A NEW VITERBI DECODER DESIGN FOR CODE RATE K/N

Hsiang-ling Li and Chaitali Chakrabarti

Department of Electrical Engineering
Telecommunications Research Center
Arizona State University, Tempe, AZ 85287-5706, USA
li@dspsun.eas.asu.edu and chaitali@asu.edu

ABSTRACT

A novel VLSI architecture is proposed for implementing a long constraint length Viterbi Decoder (VD) for code rate k/n. This architecture is based on the encoding structure where k input bits are shifted into k shift registers in each cycle. The architecture is designed in a hierarchical manner by breaking the system into several levels and designing each level independently. At each level, the number of computation units, the interconnection between the units as well as allocation and scheduling issues have been determined. In-place storage of accumulated path metrics and trace back implementation of the survivor memory have also been addressed. The resulting architecture is regular, flexible and achieves a better than linear tradeoff between hardware complexity and computation time.

1. INTRODUCTION

The maximum likelihood decoding of convolutional and trellis codes based on the Viterbi algorithm is an important problem in digital communication. Most of the existing implementations of the Viterbi algorithm are for code rate $R = 1/n$. The trellis diagram for this case has a simple and symmetric structure, and is amenable to implementation by one or more processing units. For code rate $R = k/n$, the encoder has k input bits connected to k shift registers. When the sizes of the shift registers differ by at most one bit, the encoder can be modelled as a single radix-2^k shift register [1]-[3]. A scalable architecture has been developed for this case in [3]. In general, the sizes of the shift registers may differ by more than one bit. Then the design of the Viterbi decoder (VD) should be based on the original k shift register structure. The trellis diagram now no longer has the simple structure of a R=$1/n$ encoder. A systolic architecture has been developed for this case in [4], where each processor is assigned only one state. The non-scalability of this architecture and the fact that it is less efficient when the shift registers are not of equal size, make this architecture unattractive.

In this paper we present a scalable architecture for the case when the sizes of the shift registers may differ by more than one bit. The architecture is designed in a hierarchical manner by breaking the system into several levels, and by designing each level independently. The tasks in the design of each level range from determining the number of computation units and the interconnection between the units to the allocation and scheduling of operations. Additional design issues such as in-place storage of the accumulated path metrics (APMs) and the trace back implementation of the survivor memory have also been addressed. The resulting architecture is regular, has a foldable global topology and is very flexible. It also achieves a better than linear tradeoff between hardware complexity and computation time.

2. OVERVIEW OF THE WHOLE SYSTEM

Let l_i be the length of the ith shift register, $1 \leq i \leq k$, and let the input data be binary. The state label in the trellis diagram is a string of bits obtained by connecting each shift register in series. The label can be divided into k blocks, with each block corresponding to a shift register. The input to every block is connected to the LSB of the shift register. The trellis diagram resulting from this topology can be described by the relation,

$$\bar{0}b_k^{l_k-1},\ldots,\bar{0}b_1^{l_1-1} \rightarrow b_k^{l_k-1}\bar{0},\ldots,b_1^{l_1-1}\bar{0}.$$

Here $\bar{0}$ denotes the 1-bit binary combinations and $b_i^{l_i-1}$ represents the fixed portion of the ith block $b_{i,l_i-1}\ldots b_{i,1}$. The comma "," is employed as a separation between the blocks. Thus every state is connected to 2^k states in the trellis diagram.

We aggregate all those states having the same bits in the kth block into one single node called the *supernode*. Then, the trellis diagram reduces to the diagram for code rate $1/n$. Since the connection pattern between the supernodes is the same at every stage of the trellis diagram, we refer to this communication pattern as *fixed*. Every supernode then needs to communicate with two parent and two descendant supernodes at every stage. The communication pattern between supernodes can also vary from stage to stage with a period of l_k. This is accomplished by shifting circularly the bits of the kth block by one bit at each trellis stage. We refer to such a communication pattern as *dynamic*. This results in in-place updating of all the states within a supernode. As a result, every supernode now communicates with only one other supernode at each stage of the trellis diagram. The interconnections between the states inside the supernodes are the same for both connection patterns. Figure 1 describes the global topology of the proposed system. The routing network supports either fixed or dynamic communication pattern.

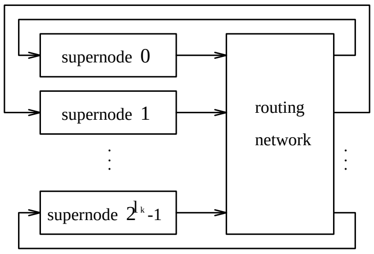

Figure 1: The architecture model for the whole system.

3. INTERNAL TOPOLOGY OF THE SUPERNODES

Processor Allocation:

The connectivity between the states in the supernodes can be described by $\bar{0}b_m^{l_m-1},\ldots,\bar{0}b_1^{l_1-1} \to b_m^{l_m-1}\bar{0},\ldots,b_1^{l_1-1}\bar{0}$, where $m=k-1$. We group those states whose state labels have the same (l_i-j_i) MSBs in the ith block, and assign an ACS (Add-Compare-Select) unit to this group of states, $0 \le j_i \le l_i-1$ and $1 \le i \le m$. Thus, every ACS unit updates the APMs of $2^{j_m+\cdots+j_1}$ states. The ACS unit is labeled as $ACS(b_m^{l'_m},\ldots,b_1^{l'_1})$, where $l'_i = l_i - j_i$, and $b_i^{l'_i}$ denotes $b_{i,l_i}\ldots b_{i,(j_i+1)}$. $j_1,\ldots,j_m$ are determined by the design constraints of the VD.

Consider the following relation between the ACS units,

$$ACS(\bar{0}b_m^{l'_m-1},\ldots,\bar{0}b_1^{l'_1-1}) \to ACS(b_m^{l'_m-1}\bar{0},\ldots,b_1^{l'_1-1}\bar{0}). \quad (1)$$

Every ACS unit has to communicate with 2^k other ACS units equally distributed between two destination supernodes (2^{k-1} ACS units in each supernode). At the nth stage (t=nT), each state receives 2^k APMs, one from each parent ACS unit. The 2^k parent ACS units are equally distributed between two parent supernodes. The updated APM at the nth stage is sent to 2^k descendant states at the $(n+1)$th stage. The 2^k descendant states are equally distributed between two descendant ACS units in two descendant supernodes. For dynamic communications, every ACS unit communicates only with 2^m ACS units in a parent supernode and 2^m ACS units in a descendant supernode. Each ACS unit has a memory to store the APMs transferred from the parent ACS units.

Processor Clustering and Interconnections:

Since the interconnection between the ACS units is quite costly, we propose a second aggregation phase similar to [3] in order to increase the overall efficiency of the communication links to 100%. Define the set of ACS units on the left-hand side of the eqn (1) as the departure equivalence ACS units or DEACS. The DEACS label generated from the ACS units, $ACS(\bar{0}b_m^{l'_m-1},\ldots,\bar{0}b_1^{l'_1-1})$ is $DEACS(b_m^{l'_m-1},\ldots,b_1^{l'_1-1})$. The number of ACS units included in a DEACS is 2^m. Every DEACS has 2^m global links with one descendant supernode, and 2^m global links with one parent supernode. A switching network is associated with each DEACS to route the APMs such that no conflict occurs.

Example: Let $k=3$, $l_1=5, l_2=5, l_3=2, j_1=3$, and $j_2=2$. Consider the DEACS labeled $DEACS(01,0)$ in one of 4 supernodes. We index an ACS unit by the following rule $ACS(b_m^{l'_m},\ldots,b_1^{l'_1}) \longrightarrow ACS_i$, where $i=dec(b_{m,l_m}\ldots b_{2,l_2}b_{1,l_1})$. $dec(b_m\ldots b_2b_1)$ is defined as the decimal value of the binary bits $b_m\ldots b_2b_1$ and i ranges from 0 through 2^m-1. The four ACS units in $DEACS(01,0)$ are indexed as follows:
(i) $ACS(001,00) \to ACS_0$, (ii) $ACS(001,10) \to ACS_1$, (iii) $ACS(101,00) \to ACS_2$, and (iv) $ACS(101,10) \to ACS_3$. This communication pattern is illustrated in Figure 2. □

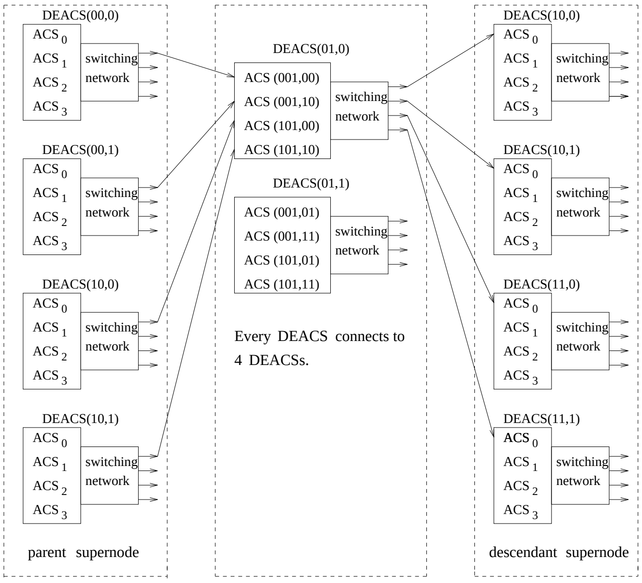

Figure 2: The clustering and connectivity between ACS units for the example where $m=2$.

So far we have described the following four levels in the hierarchical design of the VD.
(i) The highest level is the supernode level with 2^{l_k} supernodes. (ii) The second level is the DEACS level with

$2^{l'_m+\cdots+l'_1}/2^m$ DEACSs in each supernode. (iii) The third level is the ACS unit level with 2^m ACS units inside a DEACS. (iv) The lowest level is the state level. $2^{j_m+\cdots+j_1}$ states are updated by an ACS unit, and $2^{l_m+\cdots+l_1}$ states are aggregated into one supernode.

Processor Scheduling: Consider the ACS units in $DEACS(b_m^{l'_m-1},\ldots,b_1^{l'_1-1})$. Let every ACS unit labeled $ACS(b_{m,l_m}b_m^{l'_m-1},\ldots,b_{1,l_1}b_1^{l'_1-1})$ (out of 2^m ACS units) communicate with one distinct descendant ACS unit during a time interval of $T/2^m$. The descendant ACS unit to be communicated with at the nth time interval is $ACS(b_m^{l'_m-1}b_{m,j_m},\ldots,b_1^{l'_1-1}b_{1,j_1})$, where $dec(b_{m,j_m}\ldots b_{1,j_1}) = [dec(b_{m,l_m}\ldots b_{1,l_1}) + n - 1] \bmod 2^m$ and $n = 1, 2, \ldots, 2^m$.

In one time interval, every ACS unit updates sequentially the APMs of $2^{j_m+\cdots+j_1}/2^m$ states according to the increasing order of the number, $dec(b_m^{j_m-1}\ldots b_2^{j_2-1}b_1^{j_1-1})$, where $b_i^{j_i-1}$ denotes $b_{i,(j_i-1)}\ldots b_{i,1}$. The permutations demanded by this scheduling are structured such that they can be easily implemented using registers and some control logic, and do not require general-purpose routing networks.

4. MEMORY MANAGEMENT

In a VD, memory is required to store the APMs (that are distributed among the ACS units) and the survivor sequences.

In-place Storage of the Accumulated Path Metrics:

To update the APMs of the states, an ACS unit has to read a set of parent APMs from its local memory in a specified order dictated by the update schedule. At the same time, this ACS unit receives from parent ACS units a set of newly updated parent APMs for the next cycle that are stored sequentially into the original positions occupied by the old APMs. For the case when $j_1 \geq 2$ and $j_i = 2$ for $i = 2, \ldots, m$, the local memory of an ACS unit has $2^{j_m+\ldots+j_1} = 2^{2(m-1)+j_1}$ memory cells and the address of each cell consists of $2m + j_1 - 2$ bits. Each cell stores 2 parents APMs which are fetched sequentially. The first m address bits represent the block (out of 2^m blocks) that this cell belongs to, and the remaining bits index the cell in the block. The address generator needs to support only two kinds of memory access patterns. The addresses of a parent APM in the two access patterns are related by $A^mB^mC^{j_1-2} \longrightarrow B^mA^mC^{j_1-2}$, where A^m or B^m represents a block of m bits and C^{j_1-2} represents a block of $j_1 - 2$ bits. The structure of the data buffer unit for this case is shown in Figure 3. It consists of 2 blocks with 2^{j_1-2} memory banks in each block. The unit is pipelined in order that the ACS computations for one interval can be overlapped with data fetch for computations of the next interval. 2^{m-1} iterations are required to finish updating all the states in an interval. In each iteration, the 2^{j_1-2} banks in block 1 are accessed one after the other, each access lasting 2 time steps. Assume that the ACS unit, $ACS(b_{m,l_m}b_m^{l'_m-1},\ldots,b_{1,l_1}b_1^{l'_1-1})$ is updating the states in the ith interval. Here a_i is equal to $[dec(b_{m,l_m}\ldots b_{1,l_1})+i]$ mod $2^m * 2^{j_1-2}$, $i = 1\ldots m$. The parent APMs for the computations of the $(i+1)$th interval are loaded in cyclic mode; first into the 0th position of each of the 2^{j_1-2} banks in block 2, then into the 1st position of each of the banks, and so on.

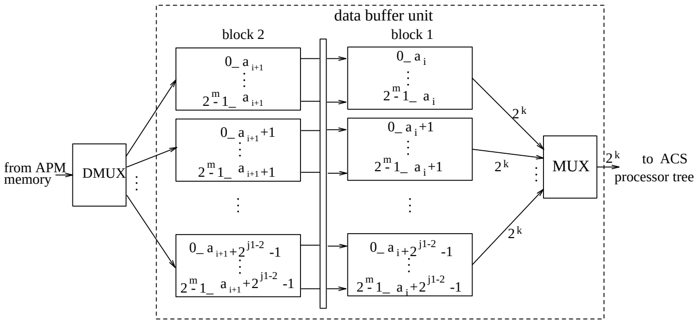

Figure 3: The memory scheme for the case when $j_1 \geq 2$ and $j_i = 2$ for $2 \leq i \leq m$.

For other sets of j_is values, the memory accesses are not as simple and the in-place storage cannot always be utilized. This is not really restrictive, since the values of j_is are chosen by the designer. In fact, the performance of the VD is affected by the value of $j_m + \cdots + j_1$, and not on the individual values of j_is. Consider an example where $k = 4$, and the lengths of the shift registers are 4, 3, 3, and 2. Let $j_1 + j_2 + j_3 = 7$ from performance considerations. We choose $l_4 = 2$, $l_1 = 4$, $l_2 = 3$, $l_3 = 3$ such that $j_1 = 3$, $j_2 = 2$ and $j_3 = 2$ can be chosen for efficient APM memory design.

Survivor Memory Management

The trace-back technique is a more efficient approach for reconstructing the survivor sequence. In our design of the VD, the time required to complete one global cycle is $2^{j_m+\cdots+j_1}$ clock cycles, which is usually long enough to perform the trace back operation. Hence, the throughput is not actually limited by the decoding rate. The classical one or K-pointer trace-back technique can be easily employed here, the specific choice depending on the value of $j_m + \cdots + j_1$. The structure of the survivor memory used is similar to the one proposed in [5]. Assume that the size of the RAM is $N \times S$, where N is equal to the number of states and S is a function of the survivor merging length, L. Every column in the RAM corresponds to a single column in the trellis diagram. Each memory cell stores two values, a decision pointer (DP) and the corresponding branch label (k input bits in every cycle). The DP consists of the k MSBs of every block in a state label. Some additional logic circuits are required to generate the address of the DP of the previous state label. There are three basic operations performed in the trace-back technique, namely trace-back, read-out and write, which require T, R, and W time units, respectively. The RAM implementation can be completely described by the following two rules:

Rule 1: After the collision of a read pointer and the write pointer, the read pointer first traces back L columns and then reads out D branch labels, while tracing back D columns. At the end of read-out, it collides with the write pointer again and repeats the above procedure. The write pointer writes new data into D columns after colliding with a read pointer. Thus, $S = L + D + K * D$.

Rule 2: Between successive collisions of a read pointer and the write pointer, the write pointer writes out $K \times D$ columns. Thus $L * T + D * R = K * D * W$.

A simple two stack LIFO structure with each stack $D * k$ in depth is required to perform the bit order reversal and equalize the latencies of all decoded bits [6]. The overall latency of the K-pointer trace-back technique with the two-stack structure is $(K + m + 2) * D * W$, assuming L=mD.

5. TRADEOFFS BETWEEN AREA AND COMPUTATION TIME

Let the propagation delay of an ACS unit be T_d and the data path be pipelined to N levels. If the delay of each pipelining latch is t_l, then the delay per stage of the pipelined ACS unit becomes, $\frac{T_d + N * t_l}{N} = T_c$, *and* $T_c < T_d$. T_c is set equal to the clock period (i.e. one time step). Compare the proposed VD with the purely state-parallel VD (where one ACS unit is assigned to every state). The global cycle time of our VD is increased by a factor of $\frac{(2^{j_m + \cdots + j_1} + N) * T_c}{T_d} = (1 + \frac{2^{j_m + \cdots + j_1}}{N}) * \frac{1}{1 - t_l/T_c}$ while the hardware complexity is reduced by a factor of $2^{j_m + \cdots + j_1}$. The tradeoff between the hardware complexity and the execution time is then given by $(1 + \frac{2^{j_m + \cdots + j_1}}{N}) * \frac{1}{1 - t_l/T_c} * \frac{1}{2^{j_m + \cdots + j_1}}$, which is approximately equal to $\frac{1}{N} * \frac{1}{1 - t_l/T_c} \approx \frac{T_c}{T_d} < 1$. Since the number of global communication links is also reduced by a factor of $2^{j_m + \cdots + j_1}$, we conclude that the proposed architecture has a better than linear tradeoff between complexity and speed. The factor $2^{j_m + \cdots + j_1}$ is determined by the value of $j_m + \cdots + j_1$, and so the individual values of $j_m, \ldots, j_1$ cannot directly affect the performance of the VD. Hence, we are able to choose suitable values for $j_m, \ldots, j_1$ to obtain different ACS unit allocations or efficient APM memory access, and still maintain the same performance improvement. This makes the proposed architecture highly flexible.

6. FOLDING THE GLOBAL TOPOLOGY

In the analysis so far, the number of supernodes is 2^{l_k} since the kth block has been chosen at the highest level. While any of the blocks could have been chosen at the highest level, if the area constraints demanded 2^l supernodes, where $l < l_i \; \forall i$, then the global topology would have to be folded. Let a *hypernode* be defined as a collection of supernodes which have the same $l' = l_k - j_k$ MSBs. Thus each hypernode consists of 2^{j_k} supernodes and the global topology consists of $2^{l'}$ hypernodes. All the hypernodes are processed in parallel and each hypernode sequentially processes the supernodes assigned to it. The scheduling and interconnections between the ACS units in a hypernode follow the same rules as discussed in Section 3.

7. CONCLUSIONS

In this paper we present a novel architecture to implement long constraint length Viterbi decoder for code rate k/n for the case when k bits are input to k shift registers. The architecture has been designed in a hierarchical fashion by breaking the system into several levels, and designing each level independently. The resulting architecture is very regular, achieves better than linear tradeoff between hardware complexity and computation time. Moreover it supports folding of the global topology without affecting the design of the lower levels. Another notable feature of this architecture is its flexibility. Different architectures can be obtained for different parameter choices (j_is), and yet all achieve the same performance improvement as long as the value of $(j_k + \ldots + j_1)$ remains fixed.

8. REFERENCES

[1] H.D.Lin and C.B.Shung, "General in-place scheduling for the Viterbi Algorithm," in *Proc. ICASSP* 1991, pp.1577-1580

[2] P.J.Black and T.H.-Y.Meng, "A unified approach to the Viterbi algorithm state metric update for shift register process," in *Proc. ICASSP*, vol.5, 1992, pp.629-632.

[3] F.Daneshgaran, *VLSI Architectures for Parallel Implementation of Long Constraint Length Viterbi Decoders.* Ph.D. thesis, Univ. of California, Los Angeles, 1992.

[4] C.Y.Chang and K.Yao, "Systolic Array Processing of the Viterbi Algorithm," *IEEE Trans. on Information Theory*, pp.76-86, Jan 1989.

[5] R.Cypher and C.B.Shung, "Generalized Trace-back Techniques for Survivor Memory Management in the Viterbi Algorithm," *Journal of VLSI Signal Processing*, vol.5, pp.85-94, 1993.

[6] G.Feygin and P.G.Gulak, "Architectural tradeoffs for survivor sequence memory management in Viterbi decoders," IEEE Trans. Commun., pp.425-429, Mar. 1993.