VLSI Implementation of WiMax Convolutional Turbo Code Encoder and Decoder

Journal of Circuits, Systems, and Computers

VLSI IMPLEMENTATION OF WiMax CONVOLUTIONAL TURBO CODE ENCODER AND DECODER

MAURIZIO MARTINA, MARIO NICOLA, GUIDO MASERA*

Dipartimento di Elettronica, Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino I-10129, Italy

maurizio.martina@polito.it, mario.nicola@polito.it, guido.masera@polito.it

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

A VLSI encoder and decoder implementation for the IEEE 802.16 WiMax convolutional turbo code is presented. Architectural choices employed to achieve high throughput, while granting a limited occupation of resources, are addressed both for the encoder and decoder side, including also the subblock interleaving and symbol selection functions specified in the standard. The complete encoder and decoder architectures, implemented on a 0.13 μm standard cell technology, sustain a decoded throughput of more than 90 Mb/s with a 200 MHz clock frequency. The encoder has the complexity of 9.2 kgate of logic and 187.2 kbit of memory, whereas the complete decoder requires 167.7 kgate and 1163 kbit.

Keywords: VLSI architecture; Convolutional Turbo Code Encoder and Decoder; High throughput parallel architecture

1. Introduction

Modern wireless communication standards are facing the growing demand for high throughput imposed by nomadic fruition of multimedia services and applications. However, harsh conditions of wireless channels impose to employ channel codes to grant reliable data delivery. As a significant example, the IEEE 802.16 WiMax standard for broadband wireless access [1] employs convolutional codes (CC) [2], block turbo codes (BTC) [3], convolutional turbo codes (CTC) [4] and low density parity check codes (LDPC) [5].

CTCs are among the most powerful error correcting codes, but the iterative BCJR algorithm [6] required to decode these codes exhibits a high computational complexity. As a consequence, when high throughput ought to be achieved, as in the WiMax standard, dedicated hardware implementation (ASIC, Application Specific

*This work is partially supported by the MEADOW (MEsh ADaptive hOme Wireless nets) project, funded by the Italian government.

Integrated Circuit) is mandatory. Even if several works address CTC implementation, it is still a major subject of interest in the scientific literature. In fact, modern communication systems, such as [1], impose throughputs of many tens of Mb/s: consequently, a clock frequency of several hundreds of MHz must be employed for the decoder. Though current scaled CMOS technologies allow to reach clock frequencies of several hundreds of MHz, such high clock frequencies can increase ASIC unreliability and nonrecurrent costs. Parallelization is an effective methodology to achieve high throughputs while keeping low the clock frequency (few hundreds of MHz in this case) [7]. However, as pointed out in several works, e.g. [8], [9], [10], [11], [12], the design of parallel CTC decoder architectures has to deal with the problem of collisions in memory access. The collision problem is often exacerbated by the need for supporting several block size values that imply multiple interleaving laws: in this case, the parallel decoding architecture is requested to avoid, or at least limit, collisions for all supported interleavers. As a significant example Almost regular permutations (ARP) are proposed in [10]: these permutations are similar to those adopted in WiMAX and enable the implementation of parallel interleaving architectures.

Since the DVB-RCS and WiMax standards employ double binary CTC [13], some recent works address CTC double binary decoders implementation both as dedicated solutions [14], [15], [16], [17] and programmable architectures [18], [19], [20].

The aim of this paper is twofold: i) to give general guidelines to design a complete double binary CTC encoder and decoder VLSI architecture; ii) to detail the VLSI architecture of all the blocks involved in the WiMax CTC, namely, as depicted in Fig. 1 [1], CTC encoder, subblock interleaver and symbol selection, on the transmitter side, and symbol deselection, subblock deinterleaver and CTC decoder on the receiver side. The rest of the paper is structured as follows: in Section 2 the main CTC principles are briefly recalled. Section 3 details the proposed VLSI architectures; in particular Section 3.1 deals with the CTC encoder, the subblock interleaver and the symbol selection architectures, whereas Section 3.2 describes the symbol deselection, the subblock deinterleaver and the CTC decoder architectures. Finally in Section 4 guidelines to design a complete double binary CTC encoder and decoder architecture are presented, the gate count and the amount of memory required by the different blocks are discussed and compared with other works available in the literature; in Section 5 conclusions are drawn.

2. Theory of operation

The WiMax CTC encoder is based on the parallel concatenation of two 8-state, double binary, circular recursive systematic CCs [1], where CC1 receives the information symbols in natural order and CC2 receives the information symbols in a scrambled order according to the interleaver Π. Each CC receives an information symbol u made of a couple of bits (A_i, B_i) and produces two parity bits (Y_i, W_i). Thus, a CC coded symbol c is made of four bits (A_i, B_i, Y_i, W_i), whereas a complete CTC coded symbol is made of six bits (A, B, Y_1, W_1, Y_2, W_2), where index 1 represents

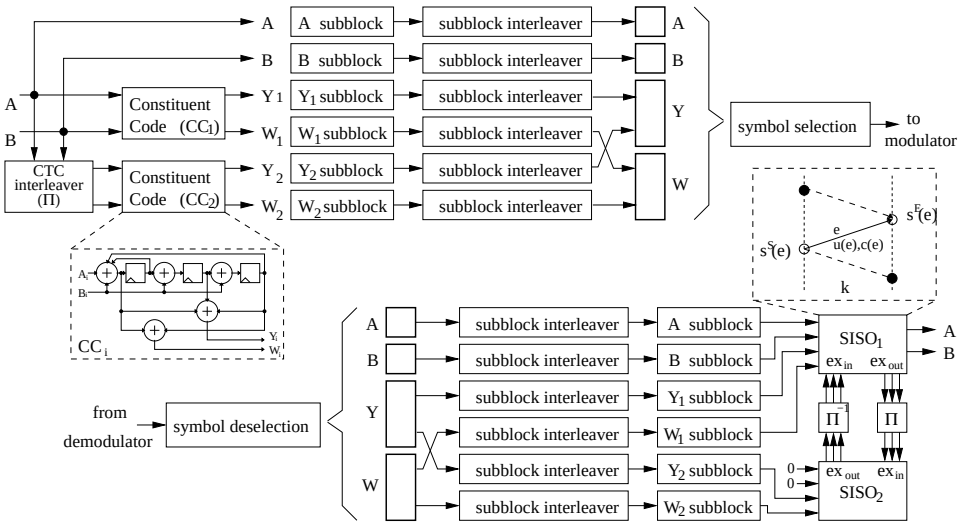

Figure 1. WiMax CTC complete encoder and decoder chain.

the output of the in-order CC and 2 represents the output of the interleaved one. Interleaved uncoded bits (A_2, B_2) are not sent. As a consequence, a frame made of N couples of bits becomes a frame of $6N$ CTC coded bits. The $6N$ bits of the CTC coded frame are scrambled and arranged as an array by the subblock interleaver. Finally, the symbol selection allows to match the actual encoder rate to the channel condition by sending: a proper slice out of the $6N$ bits (puncturing), all the $6N$ bits (no puncturing), more than $6N$ bits (repetition).

At the decoder side the symbol deselection receives soft values produced by the demodulator, usually in the form of log-likelihood ratios (LLRs) and arranges them for the subblock deinterleaver. If the encoder performs puncturing, the symbol deselection sets to zero the punctured LLRs, whereas if repetition is performed the symbol deselection combines the multiple LLRs values to increase the information reliability.

In the CTC decoder, the SISO (Soft In Soft Out) module executes the BCJR algorithm, in its logarithmic form [21,22]. Each SISO module receives the intrinsic log-likelihood ratios (LLRs) of coded symbols c from the channel and outputs the LLRs of information symbols u. The two SISO modules exchange extrinsic LLRs $(\lambda_k[u])$ by means of interleaving memories Π and Π^{-1} (Fig. 1). The output extrinsic LLRs of symbol u at the k-th step $(\lambda_k[u;O])$ are computed as:

$$\lambda_k[u;O] = \overset{*}{\max}_{e:u(e)=u}\{b(e)\} - \overset{*}{\max}_{e:u(e)=\tilde{u}}\{b(e)\} - \lambda_k[u;I] \tag{1}$$

where $\tilde{u}$ is an input symbol taken as a reference (usually $\tilde{u} = 0$), e represents a certain transition on the trellis, $u(e)$ is the uncoded symbol u associated to e and $b(e)$ is a transition metric detailed in the next paragraph. The $\overset{*}{\max}\{x_i\}$ function

[21,23] is implemented as $\max\{x_i\}$ followed by a correction term stored in a small Look-Up-Table (LUT) [23]; this solution is named Log-MAP. The correction term, usually adopted when decoding binary codes, can be omitted for double binary turbo codes [13] with minor error rate performance degradation. As a consequence, in the following of this paper, we will refer to $\max\{x_i\}$ instead of $\overset{*}{\max}\{x_i\}$ (Max-Log-MAP).

Since the double binary CTC works on couples of bits, each SISO produces three extrinsic LLRs; thus, in general, the terms $\lambda_k[u;O]$ and $\lambda_k[u;I]$ are vectors. The term $b(e)$ in (1) is defined as:

$$b(e) = \alpha_{k-1}[s^S(e)] + \gamma_k[e] + \beta_k[s^E(e)] \tag{2}$$

$$\alpha_k[s] = \max_{e:s^E(e)=s} \left\{\alpha_{k-1}[s^S(e)] + \gamma_k[e]\right\} \tag{3}$$

$$\beta_k[s] = \max_{e:s^S(e)=s} \left\{\beta_{k+1}[s^E(e)] + \gamma_k[e]\right\} \tag{4}$$

$$\gamma_k[e] = \pi_k[u(e);I] + \pi_k[c(e);I] \tag{5}$$

where $s^S(e)$ and $s^E(e)$ are the starting and the ending states of e, $\alpha_k[s^S(e)]$ and $\beta_k[s^E(e)]$ are the forward and backward metrics associated to $s^S(e)$ and $s^E(e)$ respectively [6] (see Fig. 1). The $\pi_k[c(e);I]$ term in (5) is computed as a weighted sum of the $\lambda_k[c;I]$ terms:

$$\pi_k[c(e);I] = \sum_{i}^{n_c} c_i(e)\lambda_k[c_i(e);I] \tag{6}$$

where $c_i(e)$ is one bit of the coded symbol associated to e and n_c is the number of bits forming a coded symbol. For a double binary CTC $n_c = 4$, $c_i(e) \in \{A, B, Y, W\}$ and the $\pi_k[u(e);I]$ terms are piece wise functions:

$$\pi_k[u(e);I] = \begin{cases} 0 & \text{if } u(e) = (\text{`0'},\text{`0'}) \\ \lambda_k^{\overline{A}B}[u(e),I] & \text{if } u(e) = (\text{`0'},\text{`1'}) \\ \lambda_k^{A\overline{B}}[u(e),I] & \text{if } u(e) = (\text{`1'},\text{`0'}) \\ \lambda_k^{AB}[u(e),I] & \text{if } u(e) = (\text{`1'},\text{`1'}) \end{cases} \tag{7}$$

As suggested in [24], Max-Log-MAP performance can be improved by introducing a scaling factor δ in the computation of $\lambda_k[u;O]$.

$$\lambda_k[u;O] = \delta\left(\overset{*}{\max}_{e:u(e)=u}\{b(e)\} - \overset{*}{\max}_{e:u(e)=\tilde{u}}\{b(e)\} - \lambda_k[u;I]\right) + (1-\delta)\pi_k[c^{u(e)};I] \tag{8}$$

where

$$\pi_k[c^{u(e)};I] = \begin{cases} 0 & \text{if } u(e) = (\text{`0'},\text{`0'}) \\ \lambda_k^{\overline{A}B\overline{YW}}[c,I] & \text{if } u(e) = (\text{`0'},\text{`1'}) \\ \lambda_k^{A\overline{BYW}}[c,I] & \text{if } u(e) = (\text{`1'},\text{`0'}) \\ \lambda_k^{\overline{A}B\overline{YW}}[c,I] + \lambda_k^{A\overline{BYW}}[c,I] & \text{if } u(e) = (\text{`1'},\text{`1'}) \end{cases} \tag{9}$$

is the systematic contribution of the intrinsic information (channel LLRs). For further details on the decoding algorithm, the reader can refer to [13] and [22].

3. Proposed Architecture

As stated in Section 1, the iterative nature of the CTC decoding algorithm makes the CTC decoder the system bottleneck. In fact, even if the input data for SISO1 are the output of SISO2 and viceversa, the CTC interleaver and deinterleaver scramble the data order creating a data dependency in the processing of the two SISOs. As a consequence, usually CTC VLSI architectures reuse the same hardware to perform the two SISOs operations. Alternatively, shuffling has been proposed as an effective method to introduce parallelism at the SISO level [25], [26].

In order to maximize the throughput, all the BCJR metric level parallelism strategies [26] can be employed to simultaneously compute all the branch metrics (BM) and all the state metrics (SM), namely the $\gamma_k[e]$ and $\alpha_k[s]$ or $\beta_k[s]$ values. Thus, a step in the trellis is performed in a clock cycle. As a consequence, the number of clock cycles required to complete the decoding of a WiMax frame made of N couples of bits (corresponding to N trellis steps) can be estimated as $D = 2(N + SISO_l)I$, where $SISO_l$ is the SISO latency and $2I$ is the number of half iterations. Given a certain clock frequency f_{clk}, the CTC decoder throughput T_{CTC-D}, defined as the number of decoded bits over the time required to complete the decoding process, for large values of N is approximately:

$$\lim_{N\to\infty} T_{CTC-D} = \lim_{N\to\infty} \frac{2N \cdot f_{clk}}{2I(N + SISO_l)} = \frac{f_{clk}}{I} \tag{10}$$

Given the number of iterations required to obtain satisfactory performance ($I \in [6, 10]$ as suggested in [17] and [22]), we can evaluate the throughput T_{CTC-D} as a function of the clock frequency. As a significant example let us consider the WiMax HUMAN(-OFDM) profile for 10 MHz channelization [1]: in the worst case, the downlink maximum throughput is $\hat{T}_{dl} \simeq$65 Mb/s. Thus, a clock frequency of about 400 MHz is required when only six iterations are performed. In order to ease ASIC backend design and to combat chip unreliability problems, the adoption of lower frequency parallel architectures is usually preferred. As a case of study we consider a target clock frequency $f_{clk} = 200$ MHz. Stemming from these requirements in the following subsections the architectures of the blocks depicted in Fig. 1 are detailed.

3.1. *Encoder-side Architecture*

The complete encoder architecture is obtained cascading the CTC encoder, the sub-block interleaver and the symbol selection with memory buffers. The high throughput imposed by the WiMax standard is sustained by the use of double buffers (shaded memories in Fig. 2) that grant a pipeline processing through the encoding chain. In Fig. 2 a high level block scheme of the complete encoder architecture is shown.

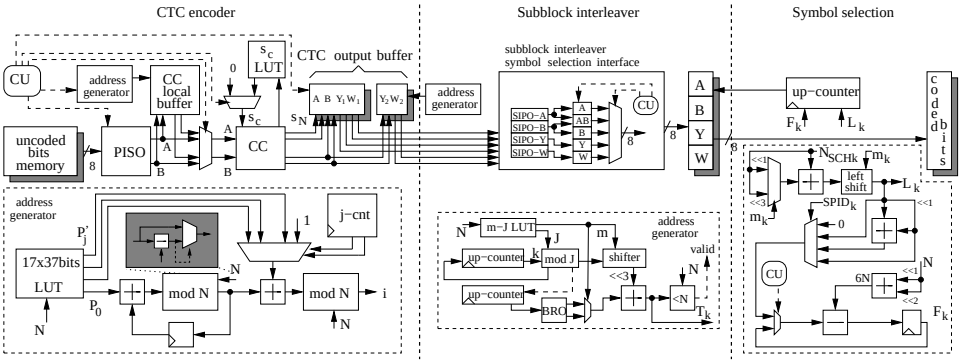

Figure 2. WiMax encoder block scheme.

3.1.1. *CTC encoder*

The WiMax CTC encoder is based on the parallel concatenation of two circular recursive systematic CCs; it receives a couple of input bits and outputs a six bit wide symbol. As a consequence, the CTC encoder architecture must be able to achieve a throughput of at least $3\hat{T}_{dl} \simeq 200$ Mb/s. Since each CC is circular recursive, a tailbiting strategy where the ending state matches the starting state [27] is employed. This state, usually referred to as circulation state $\underline{s}_c$, depends on N, namely

$$\underline{s}_c = (\underline{\underline{I}} + \underline{\underline{G}}^N)^{-1}\underline{s}_N \qquad \underline{\underline{G}} = \begin{bmatrix} 1\,0\,1 \\ 1\,0\,0 \\ 0\,1\,0 \end{bmatrix} \tag{11}$$

where $\underline{\underline{I}}$ is the identity matrix, $\underline{\underline{G}}$ is the matrix defining the WiMax constituent CC and $\underline{s}_N$ is the ending state obtained encoding the current N couples starting from the $\underline{s}_0 = [000]^T$ state (in the following we will refer to this encoding as dummy encoding).

For each frame the CTC encoder ought to perform:

- the dummy encoding of the in-order data to discover the corresponding circulation state
- the encoding of the in-order data starting the CC encoder from the circulation state
- the dummy encoding of the scrambled data to discover the corresponding circulation state
- the encoding of the scrambled data starting the CC encoder from the circulation state.

In order to reduce as much as possible the CTC encoder complexity, the reuse of the same CC encoder to perform both CC1 and CC2 is advisable. Even if the complexity of a single CC encoder is negligible, as detailed in section 4.2.1, the use of two or more CC encoders implies to at least double the CTC encoder memory

requirements. Given that the CTC encoder can process a couple per clock cycle, after $4N$ clock cycles a frame is coded. Thus, the CTC encoder output throughput is

$$T_{CTC-E} = \frac{6N \cdot f_{clk}}{4N} = 1.5 f_{clk} \tag{12}$$

In order to grant $T_{CTC-E} \geq 200$ Mb/s, we need $f_{clk} \geq 133$ MHz. This value is compatible with the target clock frequency ($f_{clk} = 200$MHz) reported in Section 3; then, the CTC encoder can be implemented as a single CC architecture managed by a simple control unit (CU), as shown in Fig. 2.

Since the uncoded bits are stored as bytes, a parallel to serial (PISO) register is employed to load the data from the memory and to properly feed the CTC encoder. The CC encoder is implemented as a simple linear feedback shift register (see the CC block in Fig. 1). As suggested in [1], all possible circulation states can be precalculated and stored into a LUT. The address to access this LUT is obtained using N mod 7 as the most significant bits and s_N as the least significant bits.

The CTC interleaver permutation algorithm specified in the WiMax standard is structured in two steps. The first step switches A and B stored at odd addresses. The second step provides the interleaved address i of the j-th couple as

$$i = (P_0 \cdot j + P_j^{'}) \bmod N \qquad j = 0, 1, \ldots, N-1 \tag{13}$$

where

$$P_j^{'} = \begin{cases} 1 & \text{when } j \bmod 4 = 0 \\ 1 + N/2 + P_1 & \text{when } j \bmod 4 = 1 \\ 1 + P_2 & \text{when } j \bmod 4 = 2 \\ 1 + N/2 + P_3 & \text{when } j \bmod 4 = 3 \end{cases} \tag{14}$$

P_0, P_1, P_2 and P_3 are constants taken from a table [1] and depend only on the number of couples N. It is worth pointing out that the two steps can be swapped. This allows to perform the first step on-the-fly, avoiding the use of an intermediate buffer to store switched couples.

The implementation of the CTC encoder interleaver can be derived as follows: if $x \in [0, 2 \cdot N - 1]$, x mod N can be implemented by means of a subtracter and a multiplexer. Unfortunately, $P_0 \cdot j + P_j^{'}$ is not granted to belong to $[0, 2 \cdot N - 1]$. As a consequence, several x mod N blocks ought to be cascaded to obtain i. However, the interleaver architecture can be simplified by rewriting (13) as

$$i = \{[(P_0 \cdot j) \bmod N] + (P_j^{'} \bmod N)\} \bmod N = [i_j^{'} + (P_j^{'} \bmod N)] \bmod N \tag{15}$$

where

$$i_j^{'} = \begin{cases} i_0^{'} = 0 & \text{when } j = 0 \\ i_j^{'} = (i_{j-1}^{'} + P_0 \bmod N) \bmod N & \text{when } j = 1, 2, \ldots, N-1 \end{cases} \tag{16}$$

A small Look-Up-Table (LUT) is employed to store P_0 mod N and the $P_j^{'}$ mod N terms; then, (15) is implemented by two parts as depicted in Fig. 2 (address

generator unit in the left side). The first part accumulates P_0 to implement the $P_0 \cdot j$ term and the mod N block produces the correct modulo N result. The second part employs the two least significant bits of a counter $(j - cnt)$ to select the proper $P_j^{'}$ mod N value, which is added to the $(P_0 \cdot j)$ mod N term. A further modulo N operation is performed at the output. Since in this architecture both the first and the second part work on data belonging to $[0, 2 \cdot N - 1]$, all the mod N operations are implemented by means of a subtracter and a multiplexer.

3.1.2. *Subblock interleaver*

The CTC encoder produces six subblocks of N bits (A, B, Y_1, W_1, Y_2, W_2). The subblock interleaver treats each subblock separately and scrambles its bits according to Algorithm 1, where m and J are constants specified by the standard [1], and $BRO_m(y)$ is the bit-reversed m-bit value of y. As a consequence, the number

Algorithm 1 Subblock interleaver address generation

```
1: k ← 0
2: i ← 0
3: while i < N do
4:    T_k ← 2^m (k mod J) + BRO_m(⌊k/J⌋)
5:    if T_k < N then
6:       i ← i + 1
7:    else
8:       discard T_k
9:    end if
10:   k ← k + 1
11: end while
```

of tentative addresses generated, N_M, can be greater than N. Exhaustive simulations show that the worst case is $N_M = 191$ that occurs with $N = 144$. Since $191/144 = 1.326$, a conservative approximation is $N_M = 4N/3$. The whole subblock interleaver architecture is obtained with one single address generator implementing Algorithm 1 to simultaneously read one bit from each of the six subblock memories. In particular, as imposed by the WiMax standard, the interleaved bits belonging to the A and B subblocks are stored separately, whereas the interleaved bits belonging to Y_1 and Y_2 are stored as a symbol-by-symbol multiplexed sequence, creating a "macro-subblock" made of $2N$ bits. Similarly a macro-subblock made of $2N$ bits is generated storing a symbol-by-symbol multiplexed sequence of interleaved W_1 and W_2 subblocks.

The throughput sustained by the proposed architecture, defined as the number

of bits output over the time required by the computation, can be estimated as:

$$T_{SI} = \frac{6N \cdot f_{clk}}{\frac{4N}{3}} = 4.5 f_{clk} \tag{17}$$

In order to grant $T_{SI} \geq 200$ Mb/s, we need $f_{clk} \geq 44$ MHz which is compatible with the target clock frequency fixed in Section 3.

To implement line 4 and 5 in Algorithm 1, three steps are required, namely the calculation of $k \bmod J$ and $\lfloor k/J \rfloor$, the calculation of $2^m(k \bmod J)$ and $BRO_m(\lfloor k/J \rfloor)$, the generation of T_k while checking $T_k < N$. It is worth pointing out that $k \bmod J$ can be efficiently implemented as an up-counter followed by a mod J block (see Fig. 2). Moreover, each time the mod J block detects $k = J$, a second counter is incremented: the final value in the second counter is $\lfloor k/J \rfloor$.

Since $m \in [3, 10]$ the $2^m(k \bmod J)$ term is implemented as a programmable shifter in the range $[0, 7]$ followed by a hardwired three position left shifter. The $BRO_m(\lfloor k/J \rfloor)$ term is obtained by multiplexing eight hardwired bit reversal networks. Finally, a valid T_k address is obtained with an adder and is validated by a comparator as shown in Fig. 2 (central part).

3.1.3. *Subblock interleaver - symbol selection interface*

The subblock interleaver processes six bits in parallel, one for each subblock. On the other hand, the symbol selection works on eight bit wide slices, as described in section 3.1.4. As a consequence, the interface between the subblock interleaver and the symbol selection must pack the bits into eight bit wide slices (see the central part of Fig. 2). Unfortunately several N values are not integer multiples of eight. Since every N is an integer multiple of four, the Y and W macro-subblocks are straightforwardly split in eight bit wide slices by means of two eight bit serial-to-parallel (SIPO) registers, followed by two output registers that are loaded when the SIPO registers have completed a slice, as depicted in Fig. 3. On the other hand, the A and B subblocks are managed with two SIPO registers and three output registers. The first register loads slices coming from the SIPO register labeled with A, the second register from the SIPO labeled B and the third from both. Thus, the AB output register contains the A and B subblocks fragments. A multiplexer selects the proper output register and stores the slices in the symbol selection input buffer. The correct scheduling of these blocks is managed by a simple CU. Y and W slices are ready to be loaded into the output registers every four clock cycles, whereas new A and B slices are loaded every eight clock cycles: this results in the scheduling reported in the timing diagram in Fig. 3. Every four clock cycles three cases are possible: i) only Y and W slices are in the output registers; ii) AB, Y and W slices are in the output registers; iii) A, B, Y and W slices are in the output registers. In the third case four clock cycles are required to store all the data in the output buffer, simultaneously new Y and W slices are prepared by the SIPOs, leading to a full throughput architecture.

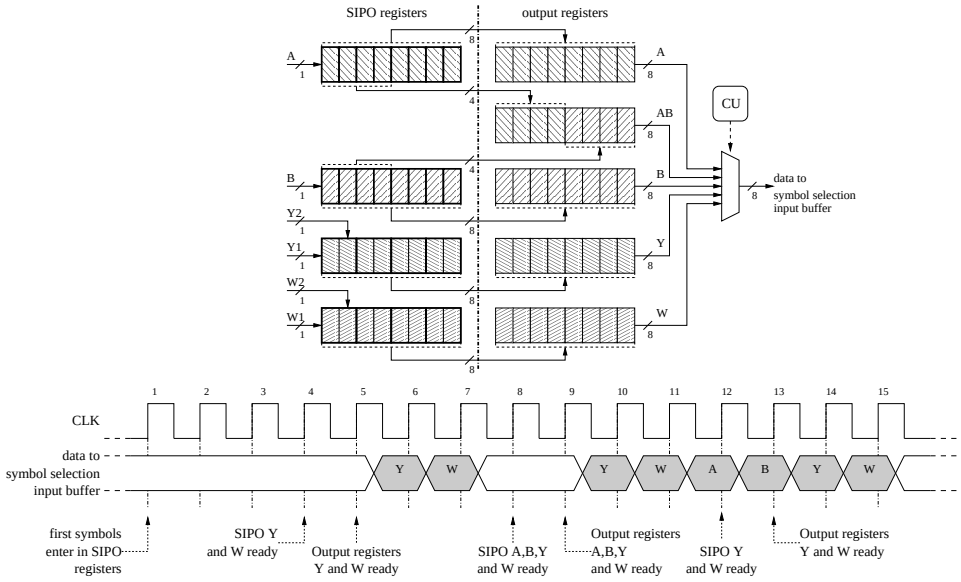

Figure 3. Subblock interleaver - symbol selection interface block scheme: most significant bit is on the right.

3.1.4. *Symbol selection*

The symbol selection (SS) chooses among the $6N$ coded bits the ones to be sent to the receiver. These bits are read from the subblock interleaver output buffer and the i-th address is

$$S_{k,i} = (F_k + i) \bmod 6N \qquad i = 0, 1, \ldots L_k - 1 \tag{18}$$

Thus, the symbol selection reads

$$L_k = 48 m_k \cdot N_{SCHk} \tag{19}$$

bits, starting from the location

$$F_k = (SPID_k \cdot L_k) \bmod 6N \tag{20}$$

where N_{SCHk}, m_k and $SPID_k$ are parameters specified by the standard for the k-index subpacket when HARQ is enabled, namely N_{SCHk} is the number of concatenated slots, m_k is the modulation order and $SPID_k$ is the subpacket ID [1].

Since $N_{SCHk} \in [1, 480]$ and $m_k \in \{2, 4, 6\}$, (19) can be rewritten as

$$L_k = \begin{cases} (2N_{SCHk} + N_{SCHk}) \cdot 2^5 \text{ when } m_k = 2 \\ (2N_{SCHk} + N_{SCHk}) \cdot 2^6 \text{ when } m_k = 4 \\ (8N_{SCHk} + N_{SCHk}) \cdot 2^5 \text{ when } m_k = 6 \end{cases} \tag{21}$$

The efficient implementation of (21) is obtained with an adder whose inputs are N_{SCHk} and the selection between two hardwired left shifted versions of N_{SCHk}

(one position and three positions), followed by a programmable left shifter (five-six positions) as shown in Fig. 2.

Similarly, since $SPID_k \in \{0,1,2,3\}$, the multiplication in (20) is avoided as

$$F_k = \begin{cases} 0 & \text{when } SPID_k = 0 \\ L_k \bmod 6N & \text{when } SPID_k = 1 \\ 2L_k \bmod 6N & \text{when } SPID_k = 2 \\ (2L_k + L_k) \bmod 6N & \text{when } SPID_k = 3 \end{cases} \tag{22}$$

Exhaustive simulations show that the implementation of mod $6N$ as a cascade of subtracters requires in the worst case twelve stages. Since this operation has to be performed only once at the beginning of the symbol selection procedure, a folded solution is adopted to save hardware resources. In order to better choose the proper trade-off between the number of clock cycles and the amount of resources, a throughput analysis is required. In the worst case, the symbol selection performs repetition and outputs up to $4 \times 6N$ bits, leading to a throughput of nearly 800 Mb/s. As a consequence, a one bit per cycle architecture is not feasible. Since both F_k and L_k are integer multiples of eight, the symbol selection is performed on slices of eight bits.

Let's consider a folded solution that produces F_k within twelve clock cycles: the proposed architecture outputs up to $24N$ bits in $24N/8+12$ clock cycles, achieving a maximum throughput

$$T_{SS} = \frac{24N \cdot f_{clk}}{\left(\frac{24N}{8} + 12\right)} \tag{23}$$

In the worst case ($N = 24$) $T_{SS} = 6.85 f_{clk}$: in order to grant $T_{SS} \geq 800$ Mb/s we need $f_{clk} \geq 117$ MHz which is lower than the target clock frequency.

3.2. *Decoder-side Architecture*

The symbol deselection, the subblock deinterleaver and the CTC decoder are connected together by means of memory buffers in order to guarantee that the non iterative part of the decoder (namely symbol deselection and subblock deinterleaver) and the decoding loop work simultaneously on consecutive data frames. In Fig. 4, a high level block scheme of the complete decoder architecture is shown.

3.2.1. *Symbol deselection*

Depending on amount of data sent by the symbol selection (puncturing or repetition), the throughput sustained by the symbol deselection (SD) can rise up to nearly 800 millions of LLRs per second. When the encoder performs repetition, the same symbol is sent more than once. Thus, the decoder combines the LLRs referred to the same symbol to improve the reliability of that symbol. As shown in Fig. 4 this can be achieved partitioning the symbol deselection input buffer into

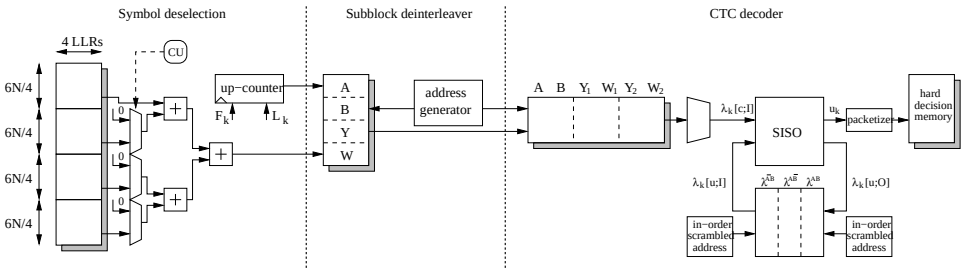

Figure 4. WiMax decoder block scheme.

four memories, each of which containing up to $6N$ LLRs. Since the proposed symbol deselection architecture can read up to four LLRs per clock cycle, it reduces the incoming throughput to about 200 millions of LLRs per second. However, the symbol deselection has to compute (20) and (19) to find the output buffer starting address and the number of elements to be written. Furthermore, in order to support the puncturing mode, the output memory locations corresponding to unsent bits must be set to zero. To ease the proposed architecture implementation, all the output memory locations are set to zero while L_k and F_k are computed. As a consequence, about two clock cycles per sample are required to complete the symbol deselection, namely $6N$ LLRs are output in $12N$ clock cycles. So that the symbol deselection throughput can be estimated as

$$T_{SD} = \frac{6N}{12N} f_{clk} = 0.5 f_{clk} \tag{24}$$

Unfortunately $T_{SD} < 200$ Millions of LLRs with f_{clk}=200 MHz. To overcome this problem we impose not only to partition the input buffer into four memories, but also to increase the memory parallelism, so that each memory location contains p LLRs. The throughput sustained by this solution is approximately

$$T_{SD} = \frac{6N \cdot f_{clk}}{\frac{12N}{p}} = \frac{p \cdot f_{clk}}{2} \tag{25}$$

A conservative choice is $p = 4$. Thus, the proposed symbol deselection architecture processes simultaneously up to sixteen LLRs.

3.2.2. *Subblock deinterleaver*

The subblock deinterleaver is implemented resorting to the same address generator described in Section 3.1.2, where T_k output is used as the write address. Since all the subblocks can be processed simultaneously, the proposed architecture deinterleaves six LLRs per clock cycle. As a consequence, the subblock deinterleaver sustains a throughput of $4.5 f_{clk}$ LLRs per second, namely up to 900 Millions of LLRs per second with f_{clk}=200 MHz.

3.2.3. *CTC decoder*

The analysis presented at the beginning of Section 3 can be exploited to derive the degree of parallelism required by the CTC decoder to sustain a maximum throughput $T \geq \hat{T}_{dl}$. As suggested in [26], each frame of received LLRs can be divided into P slices that are independently processed by P SISOs working in parallel. As a consequence, the number of clock cycles required to completely decode a frame made of N couples of bits is $D = 2(N/P + SISO_l)I$ and the achievable throughput is

$$T_{CTC-D} = \frac{2N \cdot f_{clk}}{2I(\frac{N}{P} + SISO_l)} \tag{26}$$

Since we adopt a sliding window based approach [28], where boundary metrics are inherited from an iteration to the next one, as proposed in [15] and [29], we obtain $SISO_l$=$W + \Delta$ where W is the window size and Δ is the pipeline depth of the $\lambda - O$ processor in Fig. 8. In fact, as it can be inferred from Fig. 8, we adopt the following scheduling where the forward and backward recursions are pipelined over consecutive windows [30]. The SISO performs the forward recursion and stores the results in the α-MEM temporary buffer; then it performs the backward recursion and the computation of $\lambda_k[u; O]$.

Assuming W=32 [29], Δ=5, I=8, f_{clk}=200 MHz, we can estimate the throughput of the decoder for the 17 possible values of N [1]. As shown in Fig. 5, P=3 allows to achieve $\hat{T}_{dl}$ (horizontal solid line) only for $N \geq$960, whereas with P=4 the target throughput can be reached for $N > 250$ (i.e. $N \geq$480, which is the next specified size).

Interleaver-Deinterleaver (Π, Π^{-1}) As pointed out in Section 1, a parallel CTC decoder can lead to memory collisions during scrambled half iterations. In fact, in in-order half iterations, the n-th SISO accesses only the n-th memory, whereas in scrambled half iterations the n-th SISO reads from and writes to different memories. A collision occurs when two or more SISOs try to simultaneously access the same memory. The rule employed by the SISOs to access the memories is imposed by the interleaver, which is devoted to maximize input symbols distance [31]. Thus, given the set of interleaver laws to be supported, the memory collision has to be studied case by case. For this reason concurrent interleaving [8], [9], [32] is a distinguishing feature of parallel turbo decoders.

Exhaustive simulations for the WiMax CTC show that collisions occur for P=2 and P=4 only with N=108. As a consequence, we select P as a function of N to simultaneously obtain a monotonically increasing throughput as a function of N and to avoid collisions. It is worth pointing out that, when collisions are avoided, the resulting parallel interleaver is a circular shifting interleaver [33]: the address generation is simplified with all SISOs simultaneously accessing the same location of different memories. Said idx_t^0 the memory accessed by SISO-0 at time t during

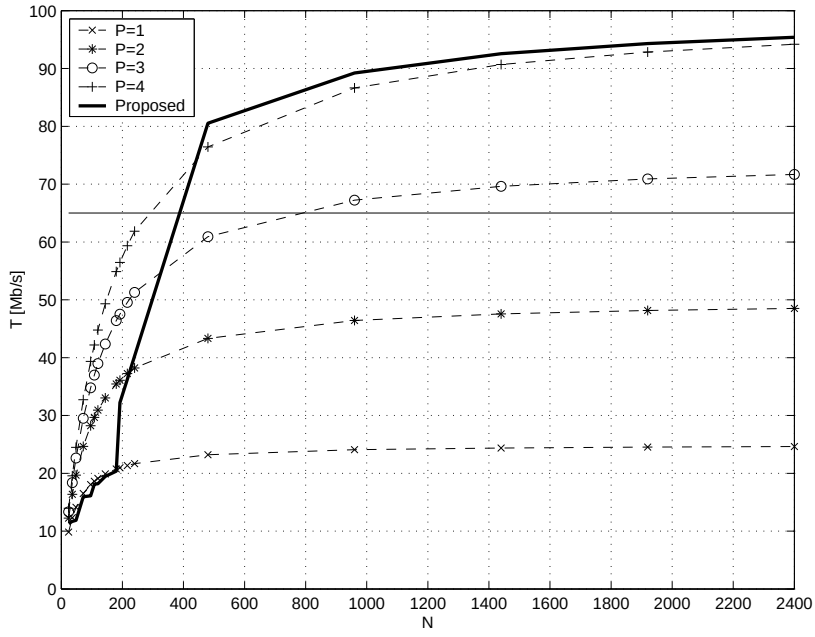

Figure 5. Parallel CTC decoder throughput as a function of the block size N for different parallelism degree values P. The horizontal line represents the target throughput.

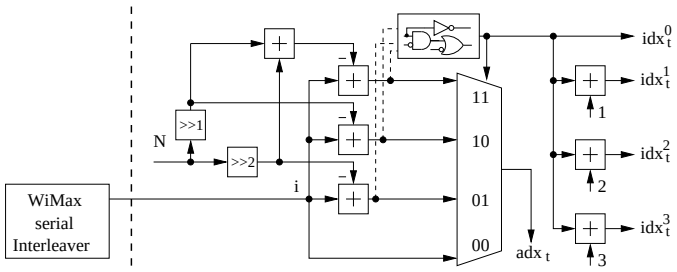

Figure 6. WiMax parallel interleaver address generator architecture.

a scrambled half iteration, the memory concurrently accessed by SISO-k is

$$\mathrm{idx}_t^k = (\mathrm{idx}_t^0 \pm k) \bmod P \tag{27}$$

Thus, the parallel CTC interleaver-deinterleaver system is obtained as a cascaded two stage architecture (see Fig. 6). The first stage efficiently implements (13), whereas the second one extracts the common memory address adx_t and the memory identifiers idx_t^k from the scrambled address i. The implementation of the first stage is described in Section 3.1.1.

The second stage of the proposed parallel interleaver address generator archi-

tecture works as follows. Since $\mathrm{adx}_t \in [0, N/P-1]$, it can be obtained from the scrambled address i produced by the first stage as

$$\mathrm{adx}_t = \begin{cases} i & \text{when } i \in [0, \frac{N}{P}-1] \\ i - \frac{N}{P} & \text{when } i \in [\frac{N}{P}, 2\frac{N}{P}-1] \\ \dots & \\ i-(P-1)\frac{N}{P} & \text{when } i \in [(P-1)\frac{N}{P}, N-1] \end{cases} \tag{28}$$

The straightforward implementation of (28) needs to calculate N/P and to allocate $P-2$ multipliers, $P-1$ subtracters, a P-ways multiplexer and few logic for selecting the proper adx_t value. The N/P division can be simplified by choosing the possible P values as powers of two. Thus, the proposed CTC decoder architecture exploits throughput/parallelism scalability to avoid collisions, namely we employ: P=1 when $N \leq 180$, P=2 when $192 \leq N \leq 240$ and P=4 when $480 \leq N \leq 2400$. Moreover, as it can be inferred from Fig. 6, multiplications are avoided resorting to simple shift operations ($x >> i = x/2^i$). The sign of the subtractions (dashed lines in Fig. 6) allows not only to select the proper adx_t but also to find idx_t^0. Then, with $P-1$ modulo P adders the other idx_t^k values are straightforwardly generated according to (27). As it can be observed, choosing P as a power of two reduces the modulo P adders to simpler, binary adders. We also impose the condition $N/(P \cdot W) \in \mathrm{N}$, which implies that each SISO processes the same number (NW_P) of windows in a data frame. This guarantees not only a 100% hardware utilization, but also full synchronization of SISOs, which results in a simpler control unit.

The proposed architecture is employed to implement the interleaver reading part. Since idx_t^k identifies the memory accessed by SISO-k at time t, the parallel interleaver architecture ought to signal to the memory which SISO is requiring the data. This operation is accomplished by a 4×4 crossbar switch (radx-switch) controlled by idx_t^k with 2 bit wide fixed inputs, as shown in Fig. 7. When the idx^k memory (EI-MEM idx^k) is read, it sends back the corresponding $\lambda[u]$ triplet to SISO-k, through a 4×4 crossbar switch (rdata-switch). This crossbar switch is controlled by the output of the radx-switch. Since each SISO outputs its data in reverse order [22], during the reading operation idx_t^k and adx_t are stored into a LIFO; idx_t^k and adx_t are read from the LIFO during the writing operation to configure a 4×4 crossbar switch (wdata-switch). The LIFO stores W words, each of which contains the four configurations for the crossbar, $4 \times 2 = 8$ bit, and the common memory location whose size depends on NW_P and W, so the number of bit required to correctly represent the common memory location address is $\lceil \log_2(W \cdot NW_P) \rceil$.

Parallel SISO architecture The global architecture of the designed parallel SISO is given in Fig. 8. Each SISO contains several processors devoted to compute the different metrics required by the BCJR algorithm; namely the α processor implements (3) and the β processor implements (4) on two consecutive windows of data. In order to perform a trellis step in one clock cycle, both the α and β processors compute in parallel all the eight new State Metrics (SMs) (see the SM proc. block

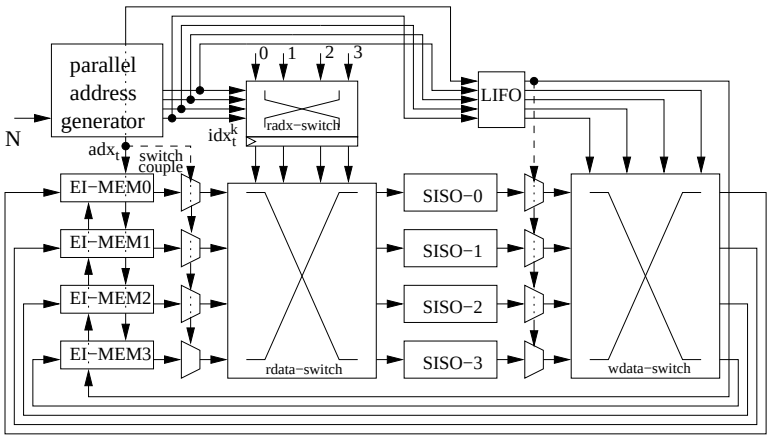

Figure 7. WiMax parallel CTC interleaver architecture.

in Fig. 8). Since the α processor works in direct order on the input data (BCJR forward recursion), whereas the β processor updates them in reverse order (BCJR backward recursion), two Branch Metrics Units (BMUs), are placed before the α and β processors. Each BMU is devoted to combine the three $\lambda_k[u;I]$ and the four $\lambda_k[c;I]$ and obtains in parallel the BMs γ_k associated to the k-th trellis section. As a consequence, a local buffer (BMU-MEM) is required to store W words each of which is made of three $\lambda_k[u;I]$ and four $\lambda_k[c;I]$. The $\lambda - O$ processor generates the $\lambda_k[u;O]$ values according to (8), receiving the β_k values directly from the β processor and loading the α_k values from a local buffer (α-MEM). We fixed $\delta = 0.75$ in (8) to ensure both good performance of Max-Log-MAP algorithm (see Fig. 12) and simplified hardware implementation. In fact, the scaling factor multiplication (scal. fact. in Fig. 8) is implemented by two adders and two hardwired shifters as $[(2a + a) + b]/4$. Moreover, the $\lambda - O$ processor includes the logic to calculate the hard decision bits.

The use of the sliding window approach implies that the BMU-MEM contains W words, each word being made of four channel LLRs ($\lambda_k[c;I]$) represented on $n_{\lambda c}$ bits and three extrinsic LLRs ($\lambda_k[u;I]$) represented on $n_{\lambda u}$ bits. Similarly, the α-MEM contains W words, each word being made of 8 SMs ($\alpha_k[s]$) represented on n_{SM} bits. However, the implementation of the β metrics inheritance strategy (intra SISO SMs inheritance) comes at the expenses of additional memory. In fact, since each SISO processes NW_P windows, an NW_P-1 words local memory (β-LOC-MEM) is required to store the SMs at the boundary of two consecutive windows (β^{prv}). Each word is made of eight SMs, each of which is represented on n_{SM} bits. Moreover, at every half iteration, each SISO requires to properly initialize its trellis portion (α^{in} and β^{in}). The correct initialization of each trellis portion is obtained by inheritance: each SISO employs the boundary SMs calculated at the previous iteration by its neighboring SISOs (inter SISO SMs inheritance). This can

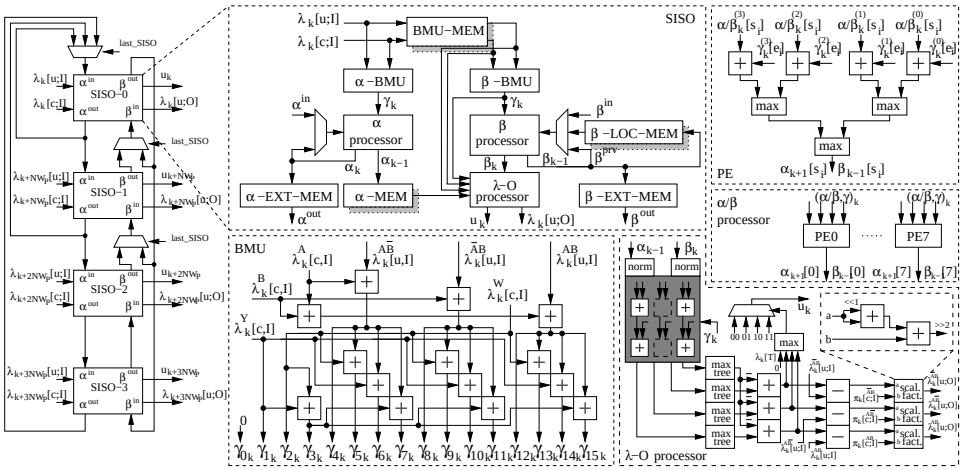

Figure 8. WiMax CTC parallel SISO architecture.

be achieved by inserting two 2-position shift registers, one for each half iteration (α-EXT-MEM and β-EXT-MEM in Fig. 8) to exchange the α^{out} and β^{out} SMs with the neighboring SISOs. Furthermore, inter SISO SMs inheritance between SISO-0 and SISO-3 allows to obtain a reliable estimation of the circulation state avoiding training operations [15]. The simple network required to implement inter SISO SMs inheritance for the variable parallelism architecture is depicted in Fig. 8 on the left side.

It is worth pointing out that the choice of the window size W, discussed in the previous paragraph, impacts not only on the decoder performance, but also on the decoder complexity. In fact, even if $N/(P \cdot W) \in \mathrm{N}$ allows to simplify SISOs synchronization, other requirements ought to be taken into proper account:

- the converge speed of the iterative decoding is not reduced significantly by choosing the minimum window size to be at least (preferably more than) six times (W_{fact}) the CC constraint length (CC_l) [34]: $W \geq W_{fact} \cdot CC_l = 6 \cdot 4 = 24$
- the throughput (26) is maximized keeping W as small as possible
- the window size W impacts on the amount of memory required by each SISO.

The total amount of memory needed for a SISO is a function of W and it is composed by the total number of bits required by each SISO memory buffer, namely the BMU-MEM, the α-MEM and the β-LOC-MEM: $M_{SISO} = M_{BMU} + M_\alpha + M_\beta$ where

$$M_{BMU} = (3n_{\lambda u} + 4n_{\lambda c})W \tag{29}$$

$$M_\alpha = 8n_{SM}W \tag{30}$$

$$M_\beta = 8n_{SM}\left(\frac{N}{P \cdot W} - 1\right) \tag{31}$$

The optimal W value to minimize M_{SISO} is

$$W_{opt} = \sqrt{\frac{8n_{SM}N}{P(3n_{\lambda u} + 4n_{\lambda c} + 8n_{SM})}} \tag{32}$$

Input/Output interface The CTC decoder input interface is simple as it is based on a double buffer. In fact one buffer stores the currently decoded frame LLRs ($\lambda[c;I]$), whereas the other acts as the subblock deinterleaver output buffer. In order to grant parallel access to the input buffer, it is partitioned into P independent memories. Furthermore, to simplify the access to the input buffer, $\lambda[c;I]$ are read only in natural order. On the contrary, the CTC decoder output buffer is more complex. Since the proposed CTC decoder starts the decoding process from SISO2 (scrambled order), during the last half iteration the hard decision bits produced by the SISOs (in-order) must be stored in the output memory. However, each SISO generates windows of hard decisions where the couples into each window are in reverse order (according to the backward recursion). Moreover, the number of active SISOs depends on the current frame size N. As a consequence, given the current number of active SISOs, a packetizer is devoted to collect the hard decisions and store them into the output memory. Since each WiMax frame contains an integer number of bytes (i.e. $2N/8 \in \mathrm{N}$), we implemented the packetizer as four SIPO registers, each of which can accommodate eight bits. Depending on the current number of active SISOs, the proper shift registers are enabled. In the worst case ($P = 4$), during the last half iteration four bytes of hard decisions ought to be stored every four clock cycles. When the SIPO0 register data is ready, it is stored into the output memory. Simultaneously, the SIPO0 register is ready to accommodate the first couple of bits of the next byte. On the other hand, to make the other SIPO registers ready to accommodate new bits, their content is moved into three output registers, as depicted in Fig. 9. Thus, within the next three clock cycles, a multiplexer selects the output register values that are stored in the output memory while the SIPO registers are ready with four new bytes.

As far as the output memory address is concerned, since every SISO produces the same amount of hard decisions, given N and P, we precalculate the starting addresses and store them into a LUT, namely the k-th SISO starting address is

$$adx_s = k\frac{N}{4P} = k\frac{W \cdot NW_P}{4} \tag{33}$$

In particular, if $W/4 \in \mathbf{N}$, every window contains an integer number of bytes and $W/4$ is the offset between two consecutive windows of bytes. A simple architecture to calculate the output memory address is obtained adding together a base address and an offset. The base address is implemented as four up-counters (one for each SISO) that start from adx_s and are updated adding $W/4$ each time a window of bytes is stored in the output memory. A multiplexer selects the k-th base address

when the byte to be written in the output memory comes from the k-th SISO. Since the hard decision are output according to the backward recursion order, the offset is implemented as mod $W/4$ down-counter. A simple CU is devoted to properly drive the multiplexer selectors and reset the down-counter.

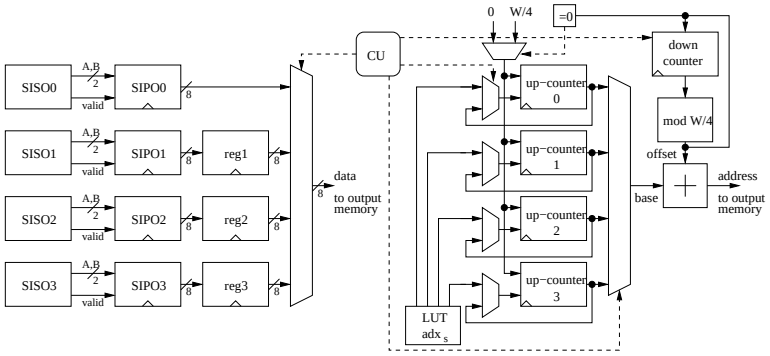

Figure 9. CTC decoder packetizer architecture.

4. General guidelines, implementation results and comparison

Stemming from the architectural choices employed in Section 3 for the WiMax CTC encoder and decoder design, we derive some general guidelines and then we show the actual results obtained for the WiMax standard.

4.1. *General guidelines*

Several considerations detailed in Section 3 have been discussed for the double binary WiMax encoder and decoder architecture. However, they can be extended to the design of a general double binary CTC system.

To reduce the amount of resources required by the CTC encoder we employ a folded architecture where one CC is reused to implement both the in-order and the scrambled coding. The use of a folded architecture is driven by a throughput based analysis, namely (12) can be rewritten for a general case to understand if a folded architecture is suited for a given application. Said R the code rate of a single CC, the number of bits output in a frame by a double binary CTC encoder is $2N(2/R-1)$. An architecture that performs one trellis step per cycle, as the one detailed in this work, requires $2N$ clock cycles to perform both in-order and scrambled coding operations and $2N$ clock cycles for the two dummy encodings. Thus, said $\hat{T}_{CTC-E}$ the throughput required by the application, the clock frequency can be obtained as

$$f_{clk} \geq \frac{2R}{2-R}\hat{T}_{CTC-E} \tag{34}$$

On the other hand, the throughput of the CTC decoder can be increased by introducing parallelism. As detailed in Section 3, choosing P as a power of two grants a significant simplification in the decoder design. Said $\hat{T}_{CTC-D}$ the throughput required by the application, the clock frequency can be obtained as

$$f_{clk} \geq \frac{I}{N} \cdot \hat{T}_{CTC-D} \cdot \left(\frac{N}{P} + W + \Delta \right) \tag{35}$$

It is known that in binary turbo code decoders LLRs are commonly used. On the contrary, in double binary turbo decoders the use of logarithmic probabilities (LP) instead of LLRs allows to save a certain amount of logic in the SISO architecture [15]. However, the use of 4 LPs instead of 3 LLRs has a negative impact on both the interleaver memory and the SISO memory footprint. In order to select the most suitable approach, we implemented both the LLR based SISO (SISO-LLR) and the logarithmic probabilities based SISO (SISO-LP) in VHDL and synthesized them on a 0.13 μm standard cell technology considering the worst case for the window size $W = 48$. Moreover, we generated the dual port SRAMs to implement the interleaver memory both for the SISO-LLR (2p-LLR) and the SISO-LP (2p-LP) and the single port SRAMs to implement SISO memory as a "ping-pong" buffer for both the cases (1p-LLR and 1p-LP). Fig. 10 shows the complexity growth of SISO-LLR, SISO-LP,

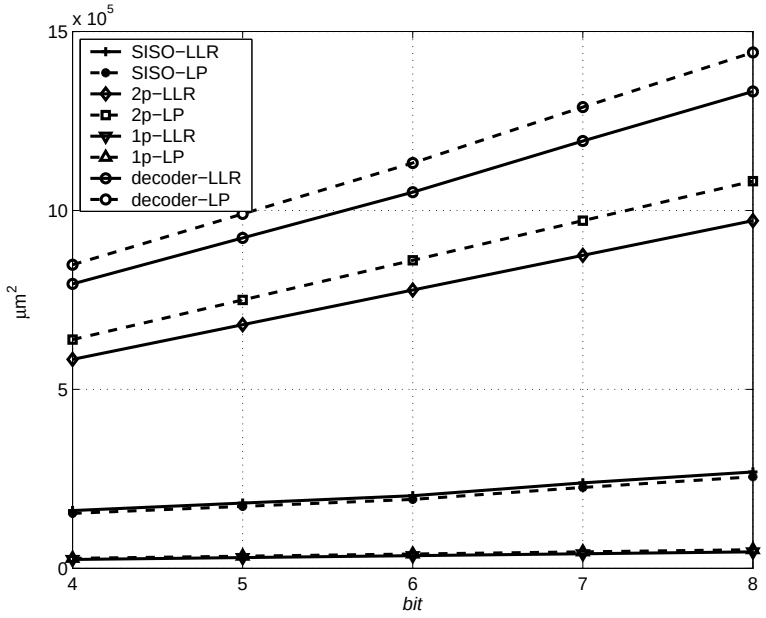

Figure 10. Complexity growth [μm^2] of a double binary turbo decoder building blocks as a function of the number of bits (*bit*) to represent the input data.

2p-LLR, 2p-LP, 1p-LLR and 1p-LP in μm^2 as a function of the number of bits

(*bit*) used to represent the LLRs or the LPs. The range explored in this analysis ($bit \in [4, 8]$) shows that the SISO-LLR complexity is slightly larger than that of the SISO-LP. However, the amount of memory required by a SISO-LP based decoder increases more than that of a SISO-LLR based one. In Fig. 10 the complexity of a single SISO decoder, including the interleaver, is also shown. Further experiments show that increasing P, the overhead required by the LP based decoder with respect to the LLR based one decreases from 7.6% ($P = 1$) to 2.2% ($P = 8$). However, the LLR based decoder is still less complex.

Moreover, to properly design the CTC decoder architecture the size of the window is extremely important as it impacts not only on the decoder throughput but also on the complexity. As discussed in Section 3, a good choice is $W_m \leq W \leq W_M$ with

$$W_m = \min\{W_{fact} \cdot CC_l, N\} \tag{36}$$

$$W_M = \max\{W^*, W_{opt}\} \tag{37}$$

$$W^* = \min_W \left\{ \frac{N}{P \cdot W} \in \mathbf{N} : \quad W \geq W_m \right\} \tag{38}$$

$$W_{opt} = \sqrt{\frac{\#s \cdot n_{SM} \cdot N}{P(\#\lambda u \cdot n_{\lambda u} + \#\lambda c \cdot n_{\lambda c} + \#s \cdot n_{SM})}} \tag{39}$$

where $\#s$ is the number of states, $\#\lambda u$ is the number of extrinsic information LLRs and $\#\lambda c$ is the number of channel LLRs. However, to have the throughput T monotonically increasing with N, W values are chosen as shown in Table 1 Of course this choice satisfies the worst case maximum throughput $\hat{T}_{dl}$ only for $N \geq 480$, whereas for $N < 480$ graceful throughput reduction is achieved.

For the WiMax CTC (39) becomes (32) and, according to the literature [23], [35], we choose $n_{\lambda u} = 6$ and $n_{\lambda c} = 8$. As a consequence, eleven bits are required for the BMs representation. Moreover, resorting to the SM wrapping technique proposed in [36], we obtain $n_{SM} = 12$. Fig. 11 shows that M_{SISO} as a function of W has a greater slope for $W < W_{opt}$ than for $W > W_{opt}$. As a consequence, if other conditions impose to select $W \neq W_{opt}$ our analysis suggests that $W \geq W_{opt}$ is preferred. Since $W_{opt} \propto \sqrt{\frac{N}{P}}$, Fig. 11 shows the curves obtained for all the different N/P values.

4.2. *VLSI architecture implementation results and comparison*

The complete encoder and decoder architectures, described as parametric VHDL modules, have been synthesized with Synopsys Design Compiler on a 0.13 μm standard cell technology.

4.2.1. *Encoder side*

Post synthesis results show that the CTC encoder requires 3.5 kgate of logic and $2N = 4.8$ kbit of memory for its local buffer, whereas the subblock interleaver, the

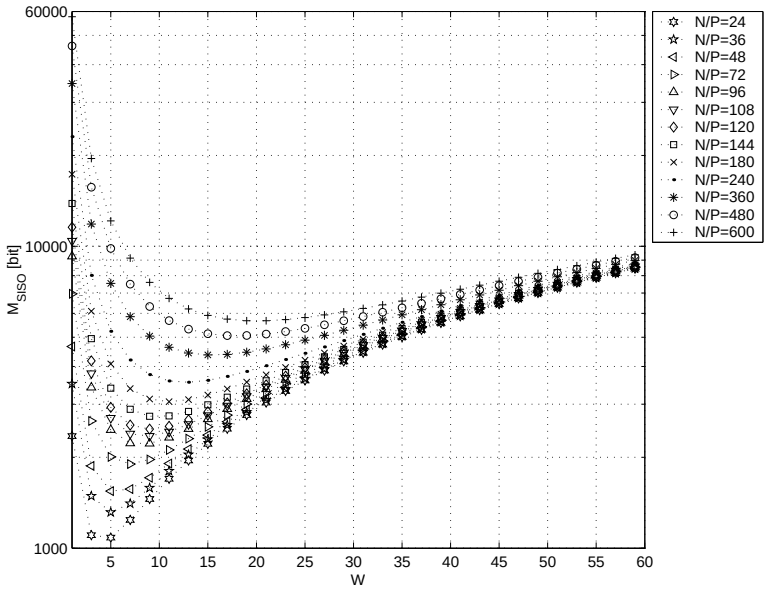

Figure 11. Amount of memory required by each SISO (M_{SISO}) in the proposed parallel CTC decoder as a function of the window size (W) for different N/P ratios.

symbol selection and their interface block require 1.7 kgate, 2.2 kgate and 1.8 kgate respectively. In order to accommodate the maximum frame ($N = 2400$), the double input buffer requires $2 \times 2N = 9.6$ kbit of memory. Similarly, the CTC encoder double output buffer requires $2 \times 6N = 28.8$ kbit. The same amount of memory is required to store the data after the subblock interleaver. Further $2 \times (4 \cdot 6N) = 115.2$ kbit of memory are required for the symbol selection double output buffer. Thus, as summarized in Table 2 the complete encoder architecture requires 9.2 kgate of logic and 187.2 kbit of memory.

4.2.2. *Decoder side*

As pointed out through Section 3, the CTC decoder is the most critical part of the design as it requires a noteworthy amount of resources. In particular the choice of P and W has a significant impact on both performance and complexity. Of course, proper P and W must be selected for each frame size (17 cases for WiMax [1]), as reported in Table 1, where NW_P and the resulting throughput T_{CTC-D} (bold line in Fig. 5) are also given. For the sake of completeness Table 1 also shows the W_{opt} value obtained from (32). It is worth pointing out that the actual window size W has been selected considering not only W_{opt} but also the other conditions discussed in Section 3. As detailed in Table 2 the complete decoder architecture requires 167.7 kgate of logic and nearly 1.2 Mbit of memory, where 11 kgate are devoted to the symbol deselection, 1.7 kgate to the subblock deinterleaver and the parallel CTC decoder

Table 1. Parallelism degree (P), actual window size (W), the optimal window size (W_{opt}) obtained from (32), number of windows per SISO (NW_P) and throughput (T_{CTC-D}) achieved by the WiMax CTC decoder parallel architecture for the 17 N values.

N	P	W	W_{opt}	NW_P	T [Mb/s]
24	1	24	4	1	11.3
36	1	36	5	1	11.7
48	1	48	6	1	11.9
72	1	36	7	2	15.9
96	1	48	8	2	16.1
108	1	36	9	3	18.1
120	1	40	9	3	18.2
144	1	36	10	4	19.5
180	1	36	11	5	20.4
192	2	48	8	2	32.2
216	2	36	9	3	36.2
240	2	24	9	5	40.3
480	4	24	9	5	80.5
960	4	24	13	10	89.2
1440	4	24	16	15	92.5
1920	4	24	18	20	94.3
2400	4	24	20	25	95.4

requires 155 kgate and 116.2 kbit. The complete decoder memory requirement takes into account: the symbol deselection double buffer ($2 \times (4 \cdot 6N \cdot n_{\lambda c}) = 691.2$ kbit), the buffer between the symbol deselection and the subblock deinterleaver ($2 \times (6N \cdot n_{\lambda c}) = 172.8$ kbit), the CTC decoder input buffer (172.8 kbit) and the CTC decoder output buffer ($2 \times 2N = 9.6$ kbit)

Each SISO requires about 37 kgate for the logic and 14.2 kbit ($2 \times M_{SISO}$) for its local "ping-pong" buffers (BMU-MEM, α-MEM, β-LOC-MEM). The serial address generator to produce the interleaved addresses requires about 1.5 kgate, whereas the complete parallel interleaver depicted in Fig. 7 requires 2.8 kgate for the logic, 1728 bit for the LIFO ($2 \times [8 + \lceil \log_2(W \cdot NW_P) \rceil] \cdot W$) and 57.6 kbit ($3N \cdot n_{\lambda u}$) for the extrinsic information memory (EI-MEM); the CTC packetizer requires 4.2 kgate.

Even if in the literature several works deal with the design of VLSI architectures for turbo decoders, few of them concern the implementation of double binary CTC decoders. Significant examples are [14], [17] and [18]. The proposed architecture shows excellent performance and complexity figures compared both to a custom implementation [14] and to a programmable solution [18]; ([18]-I refers to the single processor solution, whereas [18]-II is related to the 16 processor architecture). Moreover, each SISO in the proposed architecture is slightly less complex than the one described in [17] in terms of logic even taking into account the resources required by the serial

Table 2. Architectures comparison: CMOS technology process (TP), logic (L), memory (M), clock frequency (f_{clk}) and throughput (T). The proposed architectures results are highlighted in bold.

Architecture		TP [μm]	L [kgate]	M [kbit]	f_{clk} [MHz]	T [Mb/s]
Enc.	**CTC, our**	**0.13**	**3.5**	**4.8**	**200**	**300**
	SI/D, our	**0.13**	**1.7**	**0**	**200**	**900**
	SS, our	**0.13**	**2.2**	**0**	**200**	**1600**
	Enc. our	**0.13**	**9.2**[a]	**187.2**[b]	**200**	**1600**
Dec.	**SD, our**	**0.13**	**11**	**0**	**200**	**400**[c]
	CTC [14]	-	480	713	200	-
	CTC [18]-I	0.09	97	-	335	7.4
	CTC [18]-II	0.09	1552	-	335	100
	CTC [17]	0.18	51	11.7	200	24.26
	CTC II [17]	0.18	1.2	-	200	24.26 [d]
	SISO, our	**0.13**	**37**	**14.2**	**200**	**95.39**[d]
	CTC II, our	**0.13**	**2.8**	**59**	**100**	**95.39**[d]
	CTC, our	**0.13**	**155**	**116.2**	**200**	**95.39**
	Dec. our	**0.13**	**167.7**	**1163**[b]	**200**	**95.39**

[a] Including the 1.8 kgate required by the subblock interleaver - symbol selection interface.
[b] Including all the input/output buffers required.
[c] The throughput is in Millions of LLR per second.
[d] The throughput is referred to the complete decoder.

address generator. On the other hand, as the architecture proposed in [17] uses SMs quantization, it has better memory requirements figure.

Finally, in Fig. 12 (star-dashed curve) we show as a significant example the Bit Error Rate (BER) at different Signal to Noise Ratios (SNR) for the case $N = 2400$, $P = 4$, $W = 24$, $n_{\lambda u} = 6$, $n_{\lambda c} = 8$, $\delta = 0.75$ using the Max-Log-MAP algorithm. The circle-dashed curve represents the results obtained from our floating point software model for $N = 2400$, $P = 4$, where each SISO processes a single window, namely $W = 600$ and $NW_P = 1$, with the Log-MAP algorithm. As it can be inferred, the proposed architecture features extremely reduced SNR loss (less than 0.15 dB) compared with the floating point Log-MAP case.

5. Conclusions

In this paper design criteria for double binary CTC encoder and decoder architectures have been presented. Moreover, the VLSI implementation of optimized architectures for the WiMax complete encoder and decoder are described. The proposed architectures sustain a decoded bits throughput of more than 90 Mb/s with

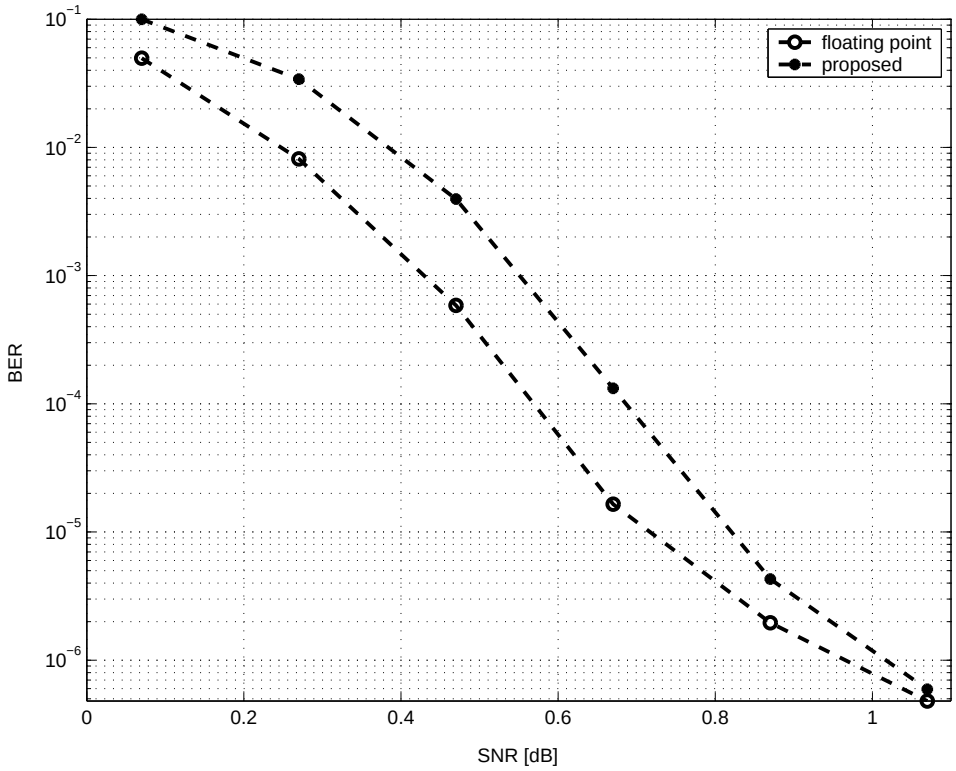

Figure 12. Proposed WiMax CTC decoder performance: BER as a function of the SNR

a clock frequency of 200 MHz requiring 9.2 kgate of logic and 187.2 kbit of memory for the complete encoder and 167.7 kgate and 1163 kbit for the complete decoder.

Bibliography

1. "IEEE Std 802.16, part 16: air interface for fixed broadband wireless access systems," Oct. 2004.
2. G. D. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, Mar 1973.
3. R. M. Pyndiah, "Near-optimum decoding of product codes: block turbo codes," *IEEE Transactions on Communications*, vol. 46, no. 8, pp. 1003–1010, Aug 1998.
4. C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error correcting coding and decoding: Turbo codes," in *IEEE International Conference on Communications*, 1993, pp. 1064–1070.
5. R. G. Gallager, "Low density parity check codes," *IRE Transactions on Information Theory*, vol. IT-8, no. 1, pp. 21–28, Jan 1962.
6. L. 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. 3, pp. 284–287, Mar 1974.
7. A. Giulietti, B. Bougard, V. Derudder, S. Dupont, J. W. Weijers, and L. V. der Perre,

"A 80 mb/s low-power scalable turbo codec core," in *IEEE Custom Integrated Circuits Conference*, 2002, pp. 389–392.
8. M. J. Thul, N. Wehn, and L. P. Rao, "Enabling high-speed turbo-decoding through concurrent interleaving," in *IEEE International Symposium on Circuits and Systems*, 2002, pp. 897–900.
9. F. Speziali and J. Zory, "Scalable and area efficient concurrent interleaver for high throughput turbo-decoders," in *IEEE Euromicro Symposium on Digital System Design*, 2004, pp. 334–341.
10. C. Berrou, Y. Saouter, C. Douillard, S. Kerouedan, and M. Jezequel, "Designing good permutations for turbo codes: towards a single model," in *IEEE International Conference on Communications*, 2004, pp. 341–345.
11. A. Tarable and S. Benedetto, "Mapping interleaving laws to parallel turbo decoder architectures," *IEEE Communications Letters*, vol. 8, no. 3, pp. 162–164, Mar 2004.
12. A. Tarable, L. Dinoi, and S. Benedetto, "Design of prunable interleavers for parallel turbo decoder architectures," *IEEE Communications Letters*, vol. 11, no. 2, pp. 167–169, Feb 2007.
13. C. Berrou, M. Jezequel, C. Douillard, and S. Kerouedan, "The advantages of non-binary turbo codes," in *IEEE Information Theory Workshop*, 2001, pp. 61–63.
14. A. Bartolazzi, G. Cardarilli, A. Del-Re, D. Giancristofaro, and M. Re, "Implementation of DVB-RCS turbo decoder for satellite on-board processing," in *IEEE International Conference on Circuits and Systems for Communications*, 2002, pp. 142–145.
15. C. Zhan, T. Arslan, A. T. Erdogan, and S. MacDougall, "An efficient decoder scheme for double binary circular turbo codes," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2006, pp. 229–232.
16. S. Papaharalabos, P. Sweeney, and B. Evans, "Constant log-MAP decoding algorithm for duo-binary turbo codes," *IET Electronics Letters*, vol. 42, no. 12, pp. 709–710, Jun 2006.
17. J. H. Kim and I. C. Park, "Double-binary circular turbo decoding based on border metric encoding," *IEEE Transactions on Circuits and Systems II*, vol. 55, no. 1, pp. 79–83, Jan 2008.
18. O. Muller, A. Baghdadi, and M. Jezequel, "ASIP-baser multiprocessor SOC design for simple and double binary turbo decoding," in *Design, Automation and Test in Europe Conference and Exhibition*, 2006, pp. 1330–1335.
19. T. Vogt and N. Wehn, "A reconfigurable application specific instruction set processor for Viterbi and Log-MAP decoding," in *IEEE Workshop on Signal Processing Systems Design and Implementation*, 2006, pp. 142–147.
20. M. C. Shin and I. C. Park, "SIMD processor-based turbo decoder supporting multiple third-generation wireless standards," *IEEE Transactions on VLSI*, vol. 15, no. 7, pp. 801–810, Jul 2007.
21. P. Robertson, P. Hoeher, and E. Villebrun, "Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding," *European Transactions on Telecommunications*, vol. 8, no. 2, pp. 119–125, Mar-Apr 1997.
22. S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-input soft-output modules for the construction and distributed iterative decoding of code networks," *European Transactions on Telecommunications*, vol. 9, no. 2, pp. 155–172, Mar/Apr 1998.
23. G. Montorsi and S. Benedetto, "Design of fixed-point iterative decoders for concatenated codes with interleavers," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 5, pp. 871–882, May 2001.
24. J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder," *IEE Electronics Letters*, vol. 36, no. 23, pp. 1937–1939, Nov 2000.

25. J. Zhang and M. P. C. Fossorier, "Shuffled iterative decoding," *IEEE Transactions on Communications*, vol. 53, no. 2, pp. 209–213, Feb 2005.
26. O. Muller, A. Baghdadi, and M. Jezequel, "Exploring parallel processing levels for convolutional turbo decoding," in *IEEE International Conference on Information and Communication Technologies: from Theory to Applications*, 2006, pp. 2353–2358.
27. J. Sun and O. Y. Takeshita, "Extended tail-biting schemes for turbo codes," *IEEE Communications Letters*, vol. 9, no. 3, pp. 252–254, Mar 2005.
28. S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Algorithm for continuous decoding of turbo codes," *IET Electronics Letters*, vol. 32, no. 4, pp. 314–315, Feb 1996.
29. A. Abbasfar and K. Yao, "An efficient and practical architecture for high speed turbo decoders," in *IEEE Vehicular Technology Conference*, 2003, pp. 337–341.
30. 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, Jun 2007.
31. A. Giulietti, L. V. der Perre, and M. Strum, "Parallel turbo coding interleavers: avoiding collisions in accesses to storage elements," *IET Electronics Letters*, vol. 38, no. 5, pp. 232–234, Feb 2002.
32. J. Kwak and K. Lee, "Design of dividable interleaver for parallel decoding in turbo codes," *IET Electronics Letters*, vol. 38, no. 22, pp. 1362–1364, Oct 2002.
33. S. Dolinar and D. Divsalar, "Weight distributions for turbo codes using random and nonrandom permutations," *TDA Progress Report*, vol. 42-122, pp. 56–65, Aug 1995.
34. 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, Feb 1998.
35. H. Michel and N. Wehn, "Turbo decoder quantization for UMTS," *IEEE Communications Letters*, vol. 5, no. 2, pp. 55–57, Feb 2001.
36. A. P. Hekstra, "An alternative to metric rescaling in Viterbi decoders," *IEEE Transactions on Communications*, vol. 37, no. 11, pp. 1220–1222, Nov 1989.