Further Specialization of Clustered VLIW Processors: A MAP Decoder for Software Defined Radio

Pablo Ituero and Marisa López-Vallejo

Turbo codes are extensively used in current communications standards and have a promising outlook for future generations. The advantages of software defined radio, especially dynamic reconfiguration, make it very attractive in this multi-standard scenario. However, the complex and power consuming implementation of the maximum *a posteriori* (MAP) algorithm, employed by turbo decoders, sets hurdles to this goal. This work introduces an ASIP architecture for the MAP algorithm, based on a dual-clustered VLIW processor. It displays the good performance of application specific designs along with the versatility of processors, which makes it compliant with leading edge standards. The machine deals with multi-operand instructions in an innovative way, the fetching and assertion of data is serialized and the addressing is automatized and transparent for the programmer. The performance-area trade-off of the proposed architecture achieves a throughput of 8 cycles per symbol with very low power dissipation.

Keywords: Application specific instruction-set processor (ASIP), maximum *a posteriori* (MAP), soft-input soft-output (SISO) decoder, software defined radio (SDR), turbo code, very long instruction word (VLIW) architectures.

I. Introduction

Turbo codes, introduced in 1993 [1], achieve outstanding bit error rate performance approaching the Shannon limit, the theoretical maximum information transfer rate over a noisy channel. The main drawback of turbo codes is the complex decoder structure which entails a power and area consuming VLSI implementation. Among all algorithms that can compute turbo decoding, the maximum *a posteriori* (MAP) algorithm [2] provides the best performance at low signal to noise levels. This algorithm is carried out in a soft-input soft-output (SISO) decoder, whose implementation causes the complexity of the turbo decoder. Due to their excellent performance, most wireless communication systems, characterized by a low signal to noise ratio, make use of turbo codes. For instance, the most important third generation wireless mobile standards such as UMTS, W-CDMA (3GPP), and cdma2000 (3GPP2) have adopted turbo codes in their specifications. Turbo codes are employed by other present industrial standards, such as Consultative Committee for Space Applications (CCSDS) telemetry channel coding, DVB-RCS, IEEE 802.11n, and IEEE 802.l6ab. Furthermore, turbo equalization and turbo space-time decoders have opened new niches and surely will lead the way for future techniques and paradigms.

In this context, software defined radio (SDR) seems an appealing solution to deal with the proliferation of wireless standards with different frequency and modulation techniques. With this approach SDR-enabled user devices can be dynamically programmed in software to reconfigure their characteristics for better performance, richer feature sets, advanced new services that provide choices to the end-user, and new revenue streams for the service provider.

However, current processors are not cheap enough or

Manuscript received Mar. 15, 2007; revised Nov. 11, 2007.

This work was funded by the CICYT project OPTIMA (TIC2006-00739) of the Spanish Ministry of Science and Technology.

Pablo Ituero (phone: + 34 915495700 ext. 4235, email: pituero@die.upm.es) and Marisa López Vallejo (email: marisa@die.upm.es) are with the Integrated Systems Laboratory, Electronic Engineering Department, ETSI Telecomunicación, Universidad Politécnica de Madrid, Madrid, Spain.

energy-efficient enough to support realistic implementations of portable software radios [3]. Thus, more sophisticated architectures make sense, because radios, like other signal-processing applications, exhibit task-level parallelism that can easily be mapped onto a concurrent architecture, as in the case of clustered VLIW processors. Moreover, when the architecture matches the application, we talk about application-specific instruction-set processors (ASIPs), which solve most problems related to ASIC implementations [4]. A higher volume of units is demanded because there are more applications or different generations of the same application that fit onto it. Additionally, for the application developer who uses an ASIP instead of an ASIC, the time to market is reduced because it is cheaper and there is lower risk. Furthermore, in comparison to general purpose processors, the power overhead related to programmability can be mitigated by ASIP architectures, especially if they are very dedicated.

In this paper, we present a dual-clustered VLIW processor that implements a SISO decoder. This work attempts to bridge the gap between application specific designs and processors, offering the customization degree of application specific designs together with the strength of versatility of processors. Moreover, using the proposed architecture, the embedded applications have low area and power requirements, which allows the use of multiple cores in a single chip. To carry out these goals, the proposed architecture includes a customized datapath, which allows very high efficiency while keeping low level details transparent to the programmer. Also, an advanced mechanism to feed data to multi-operand instructions has been devised to improve the processor performance along with automated addressing, which greatly simplifies the programming. The resulting architecture displays robust performance together with the flexibility to tackle different codes, code lengths, procedures, and algorithms, which means that the processor is able to meet the most demanding industrial standards. All these characteristics make our design especially attractive for SDR and embedded systems applications.

The remainder of this paper is structured as follows. Section II describes the basis of the MAP algorithm. Sections III to V put forward the ASIP architecture, from the datapath to the controller. In section VI, two application examples are described. In sections VII and VIII we present the most important results of the design and compare them with relevant previous works. Finally, section IX gives our conclusions.

II. The MAP Algorithm

In a communication system that employs turbo codes, on the transmission side, a turbo encoder produces and sends three kinds of outputs: a systematic word (the input information); a parity word from a convolutional encoder fed with the input information; and one or more parity words from one or more convolutional encoders, whose inputs are randomized versions of the input information. This randomization, performed by an interleaver, is known by the decoder and makes the code very robust against channel variability. The turbo decoder performs a soft decoding with each pair of received systematic-parity data by means of an SISO decoder. In each SISO decoding, the system gains soft information, which is used for the subsequent SISO decoding in an iterative process. The decoding ends when the threshold reliability is reached.

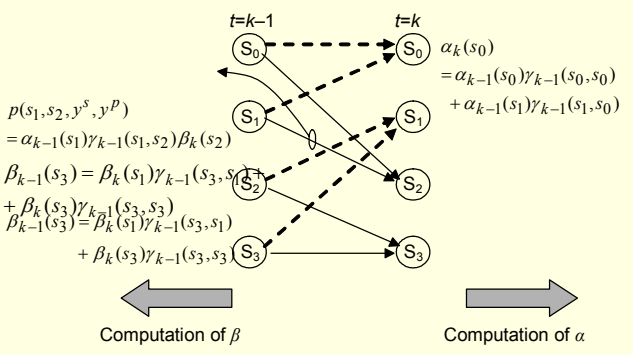

Fig. 1. Parameters involved in the decoding process.

The MAP algorithm [2] computes the soft decoding with a very high reliability. Figure 1 shows a section of the trellis produced by the encoder. In this figure the main parameters of the algorithm are shown. The transmitted symbol is represented by u_k. The probability that a transition is produced is given by $p(s', s, y^s, y^p) = \alpha_k(s')\,\gamma_k(s',s)\,\beta_{k+1}(s)$, where s' and s are the origin and destination states, respectively; y^s and y^p are the noisy received systematic and parity data respectively; $\alpha_k(s')$ is the forward path metric; $\beta_{k+1}(s)$ is the backward path metric; and $\gamma_k(s',s)$ is the branch metric. An SISO decoder implements this algorithm. It receives the noisy data along with the *a priori* soft information of the previous decoding L^e_{in} and a channel constant, Lc, related to the energy per channel bit and the channel noise level. The SISO decoder yields the log-likelihood ratio (LLR) and the extrinsic information for the next decoder, L^e_{out}—the soft information that this decoding has gained.

The MAP algorithm can be simplified by working in the logarithmic domain. In this domain, multiplications become additions, and additions can be computed by the Jacobian logarithm ($\ln(e^a + e^b) = \max\{a,b\} + f(|a-b|)$). If the last term—known as the correction term—is considered, the algorithm is referred to as the LM (Log-MAP) algorithm; in contrast, if the correction term is neglected, the algorithm is referred to as the max-LM (max-Log-MAP) algorithm [5]. In the logarithmic domain, the metrics are calculated by the following

expressions[1]:

$$\bar{\gamma}_k(s',s) = \frac{1}{2}u_k(L^e_{in}(u_k) + L_c y^s_k) + \frac{1}{2}L_c y^p_k x^p_k, \quad (1)$$

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

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

$$\begin{aligned} LLR(u_k) = &\max_{S^+}{}^*\{\bar{\alpha}_k(s) + \bar{\gamma}_k(s',s) + \bar{\beta}_{k+1}(s)\} \\ &- \max_{S^-}{}^*\{\bar{\alpha}_k(s') + \bar{\gamma}_k(s',s) + \bar{\beta}_{k+1}(s)\}, \end{aligned} \quad (4)$$

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

The α and β computations are recursive and depend on the previous and subsequent values, respectively. That is, $\alpha_k(s)$ needs two α values from t=k–1 to be computed. This is the reason why α and β are known as the forward and backward metrics, respectively; they both are also known as the state metrics.

Equations (1) to (5) are the basis of the architecture described in this paper. There are several ways to execute the algorithm which entail a trade-off between area and latency. A straightforward procedure computes the α's in the forward recursion and the β's, 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 α's and leads to high latency. To overcome this problem, various mechanisms have been proposed, such as the sliding windows mechanism [6], which divides the computation into small blocks, significantly reducing the storage needs and the latency. Both approaches will be tackled in our software defined radio processor.

Table 1 shows the computational needs of each equation, specifically the number of inputs, outputs, additions, multiplications, and add-compare-select (ACS) structures. Note that the elevated number of inputs and outputs will set hurdles for the design of the processor.

Table 1. Computational needs of MAP algorithm.

Operation	Inputs	Outputs	Add.	Mult.	ACS
γ	4	3	2	1	0
α, β	10	8	0	0	16
LLR-L^e_{out}	19	2	10	0	32

1) To simplify the equations, ln x is denoted by $\bar{x}$, and max* stands for the Jacobian expression max{a,b} + f(|a-b|).

1. Gamma Operation Details

As stated in [7], "good" RSC encoders for turbo codes generate a trellis which can be grouped into 2^{m-1} butterfly pairs, each of which is determined by a unique substrate—*m* denotes the memory in bits of the encoder. A butterfly pair is illustrated in Fig. 2, where x^s_k and x^p_k represent the systematic and the parity outputs of the encoder, respectively. In a trellis section, half of the pairs have codewords (-1,-1) and (1,1); the other half have codewords (-1,1) and (1,-1). If the γ computation of (1) is considered, it is clear that for time instant *k*, all the parameters remain constant except x^s_k and x^p_k; therefore, we can write

$$\begin{aligned} &\bar{\gamma}_k(s',s) = x^s_k\gamma^a + x^p_k\gamma^b, \\ &\gamma^a = 1/2u_k(L^e_{in}(u_k) + L_c y^s_k), \\ &\gamma^b = 1/2L_c y^p_k. \end{aligned} \quad (6)$$

Therefore, we have four possible values for γ depending on the codeword. However, we must take into account a property of the previously mentioned butterfly pairs. In a butterfly pair, there are only two values of γ and they are opposites:

$$\gamma^{even} = (+1)\gamma^a + (+1)\gamma^b = -\{(-1)\gamma^a + (-1)\gamma^b\}, \quad (7)$$

$$\gamma^{odd} = (+1)\gamma^a + (-1)\gamma^b = -\{(-1)\gamma^a + (+1)\gamma^b\}. \quad (8)$$

This leads to the conclusion that for a butterfly pair, only a single γ computation must be performed because the other one is computed as the opposite to the former. We could go even further and state that, for a whole trellis section, there is no need to calculate more than two γ's, namely γ^{even} and γ^{odd}.

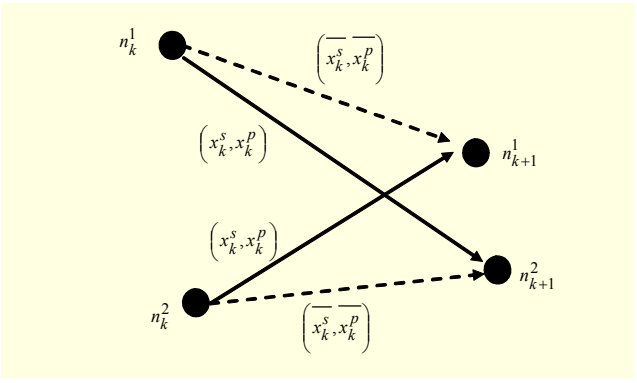

Fig. 2. Butterfly pair.

III. General Architecture

Our SISO decoder is a microprogrammed VLIW ASIP based on a Harvard microprocessor architecture with separate data and program memories, and strongly wired addressing. More specifically, it consists of a set of heterogeneous functional units, which execute operations on various data

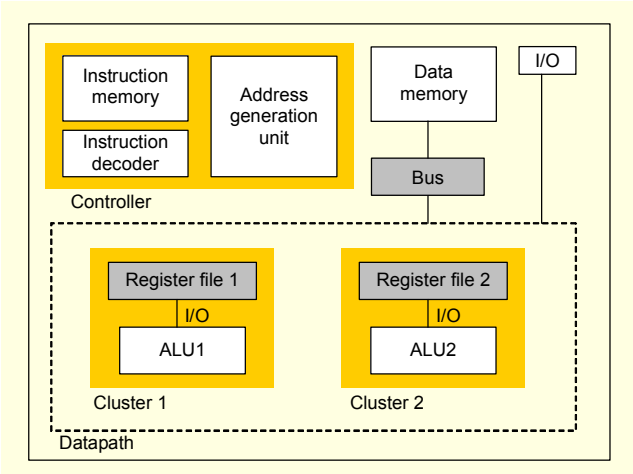

Fig. 3. General architecture of the SISO decoder.

elements. A microcoded centralized controller provides the functional units with the necessary control information. Each cycle, an instruction is executed. This instruction details all datapath and memory actions to be carried out in that cycle. 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 instruction pipeline [8]. The global architecture is shown in Fig. 3.

The whole processor is meticulously optimized for the execution of the MAP algorithm operations. The main characteristic of these operations is the high number of input and output operands. We have targeted low area and low power consumption, always taking into account that the system has to be compliant with third generation standards regarding data throughput. Based on [8], three specialization dimensions have been considered in designing the VLIW machine:

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

From a general perspective, as shown in Fig. 3, the datapath is composed of two clusters, that is, two functional units, which possess their own independent register files. All the design decisions referent to the datapath are explained in the next section. The master controller includes the instructions memory, the instruction decoder, and the address generator unit. Section V details the functioning of the controller.

The data memory, accessed by a bus, stores the α values processed in the forward recursion, and it does not affect the timing of the system. Its minimum size is dependent on the algorithm and procedure because different block and window lengths lead to different data memory sizes. The I/O circuitry handles the external interface buffering input and output data. Specifically, all the external data required for computation—L^e_{in}, L_c, y_k^s and y_k^p—is received at the I/O block and forwarded to corresponding places in the register files. In the same way, the output data, L^e_{out}, is forwarded to the I/O block, which handles the assertion of this data.

IV. Datapath

All the computations of the system are carried out in the datapath of the processor; therefore, the main design effort has been focused on it. This module is responsible for the frequency and latency of the whole decoder. Moreover, it takes up most of the area and power consumption of the system.

The main tasks in the design process were deciding the number of functional units to be used and mapping the equations into these functional units. The major challenge was to explore the great diversity of solutions and to evaluate and select the most appropriate. The only restrictions that biased our design were the requirements in terms of data throughput of the third generation standards that our software defined radio processor had to fulfill. Therefore, the variety of solutions was very wide. The departure point was the set of equations that make up the MAP algorithm. The high number of inputs and outputs in each operation was clearly incompatible with the goal of reducing connectivity and area needs. This demonstrated the necessity of an unconventional solution. A traditional architecture would gather all the data in each operation in the same cycle, process it during a number of cycles, and then assert all the results again in the same cycle. This strategy would imply an elevated number of connections, inputs, and outputs in each functional unit. Furthermore, it would leave little possibility for future upgrades. To solve this problem we came up with the idea of a set of pipelined functional units, which gather and assert data in a sequential manner from and to their register files. This complicates the control of the operations; however, it reduces connectivity and area requirements and provides the architecture with improved flexibility. Performance is also improved because some of the data is produced and can be used by another functional unit before an operation is completely finished.

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

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 clusters. We have considered three sorts of dependencies:

- operator reutilization,
- time execution dependencies,
- area and power considerations.

Regarding operator reutilization, from the analysis of the algorithm equations, we infer that the multiplication operator is present only in the γ computations of (1). Moreover, the rest of the computations include the ACS* operator, which performs the max^*. This is the first hint that the γ operation should be mapped to an independent functional unit, whereas the rest could share some hardware structures.

Focusing now on time execution, as stated in section II, the algorithm is executed by two procedures. A straightforward procedure first computes all the γ's and α's, and then calculates the β's, the *LLR* and the L^e_{out}, using the previously calculated values. The sliding windows procedure 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 β values are calculated after the α values. This implies a serial computation.
- *LLR* and L^e_{out} computations need a complete set of α's or β's calculated; therefore, these operations are necessarily serial.

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

To minimize the area, resource sharing must be exploited as much as possible by different computations. This is another good reason for mapping the α, β, *LLR*, and L^e_{out} into a single functional unit.

Finally, it has been demonstrated that one of the best strategies to reduce power consumption is to reduce the number of memory accesses [9]. With a functional unit performing the γ computation in parallel with the rest, we consume these values on the fly and avoid storing and afterward retrieving them from the memory. This also serves to justify the mapping of the γ computation into a functional unit of its own. Moreover, it reduces memory bottlenecking.

2. Development of Functional Units

In addition to traditional design considerations, the sequencing of data fetching and assertion was also analyzed. While the implementation of the γ computation into a functional unit (FU) is not complicated, the fusion of the rest into a single FU entails several design trade-offs that do not always appear to be straightforward. The FU that implements the α, β, *LLR*, and L^e_{out} operations will be described initially because it fixes the data throughput of the whole system, including the other FUs. In particular, we will detail the implementation of each equation and then all the structures will be combined to form the FU.

As shown in (2) and (3), the α and β computations perform the same basic operations; therefore, they will be dealt with as one. If we center on the α computation and assume an 8-state turbo code, a trellis section requires 8 inputs for the previous α's, 2 inputs for the γ's, and 8 outputs for the calculated α's. Also, there are 8 equal maximization operations in this trellis section. Taking into account the butterfly property, two butterfly pairs can be computed making use of just five inputs as long as they share $\gamma_k(s',s)$. This means that the data of a single operation is fetched and asserted along several cycles, in contrast with general purpose processor (GPP) architectures in which all the data is fetched and then asserted simultaneously. In the first cycle, half of the α's and one γ are fetched, and in the second cycle, the rest of the data is input. In the second cycle the first set of 4 α's is also asserted. The second group of 4 α's will be output in the third cycle. This computation attains a throughput of 2 cycles/symbol. Figure 4 displays the general α and β computation structure. Note that the normalization module norm will be described in section IV.4.

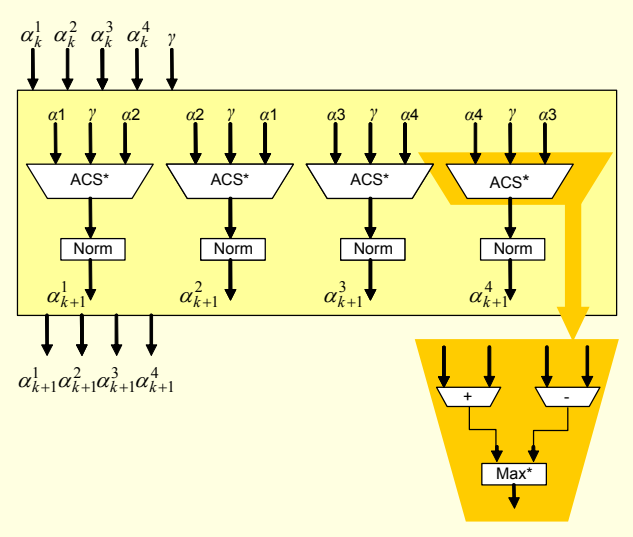

Fig. 4. Implementation of the α and β computations.

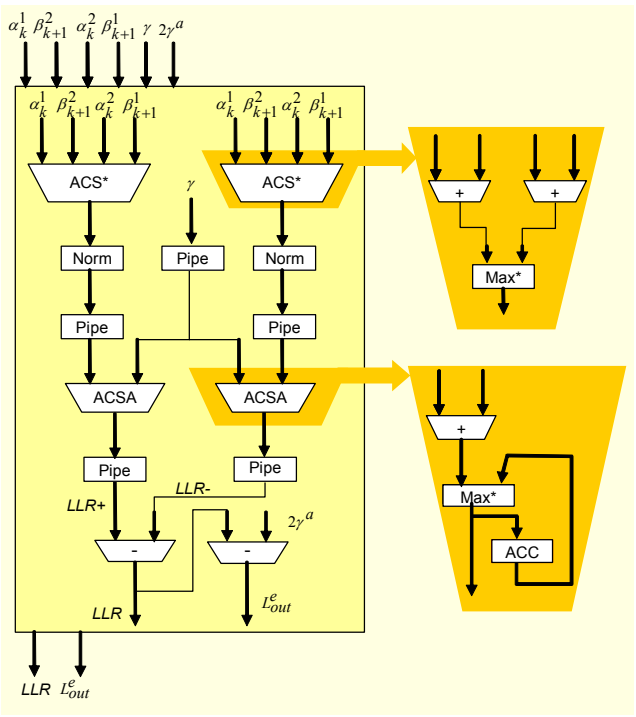

Fig. 5. Pipelined implementation of the *LLR* and L^e_{out} computations.

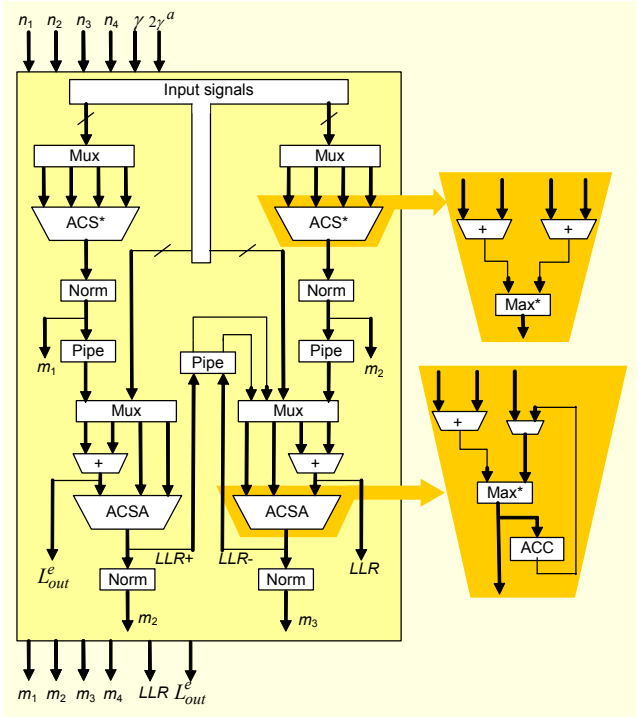

Fig. 6. α-β-*LLR*-L^e_{out} functional unit.

The *LLR* and L^e_{out} computations are described by (4) and (5). For an 8-states Turbo code, a trellis section with 4 butterflies requires 8 inputs for the α's, 8 inputs for the β's, three inputs from the γ unit and two outputs for *LLR* and L^e_{out}. In order to minimize the connectivity, all the operations corresponding to just one butterfly are computed in parallel in one cycle, a throughput of 4 cycles/symbol is attained. The data is fetched sequentially so that in each cycle it is only necessary to fetch the data corresponding to a butterfly pair; since a butterfly uses two α's, two β's and one γ, only five inputs are required. Considering that equally signed transitions in a butterfly pair share the same value of γ, we can extract this term out of the maximization operand, so that it is added afterward. Figure 5 shows our pipelined approach in which the first stage encloses two ACS* structures exactly equal to those of the α and β computations, the second stage is implemented with ACSA modules, which introduce an accumulator to allow several iterations, and finally, the third stage includes two adders.

The total numbers of adders and *max** modules in both α and β computations and *LLR* and L^e_{out} computations are the same. This was intended because they are to be mapped into a unique structure. The merging of FUs into a single FU is shown in Fig. 6. All pipeline registers separate *add-max** structures, yielding a design that is both well-balanced and independent of the *max** module implementation.

Each received symbol has to go through the α, β and *LLR* and L^e_{out} computations at 2, 2, and 4 cycles/symbol, respectively; therefore, the global execution of the processor entails a throughput of 8 cycles/symbol. This value will be compared with current implementations of the turbo decoder in section VII .

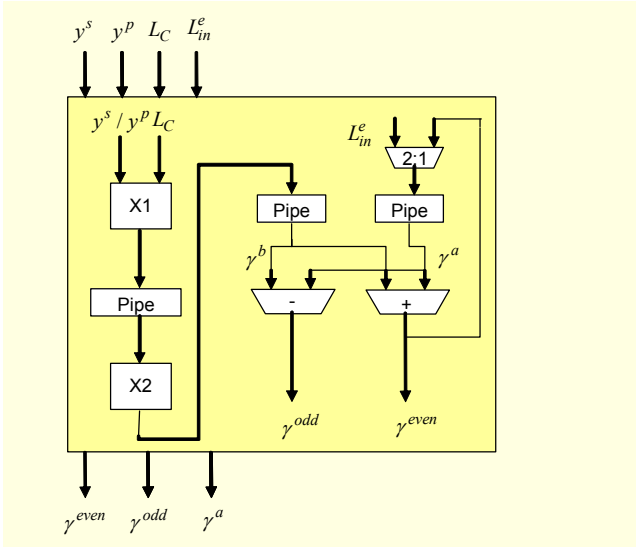

Fig. 7. Implementation of the γ computation.

The γ operation is described by (1). This operation possesses 4 inputs: L^e_{in}, L_c, y^s_k, and y^p_k, and has to assert 3 outputs: γ^{even}, γ^{odd}, and γ^a. Since this computation is to be performed in parallel with the rest, it has to fulfill the timing of the most restrictive operation in terms of throughput, that is,

2 cycles/symbol for the α and β computations. Figure 7 illustrates the implementation of this unit. A pipelined multiplier is introduced to reduce the combinational delay between registers.

3. Global Datapath Microarchitecture

Figure 8 shows the three stages of our datapath microarchitecture, each register file (RF) is divided into two register banks (RBs). Dashed lines represent pipeline registers. The operations executed inside the ALUs entail several pipeline stages. Arrows crossing the pipelines downward are registered, whereas the arrows crossing the pipelines upward are not registered.

To understand the microarchitecture it is necessary to review some of the connectivity requirements of the MAP algorithm computations. The α and γ metrics are calculated during the forward recursion. The α metrics are used by the subsequent α computation and by future *LLR* computations; therefore, some of these results are bypassed. The rest are stored temporarily in the RF, and they all are sent to the data memory via a data bus. In contrast, γ metrics are consumed on the fly by the next α operation; therefore, some values are again bypassed, and the rest are registered and fed later. In the backward recursion, the γ operation is performed in parallel with the β and *LLR* operations. The β operation is the same as the α in the forward recursion except that they do not need to be stored because they are used immediately after they are computed. The *LLR* operation requires the data from all the other operations, specifically, the γ values stored in the RF; the β values that were just calculated, some of which are bypassed, while the rest are fetched from the register file; and the α values that were computed during the forward recursion, which are now also fetched from the RF.

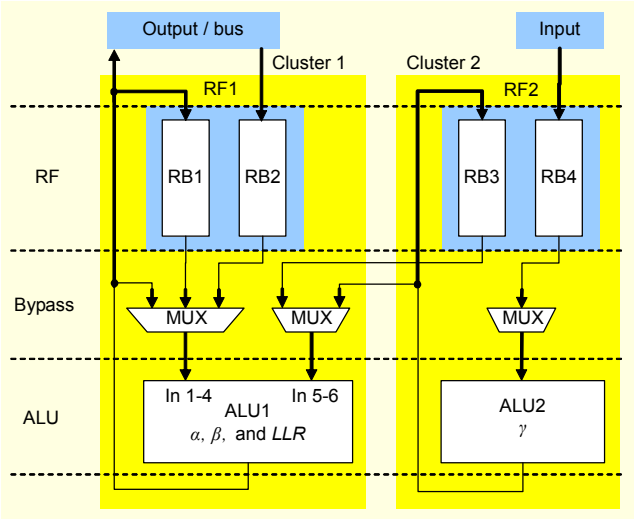

Fig. 8. Pipeline structure.

Table 2. Connectivity requirements.

Computation	Data	Fetched from
γ	Input	RB4
α	γ	Bypass and RB3
	Previous α	Bypass and RB1
β	γ	Bypass and RB3
	Previous β	RB1
LLR	γ	RB3
	β	Bypass and RB1
	α	RB2

Table 2 summarizes all these requirements and specifies the RB from which the data is fetched. State metric initialization is performed by loading the initialization values in RB1.

Each cluster contains one RF, and each RF is divided into two RBs. The RFs are tailored to the needs of the algorithm, and only the necessary connections are hardwared. 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 address generation unit (AGU) so that the programmer does not need to reference any register at all. Cluster 1–which computes α, β, and *LLR*–has two banks: RB1, for the data produced in the α and β computations, and RB2, to momentarily store the α's that were stored in the data memory and were fetched from the bus. Cluster 2–which computes γ–also has two RBs: 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 the 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.

4. Normalization Implementation Details

The state metrics get bigger and bigger as the decoding algorithm proceeds. To avoid this huge increase, some normalization scheme must be adopted. In [10], a new normalization scheme was proposed. At each time instant, all the α's or β's are compared with 2^{q-2}, q being the number of bits of the state metrics. If any one of them is greater than 2^{q-2}, all the α's or β's are subtracted from 2^{q-2}; otherwise, they remain untouched. This method was chosen for this work because it significantly simplifies normalization in comparison with previous approaches [5] since just a one-bit flag must be

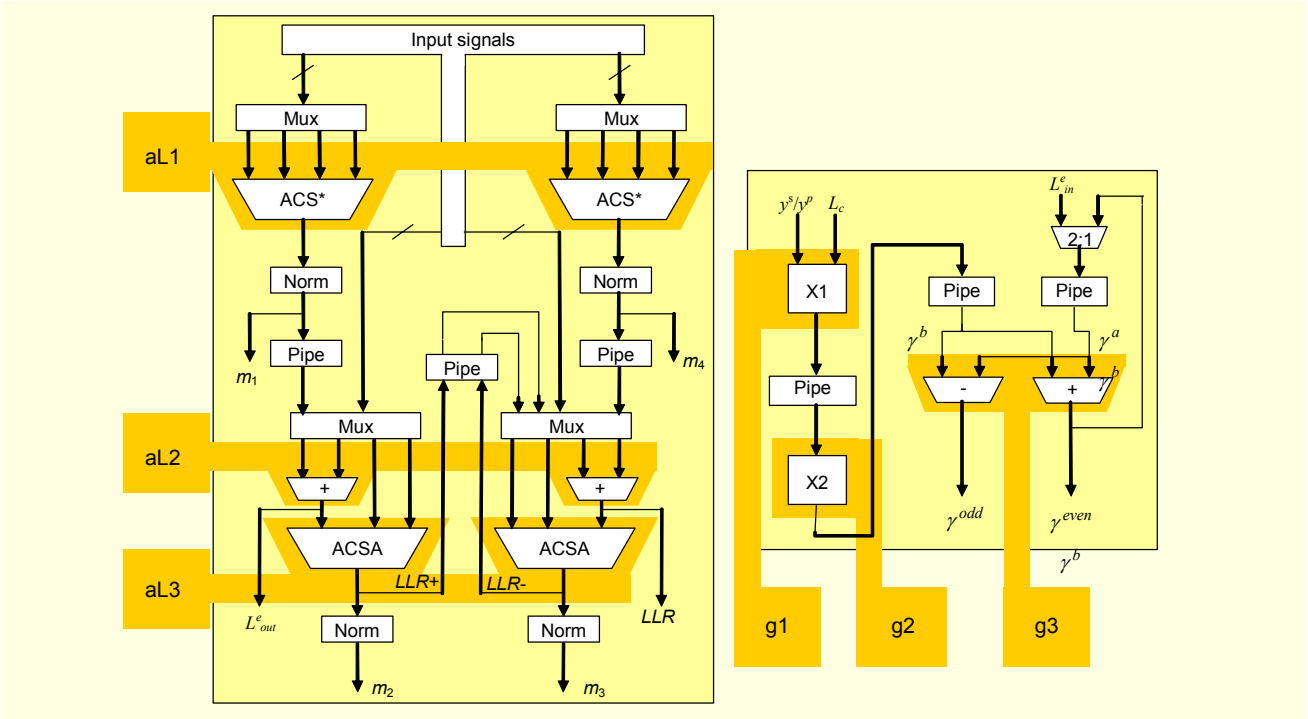

Fig. 9. Block division of the pipeline computation.

stored along with the state metrics of a time instant to express the normalization state. It is also very efficient since simple combinational logic can be used to implement it.

However, since serialization is assumed, we have to slightly modify the previous normalization scheme. In our work, $>2^{q-2}$ is calculated throughout several cycles, so it is impossible to decide whether -2^{q-2} should be computed until we have finished with all the state metrics. Our modification consists of flagging whenever any state metric is greater than 2^{q-2}. This flag is registered and used in the next block of state metrics to decide if normalization at the input must be applied. Hence, unnormalized values are stored, and they are not normalized until input for the calculations of the next set of state metrics.

For *LLR* and L^e_{out} computations, both α and β parameters might need normalizing because they both are stored without normalizing. Therefore, two flags must be input, one for α and one for β; therefore, two normalization modules are needed. Each module subtracts 2^{q-2} when its corresponding flag is activated.

V. Control

The control of the system relies on a pipelined architecture able to deal with multicycle operations. The design of pipelined general purpose processors is widely explained in the literature, as for example, in [11]; therefore, we will focus on the special characteristics of our architecture. For the sake of clarity, the datapath is divided into six independent blocks as shown in Fig. 9. Each of them constitutes an independent logic element inside the clusters that the operations will employ throughout the pipeline execution.

As explained in section IV, the novel contribution of our design is the sequencing of the data fetching and assertion inside the same operation. Figure 10 details the blocks that each operation uses in each pipeline stage. It also shows when the inputs are collected and when the outputs are asserted for each computation. In the figure, DF and DA denote data fetch and data assert. As expected, the γ operation is performed in parallel with the rest using blocks inside the second cluster. Table 3 details the latency and throughput or initialization interval of each operation.

In contrast with a GPP that deals with instructions as a block of pipeline stages, our controller is microprogrammed to direct each stage of the pipeline independently. This provides it with a higher flexibility and optimizes the resources utilization ratio, although this complicates the work of the programmer in charge of maintaining the pipeline [8]. Depending on the particular turbo code (different standards and code sizes) and procedure (straightforward procedure, sliding windows) each operation executes differently regarding data and interconnections. The pipeline, shown in Fig. 10, may also vary. The order of the stages may change, or new stages may be added; therefore, it is more convenient to control each part of the operation separately. With this scheme, the controller conducts the order of the stages in each operation and is able to

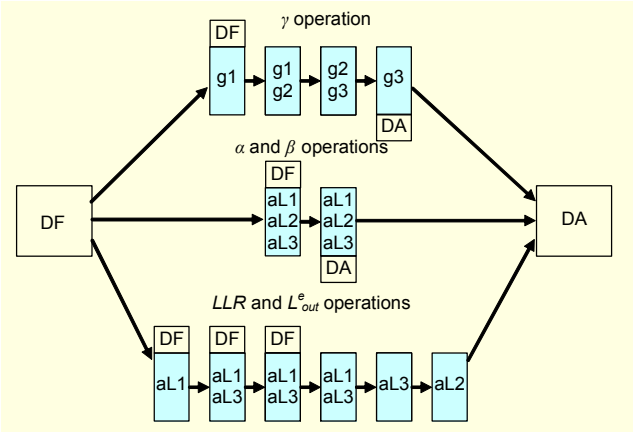

Fig. 10. Pipeline structure.

Table 3. Latencies and throughputs for operations.

Operation	Latency	Throughput
γ	4	2
α, β	2	2
LLR-L^e_{out}	6	4

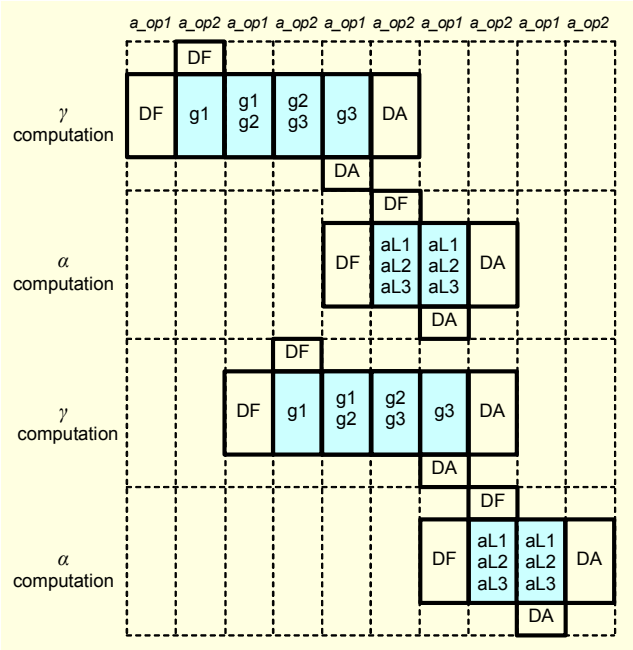

Fig. 11. Forward recursion.

stall it or to execute it in a different sequence.

Figures 11 and 12 illustrate the execution of the forward and backward recursions, respectively. In the upper part, the assembly code that controls the first cluster is shown. Clusters 1 and 2 are used simultaneously since the γ computation is executed in parallel with the rest. This parallelism is vertical. In the backward recursion, the β and *LLR* operations are totally coordinated so as not to leave the first cluster unused in any cycle. The *LLR* operation must be stalled during two cycles in order to fulfill the data dependency requirements; however, during these two cycles, cluster 1 is used by the β operation.

1. Address Generation Unit (AGU)

At the core of the controller, the AGU manages all the addresses involved in the system, namely, 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, such as the block and window size, and this unit calculates the appropriate addresses transparently. In a GPP, the programmer does not need to specify the program memory address of each instruction. Only in the case of branches and jumps, the programmer provides an address or the difference between the current and the potential address. The controller of the GPP is in charge of auto-incrementing and calculating the increments or decrements in the address. In our ASIP, since the data is always accessed in a well-known pattern, all these properties are extended to the rest of the memories of the system, including the RFs. 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. In the case of the sliding windows mechanism, the data memory of the system is reduced to the size of the window. Thus the AGU needs different addresses for this memory and the external interfaces. Specific instructions provide the controller with the size of the data block and the window length. Referring the registers of multi-operand instructions makes the duty of the programmer very cumbersome. In the case of the *LLR* operation, there are as many as 17 inputs; therefore, it is also necessary for this register referral to be transparent. The AGU automatically performs the register addressing for each operation by means of a lookup table.

2. Assembler

An assembly language for the controller was developed. To ease its use, an assembler translates it into machine code, which fills up the instruction memory. The format of the instruction word, shown in Table 4, consists of four fields. The first and second fields contain instructions for clusters one and two, respectively; the third field handles the control flow of the

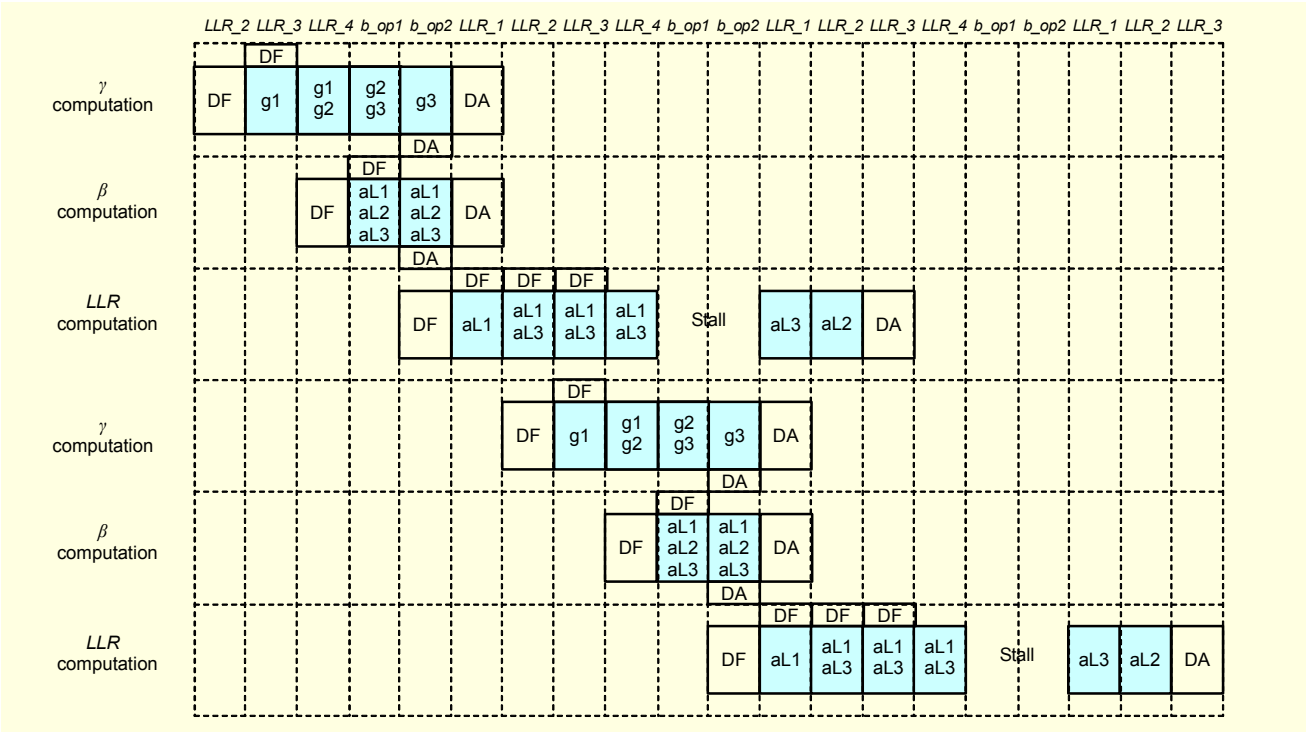

Fig. 12. Backward recursion.

system and some addressing which is not hardwared in the register file; the fourth field, stores numeric values that are passed as parameters to the computation. The codes for the first field are already known: *a_op1* and *a_op2* for α and β computations and *b_op1*, *b_op2*, *LLR_1*, *LLR_2*, *LLR_3*, and *LLR_4* for the alternating β and *LLR* computations. Just two codes are needed to control the second cluster, *g_op1* and *g_op2*. In the uppermost part of Figs. 11 and 12, the usage of these different codes is illustrated.

Table 4. Instruction word format.

Instruction FU1	Instruction FU2	Control	Data

VI. Examples

To exhibit the adaptability of the architecture, two different scenarios are discussed: a performance-driven design and an area-driven design. The processor can be programmed to execute different procedures, including direction procedure and the sliding windows mechanism and different standards, including UMTS, CMDA 2000, and any other 8-state turbo code, depending on the external requirements of our software defined radio processor. Furthermore, the architecture can vary depending on the hardware constraints. The size of the data memory can be reduced to the length of a window, and the *max** module can map either the max-LM operation or the LM operation.

1. Scenario A: Performance-Driven Design

In a performance-driven scenario, we propose a parallel sliding windows schema with n processors working concurrently. Each core independently decodes a set of data so that the throughput is incremented n times. Depending on the power and performance needs of the system, the number of activated cores changes. For example, under low power availability circumstances, only one or few processors work and the rest remain in a drowsy state. If a faulty processor is detected, it is deactivated and the rest assume its responsibilities.

Figure 13 displays the assembly code of each of the processors in this scenario. The execution starts with a loop that computes α's previous to the beginning of the window until it gets reliable values; then, the second loop computes and stores the reliable α's until the end of the window; afterward, the third loop computes β'*s* subsequent to the window until reliable values are achieved; finally, the fourth loop takes the previously stored α's and performs the β and *LLR* computations.

2. Scenario B: Area-Driven Design

In a different scenario, in which the area is the most critical parameter, a single core approach is proposed. The processor

```
Begin
a_op1  g_op1  fwd jmp_alpha          X007F  %Compute previous alphas in
a_op2  g_op2  IN_RB1                 X0000  %forward recursion until
a_op1  g_op1  cl_cmp1                X0000  %the beginning of the window
a_op2  g_op2  LOOP1                  X0000  %LOOP1: Alpha Computation
a_op1  g_op1                         X0000  %LOOP1.a
a_op2  g_op2  BR1_ne                 X0000  %LOOP1.b; branch if not equal
a_op1  g_op1  st_alpha               X0000  %Compute & store reliable alphas
a_op2  g_op2  ld_cmp1                X00FF  %throughout the window size
a_op2  g_op2  LOOP1                  X0000  %LOOP1: Alpha Computation
a_op1  g_op1                         X0000  %LOOP1.a
a_op2  g_op2  BR1_ne                 X0000  %LOOP1.b; branch if not equal
a_op1  g_op1  bwd jmp_beta           X007F  %Compute future betas in
a_op2  g_op2                         X0000  %backward recursion until
a_op1  g_op1  ld_cmp1                X0100  %the end of the window
a_op2  g_op2  LOOP1                  X0000  %LOOP1:Beta Computation
a_op1  g_op1                         X0000  %LOOP1.a
a_op2  g_op2  BR1_ne                 X0000  %LOOP1.b; branch if not equal
a_op1  g_op1                         X0000  %Compute alternating beta-LLR
b_op1  g_op2                         X0000  %in backward recursion, fetching
b_op2  g_op2  IN_RB12                X0000  %previously stored alphas until
LLR_1  g_op1  cl_cmp1                X0000  %the beginning of the window
LLR_2  g_op2  LOOP1                  X0000  %LOOP1:Alternating Beta-LLR
LLR_3  g_op1                         X0000  %LOOP1.a
LLR_4  g_op2                         X0000  %LOOP1.b
b_op1  g_op1                         X0000  %LOOP1.c
b_op2  g_op2                         X0000  %LOOP1.d
LLR_1  g_op1                         X0000  %LOOP1.e
LLR_2  g_op2  BR1_ne  wt_en          X0000  %LOOP1.f; branch if not equal
LLR_3  g_op1                         X0000  %
LLR_4  g_op2                         X0000  %
LLR_1  g_op1                         X0000  %
LLR_2  g_op2  wt_en                  X0000  %Assert last data
End
```

Fig. 13. Assembly code for scenario A.

executes a serial sliding windows mechanism with a minimum window length to reduce the data memory. The computation covers the data in several windows, dealing with each one in the same manner as the n processors were dealt with in the previous scenario. The assembly code that executes this computation is shown in Fig. 14. An external loop controls the number of windows to be processed; and three internal loops perform the α, β, and the alternating β and *LLR* computations.

VII. Previous Work and Comparison

ASIPs have undergone a great development in the area of signal processing, finding their place in electronics design [4]. The challenges and future trends, like retargetable compilers and synthesis tools, were discussed in [8], and the first steps in finding standard methodologies and CAD tools have also been proposed [12], [13], to help designers in several tasks, including hardware/software partition and early design exploration. However a complete automated methodology is still far off. In recent years, novel ASIP architectures have been proposed to implement a variety of signal processing problems[14], [15]. Concerning data forwarding in clustered

```
Begin
a_op1  g_op1  ld_cmp2               X000F  %Define number of windows
a_op2  g_op2  LOOP2 fwd IN_RB1       X0000  %LOOP2:window computation
a_op1  g_op1  ld_cmp1 st_alpha       X00FF  %Forward alpha
a_op2  g_op2  LOOP1                  X0000  %LOOP1:Alpha Computation
a_op1  g_op1                         X0000  %LOOP1.a
a_op2  g_op2  BR1_ne                 X0000  %LOOP1.b; branch if not equal
a_op1  g_op1  bwd jmp_beta           X007F  %Compute future betas in
a_op2  g_op2                         X0000  %backward recursion until
a_op1  g_op1  ld_cmp1                X0100  %the end of the window
a_op2  g_op2  LOOP1                  X0000  %LOOP1:Beta Computation
a_op1  g_op1                         X0000  %LOOP1.a
a_op2  g_op2  BR1_ne                 X0000  %LOOP1.b; branch if not equal
a_op1  g_op1                         X0000  %Compute alternating beta-LLR
b_op1  g_op2                         X0000  %in backward recursion, fetching
b_op2  g_op1  IN_RB12                X0000  %previously stored alphas until
LLR_1  g_op2  cl_cmp1                X0000  %the beginning of the window
LLR_2  g_op1  LOOP1                  X0000  %LOOP1:Alternating Beta-LLR
LLR_3  g_op2                         X0000  %LOOP1.a
LLR_4  g_op1                         X0000  %LOOP1.b
b_op1  g_op2                         X0000  %LOOP1.c
b_op2  g_op1                         X0000  %LOOP1.d
LLR_1  g_op2                         X0000  %LOOP1.e
LLR_2  g_op1  BR1_ne wt_en           X0000  %LOOP1.f; branch if not equal
LLR_3  g_op2                         X0000  %
LLR_4  g_op1                         X0000  %
LLR_1  g_op2                         X0000  %
LLR_2  g_op1  BR2_ne wt_en           X0000  %LOOP2 branch if not equal
End
```

Fig. 14. Assembly code for scenario B.

VLIW processors, [16] proposed a low power solution that bypasses all short-lived variables and achieves an improvement of 7.8% in power consumption. More recently, [17] demonstrated that it is the bypass network which limits the clock speed and not the register file.

In less than 15 years since turbo codes were introduced, almost every aspect of turbo decoder hardware design has been covered in a wide variety of studies, from ASIC to GPP implementations. Table 5 summarizes this development comprising the whole electronic spectrum with significant works from the literature along with commercial intellectual properties (IPs). Since different generations and technologies are used, we have taken the number of clock cycles per SISO decoded symbol as the comparison metric because it depends on the architecture itself. To be more precise we are comparing the number of cycles that each architecture takes to SISO decode one symbol or the number of cycles dedicated to one symbol.

In each half Turbo decoding, this can be obtained by cycles/symbol=freq./(2×iter.×th.) where freq. is the frequency of the system, iter. is the number of iterations and th. is the throughput of the system.

The table is opened by a GPP that implements both the LM and the max-LM. It is remarkable that, for such a processor, the effort to apply the correction term of the LM accounts for over 85% of the whole computation [18]. Then, a fixed-point DSP

Table 5. Comparison with previous works.

Work	Processor/tecnology	Architecture	Algorithm	Th. (kbps)	Freq. (MHz)	It.	Cycles/symbol
[18]	Intel Pentium III	GPP	LM	51	933	1	9,147
[18]	Intel Pentium III	GPP	max-LM	366	933	1	1,275
[19]	TI TMS320C6201	Fixed-point DSP	max-LM	282	200	1	355
[20]	Xirisc	RISC with pGA	Linear-LM	270	100	1	185
[21]	Motorola 56603	Low-power DSP	max-LM	48.6	80	5	165
[22]	ST-M. ST120	VLIW DSP, 2 ALU	LM	200	200	5	100
[22]	ST-M. ST140	VLIW DSP, 4 ALU	LM	600	300	5	50
[22]	ST-M. ST120	VLIW DSP, 2 ALU	max-LM	540	200	5	37
[21]	ARC-Tensilica	Configurable RISCs	LM	303	100	5	33
[22]	ADI TigerSharc	VLIW DSP, 2 ALU	LM	666	180	5	27
[22]	ST-M. ST140	VLIW DSP, 4 ALU	max-LM	1,875	300	5	16
Ours	Xilinx xc2v4000-4	VLIW ASIP	selectable	10,000	80	0,5	8
[23]	90 nm Std-Cell	ASIP	max-LM	7,400	335	3	7.5
[24]	Altera APEX 20K	IP	max-LM	2,000	50	5	3
[25]	65nm Std-Cell	ASIP	LM-Viterbi	20,000	400	5	2
[26]	0.25 µm CMOS 5 m	ASIC	selectable	5,480	135	6	2
[27]	0.18 µm CMOS 6 m	ASIC	LM	2,146	93	10	2
[28]	0.18 µm CMOS 6 m	ASIC	LM	27,600	285	5	1
[29]	Xilinx xc4vsx25-12	IP	max-LM	20,200	293	5	1

achieves a reduction in the number of cycles per symbol by a factor of four, employing some optimization techniques, such as flattening multidimensional arrays into one-dimension to enhance compilation [19]. In [20], a reconfigurable DLX-based RISC processor, developed by the University of Bologna, is used along with an FPGA, called PiCoGa (pGA), which implements special hardware kernels. Linear-LM displays a very close performance to the LM, so the results are good in comparison with the GPP. In [21] and [22], carried out by the University of Kaiserslautern, metric cycles per symbol were first introduced. These studies analyze a variety of VLIW DSPs, including a low-power, low-cost DSP and a configurable RISC considering both max-LM and LM. The number of cycles is reduced by one order of magnitude down to 16 under the best scenario with 4 ALUs and max-LM. A recent work by the same university, [25], uses the idea of the ASIP for turbo decoding, aiming for a system with a very high throughput using a pipeline of 11 stages and extended configurability. It supports convolutional codes, turbo codes, and duobinary turbo codes. This architecture achieves 2 cycles/symbol by replicating several times the units that process the butterfly pairs. In our opinion, this is an outstanding work suitable for systems that have strict constraints of throughput and require a high degree of reconfigurability. However, power and area issues, very important in the embedded systems field, are weakly tackled and left as secondary concerns. Another ASIP was recently proposed, [23], that extensively replicates processing nodes to achieve high data throughput. In this case, 7.5 cycles/symbol are achieved, and again, power and area issues are barely treated. An SIMD processor that implements the whole turbo decoder was proposed in [26]. In this case, the processor control interleaving hardware block interfaces with an external host and stopping criterion. It uses a configurable ASIC SISO decoder that is claimed to produce one decoded output every two cycles. Although it could seem similar to our present work, it has a different aim. In fact, our design could be embedded in the previous design as a co-processor that performs the SISO decoding. As examples of ASIC designs, we have included a unified turbo Viterbi decoder [27] and a very fast SISO decoder [28], which pipelines the add-compare-select kernels. Finally, two commercial IPs are presented in [24] and [29]. A very impressive frequency is attained by Xilinx in its Virtex-4. Figure 15 displays all of this data making use of a logarithm scale. From left to right, it shows maximum versatility to maximum particularity, from short to long design time, and from inexpensive to expensive. As shown, our work stands in the boundary between processors and application specific designs, providing the programmability

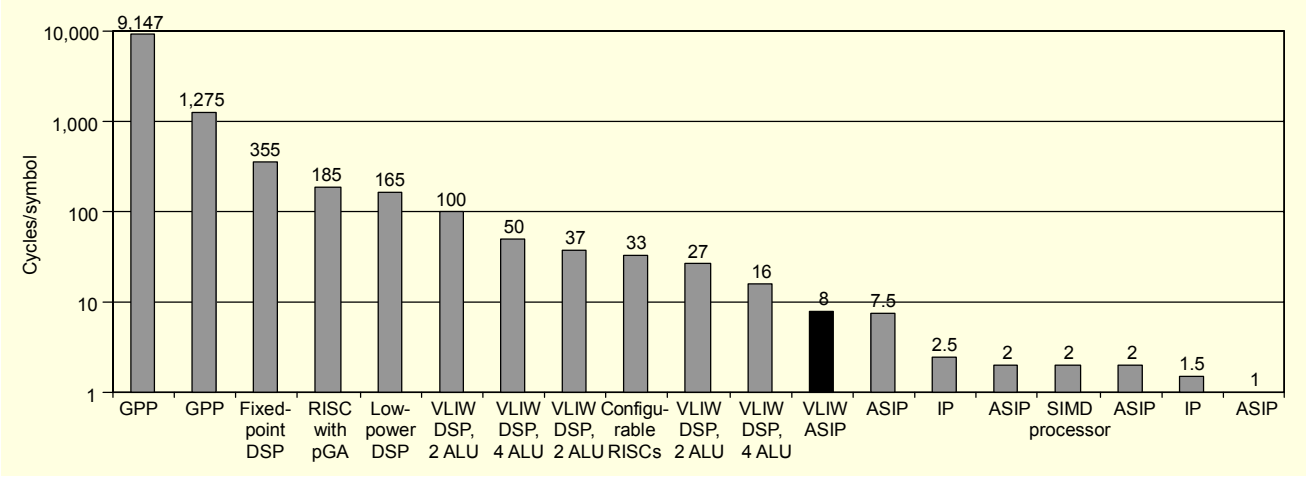

Fig. 15. Comparison of the number of cycles per instruction for different implementations of the turbo decoder.

of the former along with the tailoring and good performance of the latter. Furthermore, all this comes with a very low cost in area and power, as will be shown in section VIII.

Recently, several works have addressed many other aspects of the VLSI implementation of turbo decoders. An excellent analysis of low-power issues was presented in [30], considering previous and original techniques for both AWGN and fading channels, obtaining up to 80% energy savings. Memory optimizations which impact delay and energy are explored thoroughly in [31] and [32]. The sliding windows approach was analyzed in [33], [34], and more recently in [35]. In [33], new schemes are proposed for parallel sliding windows, targeting high-speed applications. In [34], a meticulous analysis is presented of the dataflow optimizations for several versions of the algorithm, achieving savings in area and power of up to a 53%. Other approaches have dealt with the issue of configurability, most of which are based on the replication of functional units to decode in parallel [36], [37]. Modifications to established algorithms have also been proposed in [38] and [39]. In [38], an algorithm which is less complex than previous algorithms and with a performance equivalent to the max-log-MAP was presented. The method proposed in [39] reduces the memory access rate by 90% by performing a reverse calculation of the backward metric, leading to a 30% power reduction.

VIII. Results

The design and manufacture if ASIC is becoming harder and more expensive, while improvements in programmability and flexibility incur performance and power overhead costs [4]. This work is an attempt to bridge that gap by offering a customized application specific design with the advantage of versatility. We have introduced three initiatives towards the customization of the architecture:

- The datapath has been carefully tailored, achieving a throughput of 8 cycles/symbol and a ratio of resource utilization of approximately 90% for every computation, as shown in Table 6.
- The serialization of data retrieval and assertion entails the reduction of the connectivity needs and the number of registers that are used internally by the datapath, as shown in Table 7. Since multi-operand instructions are executed in the same FU, in this particular design, there is no impact on the performance. However, in a multi-cluster architecture an important improvement could be achieved by passing data to other FUs before the full instructions are finished.
- All the addresses of the RFs, memories, and external interfaces are automatically assigned by the AGU, which greatly helps programmers avoid the reference of tens of operands in the same instruction.

These characteristics push our design further from a GPP, boosting performance and reducing area and power consumption, although the loss of programmability is evident since a conventional C program could not be run on it. In relation to ASIC designs, our goal was to take advantage of the processor characteristics, such as resource reutilization and, of course, programmability to widen the gap in terms of area,

Table 6. Resource utilization results.

Computation	Resources utilization
α and β computations	89.0%
Alternating β and *LLR* computations	92.5%
Direct procedure	91.6%
Sliding windows mechanism	91.1%

Table 7. Register reduction caused by fetching and assertion serialization.

Pipeline stage	Standard approach	Serialized approach
ALU1 input	19	5
ALU1 output	8	6
ALU2 input	4	2
ALU2 output	3	2

Table 8. Prototype area comparison.

Work	Platform	Design	Algorithm	Area
[36]	Xilinx Virtex-II	SISO	Max-LM	2134 slices/8 BR
[36]	Xilinx Virtex-II	SISO	LM	2500 slices/8 BR
[29]	Xilinx Virtex-4	Turbo	Max-LM	3481 slices/17 BR
[24]	Altera APEX 20K	Turbo	Max-LM	5500 LE/135 ESB
Ours	Xilinx Virtex-II	SISO	Max-LM	517 slices/3 BR
Ours	Xilinx Virtex-II	SISO	LM	555 slices/3 BR

Table 9. Power measures results.

	LM @ 60 MHz	Max-LM @ 80 MHz
Dynamic power (mW)	90.1	102.43
Quiescent power (mW)	336.6	336.6
Total power (mW)	426.7	439.03

power dissipation, and versatility.

An important aspect of deploying any new architecture is verification, which usually requires lengthy software simulation of a design model [40]. Our design is clearly oriented toward a standard-cell implementation; however, as a first approach, it was prototyped in a Xilinx Virtex-II 4000 FPGA for test and comparison purposes. FPGA devices make hardware emulation practical and cost effective for new processor designs. Table 8 shows the area results contrasted with other works implemented in FPGAs that provide area information. Even taking into account that two of the works implement the whole turbo decoder, the reduction in area is evidently over a factor of 4; however, the comparison is not completely fair since the designs and algorithms adopted in these works are different from ours. The frequency attained for the max-LM was 80.57 MHz and 60.39 MHz for the LM. It was significantly biased by the architecture of the FPGA; nevertheless, our critical path coincides with that of fast ASIC implementations, namely, mux-ACS-normalize; therefore, similar working frequencies are expected.

The power estimates were obtained with the Xilinx power measurement tool, XPower. The values are presented in Table 9. The constant quiescent power is caused not only by the area that the system takes up, a very small portion of the FPGA, but also by the rest of the array, which also needs to be powered. Dynamic power consumption is around two orders of magnitude above 0.25 μm standard-cell designs, which employ low-power policies [39]. This difference is due to the architecture of the FPGA, which supposes an elevated hardware overhead. To our knowledge, no other work prototyping in an FPGA has provided power measurements; therefore, no comparison is possible. From a more abstract point of view, comparing our proposed architecture with more general processors, the almost optimum resource utilization, the smaller area, and the lower working frequency entail a reduction in both dynamic and static power consumption. In relation to ASIC designs, the significant area reduction leads to a reduction in static power consumption as well.

IX. Conclusion

We presented a clustered VLIW ASIP architecture, which implements a MAP decoder, employing either the max-LM or the LM algorithm. The datapath of the system was carefully optimized by extensive use of butterfly pair properties. The high concurrence of the processor attains a throughput of 8 cycles per symbol. Two original customizations were introduced. The fetching and assertion of data in multi-operand instructions was demonstrated to reduce connectivity and area requirements, leading to a potential performance improvement in multi-clustered architectures. The automation of addressing greatly simplifies the programming of the machine. Compared with previous ASIC designs, a reduction of area by over a factor of 4 is achieved. The design exhibits great flexibility and is compliant with most recent industrial standards.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," *IEEE Trans. on Comm.*, vol. 44, no. 2, May 1993.

[2] 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*, Mar. 1974, pp. 284-287.

[3] W. Wolf, "Building the Software Radio," *IEEE Computer*, 2005, pp. 87-89.

[4] K. Keutzer, S. Malik, and A.R. Newton, "From ASIC to ASIP: The Next Design Discontinuity," *IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors*, 2002, pp. 84-90.

[5] P. Robertson and P. Hoeher, "Optimal and Sub-Optimal Maximum a Posteriori Algorithms Suitable for Turbo Decoding,"

European Trans. Telecommunication, no. 8, Mar./Apr. 1997, pp. 119-125.

[6] A.J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the Map Decoder for Convolutional Codes," *IEEE J. Selected Areas in Comms*, no. 2, 1998, pp. 260-264.

[7] Y. Wu, W.J. Ebel, and B.D. Woerner, "Forward Computation of Backward Path Metrics for MAP Decoder," *IEEE VTC*, 2000.

[8] M.F. Jacome and G. de Veciana, "Design Challenges for New Application-Specific Processors," *Design&Test of Computers*, vol. 17, no. 2, Apr./June 2000, pp. 40-50.

[9] C. Schurgers, F. Catthoor, and M. Engels, "Energy Efficient Data Transfer and Storage Organization for a Map Turbo Decoder Module," *ISLPED*, IEEE, 1999, pp. 76-81.

[10] Z. Wang, H. Suzuki, and K. Parhi, "VLSI Implementation Issues Of Turbo Decoder Design For Wireless Applications," *IEEE Workshop on Signal Processing Systems*, Oct. 1999, pp. 503-512.

[11] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2002.

[12] 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. Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 8, Aug. 2002.

[13] A. HoffMann, H. Meyr, et al., "A Novel Methodology for the Design of Application Specific Integrated Precessors (ASIP) Using a Machine Description Language," *IEEE Trans. Computer Aided Design*, vol. 20, no. 11, 2001.

[14] Z. Liu, K. Dickson, and J.V. McCanny, "Application-Specific Instruction Set Processor for SoC Implementation of Modern Signal Processing Algorithms," *IEEE Trans. Circuits and Systems—I: Regular Papers*, vol. 52, no. 4, Apr. 2005.

[15] H. Peters, R. Sethuraman, et al., "Application Specific Instruction-Set Processor Template for Motion Estimation in Video Applications," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 15, no. 4, Apr. 2005.

[16] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Low-Power Data Forwarding for VLIW Embedded Architectures," *IEEE Trans. Very Large Scale Integration Systems*, vol. 10, no. 5, 2002.

[17] A. Terechko, M. Garg, and H. Corporaal, "Evaluation of Speed and Area of Clustered VLIW Processors," *18th Int'l Conf. VLSI Design*, 2005, pp. 557-563.

[18] M. Valenti and J. Sun, "The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radios," *Int'l Journal of Wireless Information Networks*, vol. 8, no. 4, 2001.

[19] W. Ebel, "Turbo-Code Implementation on c6x," *Tech. Rep., Alexandria Research Inst.,Virginia Polytechnic Inst. State Univ.*, 1999.

[20] A. La Rosa, L. Lavagno, and C. Passerone, "Implementation of a UMTS Turbo Decoder on a Dynamically Reconfigurable Platform," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, 2005.

[21] H. Michel, A. Worm, M. Munch, and N. Wehn, "Hardware/Software Trade-Offs for Advanced 3G Channel Coding," *Design, Automation and Test in Europe Conf. and Exhibition*, 2002, pp. 396-401.

[22] F. Kienle, H. Michel, F. Gilbert, and N. Wehn, "Efficient MAP-Algorithm Implementation on Programmable Architectures," *Advances in Radio Science*, no. 1, pp. 259-263, 2003.

[23] O. Muller, A. Baghdadi, and M. Jzquel, "Asip-Based Multiprocessor SOC Design for Simple and Double Binary Turbo Decoding," *Proc. the Conf. Design, Automation and Test in Europe (DATE), Munich, Germany*, ACM, Ed., 2006.

[24] A. Corporation, *MegaCore Function User Guide Turbo Encoder/Decoder, 2003*.

[25] T. Vogt and N. Wehn, "A Reconfigurable Application Specific Instruction Set Processor for Viterbi and Log-Map Decoding," *IEEE Workshop on Signal Processing (SIPS)*, Oct. 2006.

[26] M.C. Shin and I.C. Park, "A Programmable Turbo Decoder for Multiple 3G Wireless Standards," *IEEE Int'l Solid-State Circuits Conf.*, vol. 1, 2003, pp. 154-484.

[27] 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*, vol. 37, no. 11, 2002.

[28] S.-J. Lee, N. Shanbhag, and A. Singer, "A 285-MHz Pipelined MAP Decoder in 0.18-/spl mu/m CMOS," *IEEE Journal of Solid-StateCircuits*, vol. 40, no. 8, 2005.

[29] I. Xilinx, *3GPPTurbo Decoder v2.0, 2006*.

[30] J. Kaza and C. Chakrabarti, "Design and Implementation Of Low-EnergyTurbo Decoders," *IEEE Trans. VLSI Systems*, vol. 12, no. 9, Sept. 2004, pp. 968-977.

[31] C. Schurgers, F. Catthoor, and M. Engels, "Memory Optimization of MAP Turbo Decoder Algorithms," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, 2001.

[32] S. Kim, S.Y. Hwang, and M.J. Kang, "A Memory-Efficient Blockwise Map Decoder Architecture," *ETRI Journal*, vol. 26, no. 6, Dec. 2004, pp. 615-621.

[33] Z. Wang, Z. Chi, and K. Parhi, "Area-Efficient High-Speed Decoding Schemes for Turbo Decoders," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 6, 2002.

[34] M. Mansour and N. Shanbhag, "VLSI Architectures for SISO-APP Decoders," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 4, 2003.

[35] C.Wu, M. Shieh, C.Wu, Y. Hwang, and J. Chen, "VLSI Architectural Design Tradeoffs for Sliding-Window Log-MAP Decoders," *IEEE Trans. VLSI Systems*, vol. 13, no. 4, Apr. 2005, pp. 439-447.

[36] M. Thul and N. Wehn, "FPGA Implementation Of Parallel Turbo-Decoders," *IEEE 17th Symp. Integrated Circuits and Systems Design*, Sept. 2004, pp. 198-203.

[37] G.. Prescher, T. Gemmeke, and T. Noll, "A Parametrizable Low-Power High-Throughput Turbo-Decoder," *IEEE ICASSP*, Mar. 2005, pp. 25-28.

[38] J. Tan and G. Stuber, "New SISO Decoding Algorithms," *IEEE Trans. Comm.*, vol. 51, no. 6, 2003.

[39] D.-S. Lee and I.-C. Park, "Low-Power Log-MAP Turbo Decoding Based on Reduced Metric Memory Access," *IEEE Int'l Symp. Circuits and Systems*, vol. 4, 2005, pp. 3167-3170.

[40] M. Gschwind, V. Salapura, and D. Maurer, "FPGA Prototyping of a RISC Processor Core for Embedded Applications," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 2, 2001.

Pablo Ituero is a PhD candidate in the Department of Electronic Engineering at the Universidad Politécnica de Madrid, Spain. He received his MS degree in telecommunications with a major in electronics from the same university in 2005 and his MS in electrical Engineering, specialized in System-on-a-Chip-design from the Royal Institute of Technology, Sweden, also in 2005. His research interests include high-performance processor architectures and low-power, thermal-aware electronic design.

Marisa López-Vallejo is an associate professor in the Department of Electronic Engineering at the Universidad Politécnica de Madrid, Spain. She received the MS and PhD degree from the same university in 1993 and 1999, respectively. She was with Lucent Technologies at Bell Laboratories as a member of the technical staff. Her research activity is currently focused on low-power design, CAD for hardware/software codesign of embedded systems, and application-specific high-performance programmable architectures.