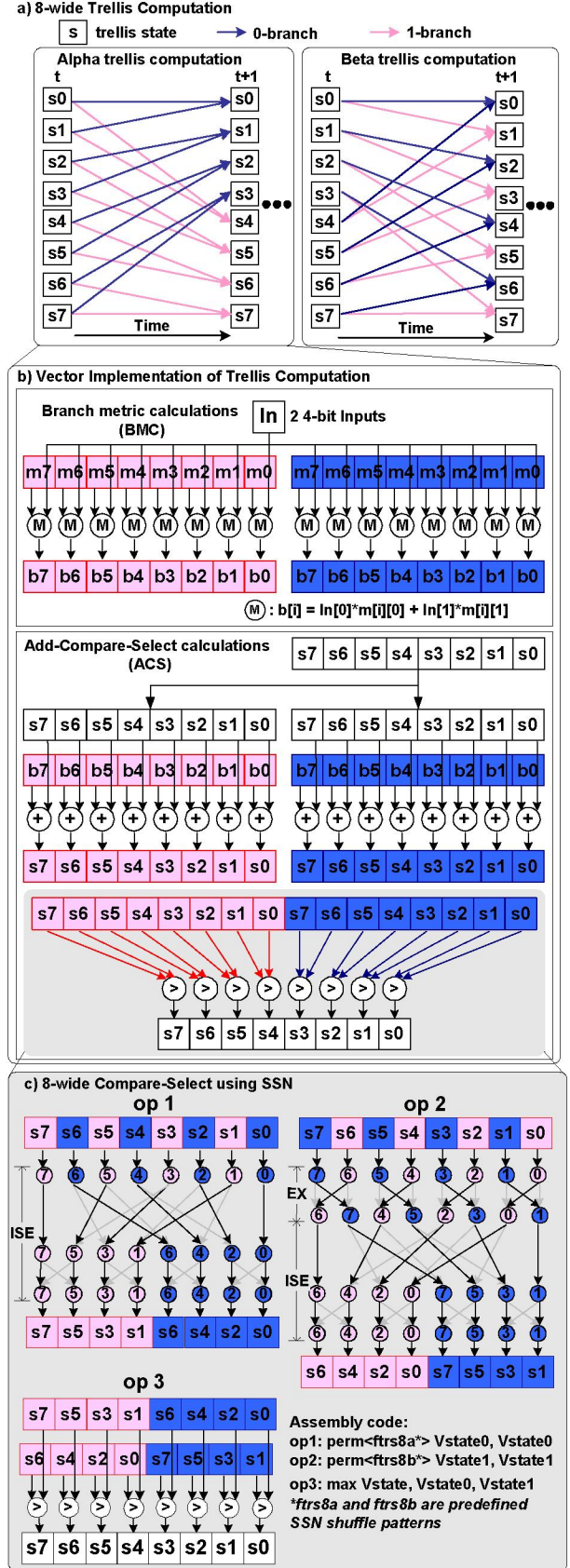

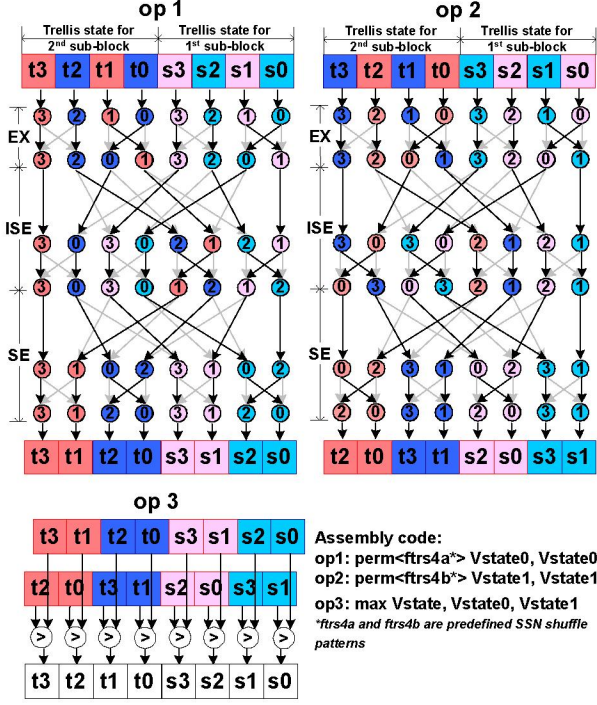Figure 5: Trellis state computation, and SIMD implementation using the SSN

Figure 6: Parallel window trellis compare-select for 2 4-state trellis on an 8-wide SSN
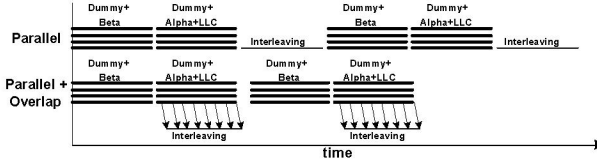

Figure 7: Computation time of 1 iteration of Turbo decoding for parallel processing vs. parallel processing with overlapping interleaving

different from that shown in Figure 5c, with both op1 and op2 requiring more iterations to complete. Although processing parallel sub-blocks does not increase the number of SIMD operations, additional scalar support is required.

Interleaver Implementation. While the SISO decoder computations can be parallelized, interleaving is a data shuffling function that cannot utilize the processing power of the SIMD pipeline. Figure 7 shows the computation time of 1 iteration of the Turbo decoder for two processing scenarios. With SISO parallelization, the interleaver underutilizes the processor's resources and limits the overall achievable throughput. In order to alleviate this sequential bottleneck, we propose a technique to overlap the interleaving operation in the background of the SISO decoder. This is based on the observation that in a MAX-Log-MAP decoder, output data is produced one element at a time during the LLC computation.

In this method, the interleaving is done during the memory transfer. It requires the DMA controller to be programmed to generate the source and destination addresses for each memory transfer. If the memory transfer rate is faster than the output rate of SISO decoder, then the latency of interleaving can be completely hidden behind the compu-

tation. In the block interleaving specified in W-CDMA, each block element's address can be calculated by adding the row offset and the column offset. In our implementation, this requires 2 additions, 3 memory reads, and 1 memory write, which translates into 9 cycles. The SISO decoder produces an output every 9.25 cycles, enabling us to completely overlap the interleaving latency with the computation latency.

4.2 W-CDMA Turbo Implementation Results

SISO Decoder Throughput Analysis. In this section, we examine the achievable SISO decoding throughput as a function of algorithm specifications and architectural configurations. Let K be the RSC encoder constraint length, then the size of trellis state is defined as $S = 2^{K-1}$. We assume that the SIMD width, W, is equal or greater than trellis state width: $W \geq S$. To fully utilize the SIMD pipeline, we implemented the parallel sliding window schedule, where N windows are processed in parallel, and $N = \frac{W}{S}$. If we let M be the total number of sub-blocks for one block of Turbo decoding, then each window computes $\frac{M}{N}$ sub-blocks. In the case where $W < S$, trellis computation can still be implemented. The details are omitted due to space limitations.

Let T_{block} be the average number of cycles to compute alpha, beta, LLC, and dummy computations for one sub-block of size L. If we let T_α be the number of cycles to compute one SIMD alpha trellis update, then the latency to compute one alpha trellis update is $\frac{T_\alpha}{N} + 3C_L$, where C_L is the number of cycles to load one scalar value from memory. The SIMD alpha trellis latency is divided by N, because N windows of trellis are computed at once. Three scalar loads are needed for loading two inputs (rate $\frac{1}{2}$ decoding), and one extrinsic value. The SIMD beta trellis updates, T_β, follow the same set of operations as the alpha computation, with three scalar loads. The SIMD LLC computation, T_{LLC}, computes N decoded bits at once. The SIMD dummy trellis computations are also done in groups of N windows, with each window requiring three scalar memory loads. For a sub-block of size L, the dummy alpha and beta calculations have to be done on at least $5K$ (K equals 4 here) elements to stabilize the trellis states and not affect the overall error correction performance. The overall latency T_{block} is shown in Equation 5, where T_d is the total dummy computation latency for one sub-block.

$$T_{block} = T_d + L(\frac{T_\alpha + T_\beta + T_{LLC}}{N} + 6C_L) \quad (5)$$

$$T_d = 5K(\frac{T_{d\alpha} + T_{d\beta}}{N} + 6C_L) \quad (6)$$

As shown in Equation 6, T_d is a function of dummy alpha and dummy beta computations. Let $T_{d\alpha}$ be the number of cycles for one SIMD dummy alpha trellis computation, and $T_{d\beta}$ be the number of cycles for one SIMD dummy beta trellis computation. In our implementation, $T_{d\alpha} = 10$ and $T_{d\beta} = 10\frac{N}{M}$. Dummy beta calculations are scaled by $\frac{N}{M}$ because the beta trellis states of the N sliding windows need to be initialized once for every M sub-blocks. In W-CDMA, Turbo decoding block size = ML, and ranges from 320 bits to 5114 bits [4]. Given $L = 100$, the number of sub-blocks, M, varies from 4 to 52. In our throughput calculation, we assume the longest $T_{d\beta}$ latency with $M = 4$, and the total Turbo decoding black size = 400.

In our implementation, $T_\beta = 11$, $T_\alpha + T_{LLC} = 25$, $T_d = 220$ and N, the number of windows processed in parallel, is 4

. Alpha and LLC computations have been grouped together because they are executed together. A scalar load takes 3 cycles, but if we use prefetching instruction, we can shorten it to 1 cycle: $C_L = 1$. Six scalar load operations are required for alpha and beta. The length of one sub-block, L, is 100. Based on the numbers shown above, the overall latency is $T_{block} = 1720$.

Architectural Implications. As shown in Equation 5, increasing the number of concurrent sub-blocks, N, decreases cycle count. This can be achieved by increasing the SIMD width W. However, doubling W doubles the size of the processor, which also doubles the power consumption. The other trade-off is the length of a sub-block, L, as longer sub-blocks reduce the relative ratio of dummy calculations per decoded output. However, longer sub-blocks also require more memory to store alpha metric values. The constraint between SIMD memory and sub-block size is $E_v \geq 2WL$, where E_v is the size of local SIMD memory. This means that we should choose the largest sub-block size that can fit in the SIMD memory. Our DSP processor has an 8KB SIMD memory, which holds 128 512-bit entries. With 28 entries reserved for holding spilled temporary register values, the sub-block size L is chosen to be 100.

Throughput Results. The overall decoding throughput of the Turbo decoder is determined by I times the combined latencies of the 2 SISO decoders and the 2 interleavers, where I is the number of iterations. In our implementation, because the interleaver latencies are hidden, the throughput is only dependent on the SISO decoders' performance. Equation 7 shows the Turbo decoder throughput, R_{Turbo}, as a function of the processor's clock speed C_p, the number of Turbo iterations I, the average latency for a SISO decoder to produce one bit of decoding output T_{1bit}, and additional computations for extrinsic value scaling C_M.

$$R_{Turbo} = \frac{C_p}{2I(T_{1bit} + C_M)} \qquad (7)$$

Because the Turbo decoder is a block decoder, we define SISO decoder latency as $T_{1bit} = \frac{T_{block}}{L} = 17.2$, where T_{block} is the latency for processing one data block of size L. With our SDR processor running at 400MHZ, $C_p = 400M$, and $C_M = 2$, we are able to achieve $1.73Mbps$ and $2.08Mbps$, with $I = 6$ and $I = 5$ respectively. Note that W-CDMA's DCH (Data CHannel) requires a data rate of 2Mbps.

If we wish to achieve higher throughput, we will need to resort to other optimizations techniques. We can scale up the frequency, increase the SIMD width, or map the algorithm onto multiple processors. In particular, our SIMD pipeline has a 32-wide 16-bit datapath, but most Turbo decoder computations only require 8-bit precision. With some extra hardware logic, we can support two 8-bit computation on every 16-bit datapath, making our SIMD pipeline a 64-wide 8-bit datapath. This can potentially double the overall throughput of the SISO decoder. Compiler optimization techniques, such as software pipelining, are also viable options. Finally, our previous study [8] has shown that our DSP processor consumes approximately 800mW in 180nm technology. Scaling down to 90nm, the same throughput can be achieved with a power consumption of approximately 100mW.

5. CONCLUSION

In this paper, we presented a study on algorithm-architecture co-design of Turbo decoder for Software Define Radio. We first highlighted key DSP architectural features that benefits Turbo decoder processing. We then explain the algorithm design trade-offs and software implementation issues of a Turbo decoder. As a case study, we implemented a Turbo decoder for the W-CDMA standard. Our results show that we are able to achieve 2Mbps decoding throughput while consuming sub-watt power on our DSP processor running at 400MHZ.

7. REFERENCES

[1] E. Boutillon, W. Gross, and P. G. Gulak. VLSI Architectures for the MAP Algorithm. In *IEEE Trans. on Communications*, volume 51, no. 2, pages 175–185, Feb. 2003.

[2] W. Ebel. Turbo-Code Implementation on C6x. In *Tech. Report, Alexandria Research Inst., Virginia Polytechnic Inst. State Univ.*, 1999.

[3] F. Gilbert, M. J. Thul, and N. Wehn. Communication Centric Architectures for Turbo-Decoding on Embedded Multiprocessors. In *Proc. Design, Automation and Test Europe*, pages 356–461, Mar. 2003.

[4] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access For Third Generation Mobile Communications*. John Wiley and Sons, LTD, New York, New York, 2001.

[5] H. C. Hunter and J. H. Moreno. A New Look at Exploiting Data Parallelism in Embedded System. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 159–169, 2003.

[6] F. Kienle, H. Michel, F. Gilbert, and N. Wehn. Efficient MAP-algorithm Implementation on Programmable Architectures. In *Advances in Radio Science*, volume 1, pages 259–263, 2003.

[7] H. Lee et al. Software Defined Radio - A High Performance Embedded Challenge. In *Proc. 2005 Intl. Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, Nov. 2005.

[8] Y. Lin et al. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[9] M. Mansour and N. Shanbhag. VLSI Architectures for SISO-APP Decoders. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 11, no. 4, pages 627–650, Aug. 2003.

[10] H. Michel, A. Worm, M. Munch, and N. Wehn. Hardware/Software Trade-offs for Advanced 3G Channel Coding. In *Proc. Design, Automation and Test Europe*, pages 396–401, Mar. 2002.

[11] S. Rixner et al. Register Organization for Media Processing. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 375–386, Jan. 2000.

[12] A. L. Rosa, L. Lavagno, and C. Passerone. Implementation of a UMTS Turbo Decoder on a Dynamically Reconfigurable Platform. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, no. 1, pages 100–106, Jan. 2005.

[13] Z. Wang, Z. Chi, and K. Parhi. Area-Efficient High-Speed Decoding Schemes for Turbo Decoders. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 10, no. 6, pages 902–912, Dec. 2002.