DESIGN AND ANALYSIS OF LDPC DECODERS FOR SOFTWARE DEFINED RADIO

Sangwon Seo, Trevor Mudge

Advanced Computer Architecture Laboratory
University of Michigan at Ann Arbor
{swseo, tnm}@umich.edu

Yuming Zhu, Chaitali Chakrabarti

Department of Electrical Engineering
Arizona State University
{yuming, chaitali}@asu.edu

ABSTRACT

Low Density Parity Check (LDPC) codes are one of the most promising error correction codes that are being adopted by many wireless standards. This paper presents a case study for a scalable LDPC decoder supporting multiple code rates and multiple block sizes on a software defined radio (SDR) platform. Since technology scaling alone is not sufficient for current SDR architectures to meet the requirements of the next generation wireless standards, this paper presents three techniques to improve the throughput performance. The techniques are use of data path accelerators, addition of memory units and addition of a few assembly instructions. The proposed LDPC decoder implementation achieved 30.4 Mbps decoding throughput for the n=2304 and R=5/6 LDPC code outlined in the IEEE 802.16e standard.

Index Terms— LDPC, Min-sum iterative decoding, SDR, SODA, SIMD

1. INTRODUCTION

Low density parity check (LPDC) codes have excellent error correction performance that approaches the Shannon capacity limit [1], [2]. As a result, they have been adopted in many current and next generation wireless protocols such as DVB-S2 and the IEEE 802.16e standard (WiMAX). Decoders used for LDPC codes have high throughput requirements and have been successfully implemented using ASICs and FPGAs [3]. However, the emergence of a wide variety of wireless protocols that are rapidly changing makes custom hardware for these decoders relatively time consuming and expensive to develop.

Software Defined Radio (SDR) is a programmable hardware platform capable of supporting software implementations of wireless communication protocols for physical layers [4]. This paper presents a case study for a LDPC decoder implementation that supports multiple code rates and multiple block sizes on a SDR platform, SODA (Signal-processing On-Demand Architecture). SODA is a multiprocessor architecture, where each processor is equipped with a 32-wide SIMD (Single Instruction Multiple Data) pipeline, a scalar pipeline and scratchpad memories. When the LDPC matrix is represented by structured submatrices, the data-level parallelism can be efficiently handled by the SIMD pipeline. However the current SODA architecture is unable to meet the high decoding throughput and the scalability requirements (multiple block sizes and multiple code rates) of the IEEE 802.16e standard. In this paper we present use of data path accelerators, addition of memory units and addition of a few assembly instructions to address the throughput and scalability requirements. The proposed LDPC decoder implementation achieves 30.4 Mbps decoding throughput for the n=2304 and R=5/6 LDPC code outlined in the IEEE 802.16e standard.

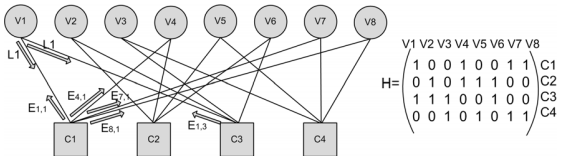

Fig. 1. LDPC matrix H and the corresponding bipartite graph

The rest of the paper is organized as follows. Section 2 gives a brief overview of LDPC codes. Section 3 introduces SODA, the SIMD-based high-performance DSP processor for SDR and mapping of the LDPC decoder onto SODA. Section 4 describes LDPC accelerators, memory controller/buffer organization and assembly support required for the high throughput scalable LDPC decoder implementation. Section 5 presents memory and throughput analysis of the augmented architecture. Section 6 concludes the paper.

2. LDPC BASICS

2.1. Introduction

A LDPC code is a class of linear block codes whose codewords satisfy a set of linear parity-check constraints [1]. These constraints are typically defined by an m-by-n parity-check matrix H, whose m rows specify each of the m constraints (the number of parity checks), and n represents the length of a codeword. H is also characterized by W_r and W_c, which represent the number of 1's in the rows and columns, respec-

tively. A LDPC code can be represented by a bipartite graph, which consists of two types of nodes, Variable Nodes (VN) and Check Nodes (CN). Check node i is connected to variable node j whenever h_{ij} of H is non-zero. Fig. 1 describes the matrix H and the corresponding bipartite graph of a simple LDPC code.

2.2. LDPC Decoding Process

LDPC codes are decoded iteratively using a message passing algorithm [1]. This algorithm involves exchanging the belief information among the variable nodes and check nodes that are connected by edges in the bipartite graph. Let I_n be the intrinsic information from the received signal, L_n be the reliable information for variable node n, $L_{n,m}$ be the information conveyed from variable node n to check node m, and $E_{n,m}$ be the extrinsic information generated in check node m that is passed to variable node n. The belief information is updated in an iterative manner and implemented in two phases. In the first phase, the variable nodes send their belief information, $L_{n,m}$, to check nodes connected to them; in the second phase, the check nodes send the updated belief information (new $E_{n,m}$) to the variable nodes connected to them for updating L_n (See Fig. 1). The iteration steps are summarized in Algorithm 1.

Algorithm 1: Min-sum LDPC Decoding Algorithm

1. Initialization: $E_{n,m}$ = 0, $L_n = I_n$
2. VN to CN: $L_{n,m} = L_n$ - $E_{n,m}$
3. Update $E_{n,m}$: $E_{n,m}^{new} = f(L_{n',m} | n' \in S \subset N(m))$
4. Update L_n: $L_n^{new} = L_{n,m} + E_{n,m}^{new}$
5. Repeat the steps 2,3,4 for NUM iteration times
6. Make a decision of bit n based on the corresponding L_n value

Here, $N(m)$ is the set of variable nodes which are connected with check node m in the bipartite graph. Similarly, $M(n)$ is the set of check nodes which are connected with variable node n. The decoding algorithms differ in how the function f in Step 3 of Algorithm 1 is evaluated.

There are three options for the LDPC iterative decoding algorithm: Belief Propagation (BP), λ-min and min-sum algorithms [5]. Although BP and λ-min algorithms show better error correction performance compared to min-sum algorithm, these algorithms require a look-up table for hyperbolic function values, which requires additional memory space. The min-sum algorithm is selected here because of the limited memory size and easy computation patterns. The min-sum algorithm f is shown as follows. Here, $n' \in N(m)$, $n' \neq n$.

$$E_{n,m}^{new} = - \left(\prod_{n'} sign(L_{n,m})\right) \times \min_{n'} |L_{n,m}|$$

As can be seen, the operations in the min-sum LDPC decoding algorithm are limited to addition, subtraction and finding a minimum value, all of which can be supported by our SDR architecture described in Section 3.

Theoretically, the LDPC decoding process finishes when all parity-check equations are satisfied. In reality, a predefined number of iterations (NUM) based on SNR is generally used.

2.3. LDPC Matrix Partition

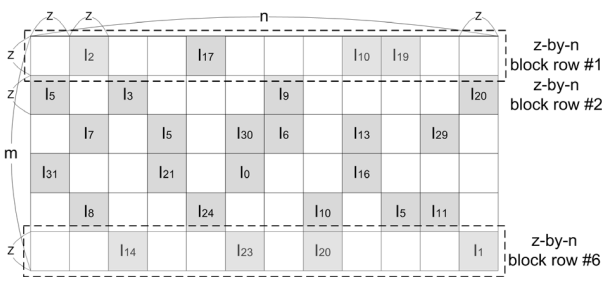

Fig. 2. Partitioning of H into z-by-z cyclic identity matrices

A LDPC matrix H has randomly distributed 1's which results in complex data routing and is a major challenge for building a high-performance and low-power LDPC decoder. [3] and [6] show that introduction of some structural regularity in the matrix does not degrade its error correction performance. Moreover the regularity enables partially parallel implementation of LDPC decoders and has been utilized in the IEEE 802.16e standard. Fig. 2 shows the partitioning of H into z-by-z cyclic identity submatrices. Here, I_x represents a cyclic identity matrix with rows shifted cyclically to the right by x positions. This characteristic reduces the routing overhead and has been exploited efficiently in our architecture. Fig. 2 also shows how the $\frac{n}{z}$ of the identity matrices along a row can be grouped to form a block row. So, in essence, the H matrix can also be partitioned into $\frac{m}{z}$ block rows each of size z-by-n.

3. SDR ARCHITECTURE

In this section, we present the SIMD-based SDR architecture, SODA [4]. This architecture was initially designed to support wireless protocols such as WCDMA and IEEE 802.11a.

3.1. SODA Overview

The SODA multiprocessor architecture is shown in Fig. 3. It consists of multiple data processing elements (PEs), one control processor and a global scratchpad memory, all of which are connected through a shared bus. Each SODA PE consists of five major components: 1) a 32-way, 16-bit datapath SIMD pipeline for supporting vector operations. Each datapath includes one 16-bit ALU with multiplier and a 2 read-port, 1 write-port 16 entry register file. Intra-processor data movements are supported through a SIMD Shuffle Network

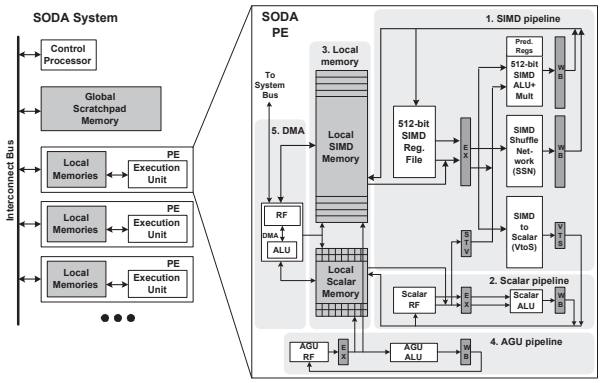

Fig. 3. SDR architecture: SODA [4]

(SSN); 2) a 16-bit datapath scalar pipeline for sequential operations. The scalar pipeline executes in lock-step with SIMD pipeline; SIMD-to-scalar and scalar-to-SIMD operations exchange data between two pipelines; 3) two local scratchpad memories for the SIMD pipeline and the scalar pipeline; 4) an AGU (Address-Generation-Unit) pipeline for providing the addresses for local memory accesses; and 5) a programmable DMA (Direct-Memory-Access) unit to transfer data between scratchpad memories and interface with the outside system (inter-processor data transfer). The SIMD pipeline, the scalar pipeline and the AGU pipeline execute in VLIW-styled lock-step manner, controlled with one program counter (PC) [4].

3.2. LDPC on SODA

The min-sum LDPC decoding algorithm (Algorithm 1) is map ped onto SODA in the following way. Step 2 of Algorithm 1 is applied to non-zero z-by-z submatrices. However, because Step 3 uses the $L_{n,m}$ values related with check node m, the SIMD pipeline loads z values of type L_n and aligns the data in check node order by using SSN before executing Step 2. The shuffled $L_{n,m}$ values for all non-zero z-by-z submatrices in one z-by-n block row are calculated in the SIMD datapath. After that, the SIMD-to-Scalar unit is used for finding the minimum $E_{n,m}^{new}$ among W_r of $L_{n,m}$ values for the same check node m. Next, $E_{n,m}^{new}$ and the corresponding sign indicator are used to update a L_n value (Step 4). This procedure implies that some SIMD slices execute additions and others execute subtractions based on sign values – a feature that is supported by predicated instructions in SODA. After updating the L_n values, the data is inversely shuffled and stored in variable node order. This process is repeated for every z-by-n block row in every iteration.

4. SCALABLE LDPC IMPLEMENTATION

In this section, we study a scalable LDPC decoder implementation for block size n, code rate R=k/n, and (W_c, W_r)-LDPC code as specified by the IEEE 802.16e standard on a SODA PE. We describe the enhancements that had to be made in terms of accelerators, memory units, and new assembly instructions to support multiple code rates and multiple block sizes. Fig. 4 shows the modified SIMD pipeline – the additional units have been shown using shaded blocks.

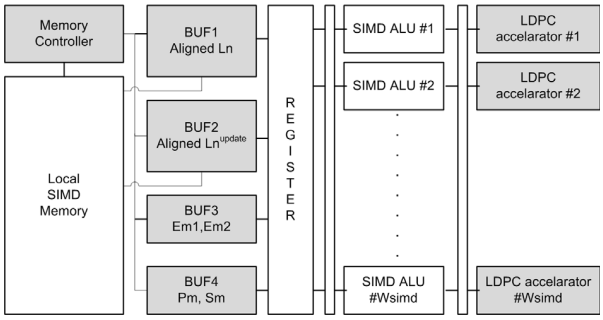

Fig. 4. Modified SIMD pipeline in a SODA PE

4.1. LDPC Accelerator

In order to meet the high decoding throughput requirements, we introduce a LDPC accelerator in every SIMD slice as shown in the Fig. 4. There are only two possible $E_{n,m}^{new}$ values for check node m in Step 3 of Algorithm 1 (which are selected from W_r values of type $L_{n,m}$): the minimum E_{m1} and the second minimum E_{m2}. Each LDPC accelerator expedites finding the minimum values using two compare/store units with two W_r-bit special registers, a selection register P_m and a sign register S_m, as can be seen in Fig. 5. The operation of the LDPC accelerator is summarized below.

The Algorithm of LDPC Accelerator

```
if (Ln,m <= Em1) \\ operations in Cmp&Store 1
{
  Em1 <= Ln,m; Em2 <= Em1;

  if (Ln,m < Em1) Pm = 1 << i; else Pm = 0;
}
else if (Ln,m < Em2) \\ operations in Cmp&Store 2
{
  Em2 <= Ln,m;
}
Sm = (Sm | sign(Ln,m)) << 1;
```

E_{m1}, E_{m2}, P_m and S_m are extracted using a flush signal and these values are used to compute $E_{m,n}$ using the following operation (Step 7 and 14 of Algorithm 2).

$$\text{if } (P_m[i] == 1)\ E_{m,n}[i] = (S_m[i])\ E_{m1}, \text{ else } E_{m,n}[i] = (S_m[i])\ E_{m2}$$

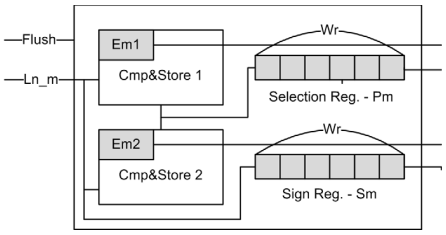

Fig. 5. LDPC accelerator

4.2. Memory Units

A major challenge in decoding LDPC codes is the large number of data alignment operations required for every z-by-z permutation matrix. z values of type L_n need to be shuffled so that they can be correctly aligned for check node processing. If z is less than the SIMD width (W_{simd}), the data alignment can be executed in one clock cycle using SSN. However, the IEEE 802.16e standard uses different z values (24, 28, 32, ..., 96) for different block sizes [7]. If z is larger than W_{simd}, many clock cycles are required for data alignment operation when SSN is used. This causes a degradation in the LDPC decoding throughput performance.

To solve the alignment issue, we propose a memory controller and buffer organization (instead of using the shuffle network) as shown in Fig. 4. BUF1 and BUF2 contain aligned L_n and L_n^{update} (to be described in Section. 4.3) respectively; BUF3 contains E_{m1} and E_{m2}; and BUF4 contains P_m and S_m.

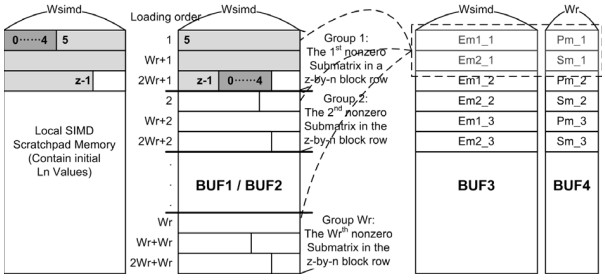

Fig. 6. Data alignment in buffers

The memory controller handles movement of L_n data between the SIMD memory and BUF1. Since the z-by-z permutation matrices in the LDPC codes used in the IEEE 802.16e standard are circular right-shifted identity matrices, each permutation matrix can be defined by a single right-shifted amount s. The alignment operation can now be achieved by two memory copy operations described below. If the shifted amount is s and the start memory address is m_{start}, the memory controller first copies MEM[$m_{start} + s$ ... $m_{start} + z - 1$] to BUF1, and then copies MEM[m_{start} ... $m_{start} + s - 1$] to BUF1. This is shown in Fig. 6 for an example where s=5, m_{start}=0. This is done for all non-zero W_r submatrices in a z-by-n block row. At the end of this process, BUF1 contains W_r groups of aligned L_n data (see Fig. 6). In a similar way, the memory controller fills BUF2 for L_n^{update} data with another shift amount (($s - s^{update}$) mod W_{simd}) (to be described in Section. 4.3). Note that the width of BUF1 and BUF2 is W_{simd}.

4.3. Modified Decoding Algorithm

Algorithm 2 shows the LDPC decoding algorithm on the modified SODA architecture. The L_n and L_n^{update} values are aligned and stored in BUF1 and BUF2 (Steps 1 and 2 of Algorithm 2). The aligned values of L_n and L_n^{update} (Step 5) along with E_{m1}, E_{m2} (Step 4), P_m and S_m (Step 6) of the first row of the first group (see for example Group 1 in Fig. 6) are fed to the ALU unit and LDPC accelerator in each SIMD slice. These values are updated in Steps 7, 8, 9 of Algorithm 2. The process is repeated for the first row of the next group (see for example Group 2 in Fig.6), and so on. After completing processing of all the first rows of all the W_r groups (Step 10), the updated values of E_{m1}, E_{m2}, P_m and S_m are stored in their respective buffers (Steps 11, 12). The updated values are used to compute $E_{m,n}^{new}$ and L_n^{update} (Step 15, 16) of the first row of each W_r group (Step 17). The process is repeated for the second row of each W_r group, and so on (Step 18). The above schedule results in high decoding throughput performance; it reduces the number of data switches and also speeds up the operation of finding the minimum values in the min-sum decoding algorithm. After processing all the data for one z-by-n block row, the data for the next z-by-n block row is loaded into BUF1 and BUF2, and the process repeats the number of z-by-n block rows(=$\frac{n(1-R)}{z}$) times.

Algorithm 2: LDPC decoding algorithm
in the modified SODA PE

1. load aligned L_n to BUF1
2. load aligned L_n^{update} to BUF2
3. load W_r for the current z-by-n block row
4. load E_{m1}, E_{m2} from BUF3
5. load L_n, L_n^{update} from BUF1, BUF2
6. load P_m, S_m from BUF4
7. compute $E_{m,n}^{curr}$ using E_{m1}, E_{m2}, P_m, and S_m.
8. update $L_{n,m}$ = L_n + L_n^{update} - $E_{m,n}^{curr}$
9. update E_{m1}, E_{m2}, P_m, and S_m using $L_{n,m}$
10. repeat step 5 to step 9 W_r times
11. store updated E_{m1}, E_{m2} (E_{m1}^{new},E_{m2}^{new}) in BUF3
12. store updated P_m, S_m (P_m^{new}, S_m^{new}) in BUF4
13. load L_n^{update} from BUF2 again
14. compute $E_{m,n}^{new}$ using E_{m1}^{new}, E_{m2}^{new}, P_m^{new}, and S_m^{new}

15. update L_n^{update} += $E_{m,n}^{new}$
16. store updated L_n^{update} in MEM
17. repeat step 12 to step 16 W_r times
18. repeat step 4 to step 17 $\lceil \frac{z}{W_{simd}} \rceil$ times
19. repeat step 1 to step 18 $\frac{n(1-R)}{z}$ times.
20. repeat step 1 to step 19 NUM times.

In order to reduce the memory for storing $L_{n,m}$, we introduce the parameter L_n^{update}, which is (-$E_{n,m}$ + $E_{n,m}^{new}$). In fact, the memory space is reduced by a factor of m by keeping one L_n^{update} value for each check node n instead of storing all $L_{n,m}$ values for every n and m combination.

Since updated L_n^{update} values are processed in check node order, inverse alignment operation is required to store the data in variable node order in memory. After L_n^{update} is stored back in memory, for the next z-by-n block row computation, the data is realigned with a different shift amount. However, these two alignment operations can be reduced to one alignment operation using another shift amount s^{update}; instead of inverse alignment operation, L_n^{update} is stored with the current shifted amount s^{update} and then, in the next iteration, the memory controller use (($s - s^{update}$) mod W_{simd}) as a shift amount to align L_n^{update}.

4.4. Assembly Support

New assembly instructions are required for the proposed architecture to improve the decoding throughput performance. Steps 1 and 2 of Algorithm 2 are independent and can be executed in parallel. These are combined to form instruction $ldpc\_mem2buf$. Similarly steps 5 and 6 of Algorithm 2 can be executed in parallel and combined to form instruction $ldbufs$. Steps 8 and 9 of Algorithm 2 can be executed in a pipelined manner through the ALU unit and the LDPC accelerator unit. We combine these two instructions and introduce a macro-operation instruction, $ldpc\_in$. To implement steps 11 and 12 of Algorithm 2, the new instruction, $ldpc\_out.(vp)$, is introduced to flush E_{m1}, E_{m2}, P_m, and S_m from LDPC accelerators and store them in BUF3 and BUF4. The additional new assembly instructions are listed below.

The New Assembly Instructions

1. $ldpc\_mem2buf$ Addr[Mem],Addr[BUF1],Addr[BUF2],S1,S2
   : send a control signal to the memory controller
   : the controller loads L_n,L_n^{update} from a memory and aligns the data with shift amounts (S1, S2) in BUF1 and BUF2
2. $ldbuf3$ V3,V4,Addr[BUF3]
   : load V3=E_{m1}, V4=E_{m2} from BUF3
3. $ldbufs$ V1,V2,P1,P2,Addr[BUF1],Addr[BUF2],Addr[BUF4]
   : load V1=L_n, V2=L_n^{update}, P1=P_m, P2=S_m from BUF1, BUF2, BUF4
4. $ldpc\_in$ V1,V6
   : 1) calculate $L_{n,m}$ with V1=L_n and V6=$L_n^{update} - E_{m,n}^{curr}$
   : 2) update E_{m1},E_{m2},P_m,S_m in LDPC accelerators with $L_{n,m}$.
5. $ldpc\_out.v$ V7,V8,Addr[BUF3]
   : extract V7=E_{m1}, V8=E_{m2} from LDPC accelerators and store them in BUF3
6. $ldpc\_out.p$ P3,P4,Addr[BUF4]
   : extract P3=P_m, P4=S_m from LDPC accelerators and store them in BUF4

The overhead of adding these new instructions is the increased instruction bit width and the instruction decoder complexity.

4.5. Scalability Issues

The proposed architecture supports different values of z and W_r corresponding to the different code sizes and code rates mandated by the IEEE 802.16e standard. The memory configuration described in Section 4.2 handles the more difficult case of when $z > W_{simd}$. Larger z results in more computations and so a larger W_{simd} would help in achieving higher decoding throughput. The penalty is the larger area, both is terms of datapath and memory, and larger power. The parameter W_r affects the decoding throughput (number of iterations in Algorithm 2). Since it also affects the buffer size and P_m, S_m registers in the LDPC accelerators, the architecture has to be designed for the largest value of W_r.

5. ANALYSIS

In this section, we study the required memory and buffer size, and also analyze the improvement in the decoding throughput due to the memory organization, datapath accelerators and assembly instruction support.

5.1. Memory Size Analysis

LDPC decoding process consists of computationally simple operations and multiple memory operations. As a result, if the memory is not organized properly, then it is highly likely that the SIMD pipeline would have to wait for the data to arrive. In a typical implementation, there are four main values that are to be stored: L_n, $L_{n,m}$, $E_{n,m}$, and shuffle information. For n=2304 and R=5/6 LDPC codes outlined in the IEEE 802.16e standard, a brute-force decoding method needs 3.456GB for storing the $L_{n,m}$ and $E_{n,m}$ values. Even if we consider only non-zero elements, the storage still requires 30KB (15KB+15KB), which is a still large memory space for an SDR platform. Therefore, a new scheme to reduce memory space should be considered. There is no way to reduce the storage of L_n because the data is used to decide the final decoded bit value. However, the storage for $L_{n,m}$ and $E_{n,m}$ can be significantly reduced.

To reduce $E_{n,m}$ storage size, we exploited the fact that there are only two possible $E_{n,m}^{new}$ values for check node m: E_{m1} and E_{m2}. This two-minimum method reduces the required memory space by a factor of W_r/2. For the case mentioned above, the storage requirement for $E_{n,m}$ values is reduced to 1.5KB. Also, instead of storing all $L_{n,m}$ values, we store L_n^{update} values, thereby reducing the storage by a factor of m(=4) to 3.75KB.

Storage	Size(B)	Ex.(KB)
MEM: L_n, L_n^{update}	$4n$	9
BUF1: L_n	$2W_{simd}W_r\lceil\frac{z}{W_{simd}}\rceil$	3.75
BUF2: L_n^{update}	$2W_{simd}W_r\lceil\frac{z}{W_{simd}}\rceil$	3.75
BUF3: E_{m1}, E_{m2}	$4W_{simd}\lceil\frac{z}{W_{simd}}\rceil\frac{n(1-R)}{z}$	1.5
BUF4: P_m, S_m	$2W_r\lceil\frac{z}{W_{simd}}\rceil\frac{n(1-R)}{z}$	0.94

Table 1. Memory/Buffer requirements for n=2304 and R=5/6 LDPC code in the IEEE 802.16e standard

Table 1 summarizes the memory and buffer requirements for a block size n, code rate R=k/n, and (W_c,W_r)-LDPC code. We list the memory requirements for n=2304 and R=5/6 LDPC code (the IEEE 802.16e standard) when W_{simd} = 32, W_r = 20, and $z = 96$ under the column 'Ex.' in the table.

5.2. Throughput Analysis

The data path accelerators, the memory units, and the new instructions all help in increasing the decoding throughput. For the n=2304 and R=5/6 LDPC code in the IEEE 802.16e standard and for NUM=10, the achievable clock cycle reductions for each of the enhancements are shown in Table 2. Here 40000 is the number of cycles in the original SODA implementation.

	Redn. (Cycles)	% redn.
LDPC Accelerators	5760(40000)	14.4
Memory Units	6912(40000)	17.3
New Instructions	4608(40000)	11.5

Table 2. Cycle reductions due to enhancements

The proposed SODA PE is implemented in 0.18um technology and is clocked at 400MHz. The LDPC decoding throughput for n=2304 and R=5/6 LDPC code can be boosted from 18.3 Mbps to 30.4 Mbps using the proposed enhancements. With technology scaling, the decoding throughput is expected to increase to around 62.2 Mbps in 90nm technology.

The area and power overhead in the datapath and memory is quite small. For instance the area of the memory controller and LDPC accelerators is negligible (5.37%) compared to the original design. However the complexity of adding CISC-type instructions requires careful evaluation.

6. CONCLUSION

In this paper, we presented a software-hardware co-design case study of LDPC decoder for SDR. We first provided an overview of LDPC codes and then showed how LDPC decoding can be done by the SDR architecture. Next we showed how use of datapath accelerators, memory buffers and additional instructions can be used to improve the decoding throughput performance. We implemented a scalable LDPC decoder for the IEEE 802.16e standard. Our results show that we can achieve 30.4 Mbps decoding throughput for n=2304 and R=5/6 LDPC code.

7. ACKNOWLEDGEMENT

This research is supported in part by ARM Ltd., NSF CSR-EHS 0615135, NSF ITR 0325761 and The Korea Foundation for Advanced Studies.

8. REFERENCES

[1] Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. IT-8, no.1, pp. 21–28, January 1962.

[2] D.J.C.MacKay; R.M.Neal, "Near shannon-limit performance of low-density parity-check codes," *Electronics letters*, vol. 32, pp. 1645–1646, August 1996.

[3] M.M.Mansour; N.R.Shanbhag, "High-throughput ldpc decoders," *IEEE Transactions on VLSI Systems*, vol. 11, no.6, pp. 976–996, December 2003.

[4] Y.Lin et. al., "Soda: A low-power architecture for software radio," *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[5] F.Guilloud; E.Boutillon; J.L.Danger, "λ-min decoding algorithm of regular and irregular ldpc codes," *3rd International Symposium on Turbo Codes & related topics*, September 2003.

[6] D.E.Hocevar, "A reduced complexity decoder architecture via layered decoding of ldpc codes," *IEEE Workshop on Signal Processing Systems*, pp. 107–112, 2004.

[7] "IEEE Std 802.16e-2005," *available at http://standards.ieee.org/getieee802/download/802.16e-2005.pdf*, February 2006.