Implementation of an LDPC decoder for IEEE 802.11n using VivadoTM High-Level Synthesis

ERNEST SCHEIBER, GUIDO H. BRUCK, PETER JUNG
Department of Communication Technologies
University of Duisburg-Essen
Duisburg, Oststr. 99, GERMANY
Email: ernest.scheiber@uni-due.de

Abstract—The increasing complexity of hardware designs calls for design methodolgies that use more abstract design entries and increased automation of the implementation process. Highlevel synthesis (HLS) has been a research topic for the past 20 years, and current tools, such as Xilinx VivadoTM HLS promise to bring HLS to widespread use. In this paper we use Xilinx VivadoTMHLS to design an LDPC decoder for 802.11n. Forward error correction decoders are typically implemented in hardware due to the high processing requirements and therefore an LDPC decoder is an appropriate example to demonstrate the power of high-level synthesis.

Keywords—High-level synthesis, FPGA, LDPC, IEEE 802.11n

1. Introduction

Low Density Parity Check (LDPC) codes were introduced by Robert G. Gallager in 1963 in his doctoral thesis [1] and since 1990s LDPC codes have been used in different communication standards. Hardware implementations are usually preferred, due to the high processing requirements of LDPC decoding. In software-defined radio systems FPGAs offer the flexibility and high processing power required to make implementations feasible. The proliferation since the 1980s of RTL (Register Transfer Level) language standards for digital design in combination with automatic tools for logic synthesis and implementation have promoted design productivity. Design tools such as Xilinx Vivado™ HLS permit hardware design at an even higher abstraction level and promise to further increase design productivity.

In this paper we show how Xilinx Vivado™ HLS can be used to implement an LDCP decoder for IEEE 802.11n. The paper is organized as follows. Section II gives an overview of the LDPC decoding algorithm. Section III presents the IEEE 802.11n LDPC structure, the decoder architecture and the implementation method. In section IV we give the resulting throughput of the decoder, the performance of decoder in terms of bit error ratio (BER) as a function of E_b/N_0 and the FPGA resources required by the LDPC decoder.

2. Technology Overview

2.1. Low Density Parity Check Codes

We define N as the length of a codeword $\mathbf{x}$, K as the number of the information bits and $M = N - K$ the number of redundancy bits in the codeword. LDPC codes are defined by their parity check matrix $\mathbf{H}$ - a sparse $M \times N$ matrix, satisfying the equation

$$\mathbf{Hx} = \mathbf{0} \tag{1}$$

for each codeword $\mathbf{x}$.

Optimal, maximum a-posteriori decoding of LDPC codes is a practically unfeasible problem. As an alternative an iterative belief propagation algorithm is used that allows decoding close to the Shannon limit. The LDPC decoding algorithm runs on a bipartite graph, with M edges corresponding to each parity check equation, called check nodes, and N edges corresponding to each component of a codeword, called variable nodes. The vertices connect variable and check nodes according to the equations defined by the parity check matrix.

The inputs to the decoding algorithm are log-likelihood ratios (LLR) for each variable node, as defined in equation (2), where x_n are the components of the sent codeword and y_n are the received values.

$$L_n = \ln \frac{P\{x_n = 1|y_n\}}{P\{x_n = -1|y_n\}} = \ln \frac{1 + E_n}{1 - E_n} \tag{2}$$

Parity check equations allow the calculation of extrinsic log-likelihood ratios for each factor in the equation. Let $\mathcal{M}(m)$ be the set of indices of the variable nodes connected to the m-th check node, then it can be shown that equation (3) holds, where the values E are the expectation of x_n conditioned on y_n.

$$E^{ext}_{j \in \mathcal{M}(m)} = \prod_{i \in \mathcal{M}(m)/j} E_i \tag{3}$$

In Fig. 1 the expectation E is plotted as a function of the LLR L. The magnitude of the expectation is subunitary and a value close to zero denotes high uncertainty in the value of the variable, while a value close ± 1 low uncertainty. According to equation (3), the sign of the extrinsic value E^{ext} is given by the product of the signs of all the contributing factors E_i. The certainty given by E^{ext} is smaller than any certainty of an any input factor. The messages sent between check and variable nodes in the decoding process are LLRs. To avoid converting between LLRs and expectations the approxiamtion given by equation (4) is used. The certainty of E^{ext} will be given by the lowest certainty among the input values E. In equation (4)

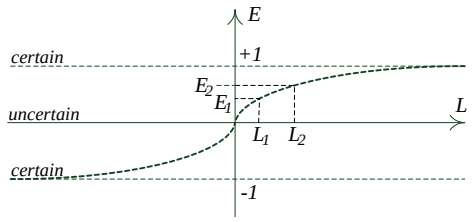

Fig. 1. The expectation E as a function of the log-likelihood ratio L

c_{mn} and v_{im} are the check node message from check node m to variable node n and the variable node message from variable node i to check node m, respectively.

$$c_{mn} = \prod_{i \in \mathcal{M}(m)/n} \operatorname{sgn}\{v_{im}\} \min_{i \in \mathcal{M}(m)/n} |v_{im}| \tag{4}$$

Variable nodes combine the extrinsic information from the check nodes and the input LLRs to generate messages for check nodes in the next iteration according to equation (5).

$$v_{nm} = L_n + \sum_{i \in \mathcal{N}(n)/m} c_{mn} \tag{5}$$

The decoding algorithm alternates between stages of check node processing and variable node processing until all parity check equations are satisfied or until the maximum number of iterations has been reached.

2.2. High-Level Synthesis with Xilinx Vivado™ HLS

Since the 1980s RTL based design has been the preferred method of design of digital systems. Further productivity gains are facilitated by design reuse through the proliferation of IP (Intellectual Property) cores. Beginning with the 1990s research was conducted on high-level synthesis i.e. hardware design that abstracts more details of the underlying hardware and allows programmers to focus on algorithm development in a C-like programming language. While in the past 20 years HLS design has not seen widespread adoption, design tools have improved to change that [2].

Vivado™ HLS is a Xilinx tool that introduces HLS design to Xilinx FPGAs. The design input is C,C+ or SystemC code. Design directives guide the synthesis process. Design directives can refer to loops, interfaces, arrays etc. The user directs the synthesizer to pipeline or unroll loops, defines interface types and partitions and reshapes arrays to maximize throughput. The output of Vivado HLS is synthesizable Verilog, VHDL and SystemC code that can be used to implement the design in hardware [3].

3. Decoder Implementation

3.1. IEEE 802.11 LDCP codes

IEEE 802.11n defines three LDPC code lengths (648, 1296, 1944) and four code rates (1/2, 2/3, 3/4, 5/6) for a total of 12 possible codes [4]. Each code is defined by a parity check matrix that is formed out of square submatrices of size 27, 54 or 81 for the three defined code lengths. In Fig. 2 the parity check matrix for length 648 and code rate 1/2 is given. Each entry in the table represents a 27 by 27 square matrix, where 0 represents the identity matrix and any other number represents a cyclic shift to the right of the identity matrix by a number of places equal to the number. A horizontal bar represents an all zero matrix.

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. Parity check matrix for code length 648 and rate 1/2

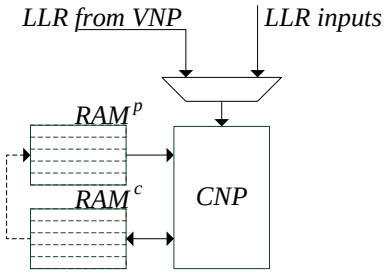

Fig. 3. Check node processor

3.2. Decoder architecture

The hardware architecture of the LDPC decoder follows the blueprint defined in [5]. In Fig. 3 the structure of a check node processor (CNP) is depicted. There are 12 CNPs, operating in parallel, used for decoding the rate 1/2 code. Each CNP is responsbile for a row of parity check equations from the matrix defined in Fig. 2. For each parity check equation the CNP computes the two minimum absolute value of the input LLRs, the index of the minimum absolute value LLR, the XOR of all the signs of the input LLRs and stores the sign of each input LLR. Each CNP uses two separate RAMs, as depicted in Fig. 3, RAM^c is used to store the results of the current iteration while RAM^p holds the results of the previous iteration.

The CNP operates in the first iteration on the input LLRs and on the LLRs computed by the variable node processor (VNP) in all subsequent iterations. The CNP uses RAM^p, holding the results of the previous iteration, to adapt the messages to each variable node according to equation (5) by substracting the term corresponding to the destination check node.

The VNP iterates through all variable nodes, selects the appropriate message out of each of the 12 RAMs holding the results of the CNPs and together with the input LLRs adds

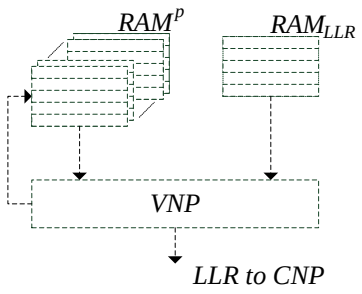

Fig. 4. Variable node processor

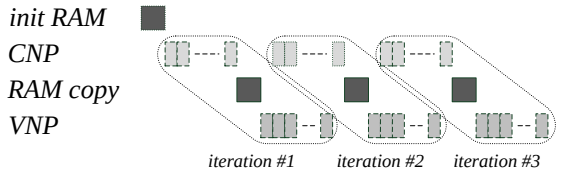

Fig. 5. Scheduling of operations for LDPC decoding

them all up. The results of the VNP flow directly into the CNP for the next iteration.

The scheduling of operations in the decoding process are depicted in Fig. 5. The RAMs used for the results of the CNPs are initialized before the decoding of each block. Each iteration has three stages: CNP processing, copying of data from RAM^c to RAM^p and reinitialization of RAM^c, and finally VNP procesing. The VNP processing of an iteration and the CNP of the next iteration run in parallel such that the computed LLRs from the VNP are fed directly into the CNPs.

3.3. HLS coding

In the following a pseudo code of the main body of the implementation of the decoder is given. Some details are omitted to highlight the essential steps. The code is structured so as to generate the scheduling of operations as depicted in Fig. 5.

1: Init RAM^p, RAM^c
2: **for** $iter = 0$ to no_iter **do**
3: **for** $n = 0$ to $647 + 27$ **do**
4: $n_b = n/27, n_i = n\%27$
5: // VNP
6: **if** $(iter > 0)\&\&(n_b < 24)$ **then**
7: **for** $c_b = 0$ to 11 **do**
8: $shift \leftarrow P[c_b][n_b]$
9: $i_c \leftarrow$ var_to_check$(n_i, shift)$
10: $c_m[c_b] \leftarrow$CM$(RAM^p[c_b][i_c])$
11: **end for**
12: $v[n_i] \leftarrow LLR[n] + \sum_{c_b=0}^{11} c_m[c_b]$
13: **end if**
14: // CNP
15: **if** $(iter < no\_iter)\&\&(n_b > 0)$ **then**
16: **for** $c_b = 0$ to 11 **do**
17: $shift \leftarrow P[c_b][n_b]$
18: $i_v \leftarrow$ check_to_var$(n_i, shift)$
19: $v_{upd} \leftarrow v_m[i_v] -CM(RAM^p[c_b][n_i])$
20: UpdateRAM$(v_{upd}, RAM^c[c_b][n_i])$
21: **end for**
22: **end if**
23: // Pass data from VNP to CNP
24: **if** $(iter == 0)\&\&(n_b < 24)$ **then**
25: $v_m[n_i] \leftarrow LLR[n]$
26: **end if**
27: **if** $(iter > 0)\&\&(n_b < 24)\&\&(n_i == 26)$ **then**
28: **for** $i_v = 0$ to 26 **do**
29: $v_m[i_v] \leftarrow v[i_v]$
30: **end for**
31: **end if**
32: **end for**
33: // RAM copy
34: $RAM^p \leftarrow RAM^c$
35: Init RAM^c
36: **end for**

The main loop (line 2) performs the maximum predefined number of iterations of the decoder: no_iter. The loop itself runs no_iter+1 times, since in the first iteration only the CNP is run and in the last iteration only the VNP (see Fig. 5). The loop starting at line 3 iterates through all the variable nodes n. Similarly to the main loop, this loop, too goes beyond the number of variables (i.e. 648) by the 27 (i.e. the length of one block in the parity check matrix). This accounts for the delay between the VNP processing and the CNP processing which can also be seen in Fig. 5. n_b and n_i are the block and the in-block indices corresponding to the variable node n.

The VNP is implemented between lines 6 and 13. For each variable node n the check node messages are gathered from RAM^p and added to $LLR[n]$ according to equation (5). The CNP will then substract the appropriate check node message to implement equation (5) exactly. All iterations of the VNP for loop can be executed simultaneously in parallel, therefore the UNROLL directive is placed on this loop.

The CNPs are implemented between lines 15 and 22. Each variable node message can at most affect 12 check nodes due to the structure of the parity check matrix (see Fig. 2). The for loop starting at line 16 performs the 12 check node updates. The directive UNROLL is placed on this loop as well since the operations can run in parallel.

To maximize throughput we place the directive PIPELINE with iteration interval 1 on the variable loop (i.e. line 3). The initiation interval constraint triggers a warning upon failure to fulfill the requirement. Pipelining has to take into account data dependencies between loop iterations. The CNP reads and updates the entries of the RAM^c. Pipelining can only be realized if the write operation from an iteration of the loop is executed before any read of the same RAM entry in subsequent iterations. In such a case data dependecies between iterations are false and this must be signalled to the Vivado HLS software by using the DEPENDENCE directive on the RAM^c variable with parameter false. This dependence is only guaranteed to be false if the check nodes are updated in the same order in all blocks. This is made possible by the delay between the VNP and CNP processing. In this way 27 v values are gathered and passed for processing to the VNP (pseudocode lines 24 to 31)

in a group. The VNP can then use the v_m in whatever order the parity check matrix P dictates so that the check nodes are always processed in ascending order.

The two RAMs: RAM^p and RAM^c containing the CNP results are coded as double arrays of the type t_check_node:

```
t_chck_node RAMp[12][27];
t_chck_node RAMc[12][27];
```

The data type t_check_node is defined as:

```
typedef struct {
        sc_uint<7> min1;
        sc_uint<7> min2;
        bool xor_all;
        sc_bv<24> sign;
        sc_uint<5> min_idx;
} t_chck_node;
```

The programmer can take advantage of the C/C++ languages capabilities to organize data according to the requirements of the application. For data of arbitrary bit widths Vivado HLS allows multiple approaches depending on the language entry used i.e. C, C++ or SystemC. We have used the SystemC data types (i.e. sc_uint) which can be used with the C++ compiler without the need to link the whole SystemC simulation kernel. By default Vivado HLS implements a separate RAM for each member of the struct, to override this we place the DATA_PACK on RAM^p and RAM^c. RAM^p and RAM^c are two-dimensional arrays, they are nevertheless implemented as one memory block by HLS. As a single memory block they do not allow the unrolling operations described before. To split the memories into 12 separately addressable blocks the ARRAY_PARTITION directive is used on the two variables RAM^p and RAM^c.

4. Results and Performance

4.1. Decoder throughput

The decoder throughput is a function of the operating clock frequency of the design f_{clk}, the number of information bits per frame N_{bits}, and the decoding latency for one frame in number of clock cycles $N_{latency}$. The decoding latency is a function of the maximum number of iterations employed. For three iterations the decoding latency is 2951 cycles. The generated design files where used to implement the design on a Spartan6 LX150T device. The resulting minimum period is 8.152 ns corresponding to an operating frequency of f_{clk}=122 MHz. The number of information bits per frame N_{bits} is 324 since the decoder works for codes of length 648 and rate $1/2$. The resulting throughput is 13.4 Mbit/s.

4.2. Bit error rate of LDPC decoder

The performance of the LDPC decoder has been determined through software Monte Carlo simulation of the HLS C++ code. The simulation results are plotted in Fig. 6 for 3 and 5 decoding iterations.

4.3. Resource utilization

As mentioned in section IV-A the decoder was implemented on a Spartan6 LX150T device. The resource utilization of the decoder is given in Table I.

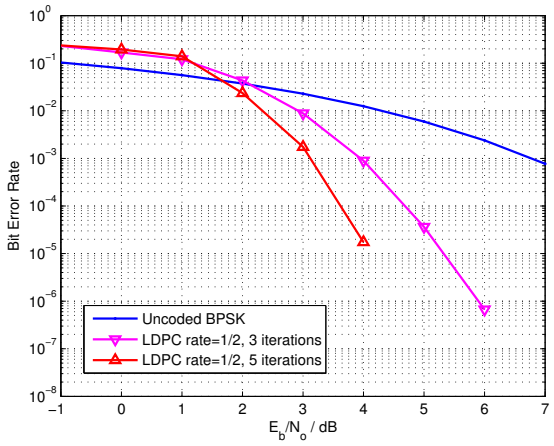

Fig. 6. Bit error rate of LDPC decoder

TABLE I. RESOURCE UTILIZATION FOR THE LDPC DECODER ON A SPARTAN-6 LX150T FPGA

Resource	Utilization	Total	Percentage
Slice registers	4272	184304	2
Slice LUTs	9072	92152	3
Total slices used	3232	23038	14
Block RAMs	56	268	20.9

5. Conclusion

In this paper we have presented a methodology to design an LDPC decoder using High-Level Synthesis technology from Xilinx. High-Level Synthesis is a powerful technology and we have shown that non-trivial designs can be created using this technology. The resulting decoder has a relatively low data throughput. Improvements may be obtained by floorplanning, since the longest path runs 82.2% through routing and only the remaining 17.8% runs through logic. Different architectures can also be investigated that have a higher level of parallelism as described in [6].

References

[1] R. G. Gallager, *Low Density Parity Check Codes*. M.I.T. Press, 1963.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[3] Xilinx, *Vivado Design Suite User Guide High-Level Synthesis*, Xilinx.

[4] "IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std. 802.11-2012*, 2012.

[5] J. Cho, N. Shanbhag, and W. Sung, "Low-power implementation of a high-throughput LDPC decoder for IEEE 802.11n standard," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, 2009, pp. 040–045.

[6] M. Peyic, H. Baba, E. Guleyuboglu, I. Hamzaoglu, and M. Keskinoz, "A low power multi-rate decoder hardware for IEEE 802.11n LDPC codes," *Microprocessors and Microsystems*, vol. 36, no. 3, pp. 159 – 166, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933111001281