MIT 6.375 Project

Viterbi Decoder

Arthur Chang
Omid Salehi-Abari
Sung Sik Woo

May 12, 2011

Abstract

The use of forward error correction (FEC) technique is known to be an effective way to increase the reliability of the digital communication and to improve the capacity of a channel. Convolutional encoder at the transmitter associated with the Viterbi decoder at the receiver has been a predominant FEC technique because of its high efficiency and robustness. However, the Viterbi decoder consumes large resources due to its complexity, and the decoding load has increased with newer communication standards; as a result, the hardware solution has become crucial for higher performance and cost effectiveness.

A Viterbi Decoder for IEEE 802.16 WiMax Standards (constraint length $K = 7$, supporting rates of $1/2, 2/3, 3/4, 5/6$) has been implemented in Bluespec on an FPGA. The design is based on MATLAB source code of Viterbi decoder. The decoder throughput has been boosted by using parallel and pipelined architecture to reach over 150Mb/s. Bluespec and FPGA played a key role in that it remarkably reduces design effort as well as verification time through high-level synthesis.

Background

Error Control Techniques

To improve the reliability of digital communication, error control techniques are typically employed. Generally, these involve inserting controlled redundancy into the transmitted data, and using this redundancy at the receiver to detect and correct transmission errors. Consider a binary symmetric channel (BSC) with cross-over probability as it is shown in Figure 1. When no error control scheme is used (i.e., each bit is sent without any encoding), the probability of bit error is $P_E = p$. Using a simple coding scheme such as Triple-repetition code (i.e. send each bit three times $1 \rightarrow 111$ and $0 \rightarrow 000$) can help to detect errors and/or correct them. In this case the probability of a message bit error is:

$P_E = P_r$ {2 or 3 code bit flips}$= 3p^2(1-p) + p^3 = 3p^2 - 2p^3$

As it can be seen, utilizing the coding scheme has improved Bit Error Rate (BER).

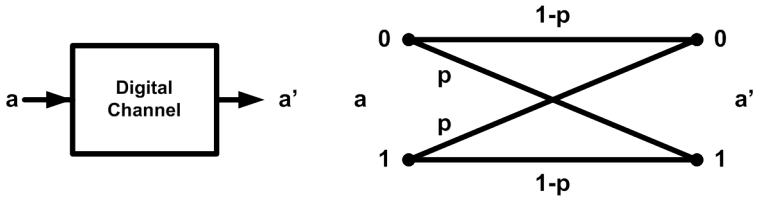

Figure 1: Binary Symmetric Channel (BSC)

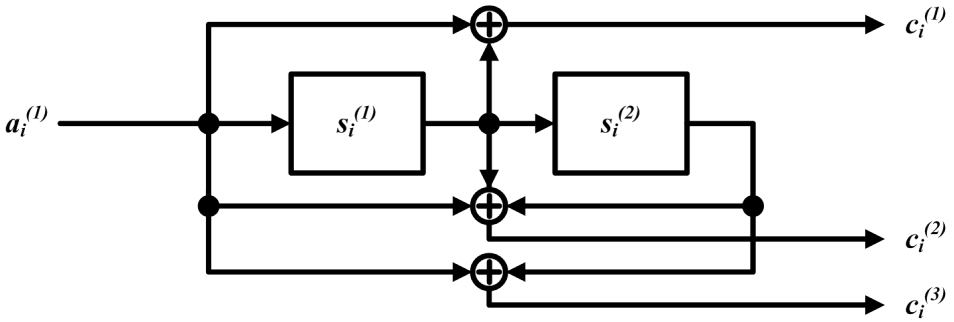

Figure 2: (3,1,2) Convolutional Encoder Block Diagram

Convolutional Codes

Convolutional codes are linear codes that have additional structure in the generator matrix. These encoding operations can be viewed as digital filtering, or convolution, operations. Unlike block codes, which take discrete blocks of k symbols to produce blocks of n symbols that only depend on the k input symbols, convolutional codes are often viewed as stream codes because they operate on continuous streams of symbols not partitioned into discrete blocks.

At each clock cycle, a (n, k, m) convolutional encoder, with m memory stages , takes one message symbol of k bits and produces one code symbol of n bits. Performance of the code can be improved by increasing m. Convolutional codes can be generated by the convolution of the message sequence with a set of generator sequences (i.e. g_1, g_2 ...). Figure 2 shows an example of a $(3, 1, 2)$ convolutional encoder. In this example, g_1, g_2 and g_3 are 110, 111 and 101, respectively.

Punctured Codes

Punctured convolutional codes are derived from ordinary codes by dropping some output bits based on a predefined puncturing matrix. The resulting code has higher rate but less redundancy, hence lower error correcting capability.

Viterbi Decoder

The Viterbi algorithm provides a simple method for decoding convolutional codes which is optimal in that it always finds the path with the smallest path metric (sum of the branch metrics). Branch metrics are the

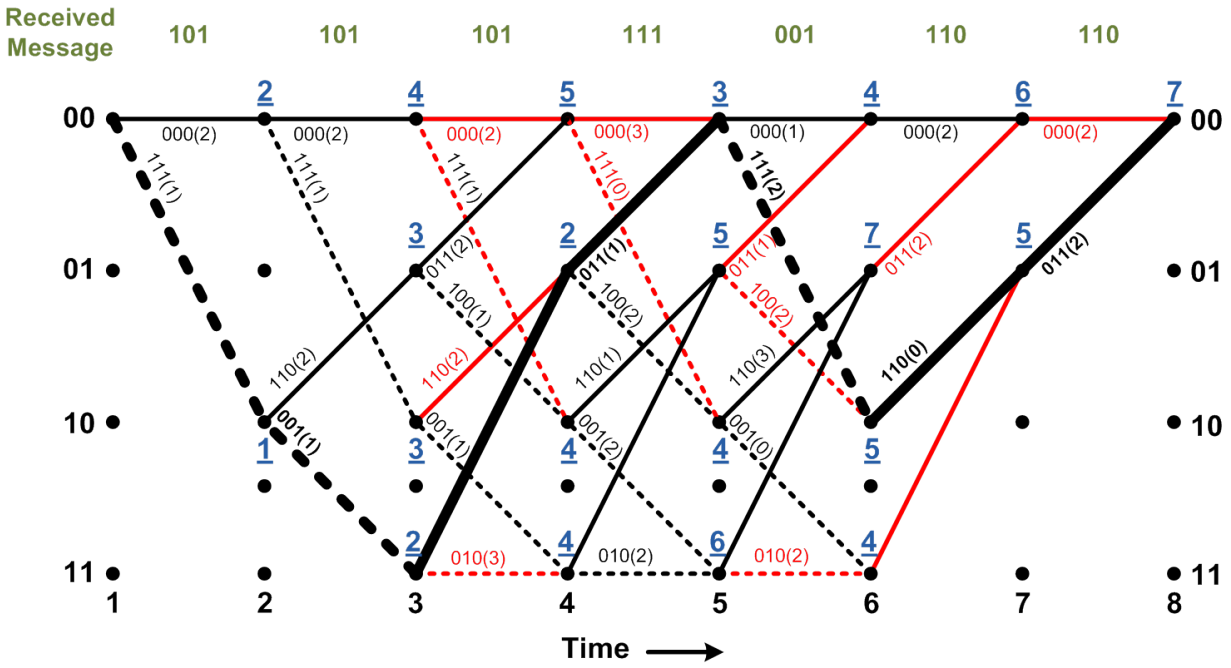

Figure 3: Trellis Diagram of (3,1,2) Encoder with Viterbi Decoder Result

normed distances between all possible symbols in the code alphabet and the received symbol.

The algorithm can be described as follows:

1. Beginning at time unit $i = K$, (where K is the constraint length and is equal to the number of memory plus one), compute the partial path metric

$$\mathrm{M}\left([r \mid c]_{i-1}\right) = \sum_{l=1}^{i-1} \mu\left(r_l \mid c_l\right)$$

   for the single path entering each state. Store the path (called the survivor) and its metric for each state. Note that r is the received message from the channel, c is the code sequence that the decoder concludes has been sent, and μ corresponds to the branch metric function.

2. Increase i by 1. Compute the partial path metric for all paths entering each state by adding the branch metric (i.e., the number of bits in which received message differs from code sequence) entering that state to the partial path metric of the corresponding survivor at the previous time unit. For each state, store the path with the smallest partial path metric (i.e. the survivor) together with its metric, and eliminate all other paths.

3. If $i \leq Nc$, repeat step 2. Otherwise, stop. (where Nc is the number of bits in a message sequence)

Figure 3 shows the trellis diagram and the result of Viterbi algorithm for the encoder shown in Figure 2 with received message of 101 101 101 111 001 110 110. The underlined number at each state shows the partial path metric of the survivor path at that state. Branches colored red represent eliminated paths. The final survivor path, shown in bold lines in the trellis diagram, has a path metric of 7, and corresponds to the decoded message sequence of 11001.

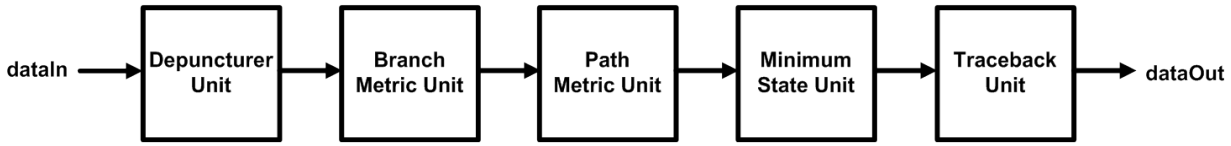

Figure 4: Block Diagram of Viterbi Decoder

Continuous Decoding

Since a final decision on the maximum likelihood path is not made until the entire received sequence has arrived, this may cause an unacceptably long decoding delay if the message length is long. One practical alternative is to pick a fix delay l and at level j make a desision on the information block at level $j-l$ based on the best survivor path at level j. Sliding window with length of $5K$ will result in near optimal decoding.

Methodology

High-Level Design

The rate $1/2$ viterbi decoder has five main components as shown in Figure 4. The branch metric unit (BMU) calculates the branch metrics, which are the normed distances between all possible symbols in the code alphabet and the received symbol. The path metric unit (PMU), whose core elements are add-compare-select (ACS) units, performs calculations on the branch metrics to get metrics for 2^K paths and selects the 2^{K-1} surviving branches based on the branch metrics. Note that the interconnects between ACS units depend on the specific code's trellis diagram. Finally, the traceback decoding unit (TBU) generates the decoded data bits.

In order to support a higher rate codes derived from a basic rate $1/2$ code by using puncturing, a depuncturer is added to the viterbi decoder. The depuncturer takes the input bitstream and inserts erasure bits where bits have been deleted before. Note that these erasure bits do not contribute to the result of the branch metric unit.

Microarchitectural Description

Depuncture Unit

The system works for a range of code rates such as $1/2, 2/3, 3/4, 5/6$, based on a rate of $1/2$. The higher code rates can be derived from the $1/2$ code by simply introducing puncturing. On the encoder side, the puncturing operation deletes certain bits from the encoded stream according to the puncture matrix. On the decoder side, the depuncture operation inserts the erasure bits at the places where bits were deleted by the puncturer. The standard puncturing matrices for the rate $1/2$ convolutional codes are used and are summarized in Table 1.

Since we always input two bits to the decoder, we can group the two consequent masks together. In this case the matrix for $5/6$ can be defined as a 5-element vector of 2-bit values (i.e. MaskG[0], MaskG[1], ... MaskG[4]). Note that mask matrices for all other rates are submatrix of matrix for $5/6$ so we will be able to

Rate	Puncturing Matrix
1/2	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$
2/3	$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$
3/4	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$
5/6	$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$

Table 1: Standard Puncturing Matrix for Rate 1/2 Convolutional Code

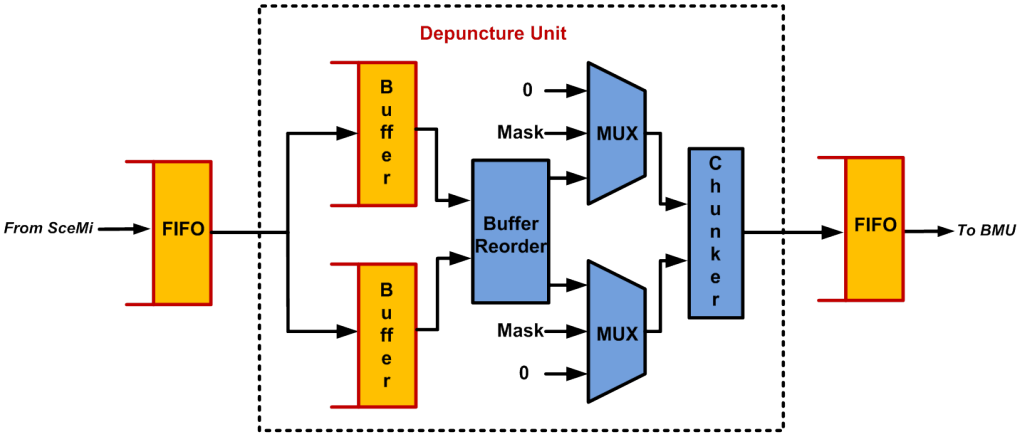

Figure 5: Microarchitecture of the Depuncture Unit

generate the matrix for each of those rate by knowing the mask matrix for rate 5/6. For example, the mask matrix for rate 2/3 are MaskG[0] and MaskG[1]. The output of the depuncturer is passed to the branch metric unit (BMU) to calculates the Hamming distance between the received codeword and all possible symbols in the code alphabet. Figure 5 shows the detailed diagram for the Depuncture Unit.

Inputs Code3 via Server sub-interface, Bits#(4) via Put sub-interface

Message type Code3 (3-bit) received message from Sce-Mi interface; First bit represent Reset signal in our system and two other bits are received message(Code).

Rate sets the rate of the decoder. Must be 2 for rate 1/2, 4 for rate 2/3, 6 for rate 3/4, and 10 for rate 5/6. Invalid rates are discarded without changing the system's current rate.

Outputs Maybe#(Tuple2#(Code2, Code2)) via Server sub-interface

Valid represents the reset signal in our signal; propagate it down the viterbi pipeline.

Message type Code2 (2-bit) received message from the channel. Used to calculate Hamming Distance against all possible codewords.

Mask type Code2 (2-bit) puncturing mask. Used to remove erasure bits' contribution in the branch metric calculation.

StateVariables

Index stores the previous MaskG index values. At initialization and reset, it will be initialized to 0 (indicating first entry in MaskG). Index will always stay zero for rate 1/2 but it will be incremented and reset when it reaches 1, 2 and 4 for rates 2/3, 3/4 and 5/6 respectively.

Rate stores the current rate of the decoder. At initilization, it is unset, and the decoder does not begin decoding until a valid rate is given.

ErrorHandling None. The depuncturer assume that the transmitter sends multiples of n bits. If not, up to 1 bit of received message may be discarded without going through the decoder.

Assumptions: Since we are designing a decoder and we would like to examine the maximum throughput of our decoder, we assumed that the input rate is high enough and it is not limiting the throughput of our design. We made the assumption that the receiver is receiving bits at a high rate and putting them into two buffers (Internal buffers) much faster than what our decoder can take them in. In this case, at each cycle, we look at the corresponding MaskG (two consequent mask). If it is 11, we took 1 bit from each internalbuffer and deq them to the out put. If the MaskG is 01 or 10, we only take one bit from one of the internal buffer and deq it to the output after adding an erasure bit to it.

Branch Metric Unit (BMU)

The branch metric unit (BMU) calculates the Hamming distance between the received codeword and all possible symbols in the code alphabet. Each Hamming distance block performs an XOR between the received codeword and the hard-coded symbol fixed at compile time and returns the number of ones in the result as the Hamming distance via a lookup table. The results are stored in a vector and passed to the path metric unit (PMU). The detailed diagram of the BMU is shown in Figure 6.

Note that for punctured codes, the erasure mask is used after the XOR to remove the erased bit's contribution in the BM before we sum up the number of ones as the Hamming distance.

Inputs Maybe#(Tuple2#(Code2, Code2)) via Server Interface

Valid represents the reset signal in our system. Since there are no state variables in the BMU, it is simply propagated down the viterbi pipeline.

Message type Code2 (2-bit) received message from the channel. Used to calculate Hamming Distance against all possible codewords.

Mask type Code2 (2-bit) puncturing mask. Used to remove erasure bits' contribution in the branch metric calculation.

Outputs Maybe#(BranchMetricVector) via Server Interface

Valid represents the reset signal in our system; propagate it down the viterbi pipeline.

BranchMetricVector 4-element vector of 2-bit hamming distance results.

Path Metric Unit (PMU)

The path metric unit (PMU) uses add-compare-select (ACS) cores to calculate and select the surviving branches based on the previously stored path metrics and the received branch metrics (BM).

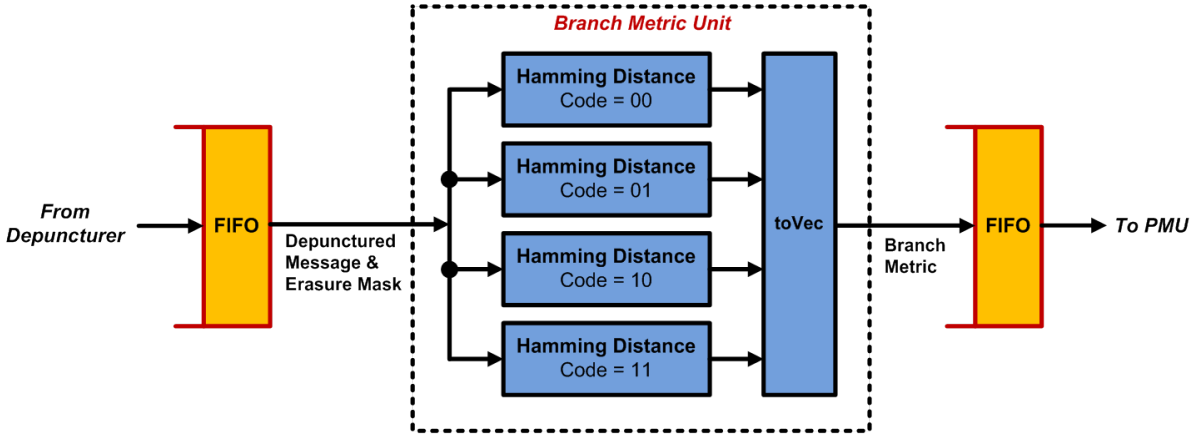

Figure 6: Microarchitecture of the Branch Metric Unit (BMU)

One registered vector of size 2^{K-1} is defined to store the path metrics (PM) for previous states. Each element of this vector is a Maybe type, containing the value of the path metric up to the previous state. A second temporary vector of size 2^{K-1} by 2 is defined to hold the result of the ACS, which contains the smaller PMs out of the two possible previous states as well as the previous state (survivor) that resulted in the smaller PM.

Two helper functions are written in Bluespec to generate lookup tables to simplify the task of implementing the PMU. The first function is the previous state function, which returns the two possible previous states for the current state of interest. This allows static, at compile time, generation of interconnects from the stored PM registers to the ACS units, each corresponding to a specific current state. The second helper function is the encoder output function, which returns the encoder output bits given the previous state and the current state. This again enables static generation of the necessary interconnects from the BM inputs to the ACS units. Figure 7 shows the detailed diagram for the PMU.

Note that if both PMs from the two possible previous states are tagged invalid, then the current state will be tagged invalid. If one of the PMs from the two possible previous states is tagged invalid, then the valid path is automatically taken for calculating the new PM. Finally, if both are tagged valid, then the normal add-compare-selct is performed, where the two possible PMs are updated with the received BMs, and the smaller one is selected as the new PM. The new PM and its previous state will be passed to the minimum state unit. Finally, the new PM will be written back to the registered vector at the end of the clock cycle.

In summary, he path metric unit consists 2^{K-1} ACS units with interconnects generated statically. Since PM for each state depends on the PM for the previous state, pipelining is not applicable for this block.

Inputs Maybe#(BranchMetricVector) via Server Interface

Valid represents the reset signal in our system. When Invalid, the previous path metrics are reset.

BranchMetricVector 4-element vector of 2-bit hamming distance results.

Outputs Maybe#(PathSampleVector) via Server Interface

Valid represents the reset signal in our system; propagate it down the viterbi pipeline.

PathSampleVector 64-element vector of Maybe#(Tuple2(State, PathMetric)). Each element represents a state and contains the state's minimum path metric value (8-bit) and the previous state

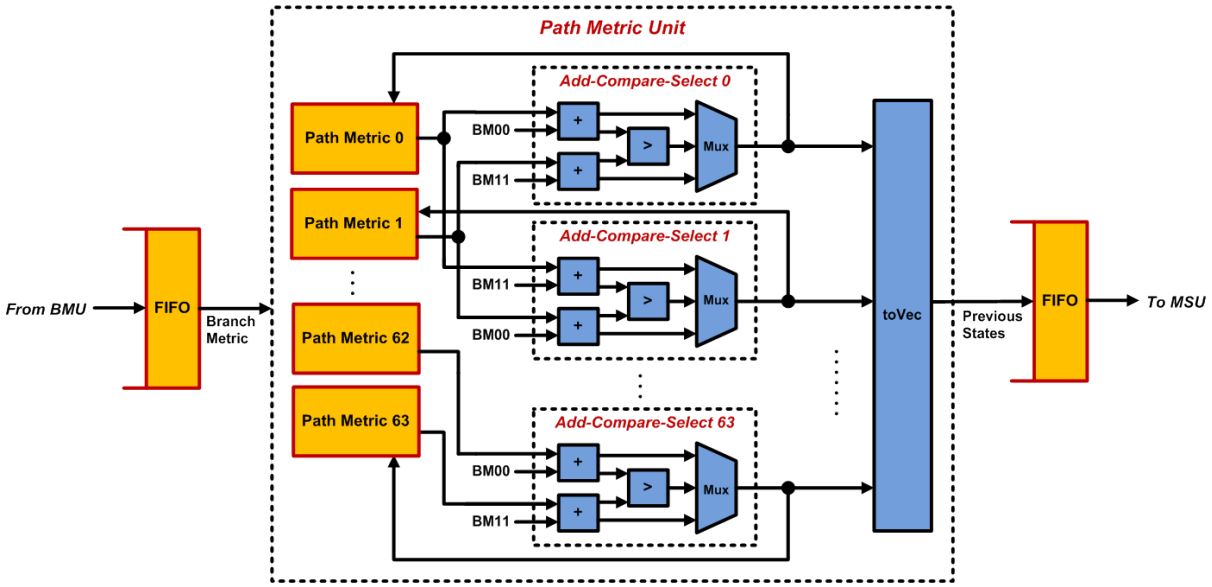

Figure 7: Microarchitecture of the Path Metric Unit (PMU)

(6-bit) that led to the current minimum path metric. Note that at initialization, only state 0 is a valid state, and more states become valid as we traverse down the trellis in the forward direction; Maybe type is used to capture this.

StateVariables

prevPM stores the previous path metric values. It a 64-element of Maybe type PathMetric (8-bit). At initialization and reset, state 0 is initialized to be valid with PM of 0 while all other states are tagged Invalid.

Assumption In the process of Add, Compare, and Select (ACS), when the two path metrics are the same, priority is given to the previous state with a leading 0. i.e., previous state xxxxx0 has priority over previous state xxxxx1. This is chosen so that the result would match MATLAB's vit.c C-MEX implementation.

Minimum State Unit (MSU)

In order avoid introducing additional delay in either the PMU or the TBU, the job of searching for the current state with the minimum path metric has been given to a dedicated module. The minimum state unit (MSU) has been implemented and inserted between the PMU and the TBU. The MSU takes the output PMs from the PMU and finds the state with minimum PM to output to the TBU. In addition, the MSU also has to pass the previous state vector from the PMU to the TBU. Since the MSU has to find the minimum state out of 64, a 6-stage is used with pairwise comparison in all stages. Any state with an invalid PM is automatically assigned the maximum possible PM (255 in 8-bit case), so they are out of the running for the minimum state.

Inputs Maybe#(PathSampleVector) via Server Interface

Valid represents the reset signal in our system. Since there are no state variables in the MSU, it is simply propagated down the viterbi pipeline.

PathSampleVector 64-element vector of Maybe#(Tuple2(State, PathMetric)). Each element represents a state and contains the state's minimum path metric value (8-bit) and the previous state (6-bit) that led to the current minimum path metric.

Outputs Tuple3#(State, StateSampleVector, Reset) via Server Interface

State returns the result of the minimum state unit.

StateSampleVector 64-element vector of State (6-bit). Each element represents a state and contains previous state that led to current state from the result of ACS. This is generated by stripping the PathMetric in the input PathSampleVector since TBU does not need that information given the minState is already calculated.

Reset propagate reset down to the TBU. A value of 1 would begin reset sequence for TBU.

Assumption In the process of searching for the state with minimum path metric, when the two path metrics are the same, priority is given to the state with a higher index, i.e. state 63 has the highest priority and state 0 has the lowest priority. This is chosen so that the result would match MATLAB's vit.c C-MEX implementation. Note that this is not the optimal choice, as state 63 represents the unlikely long sequence of 1's from the input. But this is convenient as verification can be done by directly comparing the output to MATLAB's decoded output.

Traceback Unit (TBU)

The traceback unit (TBU) estimates the sequence of the original message by tracing back the decisions made by the PMU. It performs the traceback of depth 1 at each cycle, indicating that it takes $5K$ cycles to perform the traceback of depth $5K$. To perform the traceback, a state history table of size 2^{K-1} by $10K$ is generated by storing the vector of survived previous states sent from the MSU. The reason why the size of the table is $10K$ while the depth of traceback is $5K$ is that it takes $5K$ cycles to traceback, so that each element should remain in the table for $5K$ more cycles. Note that whenever a new input comes, the whole table is shifted and the new input is stored as the first element, whereas the last input is discarded.

When the traceback of depth $5K$ is completed, the traceback unit uses the last two states to generate 1-bit decoded output. To this end, the traceback unit looks up the input table, which determines the input of the convolutional encoder given current state and next state. The dimensions of the table is 2^{K-1} by 2^{K-1}. It should be noted that the input table is statically generated at the beginning, whereas the history table is dynamically generated and updated during the traceback. Figure 8 shows the detailed diagram of the TBU.

Inputs Tuple3#(State, StateSampleVector, Reset) via Sserver Interface

State stores the result of the minimum state unit. Tracing back starts from this state.

StateSampleVector 64-element vector of State (6-bit). Each element represents a state and contains previous state that led to current state from the result of ACS.

Reset represents the reset signal in our system. A value of 1 would reset count to 0 and flush the last 36 outputs.

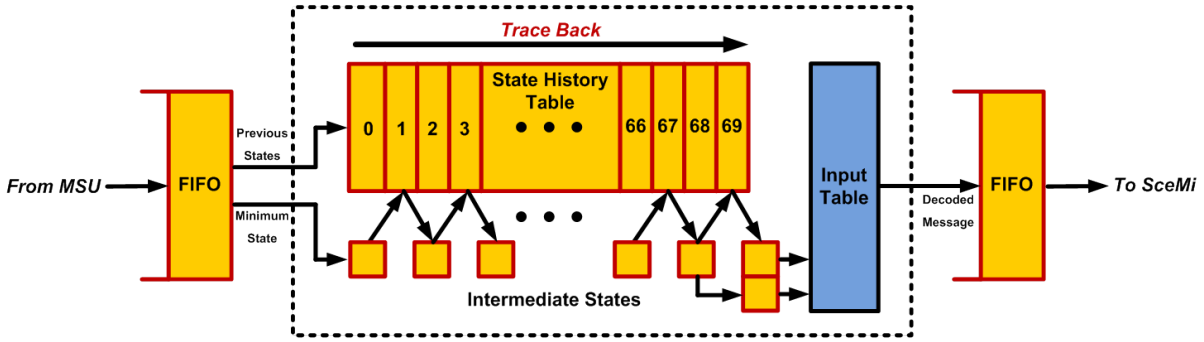

Figure 8: Microarchitecture of the Traceback Unit (TBU)

Outputs DecodedMsg via Server Interface

DecodedMsg: 1-bit output decoded by tracing back 35 StateSampleVectors and looking up the input table with the last two States.

StateVariables

historytable: 70-element vector of StateSampleVector, through which the traceback of depth 35 propagates. Each element of the vector represents a 64-element vector of the previous state that led to current state from the result of ACS. Whenever a new input arrives, every element in the table is shifted. The new input is stored as the first element, whereas the last element is discarded.

middlestate: stores intermediate states during the traceback. Since tracing back for generating 1-bit decoded output takes 35 cycles, 34 intermediate states need to be stored.

count: stores how many elements are stored in the history table. It counts up to 71 and stops increasing, indicating the table is full and it is ready to output a decoded message. At initialization and reset, count is set to be 0.

checkflush,endcount: when reset signal is received, checkflush is set to be 1 and the traceback unit starts to flush the last 36 outputs. While endcount is counting from 0 to 35, the traceback unit performs the same tracing back operation except that there is no new input. When flushing is completed, checkflush returns back to 0 and the traceback unit waits for a valid input. Endcount is reset to 0 when a valid input comes so as to prevent the traceback unit from flushing the last 36 outputs every time reset signal is received.

Interconnection

Just as general pipelined architectures, the interfaces between DPU, BMU, PMU, MSU, and TBU all use standard Get and Put interfaces from Server interface.

Design Exploration

Since the 8-bit path metrics are always increasing, there is potential for overflow for long messages with many errors. Thus, a reset scheme has been implemented. One easy way to implement such function is to

	Viterbi Decoder w/ 7-Stage TBU	Viterbi Decoder w/ 35-Stage TBU
Number of Slice Registers	28%	38%
Number of Slice LUTs	21%	23%
Critical Path Module	TBU	MSU
Clock Frequency	72.020MHz	150.400MHz

Table 2: Complete Viterbi Decoder Synthesis Results after Pipelining TBU

set a threshold such that when all the stored PMs pass the threshold, the threshold value is subtracted from all the PMs, thereby returning the minimum PM to 0. To avoid having to perform comparisons and increase critical path delay, we set the threshold using only the MSB. A new rule is added in the PMU to perform the reset when all MSBs of the stored PM values are 1. The reset action is simply resetting all the MSBs to 0 (i.e. if all the stored PM values are greater than 128, then 128 is subtracted from all of them).

The traceback unit has also been pipelined. The initial version performed combinational logic minimum state search and all $5K$ traceback lookup chain in one clock cycle. As a result, the traceback was the critical path and limited the maximum clock frequency to 10.48MHz. In order to increase the clock frequency, a pipelined minimum state unit (MSU) has been implemented. The MSU takes the output PMs from the PMU and finds the state with minimum PM to output to the TBU. In addition, the MSU also has to pass the previous state vector from the PMU to the TBU.

With the MSU, the TBU no longer needs knowledge of the path metrics. Thus the TBU's has been updated so that the state history table only stores the previous state vectors. This greatly reduced the size of the state history table and decreased the compile time significantly. The traceback function has also been pipelined into 7 and 35 stages. So in each clock cycle, the TBU traces back 5 and 1 entries respectively.

Design Verification

Initially, to ensure correct functionality of each block, small Bluespec testbenches are written to verify simple test cases with known outputs given short input vectors. This allows us to confirm correctness of each block before connecting them together to build the full decoder. After block-wise correctness is verified, we connect all the modules together to perform top-level testing by comparing the outputs of the Bluespec implementation against the outputs of the reference software implementation. MATLAB functions are written to encode randomly generated message into a convolutional code . The code is written to an input file and imported to the Bluespec implementation using the TestDriver module. The TestDriver then writes the output of the Viterbi Decoder into an output file, and the output can be compared to the original message to verify correctness.

After system correctness is verified for uncorrupted code, more complicated testing has been done to verify its error-correcting capabilities. We simulate AWGN channel with different SNRs in MATLAB to corrupt our coded bits. The corrupted code are then fed to both the Bluespec implementation and the Viterbi Decoder in the MATLAB Communications Toolbox for decoding. Bit-by-bit comparison has been performed to verify correct functionality. Finally, Sce-Mi interface has been implemented to verify that our design works both in simulation and on the FPGA for all proposed rates.

To summarize, the testbench contains 4 main blocks: random data generator, convolutional encoder, channel model, and Viterbi decoder. The random data generator generates random messages to be encoded by the convolutional encoder. These encoded messages are then corrupted by passing through an additive

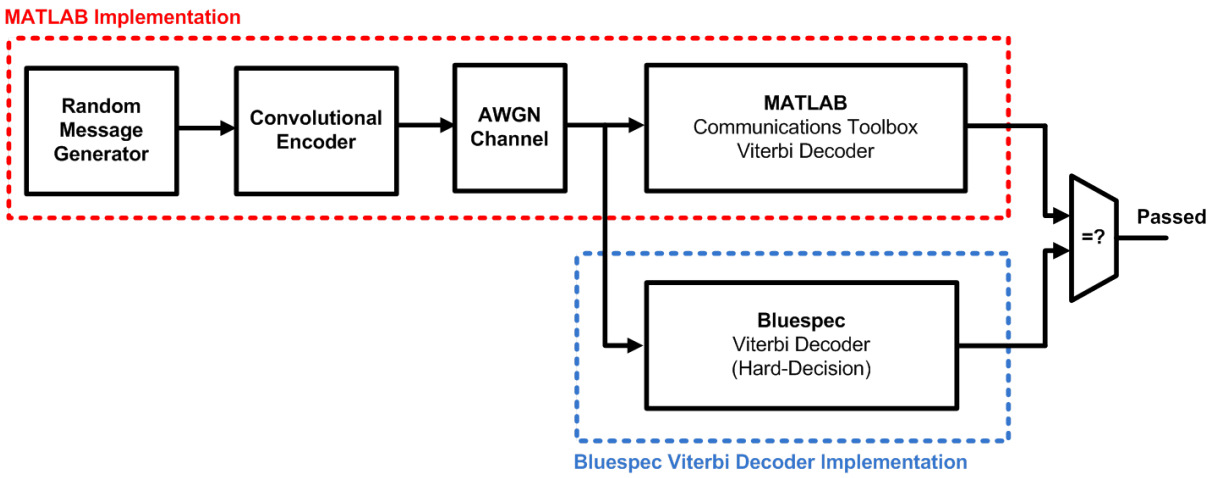

Figure 9: Testbench Setup

white gaussian noise (AWGN) channel model before going to both the reference implementation and our Bluespec implementation of the Viterbi decoders. Both write their output vectors to files so we can compare them to verify correctness. Figure 9 shows the testbench block diagram as described above.

Performance Benchmark

In the first fully functional implementation, a single-bit in and single-bit out interface was used. However the input rate severely limited the throughput of the decoder since it is not supplying bits as fast as the decoder can process. In the rate 1/2 case, the decoder can calculate 1 output bit from every 2 input bits; if input is 1-bit per cycle, the viterbi pipeline is then idling every other cycle.

Thus, to push our decoder to its throughput limit, input is chosen to be 2 bits by assuming the receiver can receive and buffer inputs much faster than the viterbi decoder can process. Note that even though 2 bits are needed for every output for rate 1/2 code, sometimes only 1 bit is needed since erasure bit is inserted to reconstruct a 2-bit message for higher rates codes. For example, in the rate 2/3 case, in the first clock cycle the mask is 11, so 2 bits are needed. But in the second cycle the mask is 01, so only 1 bit is needed and erasure bit is inserted. Therefore, a flexible FIFO interface is implemented to allow the decoder to request either 1 or 2 bits every cycle.

A separate performance benchmark is set up in Bluespec without using the Sce-Mi interface as the Sce-Mi interface cannot input bits from a file fast enough. The block diagram for the performance benchmark is shown in Figure 10. A message generator is implemented in Bluespec using the LFSR module to generate pseudo-random message bits. Two counters are used in the benchmark to characterize the performance: latency counter and cycle counter. Both counters start counting up each cycle once the LFSR has been seeded. The latency will count up until the first output comes out of the viterbi decoder while the cycle counter will count up until the last output comes out of the viterbi decoder.

To summarize the result of the performance benchmark, the latency is fixed at 88 cycles, as determined by the number of pipeline stages present in the system. The cycle count is equal to the latency plus the number of outputs, which translate to one output bit per cycle at steady state. As such, we conclude that our decoder can maintain an output throughput of 150Mb/s at 150MHz, which is the highest achievable

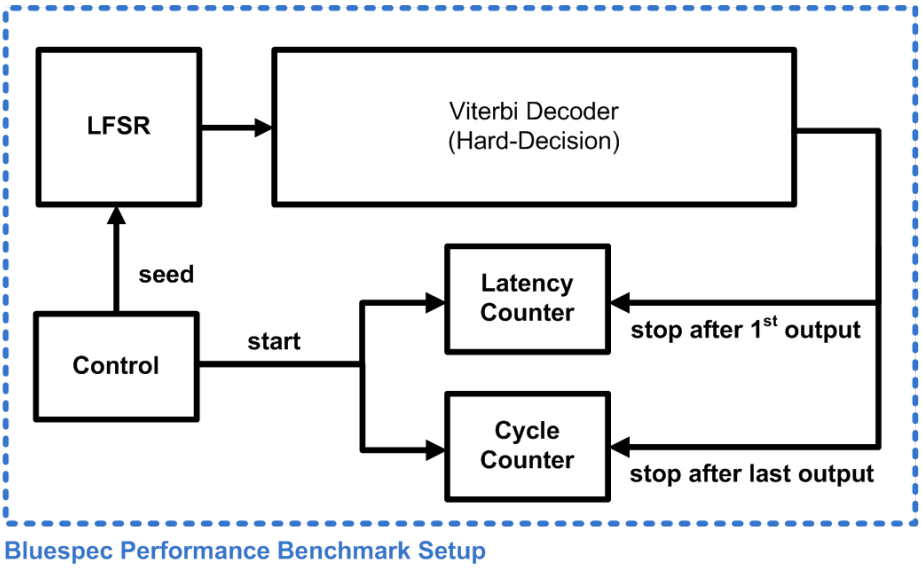

Figure 10: Performance Benchmark Setup

clock frequency from FPGA synthesis result on a Xilinx Virtex 5.

Lastly, we compare the hardware throughput to that of a software implementation. Using the C-MEX function in MATLAB, it takes more than 4 seconds to decode 1.5Mb for all rates on a PC with 2 Cores at 2.8GHz Frequency. This means that decoding 150Mb would take more than 400 seconds. Thus, our hardware implementation of the Viterbi Decoder is more than 400× faster than the software implementation.

References

[1] Viterbi, A.;, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol.13, no.2, pp. 260-269, April 1967

[2] B. Sklar, Digital Communications: Fundamentals and Applications, 2nd edition, Prentice-Hall, 2001.

[3] John G. Proakis, Digital Communications, 5th edition, McGraw-Hill, 2008.

[4] Pedroni, B.U.; Pedroni, V.A.; Souza, R.D.;, "Hardware implementation of a Viterbi decoder using the minimal trellis," *Communications, Control and Signal Processing (ISCCSP)*, 2010 4th International Symposium on, vol.,no., pp.1-4 3-4 March 2010.

[5] http://en.wikipedia.org/wiki/Viterbi_decoder

[6] http://www.dsplog.com/2009/01/04/viterbi/

[7] Lattice Semiconductor Corporation, "Block Viterbi Decoder," Datasheet, Oct. 2004.