A Scalable Architecture for LDPC Decoding

Mauro Cocco*, John Dielissen#, Marc Heijligers#, Andries Hekstra#, Jos Huisken*

* Silicon Hive, Prof. Holstlaan 4, 5656AA Eindhoven, The Netherlands
# Philips Research, Prof. Holstlaan 4, 5656AA Eindhoven, The Netherlands

Abstract

Low Density Parity Check (LDPC) codes offer excellent error correcting performance. However, current implementations are not capable of achieving the performance required by next generation storage and telecom applications. Extrapolation of many of those designs is not possible because of routing congestions. This article proposes a new architecture, based on a redefinition of a lesser-known LDPC decoding algorithm. As random LDPC codes are the most powerful, we abstain from making simplifying assumptions about the LDPC code which could ease the routing problem. We avoid the routing congestion problem by going for multiple independent sequential decoding machines, each decoding separate received codewords. In this serial approach the required amount of memory must be multiplied by the large number of machines. Our key contribution is a *check node centric* reformulation of the algorithm which gives huge memory reduction and which thus makes the serial approach possible.

1. Introduction

LDPC codes [1] are very powerful error correcting codes that have been reinvented in the 1990's after the advent of Turbo coding. The use of a random interconnect structure of the graph ensures very good error correcting capabilities of the LDPC code [1]. Direct implementation of the random interconnect structure becomes infeasible for large instance sizes, because it results in routing congestions. This article addresses prevention of those routing problems, by proposing a new architecture based on a modified version of the LDPC algorithm.

The structure of the paper is as follows. In Section 2, we introduce the LDPC decoding algorithm called UMP, which is required to present the redefinition of it. Section 3 analyses the so-called *serial approach* as a solution to the congestion problem of the direct implementation of the random interconnect structure. Section 4 introduces a modification to the UMP algorithm, resulting in memory size reductions of about a factor 3.5. In section 5, a kernel of computation has been defined. By combining the kernels of computation, we have serialized the algorithm to be able to implement it in a time-folded architecture. Section 6 presents the time-folded architecture and its implementation at different sizes. Finally, some future work and conclusions are given in Section 7 and 8.

2. Background

LDPC codes are linear codes. This means that a codeword of an *(n,k)* LDPC code must satisfy *n-k* parity check equations on its *n* codeword bits. One parity check equation is of the form that the bits of a specified subset of codeword bit positions must have even parity (i.e. sum to zero modulo 2). The whole set of *n-k* parity check equations on the *n* codeword bits can be depicted by means of a bipartite graph, see Figure 1. A bipartite graph implies two kinds of nodes. The left nodes represent the bits in what should be a codeword (*bit nodes*). The right nodes represent the parity check equations (*check nodes*). When the *j*-th parity check equation involves the *i*-th codeword bit, there is an arc in the graph between the *i*-th left node and the *j*-th right node. The number of arcs connected to a node is called its degree. E.g. Figure 1 in the left nodes have degree 3.

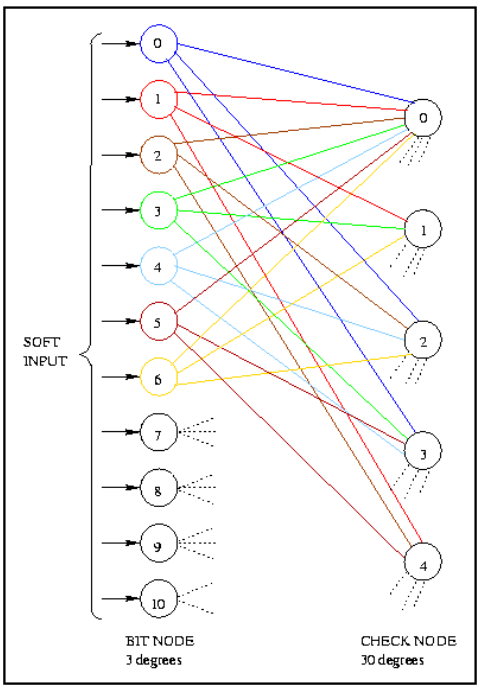

Figure 1 LDPC bipartite graph

In the literature, the LDPC decoding algorithm is also called the *message passing algorithm*, the belief propagation algorithm, or the sum-product algorithm. As

1530-1591/04 $20.00 (c) 2004 IEEE

the name message passing algorithm suggests, the decoding process revolves around the transmission of messages along the arcs of the graph, which specifies the code. These message carry probability information about the bits in the transmitted codeword, e.g. fixed point numbers representing log likelihood ratios (LLR's).

The total amount of operations in a message passing decoder is proportional to the total number of arcs. This is a reason to keep the number of arcs in the graph small. The graph is then called sparse. Information theory tells us that the whole principle of iterative decoding in a graph hinges upon the sparseness of the graph. For our purposes, a "minimum" number of arcs can be interpreted as an (average) left node degree of 3 [1]. E.g. for a coding rate R=0.9, the number of parity checks equals n-k=0.1 n. As the total number of left ends of arcs and right ends of arcs is the same, and there are 10 times fewer nodes on the right than on the left, the (average) degree of the right nodes will then be K=30.

In a classical message passing decoder, one iteration consists of a first half-iteration during which messages are sent from all left nodes over all arcs to all right nodes, and a second half-iteration during which new messages obtained from some processing in the right nodes are sent over all arcs to all left nodes, etc. Then some processing is done in all left nodes, after which a second iteration is started, etc. This continues for e.g. 30 or 40 iterations which is the decoding period. The channel output information for a codeword bit is permanently stored in the corresponding left node.

Implementing the LDPC algorithm in a straightforward manner, one would store in all nodes, all most recently received messages over all connected arcs. This enables processing and the production of as many output messages as there are input messages (equal to the degree of that node). Because all arcs together have as many left ends as right ends, this means that the same total number of input messages need to be stored on the left side as on the right side.

In a direct "one arc = one wire" implementation, in which each arc in the graph corresponds to one or more data paths, the randomness of the graph gives a very complex routing problem to implement. It is obvious that the life of a chip designer can be made easier by choosing a code for which the graph is less random. As a general rule this will degrade the error correcting capabilities of the code. Our aim is to study feasibility of not making simplifying assumptions about graph. This way we want to explore the potential of using the strongest possible codes. The LDPC codes, which we used for our implementations were produced using a random generator as in [1].

In line with Gallager [1], let the capital letter K denote the fixed degree of the right nodes. This degree variable is not be confused with the traditional lower case variable k that is the number of information bits encoded in a codeword. For rate R=0.9, we have that K=30. We note that LDPC codes with variable node degrees exist, but MacKay [3] has shown that for high rate R applications and intermediate or longer codeword lengths this brings no advantage. In fact, the error performance for higher SNR values becomes worse.

3. The Serial Approach

An important possible application area of LDPC codes is optical or magnetic storage. Next generation storage systems will be characterized by the high data rates at which the disk is read. This necessitates the availability of an error correcting decoder with large throughput (e.g. 300 Mbps). A property of storage systems is, that the decoding delay of an error correcting decoder in a player adds to the media access time. The order of magnitude of this media access time is determined by the time it takes to reposition the read head on the disk (tens of milliseconds). Therefore, decoding delays that are small relative to the head repositioning time, are deemed to be acceptable. When we multiply such admissible decoding delays in milliseconds with the huge data rate we get an admissible decoding delay measured in bits or codewords. Here, a typical LDPC codeword length n is thought to be 4,000-10,000. Depending on the details, this way one can conclude that a decoding delay of tens or even hundreds of LDPC codewords is acceptable.

The traditional way to implement LDPC is the parallel approach, but this leads to an extremely complex interconnect problem during floorplanning [2]. One possibility to avoid the extremely complex interconnect problem is going for a sequential decoding machine that processes the input nodes in a linear order from the first bit-node to the last in every iteration as shown in Figure 2. Because of the relaxed delay requirement the resulting decoding delay is not a problem.

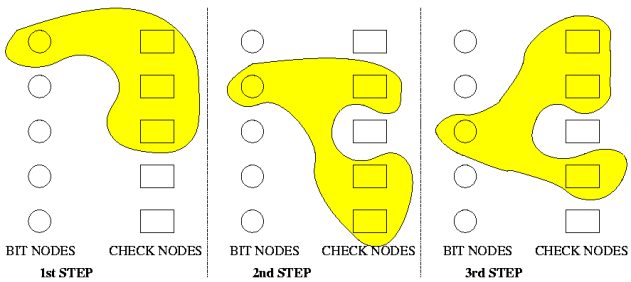

Figure 2 Serial Approach of the LDPC algorithm

The results is that the complex interconnect can be solved in random access memory. This is called the *serial approach*. To reach the required throughput performance tens or more decoding machines can be used on one chip.

Note that the serial approach requires storage of all intermediate results (messages) associated with the decoding of one codeword for each machine. With a large number of machines, this means that for the serial approach the memory requirement is critical to get sufficient machines on one chip to realize sufficient overall throughput

4. The General Idea

A number of variations of message passing decoding exist. We have chosen a simplification called the UMP algorithm of Fossorier et. al. [4,5] as the basis for our work. In principle, this simplification entails a small loss in bit error rate versus SNR performance of only a few tenths of a dB. We aim for an implementation where the use or non-use of an LDPC code brings a significant

performance improvement of several dBs, and we can afford to loose a few tenths of dB and still have a significant net advantage. Moreover, unlike the original message-passing algorithm, the UMP algorithm does not need knowledge of the SNR of the channel, thus ensuring robust operation.

The advantages of the UMP algorithm over the original message-passing algorithm are:

- We do not need complex mathematical functions nor in check nodes nor in bit nodes such as *tanh*, which would require for instance many table lookups.
- In a check node, e.g. for R=0.9, all $K=30$ output messages can be derived from just 3 numbers which are part of what we call *a dataset*, viz. a minimum, a one-but-minimum and an index; this reduces the amount of evaluations of formulas needed significantly.

For rate $R=0.9$, storing $K=30$ input messages per check node saves roughly a factor of 10 in memory space when we only store those 3 numbers. The details are explained in Section 4.2. However, such a reduction of the required amount of memory for the right nodes would then make the total left node storage requirement dominate the total storage requirement.

Our key idea is to work towards an architecture with a computational kernel that elaborates the old and the new running state information of the decoding process. These informations are captured in terms of old and new running versions of those 3 numbers per check node to achieve substantial memory savings. First we have reformulated the processing from an alternation of left-to-right and right-to-left half iterations with e.g. old algorithm state information in the left nodes and new algorithm state information in the right nodes (or vice versa) to a situation where of the aforementioned 3 numbers for all check nodes there is an old version (input to the calculations) and new version (output of the calculations). This will make all processing right-to-left-to-right in one step, so that the algorithm becomes entirely right node (i.e. check node) centric.

4.1 The UMP algorithm

The pseudo code of the UMP algorithm [4,5] is given below. The algorithm works with log likelihood ratio messages alternatively flowing from all bit nodes over all arcs to all check nodes, and back. Note that in the pseudo code below each bit node the corresponding channel output LLR is also treated as an incoming arc to that node.

```
"FOR 40 ITERATIONS DO"
 "FOR ALL BIT NODES DO"
  "FOR EACH INCOMING ARC X"
   "SUM ALL INCOMING LLRs EXCEPT OVER X"
   "SEND THE RESULT BACK OVER X"
  "NEXT ARC"
 "NEXT BIT NODE"
 "FOR ALL CHECK NODES DO"
  "FOR EACH INCOMING ARC X"
     "TAKE THE ABS MINIMUM OF THE INCOMING
                LLRs EXCEPT OVER X"
     "TAKE THE XOR OF THE INCOMING LLRs EXCEPT
                OVER X"
   "SEND THE RESULT BACK OVER X"
  "NEXT ARC"
 "NEXT CHECK NODE"
"NEXT ITERATION"
```

4.2 Hardware implications

With the UMP algorithm, one check node operation consists of the following. For each of the K bit nodes to which it is connected,

1. Compute the exclusive or (*xor*) of all hard bits output by the connected bit nodes (as determined by the signs of the output log-likelihood ratios), except the *j-th* one *(j=1,2,...,K)*.
2. Compute the minimum of all K absolute values of the log-likelihood ratios of the bit nodes to which the check node is connected, except the *j-th* one *(j=1,2,...,K)*.

A cheap way in hardware to compute all the K *xor* results is to compute the *xor* of all K input bits. The *xor* of all K input bits except one can be calculated by taking the *xor* of the aforementioned total and taking the *xor* with the input bit that is to be expected.

With respect to the K minima that have to be computed during a check node operation, we observe the following [4]. Consider the *j*-th input to the check node, and assume that its absolute value of the log-likelihood ratio is not the minimum of all K such inputs. Then, the minimum value of all K inputs except the *j-th* one will equal the overall minimum of the K values. In case, the *j-th* log-likelihood ratio does have the absolute minimum value of all K inputs, the minimum of the K inputs except the *j-th* one equals the second-to-minimum value. On the K input LLRs to a given check node, we take the minima of all numbers except one, where we first omit the first one, then the second one, then the third one, etcetera… Thus, in a check node the set of minima of K values, except the *j-th* one, can be stored using only 3 values (a *dataset*) instead of $K=30$:

- The overall minimum value,
- The overall one-but-minimum value,
- The index *j* for which the minimum input value occurs.

Now that all storage has been moved into the check nodes, a reduction in the storage requirement of roughly a factor of 10 has been obtained. Unfortunately, the sign information cannot be compressed, and therefore the memory of an architecture that contains 10,000 bit nodes is reduced to 1,000 datasets (equal to the number of check nodes for $R=0.9$ and $n=10,000$) and 30,000 sign bits, randomly connected to each other. For our investigation we have used 4 bits for the minimum and 4 for the one-but-minimum, 5 bits the address of the minimum and 30 bits for the signs; thus the total memory requirement for our algorithm is 43,000 flip-flops, instead of the initial 150,000.

5. Loop Restructuring of the UMP Algorithm

A minimal kernel of computation has been extracted from the UMP algorithm. The time folded architecture for the LDPC decoders, implements the algorithm processing the kernel computation in a linear order, see Figure 2. The computational kernel is the union of one bit node and part of the three check nodes connected to it. Note that in

every step of every iteration, the minimal kernel of computation needs the previous values and the new values as shown in Figure 3.

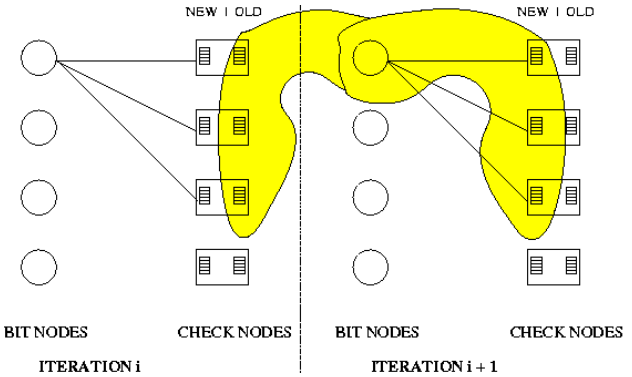

Figure 3 LDPC algorithm: the computational kernel works on the old values from the previous iteration and on the new running values

Every bit node is processed once per iteration together with the 3 check nodes, to which it is connected. As a result, every check node is called *K=30* times per iteration. Our main contribution consists of calculating the minimum and one-but minimum using these *K=30* calls. During every call a running minimum and one-but-minimum are updated. This restructuring leads to the next pseudo code:

"FOR 40 ITERATIONS DO"
 "FOR ALL BIT NODES DO"
  "CALCULATE THE OUTPUT MESSAGES FROM
   THE 3 CONNECTED CHECK NODES[1] "
  "DO RUNNING CHECK NODE UPDATES ON
   THE 3 CHECK NODES
 "NEXT BIT NODES"
"NEXT ITERATION"

Working in a serial way the LDPC decoder needs to feed the computational kernel with 3 datasets and the soft input every cycle (see Figure 5). The dataset, shown in Figure 4, is divided in the running values and the previous values of the minimum, the one-but-minimum, and the address of the minimum; the addresses of the bit node for the check node which is being processed and sign database. Furthermore, the signs of the *K=30* bit nodes connected to each check node, the total *xor* of all the signs from the previous iteration, the running total *xor* are in this dataset. After finishing an iteration, the running values becomes the old values and a new iteration starts until the complete decoding sequence is executed. To perform one iteration, the time folded architecture needs a number of cycles equal to the number of the bit nodes, thus the throughput of this architecture will be equal to:

$$Throughput = \frac{ClockFrequency}{NumberOfIterations}$$

Working at the same frequency of a full parallel architecture, it is *number of bit nodes* times slower. Note that, the clock frequency is expected to be higher than the frequency of a fully parallel architecture.

[1] using old min/one-but/address

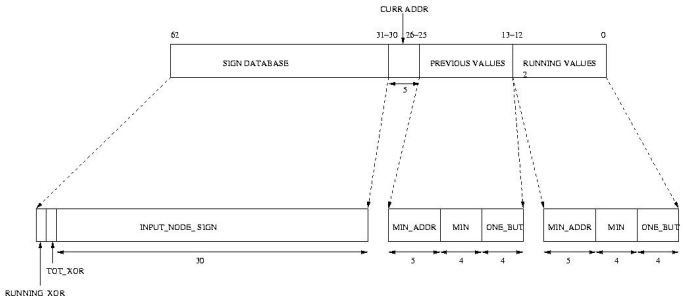

Figure 4 Datasets of the check nodes

6. Time Folded Architecture

The time folded architecture is composed of a prefetcher, the computation kernel, a memory bank, the µRom and a state machine with a program counter that controls the architecture. The schematic of the architecture is shown in Figure 5. A memory bank is required to store all the datasets of the check node. The prefetcher performs a statically scheduled cache for the datasets required by the computational kernel every cycle. It receives the control signals from the µRom memory. The µRom memory also gives the write or read command and address to the memory bank. The kernel of computation receives the datasets from the prefetcher and the channel output LLR's, to perform all the decoding operations. Its outputs are the updated datasets for the prefetcher and a serial output, which gives the decoded bit. To feed the soft inputs to the time folded architecture a single-port memory can be used with the address port connected to the program counter.

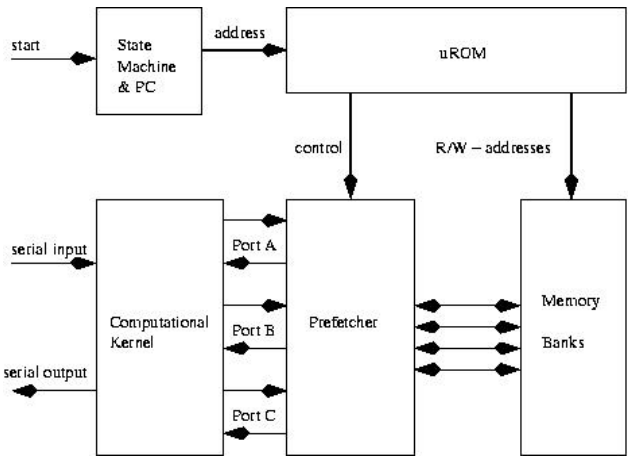

Figure 5 Schematic of the Time folded Architecture

To get the datasets to the datapath, we use memory banks and a prefetcher, which is explained in the next section.

6.1 Prefetcher and Memory Banks

The prefetcher is required for buffering, prefetching, and writing back into the memory banks the datasets required by the computational kernel. From an analysis of the communication requirements we can see that 3 datasets are read and 3 datasets are written every clock cycle. However when examining the retrieved datasets, it can be observed that for one of the ports, each dataset can be

used for 30 consecutive cycles. This implies that the datasets can stay in the prefetcher and do not have to be saved and retrieved from the memories. As a result, every clock cycle an average of 2 read and 2 write operations are required. To obtain this bandwidth we use 4 single port memory banks because their area is about half compared to a dual-port memory with the same storage capacity.

Due to the random interconnection structure, there are always conflicts on reading and writing datasets to the memories. We have written a program to solve these conflicts and this program uses three techniques, namely:

- Prefetching;
- Delayed writeback;
- Keeping datasets in the prefetcher;

Furthermore it uses the next levels of freedom:

- Allocation of datasets to the memory banks, where each check node has K datasets: one for each time it is called;
- Order of the connection table;

First of all, the connection table is sorted to reduce the resources required for all the prefetcher's operations. A heuristic is used to find a good schedule requiring adequate prefetcher and memory sizes. In general, the more an updated dataset is reused in consecutive or close-by cycles, the smaller the size of the prefetcher and memory banks are. After sorting, the program greedily runs through the list of datasets to be scheduled, trying to find a good match between prefetching and delayed write back on an arbitrary memory bank. If no solution for passing it to memory is found, the datasets are automatically stored in the prefetcher.

Special attention has been given to the initialisation and closure phase of the schedule, to allow looping: the tail must fit the head (the completed datasets of every check node has to be in the same address and memory bank so that the same schedule can be used in the next iterations).

After a successful run of the program, address assignment of the datasets in the memory bank must be done. The solution to this problem is provided by an algorithm described in [6], providing a near optimum solution (at most one memory location above the optimum) for general cases. For our specific case, the algorithm gives the optimum result in term of minimum number of memory locations.

The program produces micro code, which is put into the µRom table. Note that, in principle, from one interconnection table to the other, only the µRom contents needs to be updated. Offcourse for specific cases the prefetcher register-files sizes are different, but for these, a "worst case" size can be specified. Results show that the size of the register-files differ marginally.

The implemented time folded architecture has a block size of 9300 bit nodes, uses 4 Rams with 256 addresses by 63 bits word, and a ROM with 9305 addresses[2] with 103 bits.

6.2 Synthesis, Floor-planning and Results

The technology that has been used for implementation of the layout is a 0.13µm CMOS process with 6 metal layers. A clock period of 10ns has been selected for area dominated synthesis, and a period of 3ns for the delay dominated synthesis. The results of the synthesis are shown in the table below and a plot of the layout is shown in Figure 6.

Time folded Architecture	Freq. (MHz)	Area (mm^2)	Row Util.
Area	147	1.12	70
Delay	211	1.14	70

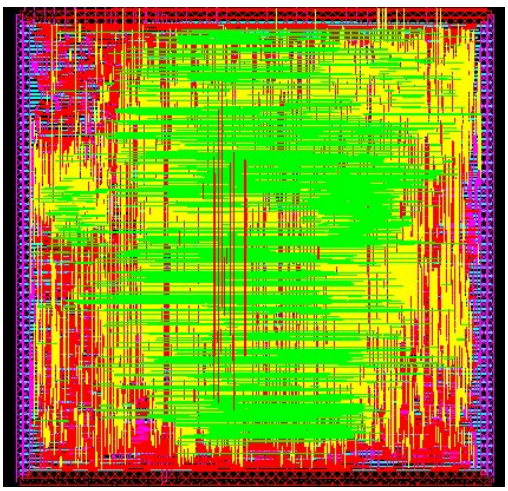

Figure 6 Layout of the time-folded architecture

The best trade-off is the synthesis by default where a 2.9% of increase in area implies a gain of 43.5% in clock frequency. To synthesise the architecture, about 300 Mb of memory was required to complete the flow. To increase the throughput of the architecture we can use more time folded architectures working in parallel (named tiles). One novelty of the proposed architecture is that only one µRom is required to control the all tiles. This is because during each iteration the operations in different tiles are the same. From the area diagram graphics shown in Figure 7 and Figure 8 we observed that for the time folded architecture almost half of the area is occupied by this µROM memory, while using more tiles in parallel the µROM memory area becomes negligible. A schematic of such architecture is shown in Figure 9. The area of LDPC decoder with a throughput of 300 Mb/s is approximately 36 mm^2 and requires 57 tiles.

6.3 Comparison

Comparing the time folded architecture with a linear extrapolation of a full parallel implementation[3], we can see that working on the same block (9300), the area of the full parallel architecture will be more than 94 mm^2, against 36 mm^2. Furthermore we can compare our time folded architecture with [2]. For a 1020 block length, bit rate 0.5 at a throughput or 1 Gbps, our architecture will require 190 tiles in parallel. Furthermore, one ROM of 1025 addresses by 93 bits and in every tile 4 RAMs of 32 addresses by 33 bits are needed. The area of our time

[2] A latency of 5 clock cycle is required between two consecutive iterations

[3] A "direct" or "flat" implementation is not possible and therefore a best case linear extrapolation is taken

folded architecture is smaller than [2]. Moreover it is scalable and does not need a new floorplan for a different connection table.

Blanksby and Howland architecture	52.5 mm^2 0.16µm CMOS
Estimation of Time folded architecture @ 1GHz with block-length 1020	25.5 mm^2 0.13µm CMOS

Note again that the LDPC algorithm works better with block lengths bigger than 4k and [2] has already routing problems for a 1k block length architecture. This is expressed in their published average wire-length of 3 mm on 0.16µm CMOS technology.

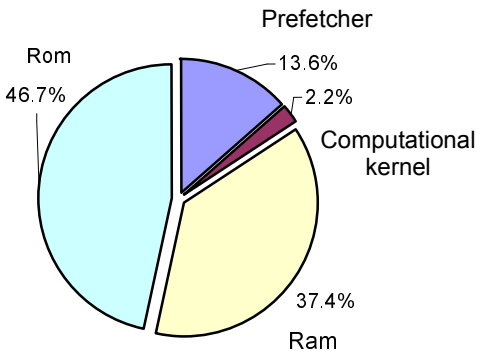

Figure 7 Area distribution of the time folded architecture

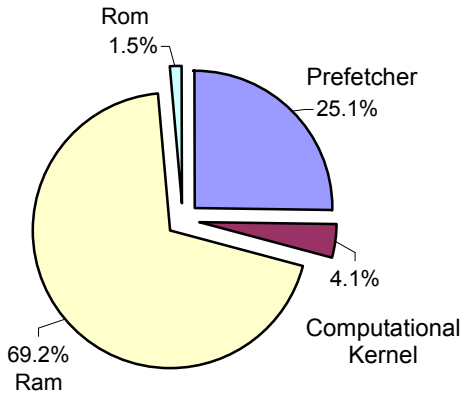

Figure 8 Area distribution for 57 tiles architecture.

7. Recommendations

Further optimizations are still possible, such as the use of pipeline stages in the datapath. Since the datapath occupies only 2% of the area in a single tile, the insertion of a pipeline stage in it, will increase its area but will have a negligible effect on the total area, while the clock frequency is expected to increase. Moreover, inserting a pipeline stage to increase the frequency, fewer tiles will be required to reach the same throughput. This will result in a considerable reduction of the chip's area.

Also, further investigations in finding better scheduling algorithms are still possible.

8. Conclusions

We have projected, investigated and implemented a time-folded architecture for an LDPC decoder. Depending on the throughput requested, more architecture-tiles can be connected in parallel, resulting in a fully scalable architecture. The architecture is capable to meet our needs of high performance: good bit error rate because of random code structure and longer codeword lengths as well as high throughput. Hence the architecture proposed in this article provides a lot of flexibility that can be tailored at specific needs with efficient area figures and speed.

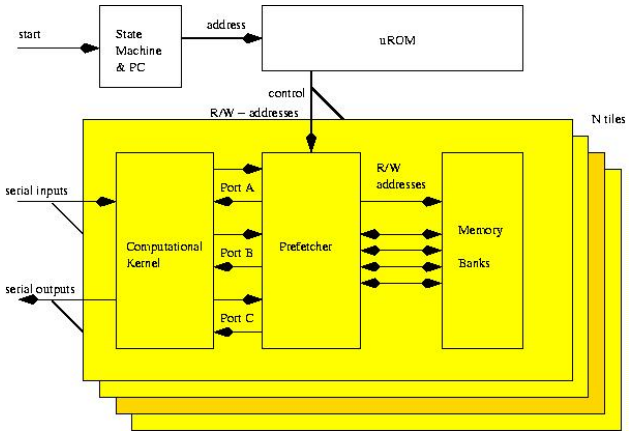

Figure 9 Schematic of the tiled architecture.

References

1. R.G. Gallager, "Low density parity check codes", IRE Trans. Information Theory, Vol. 8, pp. 21-28, 1962.
2. C.J. Howland and A.J. Blanksby, "A 690-mw 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder", Journal of Solid-State Circuits, Vol. 37 No. 3, pp. 404--412, March 2002.
3. D. MacKay, "Punctured and irregular high-rate Gallager codes", http://www.inference.phy.cam.ac.uk/mackay/CodesGallager.html.
4. M. Fossorier, M. Mihaljevic, H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," IEEE Trans. on Commun., Vol. COM-47, No. 5, pp. 673-680, May 1999.
5. J. Chen, M. Fossorier, "Near optimum universal belief propagation based decoding of low density parity check codes," IEEE Trans. on Commun., Vol. COM-50, No. 3, pp. 406-414, March 2002.
6. W.F.J. Verhaegh, J.L. van, Meerbergen, P.E.R. Lippens and A. van der Werf, "Relative location assignment for repetitive schedules", IEEE In European Conference on Design Automation with the European Event on ASIC Design, pp. 403--407, 1993.