A FPGA Implementation of Rate 1/2, 8088-bit Irregular Low Density Parity Check Decoder

Yanni Chen
Department of Electrical and Computer Engineering
University of Minnesota, Minneapolis, MN.
Email: ynchen@ece.umn.edu

Dale Hocevar
DSP Solutions R & D Center
Texas Instruments Inc., Dallas, TX.
Email: hocevar@ti.com

***Abstract*— This paper presents a FPGA implementation of irregular low density parity check decoder. The considered low density parity check code has code rate 1/2, codeword length of 8088 bits and parallel factor of 24. The partly parallel structure, memory management, message alignment and addressing generation schemes needed to realize the underlying graph connectivity will be discussed. With the target FPGA device Xilinx XC2V8000 and maximum number of 25 iterations, the *information* decoding throughput could achieve up to 40Mbps.**

I. Introduction

Low density parity check (LDPC) codes were first invented in 1962 [1] and rediscovered recently [2]. In terms of coding gain, it has been shown that LDPC codes are asymptotically superior to turbo codes [3]. With respect to the decoding complexity, LDPC decoder could be less computational even with more iterations. However, the main advantage of LDPC codes comes from the increased degree of parallelism that can be applied to many high data rate applications such as magnetic storage and wireless communications. The VLSI implementation results using ASIC and FPGA are also reported in [4] and [5], respectively.

Since better LDPC codes are both irregular and random, the code construction we considered is called irregular partitioned permutation (IPP) method [6]. It demonstrates comparable decoding performance to 3GPP Turbo codes. For instance, for rate 1/2 and codeword length of 5784 bits with maximum 50 decoding iterations, bit error rate of 10^{-6} can be achieved at the SNR of 1.4dB over AWGN channel. In addition, the flexibility of its architecture makes it easily adaptable to different code rates and block sizes.

In this paper, the VLSI design issues of the IPP constructed LDPC codes are discussed, which include partly parallel architecture design, the memory management and alignment schemes to realize the underlying graph connectivity. The structure of this paper is as follows. In Section 2, we first briefly review the code construction and iterative decoding algorithm for LDPC. Section 3 presents the block diagram of partly parallel decoder and detailed architecture of each component. Alignment and memory addressing generation schemes are also presented in this section. Synthesis results of FPGA implementation are discussed in Section 4. Finally some conclusions are drawn in Section 5.

II. Encoding and Decoding of LDPC codes

In this section, the irregular partitioned permutation code construction and its belief-propagation decoding algorithm will be briefly explained.

A. Irregular Partitioned Permutation Method

Since LDPC codes could be uniquely defined by its parity check matrix H, irregular partitioned permutation method based on the work in [7] is to construct irregular H, i.e., either the row or column weights or both are not constants. Its main idea is to first construct a small irregular block structured matrix H' with size of j × k, where the row and column degrees are properly optimized for good performance. Then each element in H' is replaced by a square m × m permutation matrix to obtain H, where m is a prime number. Each permutation matrix is a circularly shifted identity matrix with the shifting offsets given by ($b^s a^t \bmod m$), where $0 \le s < j, 0 \le t < k$ and (s, t) denoting the element location within H'. Generators a and b are chosen to have multiplicative order k, j for prime number m, respectively. In this way, an irregular LDPC code with rate R $\ge 1 - j/k$ and codeword length of m × k is constructed. For the details of the code construction, we refer the reader to [6]. For the sake of clarity, the block parity check matrix H' employed in this paper is shown here, which greatly impacts our decoder structures discussed in later sections.

H' = 12x24

B. Belief Propagation Decoding Algorithm

LDPC codes can be effectively decoded by using the iterative belief-propagation (BP) algorithm. In order to reduce

the hardware complexity, following log-BP algorithm [8] is typically adopted. For the variable-to-check message $L_{q_{mj}}$:

$$L_{qj} = \sum_{m \in M(j)} R_{mj} - 2\frac{r_j}{\sigma^2} \quad (1)$$

$$L_{q_{mj}} = L_{q_j} - R_{mj} \quad (2)$$

where M(j) is the set of check nodes connected to variable node j. For the check-to-variable message R_{mj}:

$$\begin{aligned} A_{mj} &= \sum_{n \in N(m)\, n \neq j} \Psi(L_{q_{mn}}) \\ R_{mj} &= -\Pi_{n \in N(m)\, n \neq j} Sign(L_{q_{mn}}) \Psi(A_{mj}) \end{aligned} \quad (3)$$

where N(m) is the set of variable nodes connected to check node m and the function $\Psi(x) = log(tanh(|x/2|))$. The iterative process stops when the maximum number of iterations is reached or all the parity check equations are satisfied, i.e., H . x = 0, where x is the hard decision of L_{qj}.

III. Proposed Decoder Structure

In this section, the decoder architecture will be presented according to the algorithm described above.

The block diagram of our decoder structure is shown in Fig. 1, where major computation blocks include parallel adder block (PAB), parity check update block (PCUB) and column sum block (CSB). PAB is used to perform the subtraction operation described in (2) while operands R_{mj} and L_{qj} are fetched from R storage memory and column sum memory, respectively. The parity equations check for early stopping is also performed by PAB. PCUB computes the message from check nodes to variable nodes as in (3). The full summation in (1) is completed by CSB, which also contains both the input buffer and decoded data memory. Router and reverse router blocks are used to connect the check-to-variable messages and variable-to-check messages according to the structure of block matrix H'.

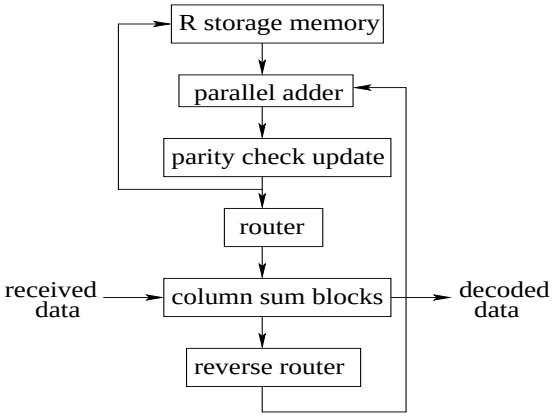

Fig. 1. Block diagram of our LDPC decoder

LDPC decoding is inherently parallizable because of data independence among rows or columns. Without much extra hardware in PCUB, all the check-to-variable messages for one particular row can be updated simultaneously. However, the complexity of fully parallel structure [4] for our frame size of 8088 bits is impractically high. Therefore, the partly parallel architecture with certain parallel factor p, i.e., p *consecutive* rows are computed at the same time, is of our interest. The case of p equal to 24 is illustrated in Fig. 2 in which the PAB as well as PCUB blocks are duplicated by 24 times. Though the number of CSB blocks stays the same, each word inside the CSB memories is widened by 24 times to accommodate the larger data flow.

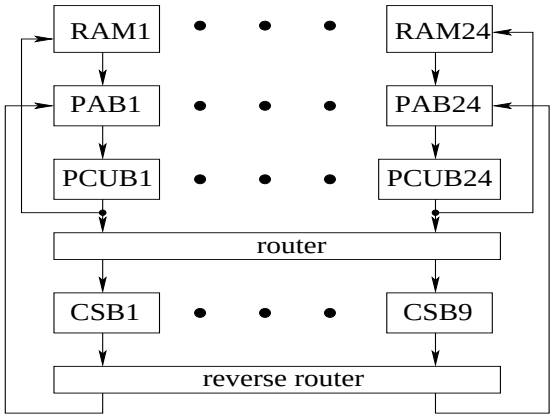

Fig. 2. Partly parallel structure with parallel factor of 24

A. Parity Check Update Block

Due to the symmetry property of the Ψ function in PCUB, its look-up table (LUT) size could be reduced by half if all the inputs are in sign magnitude data format. On the other hand, the updated check-to-variable message has to be transformed back to two's complement format before flowing into the router. The architecture of PCUB with corresponding word-length is depicted in Fig. 3, where each LUT has 32 5-bit entries in our design and each PCUB can process 7 or 8 data per clock cycle.

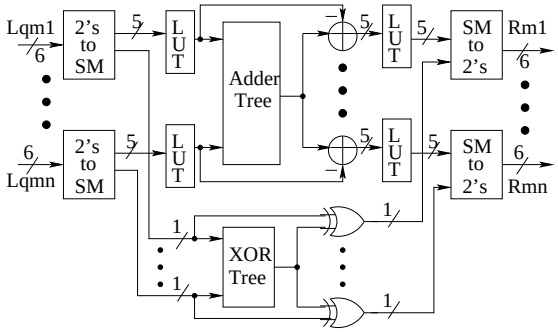

Fig. 3. Architecture of parity check update block

B. Router and Reverse Router

According to its structure, the block matrix H' could be segmented into nine sections (denoted as dashed lines in the

matrix shown in Section 2.1) such that for any row there is only one column connection in any section. In this way, for each section only one computation unit to update the variable-to-check message is necessary. Moreover, the router and reverse router blocks become fairly straightforward: during any block the routing is fixed, it switches only when a new block is started (see Fig. 4). Note that a block here represents m rows, where m is the prime number used for code construction.

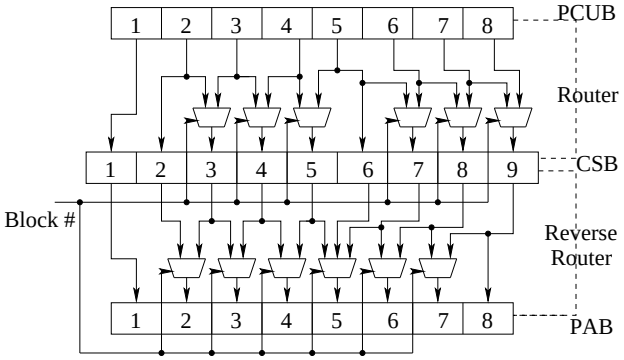

Fig. 4. Architectures of router and reverse router

In Fig. 4, the nine connections represent nine sections of the CSB while the eight blocks in PAB or PCUB come from the maximum row weight of H'. All the multiplexers in both routers are selected only by block number, which changes from 1 to j every $\lceil m/p \rceil$ clock cycles. For the reverse router, the wordlength of the incoming and outgoing data is increased to 7 bits due to the accumulation effect of CSB blocks and wire connections are anti-symmetrical to the router block.

C. Column Sum Block

The column sum block (CSB) performs the computation described in (1), which is indeed an addition-accumulation process. However, as we will see later, it turns to be the most complicated block in our decoder. Due to the segmentation of the block matrix H', there are totally nine CSBs. They share the same architecture as depicted in Fig. 5 except different memory sizes and (reverse) alignment blocks.

1) Memories: In order to fully utilize the computation blocks and increase the decoding throughput, dual memories are adopted such that the memory A is storing the accumulated check-to-variable messages in the previous iteration while the other memory B is accumulating the check-to-variable messages in the current iteration. Furthermore, their functions either as accumulation memory B or as the other memory A are exchanged or "ping-ponged" at the end of every iteration. This is accomplished by the de-multiplexiers selected by iteration number. Consequently, the parity check update can be simultaneously carried out with the variable node accumulation process, which in turn speeds up the decoding process. Similar ideas were also presented in [9]. For each one out of nine sections, the width of the received data memory is $(p \times W_r)$ and depth is ($\lceil m/p \rceil \times$ *section width*). For column sum memories A and B , each has width of $(p \times W_l)$ with the same depth as received data memory. Here W_r and W_l denote the word-lengths of received data and accumulated check-to-variable message L_{qj}, respectively.

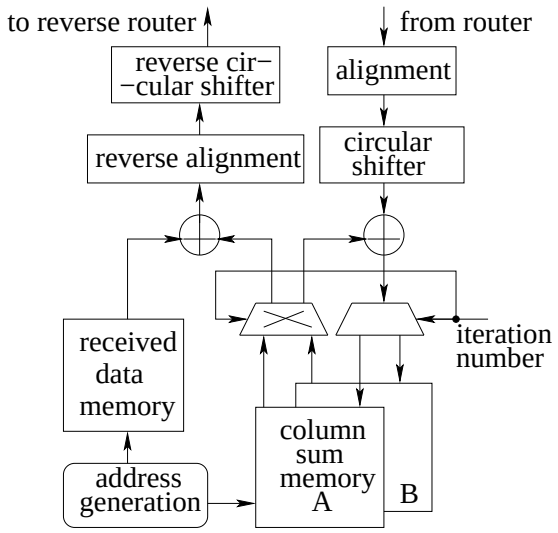

Fig. 5. Architecture of column sum block

Due to the widened word in our memory structure, the addressing generation is very simple. For each block in H', the address always starts from the *address offset*, which is obviously changed for different blocks. Then at each clock cycle, it is counted up to (*address offset* - *1*) and wraps around to zero if reaching the last memory address of that block. Received data memory and column sum memory A share the same addressing while for column sum memory B the addressing is delayed by two clock cycles because of the latency introduced by the (reverse) alignment blocks.

2) Alignment and Reverse Alignment Blocks: The alignment and reverse alignment blocks exist because of the following two reasons. First, data stored in the received data memory and column sum memory A are all in the column order for the purpose of variable node accumulation while the data going to the reverse router should be in the right row order for PAB operations. Similarly, the data coming from router via PCUB is in the row order while the accumulation process within CSB should be in the column order. Actually, this is exactly the information exchange or message passing between variable nodes and check nodes. Second, as mentioned before, different blocks in H' have different offsets, which could be transformed into two parts: *address offset* and *data offset* because of the memory's matrix structure. They can be simply calculated as follows:

$$\begin{aligned} offset &= a^s b^t \mod m \\ data\ \ offset &= offset \mod parallelism\ factor \\ address\ \ offset &= \lfloor offset/parallelism\ factor \rfloor \end{aligned} \quad (4)$$

As an example, assume prime number m is set to 337 and parallel factor is 24, the alignment issue can be described as in Fig. 6. Further assume that the *address offset* for certain block

is *addr* ($0 \leq addr \leq 14$). At clock cycle t, the word read from memory address *addr* is stored in an external register. At the next clock cycle t+1, the word at (*addr+1*) is first read out of the memory. Then its first part B1 and second part of the external register contents A2 are serially concatenated to construct the shifter input. The shifter input and its associated shift value that is solely dependent on the *data offset* then flow into a circular shifter, which will output the right-ordered word representing the messages of first 24 consecutive rows.

Finally, the reordered word is written into the column sum memory. In addition, the second part of the newly read word corresponding to *addr+1* B2 is written back into the register space that was previously occupied by A2.

In other words, in one clock cycle the alignment process involves *read-concatenate-shift-write* operations.

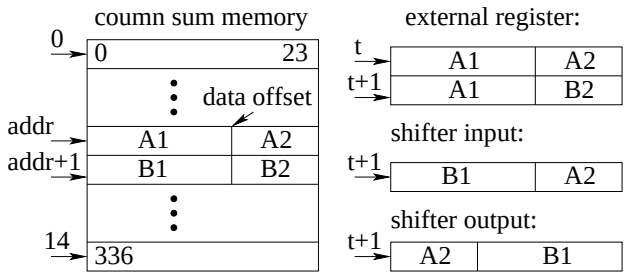

Fig. 6. Concept of alignment

It is worth mentioning that since the prime number m can not be divided by parallel factor p, the number of valid data entries in the last memory address is always less than parallel factor. In this case, all other data entries of that memory word should be set to some special values (dummy data), which we called "gap closing". In general, following four phases exist for alignment process in each block. When the memory addressing is:

1) Equal to the *address offset*: write the first word of current block from router into the external register, also write the last word of previous block into the accumulation memory B;
2) Between *address offset* and last memory address: read from and write into the same memory address. However, the reading operation is needed only when there is accumulating effect, i.e., the processed block is not the first one in its corresponding column of H';
3) Equal to the last memory address: read and write only the valid data excluding the dummy data from/to the last memory address. Use extra register to hold the new incoming data from router;
4) Between 0 and (*address offset - 1*): read from and write to the same memory address with different shift values due to the "gap closing" effect. Actually the new shift value is one more than that in the phase 2.

Compared to the alignment block, reverse alignment block is relatively simpler since only reading operations are involved. It also has four similar phases in spite of following important differences:

- Different data bus sizes due to different input word-lengths. In our design, the wordlength is 6 bits for alignment block while it is 7 bits for reverse alignment block;
- For the reverse alignment block, only the last memory word sent to reverse router with address of (*address offset - 1*) has both valid and dummy data. While for the alignment block, the word containing both valid and dummy data is the one associated with the last memory address of 14, which is not necessarily the last memory word coming from the router;
- The shift values of both blocks are p's complement, i.e., if the shift value in alignment block is s, the shift value in the reverse alignment block is p - s.

3) Circular Shifter and Reverse Circular Shifter: Circular shifter is always immediately applied after alignment block. Given any p inputs and shift value in the range of 0 to (p - 1), the shifter output will be in the expected order. Here p stands for parallel factor. The circular shifter could be generally implemented by $\lceil log_2 p \rceil = 5$ layers of multiplexers where each layer is selected by the binary encoded bits of the shift value. Same for the reverse circular shifter.

IV. FPGA IMPLEMENTATION

Based on the architectures described above, a rate 1/2 irregular LDPC code decoder with N = m × k = 337 × 24 = 8088 bits was described in VHDL and synthesized to a Xilinx Virtex-II XC2V8000ff1152 FPGA. The maximum number of iterations is set to 25. Table 1 shows the resource usage information for the major blocks in the decoder.

Table 1. Synthesis results for the major blocks

component name	number of LUTs	register bits	RAM/ROM usage
R memory	2349	35	32 18Kb BlockRAMs
PAB	6161	-	-
PCUB	9254	5	1920 (ROM32x1)
Router	1170	-	-
Reverse router	1360	-	-
CSB	52327	6739	-

Note that since distributed SelectRAM are used in CSB blocks, it is listed in the LUTs usage instead of the RAM/ROM usage. Obviously, CSB utilizes more than half of the slices. As the parallel factor is 24, the number of clock cycles to decode one frame of 4044 information bits is $\lceil 337/24 \rceil * 12 + 2 = 182$ per iteration. According to our initial synthesis results, the estimated clock frequency is 44 MHz, thus the *information* decoding throughput will be around $44 * 4044/(182 * 25) \approx$ 40 Mbps. The timing is expected to be further improved if register banks are placed on both sides of PCUB and inside CSB to obtain the pipelined decoder.

V. Conclusions

In this paper, a new decoder architecture with its FPGA implementation is presented for IPP constructed irregular LDPC codes. Detailed memory addressing generation schemes and related alignment issues are also discussed. Compared to other LDPC implementations, our decoder achieves not only better coding gain but also higher decoding throughput although relatively larger device is needed. With the maximum 25 iterations and parallel factor of 24, our rate 1/2, 8088 bits irregular LDPC decoder can achieve the *information* decoding throughput up to 40Mbps. The early stopping circuit is also embedded to reduce the overall decoding latency and power consumption. Furthermore, with the large degree of architecture sharing, the decoder could be extended to various parallel factors, code rates and block sizes.

Acknowledgment

This work was supported by Summer Student Program 2002, Texas Instruments Inc..

References

[1] R. G. Gallager, "Low density parity check codes", *IRE Trans. Info. Theory*, vol. IT-8, pp. 21-28, 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, 1996.

[3] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity approaching irregular low-density parity-check codes", *IEEE Trans. Inform. Theory*, vol. 47, pp. 619-637, 2001.

[4] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder", *IEEE J. Solid-State Circuits*, vol. 37, pp. 404-412, 2002.

[5] T. Zhang and K. K. Parhi, "A 54 MBPS (3, 6)-regular FPGA LDPC decoder", *IEEE Proc. of SIPS*, pp. 127-132, 2002.

[6] D. Hocevar, "LDPC code construction with flexible hardware implementation", *to appear in Proc. of ICC'03*, 2003.

[7] D. Sridhara, T. Fuja and R. M. Tanner, "Low density parity check codes from permutation matrices", *Conf. on Inform. Sciences and Systems, John Hopkins University* 2001.

[8] M. Chiani, A. Conti, A. Ventura, "Evaluation of low-density parity-check codes over block fading channels", *Proc. ICC*, vol. 3, pp. 1183-1187, 2000.

[9] E. Boutillon, J. Castura and F. R. Kschischang, "Decoder-first code design", *Proceeding of int'l symp. on turbo codes and related topics*, pp. 459-462, Sept. 2000.