Viterbi Decoding Techniques for the TMS320C54x DSP Generation

Henry Hendrix *Member, Group Technical Staff*

ABSTRACT

In most wireless communications systems, convolutional coding is the preferred method of error-correction coding to overcome transmission distortions. This report outlines the theory of convolutional coding and decoding and explains the programming techniques for Viterbi decoding in the Texas Instruments (TI) TMS320C54x™ generation of digital signal processors (DSPs). The same basic methods decode any convolutional code. This application report examines the problem from a generic viewpoint rather than outlining a solution for a specific standard.

Contents

List of Figures

List of Tables

1 Introduction

Although used to describe the entire error-correction process, Viterbi specifically indicates use of the Viterbi algorithm (VA) for decoding. The encoding method is referred to as convolutional coding or trellis-coded modulation. The outputs are generated by convolving a signal with itself, which adds a level of dependence on past values. A state diagram illustrating the sequence of possible codes creates a constrained structure called a trellis. The coded data is usually modulated; hence, the name trellis-coded modulation.(1)

2 Convolutional Encoding and Viterbi Decoding

2.1 Convolutional Versus Block-Level Coding

Convolutional coding is a bit-level encoding technique rather than block-level techniques such as Reed-Solomon coding. Advantages of convolutional codes over block-level codes for telecom/datacom applications are:(2)

- With soft-decision data, convolutionally encoded system gain degrades gracefully as the error rate increases. Block-level codes correct errors up to a point, after which the gain drops off rapidly.
- Convolutional codes are decoded after an arbitrary length of data, while block-level codes introduce latency by requiring reception of an entire data block before decoding begins.
- Convolutional codes do not require block synchronization.

Although bit-level codes do not allow reconstruction of burst errors like block-level codes, interleaving techniques spread out burst errors to make them correctable.

Convolutional codes are decoded by using the trellis to find the most likely sequence of codes. The VA simplifies the decoding task by limiting the number of sequences examined. The most likely path to each state is retained for each new symbol.

The TMS320C54x incorporates a special hardware unit to accelerate Viterbi metric-update computation. This compare-select-store unit with dual accumulators and a splittable ALU performs a Viterbi butterfly in four cycles.(3)

2.2 Encoding Process

Convolutional encoder error-correction capabilities result from outputs that depend on past data values. Each coded bit is generated by convolving the input bit with previous uncoded bits. An example of this process is shown in Figure 1. The information bits are input to a shift register with taps at various points. The tap values are combined through a Boolean XOR function (the output is high if one and only one input is high) to produce output bits.

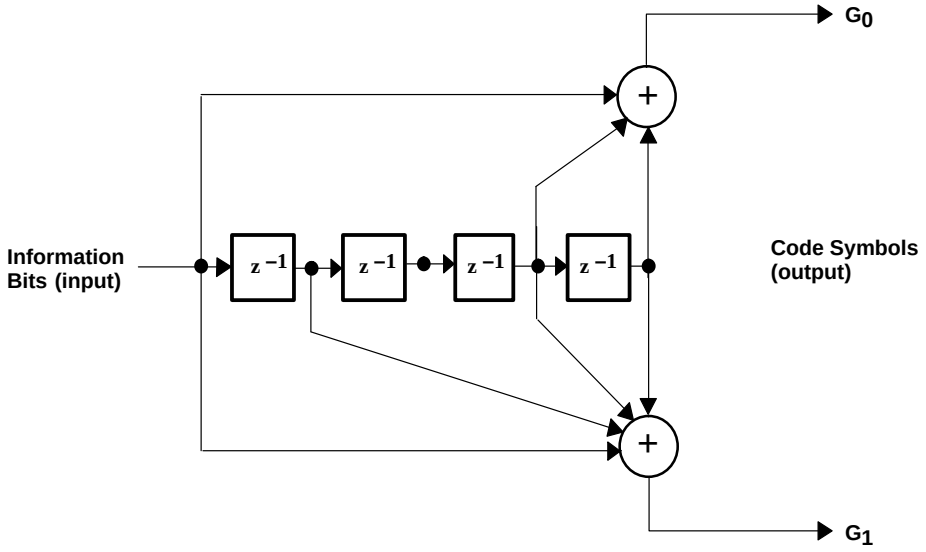


Figure 1. Constraint Length 5, Rate 1/2 Convolutional Encoder

Error correction is dependent on the number of past samples that form the code symbols. The number of input bits used in the encoding process is the constraint length and is calculated as the number of unit delays plus one.

In Figure 1, there are four delays. The constraint length is five. The constraint length represents the total span of values used and is determined regardless of the number of taps used to form the code words. The symbol K represents the constraint length. The constraint length implies many system properties; most importantly, it indicates the number of possible delay states.

2.3 Coding Rate

Another major factor influencing error correction is the coding rate, the ratio of input data bits to bits transmitted. In Figure 1, two bits are transmitted for each input bit for a coding rate of 1/2. For a rate 1/3 system, one more XOR block produces one more output for every input bit. Although any coding rate is possible, rate 1/n systems are most widely used due to the efficiency of the decoding process.

The output-bit combination is described by a polynomial. The system, as shown in Figure 1, uses the polynomials:

$$G0(x) = 1 + x^3 + x^4 \quad (1)$$

$$G1(x) = 1 + x + x^3 + x^4 \quad (2)$$

Polynomial selection is important because each polynomial has different error-correcting properties. Selecting polynomials that provide the highest degree of orthogonality maximizes the probability of finding the correct sequence.[4]

2.4 Decoding Process

Convolutionally encoded data is decoded through knowledge of the possible state transitions, created from the dependence of the current symbol on past data. The allowable state transitions are represented by a trellis diagram.

A trellis diagram for a K = 3, 1/2-rate encoder is shown in Figure 2. The delay states represent the state of the encoder (the actual bits in the encoder shift register), while the path states represent the symbols that are output from the encoder. Each column of delay states indicates one symbol interval.

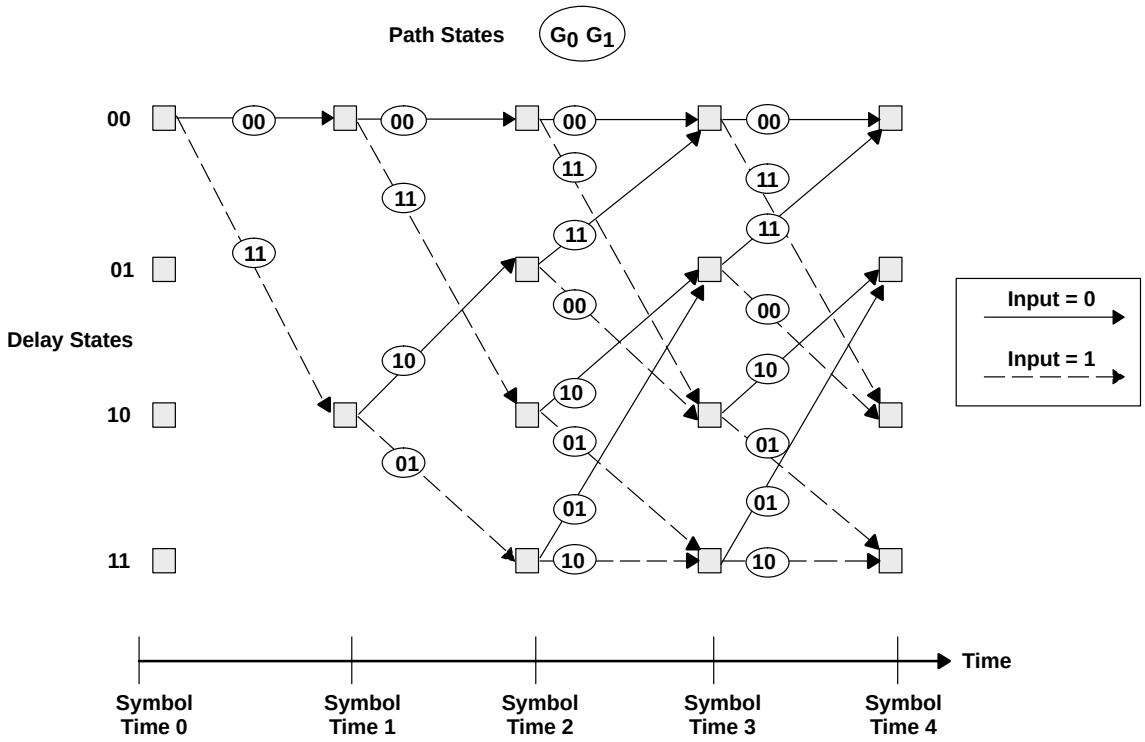


Figure 2. Trellis Diagram for K = 3, Rate 1/2 Convolutional Encoder

The number of delay states is determined by the constraint length. In this example, the constraint length is three and the number of possible states is $2^{K-1} = 2^2 = 4$. Knowledge of the delay states is very useful in data decoding, but the path states are the actual encoded and transmitted values.

The number of bits representing the path states is a function of the coding rate. In this example, two output bits are generated for every input bit, resulting in 2-bit path states. A rate 1/3 (or 2/3) encoder has 3-bit path states, rate 1/4 has 4-bit path states, and so forth. Since path states represent the actual transmitted values, they correspond to constellation points, the specific magnitude and phase values used by the modulator.

The decoding process estimates the delay state sequence, based on received data symbols, to reconstruct a path through the trellis. The delay states directly represent encoded data, since the states correspond to bits in the encoder shift register.

In Figure 2, the most significant bit (MSB) of the delay states corresponds to the most recent input and the least significant bit (LSB) corresponds to the previous input. Each input shifts the state value one bit to the right, with the new bit shifting into the MSB position. For example, if the current state is 00 and a 1 is input, the next state is 10; a 0 input produces a next state of 00.

Systems of all constraint lengths use similar state mapping. The correspondence between data values and states allows easy data reconstruction once the path through the trellis is determined.

2.5 VA and Trellis Paths

The VA provides a method for minimizing the number of data-symbol sequences (trellis paths). As a maximum-likelihood decoder, the VA identifies the code sequence with the highest probability of matching the transmitted sequence based on the received sequence.

The VA is composed of a metric update and a traceback routine. In the metric update, probabilities are accumulated for all states based on the current input symbol. The traceback routine reconstructs the data once a path through the trellis is identified. A brief psuedo-code sequence of the major steps for the VA is shown in Figure 3.

```
for each frame:
   {
   Initialize metrics
   for each symbol:
      {
      Metric Update or Add-Compare-Select (ACS)
      For each delay state:
         {
         Calculate local distance of input to each possible path
         Accumulate total distance for each path
         Select and save minimum distance
         Save indication of path taken
         }
      }
   Traceback
   for each bit in a frame (or for minimum # bits):
      {
      Calculate position in transition data of the current state
      Read selected bit corresponding to state
      Update state value with new bit
      }
   reverse output bit ordering
   }
```

Figure 3. Pseudo Code for the Viterbi Algorithm

2.6 Metric Update

Although one state is entered for each symbol transmitted, the VA must calculate the most likely previous state for all possible states, since the actual encoder state is not known until a number of symbols is received. Each delay state is linked to the previous states by a subset of all possible paths. For rate 1/n encoders, there are only two paths from each delay state. This considerably limits the calculations.

The path state is estimated by combining the current input value and the accumulated metrics of previous states. Since each path has an associated symbol (or constellation point), the local distance to that symbol from the current input is calculated. For a better estimation of data validity, the local distance is added to the accumulated distances of the state to which the path points.

Because each state has two or more possible input paths, the accumulated distance is calculated for each input path. The path with the minimum accumulated distance is selected as the survivor path. This selection of the most probable sequence is key to VA efficiency. By discarding most paths, the number of possible paths is kept to a minimum.

An indication of the path and the previous delay state is stored to enable reconstruction of the state sequence from a later point. The minimum accumulated distance is stored for use in the next symbol period. This is the metric update that is repeated for each state. The metric update also is called the add-compare-select (ACS) operation: accumulation of distance data, comparison of input paths, and selection of the maximum likelihood path.

2.7 Traceback

The actual decoding of symbols into the original data is accomplished by tracing the maximum likelihood path backwards through the trellis. Up to a limit, a longer sequence results in a more accurate reconstruction of the trellis. After a number of symbols equal to about four or five times the constraint length, little accuracy is gained by additional inputs.(5)

The traceback function starts from a final state that is either known or estimated to be correct. After four or five times, the constraint length, the state with the minimum accumulated distance can be used to initiate traceback.† A more exact method is to wait until an entire frame of data is received before beginning traceback. In this case, tail bits are added to force the trellis to the zero state, providing a known point to begin traceback.

In the metric update, data is stored for each symbol interval indicating the path to the previous state. A value of 1 in any bit position indicates that the previous state is the lower path, and a 0 indicates the previous state is the upper path. Each prior state is constructed by shifting the transition value into the LSB of the state. This is repeated for each symbol interval until the entire sequence of states is reconstructed. Since these delay states directly represent the actual outputs, it is a simple matter to reconstruct the original data from the sequence of states. In most cases, the output bits must be reverse ordered, since the traceback works from the end to the beginning.

2.8 Soft Versus Hard Decisions

Local distances are calculated for each possible path state in the metric update, producing a probability measure that the received data was sent as that symbol. The method used to calculate these local distances depends on the representation of the received data. If the data is represented by a single bit, it is referred to as hard-decision data and Hamming distance measures are used. When the data is represented by multiple bits, it is referred to as soft-decision data and Euclidean distance measures are used.

The use of soft-decision inputs can provide up to about 2.2 dB more E_b/N_0 at the same bit-error level (for 4-bit data). This is because the received data contains some information on the reliability of the data. Table 1 lists values and their significance for 3-bit quantized inputs.

† In practice, this state's metric is only slightly lower than the others. This can be explained by the fact that all paths with drastically lower metrics have already been eliminated. Some more advanced forms of the VA look at two or more states with the lowest accumulated distances, and pick the actual path based on other criteria.(6)

Table 1. Soft-Decision Values

Value	Significance
011	Most confident value
010	
001	
000	Least confident positive value
——	Null value
111	Less confident value
110	
101	
100	Most confident negative value

These soft-decision values typically come from a Viterbi equalizer, which reduces intersymbol interference. This produces confidence values based on differences between received and expected data.

2.9 Local-Distance Calculation

With hard-decision inputs, the local distance used is the Hamming distance. This is calculated by summing the individual bit differences between received and expected data. With soft-decision inputs, the Euclidean distance is typically used. This is defined (for rate 1/C) by:

$$local\_distance(j) = \sum_{n=0}^{C-1} [SD_n - G_n(j)]^2 \quad (3)$$

where SD_n are the soft-decision inputs, $G_n(j)$ are the expected inputs for each path state, j is an indicator of the path, and C is the inverse of the coding rate. This distance measure is the (squared) C-dimensional vector length from the received data to the expected data.

Expanding equation 3:

$$local\_distance(j) = \sum_{n=0}^{C-1} [SD_n^2 - 2SD_nG_n(j) + G_n^2(j)] \quad (4)$$

To minimize the accumulated distance, we are concerned with the portions of the equation that are different for each path. The terms $\sum_{n=0}^{C-1} SD_n^2$ and $\sum_{n=0}^{C-1} G_n^2(j)$ are the same for all paths, thus they can be eliminated, reducing the equation to:

$$local\_distance(j) = -2 \sum_{n=0}^{C-1} SD_nG_n(j) \quad (5)$$

Since the local distance is a negative value, its minimum value occurs when the local distance is a maximum. The leading –2 scalar is removed, and the maximums are searched for in the metric update procedure. This equation is a sum of the products of the received and expected values on a bit-by-bit basis. Table 2 expands this equation for several coding rates.

Table 2. Local-Distance Values

Rate	Local_distance(j)
1/2	$SD_0G_0(j) + SD_1G_1(j)$
1/3	$SD_0G_0(j) + SD_1G_1(j) + SD_2G_2(j)$
1/4	$SD_0G_0(j) + SD_1G_1(j) + SD_2G_2(j) + SD_3G_3(j)$

The dependence of G_n on the path is due to the mapping of specific path states to the trellis structure, as determined by the encoder polynomials. Conversely, the SD_n values represent the received data and have no dependence on the current state. The local-distance calculation differs depending on which new state is being evaluated.

The $G_n(j)$'s are coded as signed antipodal values, meaning that 0 corresponds to +1 and 1 corresponds to –1. This representation allows the equations to be even further reduced to simple sums and differences in the received data. For a rate 1/n system, there are only 2^n unique local distances at each symbol interval. Since half of these local distances are simply the inverse of the other half, only 2^{n-1} values must be calculated and/or stored.

2.10 Puncturing

Puncturing is a method to reduce the coding rate by deleting symbols from the encoded data. The decoder detects which symbols were deleted and replaces them, a process called depuncturing. While this has the effect of introducing errors, the magnitude of the errors is reduced by the use of soft-decision data and null symbols, which are halfway between a positive and negative value. These null symbols add very little bias to the accumulated metrics. In some coding schemes, no null value exists, requiring the depuncturing to use alternatively the smallest positive and negative values.(7) Using the coding scheme in Table 1, the punctured symbols are replaced by 000, then 111, etc. As expected, the performance of punctured codes is not equal to that of their nonpunctured counterparts, but the increased coding rate is worth the decreased performance.

For example, consider a 1/2-rate system punctured by deleting every 4th bit, a puncturing rate of 3/4. This means that the coding rate increases to $\frac{1/2}{3/4} = \frac{2}{3}$

The input sequence I(0) I(1) I(2) I(3) . . . is coded as:

$G_0(0)\ G_1(0)$	$G_0(1)\ G_1(1)$	$G_0(2)\ G_1(2)$	$G_0(3)\ G_1(3)$ ...

then is punctured and becomes:

$G_0(0)\ G_1(0)$	$G_0(1)$ X	$G_0(2)\ G_1(2)$	X $G_1(3)$ ...

Usually, the deleted bit represented as X, alternates between G_0 and G_1.

The bits are recombined and transmitted as:

$G_0(0)\ G_1(0)$	$G_0(1)\ G_1(2)$	$G_0(2)\ G_1(3)$ ...

Assuming the receiver is using 3-bit soft-decision inputs as shown in Table 1, the depunctured data appears as:

$G_0(0)$ $G_1(0)$	$G_0(1)$ 000	$G_0(2)$ $G_1(2)$	111 $G_1(3)$

and the normal Viterbi decoding process then is performed.

3 TMS320C54x Code for Viterbi Decoding

The TMS320C54x code for Viterbi decoding can be divided into three parts: initialization, metric update, and traceback. These same code segments, with slight modifications, are used on systems with different constraint lengths, frame sizes, and code rates.

3.1 Initialization

Before Viterbi decoding begins, a number of events must occur:

The processing mode is configured with:

- Sign extension mode on (SXM = 1)
- Dual 16-bit accumulator mode on (C16 = 1), to enable simultaneous metric update of two trellis paths

The required buffers and pointers are set:

- Input buffer
- Output buffer
- Transition table
- Metric storage (circular buffers must be set up and enabled).

Metric values are initialized.

The block-repeat counter is loaded with a number of output bits – 1 (for metric update).

The input-data buffer is a linear buffer of size FS/CR words, where FS is the original frame size in bits, and CR is the overall coding rate including puncturing. This buffer is larger than the frame size because each transmitted bit is received as a multibit word (for soft-decision data). Since these values are typically four bits or less, they can be packed to save space.

The output buffer contains single-bit values for each symbol period. These bits are packed, so that they require a linear buffer of size FS/16 words.

The transition table size in words is determined by the constraint length and the frame size ($2^{K-5} \times$ FS = number of states/16 $\times$ the frame size).

Metric storage requires two buffers, each with a size equal to the number of states (2^{K-1}). To minimize pointer manipulation, these buffers are usually configured as a single circular buffer. This entails setting the block-size register (BK) to twice the number of states (2^{K-4}) and the index register (AR0) to the number of states divided by two plus one ($2^{K-2} + 1$) to enable automatic repositioning of the pointer for the next symbol interval.

All states, except state 0, are set to the same initial metric value. State 0 is the starting state and requires an initial bias. State 0 is usually set to a value of 0, while all other states are set to the minimum possible value (0x8000), providing room for growth as the metrics are updated.

3.2 Metric Update

Most of the calculation time is spent on the metric update, since all of the states must be updated at each symbol interval. Therefore, much effort has gone into minimizing the metric update calculation time. The compare, select, and store unit (CSSU) is incorporated into the C54x™ architecture architecture to simplify this procedure. The calculations involved in the four steps of the metric update for one state follow.

1. Calculate local distance of input to each possible path.

The local distance can be described as a sum of products; for example, $SD_0G_0(j) + SD_1G_1(j)$ for a rate 1/2 system. This is a straightforward add/subtract/accumulate procedure. Only 2^{n-1} local distances must be calculated for a rate 1/n system, since one half are the inverse of the other half. The inverse local distances are accommodated via subtraction in the total distance accumulation. An example of code for rate 1/2 and 1/3 systems is:

```
; Rate 1/2 local distance calculation
;
 LD     *AR1+,16,A          ; A = SD(2*i)
 SUB    *AR1,16,A,B         ; B = SD(2*i) – SD(2*i+1)
 STH    B,*AR2+             ; temp(0) = difference
 ADD    *AR1+,16,A,B        ; B = SD(2*i) + SD(2*i+1)
 STH    B,*AR2              ; temp(1) = sum
; Rate 1/3 local distance calculation
;
 LD     *AR1+,16,A          ; A= SD(2*i)
 ADD    *AR1+,16,B,B        ; B= SD(2*i) + SD(2*i+1)
 ADD    *AR1–,16,B,B        ; B= SD(2*i) + SD(2*i+1) + SD(2*i+2)
 STH    B,*AR2+             ; temp(0) = + + +
;
 SUB    *AR1+,16,A,B        ; B= SD(2*i) – SD(2*i+1)
 ADD    *AR1–,16,B,B        ; B= SD(2*i) – SD(2*i+1) + SD(2*i+2)
 STH    B,*AR2+             ; temp(1) = + – +
;
 SUB    *AR1+,16,A,B        ; B= SD(2*i) – SD(2*i+1)
 SUB    *AR1–,16,B,B        ; B= SD(2*i) – SD(2*i+1) – SD(2*i+2)
 STH    B,*AR2+             ; temp(2) = + – –
;
 ADD    *AR1+,16,A,B        ; B= SD(2*i) + SD(2*i+1)
 SUB    *AR1+,16,B,B        ; B= SD(2*i) + SD(2*i+1) – SD(2*i+2)
 STH    B,*AR2              ; temp(3) = + + –
```

2. Accumulate total distance for each path.

Due to its splittable ALU, dual accumulators, and specialized instructions, the C54x can accumulate metrics for two paths in a single cycle if the local distance is stored in the T register. The dual add/subtract instruction, ADDSUB, adds the T register to a value from memory, stores the total in the lower half of the accumulator, subtracts the T register from the next memory location and stores the result in the upper half of the accumulator. DSADT performs the complementary calculation, storing subtraction results in the lower accumulator, and additional results in the upper accumulator.

C54x is a trademark of Texas Instruments.

3. Select and save minimum distance
4. Save indication of path taken

These previous two steps are accomplished in a single cycle, due to another specialized C54x instruction. The compare-select-store CMPS instruction uses the CSSU to:

- Compare the two 16-bit signed values in the upper and lower halves of the accumulator
- Store the maximum value to memory
- Indicate the maximum value by setting the test control bit (TC) and shifting this value into the transition register (TRN).

This selects the minimum accumulated metric and indicates the path associated with this value. The previous state values are not stored; they are reconstructed in the traceback routine from the transition register.

3.3 Symmetry for Simplification

For rate 1/n systems, some inherent symmetry in the trellis structure is used to simplify these calculations.

The path states associated with the two paths leading to a delay state are complementary. If one path has G_0 G_1 = 00, the other path has G_0 G_1 = 11. This symmetry is a function of the encoder polynomials, so it is true in most systems, but not all.

Two starting and ending states are paired in a butterfly structure including all paths between them. The four-path states in a butterfly also have symmetry as previously described (see Figure 4).

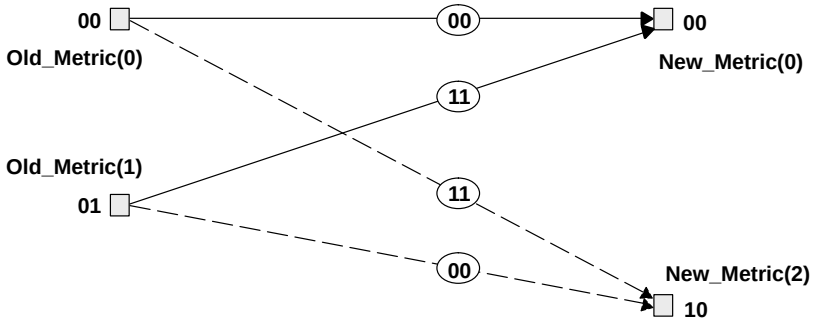


Figure 4. Butterfly Structure for K = 3, Rate 1/2 Convolutional Encoder

These symmetries provide methods to simplify the metric update procedure:

- Only one local-distance measure is needed for each butterfly; it is alternately added and subtracted for each new state.
- The prior accumulated metrics (old metric values) are the same for the updates of both new states, minimizing address manipulations.

For these reasons and to satisfy pipeline latencies, the metric update is usually performed on butterflies.

Since rate 1/n systems have 2^{n-1} absolute local distances for each symbol interval, many butterflies share the same local distances. The local distances are calculated and stored before the rest of the metric update. The following is the C54x code for a single butterfly:

```
LD    *AR2,T    ; T = local distance
DADST  *AR5,A   ; A = Old_Met(2*j) + T // Old_Met(2*j+1) – T
DSADT  *AR5+,B  ; B = Old_Met(2*j) – T // Old_Met(2*j+1) + T
CMPS   A,*AR4+  ; New_Met(j) = (Max (Old_Met(2*j)+T, Old_Met(2*j+1)–T)
                ; TRN = TRN << 1
                ; If (Old_Met(2*j)+T =< Old_Met(2*j+1)–T) Then TRN[0]=1
CMPS   B,*AR3+  ; New_Met(j+2^K–2) = (Max (Old_Met(2*j)–T, Old_Met(2*j+1)+T)
                ; TRN = TRN << 1
                ; If (Old_Met(2*j)–T =< Old_Met(2*j+1)+T) Then TRN[0]=1
```

Five instructions are required to update two states. The states are updated in consecutive order to simplify pointer manipulation. In many systems, the same local distance is used in consecutive butterflies. The T register does not have to be loaded for every butterfly, resulting in a benchmark approaching three cycles per butterfly.

3.4 Use of Buffers

Two buffers are used in the metric update: one for the old accumulated metrics and one for the new metrics. Each array is 2^{K-1} words, equal to the number of delay states. The old metrics are accessed in consecutive order, requiring one pointer. The new metrics are updated in the order 0, 2^{K-2}, 1, 2^{K-2} + 1, 2, 2^{K-2} + 2, etc., and require two pointers for addressing. At the end of the metric update, these buffers are swapped, so that the recently updated metrics become the old metrics for the next symbol interval.

In addition to the metrics buffers, the transition register must be stored. Since only one bit per state is required to indicate the survivor path, one word of memory is required for each of 16 states. Transition register (TRN) storage requires 2^{K-5} words of memory, plus an additional store instruction after every eight butterflies.

3.5 Example Metric Update

Table 3 provides an example of the metric-update procedure for a K = 5, 1/2-rate encoder as used in the Global System for Mobile Communications (GSM) system for speech full-rate traffic (TCH/FS). Two macros, BFLY_DIR and BFLY_REV, automate the butterfly calculations. In Table 3, sum and diff refer to the local distances. New(†) and Old(†) refer to the current and previous metrics for a given state. The TRN data indicates the state associated with each bit or an unknown, *x*.

Table 3. Metric-Update Operations for GSM Viterbi Decoding

Operation	Calculation
Calculate local distances	Temp(0) = SD_0–SD_1 = diff Temp(1) = SD_0+SD_1 = sum
Load T register	T = Temp(1)
BFLY_DIR	New(0) = max[Old(0)+sum, Old(1)–sum] New(8) = max[Old(0)–sum, Old(1)+sum] TRN = xxxx xxxx xxxx xx08
BFLY_REV	New(1) = max[Old(2)–sum, Old(3)+sum] New(9) = max[Old(2)+sum, Old(3)–sum] TRN = xxxx xxxx xxxx 0819
BFLY_DIR	New(2) = max[Old(4)+sum, Old(5)–sum] New(10) = max[Old(4)–sum, Old(5)+sum] TRN = xxxx xxxx xx08 192A
BFLY_REV	New(3) = max[Old(6)–sum, Old(7)+sum] New(11) = max[Old(6)+sum, Old(7)–sum] TRN = xxxx xxxx 0819 2A3B
Load T register	T = Temp(0)
BFLY_DIR	New(4) = max[Old(8)+diff, Old(9)–diff] New(12) = max[Old(8)–diff, Old(9)+diff] TRN = xxxx xx08 192A 3B4C
BFLY_REV	New(5) = max[Old(10)–diff, Old(11)+diff] New(13) = max[Old(10)+diff, Old(11)–diff] TRN = xxxx 0819 2A3B 4C5D
BFLY_DIR	New(6) = max[Old(12)+diff, Old(13)–diff] New(14) = max[Old(12)–diff, Old(13)+diff] TRN = xx08 192A 3B4C 5D6E
BFLY_REV	New(7) = max[Old(14)–diff, Old(15)+diff] New(15) = max[Old(14)+diff, Old(15)–diff] TRN = 0819 2A3B 4C5D 6E7F
Store transition register	Trans(i) = TRN

After the metrics in one symbol interval are updated, the metrics-buffer pointers are updated for the next iteration. Since the metrics buffers are set up as a circular buffer, this is accomplished without overhead using the auxiliary register update construct *AR_n + 0% in the last butterfly. The transition-data-buffer pointer is incremented by one.

3.6 Traceback Function

Traceback requires much less processing than the metric update, since only one bit per symbol interval is output for hard-output Viterbi. The calculations and code follow:

1. Calculate position in transition data of the current state.

The metric update stores one bit per delay state indicating the survivor path. Although each transition decision table entry has information from 2^{K-1} delay states, only one state is used for each iteration. Due to the order in which the states are updated, the stored transition data is scrambled. The main function of the traceback algorithm is to extract the correct bit from the transition data for each symbol interval. If the butterflies are updated in consecutive order, the transition data for one symbol interval is stored as shown in Table 4. The state values are in hexadecimal to make the structure visible.

Table 4. State Ordering in Transition Data for One Symbol Interval NIL

		Bit Number in Transition Word							
		15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
TRN Word#	0	0	2^{K-2}	1	$2^{K-2}+1$	2	$2^{K-2}+2$	3	$2^{K-2}+3$
	1	8	$2^{K-2}+8$	9	$2^{K-2}+9$	A	$2^{K-2}+A$	B	$2^{K-2}+B$
	2	10	$2^{K-2}+10$	11	$2^{K-2}+11$	12	$2^{K-2}+12$	13	$2^{K-2}+13$
	...								
	$2^{K-5}-1$	$2^{K-2}-8$	$2^{K-1}-8$	$2^{K-2}-7$	$2^{K-1}-7$	$2^{K-2}-6$	$2^{K-1}-6$	$2^{K-2}-5$	$2^{K-1}-5$

		Bit Number in Transition Word							
		7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TRN Word#	0	4	$2^{K-2}+4$	5	$2^{K-2}+5$	6	$2^{K-2}+6$	7	$2^{K-2}+7$
	1	C	$2^{K-2}+C$	D	$2^{K-2}+D$	E	$2^{K-2}+E$	F	$2^{K-2}+F$
	2	14	$2^{K-2}+14$	15	$2^{K-2}+15$	16	$2^{K-2}+16$	17	$2^{K-2}+17$
	...								
	$2^{K-5}-1$	$2^{K-2}-4$	$2^{K-1}-4$	$2^{K-2}-3$	$2^{K-1}-3$	$2^{K-2}-2$	$2^{K-1}-2$	$2^{K-2}-1$	$2^{K-1}-1$

A clearer example for a K = 6 system is shown in Table 5. There are 32 states and two transition words.

Table 5. State Ordering in Transition Table for K = 6, Rate 1/2 System

		Bit Number in Transition Word															
		15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Word #	0	0	10	1	11	2	12	3	13	4	14	5	15	6	16	7	17
	1	8	18	9	19	A	1A	B	1B	C	1C	D	1D	E	1E	F	1F

Relatively simple algorithms find the correct transition word number and the correct bit number within that transition word. Table 5 shows that each 16-bit data word contains eight pairs of transition bits that differ only in the MSB. The three LSBs and the MSB determine the bit position in the word, while the remaining bits determine the word number (see Figure 5).

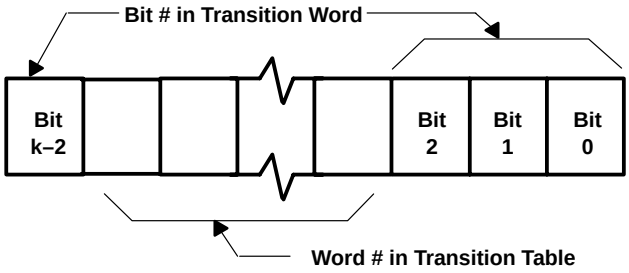


Figure 5. State Variable Representation

The algorithms extract information from a state variable indicating the current delay state. The state is updated in the algorithm reflecting the new value read from the transition table. The correct transition word is determined by masking off the bits between the three LSBs and the MSB, then adding this value to the transition table start address. This can be expressed as:

$$Word\# = (State > 3)\ \&\ MASK,\ where\ MASK = 2^{K-5} - 1. \quad (6)$$

This value is added to the current table address, which is updated each iteration. For systems with $K \leq 5$, this part of the algorithm can be eliminated, since the transition data requires only one word per symbol interval.

Finding the correct bit number within the selected transition word requires consideration of the C54x bit extraction method. The BITT instruction (test bit specified by T register) specifies the bit number in an inverse manner; that is, the bit address corresponds to 15 –TREG. The calculated bit address must correspond to this notation. The bit address is found by shifting State one bit up and appending the MSB as the LSB. This can be expressed as:

$$Bit\# = 2 \times State + [State > (K - 2)]\ \&\ 1 \quad (7)$$

This number is then loaded into the T register for the next step. Although this value can be greater than 16, the BITT instruction uses only the four LSBs of TREG. There is no need to mask off the upper bits.

2. Read selected bit corresponding to state.

The BITT instruction copies the selected bit into the TC bit. Simultaneously, the address is set back to the start of the transition table entry to position it for the next iteration.

3. Update state value with new bit.

The ROLTC instruction shifts the accumulator one bit left and shifts the TC bit into the accumulator as the LSB. This value becomes the new state, used in the next iteration.

The traceback algorithm extracts the output bits in a loop of 16, allowing the single bit from each iteration to be combined into 16-bit words. The algorithm fills the area past the last set of transition decisions with zeros to start on a 16-word boundary. The same number (X) of tail bits that are added at the transmitter must be added before padding, since the output bits represent the actual outputs for X number of prior iterations. When all entries in the transition decision table are processed, the bits in each output word are then reverse ordered to represent the outputs from start to finish.

The code for the traceback routine and bit reordering follows:

```
;
; Traceback routine
;
;  For the following example,
;    A accumulator = State value
;    B accumulator = temp storage
;    K  = constraint length
;    MASK = 2^(K–5) – 1
;    ONE = 1
;
      RSBX  OVM                    ; turn off overflow mode
      STM   #TRANS_END,AR2         ; address of end of transition table
      STM   #NBWORDS–1,AR1         ; number of output words to compute
      MVMM  AR1,AR4                ; Copy AR1 in AR4 (to use in output reversal)
      STM   #OUTPUT+NBWORDS–1,AR3  ; address pointer for output bits (end of buffer)
      LD    #0,A                   ; A contains initial State value (State=0 in this
                                     example)
      STM   #15,BRC                ; Load block repeat counter for inner loop
                                   ; do i=0,NBWORDS–1
BACK  RPTB  TBEND–1                ; do j=0,15
                                   ; Calculate bit position in transition word
      SFTL   A,–(K–2),B            ; B = A>>(K–2)
      AND    ONE,B                 ; B = B&1 = msb of State
      ADD    A,1,B                 ; B = B+A<<1 = 2*State + msb of State
      STLM   B,T                   ; T = B (bit position)
                                   ; Calculate correct transition word
      SFTL   A,–3,B                ; B = A/8 = State/8
      AND    MASK,B                ; B = B&MASK = (K–5)lsb's of State/8
      STLM   AR0                   ; AR0 = index for transition word
      MAR    *+AR2(–2^(K–5))       ; reset pointer to start of table
      MAR    *AR2+0                ; add offset to point to correct transition word
      BITT   *AR2–0                ; Test bit in transition word, reset to table start
      ROLTC  A                     ; Rotate decision in A
                                   ; enddo (j loop)
      TBEND  STL A,*AR3–           ; Store packed output
      BANZD  BACK,*AR1–            ; repeat j loop if frame not finished
      STM    #15,BRC               ; Init block repeat counter for next word
                                   ; enddo (i loop)
; Reverse order of bits within words to output in correct order
;
        MAR       *AR3+            ; Get start of output buffer
        LD        *AR3,A           ; load first word into A
 RVS    SFTA      A,–1,A           ; A>>1, A[0]–> C
        STM       #15,BRC          ;
        RPTB      RVS2–1           ; Do i=0,15
        ROL       B                ; B<<1, B[0] = C
        SFTA      A,–1,A           ; A>>1, A[0]–>C
                                   ; enddo
 RVS2   BANZD     RVS,*AR4–        ; done with all words?
        STL       B,*AR3+          ; save just completed word
        LD        *AR3,A           ; load next word
```

3.7 Depuncturing

The insertion of null (or alternating minimum positive/negative) values into the input data accomplishes depuncturing. If the inputs are properly synchronized, depuncturing can occur while the data is input. Usually, the input buffer is copied to another location where the null values are inserted. An example of code for a rate 1/2 system punctured by 2/3 follows:

```
; Depuncturing code: replaces data in 1 1 x 1 x 1 pattern
;
       STM          #FS/6,BRC       ; number of loops
       STM          #INPUT,AR3      ; address of input buffer
       STM          #DEPUNCT,AR4    ; address of depunctured data
       LD           #111b,16,A      ; ACCH = 111, ACCL = 000
       RPTB         end–1           ; repeat on blocks of 6
start  MVDD         *AR3+,*AR4+     ; copy first word
       MVDD         *AR3+,*AR4+     ; copy second word
       STL          A,*AR4+         ; insert 000 as third word
       MVDD         *AR3+,*AR4+     ; copy fourth word
       STH          A,*AR4+         ; insert 111 as fifth word
       MVDD         *AR3+,*AR4+     ; copy sixth word
end  ...
```

An alternate method, which conserves memory space, inserts the null values during metric update. This affects the local-distance calculation, because each butterfly has a different local distance, depending on whether it contains a punctured data input. This method requires a few more cycles to calculate the additional local distances, but not as many as copying the entire frame of input data. However, it makes the metric update code specific to the puncture pattern, limiting its usefulness for different data streams.

3.8 Benchmarks

Based on the previous code examples, generic benchmarks can be developed for systems of rate 1/n (before puncturing) and any constraint length. The benchmarks use the following symbols:

- R = original coding rate = 1/n = input bits/transmitted bits
- PR = puncturing rate = p/q = bits retained/total bits
- FS = original frame size (# bits) before coding
- FR = number of data frames per second

A method of comparison of the various frame sizes and rates is shown in Figure 6. The benchmark numbers, in cycles per frame, include all processing except minor processor-initialization tasks. The equivalent MIPS are found by multiplying by the frame rate, FR.

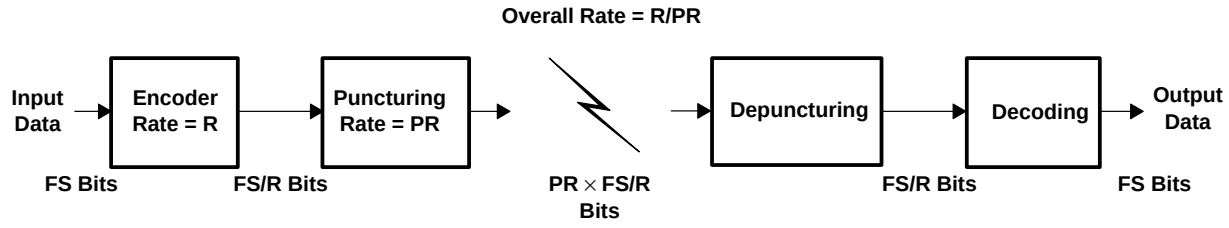


Figure 6. Data Rates for Overall System

$$\text{Metric update: Cycles/frame} = (\#\text{States}/2 \text{ butterflies} \times \text{butterfly calculation} + \text{TRN store} + \text{local dist calculation.}) \times \#\text{ bits} = (2^{K-2} \quad 5 + 2^{K-5} + 1 + n \quad 2^{n-1}) \quad FS \tag{8}$$

$$\text{Traceback: Cycles/frame} = (\text{loop overhead and data storage} + \text{loop} \quad 16) \quad \#\text{ bits}/16 = (9 + 12 \quad 16) \quad FS/16 = 201 \quad FS/16 \tag{9}$$

$$\text{Data reversal: Cycles/frame} = 43 \quad FS/16 \tag{10}$$

$$\begin{aligned} \text{Total MIPS} &= \text{Frame rate} \quad (\text{metric update} + \text{traceback} + \text{data reversal}) \text{ cycles/frame} \\ &= FR \quad [(2^{K-2} \quad 5 + 2^{K-5} + 1 + n \quad 2^{n-1}) \quad FS + (201/16) \quad FS + (43/16) \quad FS] \\ &= FR \quad FS \quad (2^{K-2} \quad 5 + 2^{K-5} + 1 + n \quad 2^{n-1} + (201 + 43)/16) \\ &= FR \quad FS \quad (2^{K-2} \quad 5 + 2^{K-5} + n \quad 2^{n-1} + 16.25) \end{aligned} \tag{11}$$

This total does not include processor setup or depuncturing time. If necessary, depuncturing requires (data copy time × # bits) = (1 cycle/bit × n × FS bits) cycles/frame. With a frame of 200 bits, a rate 1/2 system requires 400 cycles/frame, which is only 0.02 MIPS at a 50-Hz frame rate. The processor setup time for other functions is even smaller, so neither is included in the overall benchmarks. Table 6 summarizes benchmarks for some specific systems.

Table 6. Viterbi Decoding Benchmarks for Various Wireless Standards

Standard	Data Type	Coding Rate (R)	Puncture RatE (PR)	Constraint Length (K)	Frame Size (FS)	Frame Rate (FR)	Benchmark (MIPS)
GSM	Voice	1/2	–	5	189 bits	50 Hz	0.58
	Data – 9.6	1/2	57/61	5	244 bits	50 Hz	0.75
	Data – 4.8	1/3	–	5	152 bits	50 Hz	0.53
IS–136	Voice	1/2	–	6	89 bits	50 Hz	0.46
	FACCH	1/4	–	6	65 bits	50 Hz	0.42
WLL†	Voice	1/2	2/3	7	130 bits	50 Hz	1.20
	FAX	1/2	2/3	6	190 bits	50 Hz	0.97
IS–95	Forward Voice	1/2	–	9	192 bits	50 Hz	6.49
	Reverse Voice	1/3	–	9	192 bits	50 Hz	6.57

† Wireless local loop – proprietary standard

3.9 Variations in Processing

Several factors affect the processing requirements for the systems shown in Table 6. These factors include constraint lengths, coding rates, and convergence time.

As the benchmarks show, the main factor in the processing time is constraint length. Longer constraint lengths require more butterflies, more of the transition register saves, and a more complicated traceback for K > 5. In addition, longer constraint lengths require more data memory for transition register storage, and more program-memory space for metric-update code. In some cases, the butterflies can be looped to minimize program-memory requirements, but usually the local distances are used in an odd order that prevents looping.

Different coding rates mainly affect local-distance calculation rather than the overall processing requirements. A rate other than 1/n, not including puncturing, requires a complex butterfly structure that takes longer than the four-cycle butterfly. The main effect of lower coding rates is increased input/output storage, since more bits are used to represent each symbol.

Processing subframes of data reduces memory requirements. The trellis data converges on an optimal path after approximately five times the constraint length. Since typical constraint lengths are five to nine, traceback can begin after 25 to 45 bits into a frame, less than half of a typical frame size. Performing traceback at this point reduces transition data storage. Extra processing is required to determine the minimum state value at the desired time for traceback to begin.

4 Convolutional Encoding on the TMS320C54x

Convolutionally encoding data is accomplished quite efficiently on the C54x architecture, due to its shifting and dual-word processing capabilities. As previously outlined, each output bit is formed by XORing the current and selected prior input bits.

4.1 General Procedure

The following procedure generates output symbols 16 bits at a time, assuming that the input bits are packed into consecutive 16-bit words with the recent input as the MSB. If the bits do not align on 16-bit boundaries, zeros are inserted in the unused positions.

BIT #	15	14	13	12	11	10	9	8
Data(0)	X(n)	X(n–1)	X(n–2)	X(n–3)	X(n–4)	X(n–5)	X(n–6)	X(n–7)
Data(1)	X(n–16)	X(n–17)	etc.					

BIT #	7	6	5	4	3	2	1	0
Data(0)	X(n–8)	X(n–9)	X(n–10)	X(n–11)	X(n–12)	X(n–13)	X(n–14)	X(n–15)
Data(1)								

1. Load an accumulator with two consecutive 16-bit words, with the most recent bits in the upper accumulator. For example, to encode n = 31 through n = 16:

AG	AHI	ALO
x	X(31) – X(16)	X(15) – X(0)

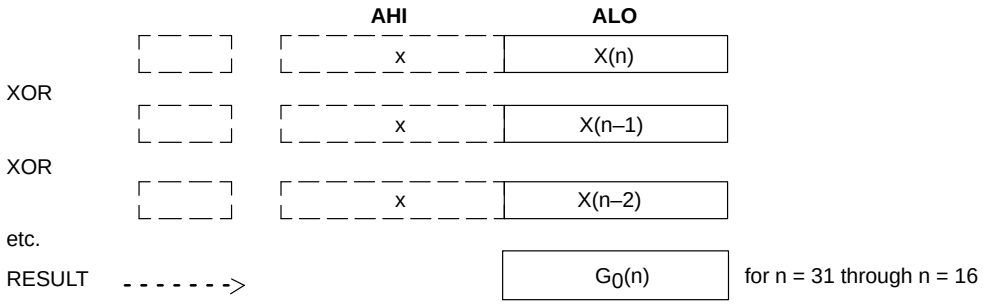
2. Store shifted 16-bit versions of the 32 bits corresponding to each delay specified in the encoder polynomial:

   X(n – 1) = X(30) –> X(15) (delay of 1 for n = 31 through n = 16)

   X(n – 2) = X(29) –> X(14) (delay of 2 for n = 31 through n = 16)

   X(n – 3) = X(28) –> X(13) (delay of 3 for n = 31 through n = 16)

3. XOR the appropriate delayed values with the input data. The input must be reloaded placing it properly in the lower accumulator.



4. Store the 16 encoded bits in the lower half of the accumuator as one set of output bits, G_0
5. Repeat 1–4 for all additional output bits (G_1, G_2, etc.)

The number of iterations of the main loop previously described is determined by the number of bits to be encoded. Some pre- or post-processing of the data may be required, depending on the data format in the input and output buffers.

4.2 Example Code

To illustrate the process, consider the following example with a coding rate of 1/2, a constraint length of 5, and polynomials $G_0 = 33$, $G_1 = 27$. Figure 7 shows the encoder structure.

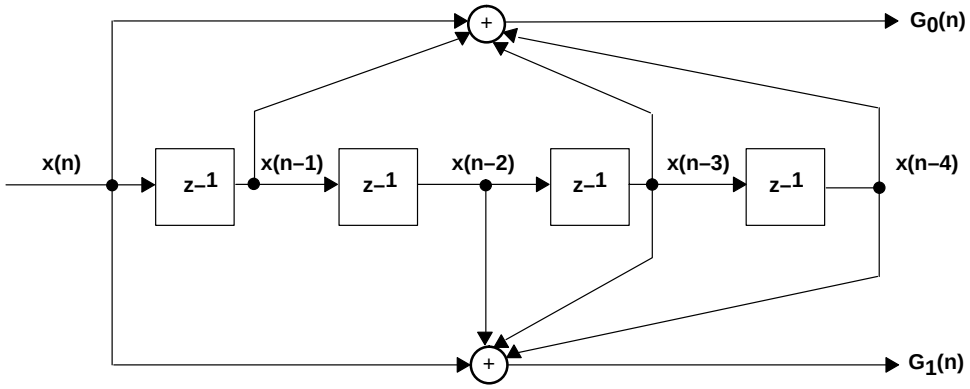


Figure 7. Constraint Length n, Rate 1/2 Convolutional Encoder

The code is:

```
start  stm     #input,AR2      ; set AR2 –> input buffer
       stm     #output,AR3     ; set AR3 –> output buffer
       stm     #nwords–1,BR    ; set loop counter to [(number of bits)/16]–1
       ld      #xn1,dp         ; set data page pointer
       rptb    endloop–1       ; repeat for each 16–bit input
       ld      *ar2+,16,       ; load X(n) thru X(n–15) into A upper accumulator
       or      *ar2–,a         ; load X(n–16) thru X(n–31) into A lower accumulator
       sth     a,1,xn1         ; store delay by 1
       sth     a,2,xn2         ; store delay by 2
       sth     a,3,xn3         ; store delay by 3
       sth     a,4,xn4         ; store delay by 4
       ld      *ar2,a          ; load X(n) into low accumulator
       xor     xn1,a           ; A = X(n) XOR X(n–1)
```

```
        xor     xn3,a           ; A = X(n) XOR X(n–1) XOR X(n–3)
        xor     xn4,a           ; A = X(n) XOR X(n–1) XOR X(n–3) XOR X(n–4)
        stl     a,*ar3+         ; save as G0, point AR3 to next output word
        ld      *ar2+,          ; load X(n) into low accumulator
                ‘               ; point AR2 to next 16 input bits
        xor     xn2,a           ; A = X(n) XOR X(n–2)
        xor     xn3,a           ; A = X(n) XOR X(n–2) XOR X(n–3)
        xor     xn4,a           ; A = X(n) XOR X(n–2) XOR X(n–3) XOR X(n–4)
        stl     a,*ar3+         ; save as G1, point AR3 to next output word
endloop
```

4.3 Improvements to the Code

The preceding encoding loop uses a straightforward technique to show the general method. Some of the XOR and LOAD operations are performed twice, once for each output. By reordering these operations, the duplicate operations are removed, reducing the required cycles by two. The ’C54x second accumulator (B) can be used as the source of delayed values, rather than memory locations, saving an additional three cycles.

```
start   stm   #input,AR2      ; set AR2 –> input buffer
        stm   #output,AR3     ; set AR3 –> output buffer
        stm   #nwords–1,BRC   ; set loop counter to [(number of bits)/16]–1
        rptb  endloop–1       ; repeat for each 16–bit input
        ld    *ar2+,16,a      ; load X(n) thru X(n–15) into A upper accumulator
        or    *ar2–,a         ; load X(n–16) thru X(n–31) into A lower accumulator
        ld    *ar2+,16,b      ; load X(n) thru X(n–15) into B upper accumulator
        or    *ar2,b          ; load X(n–16) thru X(n–31) into B lower accumulator
        xor   b,3,a           ; A = X(n) XOR X(n–3)
        xor   b,4,a           ; A = X(n) XOR X(n–3) XOR X(n–4)
        xor   b,1,a           ; A = X(n) XOR X(n–1) XOR X(n–3) XOR X(n–4)
        sth   a,*ar3+         ; save as G0, point AR3 to next output word
        xor   b,1,a           ; A = X(n) XOR X(n–3) XOR X(n–4)
                              ;   (removes previous X(n–1) term)
        xor   b,2,a           ; A = X(n) XOR X(n–2)
        sth   a,*ar3+         ; save as G1, point AR3 to next output word
endloop
```

An additional two cycles could be saved by using the dual-load instruction, DLD, for the A and B accumulators. The memory-addressing scheme using this instruction requires that the first of the two words be on an even-address boundary (LSB = 0). Since this requirement is not always satisfied in this application, dual loading can only be used every other loop. This improvement could be accommodated by a larger loop that processes the two input words. These changes are typical of improvements that can be made to the ’C54x code, by efficiently using its resources.

4.4 Benchmarks

The example kernel requires 11 x N cycles, where N is the number of bits divided by 16. A general-purpose benchmark is dependent on a number of factors, including the number of terms in the polynomial, the constraint length, and the coding rate. The following is a worst-case estimate:

$$\textit{Main loop cycles}\ (\textit{worst case}) \ = \ 4 + RxK \tag{12}$$

where K is the constraint length (equivalent to the number of XORs + storage for each output bit), and R is the number of output bits.

5 Conclusion

Convolutional coding is used by communication systems to improve performance in the presence of noise. The Viterbi Algorithm and its variations are the preferred methods for extracting data from the encoded streams, since it minimizes the number of operations for each symbol received. The TMS320C54x generation of DSPs allows high Viterbi decoding performance. The core Viterbi butterfly can be calculated at a rate approaching three cycles per butterfly through the use of a splittable ALU, dual accumulators, and a compare-select-store unit.

From the benchmarking equations outlined in this application report, users can quickly determine the processing requirements for any rate 1/n Viterbi decoder.

6 References

1. Ziemer, R.E., and Peterson, R. L,, *Introduction to Digital Communication,* Chapter 6: “Fundamentals of Convolutional Coding,” New York: Macmillan Publishing Company.
2. Edwards, Gwyn, “Forward Error Correction Encoding and Decoding,” Stanford Telecom Application Note 108, 1990.
3. *TMS320C54x User’s Guide* (SPRU131).
4. Clark, G.C. Jr. and Cain, J.B. *Error-Correction Coding for Digital Communications*, New York: Plenum Press.
5. Michelson, A.M., and Levesque, A.H., *Error-Control Techniques for Digital Communications*, John Wiley & Sons, 1985.
6. Chishtie, Mansoor, “A TMS320C53-Based Enhanced Forward Error-Correction Scheme for U.S. Digital Cellular Radio,” *Telecommunications Applications With the TMS320C5x DSPs*, 1994, pp. 103-109.
7. “Using Punctured Code Techniques with the Q1401 Viterbi Decoder,” Qualcomm Application Note AN1401-2a.
8. *Viterbi Decoding Techniques in the TMS320C54x Generation* (SPRA071).

7 Bibliography

Chishtie, Mansoor, “U.S. Digital Cellular Error-Correction Coding Algorithm Implementation on the TMS320C5x,” *Telecommunications Applications With the TMS320C5x DSPs,* 1994, pp. 63-75.

Chishtie, Mansoor, “Viterbi Implementation on the TMS320C5x for V.32 Modems,” *Telecommunications Applications With the TMS320C5x DSPs*, 1994, pp. 77-101.

Appendix A Viterbi API

The following section describes a set of routines which perform convolutional encoding and viterbi decoding. The GSM_enc and GSM_viterbi routines are an implementation of the GSM Half Rate convolutional encoder and Viterbi decoder. The Viterbi_upck routine unpacks the encoded data and transforms it into 3-bit signed, antipodal soft decision values. This routine simulates the transmission of the data through a channel and the soft decisions made at the receiver. We assume that the channel is perfect and we didn't add any noise to the transmitted signal. The user can add noise to the transmitted signal in order to simulate a noisy channel.

The routines can be called in the following order:

```
main()
{

GSM_enc(frame, enc_out, FRAME_WORD_SZ);
viterbi_upck(enc_out, g0g1, FRAME_WORD_SZ);
GSM_viterbi(FRAME_BIT_SZ, METRIC_SZ, metrics, g0g1, trans, dec_out);
}
```

The following section gives the definition of all the parameters.

GSM_enc *Convolutional Encoder*

Syntax

void GSM_enc (int *in, int *out, ushort frame_word_sz)

Arguments

in[frame_word_sz] Pointer to input array of size frame_word_sz.This array contains a 189-bit GSM frame. The last word is padded with zeros.

The bits are arranged in the following order:

bit(15) .... bit(0)
bit(31) ... bit(16)
....
....

out[2 x frame_word_sz] Pointer to output array of size 2 x frame_word_sz. The last two words are padded with zeros.

The bits are arranged in the following order:

G0(15) ... G0(0)
G1(15) ... G1(0)
G0(31) ... G0(16)
G1(31) ... G1(16)
....
....

frame_word_sz Number of 16-bit words required to hold all the bits of a GSM frame.

Description

The function performs half rate convolutional encoding. The generating polynomials are:

g0(D) = 1 + D3 + D4
g1(D) = 1+ D + D3 + D4

At each stage of the algorithm, 2 bits g0(i) and g1(i) are computed from the unit delays and the input bit i.

Benchmarks

Cycles	Core:	frame_word_sz x 9
	Overhead:	32
Code size	27 words	

viterbi_upck *Unpack Routine*

Syntax

void viterbi_upck (int *enc, int *g0g1, ushort frame_word_sz)

Arguments

enc[2 x frame_word_sz] Pointer to encoded input array of size 2x frame_word_sz. The last two words are padded with zeros.

The bits are arranged in the following order:

G0(15) ... G0(0)
G1(15) ... G1(0)
G0(31) ... G0(16)
G1(31) ... G1(16)
....
....

g0g1[2 x frame_bit_sz] Pointer to soft data output array of size 2 x frame_bit_sz, 2x189 words.

The words are arranged in the following order:

G0(0)
G1(0)
G0(1)
G1(1)
...
G0(k)
G1(k)
...

frame_word_sz Number of 16-bit words required to hold all the bits of a GSM frame.

Description

This code separated the packed encoded data into individual G0 and G1 bits to allow simulation of transmission over a channel which induces errors. The received data is an array of G0 and G1 bits represented as a 3-bit signed antipodal values (0≧ 7, 1≧ –7).

Benchmarks

Cycles Core: (frame_word_sz –1) x 242
Overhead: 223

Code size 69 words

GSM_dec *Viterbi Decoder*

Syntax

void GSM_dec (ushort frame_bit_sz, ushort metric_sz, int *m, int *sd, int *trans, int *output)

Arguments

frame_bit_sz	Number of bits in a GSM frame, i.e., 189.
metric_sz	Size of metrics table for the GSM half rate decoder, i.e., 32.
m[metric_sz]	Pointer to metrics table of size metric_sz.
sd[2 x frame_bit_sz]	Pointer to input soft data array of size 2 x frame_bit_sz.
trans[frame_bit_sz]	Pointer to transition table of size frame_bit_sz.
output[frame_word_sz]	Pointer to decoded data output array of size frame_word_sz, i.e., 12.

Description

This function performs Viterbi decoding for data encoded with the GSM Half Rate Convolutional Encoder (R= ½, K= 5, Frame = 189).

Benchmarks

Cycles

Core: Metric Update 44 x frame_bit_sz
Traceback 7 x frame_bit_sz + 8 x frame_word_sz

Overhead: 84

Code size 111 words

Appendix B Glossary

Antipodal value

A diametrically opposite value; for instance 1 for –1, 0 for +1, x(t) and x(t)

Block code

A fixed-length code format that consists of message and parity bits (block = m + p)

Butterfly

The butterfly flowgraph diagram represents the smallest computational unit in a logic calculation showing the encoding or decoding of outputs from given inputs.

Constellation points

Points corresponding to specific magnitude and phase values produced by an encoder. Constellation diagrams show the real and imaginary axis or vector spaces in two dimensions.

Constraint length

The range of blocks over which recurrent or convolutional-coded check bits are operative. The output bits are dependent on the message bits in the previous m–1 bits (where m is the constraint length).

Euclidean distance

Euclidean distance refers to the distance between two M-dimensional vectors, as opposed to the Hamming distance that refers to the number of bits that are different. It is analogous to the analog distance as opposed to the digital distance.

Hamming distance

Named after R. L. Hamming, the Hamming distance of a code is the number of bit positions in which one valid code word differs from another valid code word.

Local distance

The distance in a path state calculated to a symbol or constellation point from the current input

Metric

A function in mathematics relating to the separation of two points relating to the Hamming or Euclidean distance between two code words.

Puncture coding

A technique that selectively and periodically removes bits from an encoder output to effectively raise the data rate. The output from a convolutional encoder passes through a gate circuit controlled by a puncture matrix that defines the characteristics of the error protection for the bits in a block of information.

Survivor path

In a Viterbi decoder, the survivor path is the minimum accumulated distance (shortest time) calculated for each input path.

Trellis

A tree diagram where the branches following certain nodes are identical. These nodes (or states), when merged with similar states, form a graph that does not grow beyond 2^{k-1} where k is the constraint length.

Viterbi decoding

A maximum likelihood decoding algorithm devised by A. J. Viterbi in 1967. The decoder uses a search tree or trellis structure and continually calculates the Hamming (or Euclidean) distance between received and valid code words within the constraint length.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265