A HIGH-SPEED MAP ARCHITECTURE WITH OPTIMIZED MEMORY SIZE AND POWER CONSUMPTION[1]

Alexander Worm[2], Holger Lamm, Norbert Wehn
Institute of Microelectronic Systems
University of Kaiserslautern
Kaiserslautern, Germany
Email: {worm, lamm, wehn}@eit.uni-kl.de

Abstract - This paper presents a novel high-speed maximum a posteriori (MAP) decoder architecture with optimized memory size and power consumption. Area and power consumption are both reduced significantly, compared to the state-of-the-art. The architecture is also capable of decoding recursive systematic convolutional codes which are the constituent codes of the revolutionary Turbo-Codes and related concatenation schemes. The architecture is highly scalable with respect to throughput, expanding its applicability over a wide range of throughput requirements (300 Mbit/s–45 Gbit/s and above).

INTRODUCTION

The MAP algorithm is a maximum-likelihood decoding method which minimizes the probability of symbol (or bit) error. In other words, a MAP decoder finds for each time-step the most likely information bit to have been transmitted given a received noisy or distorted sequence, thus minimizing the bit-error rate (BER). This is unlike a Viterbi decoder [1] which finds the most likely information bit sequence (or code word). Most importantly, the MAP decoder inherently provides a soft output, that can be effectively used for decoding concatenated codes. With respect to coding gain, the MAP algorithm is superior to the soft output Viterbi algorithm (SOVA) [2] which produces qualitatively inferior soft outputs. This is especially important for latest code concatenation schemes, e. g. Turbo-Codes [3]. The MAP algorithm is therefore an increasingly important building block in present and future communication systems. The significant interest during recent years in high-speed implementations of the Viterbi algorithm can be expected to migrate towards high-speed MAP implementations.

Architectural experiments have shown that up to 50–80% of the area cost in (application-specific) architectures for real-time signal processing is due to memory units. The power cost is even more heavily dominated by storage and transfers of complex data types [4]. Smaller on-chip memories, if accessed in a first-in

[1] Patent application filed by Motorola, Inc.

[2] Alexander Worm is the recipient of a Motorola Partnerships in Research Grant.

first-out (FIFO) manner, are often implemented as register-chains which are power hungry due to the high clock-net load and switching activity. Power consumption depends linearly on both switching rate and capacity and can therefore be reduced very effectively by minimizing FIFO memory size. Thus, memory transfer and size optimization on an architectural level is an essential prerequisite to obtain area- and energy-efficient high-speed decoder implementations. This is especially true for a high-speed MAP decoder which requires a lot of memory due to the forward-backward structure of the algorithm and due to heavy pipelining.

THE MAP ALGORITHM

The MAP algorithm will not be described in detail here (see, e. g., [5, 3, 6]). We state the important results for non-systematic convolutional (NSC) and for recursive systematic convolutional (RSC) codes only. The MAP algorithm computes the *log-lilekihood ratio* (LLR) of the source symbol I_k in step k, conditioned on the knowledge of the received distorted symbol sequence $z_0^N = (z_0, z_1, \ldots, z_k, \ldots, z_N)$:

$$\Lambda_k = \log \frac{\Pr\{I_k = 1 | z_0^N\}}{\Pr\{I_k = 0 | z_0^N\}}. \tag{1}$$

Computation of $\Pr\{I_k | z_0^N\}$ is done by determining the probability to reach a certain encoder state m after having received k symbols $z_0^{k-1} = (z_0, z_1, \ldots, z_{k-1})$:

$$\alpha_k(m) = \Pr\{m | z_0^{k-1}\} \tag{2}$$

and the probability to get from encoder state m' to the final state in step N with the symbols z_{k+1}^N:

$$\beta_{k+1}(m') = \Pr\{z_{k+1}^N | m'\}. \tag{3}$$

The probability of the transition $m \to m'$ using the source symbol I_k, under knowledge of the received symbol z_k, is called γ_k:

$$\gamma_k(m, m', I_k) = \Pr\{m, m', I_k | z_k\}. \tag{4}$$

The probabilities α_k and β_{k+1} can be found recursively over the γ_k, which are a function of the received symbols and the channel model. Knowing these values for each transition $m \to m'$, the probability of having sent the symbol I_k in step k is the sum over all paths using the symbol I_k in step k. With $\phi(I_k)$ being the set of all transitions with symbol I_k, we can write:

$$\Pr\{I_k | z_0^N\} = \sum_{(m,m') \in \phi(I_k)} \alpha_k(m) \cdot \gamma_k(m, m', I_k) \cdot \beta_{k+1}(m'). \tag{5}$$

In the case of NSC codes, equation (5) can be written as:

$$\Pr\{I_k | z_0^N\} = \sum_{m' \in \tilde{\phi}(I_k)} \alpha_{k+1}(m') \cdot \beta_{k+1}(m'). \tag{6}$$

Here, $\tilde{\phi}(I_k)$ is the set of all states reached by a transition with symbol I_k in step k.

It has been pointed out in [7] that it is mandatory to implement the MAP algorithm in the log-domain (Log-MAP) to avoid numerical problems without degrading the decoding performance. This simplifies the arithmetic operations: multiplication becomes addition, and addition becomes the max*-operator [7]. The Max-Log-MAP algorithm [7] is obtained by using the approximation $\max^*(\xi_1,\xi_2) \approx \max(\xi_1,\xi_2)$. Throughout the remainder of this paper, the definitions $\delta = \log\alpha$, $\varepsilon = \log\beta$, $\mu = \log\gamma$ are introduced for notational convenience.

RELATED WORK

Efficient MAP decoder implementation on custom hardware is a widely unexplored area of research. The most advanced approach was presented by Dawid *et al.* in [8, 6]. Their architecture is denoted in the following as *D-NSC architecture*.

For high-speed architectures, the frame can be tiled into a number of windows, each of them W steps wide. Each window can be processed independently from each other, thus providing a high degree of parallelism. The data-dependency graph of the D-NSC architecture introduced in [6] for one of those windows is shown on the left side of Figure 1 for $W = 6$. For reference, the trellis of a constraint length $K = 3$ code is shown on the top. The trellis of a convolutional code is the unrolled encoder state-chart where nodes having the same label are merged. Throughout the remainder of this paper, a code constraint length of $K = 3$ is assumed where applicable, by way of example only. The trellis steps are numbered from left to right; in the data-dependency graph, time proceeds from top to bottom. The data-dependency graph visualizes the data flow between the arithmetic units. There are different arithmetic units for forward acquisition (explained below), backward acquisition, forward recursion, backward recursion, forward recursion with LLR calculation, and backward recursion with LLR calculation. The data-dependency edges each correspond to several values. For example, 2^{K-1} values correspond to each δ and ε edge of the data-dependency graph; a respective number of values corresponds to each μ edge.

The algorithm works as follows: Within a window, forward and backward recursion start simultaneously at different trellis steps, $k-M$ and $k+M$. The branch metrics $\mu_{k-M}\cdots\mu_{k+M-1}$ are assumed to be pre-computed. Theory shows that the path metrics δ and ε, which are unknown at this stage, set to an arbitrary value, will converge during the first M recursions towards reliable values [8, 6] (M is called the acquisition depth). This phase is called the acquisition phase and, in [6], equals half of the window width W, i. e. $W = 2M$. Reasonable values for the acquisition length M are determined by the encoder, e. g. $M = 10$–20 for a constraint length $K = 3$ code [6]. At the end of the acquisition phase, the path metrics have become reliable and the values can be stored. After another M steps, both recursions have reached the end of the window and are continued in the next one: ε_{k-M} and δ_{k+M} are fed into the left and right neighboring window, respectively; δ_{k-M} and ε_{k+M} are obtained from these windows, and recursions continue with concurrent calculation of the outputs Λ_k. As said above, all windows of a frame are processed in parallel. Thus no timing problems arise at path metric exchange between adjacent windows. Except proper initialization of the path metrics, no special arrangements are necessary at either end of a frame.

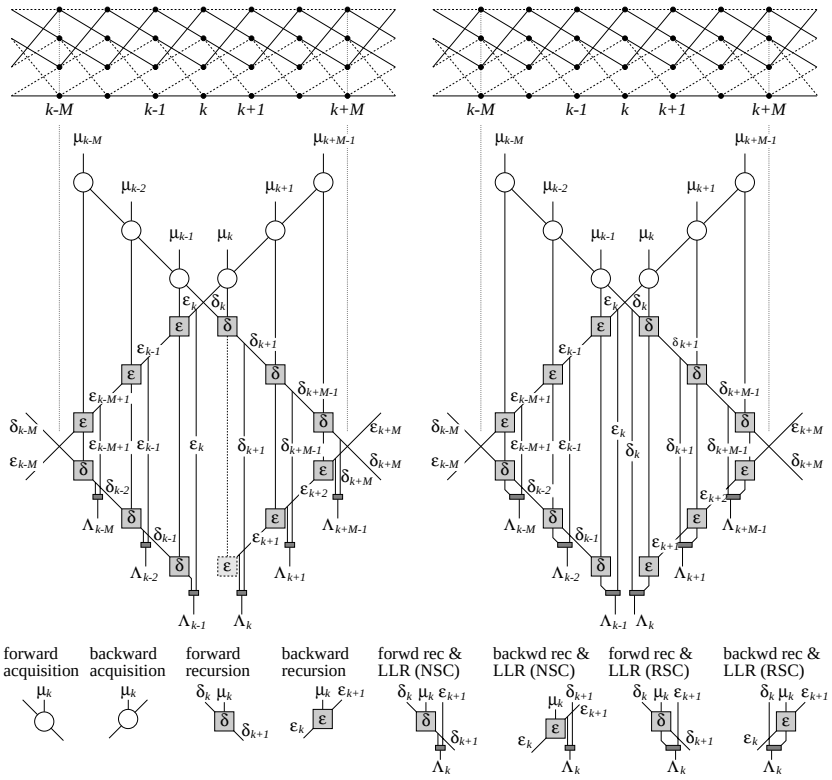

Figure 1: Detailed data-dependency graph of the D-NSC and D-RSC architecture (one window)

The *D* in the denotation *D-NSC* was chosen because the rhombus shape in the lower part of the window resembles a diamond. The D-NSC architecture lacks two important requirements for contemporary and future applications: First, it is restricted to decoding NSC codes. The architecture is not capable of decoding RSC codes. However, these are essential for state-of-the-art communications, e. g. Turbo-Codes. Second, the D-NSC architecture was tuned for high-speed, but was not fully optimized with respect to area and power consumption. This will be proved by comparison with our newly introduced architecture which is superior with respect to area and power consumption while retaining throughput. The D-NSC architecture is therefore sub-optimal with respect to cost-efficiency.

NEW HIGH-SPEED MAP ARCHITECTURE

NSC and *RSC* denote different schemes to construct convolutional codes, given a set of elements capable of storing data (e. g. registers). Being largely disregarded until recently, RSC codes have significantly gained in interest as constituent codes of novel code concatenation schemes [3].

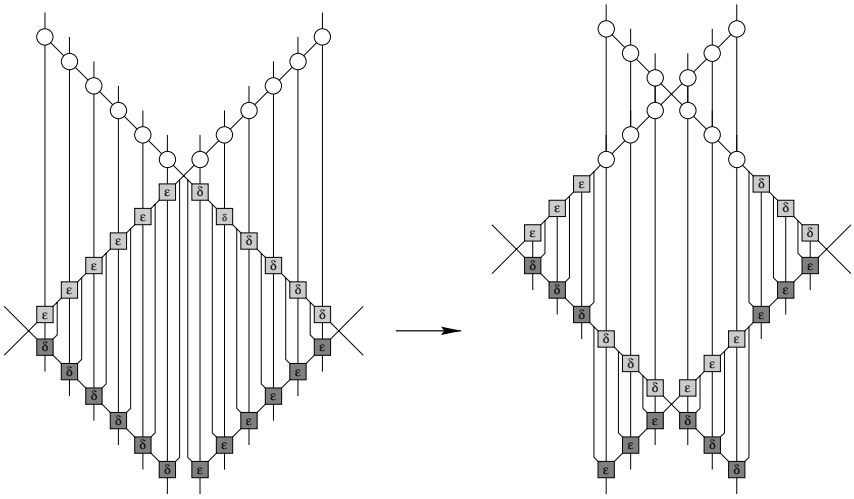

Figure 2: Deduction of the X-RSC data-dependency graph from the D-RSC data-dependency graph

The D-NSC decoder requires the values δ_{k+1} and ε_{k+1} to calculate an output Λ_k. An architecture for one window consumes the branch metrics $\mu_{k-M} \dots \mu_{k+M-1}$ and produces the soft outputs $\Lambda_{k-M} \dots \Lambda_{k+M-1}$. The LLR units therefore need the path metrics $\delta_{k-M+1} \dots \delta_{k+M}$ and $\varepsilon_{k-M+1} \dots \varepsilon_{k+M}$. The consequences for the internal data dependencies of the recursion units are shown on the left side of Figure 1: Forward recursion units calculate the successor δ_{k+1} and issue δ_{k+1} afterwards. Backward recursion units instead issue ε_{k+1} first and then calculate the predecessor ε_k. The same applies for the internal data dependencies of the recursion-and-LLR units. Forward recursion determines the successor and then feeds the result into the LLR part, while backward recursion feeds the received value into the LLR and then determines the predecessor. Thus, a forward path metric δ_k is always used *after* determining the successor. It has passed through one more recursion unit than the backward path metric ε_k and is provided with one extra acquisition cycle.

For RSC code decoding capability, the LLR calculation has to be modified. This yields the data-dependency graph on the right side of Figure 1. We name the corresponding architecture *D-RSC architecture*. As it can be seen, the data-dependencies just differ in details. Instead of δ_{k+1} and ε_{k+1}, the LLR units consume δ_k, μ_k, and ε_{k+1}. Hence, the forward recursion units, like the backward recursion units, hand out δ_k *before* determining the successor. Most of the calculations needed for the LLR are intermediate results of the forward (or backward) recursion. Thus they are just slightly more costly than the LLR units of the D-NSC decoder.

Optimized Architecture

For the RSC decoder scheme derived in the previous section, we introduce a new data-dependency graph which will lead to an architecture with reduced area demand

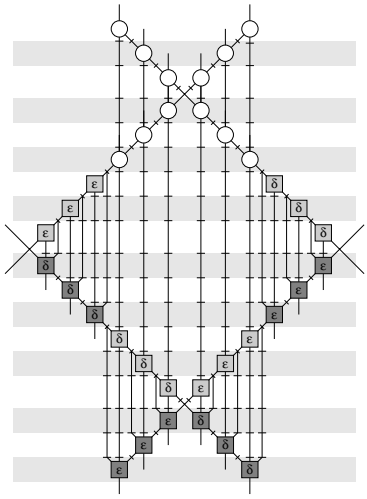

Figure 3: X-RSC architecture for $M = 6, \eta = 2$

and power consumption. This new data-dependency graph can (of course) also be used to build optimized NSC decoders.

Memory usage in heavily pipelined architectures is proportional to the life-time of a value which is in our figures visualized as the length of the corresponding data-dependency edge. In the data-dependency graph of the D-RSC architecture in Figure 1, the longest δ/ε paths are those of the first few values generated after finishing acquisition, with indices around k. Removing these edges could have great effect on memory.

The new *X-RSC data-dependency graph* is shown on the right side of Figure 2. LLR calculation has been merged with recursion for clarity, indicated by darker shading. Note that if the recursions are continued at the bottom of Figure 2 (left), the produced values could be directly used for LLR calculation. As a result, the recursions creating the upper borders of the rhombus shape do not need to produce reliable δ and ε values. They can be used for acquisition. This leads to the data-dependency graph of the novel *X-RSC architecture* shown on the right side of Figure 2. The letter *X* in *X-RSC* refers to the X-shaped structure at the bottom. In the following section, we show the advantage of this new data-dependency graph with respect to memory demand and power consumption.

Comparison

We consider maximum throughput architectures. To achieve maximum throughput, architectures have to be fully pipelined. The life-times of the values then directly correspond to the respective number of pipeline stages. This is assumed throughout the remainder of this section. Sharing can be applied on several levels in order adapt the architecture throughput to the system throughput.

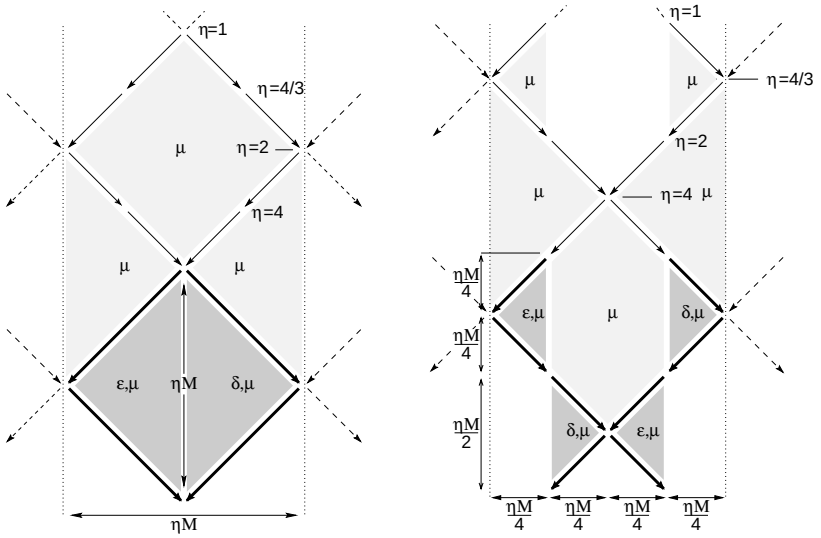

Figure 4: D-RSC (left) and X-RSC (right) data-dependency graph for general η

	latency l	arithmetic	δ/ε words	μ words
D-RSC	$(\eta+1)M-1$	$1+\frac{1}{\eta}$	$\frac{\eta^2M^2}{2}+\eta M+2M-2$	$\frac{\eta^2M^2}{2}+\eta M^2$
X-RSC	$(\eta+1)M-1$	$1+\frac{1}{\eta}$	$\frac{\eta^2M^2}{4}+\eta M+2M-2$	$\eta M^2+\frac{3\eta^2M^2}{8}$

Table 1: Latency, normalized number of arithmetic units, and memory demand for fully pipelined D-RSC and X-RSC architectures. One word (which corresponds to one FIFO stage) stores 2^{K-1} δ or ε values or a respective number of μ values.

In the previous sections, the length of the recursion paths producing reliable values (which is equal to the window width W) was assumed to be twice the acquisition depth M, i. e. $W = 2M$. In general, we write $W = \eta M$. Figure 3 features a fully pipelined X-RSC architecture with $M = 6$ and $\eta = 2$. Small horizontal and diagonal bars denote pipeline registers arranged along the data-dependency edges and between the arithmetic units to realize FIFO functionality.

For a fixed window width, η is inverse proportional to M. For $\eta < 4/3$, the acquisition paths cross the window borders and therefore have to be started in the neighboring windows. Different η obviously influence memory size, latency of the architecture, and arithmetic demand. This situation is depicted in Figure 4 for D-RSC and X-RSC data-dependency graphs. There, grey shaded areas denote data storage and the degree of shading reflects the amount of data to be stored. For example, the shading of the lower rhombus shape in the D-RSC data-dependency graph is darker than the shading of the upper rhombus and the two triangular shapes, because δ and ε values have to be stored in addition to μ values. Note that the total area of dark shading in the X-RSC data-dependency graph is significantly smaller than in the D-RSC data-dependency graph.

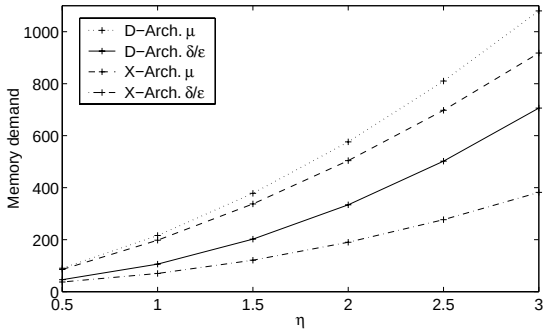

Figure 5: Memory demand in terms of FIFO steps for D-RSC and X-RSC architectures for $M = 12$, $\eta = 2$

For fully pipelined implementations, data dependencies are mapped on FIFOs (register chains, for instance), so memory usage is a linear function of the dependency length[3]. There is a pipeline step between successive forward and backward recursions. Thus, the architecture, from the first acquisition to the last result, has a latency of $l = (\eta + 1) \cdot M - 1$. This fully-pipelined architecture accepts a new frame every cycle: up to $(\eta + 1)M$ frames are processed simultaneously.

Table 1 features the architectural latency, the normalized number of arithmetic units, and the amount of memory to store the δ, ε, and μ values for both architectures. The table shows that our new architecture yields a significant improvement: for example, memory area is reduced by 30% for $M = 12$ and $\eta = 2$ and bit-widths of 6 and 7 for μ and δ/ε, respectively. If we assume that memory accounts for approximately half of the equivalent gate count (the quota increases with window width) and taking the high switching activity of register chains into account, we conclude that a 20% reduction in power consumption is achievable for this example by using an architecture which is based on our novel X-shaped data-dependency graph. Thus, in comparison to the D-RSC architecture, memory demand for the X-RSC architecture has been significantly reduced, saving area and power consumption, while arithmetic and latency stay the same. Figure 5 compares for $M = 12$ and various η the memory requirements of the novel X-RSC architecture with the D-RSC architecture derived from the previously known D-NSC architecture. In general, memory demand and latency are smaller for small η, but the number of arithmetic units is increased. So it is necessary to find a tradeoff with respect to area and power consumption. The increase of δ/ε for low η is a result of the rising influence of the $2M - 2$ registers between acquisition steps.

Normally, a trade-off between communications performance (here: achievable BER at a signal-to-noise ratio E_s/N_0) against VLSI performance (throughput, area, power consumption) is necessary. However, we can show that using our new X-

[3]Small local SRAM could turn out more energy-efficient for large FIFOs. This is not considered here, since in most cases this is offset by the SRAM address decoding and sensing overhead. Moreover, many SRAM generators would not accept the required small SRAM depths combined with large bit-widths. However, the amount of memory stays the same, independent of the implementation style.

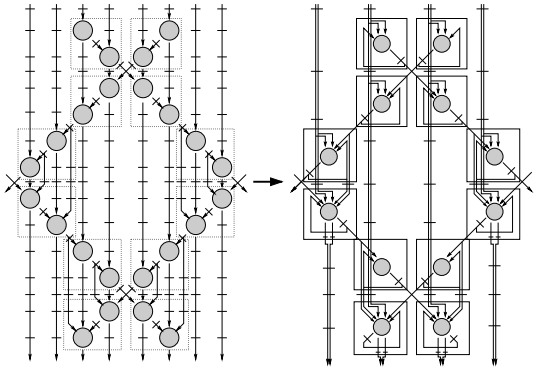

Figure 6: Sharing arithmetic units

	Max-Log-MAP decoder			Log-MAP decoder		
	D-RSC	X-RSC	Δ	D-RSC	X-RSC	Δ
Area [μm^2]	5167903	4275461	-18%	7975967	7070060	-11%
Power [mW]	737	586	-20%	944	798	-15%

Table 2: Area and power consumption of one window ($M = 12$, $\eta = 2$, $\theta = 1$)

shaped data-dependency graphs instead of the D-shaped data-dependency graphs does not alter communications performance. Moreover, parallel window processing does not degrade the BER if the acquisition depth M is large enough (see above). Thus no simulation results are shown.

SHARED ARCHITECTURES

The new architecture processes one frame every cycle. Because each window needs information from its left and right neighboring windows, the structure has to be repeated horizontally to fill up the block width. This technique yields a very high throughput which is probably too high for many applications. An optimized architecture (i. e., architectural throughput should not be much higher than system throughput) for moderate throughput can be derived by sharing arithmetic units within a window or by sharing windows. (Reducing clock speed is also an alternative, but this results in an inefficient architecture.) When sharing arithmetic units, $\theta > 1$ consecutive recursion steps are carried out on the same hardware. This reduces pipeline length and increases the data init interval by θ. For example, for $\theta = 2$ the number of arithmetic units is halved, but they have to be enhanced by multiplexers. As depicted in Figure 6, the shared architecture behaves like an architecture with narrower window-width $\frac{\eta M}{\theta}$. Thus, the amount of memory is also reduced. Note that in Figure 6 no distinction in drawing between the various types of arithmetic units has been made for simplicity. To share several windows, one window is built in hardware. In this case, the frames are queued window by window into the pipeline.

RESULTS

Parameterizable VHDL models have been developed and synthesized to validate the theoretically derived data about area reduction and the estimated savings in power consumption. We used a $0.2\,\mu$m state-of-the-art standard cell library and the Synopsys tool suite for synthesis and power characterization.

Synthesized Max-Log-MAP decoder architectures based on the X-shaped data-dependency graph yield a throughput of 75 Gbit/s for a block size of $L_{\text{block}} = 600$ bit. The Log-MAP decoder would reach 45 Gbit/s. Throughput can by scaled down to 300 Mbit/s by means of sharing. Table 2 shows total area and power consumption for a single window of a (Max-)Log-MAP decoder with $M = 12$, $\eta = 2$, and a folding factor of $\theta = 1$ after synthesis. Both Max-Log-MAP and Log-MAP decoder have been synthesized with the same timing constraints (45 Gbit/s throughput). The improvements of the X-RSC architecture over the D-RSC architecture are significant; 20% power reduction has been achieved, which validates our estimates.

CONCLUSIONS

Based on the state-of-the-art data-dependency graph structure for high-speed MAP decoders, a new MAP dependency-graph structure and an architecture also capable of decoding RSC codes have been developed. Savings in area and power consumption of 18% and 20%, respectively, have been validated by a synthesizable VHDL model using a state-of-the-art $0.2\,\mu$m standard cell library.

References

[1] G. D. Forney, Jr. The Viterbi Algorithm. *Proc. IEEE*, 61(2):268–278, 1973.

[2] J. Hagenauer and P. Hoeher. A Viterbi Algorithm with Soft-Decision Outputs and its Applications. In *Proc. Globecom '89*, pages 1680–1686, 1989.

[3] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. In *Proc. ICC '93*, pages 1064–1070, 1993.

[4] F. Catthoor et al. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[5] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate. *IEEE Tr. Inf. Theory*, IT-20:284–287, 1974.

[6] H. Dawid. *Algorithmen und Schaltungsarchitekturen zur Maximum a Posteriori Faltungsdecodierung*. PhD thesis, RWTH Aachen, Shaker Verlag, 1996.

[7] P. Robertson, P. Höher, and E. Villebrun. Optimal and Sub-Optimal Maximum a Posteriori Algorithms Suitable for Turbo Decoding. *ETT*, 8(2):119–125, 1997.

[8] H. Dawid, G. Gehnen, and H. Meyr. MAP Channel Decoding: Algorithm and VLSI Architecture. In *VLSI Signal Processing VI*, pages 141–149. IEEE, 1993.