New Schemes in Clustered VLIW Processors Applied to Turbo Decoding*

Pablo Ituero Marisa López-Vallejo
ETSI Telecomunicación (UPM)
Ciudad Universitaria s/n. 28040 Madrid, Spain
{pituero,marisa}@die.upm.es

Abstract

State-of-the-art communication standards make extensive use of Turbo codes. The complex and power consuming designs that currently implement the turbo decoder expose the need for innovative solutions. In recent years the area of application specific processors has attracted the attention of the research community and important advances have been made possible. This work introduces an ASIP architecture for SISO Turbo decoding based on a dual-clustered VLIW processor. The machine deals with instructions of up to 21 operands in an innovative way, the fetching and asserting of data is serialized and the addressing is automatized and transparent for the programmer. An optimized architecture is achieved, flexible enough to comply with leading edge standards and adaptable to demanding hardware constraints.

1. Introduction

Turbo codes, introduced in 1993 [2], exhibit an astonishing performance and have been adopted in several industrial standards. The main drawback of these codes is the complex decoder structure which entails a power and area consuming VLSI implementation. Among all algorithms that can compute the Turbo decoding, the MAP algorithm [1] provides the best performance at low signal to noise levels. It is implemented by means of a Soft-Input Soft-Output (SISO) decoder, which causes the complexity of the Turbo decoder.

Current submicron technologies have allowed extraordinary integration densities in digital circuits. However, as processes scale down, uncertainty increases and the implementation of ASICs becomes extremely expensive because of their high cost and low design productivity. Only high production volumes can allow manufacturing. Application Specific Instruction-set Processors (ASIPs) solve part of these problems [6]: a higher volume of units is demanded because there are more applications that fit on it. Moreover, this kind of processors allow the use of a given architecture to implement different generations of the same application. Additionally, for the application developer that uses an ASIP instead of an ASIC the time to market is reduced, it is cheaper and there is lower risk. Furthermore, the power overhead related to programmability (standard processors) can be mitigated by ASIPs architectures, specially if they are very dedicated.

In this paper we present a dual-clustered VLIW processor that implements a SISO decoder. The architecture introduces original customizations that allow the optimized execution of multi-operand instructions. The fetching and asserting of data is serialized and the addressing is automatized and transparent for the programmer. A customized datapath allows very high efficiency while providing enough flexibility —Log-MAP and Max-Log-MAP algorithms, direct procedure and sliding windows mechanism. Finally, our approach is capable to accomplish with leading edge industrial standards —UMTS, CDMA2000, W-CDMA and IEEE 802.16.

Important work has been carried out in the hardware implementation of MAP-based Turbo decoders. Regarding configurability, most approaches are based on the replication of functional units to decode in parallel [10, 4]. To the best of our knowledge, no previous MAP-based SISO-decoder has reached the degree of versatility that is described in this paper. In the last few years, ASIPs have undergone a great development in the area of signal processing, finding their place in the huge electronics design scope [6]. All this has been possible thanks to the establishment of standard methodologies and CAD tools that greatly simplify the complexity of the design [5, 7].

The structure of this paper is the following. Section 2 describes the basis of the MAP algorithm. Sections 3 to 5 put forward the ASIP architecture, from the datapath to the controller. In section 6 we present the most important results of the design. Finally section 7 draws our conclusions.

2. The MAP algorithm

This section puts forward a brief overview of the characteristics of the Turbo decoding process that are necessary

*This work was funded by the CICYT project TIC2003-07036 of the Spanish Ministry of Science and Technology.

Operation	Inputs	Outputs
Gamma	4	3
Alpha Beta	10	8
LLR L^e_{out}	19	2

Table 1. Operands in the MAP equations.

to understand the architecture described later. For further information, reader is referred to [1].

The MAP algorithm can be described by the next set of equations

$$\overline{\gamma}_k(s',s) = \tfrac{1}{2}u_k\left(L^e_{in}(u_k) + L_c * y^s_k\right) + \tfrac{1}{2}L_c * y^p_k x^p_k \quad (1)$$

$$\overline{\alpha}_k(s) = \underset{s' \in S_{k-1}}{max^*}\{\overline{\alpha}_{k-1}(s') + \overline{\gamma}_{k-1}(s',s)\} \quad (2)$$

$$\overline{\beta}_{k-1}(s') = \underset{s \in S_k}{max^*}\{\overline{\beta}_k(s) + \overline{\gamma}_{k-1}(s',s)\} \quad (3)$$

$$LLR(u_k) = \underset{S^+}{max^*}\{\overline{\alpha}_k(s') + \overline{\gamma}_k(s',s) + \overline{\beta}_{k+1}(s)\} - $$
$$- \underset{S^-}{max^*}\{\overline{\alpha}_k(s') + \overline{\gamma}_k(s',s) + \overline{\beta}_{k+1}(s)\} \quad (4)$$

$$L^e_{out}(u_k) = LLR(u_k) - L_C y^s_k - L^e_{in}(u_k) \quad (5)$$

Given a sequence of received symbols (being u_k the k^{th} symbol), the algorithm consists of a forward recursion (estimation of the state sequence of the encoder in forward direction, represented by state metric $\alpha_k(s')$), followed by a backward recursion (estimation of the state sequence of the encoder in backward direction, represented by state metric $\beta_{k+1}(s)$), and a soft output calculation, LLR and L^e_{out}, based on the use of state metrics and external information (L^e_{in}, L_c, y^s_k and y^p_k). Finally, $\gamma_k(s',s)$ is the branch probability between states.

For our purposes equation 1 can be rewritten as:

$$\overline{\gamma}_k(s',s) = u_k\gamma^a + x^p_k\gamma^b \quad (6)$$

There are several ways to execute the algorithm which entails a trade-off between area and latency. A straightforward procedure computes the alphas in the forward recursion and the betas, the LLR and the L^e_{out} in the backward recursion; this procedure achieves a low total execution time, however it requires a whole block of memory to store the alphas and presents a huge latency. To overcome this problem some mechanisms have been proposed such as the *Sliding Windows Mechanism* [12] that divides the computation into small blocks which reduces significantly the storage needs and the latency.

To summarize, table 1 shows the number of operands in each equation. Note that the elevated number of inputs and outputs will set hurdles to the design of the processor.

3. General Architecture

Our SISO decoder is a microprogrammed VLIW ASIP which consists of a set of heterogeneous FUs (Functional Units). The machine has only one thread of control, although several computations are performed in parallel. It is the duty of the programmer to set up and maintain the instructions pipeline [5]. The global architecture is depicted in figure 1.

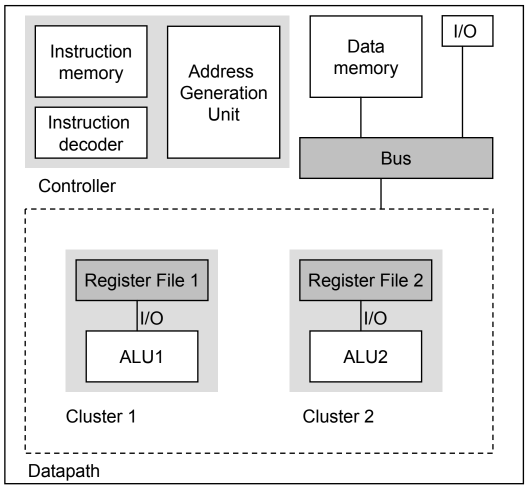

Figure 1. General architecture.

The whole processor is meticulously optimized for the execution of the MAP algorithm operations. Based on [5] three specialization dimensions have been considered to design the VLIW machine:

- The mapping of the decoding algorithm equations into independent FUs, in such a way that several operations can be carried out in parallel. This implies the tailoring of the instruction set.
- The customization of the register files to the needs of the algorithm and the interconnections among the FUs.
- The organization of the memory system.

In a general perspective, as shown in figure 1, the datapath is composed of two clusters, i.e. two FUs that possess their own independent register files. All the design decisions referent to the datapath are explained in the next section. The master controller consists of the instruction memory, the instruction decoder and the address generation unit; section 5 details the functioning of the controller.

4. Datapath

All the computations of the system are performed in the Datapath of the processor, therefore the main design effort was carried out on it. This module is responsible for the frequency and latency of the whole decoder.

The main tasks in the design were deciding the number of FUs to be used and mapping the equations into these FUs.

The only limitation that biased our design were the requirements in terms of data throughput of the third generation standards, therefore the variety of solutions was very ample. The departure point was the set of equations that conform the MAP algorithm; the high number of inputs and outputs in each operation was visibly incompatible with the goal of reducing connectivity and area needs, this exposed the necessity of an unconventional solution. A traditional architecture would fetch all the data in each operation at the same cycle, process it during a number of cycles and then assert all the results again at the same cycle; this strategy would imply an elevated number of connections, inputs and outputs in each FU, and furthermore it will leave few possibilities to future upgrades. To unravel this problem we came up with the idea of a set of pipelined FUs that fetch and assert data in a sequential manner from and to their register files. This complicates the control of the operations, however reduces connectivity and area requirements and provides the architecture with an improved flexibility. Furthermore the performance is increased since part of the data is produced and can be used by another FU before an operation is completely finished.

Internal pipelining of each operation was carefully pondered to eventually get a well-balanced structure that achieves a high throughput. 100% hardware resources utilization for every operation represents the ideal goal in an ASIP datapath, in this work we have almost reached that goal (see section 6).

4.1. Clustering

A set of relationships and dependencies can be established in the MAP algorithm operations. These dependencies are the key to the selection of the number of clusters, they also determine the interconnection needs between the clusters. We have considered three sorts of dependencies:

- Operator reutilization.
- Time execution dependencies.
- Area and power considerations.

As far as operator reutilization is concerned, from the analysis of the algorithm equations we infer that the multiplication operator is present just in the *gamma* computations, moreover the rest of the computations include the Add-Compare-Select (ACS) operator. This is the first hint that the *gamma* operation should be mapped to an independent FU, whereas the rest could share some hardware structures.

Focusing now on the time execution, as stated in section 2 the algorithm is to be executed by two procedures. A straightforward procedure first computes all the *gammas* and *alphas* and then calculates the *betas*, the *LLR* and the L^e_{out} using the previously calculated values. The *Sliding Windows Mechanism* divides the computation into small blocks, however the progress inside each block is exactly the same as in the whole straightforward procedure. Three dependencies are deduced:

- Since the results of the $\gamma_k(s', s)$ computation are used by the rest, it would be very convenient —in terms of throughput— to perform it in parallel with the rest.
- In both procedures the *beta* values are calculated after the *alpha* values, this implies a serial computation.
- LLR and L^e_{out} computations need a complete set of alphas or betas calculated; thus these former operations are necessarily serial with the latter.

Serial computation suggests resources reusability, whereas parallel computation implies new components to be instantiated. Hence, having into account the previous dependencies, the idea of two FUs —one mapping the gamma computation, the other mapping the rest— begins to take definite shape.

In order to minimize the area, resource sharing must be exploited as much as possible by different computations. This is another good defend for the mapping of the *alpha, beta, LLR* and L^e_{out} into a single FU.

Finally, it has been proven that one of the best strategies to lessen the power consumption is to reduce the number of memory accesses [9]. With a FU performing the *gamma* computation in parallel with rest, we consume these values *on the fly* and avoid storing and afterward fetching them from the memory. This also serves to justify the mapping of the *gamma* computation into a FU of its own.

4.2. Functional Units Development

Let us now move toward the mapping of the equations into FUs. Apart form the traditional design considerations, the sequencing of the data fetch and assertion was also analyzed. While the implementation of the *gamma* computation into a FU should not imply a big effort, the fusion of the rest into a single FU entails several design trade-offs that not always appear to be straightforward. The latter FU will be described initially since it fixes the data throughput of the whole system, including the other FU. In particular we will detail the implementation of each equation and then all the structures will be mixed to form the FU.

The structure that implements the *alpha* and *beta* computations, which perform the same operations, is shown in figure 2. In a first cycle half the alphas and one gamma are fetched and in a second cycle the rest of the data is input, moreover in the second cycle the first set of 4 alphas is asserted and the remaining 4 alphas are output in a third cycle.

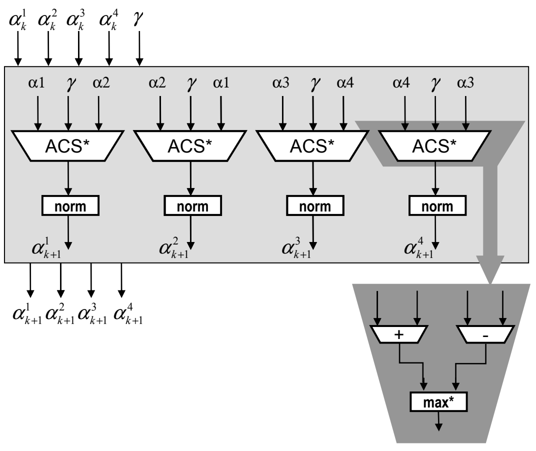

Figure 2. Alpha-Beta Implementation.

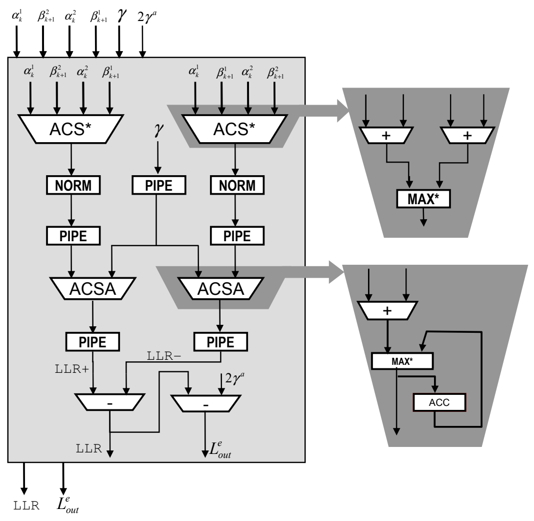

Figure 3. LLR-L_{out}^{e} Implementation.

Figure 3 shows our pipelined approach for the *LLR* and L_{out}^{e} computations. The first stage encloses two ACS* structures exactly equal to those of the *alpha-beta* computations, the second stage is implemented with ACSA modules that introduce an accumulator to allow several iterations, and finally the third stage includes two adders.

The merging into a single FU is shown in the leftmost block of figure 4. All pipeline registers separate add-max^{*} structures, which yields a design that is both well-balanced and independent of the max^{*} module implementation.

The implementation of the *gamma* computation is illustrated in the rightmost block of figure 4. A pipelined multiplier is introduced to reduce the combinational delay between registers.

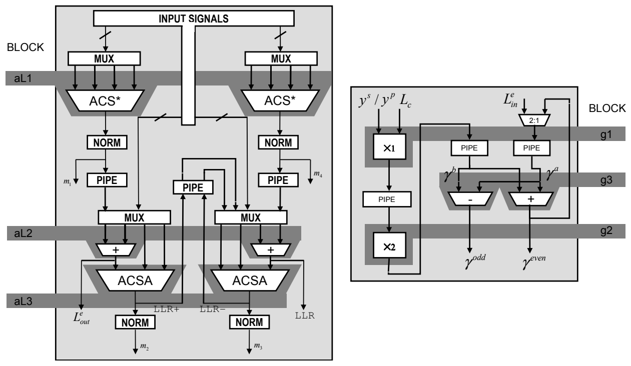

Figure 4. Pipeline block division.

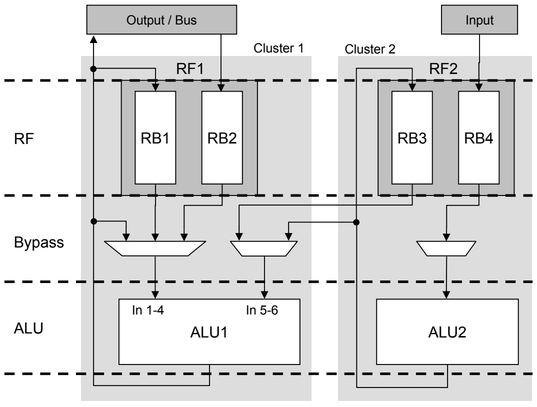

Figure 5. Pipeline structure

4.3. Global Datapath Microarchitecture

Figure 5 shows the three stages of our datapath microarchitecture. Dashed lines represent pipeline registers, nevertheless the operations executed inside the ALUs entail several pipeline stages. Arrows crossing the pipeline lines downward are registered whereas the arrows crossing the pipeline lines upward are not registered. Table 2 summarizes all the connectivity requirements of the system and specifies the register bank from where each data is fetched.

Each cluster contains one register file and each register file is divided into two register banks. Only the necessary

Computation	Data	Fetched from
Gamma	External Input	RB4
Alpha	Gamma	Bypass and RB3
	Previous Alpha	Bypass and RB1
Beta	Gamma	Bypass and RB3
	Previous Beta	RB1
LLR	Gamma	RB3
	Beta	Bypass and RB1
	Alpha	RB2

Table 2. Connectivity requirements.

Operation	Latency	Throughput
Gamma	4	2
Alpha Beta	2	2
LLR L^e_{out}	6	4

Table 3. Latencies and throughputs.

connections are hardwared in the register file, nevertheless it is highly flexible as far as Turbo codes are concerned; Moreover since we are dealing with so many parallel signals, it would be very complicated for the programmer to specify all the registers in each operation, therefore this task is left for a special unit —the AGU (Address Generation Unit), described later— so that the programmer does not need to reference any register at all. Cluster 1 —alpha, beta and LLR— has one bank, RB1, for the data produced in the alpha and beta computations and another, RB2, to momentarily store the alphas fetched from the bus that were stored in the data memory. Cluster 2 —gamma— holds also two banks, one for the results of the operations, RB3, and another to store and buffer the input data of the system. The register banks inside a cluster work independently, thus RB2 loads data from memory and RB1 fetches data from ALU1 simultaneously.

The bypass network selects the source of each ALU1 input. The control of this network is carefully designed so that it is transparent to the programmer, this is not a simple task since in some cases each of the six inputs comes from a different source.

5. Control

The controller of the system relies on a pipeline architecture able to deal with multicycle operations. For the sake of clarity, the datapath is divided into six independent blocks as shown in figure 4, each of them constitutes an independent logic element inside the FUs that the operations will employ throughout the pipeline execution.

As explained in section 4, the peculiarity of our design is the sequencing of the data fetch and assertion inside the same operation. Figure 6 details the blocks that each operation uses in each pipeline stage, furthermore it shows when data are collected and when they are output for each computation. Table 3 details the latency and throughput or initialization interval of each operation.

In contrast with a general purpose processor, our controller is microprogrammed to direct each stage of the pipeline. This provides it with a higher flexibility and optimizes the resources utilization ratio, although complicates the labor of the programmer who is in charge of maintaining the pipeline [5]. With this scheme the controller conducts the order of the stages in each operation, being able to stall it or to execute it in a different order.

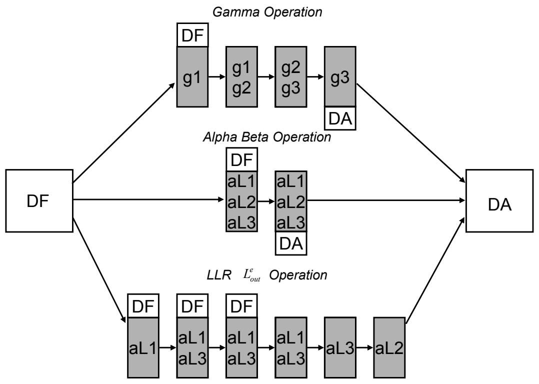

Figure 6. Pipelining of operations

At the core of the controller, the AGU manages all the addresses involved in the system, i.e. the addresses of the data memory, the program memory, the input interface, the output interface and the register file. The programmer only has to provide certain parameters like the block and window size and this unit calculates the appropriate addresses transparently. During the forward recursion the data memory and the input interface need auto-incrementing addresses whereas in the backward recursion the data memory and the input and output interfaces need auto-decrementing addresses. Referring the registers of multi-operand instructions makes the duty of the programmer very cumbersome, in the case of the LLR operation as much as 19 data are input, therefore it was also necessary to make this register referring transparent.

6. Results

Our design is clearly oriented toward a standard-cell implementation, however, as a first approach it was prototyped in a Xilinx VirtexII 4000 FPGA for test and comparison purposes. Table 4 compares our design with significant previous works; since different generations and technologies are used, we have taken the number of clock cycles per decoded symbol as the comparison metric because it depends just on the architecture itself. Moving from top to bottom we go from maximum versatility to maximum particularity. The table is opened by a GPP that implements the max-log-MAP [11]. The first work to introduce the metric cycles per symbol was [8], carried out by the University of Kaiserslautern; it analyzes a VLIW DSP, a low-power, low-cost DSP and a configurable RISC. As a representation of ASIC designs we have included the unified Turbo/Viterbi decoder described in [3]. Finally [13] is a representation of commercial IPs. As shown, our work stands in the boundary between processors and application specific designs, providing the programmability of the former along with the tailoring and good performance of the latter.

The implementation took 517 slices and 3 BR (Block

Work	Processor/Tecnology	Architecture	Algorithm	Th. (kbps)	Freq (MHz)	Iter.	Cycles/Symbol
[11]	Intel Pentium III	GPP	max-LM	366	933	1	1275
[8]	Motorola 56603	Low-power DSP	max-LM	48,6	80	5	165
[8]	ST-M. ST120	VLIW DSP, 2 ALU	LM	200	200	5	100
[8]	ARC-Tensilica	Configurable RISCs	LM	303	100	5	33
Ours	Xilinx xc2v4000-4	VLIW ASIP	selectable	10000	80	0,5	8
[3]	0,18m CMOS 6metal	ASIC	LM	2146	93	10	2
[13]	Xilinx xc4vsx25-12	IP	max-LM	20200	293	5	1

Table 4. Comparison with previous works.

Computation	Resources Util.
Alpha and Beta Computation	89.0%
Alternating Beta-LLR Computation	92.5%
Direct Procedure	91.6%
Sliding Windows Mechanism	91.1%

Table 5. Resource utilization results.

Pipeline Stage	Standard Approach	Serialized Approach
ALU1 input	19	5
ALU1 output	8	6
ALU2 input	4	2
ALU2 output	3	2

Table 6. Register reduction.

RAM) for the max-log-MAP and 555 slices and 3 BR for the log-MAP. The frequency attained for the max-log-MAP was 80.57 MHz and 60.39 MHz for the log-MAP. These values are importantly biased by the architecture of the FPGA, nevertheless our critical path coincides with that of fast ASIC implementations —mux-ACS-normalize— therefore similar working frequencies are expected. Furthermore, even under this hardware platform, the design achieves a data throughput able to comply with demanding standards.

The datapath has been carefully tailored, achieving a throughput of 8 cycles per symbol and a ratio of utilization of approximately 90% for every computation as shown in table 6. The serialization of the data fetching and asserting entails the reduction of the connectivity needs and the number of registers that are used internally by the Datapath, this is shown in table 6. Finally the automation of the addressing greatly simplifies the programming of the processor.

7. Conclusions

An ASIP for SISO decoding has been presented with two original customizations for clustered VLIW architectures. Specifically the fetching and asserting of data in multi-operand instructions has proven to reduce connectivity and area requirements and shows a potential performance improvement in multi-clustered architectures. The automation of the addressing highly simplifies the programming of the machine. The architecture achieves a throughput of 8 cycles/symbol and is compliant with most recent standards.

References

[1] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. on Information Theory*, pages 284–287, March 1974.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: Turbo-codes. *IEEE Transactions on Communications*, 44(2), May 1993.

[3] M. Bickerstaff, D. Garrett, T. Prokop, C. Thomas, B. Widdup, G. Zhou, C. Nicol, and R.-H. Yan. A unified turbo / viterbi channel decoder for 3GPP mobile wireless in 0.18/spl mu/m CMOS. *IEEE Journal of Solid-State Circuits*, 37(11), 2002.

[4] G.Prescher, T. Gemmeke, and T. Noll. A Parametrizable Low-Power High-Throughput Turbo-Decoder. In *IEEE ICASSP '05*, pages 25–28, March 2005.

[5] M. F. Jacome and G. de Veciana. Design challenges for new application-specific processors. *Design&Test of computers*, 17(2):40–50, Apr-Jun 2000.

[6] K. Keutzer, S. Malik, and A. R. Newton. From ASIC to ASIP: The Next Design Discontinuity. In *IEEE International Conference on computer Design: VLSI in computers and Processors*, pages 84–90, 2002.

[7] V. S. Lapinskii, M. F. Jacome, and G. A. de Venecia. Application-Specific Clustered VLIW Datapaths: Early Exploration on a Parameterized Design Space. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 21(8), August 2002.

[8] H. Michel, A. Worm, M. Munch, and N. Wehn. Hardware/software trade-offs for advanced 3g channel coding. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 396–401, 2002.

[9] C. Schurgers, F. Catthoor, and M. Engels. Energy efficient data transfer and storage organization for a map turbo decoder module. In *ISLPED'99*, pages 76–81. IEEE, 1999.

[10] M. Thul and N. Wehn. FPGA Implementation Of Parallel Turbo-Decoders. In *IEEE 17th Symposium on Integrated Circuits and Systems Design*, pages 198–203, Sept. 2004.

[11] M. Valenti and J. Sun. The UMTS Turbo Code and a Efficient Decoder Implementation Suitable for Software-Defined Radios. *International Journal of Wireless Information Networks*, 8(4), 2001.

[12] A. J. Viterbi. An Intuitive Justification and a Simplified Implementation of the Map Decoder for Convolutional Codes. *IEEE Jnl. Selected Areas in Comms*, (2):260–264, 1998.

[13] I. Xilinx. *3GPP Turbo Decoder v2.0, 2006.*