Fast SUBMAP decoders for duo-binary turbo-codes

Yannick Saouter, Claude Berrou

Fast SUBMAP decoders for duo-binary turbo-codes

Yannick Saouter[1] and Claude Berrou[2]

Abstract - **Since 1993, turbo-codes [1] have been used in several communications systems because of their high performance in error correction over noisy channels. In data frame applications, the main decoding algorithm used in this technology is the suboptimal *maximum a posteriori* algorithm (SUBMAP). In this paper, we describe several implementations of a SUBMAP decoder for duo-binary codes, the final one being optimized for speed.**

Index Terms - **Turbo-codes, SUBMAP, VLSI, Gigabit networking.**

I. Introduction

ETSI (European Telecommunications Standards Institute) is in the process of adopting new channel coding for interactivity in Digital Video Broadcasting (DVB) for satellite (RCS: Return Channel over Satellite) and terrestrial (RCT: Return Channel over Terrestrial) transmission. The error-correcting code is a duo-binary 8-state convolutional turbo code, specified for various block sizes (from 12 to 216 bytes) and coding rates (from 1/3 to 6/7). In this paper, we study the perspective of the silicon integration of decoders for DVB-RCS with high throughputs ($\geq$ 300 Mbits/s). To achieve this, we consider the SUBMAP as the component decoder in the turbo decoding iterative process. This paper describes the design of the device, with the prospect of integration into 0.18 μm technology.

II. Turbo-codes

In this section, we make a brief description of turbo encoding. Full details on this topic can be found for instance in [2]. Typically a turbo-code is built with two recursive convolutional encoders processing the same set of data: the first one performs the encoding in the natural order while the second one operates on a permuted order. In data frame applications, which are used in most communications systems, the permutation operates on a single frame. The choice of permutation is critical for performance at low error rates. The system output is thus composed of the data taken in natural order and the two redundancy sets produced by each convolutional encoder. Those outputs are then emitted on the transmission channel.

The decoder samples input data from the transmission channel. These data are not binary hard decisions but soft decisions i.e. the data have integer values which are proportional to the intensity of the signal read from the channel. The decoder thus operates in a cascading way: the first decoder uses the sampled data in their natural order together with the first redundancy set to determine a first guess of the data. This first estimate, called the *extrinsic value*, is then used by the second decoder, together with the permuted data samples and the second redundancy set to produce a second estimate which will be used by the next decoder. Typically such systems are composed of 8 to 16 decoders serially connected and exchanging extrinsic values.

Although initially designed with binary convolutional codes, M-ary convolutional codes can also be used in a turbo-code system. They allow better correcting performance and limit the flattening imperfection observed with classical turbo-codes [3]. Circular codes [4] are also extremely useful since they avoid the use of terminating symbols. As a consequence, our implementation was made around a duo-binary circular code. The length of the frame was set to 992 duo-binary symbols, hence 248 bytes, whose entire specification can be found in [5].

III. The SUBMAP Algorithm

This section is devoted to the SUBMAP algorithm which was introduced by Robertson et al. [6] as a simplification of the MAP algorithm [7]. The SUBMAP algorithm is a SISO (soft-in soft-out) decoding algorithm for convolutional codes: given input data, it computes an estimate of the probability that an information bit is equal to 0 or 1, unlike the conventional Viterbi algorithm which gives only a binary decision. Convolutional codes are lattice codes: they can be described by a set of vertices, called *states* of the code. Each state has two outgoing edges (four in the case of duo-binary codes) leading to a new state. Each edge is labelled by the information symbol and the produced redundancy. It has to be noted that if we transform the outgoing edges by incoming ones, we again obtain a lattice code called the *reciprocal code*. The SUBMAP algorithm takes advantage of these two codes. Thus we have two different phases: the first one using the initial code called the *forward* phase and the second one using the reciprocal code called the *backward* phase.

The forward phase computes at each time t, for any state s of the initial code, an estimate of the probability that the encoder was in state s at time t during the encoding of the data sequence. This value is called the *state metric* of the state s. More precisely, if $X_1(t)$,

[1] Yannick.Saouter@enst-bretagne.fr
[2] Claude.Berrou@enst-bretagne.fr

$X_2(t)$ are the values read from the channel for the first and second bit of the data respectively, $R(t)$ is the value read for the redundancy and $EXTR(a,t)$ the extrinsic value at time t for any of the four symbols a of information data tokens and then the forward state metric of state s at time t is $FSM(s,t) = Min_a\{IFSM(a,s,t)\}$ where $IFSM(a,s,t)$ is the incoming forward state metric of state s at time t for symbol a which is computed by the formula:

$$\begin{aligned} IFSM(a,s,t) = \\ & SM(PrevState(a,s),t-1) \\ & +(-1)^{(a\&01)}X_1(t)+(-1)^{(a\&10)}X_2(t) \\ & +(-1)^{PrevRed(a,s)}R(t)+EXTR(a,t) \end{aligned}$$

where $PrevState(a,s)$ is the state whose outgoing edge labelled with the information symbol a leads to s, $(a\&01)$ (resp. $(a\&10)$) is the value of the first (resp. second) bit of symbol a and $PrevRed(a,s)$ is the value of the redundancy labelling the edge between the states $PrevState(a,s)$ and s. Similarly, the backward phase computes the backward state metrics $BSM(s,t)$ on the reverse frame, using the transitions of the reciprocal code.

The next step of the algorithm takes into account the values of the forward the and backward metrics. This step produces values called ***weights*** for each symbol a at any time t: the smaller the value of the weight for symbol a, the greater the probability for this symbol at time t. The formula for the weight is:

$$\begin{aligned} W(a,t) = \\ & Min_s\{FSM(PrevState(a,s),t-1)+BSM(s,t) \\ & +(-1)^{(a\&01)}X_1(t)+(-1)^{(a\&10)}X_2(t) \\ & +(-1)^{PrevRed(a,s)}R(t)+EXTR(a,t)\}. \end{aligned}$$

At this point the output extrinsic value, which will be given as an input to the following decoder, can be computed as:

$$\begin{aligned} EXTROUT(a,t) = \\ & 1/2(W(a,t)-(-1)^{(a\&01)}X_1(t)-(-1)^{(a\&10)}X_2(t)) \\ & -EXTR(a,t). \end{aligned}$$

IV. Global architecture

In fact, the whole SUBMAP algorithm is not implementable in practical cases because it would require a large amount of data memory.

Several solutions have been proposed to circumvent the problem [8, 9]. The one we chose, proposed by Viterbi [9], uses the property of convergence of the state metrics along the lattice: when performing a backward or forward estimation of the state metrics on a frame, after a number of data samples which is typically five or six times the constraint length of the code, the estimates do not vary much. Thus, in the backward phase, we split the data samples frame into windows whose length is large enough to ensure the convergence of the state metrics. Then the backward metrics are computed by two backward processors: one training on even windows and computing on odd windows, the other one training on odd windows and computing on even windows. By using a proper pipeline execution scheme of the forward phase, it is then necessary to store only two windows of metrics. The length of each window was set to 48 and it thus leads to a global memory size of $2 \times 48 \times 8 \times 8 = 6144$ RAM memory bits. The global architecture of the SUBMAP decoder is presented in figure 1.

V. Speed improvements

Three architectures of the same decoder were derived: the first one was designed in such a way as to be minimal in area with a minimal working frequency of at least 250 MHz, while the second and third designs were optimized for speed. In the second design, we nevertheless limit the resulting increase in area for the circuit, while the third design even uses costly time optimizations.

A. Anticipation of the branch metrics computations

In a SUBMAP design, the critical path is in the computation of state metrics. Indeed, at each clock edge, the new state metrics have to be computed from the previous ones and the data samples. Thus, in order to obtain high speed, it is necessary to limit the number of explicit operations to be performed in the critical path. One classical optimization is to compute a priori the value $(-1)^{(a\&01)}X_1(t) + (-1)^{(a\&10)}X_2(t) + (-1)^{PrevRed(a,s)}R(t) + EXTR(a,t)$ occurring in the expression of $SM(s,t)$, for any symbol a and any state s. This computation can be done in a pipeline way, delaying the explicit computation of $FSM(s,t)$ by the length of the pipeline. This optimization is necessary for all designs. With this feature, the critical path then contains a single addition and a minimum amongst four values.

B. Efficient overflow strategy

In fact, state metrics have to be realigned. Indeed, if nothing is done, the state metrics will overflow. Since only differences of state metrics are relevant in the SUBMAP algorithm, one may subtract the same quantity from each of them at any time, without modifying the algorithm. The conventional method is at any time t to subtract $Min_s\{FSM(s,t)\}$ from all the terms $FSM(s,t)$. However this method would involve computing the minimum out of eight values and making eight parallel subtractions. This method is costly in terms of area and since it occurs inside the critical path, it is also prohibitive for the frequency of the circuit. A

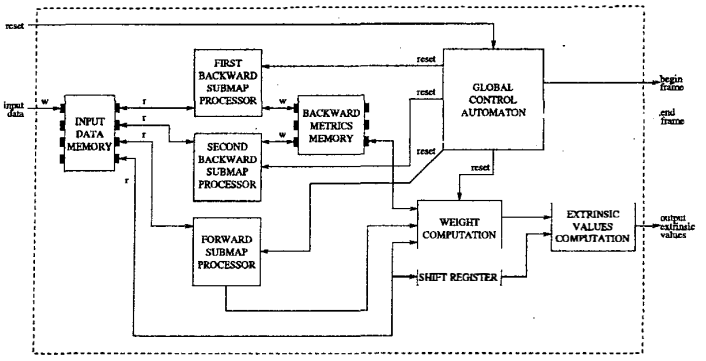

Fig. 1: Overall architecture of the SUBMAP decoder

more efficient technique is to compute the state metrics with one more significant bit than necessary and if all these supplementary bits are equal to 1 at the same time for all the states, they are all set to 0 in parallel. This amounts to making a subtraction of $2^{(k-1)}$ where k is the number of bits for the state metrics. This avoids the use of a real subtracter and the overflow detection is just an **or** gate with 8 entries. This technique is beneficial in all cases and thus has been used in the three designs.

C. Pipelining

Pipelining was also used in the three implementations. It proves to be necessary in the WEIGHT COMPUTATION module and in the EXTRINSIC VALUES COMPUTATION module in order to ensure that their critical path is shorter than the critical path of state metrics computation. But it is also necessary to avoid the latency of read/write memories becoming a bottleneck. As a consequence, access to memories were buffered: a read or a write instruction is taken into account one clock edge later and the result, in the case of a read instruction, is available again one clock edge later. In this way, the access time of the memory devices does not appear in the critical path of the circuit. It has also to be noted that multiple access to the memories (up to 4 in parallel) was made possible by organizing them into separate banks which support a maximum of two read instructions or one write instruction.

D. Computing the minima

The following optimization concerns only the last two designs. In the critical path, a minimum amongst four values has to be performed at each clock edge. The standard way to realize this is to arrange the computations in binary tree fashion. This architecture requires three comparators and two successive comparator stages occur in the critical path. However it is possible to organize the computation in such a way that the critical path contains a single comparator stage. Indeed, to compute the minimum of the four values V_1, V_2, V_3 and V_4, it is possible to perform all the six comparisons $V_i < V_j$ with $i < j$ in parallel. Then the value V_1 will be the minimum if all the boolean values $V_1 < V_2$, $V_1 < V_3$ and $V_1 < V_4$ are true. Likewise V_2 will be the minimum if the boolean values **not**$(V_1 < V_2)$, $V_2 < V_3$ and $V_2 < V_4$ are true. Thus for all the four values V_i, it is possible to compute the boolean value which is true if and only if V_i is the minimum value (this boolean value will be the result of an **and** operation over three comparisons possibly negated). These booleans are then used to correctly drive as the output value the unique value V_i which has its corresponding boolean equal to 1. This selection was made by multiplexers and logical **or** gates in the second design and by three-state output buffers in the third design. Since most of the FPGA circuits do not have three-state buffers in their internal logic, the second design can address FPGA technology while the third one is reserved for ASIC technology. With this technique, the critical path is reduced by a comparator and the small supplementary number of logical gates has a negligible propagation time compared with a comparator device.

E. Fast operators

The last optimization, only used in the third design, was to adopt fast operators instead of the basic ripple carry adders used previously. Several designs were compared: the basic ripple carry adder, the basic carry lookahead adder, the Brent-Kung adder [10], the flat adder (whose carries are all computed in parallel in binary tree structures), the redundant adder [11], the carry lookahead with 1 conditional carry stage and the carry lookahead with 2 conditional carry stages. The structure of a conditional carry adder is organized as

follows: the incoming operands are split into two parts, their least significant bits and their most significant bits. Three additions are performed in parallel: the addition of the least significant bits, the addition of the most significant bits with an incoming carry equal to 0 and the addition of the most significant bits with an incoming carry equal to 1. The output carry of the first adder is then used to select the correct most significant bits of the result amongst the results performed by the last two adders. Of course, this technique can be used recursively on the three component adders. Table 1 summarizes the performances of all the adders considered. These values were obtained by the SYNOPSYS development chain with a 0.18 μ CMOS technology.

While the redundant adder design is the fastest, it is also by far the largest and was discarded for this reason. It is also noteworthy that it is disadvantaged by the short format of its operands. Indeed, its propagation time is constant whatever size the operands are, while for instance carry lookahead adders with 2 conditional carry stages have a propagation time growing linearly with operand size. Thus if the operands had been larger, the redundant adder would have been by far the fastest, while in our implementation the difference is small. We then decided to use the latter type, since the area difference compared with the pure carry lookahead design is negligible.

TABLE 1: Performance and Area for an 8-bit Adder

Architecture type	Area	Critical path time
Ripple Carry Adder	130 gates	1.55 *ns*
Carry Lookahead Adder	183 gates	1.33 *ns*
Brent-Kung Adder	150 gates	2.76 *ns*
Flat Adder	227 gates	2.42 *ns*
Redundant Adder	413 gates	0.95 *ns*
CLA (1 cond. carry stage)	183 gates	1.21 *ns*
CLA (2 cond. carry stages)	189 gates	1.10 *ns*

VI. Performance

In our implementations, the input data were sampled as signed 4 bit integers, while the extrinsic values are unsigned and 6 bits wide, and the state metrics were computed as unsigned 8 bit wide integers. Such values were determined by simulations and exhibit little degradation in terms of power of correction with respect to the ideal case (integers with infinite precision). The implementations were described in the VHDL description language and compiled with the SYNOPSYS development chain to target a 0.18 μ CMOS technology. Table 2 summarizes the performances of all the SUBMAP designs. The SUBMAP decoder is not fully pipelined: 192 preliminary clock edges are necessary before the first quadruplet of extrinsic values is output. Consequently, in our example with data frames of 992 duo-binary symbols, the output data rate of the optimized design is equal to $2 \times \frac{1}{1.87} \times \frac{992}{992+192} = 0.896\ Gbit/s$.

TABLE 2: Performance and Area for the SUBMAP Architectures

	SUBMAP
BASIC	3.54 ns / 282 MHz 28492 gates + 6144 RAM bits
ADVANCED	2.50 ns / 400 MHz 37657 gates + 6144 RAM bits
OPTIMIZED	1.87 ns / 534 MHz 54010 gates + 6144 RAM bits

VII. Conclusion

In this article, we have described three implementations of a SUBMAP decoder for duo-binary turbo-codes. The optimized structure exhibits an output data rate approaching 1 Gbit per second and is thus suitable for most of the high rate transmission systems.

In our future work, we plan to investigate turbo-codes over large M-ary symbol alphabets, that will certainly enable higher data rates to be obtained.

References

[1] C. Berrou and A. Glavieux. Near Shannon limit error correcting coding and decoding: Turbo-codes. In *Intl. Conf. on Comm.*, vol. 2, pp. 1064–1070, Geneva, Switzerland, 1993.

[2] B. Vucetic and J. Yuan. *Turbo Codes - Principles and Applications.* Kluwer Academics Publishers, 2000.

[3] C. Berrou and M. Jezequel. Non-binary convolutional codes for turbo coding. *Electr. Let.*, 35(1):39, Jan. 1999.

[4] C. Bettstetter. Turbo decoding with tail-biting trellises. Diplomarbeit, July 1998.

[5] C. Douillard, M. Jezequel, C. Berrou, N. Brengarth, J. Tousch, and N. Pham. The Turbo Code Standard for DVB-RCS. In *2nd Intl. Symp. on Turbo Codes & Related Topics*, pp. 535–538, Brest, France, Sept. 2000.

[6] P. Robertson, P. Hoeher, and E. Villebrun. Optimal and Suboptimal Maximum a Posteriori Algorithms suitable for Turbo Decoding. *Euro. Tr. Telecomm.*, 8:119–125, Mar.-Apr. 1997.

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

[8] A. Dingninou, F. Raouafi, and C. Berrou. Organisation de la mémoire dans un turbo décodeur utilisant l'algorithme SUBMAP. In *GRETSI'99*, pp. 71–74, Vannes, France, Sept. 1999.

[9] A.J. Viterbi. An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes. *IEEE J. on Sel. Areas in Comm.*, SAC-16(2):260–264, Feb. 1998.

[10] R.P. Brent and H.T. Kung. A Regular Layout for Parallel Adders. *IEEE Tr. Comp.*, C-31:260–264, Mar. 1982.

[11] C.Y. Chow and J.E. Robertson. Logical Design of a Redundant Binary Adder. In *Proc. of the 4th Symp. on Comp. Arith.*, pp. 109–115, Oct. 1978.