SLATE: A Combined Architecture for LDPC and Turbo Decoding

Stevo Bailey, Ben Keller, and Paul Rigge
University of California, Berkeley
{stevo, bkeller, rigge}@eecs.berkeley.edu

***Abstract*—LDPC and turbo codes are channel codes commonly used for wireless communication. Decoding algorithms are computationally demanding, and so efficient implementations are often inflexible, targeting only the codes specified by a given standard. When support for multiple standards is needed, multiple decoders are generally used. We study the algorithms for decoding each standard and find that some functional units can be shared between both types of decoders. This work presents a design for a combined architecture decoder that supports the codes defined in the 802.11ac and LTE standards. We present some preliminary performance results and show that the majority of the area is consumed by LDPC-specific components. We conclude that this combined architecture allows a turbo decoder to be added to an LDPC decoder with little overhead.**

I. Introduction

High performance channel codes are essential for achieving efficient, reliable communication. However, these codes, such as LDPC or turbo codes, have computationally demanding decoding algorithms. Implementations of these decoders are often special-purpose hardware targeting the specific codes designated by a standard, especially as mobile standards continue to demand increased throughput and power efficiency. However, sharing flexible computation resources can reduce cost and provide more space for other inflexible resources, such as batteries in mobile applications. Flexible computation units are especially useful for supporting different standards and providing a path for supporting future standards with existing hardware.

In this report, we present SLATE (SLATE is an LDPC And Turbo Engine), a hardware design that implements both the 802.11n and LTE Advanced standards. Presently, such systems generally have separate hardware for decoding the LDPC codes of 802.11 and the turbo codes of LTE. As both decoders are relatively large and perform similar functions, it is desirable to combine them into a flexible decoder. However, the computational demands and high throughput requirements make a truly general purpose decoder impractical. This paper describes how to integrate an LDPC decoder with a turbo decoder in an efficient architecture.

The remainder of the report is structured as follows. Section II provides background on the algorithms and computational requirements for the decoders. Section III presents the chosen decoder architectures and their designs. Section IV discusses the components that can be shared in a combined LDPC/Turbo architecture. Section V presents a discussion about the performance of our design. Section VI gives some ideas for improving this design, and Section VII concludes.

II. Background

Channel coding attempts to protect data from corruption by a noisy channel. An encoder E maps data from an alphabet $\mathcal{A}$ to a subset of the channel alphabet $\mathcal{X}$ called $\mathcal{C}$. Each element of the codebook $\mathcal{C}$ is called a codeword. For a codeword $c = E(x)$, where $c \in \mathcal{C}$ and $x \in \mathcal{A}$, a decoder receives a signal y that has been corrupted by the channel. The goal of a (possibly randomized) decoder D is to produce an output that maximizes $\Pr[D(y) = x]$. For many codes, optimal decoders have impractically high computational complexity; however, LDPC and turbo codes have iterative approximate algorithms that produce good results at acceptable computational cost.

A decoder receives soft information from a channel demodulator and produces soft (or hard) estimates of the received bits. For numerical reasons, these soft bits are often represented as log-likelihood ratios, defined here as

$$l(u) = \log \frac{\Pr[u = 0]}{\Pr[u = 1]}$$

A. LDPC Decoding

A low-density parity-check (LDPC) code is a linear block code defined by its parity check matrix $\mathbf{H}$, where a parity check matrix has the property that for every codeword c, $\mathbf{H}c^T = 0$. LDPC codes are so named because $\mathbf{H}$ is sparse, a property which allows for an efficient decoder implementation. $\mathbf{H}$ can also be represented graphically as a factor graph, as shown in Figure 1. A bit i from the channel is represented by a variable node $\mathrm{VN}(i)$ and each parity check j is represented by a check node $\mathrm{CN}(j)$. An edge between variable node i and check node j exists if $\mathbf{H}_{j,i} = 1$.

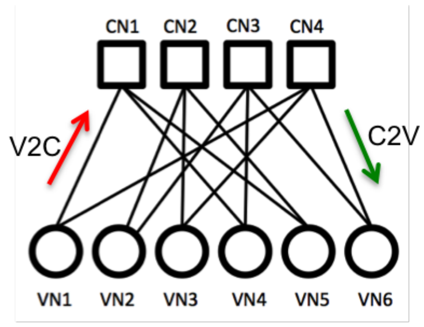

Fig. 1: Factor graph representation of a simple linear block code, with circles representing variable nodes and squares representing check nodes. (Image credit: Matt Weiner)

0	-	-	-	0	0	-	-	0	-	-	0	1	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	0	0	-	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	0	-	-	-	-	-	0	0	-	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	0	0	-
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
3	-	-	-	16	-	-	2	25	5	-	-	1	-	-	-	-	-	-	-	-	-	-	0

Fig. 2: One parity check matrix for 802.11ac code. The block length is 648 bits, rate is 1/2, and subblock size is 27 bits. A non-blank entry represents a cyclically shifted identity matrix.

The 802.11n standard [1] uses specially structured LDPC codes said to be quasi-cyclic. An example of such a structured LDPC code is show in Figure 2. Note that each non-blank entry is a cyclically shifted identity matrix, so a hardware implementation can use efficient shifters to simplify routing.

The offset min-sum algorithm is a popular variant of the message passing algorithm for decoding LDPC codes. It is called message passing because computations are performed locally at check and variable nodes and sent as messages to other check and variable nodes to act as input for the next iteration of messages. Connections between the variable and check nodes are determined by the parity check matrix **H**. The check node computation is performed as

$$\begin{aligned}\mathrm{C2V}(i,j) = & \prod_{i' \in N(i)\setminus\{j\}} \operatorname{sgn}\left(\mathrm{V2C}(j,i')\right) \\ & \max\left(\min_{i' \in N(j)\setminus\{i\}} |\mathrm{V2C}(j,i')| - \beta, 0\right)\end{aligned}$$

where $N(i)$ is the neighborhood of i defined as all the nodes to which node i is connected, and β is the offset. The variable node computation is

$$\mathrm{V2C}(i,j) = \sum_{j' \in N(i)\setminus\{j\}} \mathrm{C2V}(j',i) + l(u_i)$$

where $l(u_i)$ is the ith LLR from the channel. Messages are passed back and forth until some stopping condition is met: either a valid codeword is found or the decoder reaches some maximum number of iterations.

B. Turbo Decoding

The turbo coding scheme in LTE is a parallel concatenated convolution code [2]. The encoder is shown in Figure 3; uncoded bits are fed into the three-state encoder, producing a systematic output X, an output Y_1 that has been encoded with an 8-state convolutional code, and an output Y_2 that has been encoded with the same 8-state convolution code with the order scrambled by an interleaving defined by the standard. Accordingly, all LTE turbo codes are rate 1/3. The length of these outputs (i.e., the block size of the code) varies from 40 to 6144 bits.

An efficient decoder for turbo codes is shown in Figure 4. A soft-input soft-output (SISO) unit implementing the BCJR algorithm produces bit estimates that are interleaved and used as inputs by the other SISO unit [3]. The a posteriori probability computed by the SISO unit is given for a state s_k in the trellis at time k as

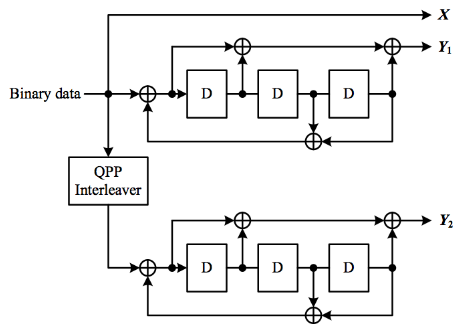

Fig. 3: Encoder for LTE turbo code [3].

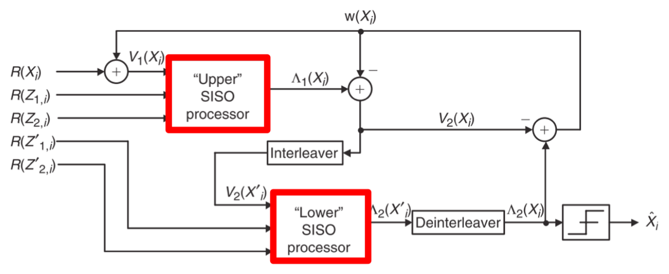

Fig. 4: Turbo decoder with SISO units. X_i and X_i' are the systematic bits in deinterleaved and interleaved order, respectively. Z_1 and Z_2 are the encoded bits, and their primed counterparts are in interleaved order. Each SISO block is considered one half-iteration of the decoding algorithm.

$$\begin{aligned}\Lambda(\hat{u}_k) = & \overset{*}{\max_{u_k=1}}\left(\alpha_{k-1}(s_{k-1} + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k)\right) \\ & - \overset{*}{\max_{u_k=0}}\left(\alpha_{k-1}(s_{k-1} + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k)\right)\end{aligned}$$

where α_k is the forward state metric, β_k is the backward state metric, and γ_k are the branch transition probabilities. The α and β terms are readily computed by

$$\begin{aligned}\alpha_k(s_k) = & \overset{*}{\max_{s_{k-1}}}\left(\alpha_{k-1}(s_{k-1})\right) \\ \beta_k(s_k) = & \overset{*}{\max_{s_{k+1}}}\left(\beta_{k+1}(s_{k+1})\right)\end{aligned}$$

where $\max^*$ approximates an exponential sum:

$$\overset{*}{\max}(a,b) = \max(a,b) + \log(1 + e^{-|a-b|})$$

The branch metrics encode the probability of moving from state s_{k-1} to state s_k given the observed channel output. Accordingly, each series of α_k and β_k can be calculated by performing forward and reverse trellis traversals respectively.

III. Decoder Architectures

Various architectures exist for both LDPC and turbo decoders. Decoders can vary in the amount of parallelism, the decoder schedule, and interface with other blocks. In this section we briefly describe our design decisions.

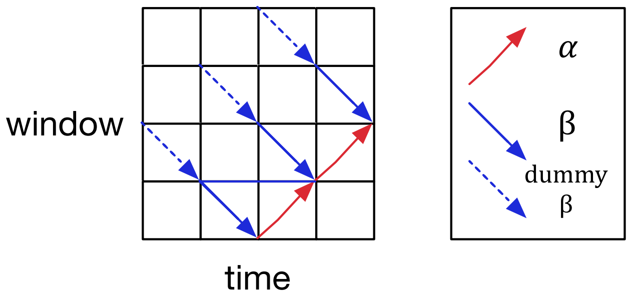

Fig. 5: This figure shows the schedule our design follows for turbo decoding. Dummy betas must be computed from the following window before betas can be computed. Betas are computed and stored in a scratch memory before alphas are computed. As alphas are computed, betas are read out in reverse order and the a posterior LLR is computed.

A. *LDPC Decoder*

For simplicity, the LDPC decoder contains a single variable node group and a single check node group, so that only one subblock is decoded at a time. All memory and marginalization is done within the variable nodes; the check nodes just compute the minima. C2V marginalization is handled by passing a V2C address with the minima, which is then compared inside the variable node [4]. V2C marginalization requires storing each C2V message for use in the next iteration. To save memory, only the sign bit and MSB of each C2V is stored. A scheduler contains all the parity check matrices and feeds the shift values into the shifters. Each node is completed in a single cycle (the design is not pipelined), so the total cycles per iteration is roughly the number of subblocks in a matrix.

B. *Turbo Decoder*

The turbo decoder performs 4-way parallel decoding on 32-bit windows. Within each parallel decoder group, "Dummy betas" are first calculated to initialize the beta (reverse) trellis traversal. Then the final dummy beta value initializes the real beta traversal for the previous window, with the betas for that window stored in a small scratchpad memory. The alpha (forward) trellis traversal allows the immediate calculation of the output LLRs for that window using the beta results stored in the scratchad (pipelining leads to a latency of several cycles). These three sets of trellis traversals can take place in parallel, as shown in the schedule in Figure 5. The decoder then writes these LLRs back to memory to be interleaved and used in the next half-iteration.

IV. Combined Architecture

Some parts of each decoder are very specialized and difficult to reuse. The LDPC decoder has cyclic shifts of sizes 27, 54, or 81; these shifts have no counterpart in the turbo decoder as the interleaver is a permutation and changes with blocksize. The LDPC decoder also has many small memories for message marginalization and accumulation, whereas turbo has a few small memories as scratchpads for each window. However, each algorithm contains arithmetic elements in common. We exploit this fact to share hardware between the two decoder implementations.

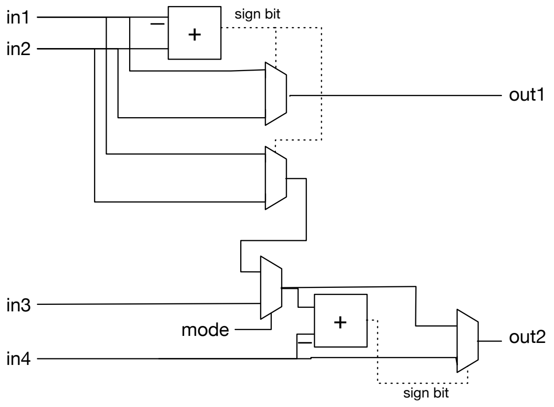

Fig. 6: Combined ACS units for LDPC and turbo. For LDPC, in1 is the previous cycle's first minimum (out1) and in4 is the previous cycle's second minimum (out2). The V2C message is in2 and in3 is unused. For turbo, out1 is the max of in1 and in2 and out2 is the max of in3 and in4.

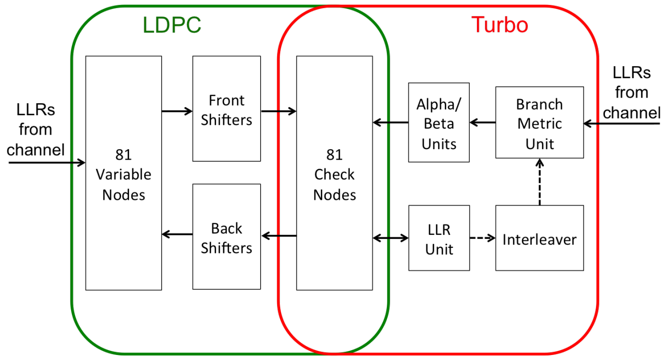

Fig. 7: Combined architecture with shared ACS units. For both modes, LLRs are read from an external memory. The turbo decoder processes four windows in parallel.

A. *Common Elements*

The add-compare-select (ACS) units found in the check nodes of the LDPC decoder and the $\max^*$ operation of the turbo decoder were identified as common elements between the two decoders. Our design does not share the first adders in the turbo decoder that sum α, β and γ, and has a simpler check node, allowing for two $\max^*$ units from every check node. Figure 6 shows block diagrams showing how the ACS functions in LDPC or turbo mode. The LUT correction for $\max^*$ was not implemented for this design. A similar hardware combination was described by [5], although their approach differs in that their design uses one check node for a single $\max^*$ operation.

B. *Combined Architecture*

The combined architecture shown in Figure 7 shares the ACS units for the LDPC decoder's check nodes and the turbo decoder's max units. One variable node group of 81 variable nodes is connected through front and back shifters to the check nodes. The variable node group and shifters are controlled by a scheduler that encodes the various parity check matrices from the standard. Each variable node must store the accumulated C2V messages for the current iteration, the old accumulated C2V messages from the previous iteration, and the C2V and V2C messages for marginalization. Our implementation just stores the top two C2V message bits and the V2C sign bit. Thus the total memory per variable node is about 78 bytes. Marginalization and conversion back and forth between sign-magnitude and two's complement formats contribute the significant combinational logic overhead seen in the variable nodes.

The three-input min blocks in LDPC mode become two max blocks in turbo mode (see Figure 6), so there are enough for 162 2-input max blocks. Each parallel turbo decoding group uses 8 max operations for each of the three trellis traversals, and 14 for the reductions to calculate the final LLRs., for a total of 38 operations per parallel group. The 162 available max blocks are therefore sufficient for four parallel windows. An interleaver and LLR unit load LLRs from the channel through our external memory.

V. Hardware Results

We implemented a parameterized generator for our combined LDPC-turbo decoder in Chisel. All memories were implemented as flip-flops. We believe that minor bugs persist in both the LDPC and turbo implementations, so we are not able to present bit-error plots for decodings completed by the hardware.

This design was synthesized with 6-bit LLRs. We also synthesized the LDPC and turbo decoding blocks individually. A summary of these synthesis results are shown in Table I as reported by Design Compiler. The area of the combined decoder is dominated by the 81 LDPC variable nodes, each of which contains a small amount of memory. Much of the turbo decoder area is similarly filled by the beta scratchpad memories, which could be more easily implemented as 8T SRAMs. The critical path of the combined decoder runs through the variable nodes, shifters, and check nodes, as the design is not pipelined. Note that the turbo interleaver was not synthesized as part of the design.

Throughput estimates are shown in Table II. These estimates assume 10 iterations per codeword, and are based on the largest block size for each code (1944 for LDPC, 6144 for turbo). Note that the turbo decoder can operate at a faster frequency than the LDPC decoder in the current implementation. The energy calculations assume that only the part of the design used for the particular decoding algorithm consumes power, and so use the standalone power results from Table I.

VI. Future Work

Because the area of our current design is dominated by LDPC-specific units, we believe that our LDPC architecture could be optimized further. More variable node groups could be added to improve throughput and allow for a larger clock period. Adding more variable node groups would also increase the available ACS resources in the check nodes, allowing for more turbo windows to be processed in parallel.

TABLE I: Synthesis Results

Design	Area (mm^2)	Clock Period (ns)	Power (mW)
Combined Decoder	1.08	2.05	370
Variable Nodes	0.85	-	301
Standalone LDPC	1.01	2.02	347
Standalone Turbo	0.13	1.04	67

TABLE II: Throughput and Energy

Code	Throughput (Mbps)	Energy (nJ/bit)
LDPC @ 500MHz	820	423
Turbo @ 500MHz	90	370
Turbo @ 1GHz	180	370

The LDPC critical paths occur between the variable nodes and check nodes, passing through the shifters. One simple pipelining solution would be to process two frames at once, alternating between them. This would permit pipeline registers within the shifters, improving throughput substantially.

Another way to share more resources between the two decoders would be to share the resources in the variable nodes. The variable node memories could double as scratchpad memories for the turbo reverse trellis traversals, and some could be implemented as multiported SRAMs, further reducing the area of the design. Sharing some of the arithmetic units from the variable nodes to perform some computations for the turbo decoder, such as the branch metric unit computation, may also be worthwhile, as long as such sharing them does not dramatically increase the complexity of the already large variable nodes.

VII. Conclusion

We designed a combined LDPC/turbo decoder that shares hardware resources to reduce area while still achieving reasonable throughput and efficiency. The area of the design is dominated by the LDPC decoder's variable nodes. We have therefore demonstrated that a lightweight turbo decoder can be incorporated into an LDPC decoder with relatively little overhead.

Acknowledgment

The authors would like to thank Bora Nikolic, Sriram Sundararajan, and Matt Weiner for fruitful discussions and useful advice.

References

[1] I. C. Society, *IEEE Std 802.11n Part 11 Amendment 5: Enhancements for Higher Throughput*.

[2] 3GPP, "E-UTRA: Multiplexing and channel coding (Release 12)," 2013.

[3] Y. Sun, J. R. Cavallaro, Y. Zhu, and M. Goel, "Configurable and Scalable Turbo Decoder for 4G Wireless Receivers."

[4] M. Weiner, B. Nikoli, and Z. Zhang, "LDPC Decoder Architecture for High-Data Rate Personal-Area Networks," pp. 1784–1787, 2011.

[5] T. S. V. Gautham, A. Thangaraj, and D. Jalihal, "Common Architecture for Decoding Turbo and LDPC Codes."