Hindawi Publishing Corporation
VLSI Design
Volume 2008, Article ID 319095, 17 pages
doi:10.1155/2008/319095

Research Article

A Programmable Max-Log-MAP Turbo Decoder Implementation

Perttu Salmela, Harri Sorokin, and Jarmo Takala

Department of Computer Systems, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland

Correspondence should be addressed to Perttu Salmela, perttu.salmela@tut.fi

Received 18 April 2008; Accepted 30 September 2008

Recommended by Mohab Anis

In the advent of very high data rates of the upcoming 3G long-term evolution telecommunication systems, there is a crucial need for efficient and flexible turbo decoder implementations. In this study, a max-log-MAP turbo decoder is implemented as an application-specific instruction-set processor. The processor is accompanied with accelerating computing units, which can be controlled in detail. With a novel memory interface, the dual-port memory for extrinsic information is avoided. As a result, processing one trellis stage with max-log-MAP algorithm takes only 1.02 clock cycles on average, which is comparable to pure hardware decoders. With six turbo iterations and 277 MHz clock frequency 22.7 Mbps decoding speed is achieved on 130 nm technology.

1. INTRODUCTION

Telecommunications devices conforming with 3G standards [1, 2] are targeted on high volume consumer markets. For this reason, there is a real need for highly optimized structures where every advantage is taken to achieve cost efficiency. In many cases, high throughput and efficiency are obtained with highly parallel hardware implementation, which is designed only for the application in hand. On the contrary, processor based implementations tend to achieve lower performance due to limited number of computing resources and low memory throughput. As advantages, the development time is rapid if there is tool support for processor generation and the processors are flexible as the highest level behavior is described with software. One solution to achieve the benefits of both processor and pure hardware based implementations is to use application-specific instruction-set processors (ASIP) with highly parallel computing resources.

Many signal processing functions can be implemented with such processors. Especially in the telecommunication field, many baseband functions, like QR decomposition, fast Fourier transform, finite impulse response filtering, symbol detection, and error correction decoding lie on the edge between dedicated hardware and programmable processor based implementations. Turbo codes [3] are included in 3G telecommunications standards [1, 2] and decoding them is one of the most demanding baseband functions of 3G receivers. Naturally, if adequate performance can be obtained, it is tempting to implement more and more baseband processing functions on processors.

For obtaining high throughput, firstly, the processor should have specialized hardware units accelerating certain functions in the application. If there is no compiler support available, too fine grained specialized units should be avoided as they may lead to extremely long instruction word and, therefore, error prone programming. Secondly, the processor should allow accurate control of all the available resources for obtaining high utilization. In other words, even if the required resources were available, it is possible that the instruction set does not support their efficient usage. As a third requirement, the processor should provide customizable interfaces to external memory banks. Otherwise, details of the memory accesses can widen the instruction word unnecessarily or the memory accesses can limit the performance, even if there were highly parallel computing resources available.

In this paper, a turbo decoder is implemented as a highly-parallel ASIP. On the contrary to our previous turbo decoder ASIP [4], far higher parallelism is applied and higher throughput is targeted. As a consequence, also higher memory throughput is required. Expensive dual-port

memory is avoided with a novel memory interface of the extrinsic information memory. The accelerating units are allowed to connect directly to the memory interfaces of the processor to enable fast memory access. The main computations of the decoding algorithm are accelerated with dedicated function units. Due to the high parallelism and throughput, the proposed ASIP could be used as a programmable replacement of pure hardware decoder. The proposed ASIP is customizable, fully programmable, and achieves 22.7 Mbps throughput for the eight-state code, $[1((1+D+D^3)/(1+D^2+D^3))]$, with six iterations at 277 MHz clock frequency.

The next section introduces previous turbo decoder implementations. In Section 3, principles of turbo decoding and details of the applied decoding algorithm are presented. In Sections 4 and 5, parallel memory access is developed for extrinsic information memory and branch metrics. The memory interfaces are applied in practice in Section 6 as an ASIP implementation is presented and compared with other implementations. Conclusions are drawn in the last section.

2. RELATED WORK

In this section, previous turbo decoder implementations and parallel memory access methods are discussed and the differences with the proposed implementation are highlighted.

2.1. Turbo decoder implementations

Turbo decoders are implemented on high-performance DSPs in [5–7]. However, their throughput is not sufficient even for current 3G systems if interfered channel conditions require several turbo iterations. Obviously, common DSPs are mainly targeted for other algorithms like digital filtering, but not for turbo decoding. The complexity of typical computations of turbo decoding is not high, but in the lack of appropriate computing resources the throughput is modest.

Higher throughput can be obtained if the processor is designed especially for turbo decoding, that is, it has dedicated computing units for typical tasks of decoding algorithms. Such an approach is applied in [8, 9] where single instruction multiple data (SIMD) processor turbo decoders are presented. In [9], three pipelines and a specific shuffle network is applied. In [8], the pipeline has specific stages for turbo decoding tasks. With this approach the computing resources are more tightly dedicated to specific tasks of the decoding algorithm. In our previous work [4], a similar processor template is used as in this study, but far lower parallelism and throughput was targeted.

Even higher throughput can be obtained with pure hardware designs like [10, 11]. However, the programmability and flexibility is lost. Naturally, the more parallelism is used the higher throughput can be obtained. For example, by applying radix-4 algorithms the decoders in [10, 12] can process more than one trellis stage in one clock cycle. A slightly more flexible solution is to use monolithic accelerator, which is accompanied with a fully programmable processor like in [13, 14]. However, a monolithic solution can be uneconomical if the memory banks are not shared. Turbo coding requires long code blocks, so the memories can take significant chip area.

When compared to DSPs in [5–7], the proposed processor is mainly optimized for turbo decoding. There are no typical signal processing resources like multipliers. The resources of the proposed processor can be used in a pipelined fashion but there is no similar pipeline as in SIMD processor in [8]. In addition, more computing resources are used in the proposed processor as the targeted throughput is one trellis stage per clock cycle. Instead of using a specific shuffle network as a separate operation [9], the permutations are integrated in the internal feedback circuits of the path metric computations in the proposed processor. On the contrary to [10, 11], the proposed processor is programmable. When compared to [13, 14], the application-specific computing resources are accessed via datapath in the proposed processor. Thus, the resources can be controlled in detail with software.

2.2. Parallel memory access in turbo decoders

Implementations processing one trellis stage in less than two clock cycles require parallel access to the extrinsic information memory. In general, parallel access can be implemented with separate read and write memories, dual-port memory, a memory running with higher clock frequency, or some kind of memory banking structure as proposed in this paper.

Unfortunately, [10–13] do not provide details of the applied parallel access method. In [15, 16] a conflict free access scheme for extrinsic information memory is developed, but the studies do not present turbo decoder implementation applying the memory access scheme. The methods are based on address generation and bank selection functions, which are derived from the interleaving patterns of the 3GPP standard. Both methods require six memory banks for conflict free accesses. In [15] also a structure with four memory banks is presented. With four banks only few access conflicts are present. As a drawback, the structures are specific to only one class of interleaver as there is a close connection of the interleaving patterns and bank selection. For the same reason, the structures depend on the additional information provided by the interleaver.

In [17] a conflict free mapping is derived with an iterative annealing procedure. The native block length of the algorithm is a product of the number of parallel component decoders and the number of memory banks. Even if the reconfiguration is mandatory for varying interleaving patterns, no hardware implementation is presented for the annealing procedure.

In [18] graph coloring is used to find mappings. It uses more memory banks than [17], but a hardware architecture for the reconfiguration is presented. The reconfiguration takes about $10K$ clock cycles for K length code block [18]. For comparison, one conflict would take one additional clock cycle. Therefore, it can be more advantageous to suffer all the conflicts instead of reconfiguration in some cases. In addition, the address computations in [18] require division and modulus, which are difficult to implement on hardware when the block length is not a power of two.

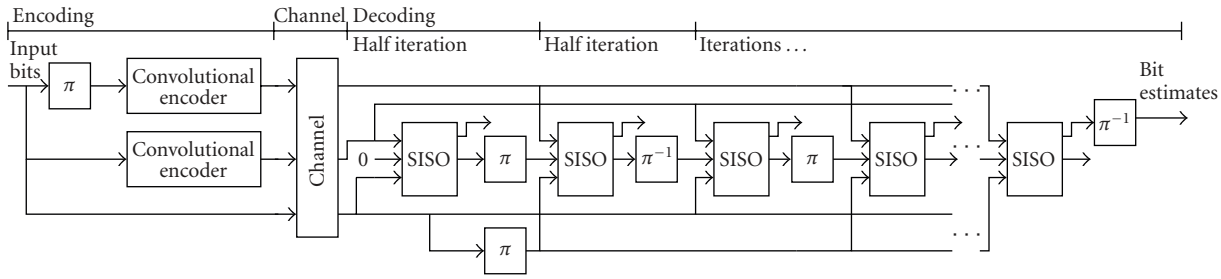


Figure 1: Turbo encoding and decoding. The decoding is an iterative process, which runs SISO component decoder several times. Interleaving and deinterleaving are denoted with π and π^{-1}, respectively.

A different approach is applied in [19–23] where buffers are applied instead of deriving conflict free address generation and bank selection functions. In [19–21] high-speed decoding with several write accesses is assumed. For each writer there is one memory bank and for each bank there is a dedicated buffer. In [20] the buffered approach is developed further and the memories are organized in ring or chordal ring structures. The work is continued in [24] where a packet switched network-on-chip is used and several network topologies are presented. To reduce the sizes of queue buffers and to prevent overflows, the network flow control is applied.

In order to conform with standards applying varying interleaving patterns a practical memory structure with four banks is proposed in this paper. The structure applies simple address generation and bank selection functions and buffering of conflicting accesses. The principles were presented in our previous work [25] and now they are applied in practice with turbo decoder implementation. Instead of solving all the conflicts with complex memory bank mechanism as in [15–18], our approach is to use a very simple memory bank selection function and to maintain a constant throughput with buffering in spite of conflicting accesses. In [15, 16], six memory banks are required for conflict free memory access. In the proposed method, only four banks are suggested.

It is shown that a modest buffer length is sufficient for 3GPP turbo codes. On contrary to previous buffered parallel access methods [19–21] our method relies on the asymmetric throughput rates of turbo decoder side and memory subsystem side. Instead of one memory bank per access, we apply a total of four banks to guarantee dual access with a modest buffer length. Furthermore, instead of dedicated buffers, we apply a centralized buffer to balance buffer length requirements, which leads to an even shorter buffer length.

3. TURBO DECODER

In this section, high-level descriptions of turbo decoding and an implementation of the decoder are given. In principle, the turbo decoder decodes parallel concatenated convolutional codes (PCCCs) in an iterative manner. In addition to the variations in the actual decoding algorithm, the implementations can be characterized also with the level of parallelism, scheduling, or the required memory throughput.

3.1. Principal operation

The functional description of the PCCC encoding and turbo decoding is shown in Figure 1. The encoding process in Figure 1 passes the original information bit, that is, systematic bit, unchanged. Two parity bits are created by two component encoders. One of the component encoders takes systematic bits in sequential order, but the input sequence of the second component encoder is interleaved. The interleaving is denoted by π in Figure 1.

The turbo decoding is described with the aid of soft-in soft-out (SISO) component decoders. The soft information is presented as logarithm of likelihood ratios. The component decoder processes systematic bit vector, $\mathbf{y}^s$, parity bit vector, $\mathbf{y}^p$, and a vector of extrinsic information $\boldsymbol{\lambda}^{\text{in}}$. As a result new extrinsic information, $\boldsymbol{\lambda}^{\text{out}}$, and soft bit estimates, $\mathbf{L}$, of the transmitted systematic bits are generated, that is,

$$(\boldsymbol{\lambda}^{\text{out}}, \mathbf{L}) = \mathbf{f}_{\text{SISO}}(\boldsymbol{\lambda}^{\text{in}}, \mathbf{y}^s, \mathbf{y}^p). \tag{1}$$

Passing the extrinsic information between the component decoders describes how a priori information of the bit vector estimates is used to generate new a posteriori information. The turbo decoding is an iterative process where generated soft information is passed to the next iteration. Every second half iteration corresponds with the interleaved systematic bits. Since the interleaving changes the order of the bits, the next component decoding cannot be started before the previous is finished. Therefore, the signals passed between the SISO component decoders in Figure 1 are, in fact, vectors whose length is determined by the code block length.

3.2. Practical decoder structure

Due to the long code block lengths a practical decoder implementation in Figure 2 consists of the actual SISO decoder and memories. Since only one component decoding phase, that is, half iteration, can be run at a time, it is economical to have only one SISO whose role is interchanged every half iteration. Although, if decoding is blockwise pipelined, then several component decoders can be used [26]. The extrinsic information is passed via a dedicated memory

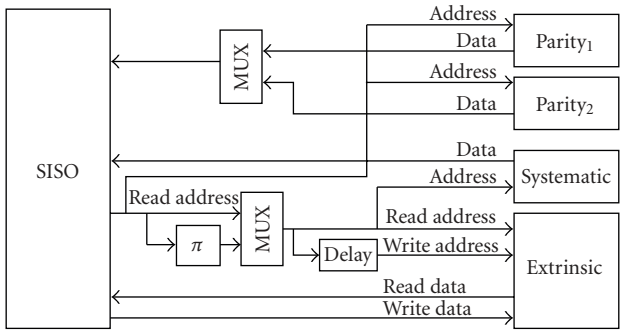


Figure 2: Practical decoder requires SISO component decoder, interleaved address generation, and memories. Extrinsic information memory is both read and written.

between the half iterations. If the component decoder is capable of processing one trellis stage every clock cycle, dual access to the extrinsic information memory is required. The interleaving takes place by accessing the memory with interleaved addresses as shown in Figure 2. In practice, the extrinsic information can reside in the memory in sequential order and no explicit de-interleaving is needed. When the interleaving is required the extrinsic information is read from and written to according to interleaved addresses. Thus, the order remains unchanged and no explicit de-interleaving is required before accessing the memory in sequential order on the next iteration.

3.3. Sliding window algorithm

The SISO component decoder can be implemented with soft-output Viterbi or some variation of a maximum a posteriori (MAP) algorithm. The MAP algorithm is also referred as the Bahl, Cocke, Jelinek, and Raviv (BCJR) algorithm according to its inventors [27]. In this study a max-log-MAP algorithm is assumed. The basic MAP algorithm consists of forward and backward processes and both forward and backward path metrics are required to compute the final outcome. Due to the long block lengths, some type of sliding window algorithm, like the one presented in [28], is usually applied to reduce the memory requirements. In the sliding window algorithm, the backward computation is not started in the end of code block but two window length blocks before the beginning of current forward process. The backward path metrics are initialized with acquisition process to appropriate values during the first window length trellis stages. After the acquisition, the backward process generates valid path metrics for the next window length stages.

Two alternative schedules for forward and backward computations are shown in Figures 3(a) and 3(b). The schedules show that the processes access the same trellis stages three times. Except for the first window length block, the blocks are first accessed in reverse order by backward path metric acquisition process, then in normal order by forward path metric computation process, and after that in reverse order by backward path metric computation process. On the contrary to three parallel processes in Figure 3(a), the schedule in Figure 3(b) has only one process running at a time. Thus, the memory throughput requirements are less demanding, but also the throughput is one third of the more parallel schedule. The proposed decoder in Section 6 applies the more parallel schedule in Figure 3(a).

3.4. Max-log-MAP algorithm

Basically, max-log-MAP algorithm can be divided into four computation tasks, which are branch metrics generation, forward metrics generation, backward metrics generation, and generation of a soft or hard bit estimate together with new extrinsic information. The forward path metric of state u at trellis stage k, $\alpha_k(u)$, is defined recursively as

$$\alpha_k(u) = \max_{u' \in U_{\text{pred}}(u)} \left(\alpha_{k-1}(u') + d_k(u', u)\right), \tag{2}$$

where $d_{u',u}(k)$ is the branch metrics, u' is the previous state, and the set $U_{\text{pred}}(u)$ contains all the predecessor states of u, that is, the states from which there is a state transition to the state u. Respectively, the backward path metrics are defined as

$$\beta_{k-1}(u') = \max_{u \in U_{\text{succ}}(u')} \left(\beta_k(u) + d_k(u', u)\right), \tag{3}$$

where the set $U_{\text{succ}}(u')$ contains all the successor states of state u'.

The soft output, L_k, is computed with the aid of the forward, backward, and branch metrics as a difference of two maximums. In the following, the minuend maximum corresponds to the state transitions with transmitted systematic bit $x^s = 0$ and the subtrahend corresponds to the state transitions with $x^s = 1$,

$$\begin{aligned} L_k = &\max_{u',u:x^s=0} \left(\alpha_{k-1}(u') + \beta_k(u) + d_k(u', u)\right) \\ &- \max_{u',u:x^s=1} \left(\alpha_{k-1}(u') + \beta_k(u) + d_k(u', u)\right). \end{aligned} \tag{4}$$

The hard bit estimate is obtained simply by the signum function, $\text{sgn}(\cdot)$, of the L_k. The new extrinsic information λ_k^{out} is computed with the aid of L_k, that is, a posteriori information λ_k^{out} is obtained as

$$\lambda_k^{\text{out}} = \frac{1}{2} L_k - y_k^s - \lambda_k^{\text{in}}, \tag{5}$$

where λ_k^{in} is the a priori information, and y_k^s is the received soft systematic bit, that is, y_k^s can have positive or negative noninteger values.

In this study, the 3GPP constituent code [1] is used as an example. The trellis of this code is shown in Figure 4. There are only eight states and four possible systematic and parity bit combinations. All the branch metrics $d_k(u', u)$ correspond to the transmitted systematic and parity bit

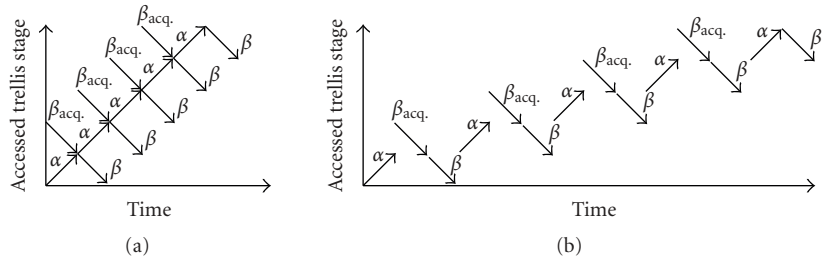


Figure 3: Schedules for sliding window algorithm, (a) three parallel processes are used, (b) one process runs at a time. Initialization of backward metrics with acquisition process is denoted with $\beta_{\text{acq.}}$, backward computation with β, forward computation with α.

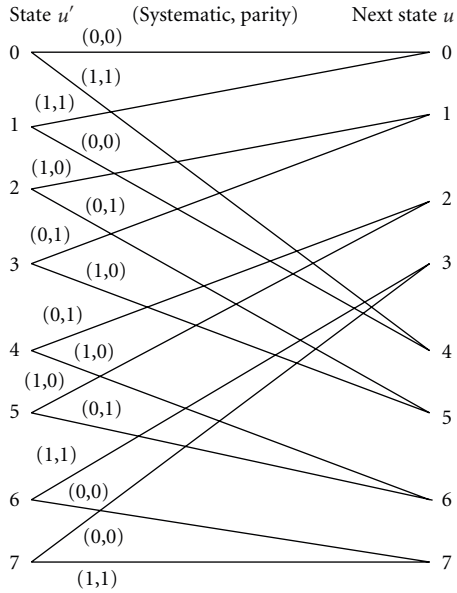


Figure 4: Trellis of eight state 3GPP constituent code. Transmitted systematic and parity bit pairs (x^s, x^p) correspond with state changes of the component encoder.

pairs (x_k^s, x_k^p). Therefore, the branch metrics notation can be defined also with the following four symbols:

$$\begin{aligned}
\gamma_k^{00} &= \underset{x_k^s=0,x_k^p=0}{d_k(u',u)} = y_k^s + \lambda_k^{\text{in}} + y_k^p,\\
\gamma_k^{01} &= \underset{x_k^s=0,x_k^p=1}{d_k(u',u)} = y_k^s + \lambda_k^{\text{in}} - y_k^p,\\
\gamma_k^{10} &= \underset{x_k^s=1,x_k^p=0}{d_k(u',u)} = -y_k^s - \lambda_k^{\text{in}} + y_k^p,\\
\gamma_k^{11} &= \underset{x_k^s=1,x_k^p=1}{d_k(u',u)} = -y_k^s - \lambda_k^{\text{in}} - y_k^p,
\end{aligned} \tag{6}$$

where the received soft parity bit is y_k^p. Previous notation shows how the branch metrics are computed. Since the branch metrics with complemented indices are negations of each other, computing only two branch metrics is sufficient if respective additions in (2)–(4) are substituted with subtractions.

4. PARALLEL MEMORY ACCESS

In this section, a parallel access scheme for the extrinsic information memory is developed. Here, expensive dual-port memory is avoided and a parallel memory approach based on multiple single-port memories is used.

4.1. Problem description

The schedule in Figure 3(a) and the computation of path metrics in (2) and (3) with branch metrics defined in (6) show that there are three parallel read accesses of extrinsic information, $\lambda_k^{\text{in/out}}$, memory. These three accesses can be replaced with one read access and appropriate buffering as will be shown in Section 5. However, the extrinsic information memory will be also written as the new values are computed and stored for later use on the next half iteration. Thus, there is a need for two accesses, that is, read and write, on the same clock cycle if one trellis stage is processed in one clock cycle.

The memory is accessed with two access patterns, namely sequential sawtooth and interleaved pattern. In sequential sawtooth access, consecutive addresses are accessed except for the window boundaries. With a window length L_{win} and a code block length, K, the sequential sawtooth access pattern can be defined as

$$P(i) = L_{\text{win}} \left\lfloor \frac{i}{L_{\text{win}}} \right\rfloor + L_{\text{win}} - 1 - (i \bmod L_{\text{win}}), \tag{7}$$

where index $i \in \{0, 1, \ldots, K - 1 + 2L_{\text{win}} + c_{\text{rw}}\}$ and constant term c_{rw} originates from from the delay of computations in the decoder. The interleaved access pattern is formed with interleaving function, $\pi(i, K)$, which generates interleaved addresses, $\pi(P(i), K)$. There is a constant difference $2L_{\text{win}} + c_{\text{rw}}$ between the read and write indices of access pattern $P(i)$, which means that in general case, there is no constant distance between read and write addresses.

The required parallel accesses can be defined with read operation, READ$(\cdot)$, and write operation, WRITE$(\cdot)$, and parallel execution, $||$, as

$$\text{READ}(P(i)) \,||\, \text{WRITE}(P(i - 2L_{\text{win}} - c_{\text{rw}})) \tag{8}$$

for sequential access and

$$\text{READ}(\pi(P(i), K)) \,||\, \text{WRITE}(\pi(P(i - 2L_{\text{win}} - c_{\text{rw}}), K)) \tag{9}$$

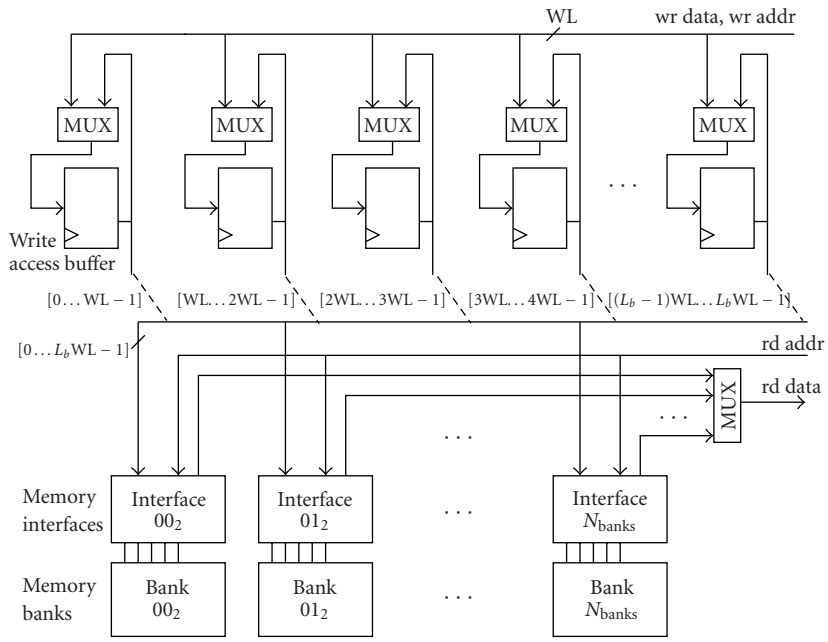


Figure 5: Proposed memory structure with buffered write operations. Word length of data and address pair is denoted with WL. Indices in brackets index the bus connected to the L_b length buffer.

for interleaved access. In the above, write operations are omitted when $i < 2L_{\text{win}} + c_{\text{rw}}$ and read operations are omitted when $i \geq K$. The index i traverses all the values $0, 1, \ldots, K - 1 + 2L_{\text{win}} + c_{\text{rw}}$ in sequential order. So, in the beginning there are only read operations and in the end only write operations.

The parallel access scheme should provide such a mapping from addresses to parallel accessible memory banks that conflicts are avoided or, alternatively, performance degradation due to the conflicts is avoided. There cannot be a constant mapping to memory banks since the interleaving function $\pi(i, K)$ varies as it is a function of code block length, K. For example, in the 3GPP standard, different interleaving patterns are specified for all the block lengths $K \in \{40, 41, 42, \ldots, 5114\}$ [1] and, therefore, the memory bank mapping should be computed on the fly. Even if graph coloring results in the minimum number of memory banks [25], such computation will be too expensive for real-time operation. To meet practical demands of on the fly mapping, a simpler memory bank mapping is required. An obvious solution for sequential access pattern is to divide the memory into two banks for even and odd addresses but it results in conflicts with interleaved addressing on every second half iteration. A dual-port memory should also be avoided as it takes more chip area than a single-port memory and the memories dominate the chip area with long code block lengths.

4.2. Parallel memory access method

The proposed parallel access method combines simple memory bank mapping and buffering of conflicting accesses. With simple bank selection the number of conflicts is decreased to a tolerable level and the performance penalty of memory access conflicts can be overcome with a short buffer. This practice of combining memory banks and buffer is illustrated in Figure 5.

The bank selection function is a simple modulo operation of the address and the number of banks, N_{banks}. Thus, when accessing the kth trellis stage the bank selection, B_{sel}, and address, B_{addr}, are generated according to

$$B_{\text{sel}} = k \bmod N_{\text{banks}},$$
$$B_{\text{addr}} = \left\lfloor \frac{k}{N_{\text{banks}}} \right\rfloor. \tag{10}$$

So, if the number of banks is a power of two, the bank selection and address generation can be generated by low-order interleaving. In other words, bank selection is implemented by simply hardwiring the low-order bits to the new positions and higher-order bits form the address.

In Figure 5 each memory bank is accompanied with an interface. The functionality of the memory interface is shown in Figure 6. In principle, the memory interface gives the highest priority for memory read operations. The read operations must be always served to allow continuous decoding. On the contrary, write operations are inserted to the buffer, which consists of registers in Figure 5. All the memory banks that do not serve the read operation are free to serve write operations waiting in the buffer. The proposed buffer must be able to be read and written in a random access manner and in parallel by all the memory bank interfaces. Thus, it must be implemented with registers. However, the length of the buffer for practical systems will be modest as will be shown later on.

Basically, the buffer balances memory accesses. Balancing is targeted also with a single shared buffer instead of

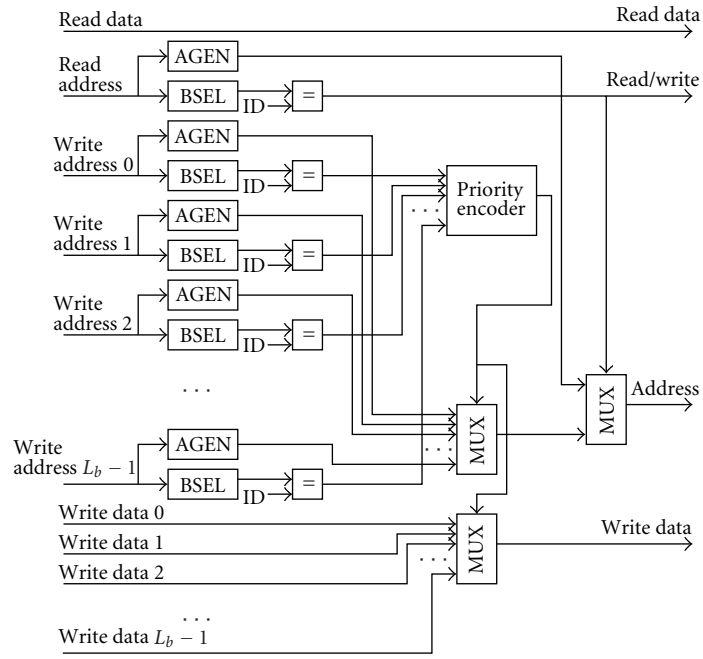


Figure 6: Memory bank interface. Length of the buffer is denoted with L_b and address generation and bank selection with AGEN and BSEL, respectively. The ID refers to the number of the interfaced bank. Write address and data signals are connected to the respective elements of the buffer.

dedicated buffers for each memory bank. If there were dedicated buffers for memory banks, their length should match the maximum requirements. However, the length of combined buffer is less than sum of dedicated buffers. This is natural, since only one buffer could be filled at a time if dedicated buffers were used.

The decoder produces memory accesses at a constant rate, two accesses per clock cycle, that is, one read and one write operation. On the contrary, the memory system is capable of maximum throughput directly proportional to the number of banks. In other words, the ability of the proposed method to perform without performance degradation is based on the asymmetric throughput rates and throughput capability between the decoder side and memory bank side.

4.3. Operation with 3GPP interleaving pattern

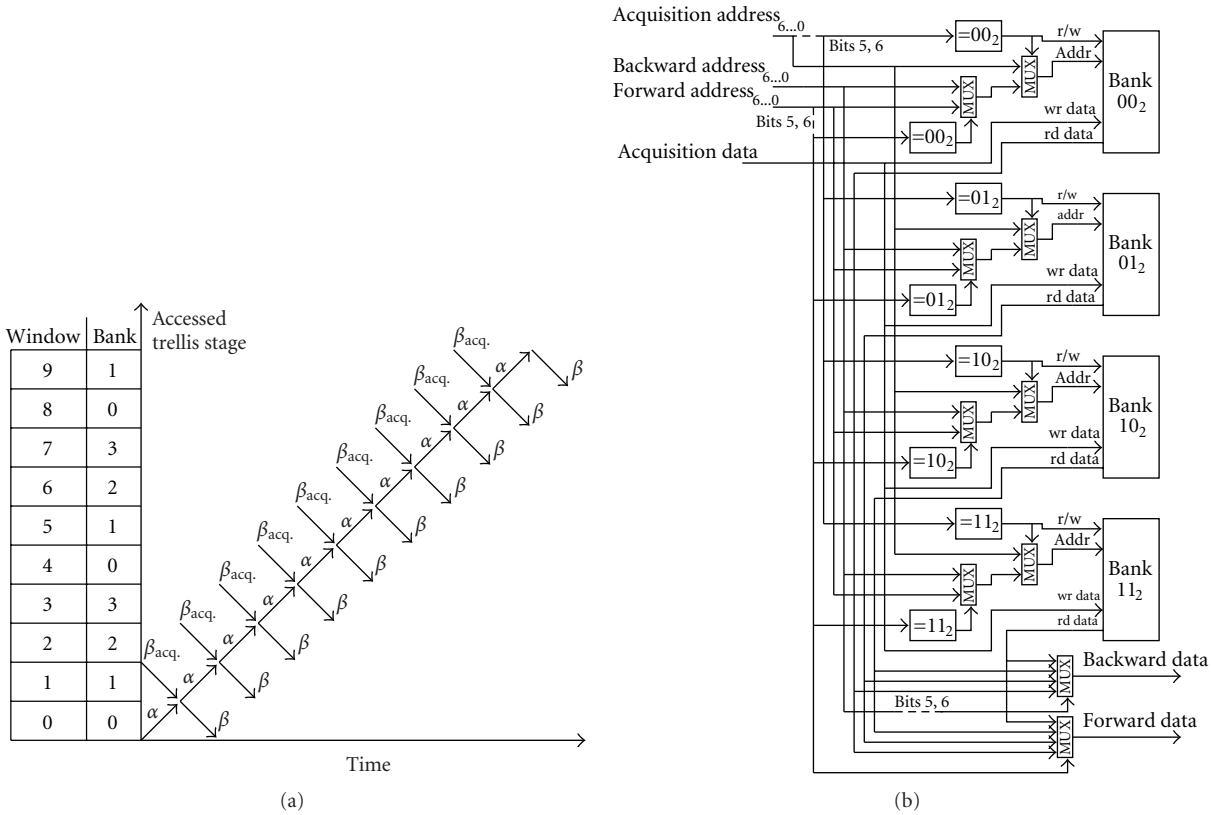
The parallel access method is used in the proposed turbo decoder processor in Section 6. Four memory banks are used in the practical implementation. With $L_{\text{win}} = 32$, $c_{\text{rw}} = 9$, and $N_{\text{banks}} = 4$ a buffer of 16 data and address pairs is sufficient to avoid buffer overflows with all the 3GPP interleaving patterns with block length, $K = 2557, \ldots, 5114$. If the code block is shorter than 2557, memory banks can always be organized as dedicated read and write memories. In addition to the number of memory banks, the required overflow free buffer length depends also on the distance between read and write operations, $2L_{\text{win}} + c_{\text{rw}}$, but it is not proportional to the distance. In other words, the required buffer length can be shorter or longer with some other values of c_{rw}.

In the end of a half iteration, there are no parallel read accesses but only write accesses for the last samples and the utilization of the buffer cannot increase. If the buffer is not emptied during this phase, extra clock cycles are spent to empty the buffer. The experimented cases with 3GPP interleaving pattern and $K = 2557, \ldots, 5114$ do not require such extra cycles, that is, the buffer is empty when the decoder issues the last write operation. Since extra clock cycles are not required, there is no performance degradation due to the buffering of conflicting accesses.

The area costs in terms of equivalent logic gates is only 3.3 kgates for the buffer and 0.5 kgates for one memory interface with $f = 100$ MHz clock frequency. With four memory banks, four interfaces are required. The complexities of memory interface and buffer are relatively low, since they do not require complex arithmetic and the buffer length is short.

4.4. Differences with other approaches

Methods in [17, 18] solve conflicts with complex memory bank mapping and address generation mechanism. However, their complexity limits their practical applicability. Methods in [15, 16] are less complex but they require six memory banks, instead of four, for conflict free access. Naturally, a memory divided into two parallel accessible banks has a lower area overhead than a respective dual-port memory. On the other hand, if the dual access requires splitting the memory into too many banks, the area overhead may exceed the costs of dual-port memory. Buffered accesses are presented in [19–21], but the ratio of memory banks to the number of parallel accesses differs and the methods are



(a) (b)

Figure 7: Buffering of branch metrics. (a) Mapping the accessed windows to four memory banks. (b) Structure of memory interface with four banks.

targeted for systems consisting of multiple parallel decoders. As a second difference there are dedicated buffers for each memory bank, which increases the total length of the buffers. Buffers and segmented memory are applied also in [22, 23]. In [23] multiple parallel decoders decoding the same code block are targeted.

5. BRANCH METRIC BUFFERING

The sliding window schedule in Figure 7(a) indicates that three trellis stages are accessed in parallel. Furthermore, computing the branch metrics according to (6) requires that also the systematic and parity bit memories are accessed in addition to the extrinsic information memory. Instead of boosting the proposed parallel memory system for even higher throughput, buffering of branch metrics can be applied. There are certain advantages of such buffering. Firstly, the buffer requires only a small amount of memory when compared to the systematic bit, parity bit, and extrinsic information memories, whose sizes are determined by the largest code block. Secondly, single-port memory can be used for all the memories. Thirdly, the access pattern of the buffer is independent of the interleaving.

The sliding window schedule in Figure 7(a) shows that even if there are sawtooth and sequential access patterns, the accessed trellis stages can be mapped to separate windows. The accessed windows do not overlap. Thus, the windows can be mapped to memory banks in order to enable parallel access. This practice is illustrated in Figure 7(a) where accessed memory banks and windows are denoted. In theory, three memory banks are required. However, four memory banks are required for a simpler and more flexible implementation. With four memory banks any delays of memory banks or processes do not cause short-term conflicts when transition from previous window to the next one takes place. In addition, implementation complexity of division and modulo operations is avoided with simple hardwired logic. Having more than four memory banks does not give additional benefits. For these reasons, four memory banks are used in Figure 7.

The forward and backward path metric computation processes read branch metrics from the buffer. The data is written to the buffer by the backward path metric acquisition process. The bank mapping in Figure 7(a) shows that the acquisition process of backwards metrics is always ahead of other processes. With window length L_{win}, the bank selection, B_{sel}, of the process accessing the kth trellis stage can be defined formally as

$$B_{\text{sel}} = \left\lfloor \frac{k}{L_{\text{win}}} \right\rfloor \bmod N_{\text{banks}}. \tag{11}$$

The address of the accessed bank, B_{addr}, is defined as

$$B_{\text{addr}} = k \bmod L_{\text{win}}. \tag{12}$$

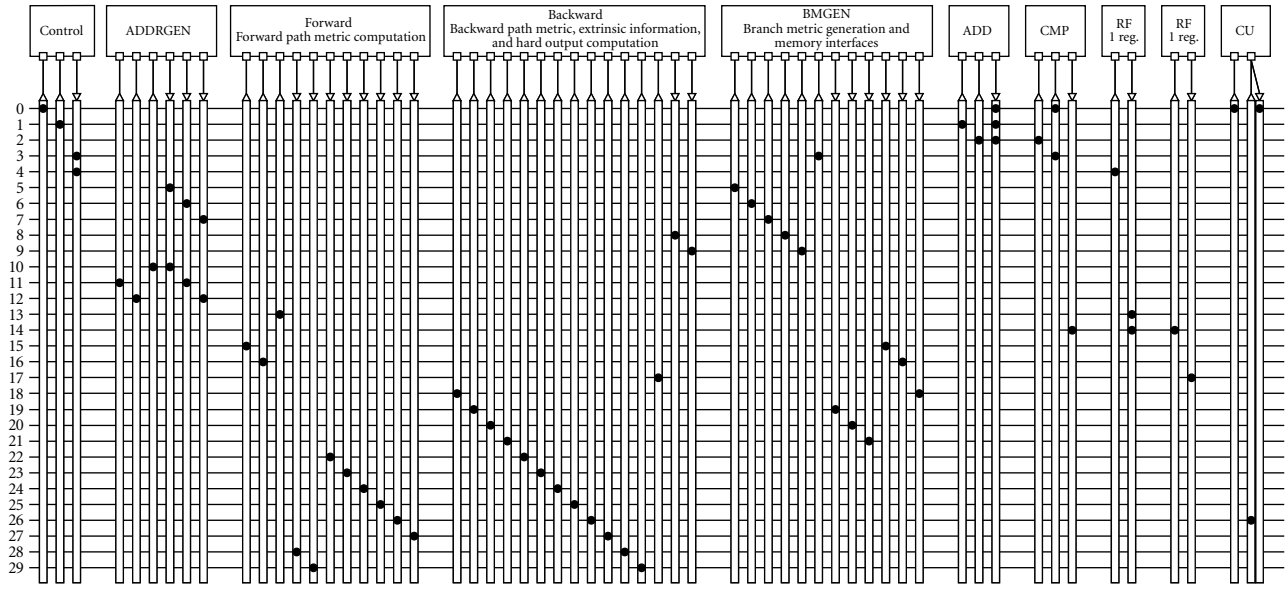


Figure 8: TTA turbo decoder processor. Filled circles denote connections between resources and buses. CONTROL: generation of control word for SFUs, ADDRGEN: address generation, FORWARD: forward path metric computation and stack memory interface, BACKWARD: acquisition, backward path metric, extrinsic information, and hard output computations, BMGEN: branch metric buffering and memory interfaces, ADD: addition unit, CMP: comparison unit, RF: register file, CU: program flow control unit.

Naturally, division and modulo operations are avoided if the window length, L_{win}, and the number of banks, N_{banks}, are powers of two. With this practice, the B_{sel} is formed by the bits $\log_2 L_{\text{win}}, \ldots, \log_2 L_{\text{win}} + \log_2 N_{\text{banks}} - 1$ of the binary presentation of k. In a similar way, the B_{addr} is given by the bits $0, \ldots, \log_2 L_{\text{win}} - 1$ of k. The structure of the memory interface applying the bitwise bank selection and address generation with $L_{\text{win}} = 32$ and $N_{\text{banks}} = 4$ is shown in Figure 7(b).

As the accessed windows do not overlap, the memory interface in Figure 7(b) can map the accesses to memory banks with simple hardwired logic. Due to the hardwired logic, the overhead and delay of the interface is kept at minimum. Furthermore, using the buffer does not require any changes to the accessing processes. In other words, the behavior of the processes remains the same as it would be with expensive multiport memory and without buffering.

6. TURBO DECODER PROCESSOR IMPLEMENTATION

The principles presented in previous sections are applied in practice as the turbo decoder is implemented on a customizable ASIP. The details of the implementation are discussed in the following subsections.

6.1. Principles of transport triggered architecture processors

In this study, we have used Transport Triggered Architecture (TTA) [29] as the architecture template. The presented principles can be applied on any customizable processor, which possesses sufficient parallelism. The TTA was chosen mainly because of the rapid design time, flexibility, and up-to-date tool support [30]. The TTA processor is an ASIP template where parallel resources can be tailored according to the application.

On the contrary to conventional operation triggered architecture processors, in TTA the computations are triggered by data, which is transported to the computing unit. The underlying architecture is reflected to the software level as the processor is programmed with data transports. The processor is programmed with only one assembly instruction, the move operation, $\rightarrow$, which moves data from one unit to another. The number of buses in the interconnection network determines how many moves can take place concurrently. In other words, instead of defining operations in the instruction word, the word defines data transports between computing resources. Such a programming approach exposes the interconnection network for the programmer and, therefore, the programmer can control all the operations in detail.

TTA processors are modular, as they are tailored by including only the necessary function units (FU). The user defined special FUs (SFU) are used to accelerate the target application by rapid application-specific computations. In addition, the SFU can be connected to the external ports of the processor, which enables custom memory interfaces. Since the processor is programmed by data transports via interconnection network, data is frequently bypassed directly from one FU to another, that is, the operands are not passed via register file (RF). Since the data is bypassed in the first hand, there is no need for a dedicated bypass network.

As the processor is tailored according to the application, the interconnection network does not have to contain all the connections. Therefore, only the connections that are required by the application program are sufficient and the

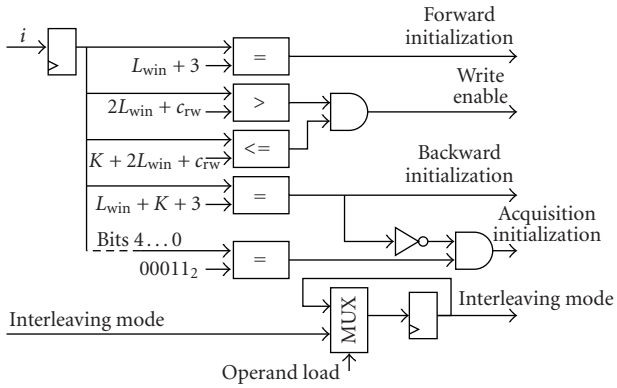


Figure 9: Control SFU evaluates several conditions in parallel and generates control word for the other SFUs. Length of sliding window is denoted with L_{win}, distance between extrinsic information read and write operations with $2L_{win} + c_{rw}$, and code block length with K. Input i is the current time step.

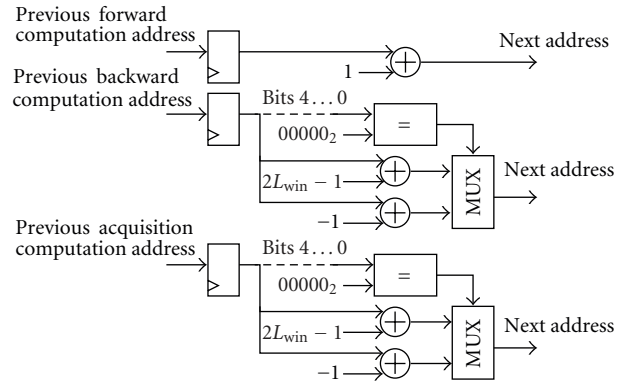


Figure 10: Address generation SFU generates one sequential and two sawtooth pattern address sequences. Length of the sliding window is denoted with L_{win}.

rest of the connections can be excluded. The exclusion of unused connections reduces the load on the buses and, therefore, it speeds up the maximum clock frequency of the processor. In this study, the proposed processor is not reconfigurable, that is, the architecture is static. If any other applications should be run with the same processor, required FUs and connections should be included in the processor.

6.2. Turbo decoder processor

The principal block diagram of the developed TTA processor is shown in Figure 8. Since the processor is targeted only to turbo decoding, it has only two conventional FUs, namely addition and comparison units. The control unit, CU, in Figure 8 is used for `jump`, `call`, and `return`, that is, the new value of program counter is written to the CU and the return address is read from the CU.

Due to the frequent bypassing of data, the processor contains only two general purpose registers in two RFs. In the turbo decoding program, the registers are used in parallel to delay continuously generated values, which are needed also on the next clock cycle. The native word length of the processor can be adjusted. In particular, the required word length depends heavily on the scaling of the input data and maximum block length. In the developed turbo decoder TTA processor, the word length of the buses and interfaces of FUs and SFUs is set to 14 bits. Naturally, the SFUs use shorter internal word length when appropriate. With relatively long word length, 14 bits, of the interconnection network, the processor could be modified to run some other applications easily. In general, if the additional FUs were inserted and the interconnection network contained all the required connections, then any other application could be run on the same processor.

6.3. Special function units

The proposed processor in Figure 8 contains five SFUs, which were designed for the application in hand. The structure and operation of these units are discussed in the following.

6.3.1. Control SFU

The purpose of the controlling SFU is to generate a control word, which is used as an argument of other SFUs. Even if the highest level control takes place in software level, the lowest level control can be implemented more conveniently in hardware. With this practice unnecessary details are hidden from the application program. The word is used to control multiplexers, initialization of state registers, and signals in the memory interfaces.

Generating the control word requires evaluating several conditionals in parallel as depicted in Figure 9. The parameter, $2L_{win} + c_{rw}$ in Figure 9 is the constant distance between read and write operations of the extrinsic information, $\lambda_k^{in/out}$, memory. The operands of the SFU are current time step, i, and interleaving mode. Even if the control could be distributed among the SFUs, the verification and any future changes, if required, are alleviated, since the control signals are packed to the single control word generated with an independent unit.

6.3.2. Address generation SFU

The address generation SFU generates addresses for accessing branch metrics, γ_k^{00} and γ_k^{01}. As it is shown by the schedule in Figure 7(a), there are three parallel processes and all of them require branch metrics. The branch metrics are generated and buffered in the branch metric computation SFU. Thus, the generated addresses are addresses of the buffer and they are not affected by the interleaving mode.

The access pattern of the addresses of the forward path metric, α, computation is sequential but the backward processes require sawtooth access patterns as show in Figure 7(a). The previous addresses are operands of the SFU and they are fed back via the interconnection network. The internal operation of the SFU is depicted in Figure 10. The window length parameter, L_{win}, in Figure 10 determines the period of the sawtooth pattern.

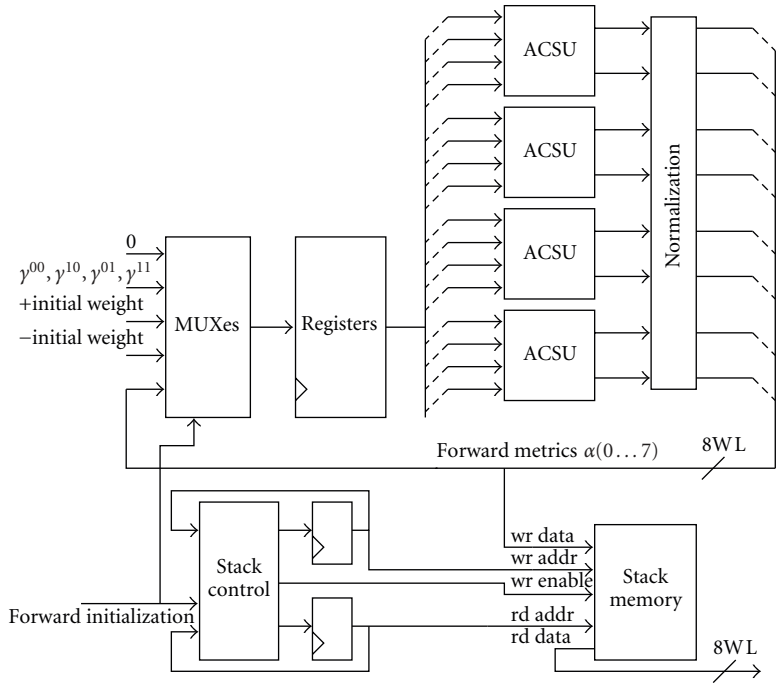


Figure 11: Forward path metric SFU contains four ACSUs, and it interfaces an external stack memory. Word length is denoted with WL.

6.3.3. Forward computation SFU

The forward computation SFU generates forward path metrics, α_k, normalizes them and continuously reverse orders one window of forward path metrics. The path metric computation in (2) requires add compare select (ACS) operations. All the path metrics for one trellis stage are computed in parallel, so the SFU contains four ACS units (ACSU) as indicated by Figure 11. Since the ACSUs followed by normalization reside in the critical path of the processor, the path metrics are fed back internally.

Reverse ordering window length block of path metrics is required, since the extrinsic information, λ_k^{out}, and hard output, $\text{sgn}(L_k)$ are computed together with the backward path metrics. The path metrics are reverse ordered with a stack in external memory. Since the stack resides in memory, the window length, that is, depth of the stack can be varied easily. All the path metrics must be pushed to the stack in parallel. Therefore, the word length is eight times the word length of forward path metrics, that is, $8 \times 11 = 88$ bits.

The SFU updates read and write pointers of the stack memory. Instead of having two stacks, the same memory area can be used for continuous reverse ordering. The new samples are stored to the memory locations which were previously loaded. The direction of the stack is interchanged after a window length of push and pop operations. In other words, the pointers are first incremented, then they are decremented and so on. With this practice, the stack memory area remains full all the time after the first window length push operations. Since the push and pop operations access always consecutive memory locations, the parallel access can be implemented trivially by two memory banks.

6.3.4. Backward computation SFU

The backward computation SFU is divided into several pipeline stages as indicated by Figure 12. The first stage computes the backward path metrics of the acquisition mode and the second stage computes valid path metrics, β_k. The first two stages are structured similarly as the forward path metric computation SFU, that is, both stages contain four ACSUs.

The next stages in Figure 12 are responsible for computing the extrinsic information, λ_k^{out}, and hard output, $\text{sgn}(L_k)$. Since there is no feedback loop, the computations can be pipelined freely to several stages. The structure takes an advantage of mapping the computations of (4) to radix-2 ACS operations, maximum operations, and radix-1 ACS operations. The mapping of computations is derived in the previous work of the authors [31]. The computation of extrinsic information, λ_k^{out}, according to (5) uses the fact that

$$\begin{aligned} \gamma_k^{00} + \gamma_k^{01} &= y_k^s + \lambda_k^{\text{in}} + y_k^p + y_k^s + \lambda_k^{\text{in}} - y_k^p \\ &= 2y_k^s + 2\lambda_k^{\text{in}}. \end{aligned} \tag{13}$$

With this practice the required term $y_k^s + \lambda_k^{\text{in}}$ can be obtained without additional memory access. Otherwise, the memory of systematic bits, y_k^s, would need dual access or there should be a long delay line preserving the values of y_k^s. Even if the backward path metric computation SFU includes a lot of arithmetic operations, it has a simple design since there is a one-to-one mapping between the computations and arithmetic units. Control signals are required only for initialization and passing forward the path metrics from the acquisition mode process.

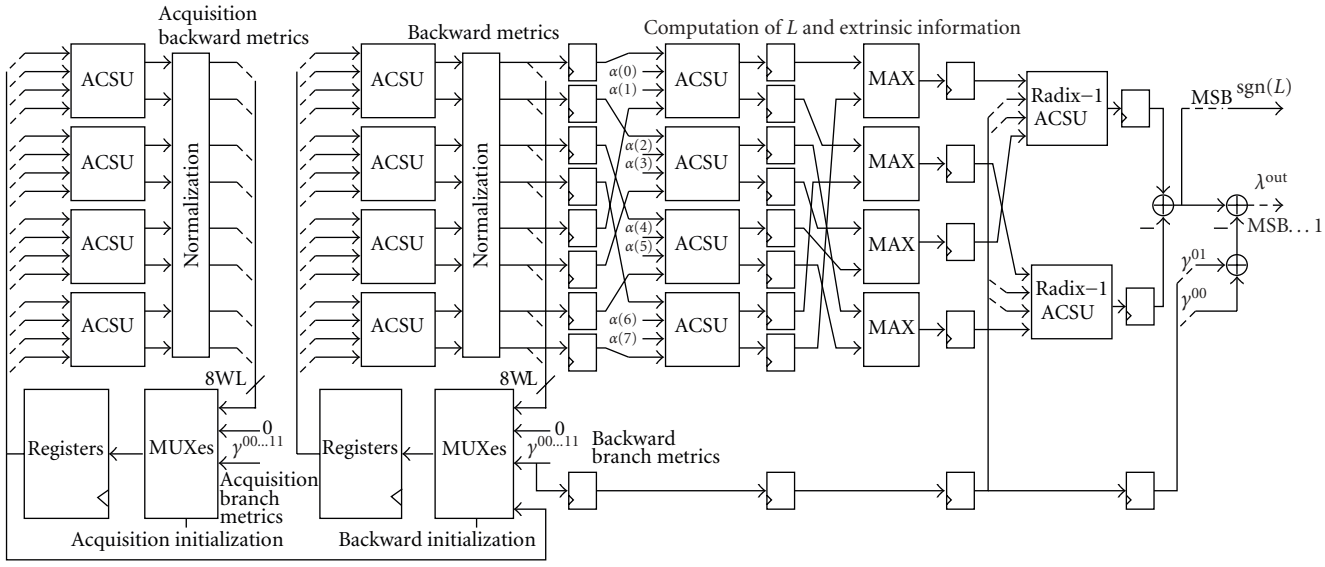


Figure 12: Backward computation SFU contains several pipeline stages for acquisition of path metrics, computation of valid backward path metrics, and computation of extrinsic information, λ_k^{out}, and hard output, sgn L_k. Word length of path metrics is denoted with WL.

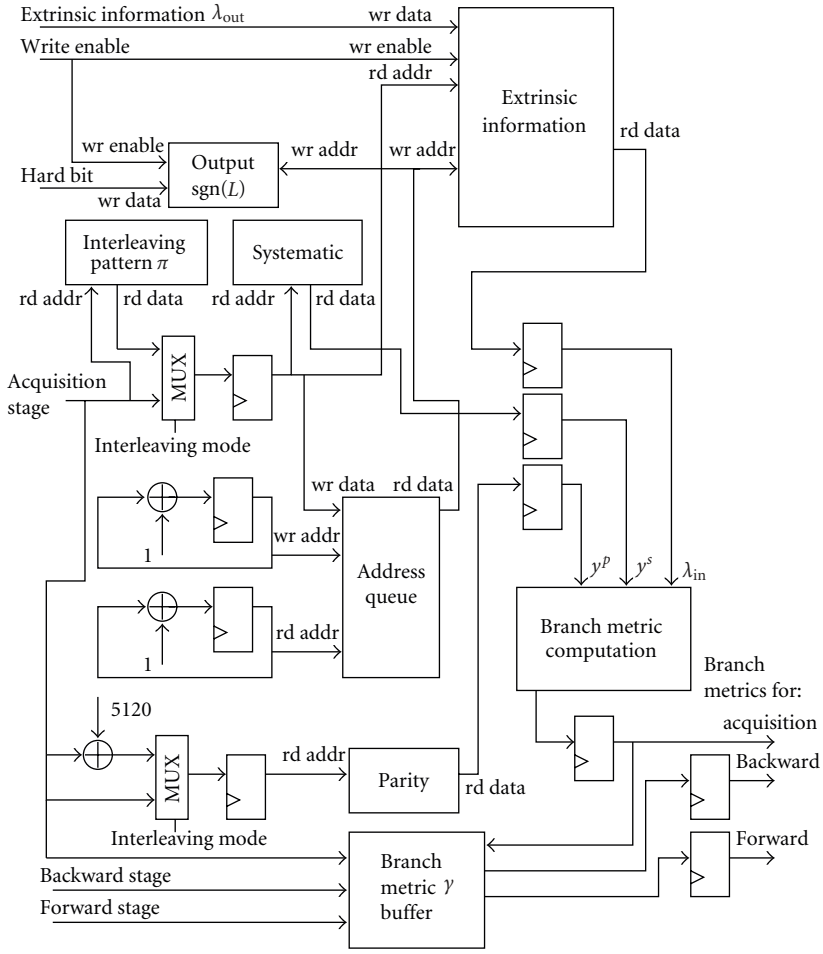


Figure 13: Branch metric computation unit interfaces external output, extrinsic information, systematic bit, parity bit, interleaver, address queue, and branch metric buffer memories.

6.3.5. Branch metric computation SFU

The SFU computes the branch metrics and interfaces the external memories for soft systematic, y_k^s, and parity bits, y_k^p, extrinsic information, $\lambda_k^{\text{out/in}}$, hard output, $\text{sgn}(L_k)$, interleaving pattern, address queue, and branch metric buffer. Generated branch metrics are buffered in memory banks as proposed in Section 5. The main advantage of the SFU is grouping of branch metric related memory interfaces into the same unit. As the SFU is used, it hides memory accesses, interleaving, and their latencies which simplifies programming and requires fewer buses in the interconnection network of the processor. The structure of the SFU is shown in Figure 13. The interleaving pattern is read from the memory but, in principle, it could be replaced with a hardware unit capable of generating one interleaved address in one clock cycle. The branch metrics are computed straightforwardly according to (6). Since the branch metrics with complemented indices are negations of each other, only the branch metrics γ_k^{00} and γ_k^{01} are buffered. The branch metrics generated for the acquisition mode backward process are passed forward and stored in the buffer. The branch metrics for the forward and backward processes are read from the buffer.

The second operation of the SFU is writing the new extrinsic information, λ_k^{out}, and hard bit estimates, $\text{sgn}(L_k)$, to the memory. The parallel memory accesses are implemented with the structure proposed in Section 4. In the proposed processor interleaving pattern is read from a dedicated memory. If a hardware interleaver were used, generating two interleaved addresses in one clock cycle would double the complexity of the interleaver. To enable an option for using hardware interleaver instead of memory, the addresses are buffered in a queue data structure, which is denoted as address queue in Figure 13. As the two access patterns of the address queue are sequential with a constant offset between them, the memory can be divided into odd and even banks trivially. In principle, the address queue buffer delays the addresses $2L_{\text{win}} + c_{\text{rw}} = 73$ clock cycles, which is the difference between read and write indices of access pattern $P(i)$ defined in (7).

Separate memory banks are used for the extrinsic information and hard bit estimates. In principle, the hard bit estimates could overwrite the extrinsic information in the last iteration. However, if stopping criterion like cyclic redundancy check [32] is applied for reducing the number of iterations, then overwriting cannot be used, as the number of iterations is not known in advance. Since the memories are interfaced with the SFU, it could also operate as an ordinary load/store unit if the encoding of the control signals were extended.

The sizes of external memories are summarized in Table 1. Naturally, the required word lengths depend heavily on the initial scaling and accuracy of the input data. In this study, the systematic and parity soft bits, y_k^s and y_k^p, are stored in 7-bit wide memory banks. The address width and data width of the interleaver memory is determined by the code block size 5120, which requires 13-bit address bus. Both the parity bits are mapped to the same memory bank, as they are not accessed during the same half iteration. Therefore, the parity memory is double sized. The forward metric stack in Table 1 requires wide word length as eight path metrics are stored in parallel.

```
procedure turbo begin
    #First iteration
     call max-log-MAP || interleaving := false
     call max-log-MAP || interleaving := true
    ...
    #Last iteration
     call max-log-MAP || interleaving := false
     call max-log-MAP || interleaving := true
end

procedure max-log-MAP begin
    call initialization_of_SFUs
    loop (K + 2 × L_win) begin
      call run_SFUs
    end
    call finish_computations
end
```

Algorithm 1: High-level program flow of the turbo decoder. Parallelism is denoted with ||. Code block length and window length are denoted with K and L_{win}, respectively.

6.4. Turbo decoder program

The turbo decoder program is programmed in parallel assembly and it follows the sliding window schedule in Figure 7(a). The highest-level pseudo code is shown in Algorithm 1. The subprograms of the `max-log-MAP` procedure are inlined to avoid jump latency. The first procedure, `initialization_of_SFUs`, feeds the initial constants to the control and address generation SFUs. The loop kernel repeats instruction words consisting of computation and loop control parts. The computation part of the instruction word feeds the control word to all the SFUs, addresses to branch metric computation SFU, branch metrics to forward and backward computation SFUs, and hard bit estimates and extrinsic information to the branch metric computation SFU. The loop control part includes addition, comparison, and conditional jump operations. In total, the instruction word consists of 30 parallel data transports.

The number of iterations of the main loop in Algorithm 1 exceeds the block length K. Additional clock cycles are taken by the first window length L_{win} trellis stages as the branch metrics buffer is filled with the values of first window. The last window requires also additional L_{win} clock cycles, as the results can not be computed before the forward path metrics for the last window are ready. Due to the latencies of the SFUs, valid results are not generated immediately. Therefore, the total number of activations of SFUs exceeds the number of iterations in the loop. The last stages are not processed in the loop to match the total number of required activations.

Table 1: External memory banks. Turbo code block length is 5120.

Memory	Address width	Data width	Size (bits)
Systematic bits, y_k^s	13	7	$5120 \times 7 = 35840$
Parity bits, y_k^p	14	7	$10240 \times 7 = 71680$
Extrinsic inf., $\lambda_k^{\text{in/out}}$	11	10	$4 \times 1280 \times 10 = 51200$
Interleaver, π	13	13	$5120 \times 13 = 66560$
Hard output, $\text{sgn}(L_k)$	13	1	$5120 \times 1 = 5120$
Address queue	7	13	$73 \times 13 = 949$
Branch metric buffer, γ_k^{00}, γ_k^{01}	5	20	$4 \times 32 \times 20 = 2560$
Fwrd. metric stack, $\alpha_k(0, \ldots, 7)$	5	88	$32 \times 88 = 2816$

6.5. Performance and complexity

Since the actual max-log-MAP algorithm is unaltered, the error correction performance of the decoder equals to typical max-log-MAP based turbo decoders with the same window and block lengths. The throughput is determined by the number of clock cycles per code block. The developed TTA turbo processor takes 10404 clock cycles with the length-5120 code block. So, the throughput of one iteration is

$$R = 5120\,\text{bits} \Big/ \left(10404 \frac{1}{f_c}\right), \tag{14}$$

where f_c is the clock frequency. The processor was synthesized on 130 nm standard cell technology with nominal conditions of 1.35 V voltage and 125°C temperature. The area in terms of logic gate equivalents of the generated netlist and the corresponding throughput are given in Table 2. The design is area-efficient as the computing resources are used efficiently, there are no large multiplexers, and the high-level structure of the processor is very simple as shown in Figure 8.

In addition to absolute valued throughput in Table 2, the relative efficiency of the developed processor and decoder program can be analyzed. The number of clock cycles per trellis stage, C_{stage}, of max-log-MAP computation, that is, half iteration is

$$C_{\text{stage}} = \frac{10404/2}{5120} = 1.016. \tag{15}$$

The efficiency can be described as a measure how close to the theoretical cycle count the achieved number of clock cycles approaches. With the applied resources, the theoretical cycle count equals to the block length. Thus the efficiency can be defined as $E = 1/C_{\text{stage}} = 0.984$. In general, if the applied algorithm contains any loops, the efficiency of processor implementation is degraded by the overhead of loop prologues and epilogues. Overhead can also be caused by jump latency or inability to fully apply software pipelining. The obtained high efficiency indicates that such an unavoidable overhead has only minor part in the total cycle count. The efficiency, E, gives also the utilization of the main SFUs. Thus, the high efficiency indicates that the main computing resources are in use most of the time and the developed turbo decoder TTA processor operates almost as efficiently as it is theoretically possible with the given resources.

Table 2: Complexity and throughput of the turbo decoder TTA processor.

Clock frequency	Area	Throughput 1 iteration	Throughput 6 iterations
100 MHz	27.9 kgates	49 Mbps	8.2 Mbps
200 MHz	31.9 kgates	98 Mbps	16.4 Mbps
250 MHz	35.7 kgates	123 Mbps	20.5 Mbps
277 MHz	43.2 kgates	136 Mbps	22.7 Mbps

6.6. Comparison

A comparison with other turbo decoder implementations is summarized in Table 3. The implementations are categorized into three classes. Pure hardware designs are not programmable. Monolithic accelerators are implementations where processor is accompanied by a dedicated hardware decoder. The third category contains processors, in which the computing resources are accessed via a datapath.

Naturally, turbo decoders applying more accurate algorithms like log-MAP instead of max-log-MAP require more area and the longer critical path lowers clock frequency. In log-MAP algorithms, an approximation, $\ln(e^a + e^b) = \max(a, b) + f(a, b)$, is used and comparisons are difficult since the accuracy of the correction term, $f(a, b)$, may vary. Typically the recursive update in (2) and (3) dominates the critical path and prevents high clock frequencies. However, the path metric computation can be accelerated also by expressing the recursion in such a way that the control signals of selection operations are computed in parallel with additions like in [33, 34].

The complexity is tabulated if it is given as logic gate equivalents excluding the memories in the respective reference. Due to the differing underlying cell structures, comparing different FPGA architectures would be difficult and the size of the memories depends on the targeted block size and technology. For example, [20] takes 250 kgates with memories but the computing units of the core decoder take only 24 kgates. Even if the memories are excluded in Table 3, it is still possible that some implementations may use register based delay lines for queue data structures and the registers are naturally included in the gate count. Such a register based approach is simple to design as it does not require address generation nor memory bank selection logic. As a drawback,

Table 3: Comparison of turbo decoder implementations.

Category	Reference	Architecture/technology	Clock frequency	Area	Throughput 1 iteration	Algorithm	Cycles/stage (C_{stage})
Pure HW design	[10]	180 nm technology	145 MHz	410 kgates	144 Mbps	log-MAP	0.50
	[11]	Virtex 5 FPGA	310 MHz	—	139 Mbps	MAX SCALE	1.12
	[35]	130 nm technology	246 MHz	44.1 kgates	112 Mbps	max-log-MAP	1.10
	[12]	Virtex 2 FPGA	56 MHz	—	79.2 Mbps	MAP	0.35
	[36]	180 nm technology	100 MHz	115 kgates	27.1 Mbps	max-log-MAP	1.84
	[37]	180 nm technology	111 MHz	85.0 kgates	25 Mbps	log-MAP	2.21
	[20]	180 nm technology	133 MHz	24.0 kgates	22.8 Mbps	log-MAP	2.92
Monolithic accelerator	[13, 38]	Turbo coprocessor of C64x	300 MHz	86.6 kgates	90.4 Mbps	log-MAP	1.66
	[14]	SISO dec. with SIMD	135 MHz	34.4 kgates	32.9 Mbps	[max]-log-MAP	2.05
Programmable processor	Proposed	TTA proc. (130 nm)	277 MHz	43.2 kgates	136 Mbps	max-log-MAP	1.02
	[8]	SIMD ASIP (65 nm)	400 MHz	64.1 kgates	100 Mbps	log-MAP	2.00
	[4]	TTA proc. (130 nm)	210 MHz	20.8 kgates	14.1 Mbps	max-log-MAP	7.46
	[9]	SIMD DSP	400 MHz	—	10.4 Mbps	max-log-MAP	19.2
	[5]	TigerSHARC DSP	250 MHz	—	9.6 Mbps	max-log-MAP	13.0
	[39, 40]	VLIW ASIP (FPGA)	80 MHz	—	5.0 Mbps	max-log-MAP	8.00
	[6]	SP-5 SuperSIMD DSP	250 MHz	—	4.7 Mbps	max-log-MAP	26.9
	[7]	C62x VLIW DSP	300 MHz	—	4.4 Mbps	max-log-MAP	34.5
	[41]	ST120 VLIW DSP	200 MHz	—	2.7 Mbps	max-log-MAP	37.0
	[42]	C55x DSP	300 MHz	—	2.0 Mbps	max-log-MAP	74.8
	[43]	PC with Pentium III	933 MHz	—	366 kbps	max-log-MAP	1275
	[44]	XiRisc reconf. proc. (FPGA)	100 MHz	—	270 kbps	log-MAP	185
	[41]	DSP56603 DSP	80 MHz	—	243 kbps	max-log-MAP	165

transferring electric charge through all the registers in a delay line consumes a lot of energy.

The throughput metrics are normalized to one iteration to alleviate comparisons. The throughput is directly proportional to the clock frequency, which results in a low throughput for some FPGA based implementations. Therefore, also the last column should be observed, as it gives the number of clock cycles per trellis stage, C_{stage}. It is calculated from the throughput and the clock frequency, unless it is given in respective reference. For [42], an achievable 300 MHz clock frequency has been assumed to calculate the throughput.

The implementations [10, 12] in Table 3 have $C_{stage} < 1$, as they apply the radix-4 algorithm. The architecture in [12] includes two component decoders. The decoders in [36, 37] support also Viterbi decoding. The area of [36] includes a path metric memory of the Viterbi decoder and an embedded interleaver. The interleaver is included also in the area of [35]. The implementation in [20] is targeted for high-speed turbo architecture consisting of several parallel decoders. The performance and complexity are reported for one decoder in Table 3. Naturally, non-programmable decoders tend to have a lot of dedicated computing resources for functions of the decoding algorithm and they have high throughput when compared to majority of programmable processors.

For [13, 38] the complexity in Table 3 includes only the turbo coprocessor, but not the accompanying C64x VLIW DSP. The accompanying processor is included in the complexity of [14] since the proportion of only the decoder part was not available. The interleaving pattern is computed with the processor and the decoder supports both max-log-MAP and log-MAP algorithms in [14]. In both implementations, the decoder is not tightly connected to the datapath of the processor, so it is not flexibly controllable. Instead, the decoder must process independently which resembles pure hardware decoders. As a second drawback of monolithic accelerators, some memory is dedicated only for the turbo decoder component.

The ASIP in [8] supports also Viterbi decoding. The ASIP has an 11 stage pipeline. There are dedicated pipeline stages for address generation, branch metric generation, state metric computation, and four stages for computing the soft output. The processor in [9] has three pipelines and trellis butterflies are alleviated with a specific shuffle network. The decoding algorithm of the processors in [39, 40] is selectable. In the table, performance of max-log-MAP is given as it achieves higher clock frequency. Our previous work in [4] applies more sequential schedule, which is presented in Figure 3(b), as it contains less computing resources than the proposed processor in this paper. Finally, Table 3 shows that conventional commercial DSPs have modest throughput

and C_{stage}. This is understandable, since their architectures are optimized mainly for high throughput multiply and accumulate operations but not for turbo decoding.

The proposed processor has the highest throughput of all the programmable turbo decoder processors. The performance is comparable with pure hardware implementations and the number of clock cycles per trellis stage, C_{stage}, is best of all the implementations, which do not apply the radix-4 algorithm. For example, even if the clock frequency is lower when compared to [11], the proposed processor has only slightly worse performance, since it has better C_{stage}. The low C_{stage} shows that the programmability and flexibility of the processor does not degrade the efficiency. The utilization of the computing resources is even higher than with the pure hardware decoder.

7. CONCLUSIONS

A programmable turbo decoder processor was presented in this study. High decoding throughput was targeted as the computing resources were designed to process one trellis stage in a clock cycle. Such a throughput requires high parallelism. As a significant result, the study showed that high parallelism can be utilized with a programmable processor if the algorithm can be partitioned to accelerating units and highest level controlling software conveniently. Complex memory access patterns and demand of several small temporary memories showed the importance of configurable memory interfaces in real implementation. A large dual-port memory was avoided with a simple parallel access method for the extrinsic information memory. Instead of fixed memory interface, the proposed processor allowed to integrate complex memory interfacing within the SFUs. With this practice, the the memory throughput requirements were met. Finally, the comparison showed that even if the proposed turbo decoder TTA processor is fully programmable, the performance was comparable with pure hardware solutions. Thus, the benefits of both implementation methods were obtained.

REFERENCES

[1] 3GPP, "3GPP TS 25.212; multiplexing and channel coding (FDD)," December 2001.

[2] 3GPP, "3GPP2 C.S0002-C physical layer standard for cdma2000 spread spectrum systems," May 2002.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and encoding: turbo-codes," in *Proceedings of IEEE International Conference on Communications (ICC '93)*, vol. 2, pp. 1064–1070, Geneva, Switzerland, May 1993.

[4] P. Salmela, H. Sorokin, and J. Takala, "A turbo decoder ASIP implementation forUMTS receiver," submitted to *International Journal of Embedded Systems*.

[5] T. A. K. K. Loo and S. A. Jimaa, "High performance parallelised 3GPP turbo decoder," in *Proceedings of the 5th European Personal Mobile Communications Conference*, pp. 337–342, Glasgow, UK, April 2003.

[6] J. G. Harrison, "Implementation of a 3GPP turbo decoder on a programmable DSP core," in *Proceedings of Communications Design Conference*, pp. 1–9, 3DSP, San Jose, Calif, USA, October 2001.

[7] J. Nikolic-Popovic, "Implementing a MAP decoder for cdma2000™ turbo codes on a TMS320C62x DSP device," Texas Instruments, SPRA629, May 2000.

[8] T. Vogt and N. Wehn, "A reconfigurable application specific instruction set processor for Viterbi and log-MAP decoding," in *Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS '06)*, pp. 142–147, Banff, Canada, October 2006.

[9] Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, "Design and implementation of turbo decoders for software defined radio," in *Proceedings of IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS '06)*, pp. 22–27, Banff, Canada, October 2006.

[10] M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24 Mb/s radix-4 Log-MAP turbo decoder for 3GPP-HSDPA mobile wireless," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC '03)*, pp. 150–151, San Francisco, Calif, USA, February 2003.

[11] Xilinx, "3GPP Turbo Decoder v3.1," DS318, May 2007.

[12] D. G. Choi, M.-H. Kim, J. H. Jeong, et al., "An FPGA implementation of high-speed flexible 27-Mbps 8-state turbo decoder," *ETRI Journal*, vol. 29, no. 3, pp. 363–370, 2007.

[13] S. Agarwala, T. Anderson, A. Hill, et al., "A 600-MHz VLIW DSP," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1532–1544, 2002.

[14] M.-C. Shin and I.-C. Park, "SIMD processor-based turbo decoder supporting multiple third-generation wireless standards," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 7, pp. 801–810, 2007.

[15] P. Salmela, T. Järvinen, T. Sipila, and J. Takala, "Parallel memory access in turbo decoders," in *Proceedings of the 14th IEEE Personal, Indoor and Mobile Radio Communications (PIMRC '03)*, vol. 3, pp. 2157–2161, Beijing, China, September 2003.

[16] D.-S. Shiu and I. Yao, "Buffer architecture for a turbo decoder," International Patent Application WO 02/093 755 A1, November 2002.

[17] A. Tarable, S. Benedetto, and G. Montorsi, "Mapping interleaving laws to parallel turbo and LDPC decoder architectures," *IEEE Transactions on Information Theory*, vol. 50, no. 9, pp. 2002–2009, 2004.

[18] X. He, H. Luo, and H. Zhang, "A novel storage scheme for parallel turbo decoder," in *Proceedings of the 62nd IEEE Vehicular Technology Conference (VTC '05)*, vol. 3, pp. 1950–1954, Dallas, Tex, USA, September 2005.

[19] M. J. Thul, N. Wehn, and L. P. Rao, "Enabling high-speed turbo-decoding through concurrent interleaving," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '02)*, vol. 1, pp. 897–900, Phoenix, Ariz, USA, May 2002.

[20] M. J. Thul, F. Gilbert, T. Vogt, G. Kreiselmaier, and N. Wehn, "A scalable system architecture for high-throughput turbo-decoders," *The Journal of VLSI Signal Processing*, vol. 39, no. 1-2, pp. 63–77, 2005.

[21] F. Berens, M. J. Thul, F. Gilber, and N. Wehn, "Electronic device avoiding write access conflicts in interleaving, inparticular optimized concurrent interleaving architecture for high throughput turbo-decoding," European Patent Application EP1401108 A1, March 2004.

[22] Z. Wang, Y. Tang, and Y. Wang, "Low hardware complexity parallel turbo decoder architecture," in *Proceedings of IEEE*

International Symposium on Circuits and Systems (ISCAS '03), vol. 2, pp. 53–56, Bangkok, Thailand, May 2003.

[23] Z. Wang and K. Parhi, "Efficient interleaver memory architectures for serial turbo decoding," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '03)*, vol. 2, pp. 629–632, Hong Kong, April 2003.

[24] C. Neeb, M. J. Thul, and N. Wehn, "Network-on-chip-centric approach to interleaving in high throughput channel decoders," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 2, pp. 1766–1769, Kobe, Japan, May 2005.

[25] P. Salmela, R. Gu, S. S. Bhattacharyya, and J. Takala, "Efficient parallel memory organization for turbo decoders," in *Proceedings of the 15th European Signal Processing Conference (EUSIPCO '07)*, pp. 831–835, Poznan, Poland, September 2007.

[26] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative decoding of concatenated convolutional codes: implementation issues," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1201–1227, 2007.

[27] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, vol. 20, no. 2, pp. 284–287, 1974.

[28] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 260–264, 1998.

[29] H. Corporaal, "Design of transport triggered architectures," in *Proceedings of the 4th IEEE Great Lakes Symposium on Design Automation of High Performance VLSI Systems (VLSI '94)*, pp. 130–135, Notre Dame, Ind, USA, March 1994.

[30] P. Jääskeläinen, V. Guzma, A. Cilio, T. Pitkänen, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Multimedia on Mobile Devices 2007*, vol. 6507 of *Proceedings of SPIE*, pp. 1–11, San Jose, Calif, USA, January 2007.

[31] P. Salmela, T. Järvinen, and J. Takala, "Simplified max-log-MAP decoder structure," in *Proceedings of the 1st Joint Workshop on Mobile Future and the Symposium on Trends in Communications (SympoTIC '06)*, pp. 10–13, Bratislava, Slovakia, June 2006.

[32] C. Bai, J. Jiang, and P. Zhang, "Hardware implementation of Log-MAP turbo decoder for W-CDMA node B with CRC-aided early stopping," in *Proceedings of IEEE Vehicular Technology Conference (VTC '02)*, vol. 2, pp. 1016–1019, Birmingham, Ala, USA, May 2002.

[33] Z. Wang, "High-speed recursion architectures for MAP-based turbo decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 470–474, 2007.

[34] I. Lee and J. L. Sonntag, "A new architecture for the fast Viterbi algorithm," *IEEE Transactions on Communications*, vol. 51, no. 10, pp. 1624–1628, 2003.

[35] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "A 58mW 1.2mm^2 HSDPA turbo decoder ASIC in 0.13 μm CMOS," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC '08)*, vol. 51, pp. 264–612, San Francisco, Calif, USA, February 2008.

[36] C.-C. Lin, Y.-H. Shih, H.-C. Chang, and C.-Y. Lee, "A low power turbo/Viterbi decoder for 3GPP2 applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 4, pp. 426–430, 2006.

[37] M. A. Bickerstaff, D. Garrett, T. Prokop, et al., "A unified turbo/Viterbi channel decoder for 3GPP mobile wireless in 0.18-μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1555–1564, 2002.

[38] T. Wolf, D. Hocevar, A. Gatherer, P. Geremia, and A. Laine, "600 MHz DSP for baseband processing in 3G base stations," in *Proceedings of the Custom Integrated Circuits Conference*, pp. 393–396, Orlando, Fla, USA, May 2002.

[39] P. Ituero and M. López-Vallejo, "New schemes in clustered VLIW processors applied to turbo decoding," in *Proceedings of International Conference on Application-Specific Systems, Architectures and Processors (ASAP '06)*, pp. 291–296, Steamboat Springs, Colo, USA, September 2006.

[40] P. Ituero and M. López-Vallejo, "Further specialization of clustered VLIW processors: a MAP decoder for software defined radio," *ETRI Journal*, vol. 30, no. 1, pp. 113–128, 2008.

[41] H. Michel, A. Worm, N. Wehn, and M. Münch, "Hardware/software trade-offs for advanced 3G channel coding," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*, pp. 396–401, Paris, France, March 2002.

[42] T. Ngo and I. Verbauwhede, "Turbo codes on the fixed point DSP TMS320C55x," in *Proceedings of IEEE Workshop on Signal Processing Systems (SIPS '00)*, pp. 255–264, Lafayette, La, USA, October 2000.

[43] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios," *International Journal of Wireless Information Networks*, vol. 8, no. 4, pp. 203–215, 2001.

[44] A. La Rosa, C. Passerone, F. Gregoretti, and L. Lavagno, "Implementation of a UMTS turbo-decoder on a dynamically reconfigurable platform," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '04)*, vol. 2, pp. 1218–1223, Paris, France, February 2004.