THE IMPLEMENTATION OF AN IRREGULAR VITERBI TRELLIS DECODER

Christopher Lavin
Telemetry Laboratory
424 Clyde Building
Brigham Young University
Provo, UT 84602
Michael Rice
Faculty Advisor

ABSTRACT

The Viterbi algorithm has uses for both the decoding of convolutional codes and the detection of signals distorted by intersymbol interference (ISI). The operation of these processes is characterized by a trellis. An ARTM Tier-1 space-time coded telemetry receiver required the use of an irregular Viterbi trellis decoder to solve the dual antenna problem. The nature of the solution requires the trellis to deviate from conventional trellis structure and become time-varying. This paper explores the architectural challenges of such a trellis and presents a solution using a modified systolic array allowing the trellis to be realized in hardware.

KEY WORDS

Viterbi Algorithm, Trellis, FPGA, Hardware

INTRODUCTION

There exists a persistent problem in aeronautical telemetry of data dropouts caused by interference due to multiple transmit antennas. A solution [6] has been proposed and tested to reduce data drop outs by using a combination of techniques to mitigate destructive interference at the receiver. A prototype aeronautical telemetry receiver built at Brigham Young University [6] implements this solution by using an ARTM Tier-1 modulation (SOQPSK) and Alamouti space-time encoding for the data. This receiver also compensates for the spacing of antennas on aeronautical test vehicles which introduces a significant delay between the two signals as seen by the receiver.

The basic structure of the transmitter and receiver is shown in Figure 1. The transmitter receives blocks of 3200 bits at a time (arriving at 10 Mbits/s) and uses the Alamouti encoding scheme to produce two equivalent (but orthogonal) sequences. Each sequence then has a 128 bit training sequence (pilot) inserted with each frame and is transmitted on its own antenna.

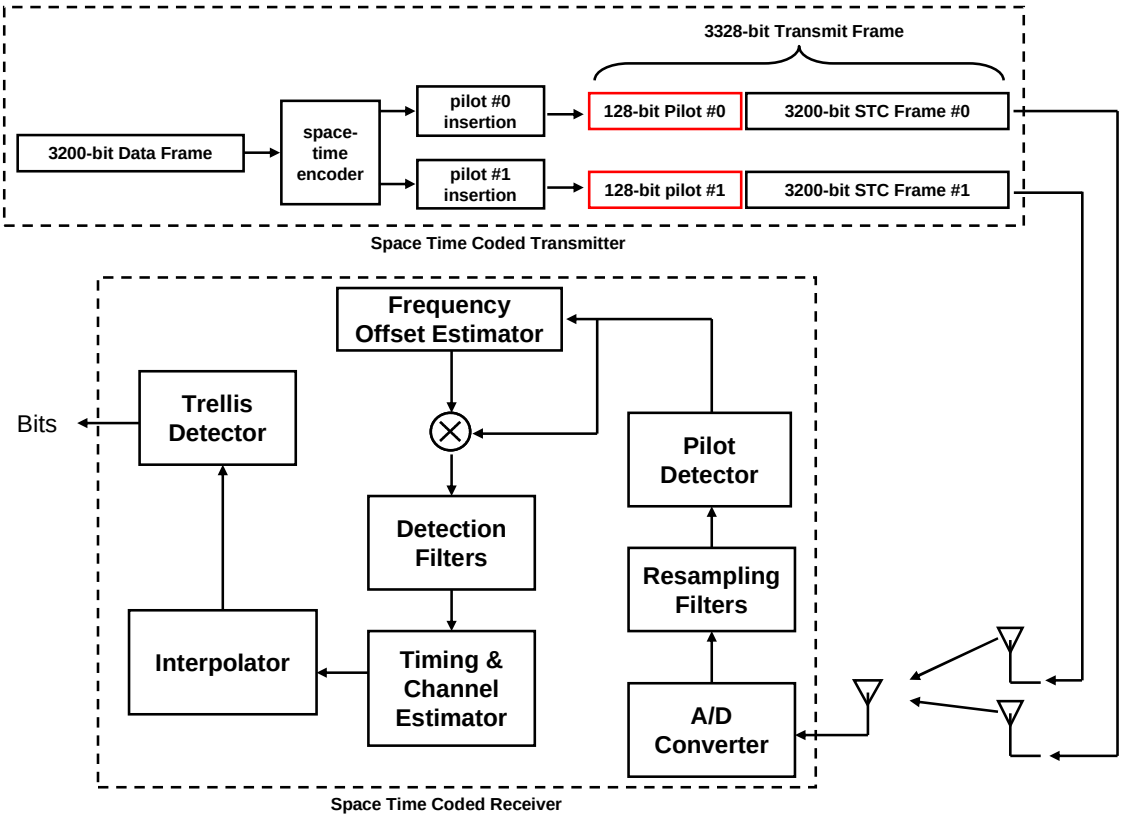


Figure 1: Block diagram of space-time coded system

The receiver can receive one or both signals and samples the 70 MHz IF at 93.3 MHz. This operation coupled with an aliasing digital down converter (not shown explicitly) centers the signal at the quarter sampling rate (23.3 MHz). After the signal is resampled to four samples per bit, the signal is sent through a frequency domain correlation that will detect the position of the pilots in the signal. Once the position of the pilot is known, it is used to determine the frequency offset, the time delay between the two received signals, and the channel gains. The frequency offset is removed and the signal is then interpolated down to one sample per bit.

The decoder for the receiver uses a Viterbi trellis decoder for bit detection. However, the combined effects of the offset nature of SOQPSK, the Alamouti space-time code, and the differential delay result in a trellis that has unique challenges for hardware implementation. With these added complexities, the trellis becomes time-varying. This new time-varying trellis differs from a conventional one in two ways: the number of states in each stage varies and the transition pattern in between each stage is different. These two irregularities make it difficult to apply conventional trellis hardware architecture because signal routing increases in complexity. The modified systolic array implementation chosen and presented here has proven its effectiveness in meeting the high performance requirements of our telemetry receiver.

BACKGROUND

A conventional trellis (see Figure 2) exhibits two favorable attributes for hardware implementation. The first occurs when the number of states in each stage of the trellis remains the same. The example in Figure 2 shows a trellis where there are four states in each of the seven stages. The second advantage stems from the repetitive transition pattern in between stages of the trellis. Traditionally, the two regularities allow for a variety of traditional architectural approaches to be considered. The trellis in our receiver, however, does not enjoy the same benefits as those in a conventional trellis. It can be seen in Figure 3 that the number of states varies as well as the transition pattern between stages. There is an additional trellis state diagram when the delay between the two received signals is negative (not shown). It has the same number of states and stages but it has different transition patterns than that shown in Figure 3. In most traditional implementations, the architectures do not map well to a time-varying trellis because of speed limitations and complications in signal routing. A few implementations are summarized here to illustrate.

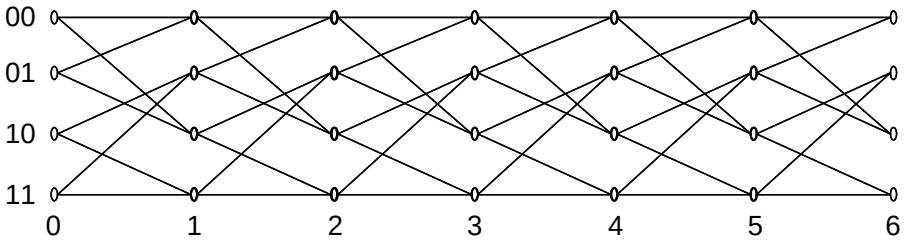


Figure 2: Example of a conventional trellis

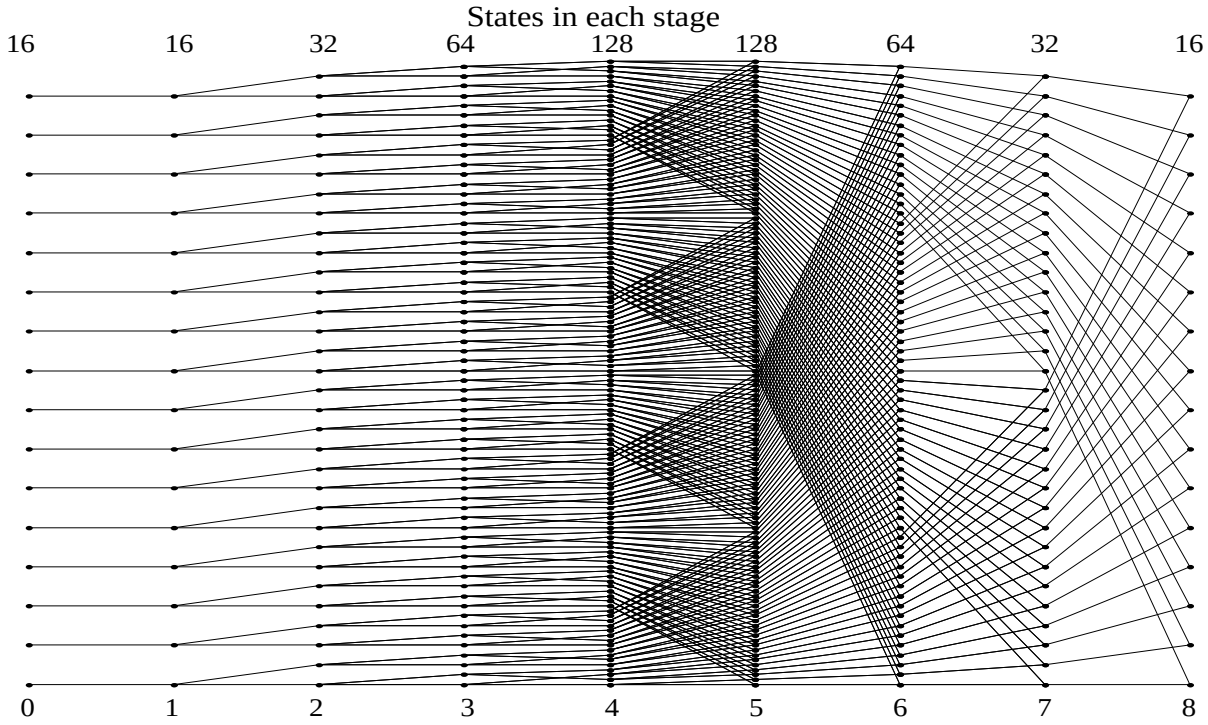
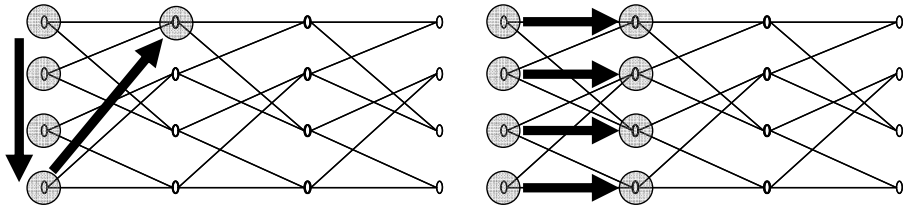


Figure 3: Space-time coded trellis

Serial and parallel implementation as illustrated in Figure 4, are two basic approaches for trellis implementations. In a serial implementation [1], the states are generally computed sequentially, one at a time. This makes for simple operation and has the smallest hardware utilization of any architectural approach. However, since the nodes must be processed sequentially, the number of nodes in the trellis can easily overwhelm the single processing element and diminish its throughput (as is the case in our trellis). In a parallel approach [2], all the states of a stage are computed at the same time. This approach increases hardware usage, but with the added benefit of higher throughput. Several complications arise, however, when using a parallel approach with our trellis. Memory is now required and several ports might be needed depending on the number of parallel nodes. Since the transition patterns are not regular, intermediate results need to be stored and retrieved at the same time. In addition, our trellis has a varying number of states making the parallel implementation difficult. If the implementation used fewer than 128 nodes, it would have to iterate in each stage. If the implementation used 128 nodes or more, several nodes would remain idle during most stages and not be utilized efficiently.



(a) Serial Execution (b) Parallel Execution

Figure 4: Serial and parallel approaches

An additional architectural approach uses a pipeline cascade [1] which was later called the Canonic Cascade Viterbi Decoder (CCVD) [3]. This architecture has shown a high rate of efficiency in its hardware utilization. Unfortunately, the time-varying nature of our trellis would complicate the control and routing of this architecture.

The systolic array architecture [4] [5] represents the closest architecture to our implementation than any other previously mentioned. This architecture assigns one processing element to the four states in each stage when applied to the example in Figure 4. Figure 5 shows how the execution pattern is carried out. The first and second iteration only execute nodes A and B. On the third iteration, however, two states are computed (nodes C and E) in parallel. This type of execution pattern is continued until the end of the trellis and the outputs are used for the next iteration of the trellis. This architecture eliminates the need to store intermediate results in a RAM (such as the case in a parallel implementation). It can also reduce the amount of time that processing nodes stand idle.

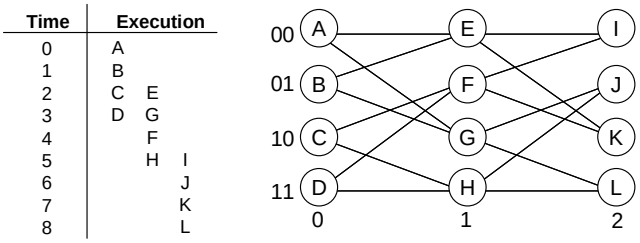


Figure 5: Systolic array approach

PROBLEM

In Viterbi algorithm hardware implementations, three main computational blocks are used for its realization: a branch metric generator (BMG), a survivor path memory unit, and an add-compare-select (ACS) block. These basic computational units can be seen in Figure 6. For every state, a branch metric must be generated requiring a BMG block with the appropriate bandwidth. The survivor path memory unit runs at the end of each pass through the trellis to produce the lowest path cost through its branches. The BMG and survivor path memory unit are simplistic and rarely create a bottleneck for throughput and performance of the decoding process. The ACS block, however, limits performance because of the magnitude of computation required and also its inherent data dependencies. The trellis must compute one ACS computation for every state of every stage. In high throughput applications, the ACS block becomes the critical bottleneck in the system because of its computational volume and dependencies. The dependencies are caused by the fact that the input of stage *k* depends on the output of stage *k-1*. In addition, the first stage depends on the output of the last stage of the trellis, limiting the rate at which a new iteration can begin. Therefore, the performance of a trellis and its architecture are determined by the placement, speed, and organization of the ACS blocks.

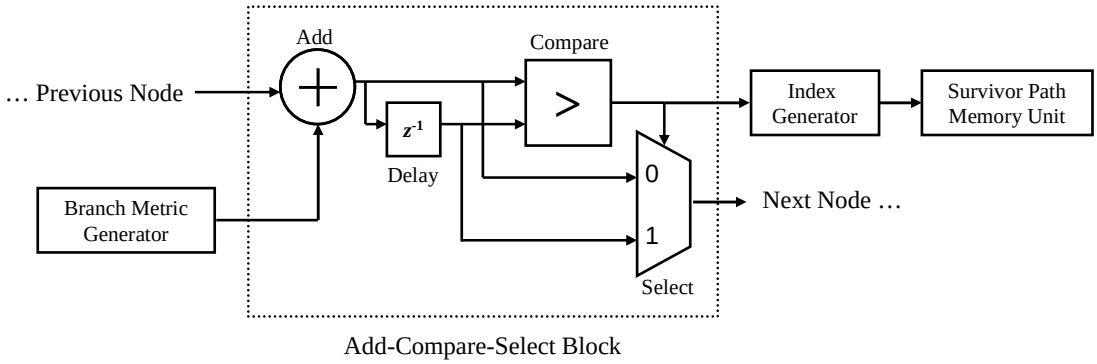


Figure 6: Basic hardware blocks for trellis

The major obstacles in the construction of the trellis in our telemetry receiver are the arrangement and connection of the ACS blocks. As mentioned above, the trellis is irregular because of the varying number of states per stage and the differing transition patterns throughout the trellis. The number of nodes as well as the throughput requirement of the receiver makes the implementation of such a trellis challenging.

The nature of this project allowed the hardware implementation of the trellis to reside on a single field programmable gate array (FPGA). Implemented on a Xilinx Virtex II Pro (XC2VP50) with 23,616 slices and a -7 speed grade, the design was clocked at 206 MHz. Using this FPGA, the major obstacles in creating a hardware implementation of the trellis were the routing of signals between ACS blocks and latency of the architecture. The difficulty in routing stemmed from the irregular transitions between states, the dual set of transitions caused by the differential delay, and the varying number of states in each stage. The latency could not exceed a certain length because of the throughput requirement of 10 Mbits/s. Evaluation of these factors led us to use a systolic array implementation because of its flexibility to address routing and latency.

SOLUTION

The systolic array based architecture made implementation possible by allowing optimization of both latency and routing. The latency became a performance limiting factor due to the recursive nature of the trellis and the data rate requirement of 10 Mbits/s. The systolic array allowed each stage of the trellis to be optimized independently whereas other architectures would have required the worst case latency for each step. The routing problem was solved by the ability to compartmentalize each transition pattern independently. This solution caused the hardware budget to increase; however, the routing of signals became much easier. The systolic array approach provided the means by which certain aspects of the trellis were optimized to reduce both latency and routing.

At 206 MHz, 82 clock cycles became the longest allowable execution time for one trellis iteration. This limit made latency an expensive commodity and was avoided in every case possible. Using the systolic array method, there were two transition stages (one in each trellis) that had data dependencies of 64 elements (or 64 clock cycles) apart. In Figure 3, this dependency can be seen in state 0 of stage 6. This dependency would have made execution impossible if one ACS block was assigned to every stage. However, to maximize throughput while minimizing hardware size, we decided to add one ACS block for every 16 states that a stage contained. Figure 7 shows how the ACS blocks were arranged according to the trellis diagram in Figure 3. This modification to the systolic array architecture guaranteed that the consumers would not out number the producers and reduced the worst case latency for a stage from 64 to 8. The latency could be further reduced by adding additional ACS blocks at the expense of more hardware and lower efficiency utilization. However, since the number of states in each stage is a multiple of 16, assigning one ACS block to every 16 states produced the most efficient hardware usage and provided the best reduction in latency for the cost of the hardware.

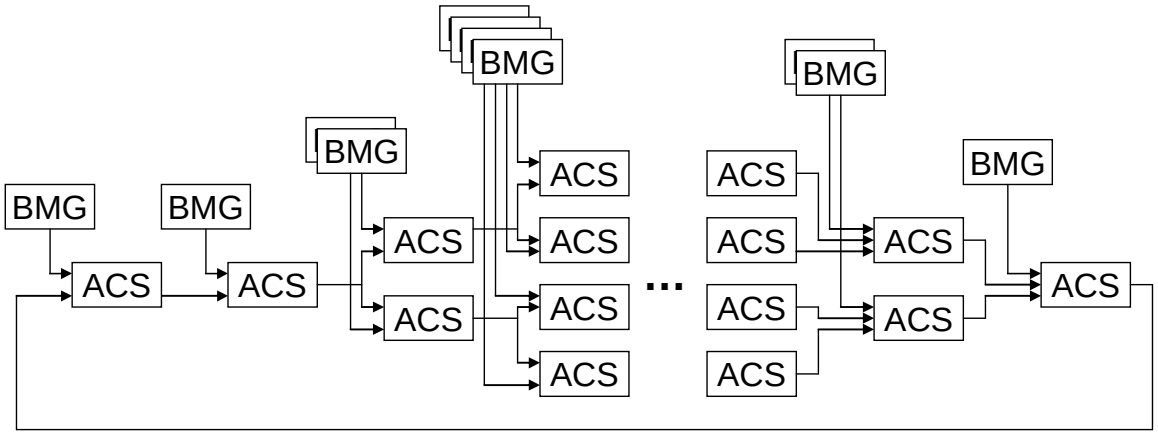


Figure 7: Diagram illustrating ACS block ratio

Since the hardware associated with processing each stage is independent of every other stage, computations can be streamlined as data moves through the trellis. This technique allows the critical path to be pipelined more easily than if other architectures were applied. The pipelined structure made it possible to take advantage of the different patterns present in between each stage transition. If all the stage transitions for all stages had to be controlled within a single

block, the complexity of the routing would be very difficult. It would also force the single block to function at a less optimum rate than customized blocks for each individual state. In addition, the design time for a single block implementation would be significantly more than a pipelined structure as meticulous attention to addressing and latency would be needed.

In our trellis implementation, difficult routing and a long design time were avoided by using the flexible systolic array architecture. The systolic array allowed the transitions in between each stage to be separate from all others. Each set of transitions were encapsulated into a single block connected in between each stage. This provided the benefit of avoiding complicated routing structures and increased latency because of complicated switching. Each set of transitions has a particular pattern that can be easily reproduced with a counter and some Boolean logic. For this reason, each transition block was simple to implement as each pattern could be implemented independently. However, if all the patterns were merged into a single block, a complicated structure would be required. Using separate transition blocks eliminated the long design time required for a block that encapsulated all of the transition patterns and the complicated routing required for such a structure.

RESULTS

An attempt to implement a serial architecture to accommodate the trellis in our receiver would be impossible with an FPGA. The clock rate at which a serialized trellis would need to run at to keep pace with the data rate would be several gigahertz. The speed grade of the parts on the Xilinx Virtex II Pro does not come close to the speed required for such operation.

Complications arise with a parallel implementation on an FPGA. By using 16 ACS blocks, the hardware could be fully utilized as all stages have a multiple of 16 states. This would require 31 iterations of the ACS blocks to complete one pass through the trellis. However, if the trellis operates at 206 MHz, it would only be allowed 2 clock cycles to complete each iteration. This would be extremely difficult to perform in that the add-compare-select generally will take 3 clock cycles to complete. This also does not include the time needed to store and retrieve the data from block RAMs. Using 32 ACS blocks would generate the throughput necessary; however, the control and routing of signals would have to be carefully orchestrated in order to meet timing constraints.

The proposed trellis was fully implemented using the Xilinx Virtex II Pro mentioned previously. It has been tested and has been shown to operate correctly. It took approximately 160 man hours to generate the trellis in hardware using Xilinx System Generator 8.1. At the time of writing, a few minor bugs were still present in the hardware; however, they are expected to be corrected in the few weeks following submission. Table 1 shows the hardware utilization statistics. The trellis occupies just over one third of the slices of the FPGA. Slices are comprised of slice flip flops and 4 input LUTs. There were 61 block RAMs used in the design mainly for the number of ports needed for the BMGs rather than the storage size. High throughput is required and therefore to deliver the calculated branch metrics at a rate equal to the speed of the trellis, the block RAMs were used generously. The block multipliers were also heavily needed in that there are a total of 30 ACS blocks in the design, each of which uses 2 block multipliers. Therefore, 60

block multipliers were dedicated to ACS blocks and the other 32 were used in the branch metric generators.

Resource	Count	Total Available	Usage Percentage of FPGA Resource
Slices	8790	23616	37%
Slice Flip Flops	14163	47232	29%
4 input LUTs	13575	47232	28%
Block RAMs	61	232	26%
Block Multipliers	92	232	39%

Table 1: Resource utilization of space-time coded trellis

CONCLUSION

We have shown that implementation of a complex irregular trellis detector is possible and feasible to implement on a single FPGA. Of all the conventional architectural methods for trellis implementation, a modified systolic array processing method is best for both speed and minimization of signal routing. Further optimization of the implementation is certainly possible to increase speed and reduce FPGA fabric usage.

REFERENCES

[1] Gulak, P., "Locally Connected VLSI Architectures for the Viterbi Algorithm", IEEE Journal on Selected Areas in Communications, vol. 6, no. 3, April, 1988, pp. 527-537.

[2] Chang, Y., "A 2-Mb/s 256-state 10-mW rate-1/3 Viterbi decoder", IEEE J. Solid-State Circuits, vol. 35, no. 6, June, 2000, pp. 826-834.

[3] Feygin, G., "A Multiprocessor Architecture for Viterbi Decoders with Linear Speedup", IEEE Transactions on Signal Processing, vol. 41, no. 9, September, 1993, pp. 2907-2917.

[4] Chang, C., "Systolic array processing of the Viterbi algorithm", IEEE Trans. Inf. Theory, vol. 35, no. 1, January 1989, pp. 76-86.

[5] Guo, M., "FPGA Design and Implementation of a Low-Power Systolic Array-Based Adaptive Viterbi Decoder", IEEE Transactions on Circuits and Systems-I: Regular Papers, vol. 52, no. 2, February, 2005, pp. 350-365.

[6] Nelson, T., Rice M., and Jensen M., "Experimental Results for Space-time Coding Using ARTM Tier-1 Modulation", Proceedings of the 41st International Telemetering Conference, Las Vegas, NV, October 2005, pp. 90-99.