ISSN: 2248-9622 www.ijera.com Vol. 2, Issue 3, May-Jun 2012, pp.2849-2861

PERFORMANCE EVALUATION OF TURBO CODES USING HARD DECISION VITERBI ALGORITHM IN VHDL

SUSRUTHA BABU SUKHAVASI[1*], SUPARSHYA BABU SUKHAVASI[1]
DR.HABIBULLA KHAN[2] , CHIRANJEEVI PILLA[3]

1*.ASSISTANT PROFESSOR, Department of ECE, K L University, Guntur, AP, India
1. ASSISTANT PROFESSOR, Department of ECE, K L University, Guntur, AP, India
2. PROFESSOR & HEAD, Department of ECE, K L University, Guntur, AP, India
3. M.TECH-VLSI STUDENT Department of ECE, K L University, Guntur, AP, India.

Abstract:
We first briefly describe the algorithms of convolutional encoder and hard decision Viterbi decoder. The focus of this work is towards developing an application specific design methodology for low power solutions. The methodology starts from high level models which can be used for hardware solution and proceeds towards high performance hardware solutions. The methodology starts from algorithmic level, concentrating on the functional correctness rather than on implementation architecture. The effect on performance due to variation in parameters like frame length, number of iterations, type of encoding scheme. Turbo codes are used for error protection, especially in wireless systems. Viterbi algorithm is widely used as a decoding technique for convolutional codes as well as a bit detection method in storage devices. The design space for VLSI implementation of Viterbi decoders is huge, involving choices of throughput, latency, area, and power. Even for a fixed set of parameters like constraint length, encoder polynomials and trace-back depth, the task of designing a Viterbi decoder is quite complex and requires significant effort. Sometimes, due to incomplete design space exploration or incorrect analysis, a suboptimal design is chosen. This work analyzes the design complexity by applying most of the known VLSI implementation techniques for hard-decision Viterbi decoding to a different set of code parameters.

1. Introduction

Convolutional codes represent one technique within the general class of channel codes. Channel codes (also called error-correction codes) permit reliable communication of an information sequence over a channel that adds noise, introduces bit errors, or otherwise distorts the transmitted signal. These codes have found many applications, including deep-space communications and voice band modems. Convolutional codes continue to play a role in low-latency applications such as speech transmission and as constituent codes in Turbo codes. One way to reduce the transmission power is to incorporate powerful forward error correction (FEC) codes to increase coding gain at the receiver which translates in less transmission power. However, Shannon showed that the development of error correction techniques with increasing coding gain have a limit, arising from
the channel capacity. Since that work, many different types of codes have been designed and their decoding algorithms are physically realized. They mainly differ in decoding performance and their hardware complexity. Traditionally, the Viterbi algorithm has been widely accepted as a choice for decoder for wireless communications because it is an optimum decoding algorithm for the convolutional-code. It performs a maximum likelihood (ML) detection of the state sequence of a finite-state discrete-time Markov process observed in memory less noise. It can also be interpreted as searching for the minimum-distance path in a trellis by dynamic programming, where the measure of distance is the log-likelihood of the corresponding state transition based on the symbols received over noisy channel.

One of the primary objectives in the design of low-energy communication system is power reduction. Power consumption is the guiding principle for both algorithm development and system trade-off evaluation. The tremendous savings in power consumption can be attained through both algorithm reformulation and architectural innovation specifically targeted for energy conservation. We first review turbo-codes. The encoding scheme and the decoding algorithm are briefly described. Then, the low-complexity sub-optimal decoding algorithm is reformulated where many complex operations have been eliminated. A simple mechanism for channel estimation is discussed. Based on the reduced complexity decoding algorithm, the VLSI multistage pipeline turbo-code decoder architecture design is presented. A further circuit complexity is minimized by appropriately choosing the finite quantization word length. Finally, the performance and the circuit complexity of the decoder are evaluated. The low complexity turbo-code decoder is compared with various.

2. Literature Survey:

Viterbi algorithm is a well-known maximum-likelihood algorithm for decoding of convolutional codes. In this article this algorithm is described using the approach suitable both for hardware and software implementations.

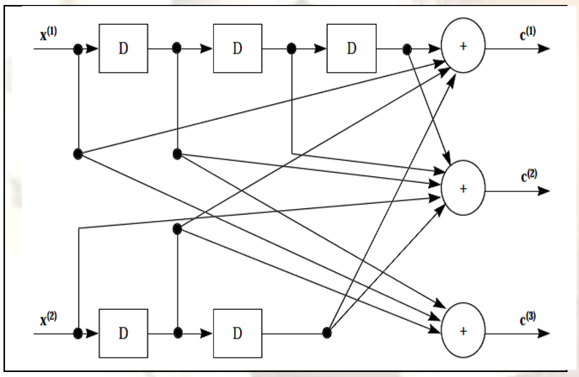
Convolutional Codes:

This chapter describes the encoder and decoder structures for convolutional codes. The encoder will be represented in many different but equivalent ways.

Also, the main decoding strategy for convolutional codes, based on the Viterbi Algorithm, will be described. A firm understanding of convolutional codes is an important prerequisite to the understanding of turbo codes.

2.1 Encoder Structure:

A convolutional code introduces redundant bits into the data stream through the use of linear shift registers as shown in Figure



The information bits are input into shift registers and the output encoded bits are obtained by modulo-2 addition of the input information bits and the contents of the shift registers. The connections to the modulo-2 adders were developed heuristically with no algebraic or combinatorial foundation. Convolutional codes are frequently used to correct errors in noisy channels. They have rather good correcting capability and perform well even on very bad channels. Convolutional codes are extensively used in satellite communications. Although Convolutional encoding is a simple procedure, decoding of a Convolutional code is much more complex task. Several classes of algorithms exist for this purpose:

1. Threshold decoding is the simplest of them, but it can be successfully applied only to the specific classes of Convolutional codes. It is also far from optimal.

2. Sequential decoding is a class of algorithms performing much better than threshold algorithms. Their serious advantage is that decoding complexity is virtually independent from the length of the particular code. Although sequential algorithms are also suboptimal, they are successfully used with very long codes, where no other algorithm can be acceptable. The main drawback of sequential decoding is unpredictable decoding latency.

3. Viterbi decoding is an optimal (in a maximum-likelihood sense) algorithm for decoding of a Convolutional code. Its main drawback is that the decoding complexity grows exponentially with the code length.

So, it can be utilized only for relatively short codes. There are also soft-output algorithms, like SOVA (Soft Output Viterbi Algorithm) or MAP algorithm, which provide not only a decision, but also an estimate of its reliability. They are used primarily in the decoders of turbo codes and are not discussed in this article.

2.2 Convolutional Codes

2.2.1. Convolutional Encoders

Like any error-correcting code, a convolutional code works by adding some structured redundant information to the user's data and then correcting errors using this information.

A convolutional encoder is a linear system. A binary convolutional encoder can be represented as a shift register. The outputs of the encoder are modulo 2 sums of the values in the certain register's cells. The input to the encoder is either the uuencoded sequence (for non-recursive codes) or the uuencoded sequence added with the values of some register's cells (for recursive codes).

Convolutional codes can be systematic and non-systematic. Systematic codes are those where an unencoded sequence is a part of the output sequence. Systematic codes are almost always recursive; conversely, non-recursive codes are almost always non-systematic. A combination of register's cells that forms one of the output streams (or that is added with the input stream for recursive codes) is defined by a polynomial. Let m be the maximum degree of the polynomials constituting a code, then K=m+1 is a constraint length of the code.

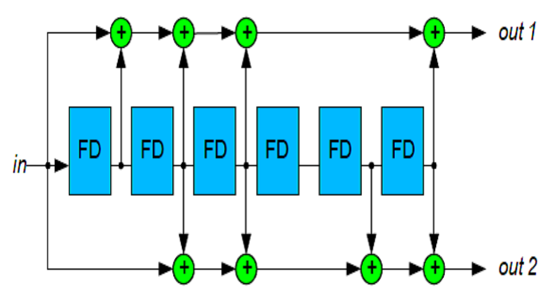


Fig. 1: A standard NASA convolutional encoder with polynomials (171,133)

For example, for the decoder on the Figure 1, the polynomials are:

- $g_1(z) = 1 + z + z^2 + z^3 + z^6$,
- $g_2(z) = 1 + z^2 + z^3 + z^5 + z^6$.

A code rate is an inverse number of output polynomials. For the sake of clarity, in this article we will restrict ourselves to the codes with rate R=1/2. Decoding procedure for other codes is similar. Encoder polynomials are usually denoted in the octal notation. For the above example, these designations are “1111001” = 171 and “1011011” = 133. The constraint length of this code is 7. An example of a recursive convolutional encoder is on the Figure 2.

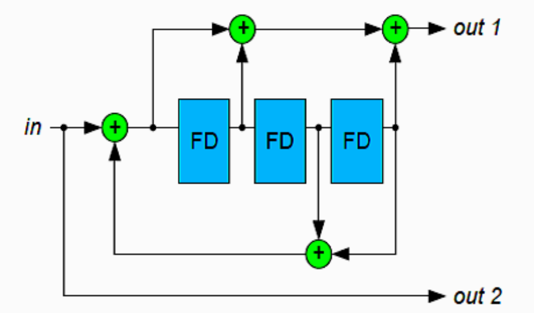


Fig. 2: A recursive convolutional encoder

Hardware Implementation

The parameters chosen in our design are as follows:

1. Constraint Length: K=3: That is the number of shifts (i.e. flip flops) in the linear shift register over which the input data bits have to be shifted before they are taken out of the shift register.

2. Code Rate: k/n = ½: That is for every input message bit, there are two output bits created out of convolution of the impulse response of the linear shift register and the input bit sequence.

3. Minimum Free Distance = 5: That is the minimum Hamming Distance between the codeword sequence and the all zeroes codeword sequence.

4. Trace Back Depth=15: That is the number of paths in Trellis Diagram or the branches in the Tree Diagram, which must be covered before the first input bit can be decoded.

Now we describe the three basic entities used in our VHDL design.

2.2.2. Encoder Representations:

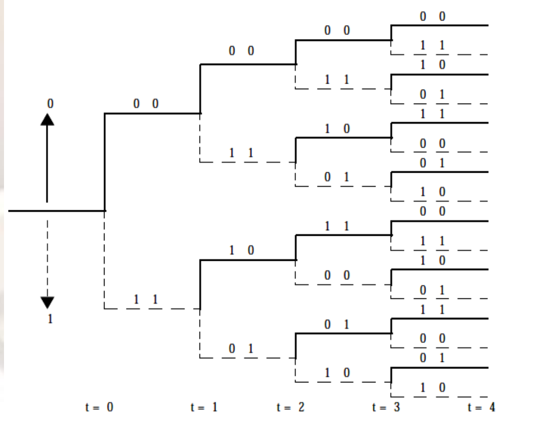
The encoder can be represented in several different but equivalent ways. They are
1. Generator Representation
2. Tree Diagram Representation
3. State Diagram Representation
4. Trellis Diagram Representation

1. Generator Representation

Generator representation shows the hardware connection of the shift register taps to the modulo-2 adders. A generator vector represents the position of the taps for an output. A “1” represents a connection and a “0” represents no connection.

2. Tree Diagram Representation

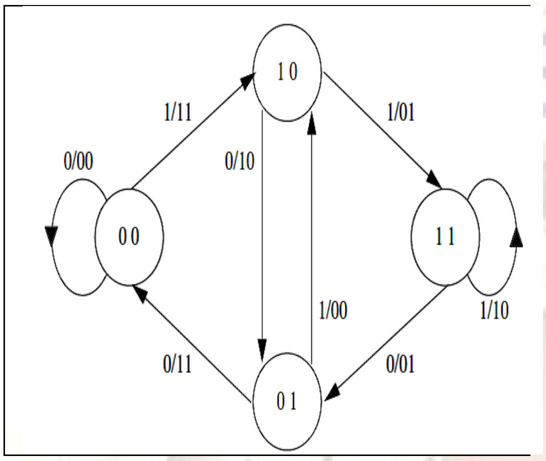
The tree diagram representation shows all possible information and encoded sequences for the convolutional encoder. Figure 2.3 shows the tree diagram for the encoder



In the tree diagram, a solid line represents input information bit 0 and a dashed line represents input information bit 1. The corresponding output encoded bits are shown on the branches of the tree. An input information sequence defines a specific path through the tree diagram from left to right. For example, the input information sequence x= {1011} produces the output encoded sequence c= {11, 10, 00, 01}. Each input information bit corresponds to branching either upward (for input information bit 0) or downward (for input information bit 1) at a tree node.

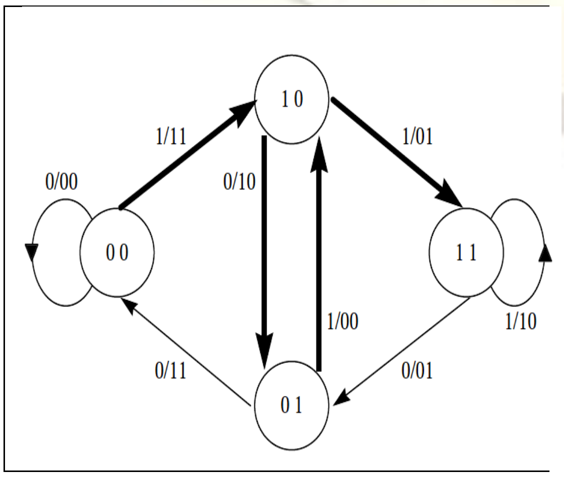
3. State Diagram Representation

The state diagram shows the state information of a convolutional encoder. The state information of a convolutional encoder is stored in the shift registers.



In the state diagram, the state information of the encoder is shown in the circles. Each new input information bit causes a transition from one state to another. The path information between the states, denoted as x/c, represents input information bit x and output encoded bits c. It is customary to begin convolutional encoding from the all zero state. For example, the input information sequence x={1011} (begin from the all zero state) leads to the state transition sequence s={10, 01, 10, 11} and produces the output encoded sequence c={11, 10, 00, 01}. Figure 2.5 shows the path taken through the state diagram for the given example.

Figure 2.5: The state transitions (path) for input information sequence {1011}.



4. Trellis Diagram

A convolutional encoder is often seen as a finite state machine. Each state corresponds to some value of the encoder's register. Given the input bit value, from a certain state the encoder can move to two other states.

These state transitions constitute a diagram which is called a trellis diagram. A trellis diagram for the code on the Figure 2 is depicted on the Figure 3.

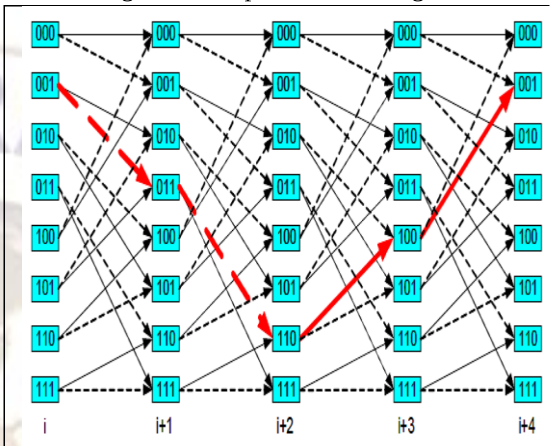


Fig. 3: A trellis diagram corresponding to the encoder on the Figure 2

A solid line corresponds to input 0, a dotted line – to input 1 (note that encoder states are designated in such a way that the rightmost bit is the newest one). Each path on the trellis diagram corresponds to a valid sequence from the encoder's output. Conversely, any valid sequence from the encoder's output can be represented as a path on the trellis diagram. One of the possible paths is denoted as red (as an example).

Note that each state transition on the diagram corresponds to a pair of output bits. There are only two allowed transitions for every state, so there are two allowed pairs of output bits, and the two other pairs are forbidden. If an error occurs, it is very likely that the receiver will get a set of forbidden pairs, which don't constitute a path on the trellis diagram. So, the task of the decoder is to find a path on the trellis diagram which is the closest match to the received sequence.

Let's define a free distance d as a minimal Hamming distance between two different allowed binary sequences (a Hamming distance is defined as a number of differing bits). A free distance is an important property of the convolutional code. It influences a number of closely located errors the decoder is able to correct.

3. Viterbi Algorithm

The Viterbi algorithm as a dynamic programming algorithm for finding the shortest path through a trellis, and the algorithm can be broken down into the following three steps:

• Weigh the trellis; that is, calculate the branch metrics.

• Recursively computes the shortest paths to time n, in terms of the shortest paths to time n-1. In this step, decisions are used to recursively update the survivor path of the signal. This is known as add-compare-select (ACS) recursion.

• Recursively find the shortest path leading to each trellis state using the decisions from Step 2. The shortest path is called the survivor path for that state and the process is referred to as survivor path decode. Finally, if all survivor paths are traced back in time, they merge into a unique path, which is the most likely signal path that we are trying to find.

Associated with each trellis state S at time n is a state metric which is the accumulated metric along the shortest path leading to that state. The state metrics at time n can be recursively calculated in terms of the state metrics of the previous iteration as follows:

PMi+1 =min (PMi +BMi,i+1, PMj+BMj,i+1); (1)
PMj+1 =min (PMi +BMi,j+1, PMj+BMj,j+1); (2)

Where i+1 is a predecessor state of i and BMi,i+1 is the branch metric on the transition from state i to state j. The qualitative interpretation of this expression is as follows. By definition, the shortest path into state j must pass through a predecessor state.

Basic Definitions

Ideally, Viterbi algorithm reconstructs the maximum-likelihood path given the input sequence.

Let's define some terms:

A soft decision decoder:

a decoder receiving bits from the channel with some kind of reliability estimate. Three bits are usually sufficient for this task. Further \ increasing soft decision width will increase performance only slightly while considerably increasing computational difficulty. For example, if we use a 3-bit soft decision, then “000” is the strongest zero, “011” is a weakest zero, “100” is a weakest one and “111” is a strongest one.

A hard decision decoder:

a decoder which receives only bits from the channel (without any reliability estimate). A branch metric – a distance between the received pair of bits and one of the “ideal” pairs (“00”, “01”, “10”, “11”).

A path metric: a sum of metrics of all branches in the path. A meaning of distance in this context depends on the type of the decoder:

• For a hard decision decoder it is a Hamming distance, i.e. a number of differing bits;
• For a soft decision decoder it is an Euclidean distance.

In these terms, the maximum-likelihood path is a path with the minimal path metric. Thus the problem of decoding is equivalent to the problem of finding such a path.

Let's suppose that for every possible encoder state we know a path with minimum metric ending in this state. For any given encoder state there is two (and only two) states from which the encoder can move to that state, and for both of these transitions we know branch metrics. So, there are only two paths ending in any given state on the next step. One of them has lesser metric, it is a survivor path. The other path is dropped as less likely. Thus we know a path with minimum metric on the next step, and the above procedure can be repeated.

3.1. Viterbi Decoder

The receiver can deliver either hard or soft symbols to the Viterbi decoder. A hard symbol is equivalent to a binary +/-1. A soft symbol, on the other hand, is multileveled to represent the confidence in the bit being positive or negative. For instance, if the channel is non-fading and Gaussian, the output of the matched filter quantified to a given number of bits is a suitable soft input. In both cases, 0 is used to represent a punctured bit. In case of hard decision demodulation, data is demodulated into either 1s or 0s, or quantized into two levels only. The process described above makes a hard binary decision about each incoming bit and then uses only the Hamming distances. This simplifiers the hardware, but does not result in optimal perform.

Implementation

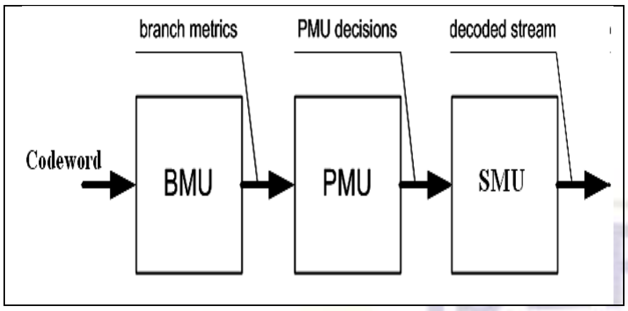
A Viterbi algorithm consists of the following three major parts:

1. Branch metric calculation – calculation of a distance between the input pair of bits and the four possible “ideal” pairs (“00”, “01”, “10”, “11”).

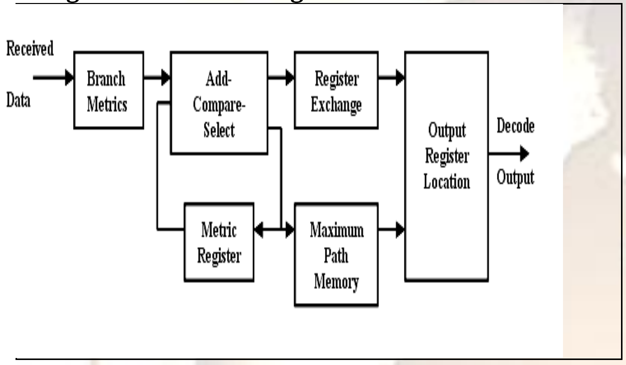
2. Path metric calculation – for every encoder state, calculate a metric for the survivor path ending in this state (a survivor path is a path with the minimum metric).

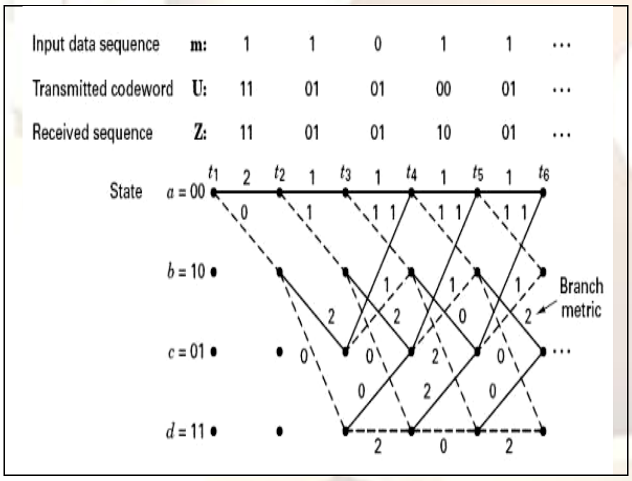
3. Traceback – this step is necessary for hardware implementations that don't store full information

about the survivor paths, but store only one bit decision every time when one survivor path is selected from the two.



The Working of Viterbi decoder in term of block diagram and trellis diagram:




Input data sequence m: 1 1 0 1 1 ...
Transmitted codeword U: 11 01 01 00 01 ...
Received sequence Z: 11 01 01 10 01 ...
State
a = 00
b = 10
c = 01
d = 11
Branch metric

For hard decision decoding, the Viterbi algorithm uses the hamming distance to find the branch metric and path metric.

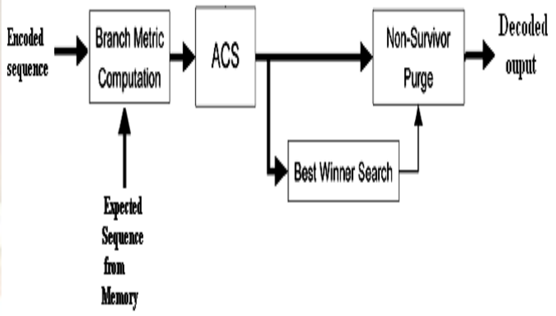
Codeword is given to branch metric unit. Branch metric unit's function is to calculate branch metrics, which are Hamming distances between every possible symbol in the codeword and the received symbol. Path metric unit summarizes branch metrics to get metrics for 2K − 1 path, one of which can eventually be chosen as optimal.

Survivor memory unit can be trace-back process or register exchange method, where the survivor path and the output data are identified. The error probabilities achieved by Viterbi algorithm depends on the code, the rate of the code, its free distance, channel SNR and demodulation Quantized output.

The quality of Viterbi decoder design is mainly measured by three criteria

- Coding gain
- Throughput
- Power dissipation

Adaptive Viterbi Decoder (AVD)

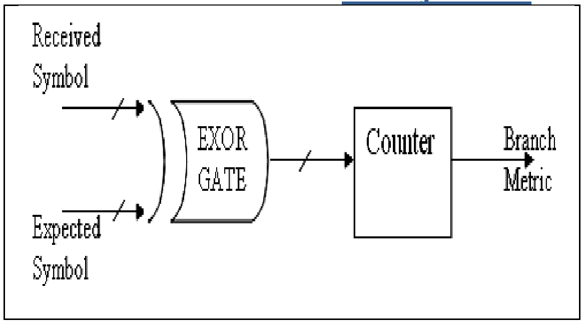


Block diagram of Adaptive Viterbi decoder

Fig shows the data flow diagram of an adaptive Viterbi algorithm, which adds two functional blocks, including the best winner search and non survivor purge, into the original Viterbi algorithm. Codeword is applied to branch metric computation unit. It calculates branch metric by comparing with expected symbol. ACS updates path metric by cumulative accumulation of branch metric. Best winner search determines final winner and give it non survivor purge unit. It deletes all paths expect winner.

The first unit is called branch metric unit BMU is the simplest block in the Viterbi decoder design. Here the received data symbols are compared to the ideal outputs of the encoder from the transmitter and branch metric is calculated. Hamming distance or the Euclidean distance is used for branch metric computation**.**

3.1.1. Branch Metric Calculation

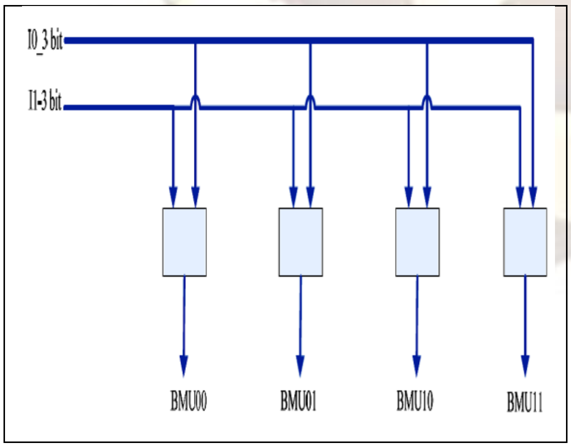


Block diagram of Branch Metric Unit

The BMU calculates the branch metrics from the input data. For hard decision BMU calculate everything in term of hamming distance. Hamming distance between the received Codeword and the expected is calculated by compares the received code symbol with the expected code symbol and counting the number of different bits. BMC (branch metric computation) unit is to calculate the branch metrics which are then moved to the ACS (add compare select) unit.

The major task of the ACS is to calculate the metrics and selected paths. The add-compare-select (ACS) unit recursively accumulates the branch metrics to path metrics for all the incoming paths of each state and selects the path with minimum path metric as the survivor path. An ACS module is shown in Figure2.5.

The two adders compute the partial path metric of each branch, the comparator compares the two partial metrics, and the selector selects an appropriate branch. ACS units determine their own local winners, the best winner search block finds the one having the best (minimum) path metric among all the winners, and the non survivor purge block deletes the local winners Methods of branch metric calculation are different for hard decision and soft decision decoders.



4.5 BMU Block

For a hard decision decoder, a branch metric is a Hamming distance between the received pair of bits and the “ideal” pair. Therefore, a branch metric can take values of 0, 1 and 2. Thus for every input pair we have 4 branch metrics (one for each pair of “ideal” values).

The branch metric uses the Hamming distance for the four possible paths. First we initial four different received hamming distance lookup table. Then each time with check the input symbol, we get the four possible distances.

The BMU perform simple check and select operations on the decision bits to generate the output. The detail hardware implementation is shown in the Figure 4.5.

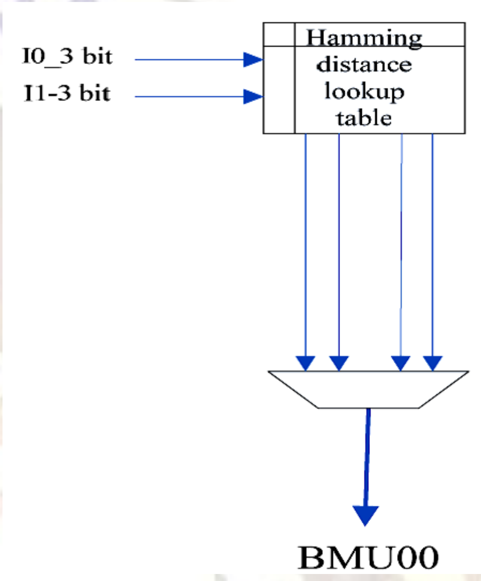


Fig: one possible path hardware implementation example

For a soft decision decoder, a branch metric is measured using the Euclidean distance. Let x be the first received bit in the pair, y – the second, x0 and y0 – the “ideal” values. Then branch metric is

$$M_b=(x-x_0)^2+(y-y_0)^2$$

Furthermore, when we calculate 4 branch metric for a soft decision decoder, we don't actually need to know absolute metric values – only the difference between them makes sense. So, nothing will change if we subtract one value from the all four branch metrics:

$$M_b=(x^2-2xx_0+x_0^2)+(y^2-2yy_0+y_0^2);$$

$$M_b^*=M_b-x^2-y^2=(x_0^2-2xx_0)+(y_0^2-2yy_0).$$

Note that the second formula, Mb * , can be calculated without hardware multiplication: x0 2 and y0 2 can be pre-calculated, and multiplication of x by x0 and y by y0 can be done very easily in hardware given that x0 and y0 are constants. It should be also noted that Mb * is a signed variable and should be calculated in 2's complement format.

Path Metric Calculation

Path metrics are calculated using a procedure called **ACS** (Add-Compare-Select). This procedure is repeated for every encoder state.

1. Add – for a given state, we know two states on the previous step which can move to this state, and the output bit pairs that correspond to these transitions. To calculate new path metrics, we add the previous path metrics with the corresponding branch metrics.

2. Compare, select – we now have two paths, ending in a given state. One of them (with greater metric) is dropped. As there are 2K−1 encoder states, we have 2K−1 survivor paths at any given time. It is important that the difference between two survivor path metrics cannot exceed δlog (K−1), where δ is a difference between maximum and minimum possible branch metrics. The problem with path metrics is that they tend to grow constantly and will eventually overflow. But, since the absolute values of path metric don't actually matter, and the difference between them is limited, a data type with a certain number of bits will be sufficient.

There are two ways of dealing with this problem:

1. Since the absolute values of path metric don't actually matter, we can at any time subtract an identical value from the metric of every path. It is usually done when all path metrics exceed a chosen threshold (in this case the threshold value is subtracted from every path metric). This method is simple, but not very efficient when implemented in hardware.

2. The second approach allows overflow, but uses a sufficient number of bits to be able to detect whether the overflow took place or not. The compare procedure must be modified in this case.

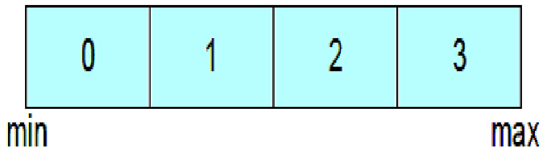
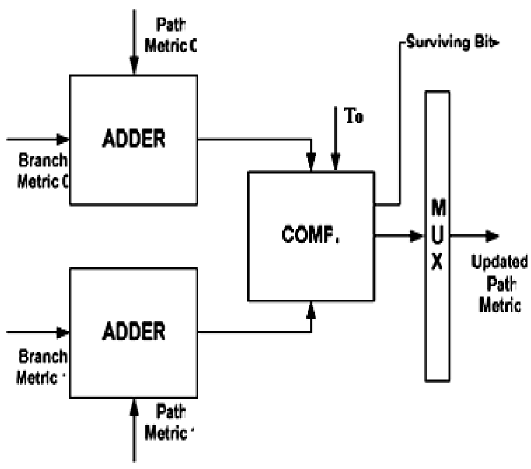


Fig. 5: A modulo-normalization approach

The whole range of the data type's capacity is divided into 4 equal parts. If one path metric is in the 3-rd quarter, and the other – in the 0-th, then the overflow took place and the path in the 3-rd quarter should be selected. In other cases an ordinary compare procedure is applied. This works, because a difference between path metrics can't exceed a threshold value, and the range of path variable is selected such that it is at least two times greater than the threshold.

3.1.2. The ACS block

When the 4 possible input distance is ready, the ACS block' butterfly module adds the results and the related distance value stored in the state metric storage to get the each two paths for the 64 initial states. The butterfly module is shown in the next figure. Since, each butterfly computes 4 possible paths and selects the two smaller distance paths form. We have totally 32 butterflies.



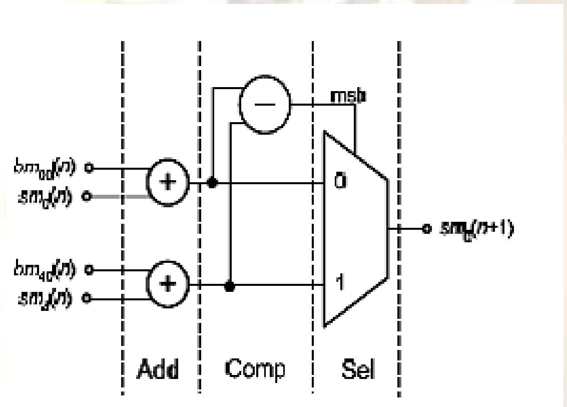
For each nod (state), the ACS module selects a smaller one as the survival path and stores them to the accumulated state metric storage block and the survivor path metric. Following Figure is the ACS module.

PMU is a critical block both in terms of area and throughput. The key problem of the PMU design is the recursive nature of the add-compare-select (ACS) operation The throughput of hard- or soft-output Viterbi decoders is set by the particular target application requirements. Depending on the implementation platform or the complexity limitations, the decoders can be built using concurrent computation of all state metrics or by resource sharing through multiplexing the computational units.

High-throughput applications require the use of fully parallel decoder implementations. The throughput of a SOVA decoder has traditionally been limited by the difficulty of pipelining the single-step ACS recursion. the transition trellis of an example eight-state hard or soft-decision Viterbi decoder. The critical-path of a traditional ACS computation extends through the sequential execution of two parallel additions, a comparison and a selection. Let represent the path metric for state, and, the branch metric of a corresponding transition from state to state , with the time step denoted by . Then, an example of the ACS recursion corresponding to state 0 is shown.

The comparison is implemented through subtraction, and the most significant bit (MSB) of the result selects the winning path. The ripple-carry implementations of both add and compare operations take advantage of the similarity in carry profiles. The amount of overhead in the critical path required for executing the subtraction only involves the computation of the MSB of the difference. Fast adder structures such as the carry-select adder will require the subsequent subtraction to follow an abrupt carry profile, which yields minimal performance gains. With large area penalties. The use of a redundant numbering system with MSB-first computations can provide performance improvement.

However, this is achieved at the expense of large area due to the carry-save representation



ACS Recursive Equations:

$$sm_0(n+1) = \min\left\{\begin{matrix} sm_0(n) + bm_{00}(n) \\ sm_4(n) + bm_{40}(n) \end{matrix}\right\}.$$

Previous high-throughput implementations of the Viterbi decoder unrolled the ACS loop in order to perform two-step iterations of the trellis recursions within a single clock period. These lookahead methods replace the original radix-2 trellis with a radix-4 trellis, at the cost of increased interconnect complexity. A radix-4 ACS computes four sums in parallel followed by a four-way comparison. In order to minimize the critical-path delay, the comparison is realized using six parallel pair-wise subtractions of the four output sums. In general, the critical-path delay increases. However, due to the doubled symbol rate, the effective throughput is improved if this increase in delay is less than twofold. An alternative approach with a lower area overhead is the concurrent ACS. Maintaining the use of a radix-2 trellis, the concurrent ACS performs the addition and comparison operations simultaneously. It requires the comparison to be realized with a four-input adder. A sub-8-ns four-input adder was implemented in 0.6- m CMOS using two layers of three-to-two carry-save adders, followed by a final carry-lookahead adder. The critical path through the four-input adder and a multiplexer determines the throughput of the concurrent ACS.

The concurrent ACS becomes the choice structure for delays between 29 and 35 FO4 delays. For low-throughput rates with critical-path delays above 35 FO4 delays, the ACS structure is the best choice in terms of both area and power consumption. In this high-throughput SOVA decoder implementation, the transformed CSA was implemented because it provided the highest decoding throughput, without incurring the excessive area and power penalties of the radix-4 ACS structure.

Traceback

It has been proven that all survivor paths merge after decoding a sufficiently large block of data (D on Figure 5), i.e. they differ only in their endings and have the common beginning. If we decode a continuous stream of data, we want our decoder to have finite latency. It is obvious that when some part of path at the beginning of the graph belongs to every survivor path, the decoded bits corresponding to this part can be sent to the output. Given the above statement, we can perform the decoding as follows:

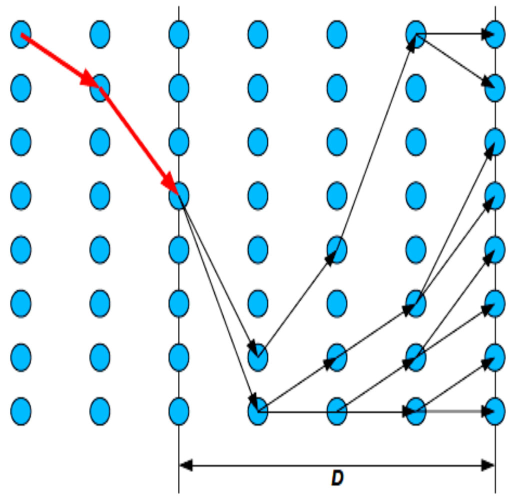


Fig. 6: Survivor paths graph example. Blue circles denote encoder states. It can be seen that all survivor paths have a common beginning (red) and differ only in their endings.

1. Find the survivor paths for N+D input pairs of bits.

2. Trace back from the end of any survivor paths to the beginning.

3. Send N bits to the output.

4. Find the survivor paths for another N pairs of input bits.

5. Go to step 2.

In these procedures D is an important parameter called decoding depth. A decoding depth should be considerably large for quality decoding, no less than 5K. Increasing D decreases the probability of a decoding error, but also increases latency.

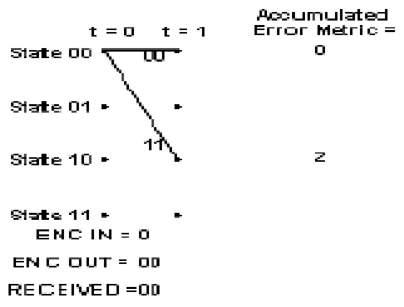
As for N, it specifies how many bits we are sending to the output after each traceback. For example, if N=1, the latency is minimal, but the decoder needs to trace the whole tree every step. It is computationally ineffective. In hardware implementations N usually equals D.

Decoding Procedure:

Each time we receive a pair of channel symbols, we're going to compute a metric to measure the "distance" between what we received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 002, and 112. That's because we know the convolutional encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00 2 and 112. The metric we're going to use for now is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two.

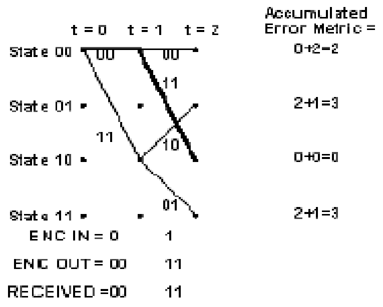
The Hamming distance (or other metric) values we compute at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, we're going to save these results as "accumulated error metric" values, associated with states. For the second time instant on, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics.

At t = 1, we received 002. The only possible channel symbol pairs we could have received are 002 and 112. The Hamming distance between 002 and 002 is zero. The Hamming distance between 002 and 112 is two. Therefore, the branch metric value for the branch from State 002 to State 002 is zero, and for the branch from State 002 to State 102 it'stwo.



Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 002 and for State 102 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. The figure below illustrates the results at t = 1.Note that the solid lines between states at t = 1 and the state at t = 0 illustrate the predecessor-successor relationship between the states at t = 1 and the state at t = 0 respectively. This information is shown graphically in the figure, but is stored numerically in the actual implementation. To be more specific, or maybe clear is a better word, at each time instant t, we will store the number of the predecessor state that led to each of the current states at t. Now let's look what happens at t = 2.

We received a 112 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 002 going from State 002 to State 002, 112 going from State 002 to State 102, 102 going from State 102 to State 01 2, and 012 going from State 102 to State 11 2. The Hamming distance between 002 and 112 is two, between 112 and 112 is zero, and between 10 2 or 012 and 112 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At t = 1, we could only be at State 002 or State 102. The accumulated error metric values associated with those states were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.



That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis. Now look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. So how do we handle that? The answer is, we compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value.

The other thing that's affected is the predecessor successor history we're keeping. For each state, the predecessor that survives is the one with the lower branch metric. If the two accumulated error metrics are equal, some people use a fair coin toss to choose the surviving predecessor state. Others simply pick one of them consistently, i.e. the upper branch or the lower branch. It probably doesn't matter which method you use. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smaller (smallest) accumulated error metric to be 14 retained for the next time instant is called the add-compare-select operation.

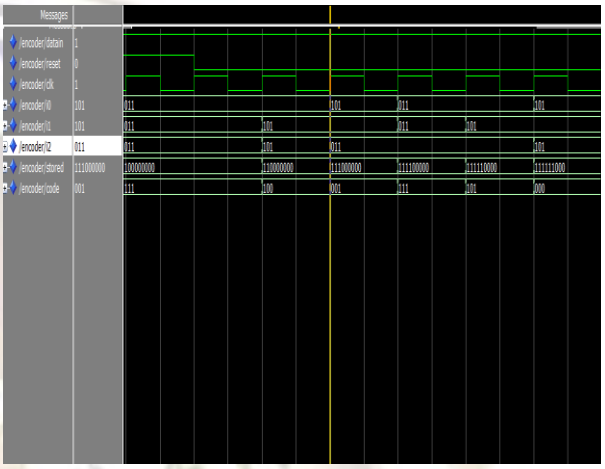
Traceback versus register-exchange approaches in power efficiency

In the traceback approach, each register storing the survivor path information updates its content only once (when it receives the new survivor path information) during the entire period of a code word. In contrast, all the registers in the register-exchange approach update their contents for each code symbol. Hence, the switching activity of the registers in a traceback approach is much lower than that for the registers in a register-exchange approach. Hence, the registers in traceback approach would dissipate less power.

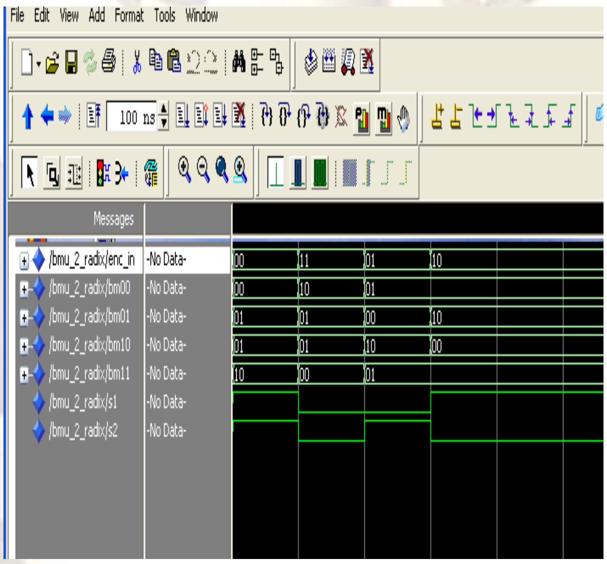
As explained earlier the registers and the traceback module are active only one clock period during the entire period of a code word in the traceback approach. So low power design techniques can be applied readily to the registers and the traceback module as proposed in this chapter. However, the same low-power techniques cannot be applied to the register-exchange approach. Hence, the traceback approach is more desirable for applications in which power dissipation is critical.

Simulation Results:

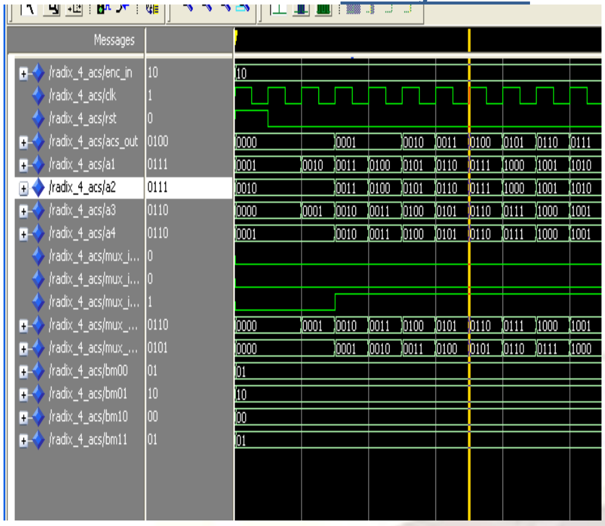
Encoder waveform



Waveform for BMU


File Edit View Add Format Tools Window
100 ns
Messages
/bmu_2_radix/enc_in -No Data-
00
11
01
10
/bmu_2_radix/bm00 -No Data-
00
10
01
/bmu_2_radix/bm01 -No Data-
01
01
00
10
/bmu_2_radix/bm10 -No Data-
01
01
10
00
/bmu_2_radix/bm11 -No Data-
10
00
01
/bmu_2_radix/s1 -No Data-
/bmu_2_radix/s2 -No Data-

Waveform for ACS


```
Messages
/radix_4_acs/enc_in     10
/radix_4_acs/clk        1
/radix_4_acs/rst        0
/radix_4_acs/acs_out    0100
/radix_4_acs/a1         0111
/radix_4_acs/a2         0111
/radix_4_acs/a3         0110
/radix_4_acs/a4         0110
/radix_4_acs/mux_i...   0
/radix_4_acs/mux_i...   0
/radix_4_acs/mux_i...   1
/radix_4_acs/mux_...    0110
/radix_4_acs/mux_...    0101
/radix_4_acs/bm00       01
/radix_4_acs/bm01       10
/radix_4_acs/bm10       00
/radix_4_acs/bm11       01
```

Conclusion:

It is noticed that the hard decision technique can detect any number of errors which are less than or equal to the correction capacity of the code.
We have completed the design of Convolutional Encoder and Viterbi Decoder that achieves minimum decoding delay, data rate upto 211 Mbps at the optimum Constraint Length K = 2, with hard decision decoding and reasonable hardware complexity

So far, this work discusses most of the known VLSI implementation techniques for the hard-decision Viterbi algorithm in standard cell CMOS technology and carefully analyzes the tradeoffs and dependencies between different design decisions.
To the best of authors' knowledge, this is the most comprehensive analysis of hard-decision Viterbi algorithm VLSI implementation based on actual designs.

References:

[1] G. Fettweis and H. Meyr, "Cascaded feedforward architectures for parallel Viterbi decoding," in *Proc. IEEE Int. Sym. Circuits Syst.*, May 1990, pp. 978–981.

[2] P. J. Black and T. H. Y. Meng, "A 1-Gb/s, four-state, sliding block Viterbi decoder," *IEEE J. Solid-State Circuits*, vol. 32, no. 6, pp. 797–805, Jun. 1997.

[3] G. Fettweis and H. Meyr, "Parallel Viterbi decoding by breaking the compare-select feedback bottleneck," *IEEE Trans. Commun.*, vol. 37, no. 8, pp. 785–790, Aug. 1989.

[4] C. B. Shung, H.-D. Ling, R. Cypher, P. H. Siegel, and H. K. Thapar, "Area-efficient architectures for the Viterbi algorithm part I: Theory," *IEEE Trans. Commun.*, vol. 41, no. 4, pp. 636–644, Apr. 1993.

[5] C. B. Shung, H.-D. Ling, R. Cypher, P. H. Siegel, and H. K. Thapar, "Area-efficient architectures for the Viterbi algorithm part II: Applications," *IEEE Trans. Commun.*, vol. 41, no. 5, pp. 802–807, May 1993.

[6] T. Jarvinen, P. Salmela, T. Sipila, and J. Takala, "Systematic approach for path metric access in Viterbi decoders," *IEEE Trans. Commun.*, vol. 53, no. 5, pp. 755–759, May 2005.

[7] M. Benaissa and Y. Zhu, "A novel high-speed configurable Viterbi decoder for broadband access," *J. Appl. Signal Process. EURASIP JASP*, no. 13, pp. 1317–1327, 2003.

[8] J. J. Kong and K. K. Parhi, "K-nested layered look-ahead method and architectures for high throughputViterbi decoder," in *Proc. IEEEWorkshop Signal Process. Syst.*, Aug. 2003, pp. 99–104.

[9] S. Hong, J. Yi, and W. E. Stark, "VLSI design and implementation of low-complexity adaptative turbo-code encoder and decoder for wireless mobile communication applications," Proc. IEEE Workshop Signal Pro- cessing Syst., pp. 233–242, 1998.

[10] S. Hong and W. E. Stark, "Power consumption vs. decoding performance relationship of VLSI decoders for low energy wireless communication system design," Proc. IEEE 6th Int.

Authors

Susrutha Babu Sukhavasi was born in India, A.P. He received the **B.Tech** degree from JNTU, A.P, and **M.Tech** degree from SRM University, Chennai, Tamil Nadu, India in 2008 and 2010 respectively. He worked as **Assistant Professor** in Electronics & Communications Engineering in Bapatla Engineering College for academic year 2010-2011 and from 2011 to till date working in **K L University**. He is a member of Indian Society For Technical Education and International Association of Engineers. His research interests

include Mixed and Analog VLSI Design, FPGA Implementation, Low Power Design and wireless Communications, Digital VLSI. He published articles in various international journals and conference.

Chiranjeevi Pilla was born in garividi, vizianagaram(dist), a.p, india. He received the B.Tech. degree in Electronics & Communications Engineering from St. Theressa inistitute of Engineering &Technology, garividi, A.P., Affiliated to the JNTU , Kakinada, A.P., India in 2009 and pursuing M.Tech Degree in VLSI technology in KL University. His research interests include Digital VLSI Design and Fault Diagnosis & testing and Verification.

Suparshya Babu Sukhavasi was born in India, A.P. He received the **B.Tech** degree from JNTU, A.P, and **M.Tech** degree from SRM University, Chennai, Tamil Nadu, and India in 2008 and 2010 respectively. He worked as **Assistant Professor** in Electronics & Communications Engineering in Bapatla Engineering College for academic year 2010-2011 and from 2011 to till date working in **K L University**. He is a member of Indian Society For Technical Education and International Association of Engineers. His research interests include Mixed and Analog VLSI Design, FPGA Implementation, Low Power Design and Wireless communications, VLSI in Robotics. He published articles in various international journals and conference.

Dr.Habibulla khan born in India, 1962. He obtained his B.E. from V R Siddhartha Engineering College, Vijayawada during 1980-84. M.E from C.I.T, Coimbatore during 1985-87 and PhD from Andhra University in the area of antennas in the year 2007.He is having more than 20 years of teaching experience and having more than 20 international, national journals/conference papers in his credit.Prof. Habibulla khan presently working as **Head of the ECE department at K L University**. He is a fellow of I.E.T.E, Member IE and other bodies like ISTE. His research interested areas includes Antenna system designing, microwave engineering, Electro magnetics and RF system designing.