2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors

Viterbi Accelerator for Embedded Processor Datapaths

Muhammad Waqar Azhar, Magnus Själander, Hasan Ali, Akshay Vijayashekar,
Tung Thanh Hoang, Kashan Khurshid Ansari, and Per Larsson-Edefors
VLSI Research Group, Dept. of Computer Science and Engineering,
Chalmers University of Technology, 412 96 Gothenburg, Sweden

***Abstract*—We present a novel architecture for a lightweight Viterbi accelerator that can be tightly integrated inside an embedded processor datapath. We investigate the accelerator's impact on processor performance by using the EEMBC Viterbi benchmark and the in-house Viterbi Branch Metric kernel. Our evaluation based on the EEMBC benchmark shows that an accelerated 65-nm 2.7-ns processor datapath is 20% larger but 90% more cycle efficient than a datapath lacking the Viterbi accelerator, leading to an 87% overall energy reduction and a data throughput of 3.52 Mbit/s.**

I. Introduction

Wireless communication requires channel coding to ensure reliable delivery of data over unreliable communication channels. The continuous drive to improve spectrum efficiency imposes a growing need to employ efficient forward error correction (FEC) schemes, for example, Viterbi, Turbo, and low-density parity-checking (LDPC) codes, for many standards (see Table 1 in the work of Krishnaiah et al. [1]). Convolutional encoding is widely used for encoding data in FEC schemes, and the subsequent decoding can be done using, for example, Viterbi decoding or Turbo decoding. Convolutional encoding along with Viterbi decoding is particularly suited to wireless channels, in which the transmitted signal is corrupted mainly by additive white Gaussian noise [2]. The decoding process in such FEC schemes is computationally intensive, and since wireless devices often are synonymous with portable devices, also the requirements on energy dissipation are extremely tight. Customized circuits can indeed efficiently handle decoding with high performance and energy efficiency. However, the lack of flexibility in customized circuitry is a major issue, since there is an ongoing evolution of wireless standards that the hardware needs to adapt to.

The advantages of integrating FEC acceleration inside a processor datapath include instruction-level flexibility and compatibility with standard software development flows. We therefore explore acceleration of Viterbi decoding within the datapath of a standard five-stage single-issue processor—as a representative of embedded processors—with the aim to identify a sweet spot between flexibility and computational efficiency for FEC implementations. We show that integrating such a Viterbi accelerator into the datapath of a placed and routed 65-nm FlexCore processor [3] reduces the execution time for the EEMBC Viterbi benchmark [4] by 90% at an area cost of only 20%.

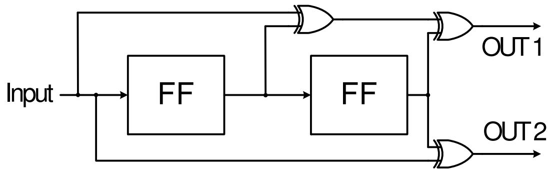

Fig. 1. Example of convolutional encoder. FF denotes flip-flop.

II. Preliminaries

We will review the encoding and decoding phases of convolutional codes, and make an estimation of hardware resources necessary for implementing the Viterbi decoding circuitry.

A. Convolutional Encoding and Viterbi Decoding

A convolutional encoder consists of a shift register with $K-1$ memory elements, where K is called the constraint length. The total number of encoder states is 2^{K-1}. For each new input, the data of the memory elements are shifted one step, which discards the least recent input. An *output symbol* is generated for each new input by applying a generator polynomial on the stored data. A convolutional encoder is shown in Fig. 1 for $K = 3$, $R = 1/2$, and with generator polynomials $G1 = 111_2$ and $G2 = 101_2$. Here, the code rate $R = m/n$ is the ratio of the number of input bits (m) to the number of output bits (n).

The key to achieving error correction is that each input bit has an influence on K successive output symbols [5]. The higher the K, the higher is the complexity of the code and the higher is the error correcting capability. On the downside, an increase in K exponentially increases the decoding complexity as well as the memory required for decoding.

Starting in a certain encoder state, the next state depends on the input bit. The possible state transitions can be visualized in a *trellis diagram* [6]. For the encoder in Fig. 1, the corresponding trellis diagram is shown in Fig. 2. Here, the solid line corresponds to an input bit of 0, whereas a dotted line corresponds to an input bit of 1. The thick lines show the trellis path for the input sequence that is shown underneath the diagram. The general idea is that a sequence of input bits generates a valid path through the trellis diagram, from left to right. In the event of transmission errors, the Viterbi decoder can find the valid path on the trellis diagram that is the closest match to the received sequence [6].

1063-6862/12 $26.00 © 2012 IEEE
DOI 10.1109/ASAP.2012.24

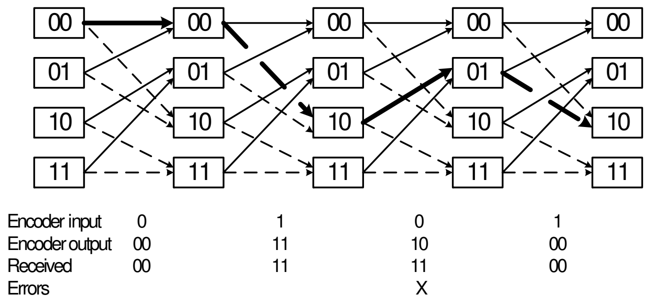

Fig. 2. Example of trellis diagram.

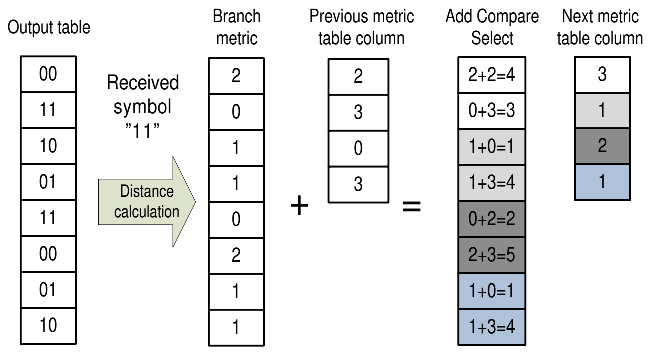

Fig. 3. Snapshot of path metric calculation in Fig. 2. Output table entry 1-4 represent symbols that result from an encoder state of “00”, “01”, “10”, “11” combined with an input bit of 0 (Fig. 1). Similarly, entry 5-8 represent the symbols that result from an input bit of 1.

Considering Fig. 2, we can traverse through a few encoding iterations. Initially the encoder is reset to state “00”. In the first iteration, the input bit is 0, so the encoder will go to state “00” as represented by a thick line. An output symbol of “00” will be transmitted and the Viterbi decoder on the other side of the channel will have to reconstruct the input bit from the received symbol. Since there are only two valid output symbols in each state, if there is an error, the decoder will try to find the state that is closest to the received symbol. In the second iteration, the input bit is 1, so the encoder will go to state “10” and transmit “11” as symbol. In the third iteration, the input is 0, leading the encoder to state “01” with a corresponding output symbol of “10”. However, for some reason, a transmission error occurs, so the received symbol on the decoder side is “11”. Thanks to the Viterbi algorithm, the original symbol can be recovered.

In the *branch metric calculation*, the difference of the received symbol and all possible encoder output combinations is computed. The difference is called *distance* and is either based on the Hamming distance—a hard-decision decoder—or the Euclidean distance—a soft-decision decoder[1]. With 2^{K-1} states and an encoder input bit of either 0 or 1, there exist 2^K output combinations. Based on the generator polynomial, the outputs are computed and stored in an *output table*.

The *path metric calculation* is the most computation-intensive portion of Viterbi decoding. It employs add-compare-select (ACS) operations on the branch metric from the previous step to compute a path metric that is the accumulated distance that is associated with each path through the states. For each pair of branches leading into a given state, the ACS operation discards the branch with the largest accumulated distance, which is the sum of the previous path metric and the branch metric itself. A *metric table* stores the accumulated distance values—the survivor paths—as the decoder receives symbols.

Fig. 3 shows a hardware-oriented view for the two steps above. The output table contains all 2^K encoder outputs. The distance between each symbol in the output table and the received symbol is calculated to get the branch metric. The path metric from the previous iteration is stored in the metric table. The current branch metric values along with the previous path metrics are used to compute the current path metric.

[1]Soft-decision decoding offers a coding gain that is approximately 2 dB higher than that for hard decoding [2].

Survivor path decoding is the last decoding step. Using a hardware-efficient traceback process, the accumulated distances are analyzed, starting by choosing state zero in the last memory column of the metric table. The two possible previous states are identified and the state corresponding to the minimal distance among those two is selected, by storing this in a survivor state table. The traceback process continues in a backward fashion until reaching the first column of the metric table. Finally, by using the survivor state table it becomes possible to recreate the original message.

As has been shown, the main idea of Viterbi decoding is to map a received sequence of symbols, of which some may be corrupted during the transmission, to the most likely *valid* sequence. The basic steps in the decoding process are

1) Branch metric calculation
2) Path metric calculation
3) Survivor path decoding

B. Memory and Computational Requirements

The decoding process is memory intensive and the buffers required for a complete, stand-alone Viterbi accelerator have a significant impact on the implementation area and power. The size of the output table is $2^{K-1} \cdot 2^m \cdot n$ bits, while the metric table is $2^{K-1} \cdot b \cdot (5K+1)$ bits, where b is the number of bits in each metric table entry[2]. Clearly, it is a challenge to implement the metric table in an area-efficient way.

Each entry in the output table is accessed once for every symbol. Consequently, this table is accessed 2^K times for every symbol received during decoding. Each entry in the metric table is accessed twice for every symbol. Consequently, this table is accessed 2^K times. In a processor-based implementation, the memory accesses contribute to a significant overhead in terms of performance and power dissipation and, thus, it is important to use memory in the accelerator to enable local memory accesses. However, there is a limit to

[2]As a rule of thumb, a minimum traceback length of $4K$ to $5K$ is required [7]. Any deeper traceback lengths increases decoding delay and decoder memory requirements, while not significantly improving the error correcting capability.

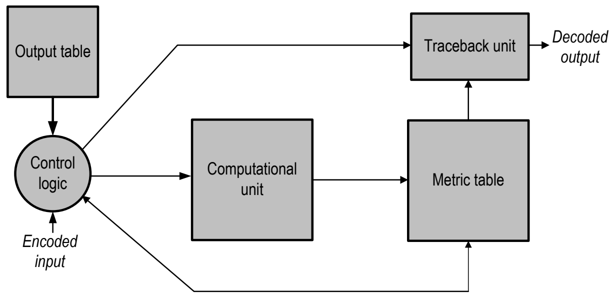

Fig. 4. Stand-alone Viterbi decoder architecture.

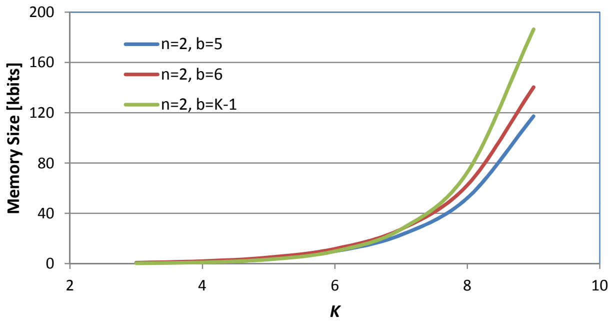

Fig. 5. Memory size for metric table.

the memory size that can be used, since the accelerator needs to be small enough to fit inside the datapath.

Two distance calculations and one ACS computation are required to calculate one new entry in the metric table. Thus, 2^K distance calculations and 2^{K-1} ACS computations are required for every received symbol, to calculate one metric table column. When one column of the metric table has been filled, the process continues for the remaining symbols.

III. Viterbi Decoding Acceleration

This work focuses on hardware-based Viterbi acceleration that can be integrated inside a processor datapath to achieve flexible, yet high-throughput decoding. An accelerator solution that is based on both software and hardware yields flexibility, however, the accelerator unit is bound to incur an area, power, and timing overhead that becomes visible when acceleration does not occur. The wider the processor application domain, the smaller the Viterbi accelerator has to be.

A stand-alone Viterbi decoder, with memory enough to store large data sets, requires huge area for reasonable constraint lengths. To explain the tradeoffs between, on the one hand, memory requirements and area, and, on the other hand, data throughput, we will first describe a basic stand-alone decoder implementation. Later, a lightweight accelerator, suitable for processor datapath integration, is presented.

A. Stand-Alone Decoder

A straightforward implementation of a stand-alone Viterbi decoder for an arbitrary constraint length K is shown in Fig. 4. The computational unit is made up of only one ACS unit and one distance calculator, of which the latter is used twice for every ACS operation. Using b to define the number of bits required for each metric table entry and n to define the number of output bits, Fig. 5 shows the metric memory size (in number of bits) for an increasing constraint length K. Stand-alone decoders for different constraint lengths were synthesized on a 65-nm cell library at a clock rate of 370 MHz[3], and verified for various test cases. Fig. 6 shows the area of the implementations for three different constraint lengths.

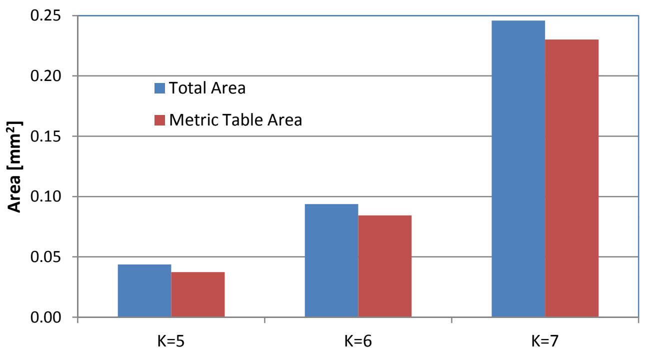

Fig. 6. Total area and metric table area for varying constraint lengths.

Considering the exponential increase of memory size with constraint length, unless K is very limited, a stand-alone decoder is not appropriate for tight integration into a datapath. As shown in Fig. 6, the major portion of the area is consumed by the metric table. Our focus is therefore on minimizing the size of the metric table.

B. Acceleration for Viterbi Decoding

To strike a good balance between flexibility and performance, a mixed software-hardware approach must be employed to make an accelerator amenable for datapath integration. In general, this means that the portions of the Viterbi code that constitute the majority of the execution time and memory accesses are accelerated. Other portions, which are not as frequently executed, are handled using the general-purpose features of the processor.

Minimizing the accelerator memory will significantly improve the trend of Fig. 5, however, the memory will still grow as constraint lengths are increased. Since memory is such a precious resource, the most efficient implementation is achieved when optimizing the Viterbi accelerator for a certain constraint length, but this is not a flexible approach.

A key feature of the scheme of this paper is that the accelerator should be able to provide significant performance improvements also for Viterbi applications that use constraint lengths that are larger than that of the accelerator. Assume, for example, an accelerator that is optimized for $K = 7$. This accelerator will be very efficient in handling applications such as DVB and GSM [1], since its memory resources are matching the applications. However, our goal is that the

[3]The choice of this operating frequency was based on predicted timing constraints in the datapath integration phase in Sec. IV.

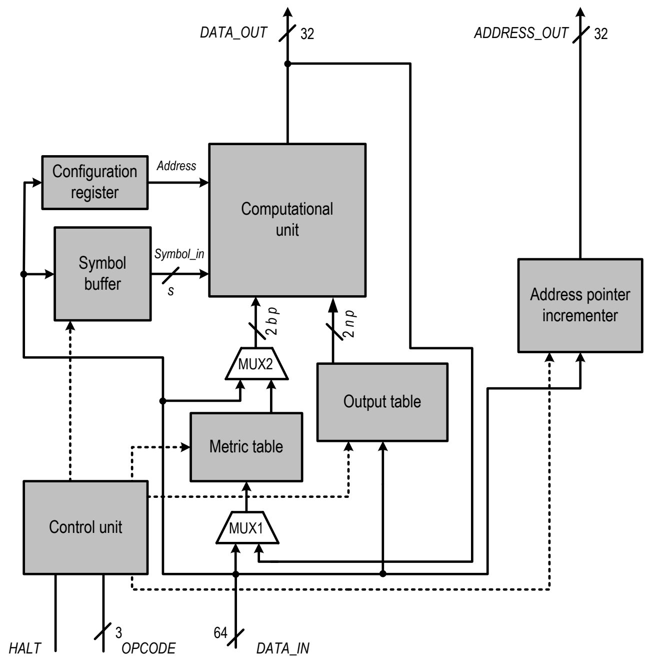

Fig. 7. Lightweight Viterbi accelerator.

accelerator of this specific example should also efficiently accelerate applications with $K > 7$.

C. Lightweight Datapath Accelerator

The first challenge in designing an accelerator is to identify and define an appropriate boundary between the hardware and the software. To this end, application profiles in conjunction with memory and computational requirements are analyzed (Sec. II-B) and the following conclusions are made:

- The branch metric and path metric calculations are computation intensive and often recurring steps.
- Initialization of the output table is done once.
- Traceback is required once, after the whole metric table is computed.
- Only the previous column of the metric table, along with the output table, is required to compute a new column.

We propose to perform only the branch and path metric calculations inside the accelerator, that is, the metric table computations are local. The computations of the output table and the traceback, however, are done in other parts of the datapath. This design decision is based on the fact that the output table is fairly static; its values depend upon the generator polynomials and are only computed once for a certain application. Thus, the output table is generated using the general-purpose portions of the datapath, in relatively few cycles, and subsequently loaded into the accelerator.

The computation-intensive branch metric calculations can be efficiently accelerated by specialized hardware. The metric table is therefore computed by the accelerator. To reduce the required memory space and memory bandwidth, only the entries of the last metric table column are kept in dedicated local memory in the accelerator as these are essential for computing the next column. The complete metric table is stored in the processor's main memory, where it is available for the traceback process.

The proposed accelerator is a compromise solution, in which we significantly decrease the memory size from a stand-alone version: We implement the metric table memory portion that ensures that a very high number of accesses are local, limiting communication power dissipation. As the size of these metric table buffers scales up with an increasing value of K, there will be a practical limit to the code complexity, that is, before the accelerator area starts to dominate the datapath. Sec. V will present quantitative data for a number of accelerators and their size in relation to a processor datapath.

Fig. 7 presents the new Viterbi decoder architecture, which in this configuration has four computational units. In this soft-decision Viterbi accelerator, each computation unit contains two Euclidian distance calculation units and one ACS unit to increase throughput. This accelerator is designed to support Viterbi decoding for literally any constraint length. When the application constraint length is less than or equal to the accelerator's K, the complete metric table computation can be done in hardware (Sec. III-C1). However, for cases when the software application's constraint length is greater than the accelerator's, a so-called *sub-state* mode is employed (Sec. III-C2).

1) Full Mode: When the accelerator memory is sufficient for the constraint length of the applications, the complete path and branch metric calculations are done by the accelerator. Fig. 8 shows the detailed sequence of steps for Viterbi decoding while using a hardware accelerator whose constraint length fits that of the application.

2) Sub-State Mode: When the accelerator memory is not sufficient for the constraint length of the applications, the accelerator operates on a subset of the states. The metric table value is provided directly from the datapath register file, so the accelerator approaches the behavior of execution units such as ALUs. Fig. 9 shows the detailed sequence of steps performed for the sub-state mode.

The gray boxes of Fig. 8 and Fig. 9 represent portions of the code that are performed in software, while transparent boxes represent portions that are accelerated by the accelerator. As far as execution data given in the figures, this is the topic of the next section.

IV. Evaluation Setup and Flow

Verifying and evaluating a software-hardware solution is a challenge. To make a comprehensive evaluation, statistics based on two different Viterbi benchmarks are provided. First, the EEMBC Viterbi benchmark [4] is used. EEMBC is a well-known benchmark suite, but it provides a limited evaluation capability as it is defined only for K = 6 and R = 1/2. Thus, to enable evaluations for other constraint lengths and code rates, a parameterizable Viterbi kernel was developed [8]. Both benchmarks employ soft-decision decoding.

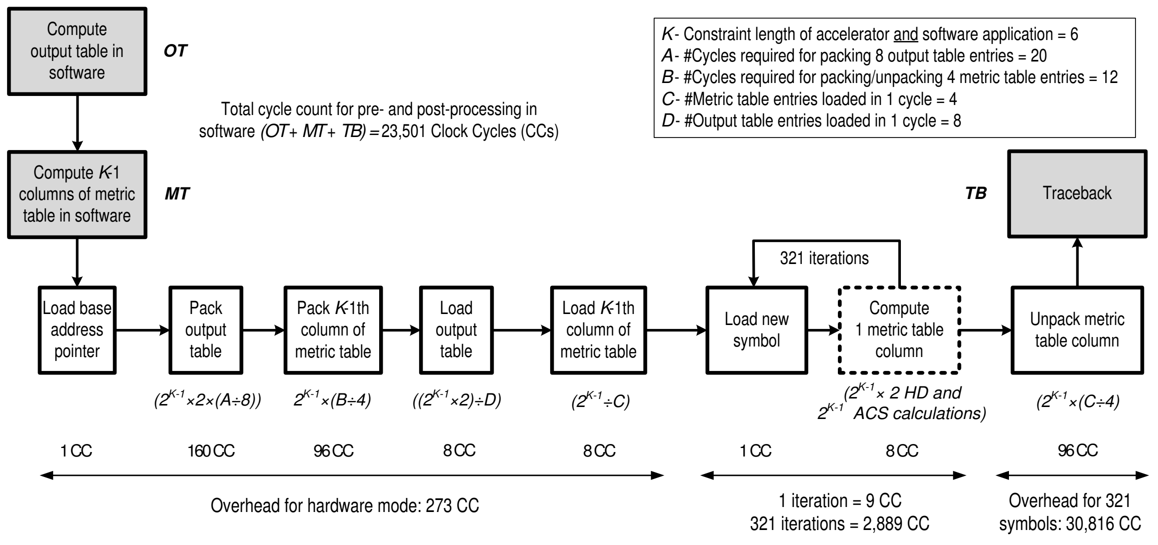
Compute output table in software
OT
Compute K-1 columns of metric table in software
MT
Total cycle count for pre- and post-processing in software (OT+ MT+ TB) = 23,501 Clock Cycles (CCs)
K- Constraint length of accelerator and software application = 6
A- #Cycles required for packing 8 output table entries = 20
B- #Cycles required for packing/unpacking 4 metric table entries = 12
C- #Metric table entries loaded in 1 cycle = 4
D- #Output table entries loaded in 1 cycle = 8
TB
Traceback
321 iterations
Load base address pointer
Pack output table
Pack K-1th column of metric table
Load output table
Load K-1th column of metric table
Load new symbol
Compute 1 metric table column
Unpack metric table column
(2^{K-1}×2×(A÷8))
2^{K-1}×(B÷4)
((2^{K-1}×2)÷D)
(2^{K-1}÷C)
(2^{K-1}× 2 HD and 2^{K-1} ACS calculations)
(2^{K-1}×(C÷4)
1 CC
160 CC
96 CC
8 CC
8 CC
1 CC
8 CC
96 CC
Overhead for hardware mode: 273 CC
1 iteration = 9 CC
321 iterations = 2,889 CC
Overhead for 321 symbols: 30,816 CC

Fig. 8. Sequence of steps while using a Viterbi accelerator with $K = 6$ in full mode. The cycle count values are based on decoding of the EEMBC Viterbi benchmark, which uses $K = 6$, a code rate R = 1/2 and 326 input symbols.

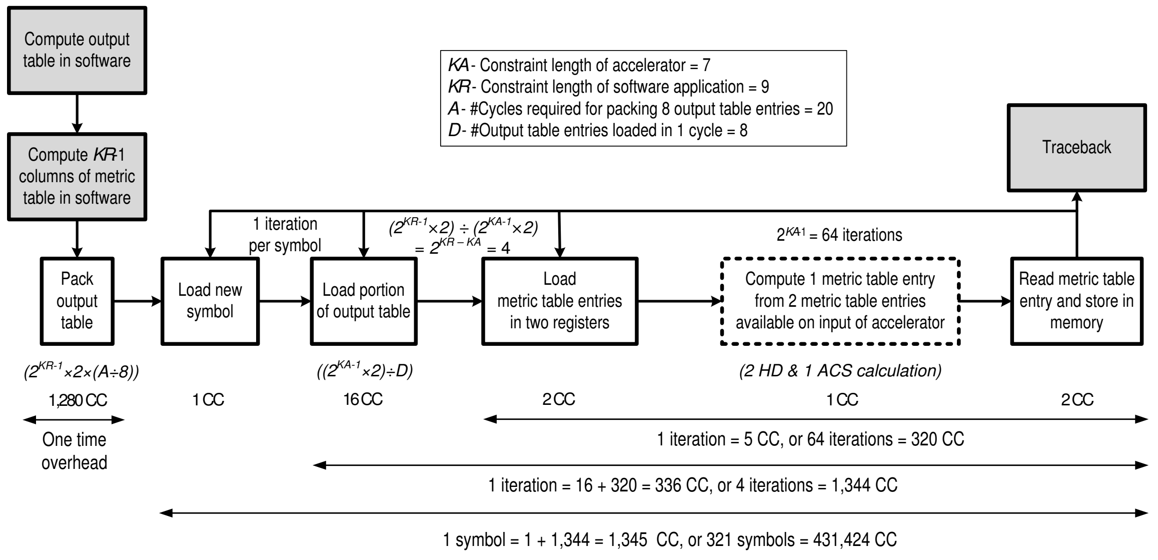
Compute output table in software
Compute KR-1 columns of metric table in software
KA- Constraint length of accelerator = 7
KR- Constraint length of software application = 9
A- #Cycles required for packing 8 output table entries = 20
D- #Output table entries loaded in 1 cycle = 8
Traceback
1 iteration per symbol
(2^{KR-1}×2) ÷ (2^{KA-1}×2) = 2^{KR−KA} = 4
2^{KA-1} = 64 iterations
Pack output table
Load new symbol
Load portion of output table
Load metric table entries in two registers
Compute 1 metric table entry from 2 metric table entries available on input of accelerator
Read metric table entry and store in memory
(2^{KR-1}×2×(A÷8))
((2^{KA-1}×2)÷D)
(2 HD & 1 ACS calculation)
1,280 CC
1 CC
16 CC
2 CC
1 CC
2 CC
One time overhead
1 iteration = 5 CC, or 64 iterations = 320 CC
1 iteration = 16 + 320 = 336 CC, or 4 iterations = 1,344 CC
1 symbol = 1 + 1,344 = 1,345 CC, or 321 symbols = 431,424 CC

Fig. 9. Sequence of steps while using a Viterbi accelerator with $KA = 7$ in sub-state mode. The cycle count values are based on decoding of a Viterbi kernel that uses $KR = 9$, a code rate R = 1/4 and 321 input symbols.

A. Evaluation Setup: Datapath

To study the processor enhancements made possible by the Viterbi accelerator in Sec. III-C, we use the FlexCore processor [3] as evaluation platform. Fig. 10 shows the baseline FlexCore processor datapath. This baseline datapath configuration consists of the units found in a five-stage single-issue pipeline, such as the MIPS R2000 datapath [9], that is, load/store (LS), register file (RF), arithmetic and logic unit (ALU), program counter (PC) and multiplier (MULT) units.

Conventional architectures have dedicated interconnects between datapath units as well as instructions that are hard-coded to make datapath operations hidden to the compiler. In contrast, in a FlexCore processor the datapath control signals—for execution units as well as interconnect—are fully exposed to the compiler. Ease of accelerator integration and efficient scheduling, by exploiting routing freedom, for identification of potential parallelism are among the advantages of this scheme. The FlexCore datapath interconnect is based on switchboxes, to essentially allow data being routed from one unit to any other unit. The datapath unit outputs are routed to a number of switchboxes, where each switchbox output is connected to the input of a certain unit. For the *full* interconnect configuration in Fig. 10, each of the ten switchboxes has nine inputs and requires four instruction-driven address bits[4]. By removing

[4]Assuming M output ports and N input ports, $N\lceil\log_2 M\rceil$ switchbox address bits are needed.

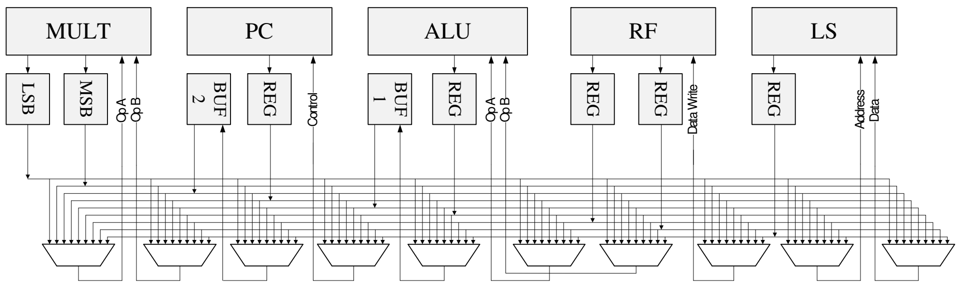

Fig. 10. Baseline FlexCore datapath.

switchbox inputs the full 90-link datapath interconnect can be reduced and customized to less area- and power-demanding interconnect configurations [10]. Throughout this paper we assume the full interconnect configuration, since it is easily scalable. To put this configuration in a perspective, a MIPS-based interconnect configuration [9] has 33 interconnect links.

The FlexCore datapath has its control bits fully exposed to the compiler: The datapath unit control field comprises signals for datapath units (that is, enable or opcode signals), while the interconnect control field includes address signals to the interconnect switchboxes.

B. Evaluation Setup: Accelerator Integration

The complete datapath integration of the Viterbi accelerator (which has its output registers embedded) is shown in Fig. 11. The input data to the Viterbi unit need to be routed from the RF and the LS units and, consequently, the accelerator unit has no need for a complete switchbox at its input. One 32-bit word from the RF unit is directly made available to the Viterbi unit. The 2-way multiplexer at the Viterbi unit input accepts one 32-bit word from the RF unit (In A) or the 32-bit word from the LS unit (In B). The interconnect control field is increased by one bit to control this multiplexer. Similarly, the output of the accelerator, whose data are intended to either be stored directly in memory or in register file, is only required to be routed to the RF and the LS units. Thus, the input switchboxes for the RF and the LS units need to accept one more data input. When adding the accelerator to the baseline configuration in Fig. 10, there is no need to extend the number of switchbox address signals, since the four bits can support 16 multiplexer inputs. However, if a reduced interconnect configuration is used, a suitable number of switchbox addresses can be ascertained after performing design exploration [11] on the application domain and its need for interconnect links. Finally, the datapath unit control field needs to be extended by three control bits, which serve as opcode for the accelerator.

C. Evaluation Flow

The RTL code of the Viterbi accelerator unit was first developed and comprehensibly verified, after which this block was made available to the configurable FlexCore VHDL generator (FlexGen). The FlexCore generator has access to RTL code for several datapath units and can generate an instance of the FlexCore processor based on different datapath configurations. Other tools at our disposal include a compiler (FlexComp) to schedule applications to a register transfer notation (RTN) code used for the processor, and a simulator (FlexSim) to perform cycle-accurate simulation and profiling of MIPS assembly and RTN code. The full evaluation flow for gathering information on execution time, post-layout timing, power, and energy dissipation is shown in Fig. 12 [11].

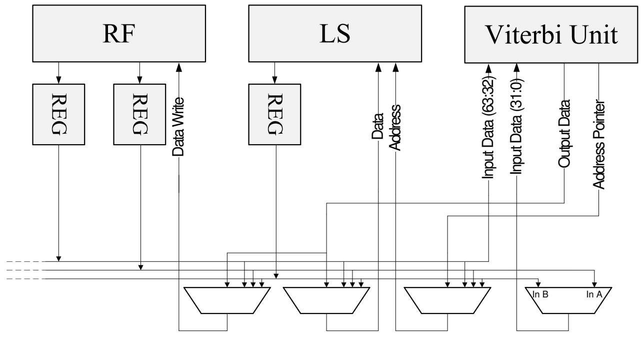

Fig. 11. Integration of the Viterbi accelerator.

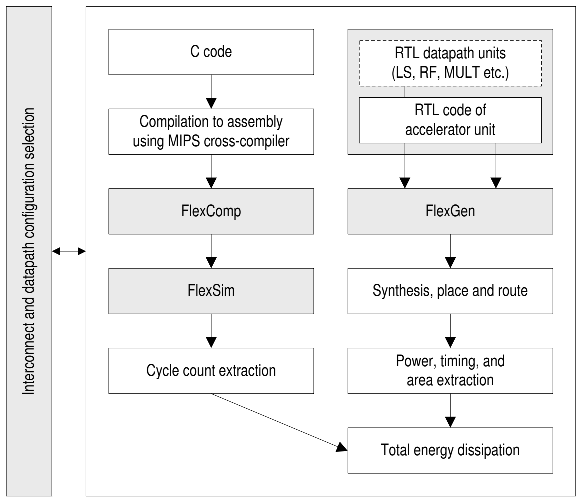
Interconnect and datapath configuration selection
C code
Compilation to assembly using MIPS cross-compiler
FlexComp
FlexSim
Cycle count extraction
RTL datapath units (LS, RF, MULT etc.)
RTL code of accelerator unit
FlexGen
Synthesis, place and route
Power, timing, and area extraction
Total energy dissipation

Fig. 12. Evaluation flow.

V. Evaluation Results

The following section will present results obtained from running the different benchmarks using different accelerators in different modes.

A. Evaluation for Full Mode

Our first evaluation is based on the EEMBC Viterbi benchmark [4], which is a soft-decision algorithm with K = 6, R = 1/2 and 326 input symbols. Fig. 8 shows not only the sequence of execution, but also the execution times in clock cycles (CC), with emphasis on the computation-intensive kernels of the benchmark. Also, the figure details the overhead associated with the accelerator, for packing/loading data into the accelerator as well as unpacking data for traceback.

Table I shows the evaluation results for the EEMBC Viterbi benchmark, based on post-layout implementations in a 1.2-V 65-nm LP-SVT cell-based flow, with a target clock rate of 370 MHz (2.7-ns timing constraint). The energy dissipation is for the whole benchmark, that is, #Cycles total $\times$ Power.

In integrating the Viterbi accelerator, the datapath has become 20% larger, which directly impacts power dissipation (a 38% increase). However, thanks to the improved performance, the reduced execution time (90% for the total benchmark or 99% for the benchmark kernel) yields an 87% energy reduction for the whole benchmark.

TABLE I
Evaluation Using EEMBC Benchmark

Architecture	#Cycles total	#Cycles kernel	Area (μm^2)	Power (mW)	Energy (μJ)
Software-only	275,076	251,575	53,737	8.97	6.67
Accelerated	26,614	3,113	64,209	12.4	0.89

We can also consider a case when we have integrated an accelerator with a larger constraint length than necessary. Assume an accelerator that supports $K = 7$. This accelerator has the same number of computational units as that for $K = 6$ and, thus, the performance gain will be the same. However, the datapath area increases to 70,370 μm^2, the power to 15.3 mW and the energy to 1.10 μJ. The constraint length overkill impacts the area and the power, but still the energy reduction is high, at 84%.

B. Evaluation for Sub-State Mode

The evaluation of the sub-state mode requires a different Viterbi benchmark than the one in the EEMBC suite. Thus, a new benchmark kernel, emphasizing the branch metric and the path metric calculations, was developed [8]. The basic block in this new benchmark computes one new metric table entry, that is, it performs one ACS and two Euclidian distance computations. This exactly models one computational cycle in the hardware accelerator, while in sub-state mode. The flow is shown in detail in Fig. 9, together with performance numbers for this mode. For the computation of the Viterbi kernel for 321 symbols, the software-only solution requires 9,614,592 cycles, while the accelerator in sub-state mode requires only 431,424 cycles.

A cumulative comparison of cycle counts for the computation of the new benchmark with and without accelerator is also presented for two different cases of benchmark constraint lengths KR (8 and 9) and for two accelerators with constraint lengths KA of 6 and 7 (Table II).

TABLE II
Evaluation Using Viterbi Kernel Benchmark

Accelerator constraint length KA	Required constraint length KR	#Cycles software	#Cycles hardware	Performance gain (%)
7	9	9,614,592	431,424	95
6	9	9,614,592	431,424	95
7	8	4,807,296	216,033	95
6	8	4,807,296	216,033	95

It is important to note that the performance gain for the sub-state mode is higher than for the full mode. This is to a large extent explained by the benchmark used, that is, the Viterbi kernel. In an evaluation of the whole decoding operation, the performance gain would reduce. Also, we have assumed constant latency for memory accesses. This assumption is not completely true in a real application scenario.

Moreover, the number of executed cycles does not depend on the constraint length of the accelerator. The reason is that it is only the accelerator's internal memory that is changed with KA; the computational unit of the accelerator is intact.

C. Evaluation of Throughput

The evaluations performed above focus on execution time and energy dissipation. For an embedded processor for convolutional decoding the possible data throughput is a critical parameter. Thus, Table III presents the throughput achieved for several combinations of KA and KR.

TABLE III
Evaluation of Throughput

KA	KR	Mode	Throughput (Mbit/s)
7	9	Sub-State	0.27
7	8	Sub-State	0.55
7	7	Full	1.84
7	6	Full	3.52
6	9	Sub-State	0.27
6	8	Sub-State	0.55
6	7	Sub-State	1.10
6	6	Full	3.52

VI. Related Work

There are several examples of decoder circuits that are more or less customized to the decoder algorithm. An implementation can be tailored to different configurations of, for example, decoding scheme, constraint length and code rate [12]. With special considerations at the circuit level, highly optimized fixed-function implementations can be obtained [13]. The circuitry can also include some rudimentary flexibility that allows for reconfiguration between a few operating modes, such as different code rates [14]. Thanks to circuit innovations, despite the 90-nm Viterbi implementation supports several constraint lengths (up to $K = 9$), it reaches a top performance of almost 2 Gb/s for $K = 6$ [15].

As there are algorithmic similarities between Viterbi and Turbo decoding, implementations that can operate on either convolutional or Turbo codes have been extensively researched in the past. Besides support for a single Turbo mode, there is, for example, one ASIC implementation that supports four different Viterbi modes [16], another ASIC implementation that supports two Viterbi modes [17], and an FPGA implementation that supports a wide range of Viterbi modes [18].

Programmability via the instruction support of a processor offers the highest degree of flexibility. A programmable FEC decoder can be implemented as a memory-mapped coprocessor, such as the loosely coupled Viterbi decoder by Hocevar and Gatherer which supports constraint lengths from 5 to 9 and code rates of 1/2, 1/3, and 1/4 [19]. Application-specific instruction-set processors (ASIPs) offer an alternate implementation route: Based on an array of processing elements, the RECFEC architecture [20] is versatile in that it handles several decoding schemes, including Viterbi, Turbo and LDPC. The FlexiTreP ASIP comprises a 15-stage pipeline that has been optimized for Turbo and convolutional codes [21]. In the context of a SIMD processing elements in an MPSoC architecture, Kunchamwar et al. demonstrate hardware accelerators that are common to different FEC kernels (Viterbi and Turbo) [22]. Research also has been carried out on FEC ASIPs that can handle both Viterbi and LDPC codes: As an extension to FlexiTreP above, FlexiChaP was introduced to also handle LDPC [23]. Kunze et al. propose another ASIP, in which one and the same functional unit handles Viterbi and LDPC [24].

Most accelerators listed above are stand-alone coprocessors, in a system-on-chip configuration. The accelerator we present is, relatively speaking, a lightweight accelerator for datapath integration efficiently providing a hybrid approach for Viterbi decoding, using both software and hardware. Since the computational intensive kernel is accelerated under strict software control, our approach provides a high degree of configurability.

VII. Conclusion

A versatile datapath accelerator for forward error correction based on Viterbi decoding has been provided. Innovative software/hardware codesign concepts, such as micro-coding and compiler control, were used to minimize memory sizes. The EEMBC Viterbi benchmark was applied to a 2.7-ns 65-nm processor datapath with and without a Viterbi accelerator; a performance increase of 90% is achieved with a datapath area overhead of 20%.

References

[1] G. Krishnaiah, N. Engin, and S. Sawitzki, "Scalable Reconfigurable Channel Decoder Architecture for Future Wireless Handsets," in *Proc. Design, Automation Test in Europe Conf.*, Apr. 2007, pp. 1–6.

[2] R. Johannesson and K. S. Zigangirov, *Fundamentals of Convolutional Coding*. Wiley-IEEE Press, 1999.

[3] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," *J. Signal Processing Systems*, vol. 57, no. 1, pp. 5–19, 2009.

[4] Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org

[5] O. O. Khalifa, T. Al-Maznaee, M. Munjid, and A.-H. A. Hashim, "Convolution Coder Software Implementation Using Viterbi Decoding Algorithm," *J. Computer Science*, vol. 4, no. 10, pp. 847–856, 2008.

[6] G. D. Forney, Jr., "The Viterbi Algorithm," *Proc. of the IEEE*, vol. 61, no. 3, pp. 268–278, Mar. 1973.

[7] J. Heller and I. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Trans. Communication Technology*, vol. 19, no. 5, pp. 835–848, Oct. 1971.

[8] Viterbi Branch Metric Kernel. [Online]. Available: http://www.flexsoc.org/download/

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1998.

[10] M. Själander, P. Larsson-Edefors, and M. Björk, "A Flexible Datapath Interconnect for Embedded Applications," in *Proc. IEEE Computer Society Annual Symp. on VLSI*, May 2007, pp. 15–20.

[11] T. T. Hoang, U. Jälmbrant, E. der Hagopian, K. P. Subramaniyan, M. Själander, and P. Larsson-Edefors, "Design Space Exploration for an Embedded Processor with Flexible Datapath Interconnect," in *Proc. IEEE Int. Conf. on Application-Specific Systems Architectures and Processors*, Jul. 2010, pp. 55–62.

[12] T. Gemmeke, M. Gansen, and T. G. Noll, "Implementation of Scalable Power and Area Efficient High-Throughput Viterbi Decoders," *IEEE J. Solid-State Circuits*, vol. 37, no. 7, pp. 941–948, Jul. 2002.

[13] M. Kawokgy and C. A. T. Salama, "A Low-Power CSCD Asynchronous Viterbi Decoder for Wireless Applications," in *Proc. Int. Symp. Low Power Electronics and Design*, 2007, pp. 363–366.

[14] M. Kamuf, V. Öwall, and J. B. Anderson, "Optimization and Implementation of a Viterbi Decoder Under Flexibility Constraints," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 55, no. 8, pp. 2411–2422, Sep. 2008.

[15] M. A. Anders, S. K. Mathew, S. K. Hsu, R. K. Krishnamurthy, and S. Borkar, "A 1.9 Gb/s 358 mW 16-256 State Reconfigurable Viterbi Accelerator in 90 nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 214–222, Jan. 2008.

[16] C.-C. Lin, Y.-H. Shih, H.-C. Chang, and C.-Y. Lee, "A Low Power Turbo/Viterbi Decoder for 3GPP2 Applications," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 4, pp. 426–430, Apr. 2006.

[17] M. A. Bickerstaff *et al.*, "A Unified Turbo/Viterbi Channel Decoder for 3GPP Mobile Wireless in 0.18-μm CMOS," *IEEE J. Solid-State Circuits*, vol. 37, no. 11, pp. 1555–1564, Nov. 2002.

[18] J. R. Cavallaro and M. Vaya, "Viturbo: A Reconfigurable Architecture for Viterbi and Turbo Decoding," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, vol. 2, Apr. 2003, pp. 497–500.

[19] D. E. Hocevar and A. Gatherer, "Achieving Flexibility in a Viterbi Decoder DSP Coprocessor," in *Proc. 52nd IEEE Vehicular Technology Conf.*, vol. 5, 2000, pp. 2257–2264, vol.5.

[20] A. Niktash, H. Parizi, and N. Bagherzadeh, "A Reconfigurable Processor for Forward Error Correction," in *Proc. Int. Conf. on Architecture of Computing Systems*, 2007, pp. 1–13.

[21] T. Vogt and N. Wehn, "A Reconfigurable ASIP for Convolutional and Turbo Decoding in an SDR Environment," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 10, pp. 1309–1320, Oct. 2008.

[22] M. K. Kunchamwar, D. P. Prasad, P. Hegde, P. T. Balsara, and R. Sangireddy, "Application Specific Instruction Accelerator for Multistandard Viterbi and Turbo Decoding," in *Proc. Int. Conf. Parallel Processing Workshop*, Sep. 2010, pp. 34–43.

[23] M. Alles, T. Vogt, and N. Wehn, "FlexiChaP: A Reconfigurable ASIP for Convolutional, Turbo, and LDPC Code Decoding," in *Proc. 5th Int. Symp. Turbo Codes and Related Topics*, Sep. 2008, pp. 84–89.

[24] S. Kunze, E. Matus, and G. P. Fettweis, "ASIP Decoder Architecture for Convolutional and LDPC Codes," in *Proc. IEEE Int. Symp. Circuits and Systems*, May 2009, pp. 2457–2460.