Implementation of an LDPC Decoder on a Vector Signal Processor

Gottfried Lechner*, Andreas Bolzer†, Jossy Sayir* and Markus Rupp‡

* Telecommunications Research Center Vienna (ftw.), Donaueitystr. 1, 1220 Vienna, Austria
email: {lechner,sayir}@ftw.at

† On Demand Microelectronics, Design Center Techgate, Donaueitystr. 1, 1220 Vienna, Austria
email: andreas.bolzer@ondemand.co.at

‡ TU Vienna, Institute for Communications and RF Engineering, Gusshausstr. 25/389, 1040 Vienna, Austria
email: mrupp@nt.tuwien.ac.at

Abstract—A parallel processor architecture - a vector signal processor (VSP) - which consists of independent computation units is presented. This architecture is used to implement the sum-product algorithm to decode low-density parity-check codes. The VSP is well suited for this parallel decoding algorithm which results in a scalable decoder that allows a tradeoff between chip area and data throughput. With increasing number of computation units a data throughput of up to 36.1MBit per second can be achieved which outperforms existing implementations on digital signal processors.

I. INTRODUCTION

Low-Density Parity-Check (LDPC) codes are binary, linear error correcting block codes, defined by sparse parity-check matrices. LDPC codes allow transmission at rates close to channel capacity. Developing dedicated hardware for decoding requires a long development process and high costs. This development often has to be repeated due to changing standards.

In this work we present a vector signal processor (VSP) architecture developed by ON DEMAND [1] which is a patented software programmable and System-on-Chip compliant core. This architecture is perfectly suited for decoding LDPC codes because it is able to exploit the parallelism of the algorithm used to decode LDPC codes. Therefore, a fast and flexible implementation with high data throughput and low power consumption can be achieved.

This paper is organized as follows: in Section II the architecture of the VSP is presented. Section III describes LDPC codes, the decoding algorithm and an approximation of this algorithm that is suited for fixed-point implementation. In Section IV and V we show how the algorithm is mapped on the architecture and describe the scheduling to avoid memory conflicts. Performance measures of this implementation and a comparison with an implementation on a digital signal processor (DSP) are presented in Section VI.

This work has been founded by the Kplus program and the Christian Doppler Pilot Laboratory for Design Methodology of Signal Processing Algorithms.

Fig. 1. Structure of a Single Slice.

II. VECTOR SIGNAL PROCESSOR ARCHITECTURE

The VSP consists of a number of calculation entities called *slices*. Each slice, as depicted in Fig. 1, consists of two independent *data memories* and an *address generator*. The *data bus* connects the memories with the *register file* and the fixed-point *calculation unit*, which contains adders, multipliers, shifters, logical entities, alignment and rounding units. A set of registers provides input data for the calculating instructions.

A block diagram of the VSP is shown in Fig. 2. Several slices are arranged in parallel. The *global arithmetic unit* enables a synchronous processing of the partial results from several slices. The *global data bus* links all slices, the global arithmetic unit, and input/output ports together. One part of the global data bus is the *cross bar* which enables the slices to exchange data efficiently. The cross bar allows every slice

Fig. 2. VSP Architecture and Program Control; VLIW...very long instruction word, SIW...slice instruction word, GIW...global instruction word, PC...program counter, TX...transmit channels, RX...receive channels; The thick lines represent control paths.

in parallel to read one message from and write one message to it in one clock cycle.

The *program sequencer* provides the *program counter* that points to a very long instruction word (VLIW) in the *program memory*. Each slice is controlled by its own instruction word that is located inside the VLIW. Global instructions, which are also provided by the VLIW, manage global data transfers, jumps and loops. Since each slice contains its individual program, different algorithms can be executed in parallel. Attention should be drawn to the fact that the slices do not share resources, since each slice consists of entities for calculation and data storage.

The VSP provides a processor architecture for efficient implementations of scalar and vector algorithms. Flexible partitioning of the algorithm to a variable number of slices offers many degrees of freedom to optimize the processing efficiency. By configuring the number of slices and the width of the data bus, the VSP can be easily tailored to the actual processing power requirements of a given application. As a consequence, this architecture does not suffer from architectural processing power limits as do standard DSPs.

III. LOW-DENSITY PARITY-CHECK CODES

LDPC codes [2] are block codes described by a parity-check matrix. The term low-density originates from the fact that the number of ones in the parity-check matrix is small compared to the block length. Therefore, the matrix is sparse, which leads to a decoding algorithm with a computational complexity that is linear in the block length [3].

This decoding algorithm works iteratively by passing messages on the edges of the associated factor graph, which is a bipartite graph containing variable nodes (representing the digits of the codeword) and check nodes (representing the

Fig. 3. Factor Graph of an LDPC Code with eight variable nodes of degree 3 and four check nodes of degree 6 processed by two slices.

parity-check equations). The messages are Log-Likelihood-Ratios (LLR), i.e. the sign represents the binary digit ($0 \rightarrow +$, $1 \rightarrow -$) and the magnitude represents the reliability of the associated decision.

Without loss of generality, the implementation described in this paper is for regular codes, i.e. every variable node is connected to d_v check nodes and every check node is connected to d_c variable nodes. A factor graph of a code with $d_v = 3$ and $d_c = 6$ is shown in Fig. 3. The connections between variable and check nodes are constructed randomly while satisfying the degree constraints and avoiding short cycles, i.e. cycles of length 4. These connections can be represented by an edge interleaver.

The operations at the variable and check nodes are described in the next paragraphs following the notation of [4]. $Z_{mn}^{(i)}$ denotes a message sent from variable node n to check node m at iteration i and $L_{mn}^{(i)}$ denotes a message sent from check node m to variable node n at iteration i . The set of neighboring check nodes of a variable node n is denoted as $\mathcal{M}(n)$ and the set of neighboring variable nodes of a check node m is denoted as $\mathcal{N}(m)$.

A. Variable Nodes

Each variable node receives one message $L_n^{(0)}$ from the channel and one message $L_{mn}^{(i)}$ from every check node it is connected to. In every iteration, the variable node has to calculate an outgoing message $Z_{mn}^{(i)}$ to every connected check node according to

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m' \in \mathcal{M}(n) \setminus m} L_{m'n}^{(i)}. \quad (1)$$

After every iteration, the slices have to calculate estimates of the a-posteriori LLRs for every variable node as

$$A_n^{(i)} = L_n^{(0)} + \sum_{m' \in \mathcal{M}(n)} L_{m'n}^{(i)}. \quad (2)$$

These operations are already suited for implementation on a fixed-point architecture and therefore, no further simplification is required at this point.

B. Check Nodes

Every check node represents one parity-check equation, which can be seen as a single parity-check code. The outgoing messages $L_{mn}^{(i)}$ of a check node can be computed as

$$L_{mn}^{(i)} = 2 \operatorname{atanh} \left[\prod_{n' \in \mathcal{N}(m) \setminus n} \tanh \frac{Z_{mn'}^{(i-1)}}{2} \right]. \quad (3)$$

This computation cannot be implemented easily on a fixed-point architecture. Therefore, we use an approximation similar to the Max-Log-MAP algorithm

$$L_{mn}^{(i)} \approx \min_{n' \in \mathcal{N}(m) \setminus n} |Z_{mn'}^{(i-1)}| \cdot \prod_{n' \in \mathcal{N}(m) \setminus n} \operatorname{sgn}(Z_{mn'}^{(i-1)}). \quad (4)$$

Bounds on the performance of this approximation and the analytical derivation of a multiplicative correction factor are presented in [5]. For the sake of simplicity we used an additive correction term as described in [4] called *offset belief propagation based decoding*

$$L_{mn}^{(i)} \leftarrow \operatorname{sgn}(L_{mn}^{(i)}) \cdot \max(|L_{mn}^{(i)}| - \beta, 0). \quad (5)$$

Note that the correction term is constant and can be implemented with low complexity. For our implementation, we used the results from [4], where the value of β is optimized when designing the code using density evolution.

IV. IMPLEMENTATION

Fig. 3 shows the factor graph of an LDPC code. The variable nodes on the left hand side, which represent the bits of the block, receive messages (quantized Log-Likelihood-Ratios) from the channel. For the VSP implementation each slice alternately processes a fixed set of variable and check nodes and contains their associated messages. After finishing the variable/check node computations, the slices have to exchange their messages according to the interleaver structure. This data exchange can be performed efficiently using the cross bar, located in the global data bus. The cross bar allows every slice in parallel to read one message from and write one message

Fig. 4. Scheduling problem for four slices exchanging two messages per slice.

Fig. 5. Path search in scheduling algorithm.

to it in one cycle. In order to avoid read and write conflicts, e.g. a slice receiving two messages in the same cycle, a proper scheduling has to be found. This issue will be addressed in the next section.

While computing the check nodes, the slices check for a valid codeword (all parity-checks have to be fulfilled) and abort the iterative process if a codeword is detected. This stopping criterion can be used to lower the average decoding time or to reduce the power consumption. After stopping the iterative decoding process when the stopping criterion kicks in or when a predetermined maximum number of iterations is reached, the slices calculate the result of the decoding process.

V. SCHEDULING

The slices have to exchange their messages according to the interleaver structure as mentioned above. The hardware architecture allows every slice to write and to read a message in one clock cycle. Therefore, a scheduling has to be found to avoid that one slice receives two or more messages from other slices in one clock cycle. This problem is illustrated in Fig. 4 for the simple example where four slices have to exchange two messages per slice. If for example slice number 3 and slice number 4 would send their first message in the same clock cycle (messages E and G respectively), a collision at the destination slice number 3 would occur.

There are two possible methods to find a proper scheduling. One is to develop an efficient search algorithm that finds a se-

$$\mathbf{H} = \begin{array}{c|c} \text{slice 0} & \text{slice 1} \\ \hline \begin{array}{cccc} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{array} & \begin{array}{cccc} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{array} \\ \hline \end{array} \begin{array}{l} \text{slice 0} \\ \text{slice 1} \end{array}$$

Fig. 6. Parity-check matrix suited for scheduling. The first slice processes columns 1 to 4 (variable nodes) or rows 1 to 2 (check nodes). The second slice process columns 5 to 8 and rows 3 to 4. Every submatrix contains 4 ones.

quence of messages that can be exchanged without collisions. This search has to be run once after constructing the code and the order of the messages to be exchanged has to be stored in the slices and in the crossbar. The algorithm searches for a set of messages that can be exchanged simultaneously (4 in the example of Fig. 4) removes these messages and continues with the reduced problem of finding a set of messages in the remaining set of messages.

For illustration, assume an example of a VSP with four slices and two messages per slice as shown in Fig. 4. The letters A ... H on the left hand side represent variable-to-check node messages that have to be interleaved. The letters on the right side depict the message locations after the interleaving process. At the start phase of the algorithm each message on the left obtains the number of its destination slice. These numbers are rearranged in $K = 4$ columns as shown in Fig. 5. The solution of the scheduling algorithm is to find paths, which contain all slice numbers. E.g. the paths 2A-1C-4F-3G and 1B-4D-3E-2H.

The second possibility is to create the parity-check matrix of the LDPC code in a way that is suitable for exchanging the messages between the slices. For the description of this method, we use the parity-check matrix representation (which is equivalent to the factor graph representation used before). Fig. 6 shows an example of such code construction. The parity-check matrix $\mathbf{H}$ is divided into submatrices, where the size of the submatrices is the size of the overall matrix divided by the number of slices. Usually, the only constraint for code construction is that every row and column contains a fixed number of ones that are placed randomly. We add the additional constraint that every submatrix contains the same number of ones denoted by P . We note that for a finite number of slices, this property is fulfilled if the block length of the code goes to infinity. Simulation results for block length 1024 showed that the performance of the code does not suffer from this constraint.

In the example described in Fig. 6, the proper scheduling is easy to find. Every submatrix contains four ones, i.e. messages to be exchanged. In the first four clock cycles, slice one sends messages to slice one and slice two to slice two.¹ In the second part, slice one sends its messages to slices two and vice versa. The extension of this algorithm to a larger number of slices

¹Of course this could be avoided since a slice does not have to send a message to itself. However, for increasing number of slices, this redundant exchange of messages decreases.

TABLE I

CYCLES PER ITERATION PER BIT C_{IB} AND BIT RATE R_B (FOR $I = 10$)

L	slices	4	8	16	24	32	48	64
C_{IB}	cycles	8.9	4.4	2.2	1.5	1.1	0.74	0.55
R_B	Mbit/s	2.3	4.5	9	13.5	18	27	36.1

is straightforward.

The disadvantage of this algorithm is that it works only if the number of ones per submatrix P is an integer, which is only the case for specific numbers of slices. For a given variable node degree d_v , P can be computed as

$$P = \frac{N \cdot d_v}{L^2}, \quad (6)$$

where N is the block length of the code and L is the number of slices. For parameters $N = 1024$ and $d_v = 3$, P is only an integer for $L \in \{1, 2, 4, 8, 16, 32\}$. This disadvantage is usually not a restriction in practice.

VI. RESULTS

The decoder was implemented in VSP assembly language and the number of cycles required per iteration was simulated and calculated. We assumed a typical regular LDPC decoder with three edges per variable node and six edges per check node. The algorithm runs on an L -slice VSP and needs an average of I iterations to reach a valid code word. The number of clock cycles per iteration per bit C_{IB} can then be calculated with

$$C_{IB} = \frac{35.5}{L} + \frac{26}{N}, \quad (7)$$

where the second term is negligible for large block lengths N (typical: $N = 1,000 \dots 10,000$). Assuming a VSP running at 200MHz and setting the number of iteration to $I = 10$, the bit rate R_B can be computed. Table I depicts C_{IB} and the bit rate R_B for various numbers of slices.

A. Comparison with DSP Implementation

We compared our implementation with implementations of LDPC decoders on DSPs presented in [6], [7]. In [6], a Texas Instruments TMS320C64xx DSP was used. By exploiting the parallel computation units of this DSP, a data throughput of 5.4Mbps (at 10 iterations and a clock frequency of 600MHz) was achieved. Although the DSP is able to perform operations in parallel, it is not able to exploit the parallelism of the LDPC decoder and therefore this structure is not scalable. The VSP architecture presented is able to achieve approximately the same throughput with eight slices running at 200MHz. However, this architecture can be scaled to a larger number of slices, which directly results in an increased data throughput. This is illustrated in Fig. 7. Furthermore, the reduced clock frequency in comparison with the DSP reduces the power consumption, which is especially important for mobile devices. Dedicated hardware solutions as presented in [8]–[10] achieve a higher data throughput but require a longer development time and can not be adapted to changing standards.

Fig. 7. Throughput for a VSP with various numbers of slices compared to a standard DSP.

VII. CONCLUSION

The analysis of LDPC codes and their implementation showed that the parallel and scalable nature of the Vector Signal Processor enables a high performance and flexible software solution, which meets the required bit rate while fulfilling low-power constraints. The programmability helps the product developer to react to diverse and changing customer requirements. In comparison to a DSP solution, the advantage of this architecture is the flexibility in processing power and the reduced power consumption. The very short implementation time and flexibility is an advantage compared to dedicated decoder hardware structure.

REFERENCES

- [1] A. Bolzer, G. Krottendorfer, and M. Riener, "A new vector processor architecture for high performance signal processing," in *Proc. XII. European Signal Processing Conference (EUSIPCO 2004)*, Vienna, Austria, September 2004, pp. 2031–2034.
- [2] R. Gallager, "Low density parity check codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, January 1962.
- [3] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, March 1999.
- [4] J. Chen and M. P. C. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Communications Letters*, vol. 6, no. 5, pp. 208–210, May 2002.
- [5] G. Lechner and J. Sayir, "Improved sum-min decoding of LDPC codes," in *Proc. International Symposium on Information Theory and its Applications (ISITA 2004)*, Parma, Italy, October 2004, pp. 997–1000.
- [6] G. Lechner, J. Sayir, and M. Rupp, "Efficient DSP implementation of an LDPC decoder," in *Proc. International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, Montreal, Canada, May 2004, pp. IV-665–IV-668.
- [7] T. Bhatt, K. Narayanan, and N. Kehtarnavaz, "Fixed point DSP implementation of low-density parity check codes," in *Proc. Ninth DSP Workshop*, Hunt, Texas, October 2000.
- [8] T. Zhang and K. K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," in *Proc. IEEE Workshop on Signal Processing Systems (SIPS'02)*, October 2002, pp. 127–132.
- [9] F. Kienle, M. J. Thul, and N. Wehn, "Implementation issues of scalable LDPC-decoders," in *Proc. 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, September 2003, pp. 291–294.
- [10] M. M. Mansour and N. R. Shanbhag, "A 1.6 Gbit/sec 2048-bit programmable and code-rate tuneable LDPC decoder chip," in *Proc. 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, September 2003, pp. 137–140.