Design of a High-Throughput Low-Power IS95 Viterbi Decoder

Xun Liu Marios C. Papaefthymiou
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, Michigan 48109

ABSTRACT

The design of high-throughput large-state Viterbi decoders relies on the use of multiple arithmetic units. The global communication channels among these parallel processors often consist of long interconnect wires, resulting in large area and high power consumption. In this paper, we propose a data-transfer oriented design methodology to implement a low-power 256-state rate-1/3 IS95 Viterbi decoder. Our architectural level scheme uses operation partitioning, packing, and scheduling to analyze and optimize interconnect effects in early design stages. In comparison with other published Viterbi decoders, our approach reduces the global data transfers by up to 75% and decreases the amount of global buses by up to 48%, while enabling the use of deeply pipelined datapaths with no data forwarding. In the RTL implementation of the individual processors, we apply precomputation in conjunction with saturation arithmetic to further reduce power dissipation with provably no coding performance degradation. Designed using a 0.25 μm standard cell library, our decoder achieves a throughput of 20 Mbps in simulation and dissipates only 450 mW.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms

Design, Performance

Keywords

Communications, Pipelining, Bus reduction

1. INTRODUCTION

Viterbi decoders (VDs) are widely used in digital wireless communication systems due to their powerful error correction capabilities. The quality of a VD design is mainly measured by 3 criteria: coding gain, throughput, and power dissipation. High coding gain results in low data transfer error probability. High throughput is necessary for high-speed applications such as 802.11a wireless LAN. The design of VDs with high coding gain and throughput is made challenging by the need for low power, since VDs are often placed in communication systems running on batteries.

The design of single-chip VDs has been a very active research area for the past 15 years. Figure 1 shows the throughput, power dissipation, and number of states for a large collection of VD designs [1, 3, 8, 27]. In general, these VDs fall within the region between the two dashed lines. While small-state VDs can have throughput over 1 Gbps, throughput decreases with the increase in the number of states. Consequently, the design of high-throughput large-state VDs, though crucial for applications such as satellite communications [2], has remained largely unexplored.

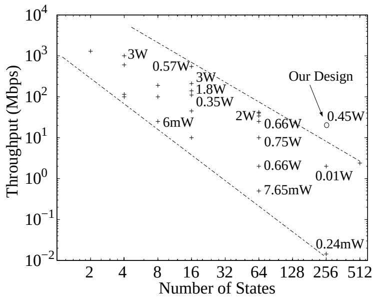

Figure 1: Performance of various VDs.

The main challenge in implementing high-throughput large-state VDs with low power dissipation is the rapid growth in the computational complexity of a VD with the increase in its state count. Although parallel arithmetic processors can be introduced to speed up the computation, they often generate extremely complex interconnect routing problems, degrading both system throughput and power dissipation. Therefore, achieving high speeds using a large-state VD with low power dissipation requires careful consideration of global data transfer and interconnect issues.

In this paper, we present a low-power design methodology for building high-throughput large-state VDs. We also discuss the application of our methodology to the design of a 256-state rate-1/3 IS95 VD. Our VD chip achieves high power efficiency due to our comprehensive design approach that focuses on the reduction of global data transfers, the minimization of global buses, and the maximization of datapath pipeline depths with no data forwarding requirements. The proposed data-transfer oriented optimization consists of 3 steps. Iterative bi-partitioning is first performed on the data flow graph of the VD to cluster all operations with minimal global data exchanges. This step is highly effective and reduces the number of data transfers by up to 75% compared with previously published partitioning approaches. In the second step, operations are packed into slices across different partitions to minimize the volume of global buses. Without stalling any of the processors, this optimization decreases global buses by up to 48% compared with the introduction of buses between every pair of processors. Finally, the execution order of the operation slices is scheduled so that the datapaths of the computation processors are deeply pipelined without requiring any data forwarding circuitry. This scheduling step provides high computation capacity and avoids power dissipation due to data hazard detection and data transfer.

In addition to our inter-processor level optimizations, we describe a novel application of precomputation for reducing power dissipation at the processor level. Specifically, we combine precomputation with saturation arithmetic computation to reduce the power dissipation of datapaths by shutting down datapath circuits that do not contribute to the decoding results. Our approach is provably effective in reducing power dissipation without degrading decoding quality.

Our high-level optimization results in a low-power VD architecture consisting of 8 parallel processors connected by 4 global buses. Our VD was synthesized using a 0.25 μm standard cell library. The processors were placed in an array structure manually and then routed automatically. In simulations using layout, our decoder achieves a throughput of 20 Mbps while dissipating only 450 mW. To our knowledge, this dissipation is the lowest among published VDs with the same number of states and throughput.

The remainder of the paper has 7 sections. In Section 2, we give background on the Viterbi algorithm, previous VD designs, and several techniques for low-power parallel DSPs. Section 3 describes our partitioning approach. Our packing method is presented in Section 4. Our scheduling scheme is presented in Section 5. Section 6 describes the architecture and implementation of our VD. Chip performance is summarized in Section 7. Section 8 concludes our paper.

2. BACKGROUND

2.1 Viterbi decoding

Figure 2 demonstrates Viterbi decoding using a simple example. Figure 2(a) shows a Viterbi encoder with 3 registers and, therefore, 8 possible states. In each clock cycle, one bit I is shifted into the encoder, and 2 bits $\{O_1, O_0\}$ are generated. Figure 2(b) gives the trellis representation of the state diagram in the decoder. The objective is to compute the state sequence in the encoder using the received code $\{C_1, C_0\}$. Each row of vertices represents a single state labeled on the left. Each column represents the states at different times. The edges describe the possible state transitions and are labeled by the corresponding encoder outputs. For example, the edge $u \to t$ represents the transition from state 0 to state 0. It is labelled with $\{00\}$, the encoder output of this transition. Similarly, the edge $v \to t$ is labeled $\{11\}$. Both fanout and fanin of every vertex are 2, and the trellis topology is identical regardless of the code $\{C_1, C_0\}$.

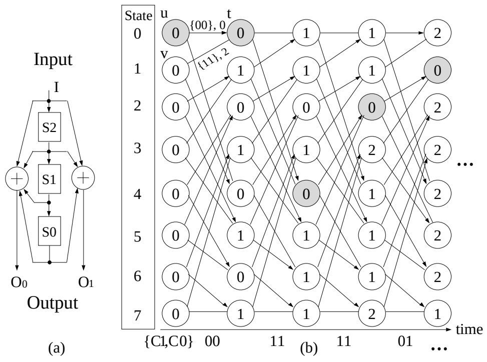

Figure 2: Viterbi decoding example.

Viterbi decoding proceeds in 3 steps. In the first step, each edge e is assigned a weight W_e, which is the difference between the edge label and the code $\{C_1, C_0\}$. For example, $W_e(v \to t) = 2$, because the Hamming distance between $\{11\}$ and $\{00\}$ is 2. Similarly, $W_e(u \to t) = 0$.

In the second step, the weight L of each vertex at column $n+1$ is computed recursively by:

$$L = \min\{L_1 + W_1, L_2 + W_2\} , \tag{1}$$

where L_1 and L_2 are vertex weights of the two predecessors in column n, and W_1 and W_2 are the weights of the corresponding edges. This computation is called add-compare-select (ACS) recursion. Figure 2(b) shows all vertex weights, with arrows representing the selection results. (Vertex weights in the first column are initialized to zero, assuming no prior information indicating the initial state of the VD.)

The final step is called *survivor path tracing*. Intuitively, the minimum-weight path through the trellis represents the most likely state transitions. Starting from the minimum-weight vertex in the last trellis column, a path is traced backward based on the decisions in the ACS update steps. The result is marked by the grey vertices in Figure 2(b).

VD architectures can be classified into three groups: serial, parallel, and intermediate. Serial approaches use only one ACS processor to compute all the vertex weights sequentially. They are only applicable to low-throughput large-state VDs due to their lack of processing capacity.

Parallel architectures consist of the same number of ACS processors as that of states. The main challenge in these designs is to route the connections, called shuffle exchange networks, among all processors. Various graph representations such as de Bruijn graphs [15], hypercubic networks, cube-connected cycles [4], and ring structures [21] achieve only limited interconnect reduction and are not effective for VDs with hundreds of states. Locally connected array architectures avoid global interconnects but suffer from low computation efficiency, because processors are used to transfer data as well as compute path weights [9]. Bit-serial computation approaches reduce interconnect routing by transferring data one bit at a time, but their throughput is severely limited. Due to the data dependency of adjacent trellis columns, pipelining cannot be used in parallel architectures. Interleaving requires changes in communication protocols and is inconvenient to apply [14].

Intermediate solutions, where each ACS processor is shared by more than one trellis state, are promising for large-state VDs [2, 20]. Many such architectures have been proposed, including systolic arrays [5] and 4-level cluster networks [11]. These approaches tend to focus on scalability, overlooking power and performance optimization of the decoder. No operation scheduling has been used to reduce global buses. In [12], operation scheduling was discussed but only in the restricted context of in-place memory update [18]. Pipelining of operations was explored in [6]. However, that work did not present any hardware implementation.

2.2 Low-power parallel DSP system design

In addition to VDs, parallel architectures have been applied to numerous computation-intensive DSP problems [7, 10, 19, 24]. In such systems, multiple computation and memory units operate in parallel to achieve high throughput and low power [13]. The literature on this topic is very extensive, and this section gives a very brief and certainly non-exhaustive discussion of related work.

Parallel hardware synthesis research addresses operation partitioning, scheduling, and binding problems, which are known to be NP-complete and are solved using various heuristics [23]. Design automation research in parallel synthesis is increasingly focusing on data transfer and scheduling over global buses [22], because power dissipation is dominated by data storage and transfer through global interconnect. Such data-transfer oriented scheduling is data-dependent, and problem-specific solutions are often applied [26]. The application of scheduling and binding to reduce interconnect power was investigated in [16]. That work relies on the simulated annealing algorithm and was applied only on small designs such as 2-step FIR filters.

3. REDUCING GLOBAL DATA TRANSFERS

We chose to implement our VD using an intermediate architecture due to its potential for achieving low power. As is the case with all systems consisting of multiple processors, global data transfers among various parallel datapath units could contribute significantly to power dissipation due to long interconnects. This section describes our operation partitioning approach that assigns operations to different processors so that the total communication volume is minimized. As the first step of our optimization methodology, this partitioning lays the ground for the subsequent operation packing for minimal global buses, described in Section 4.

Our analysis focuses on the operations that compute the vertex weights of a single trellis column, since there are no data dependencies among them. In our scheme, the weight of each state is stored in the processor that computes it. Consequently, we build a directed data flow graph (DFG) representing the data transfer pattern of the Viterbi algorithm as follows. For each state in the trellis, a vertex is introduced in the DFG. A directed edge $u \to v$ is introduced if and only if the weight update of v depends on the weight of u in the previous column. Figure 3 illustrates DFG construction using the simple 8-state VD example from Section 2. Figure 3(a) shows adjacent trellis columns and their connections. The DFG in Figure 3(b) is generated by merging the two vertices of the same state together.

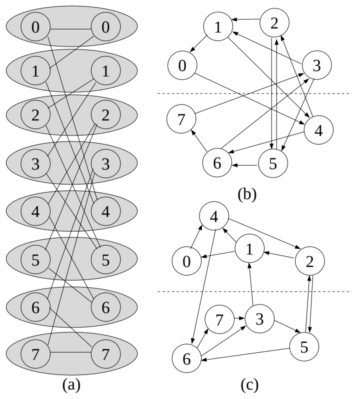

Figure 3: Operation partitioning.

The problem of partitioning all operations to minimize data transfer volume can be cast as a multi-way partitioning problem with minimal cut-size and can be solved using iterative bi-partitioning. Figure 3(c) gives the solution for optimally partitioning the DFG of the 8-state VD into 2 parts. Its global transfer volume is 4, a 50% reduction compared with the simple partitioning approach in Figure 3(b).

Figure 4 shows the effectiveness of our operation partitioning on the IS95 VD. It gives the ratio of global, i.e. inter-partition, data transfers to all data transfers versus the number of partitions. For comparison, we show the results of [25], the only approach we found in previous VD designs that uses partitioning specifically for interconnect reduction. Our partitioning approach reduces the

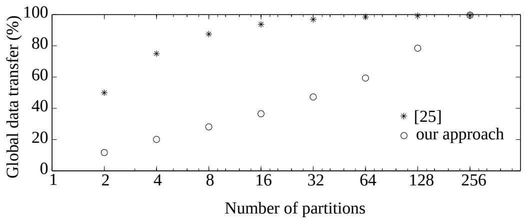

Figure 4: Reduction of global data transfers.

global transfer volume by up to 75%. The largest absolute reduction is achieved in the range of 4 to 32 partitions. Global data transfers increase monotonically with the number of partitions. For 64 or more partitions, more than half of the total data transfers are global, suggesting increased power dissipation in systems with high computation parallelism. In these cases, introducing more parallel processors to increase throughput is inefficient in terms of power dissipation. Pipelining individual processors is a promising alternative for high-throughput and low-power VD implementations. However, pipelining may cause data hazards and often requires power-consuming data forwarding. To that end, Section 5 describes our operation scheduling scheme for non-forwarding pipelining.

4. MINIMIZING GLOBAL BUSES

This section describes our operation packing scheme for global bus minimization. To perform global data transfers, we use a hierarchical bus structure that is derived directly from our partitioning procedure. Each bi-partitioning step generates a unique type of buses. Figure 5(a) shows the bus structure resulting from 8-way partitioning. Blocks $P_i, i = 0, ..., 7$, represent the operation clusters. The dotted lines illustrate the partitioning cuts. The triple-line arrow in the middle gives the highest level cut and points to the corresponding bus. Similarly, the double-line and single-line arrows represent the cuts in the following iterative bi-partitioning procedure and indicate their associated buses. Consequently, bus types can be represented using a full binary tree with all operation partitions as the leaves shown in Figure 5(b). A *bus configuration* can be described by a non-negative integer assignment to all non-leaf nodes in the binary tree. For instance, the assignment in Figure 5(b) represents two buses connecting all processors and one bus connecting $p4$ and $p5$. These buses form a partial order, and a bus u can *cover* another bus v if and only if v is at the same position as u or in the subtree rooted at u.

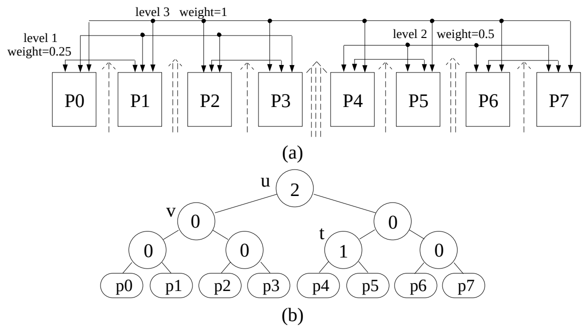

Figure 5: Bus structure example.

A data transfer between two processors can be performed through all buses whose subtrees contain both processors. Based on our partial order definition, the *transfer requirement* is the least upper bound of the two processor leaves in the data transfer. For example, in Figure 5(b), the transfer requirement of a read access between $p4$ and $p5$ is a t-type bus. The weight update operation of a trellis vertex k will require two data transfers if both fanin vertices of k are not in the same partition as k. The *operation requirement* of updating the weight of k is computed by adding the transfer requirements of all data transfers involved. Effectively, this definition ensures that all transfers of an operation can be performed in the same cycle. For example, if an operation requires $p4$ to read $p3$ and $p5$, its operation requirement is one u-type bus and one t-type bus.

In our parallel VD system, operations are executed in *slices* simultaneously by different processors. Each slice contains exactly one operation from each partition. The *slice requirement* is the sum of operation requirements of the operations within. A bus configuration can *satisfy* a slice if and only if each transfer requirement in the slice is covered by a unique bus within the configuration. *Operation packing* is the procedure of forming all slices given a partitioning. A bus configuration is *valid* for a packing solution S, if it satisfies all the slices in S.

Bus resources are quantified by assigning a weight to each bus. The weight of a bus B is proportional to the number of processors connected to B and is normalized by the total number of processors used. The *cost* of a bus configuration is the sum of weights of all buses within. The problem of *operation packing for minimal buses* is to find a packing solution which has a minimum-cost valid bus configuration.

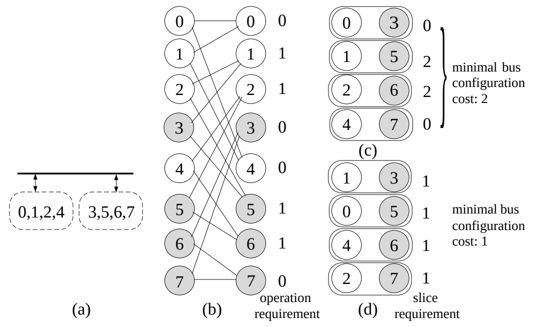
0,1,2,4
3,5,6,7
operation requirement
slice requirement
minimal bus configuration cost: 2
minimal bus configuration cost: 1
(a)
(b)
(c)
(d)

Figure 6: Packing for minimal bus weight.

Figure 6 uses the 8-state VD example to explain the procedure of operation packing for minimal buses. Figure 6(a) shows the VD structure consisting of two processors represented by blocks and connected by buses. Based on the partitioning result from Figure 3(c), operations executed by these two processors are listed within the blocks. Since only one bus type exists, the transfer requirement is a scalar number. Figure 6(b) shows part of the trellis, in which the operation requirement for the weight update of each state is listed on the right. Figure 6(c) gives a packing result in which each pair of operations on the same row form a slice. Since the slice requirement of $\{1,5\}$ is 2, at least 2 buses have to be used. If the packing is changed to the one in Figure 6(d), only one bus can handle all the data transfers, resulting in a 50% reduction of bus resources.

We designed a heuristic to solve the problem of operation packing for minimal buses. Intuitively, since a valid bus configuration must satisfy all slice requirements, the optimal packing solution should distribute the transfer requirements evenly among all slices. Our heuristic procedure, called HS, first packs the operations so that the transfer requirements for the highest level buses of each slice vary by at most 1. Subsequently, it sets the highest level bus number in the bus configuration accordingly. HS then iteratively proceeds to the next lower level and balances the bus requirement by swapping operations between pairs of slices. At each level, the swapping is done in such a way that the current bus configuration is kept valid. The packing procedure terminates when bus numbers at the lowest level of the bus configuration are computed.

Table 1: Bus reduction through operation packing.

Partitions	2	4	8	16	32	64	128
BC_{HS}	2	2	4	8	16	42	111
BC_{simple}	2	3	7	15	31	63	127
Reduction(%)	0	33	43	47	48	33	13

Table 1 shows the results of applying our packing heuristic on the IS95 VD. The numbers of partitions are listed in Row 1. Given any partition number, we applied iterative bi-partitioning followed by our heuristic HS. Row 2 lists the minimal total bus cost BC_{HS}. For comparison, the results BC_{simple} of a simple bus structure where one bus is introduced between any two partitions is given in Row 3. This simple approach still has to introduce stalls, when an operation currently reads two data from another partition, due to lack of bus resources. Procedure HS not only guarantees no-stall execution but also reduces the global buses by 31% on the average. Our approach is most effective in the range of 8 to 32 partitions.

5. NON-FORWARDING SCHEDULING

Data forwarding is often used to resolve data hazards in pipelined datapaths. However, in parallel systems, data hazards occur not only within but also among processors. Data dependencies checking and global data transfers can cause excessive interconnect routing overhead and power dissipation. In this section, we present our operation scheduling scheme for maximizing the depth of non-forwarding pipelines.

In Viterbi decoding, data dependencies only happen between adjacent trellis columns. These limited and fixed data dependencies allow us to use operation scheduling to avoid data hazards without using data forwarding or stalling. In the remainder of this section, we refer to the number of pipeline stages between data fetching and writing as the datapath pipeline depth, because pipelining beyond this range does not cause data hazards.

Since parallel processors execute operations concurrently, we must consider data dependencies between operation slices. The problem of *non-forwarding scheduling* is to find a slice execution order such that the slices in the pipeline never depend on each other's result. The problem can be formulated as an ordering problem on a dependency graph, which is constructed as follows. For each operation slice, a vertex is introduced in the graph. If the weight update of any state in a slice u requires data from a state in another slice v, an edge $u \rightarrow v$ is introduced. The computation is N-stage pipelineable if and only if there exists an order of all the vertices such that, for every k, $0 < k < N$, the kth vertex has no outgoing edges connected to the last $N-k$ vertices.

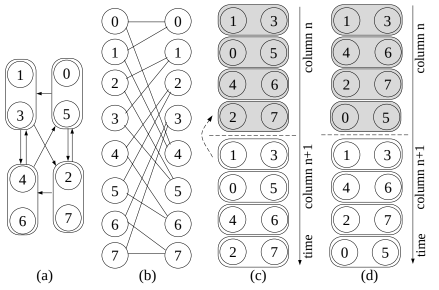

Figure 7: Scheduling for non-forwarding pipelining.

We demonstrate the effectiveness of non-forwarding scheduling using the previous 8-state VD example. Figure 7(a) gives the dependency graph. The vertices are derived based on the partitioning and packing results from Figures 3(c) and 6(d), respectively. The directed edges are added according to the trellis data transfer in

Table 2: Maximum non-forwarding pipeline depth.

Partitions	2	4	8	16	32	64	128
P_{depth}	63	28	9	2	0	0	0

Figure 7(b). Figure 7(c) shows the execution of slices using the order in Figure 6(d). The weight computation of column *n* is shaded. Since there is data dependency between slices {1,3} and {2,7}, pipelining is not possible without data forwarding or stalling. However, pipelining can be applied without data forwarding or stalling if the slices are scheduled as in Figure 7(d).

We implemented a heuristic scheduler to maximize pipeline depth without data forwarding. Intuitively, since the data dependencies occur only between the first and last several slices in the execution order, our scheduler places the slices with fewer dependencies earlier. It first picks the slice with the least dependencies and then iteratively picks the next slice such that the upper bound of the pipeline depth derived by the chosen slices is maximized.

Table 2 shows the results from the application of our scheduler to the IS95 VD. The non-forwarding pipeline depth is drastically reduced as the number of partitions increases, reflecting the reduction in the number of operations in each partition and the simultaneous execution of increasingly many operations. From this table, it is evident that architectures with 16 or more parallel processors allow very limited or zero non-forwarding pipeline depth.

6. SYSTEM ARCHITECTURE

Using our analysis in Sections 3, 4, and 5, we have designed a system with 8 processors (subcores) that has low global communication volume, a small number of global buses, and a large non-forwarding pipeline depth. The block diagram of our design is shown in Figure 8. The 8 subcores are connected by 4 global address/data (A/D) buses. Each subcore is a 16-bit pipelined microprocessor, whose internal structure is detailed on the right-hand side of Figure 8.

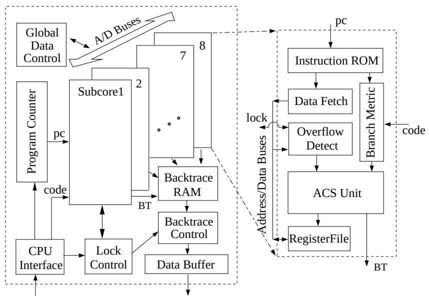

Figure 8: Viterbi decoder architecture.

The **instruction ROM** is derived from our high-level data transfer oriented optimization. The 16-bit instruction format is given in Figure 9. The *Ex* bits indicate the sources of two operands (local register file or global buses). The *Addr/port* fields give the local register file addresses or global port numbers. The *ICodes* and *D* are the label and decoding data of the state transition edge associated with the first operand. The label and decoding data of the state transition edge for the second operand can be derived using bit-inversion of *ICodes* and *D*, respectively.

The **data fetch unit** gets the operands based on the instructions. For each global bus, we use a centralized control. Synchronized with the subcores, global buses deliver the right data at the right time. The data fetch unit just latches the data in and no bus conflict detections are needed, resulting in low power dissipation.

The **branch metric unit** computes the difference between the received codes x_i and the label y_i of each trellis edge $\mathcal{D}$. We used soft-

15	14-10	9	8-4	3-1	0
Ex1	Addr/Port1	Ex2	Addr/Port2	ICodes	D

Figure 9: Instruction format.

decision Viterbi decoding with 8-level quantization, thus avoiding the loss of coding gain associated with hard-decision decoding [17]. Since there are 3 codes for each state transition, $\mathcal{D}$ is calculated as follows:

$$\mathcal{D} = \sum_{i=0,1,2} (x_i - y_i)^2 \ . \tag{2}$$

Our **ACS unit** is a 16-bit datapath pipelined into 3 stages. A novel power saving technique applied to the design of our ACS unit is the combination of precomputation and saturation computation that allows overflow. Specifically, once one or two operands are detected as overflows, part of the ACS is shut off to save power. In previous VD designs, datapaths were adjusted to hold the maximal path weight, and no overflow was allowed. Therefore, the trellis had to be frequently backtraced when the *maximal* path weight exceeded the data limit of the registers. This approach may result in unnecessary power consumption. Since in the Viterbi algorithm, the result is the path with minimum weight, the computation is valid as long as the minimum path weight does not exceed the data limit of the register.

In our VD design, we use saturation computation, in which weights exceeding the maximal data value $M = 2^{16} - 1$ are represented as overflows. All path weights are monitored as they are written into the register files. When the minimum path weight is greater than or equal to $M/2$, all finite path weights are decreased by $M/2$. Overflows remain as overflows, however. Our approach degrades the VD quality only when the weight of the correct path overflows. This condition requires the weight of the correct path to exceed that of an incorrect one by $M/2$. The probability of such an event is less than 2^{-500} in our design. (Proof omitted due to page limits.)

The **register file** contains 64 registers. Two 16-bit registers are used for each of the 32 states in the corresponding partition to decouple the write and read accesses. The register file has 1 write port and 4 read ports, 3 of which are global read ports. The global read accesses are pipelined into 3 stages to handle delays due to the long interconnects.

With the data overflow detection, we perform backtracing at the frequency determined by the decoding latency requirement. Our **backtrace memory** consists of two independent parts, each handling 4 subcores. We use a ping-pong architecture to handle synchronized reads and writes with a single port. Because the write addresses follow a round-robin pattern, we divide the backtrace memory into two banks for odd and even addresses so that every two consecutive writes use either bank exactly once. Therefore, we are able to read the memory at a speed twice as slow as the write accesses. This speed is fast enough, because read accesses are performed in the backtracing procedure, which is much slower than the write accesses during the path weight computation.

7. CHIP SUMMARY

Our VD was implemented using *Verilog* and synthesized using *Design Analyzer* from Synopsys and a 0.25 μm standard-cell library. The layout was generated using *Silicon Ensemble* from Cadence in a hierarchical fashion. Each sub-module was laid out as a standard cell array. Manual floorplanning was performed to minimize the length of global buses. Figure 10 shows the floorplan and layout of our VD. The subcores are rotated so that the register files (RF) and the fetch units (FU) can access the global channels easily. The backtrace control (BTC) and memory (BTM) are placed in the middle as well as the output buffer (SB).

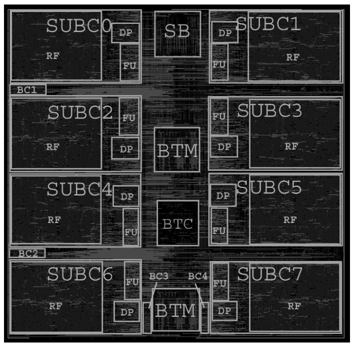

Figure 10: Viterbi decoder layout.

Table 3: Chip Summary.

technology	0.25 μm	metal layers	5
chip size	$2.4 \times 2.4mm^2$	transistor count	325K
core frequency	640MHz	supply voltage	2.5V
throughput	20 Mpbs	power	450 mW

After layout, the capacitance of all interconnects was extracted, and the timing of each gate was adjusted according to the output load. Our design was then resimulated using *Verilog* with the back-annotated timing and capacitance. The power dissipation was estimated using *PrimePower* from Synopsys. Table 3 summarizes our chip performance. Our decoder has a maximal throughput of 20 Mbps while dissipating only 450 mW.

8. CONCLUSION

In this paper, we present a comprehensive methodology for the design of a high-throughput and low-power 256-state rate-1/3 IS95 Viterbi decoder. Our high-level design optimization focuses on the power reduction of global data communications. We perform a 3-phase partitioning, packing, and scheduling optimization to reduce global data transfer volume, minimize global buses, and implement deep pipelining without data forwarding. Furthermore, we apply precomputation in conjunction with saturation computation to reduce power dissipation within each datapath.

The resulting VD is a parallel system with 8 pipelined processors connected through 4 global buses. Generated using commercial ASIC tools in a 0.25 μm standard cell library, our decoder has a throughput of 20 Mbps and a power dissipation of 450 mW, the lowest among published VDs with 256 states and same throughput.

9. ACKNOWLEDGMENTS

This research was supported in part by NSF under Grant No. CCR-0082876 and by ARO under Grant No. DAAD-19-99-1-0304.

10. REFERENCES

[1] P. J. Black and T. H. Meng. A 1-Gb/s four-state sliding block Viterbi decoder. *IEEE J. of Solid-State Circuits*, 32(6):797–805, June 1997.

[2] M. Bóo, F. Argüello, J. D. Bruguera, R. Doallo, and E. L. Zapata. High-performance VLSI architecture for the Viterbi algorithm. *IEEE Trans. Communications*, 45(2):168–176, Feb. 1997.

[3] Y. Chang, H. Suzuki, and K. K. Parhi. A 2-Mb/s 256-state 10-mW rate-1/3 Viterbi decoder. *IEEE J. of Solid-State Circuits*, 35(6):826–834, June 2000.

[4] P. Chau and K. Stephen. Scaling and folding the Viterbi algorithm trellis. In *Workshop on VLSI Signal Processing*, pages 479–489, 1992.

[5] F. Daneshgaran and K. Yao. The iterative collapse algorithm: A novel approach for the design of long constraint length Viterbi decoders - Part I. *IEEE Trans. Communications*, 43(2):1409–1418, Feb. 1995.

[6] H. Dawid, S. Bitterlich, and H. Meyr. Trellis pipeline-interleaving: a novel method for efficient Viterbi decoder implementation. In *IEEE International Symposium on Circuits and Systems*, May 1992.

[7] H. De Man, F. Catthoor, G. Goossens, J. Vanhoof, J. V. Meerbergen, S. Note, and J. Huisken. Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms. *Proc. of the IEEE*, 78(2):319–335, Feb. 1990.

[8] G. Fettweis and H. Meyr. High-speed parallel Viterbi decoding: Algorithm and VLSI-architecture. *IEEE Communications Magazine*, pages 46–55, May 1991.

[9] P. G. Gulak and T. Kailath. Locally connected VLSI architectures for the Viterbi algorithm. *IEEE J. on Selected Areas in Communications*, 6(3):527–537, Apr. 1988.

[10] R. Hartenstein, J. Becker, M. Herz, R. Kress, and U. Nageldinger. A partitioning programming environment for a novel parallel architecture. In *International Parallel Processing Symposium*, pages 544–548, 1996.

[11] H. Li and C. Chakrabarti. A new architecture for the Viterbi decoder for code rate k/n. *IEEE Trans. Communications*, 44(2):158–164, Feb. 1996.

[12] H. Lin and C. B. Shung. General in-place scheduling for the Viterbi algorithm. In *International Conf. on Acoustics, Speech, and Signal Processing*, pages 1577–1580, 1991.

[13] R. S. Martin and J. P. Knight. Optimizing power in ASIC behavioral synthesis. *IEEE Design and Test of Computers*, 13(2):58–70, 1996.

[14] S. R. Meier. A Viterbi decoder architecture based on parallel processing elements. In *Global Telecommunications Conference*, pages 1323–1327, 1990.

[15] B. K. Min and N. Demassieux. A versatile architecture for VLSI implementation of the Viterbi algorithm. In *International Conf. on Acoustics, Speech, and Signal Processing*, pages 1101–1104, 1991.

[16] P. Prabhakaran, P. Banerjee, J. Crenshaw, and M. Sarrafzadeh. Simultaneous scheduling, binding and floorplanning for interconnect power optimization. In *International Conf. on VLSI Design*, pages 423–427, 1999.

[17] J. G. Proakis. *Digital Communications*. McGraw-Hill Inc., New York, 1995.

[18] C. M. Rader. Memory management in a Viterbi decoder. *IEEE Trans. Communications*, 29(9), Sept. 1981.

[19] L. Raffo, S. P. Sabatini, M. Mantelli, A. D. Gloria, and G. M. Bisio. Design of an ASIP architecture for low-power visual elaborations. *IEEE Trans. VLSI Systems*, 5(1), Mar. 1997.

[20] C. B. Shung, H. Lin, R. Cypher, P. H. Siegel, and H. K. Thapar. Area-efficient architectures for the Viterbi algorithm- part I: Theory. *IEEE Trans. Communications*, 41(4):636–644, Apr. 1993.

[21] J. Sparsø, H. N. Jørgensen, E. Paaske, S. Pedersen, and T. Rübner-Petersen. An area-efficient topology for VLSI implementation of Viterbi decoders and other shuffle-exchange type structures. *IEEE J. of Solid-State Circuits*, 26(2):90–96, Feb. 1991.

[22] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPs. In *Design Automation Conference*, pages 388–392, June 1995.

[23] C. Wang and K. K. Parhi. High-level DSP synthesis using concurrent transformations, scheduling, and allocation. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 14(3):274–295, Mar. 1995.

[24] J. P. Wittenburg, W. Hinrichs, J. Kneip, M. Ohmacht, M. Bereković, H. Lieske, H. Kloos, and P. Pirsch. Realization of a programmable parallel DSP for high performance image processing applications. In *Design Automation Conference*, pages 56–61, June 1998.

[25] C.-M. Wu, M. Shieh, C.-H. W, and M. Sheu. An efficient approach for in-place scheduling for path metric update in Viterbi decoders. In *International Symposium on Circuits and Systems*, pages 61–64, May 2000.

[26] S. Wuytack, F. Catthoor, G. D. Jong, and H. J. De Man. Minimizing the required memory bandwidth in VLSI system realizations. *IEEE Trans. VLSI Systems*, 7(4):433–441, Dec. 1999.

[27] A. K. Yeung and J. M. Rabaey. A 210Mb/s radix-4 bit-level pipelined Viterbi decoder. In *IEEE International Solid-State Circuits Conference*, pages 88–90, 1995.