Viterbi Implementation on the TMS320C5x for V.32 Modems

Mansoor A. Chishtie
Digital Signal Processing Applications — Semiconductor Group
Texas Instruments Incorporated

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Introduction

Error-control coding plays an increasingly important role in today's communication systems. Described concisely, *error-control coding involves the addition of redundancy to transmitted data so as to provide the means for detecting and correcting errors that inevitably occur in any real communications process* [1].

Such coding techniques are particularly useful for transmission over limited-power channels like general-switched telephone network (GSTN). Adding redundancy to the transmitted data and making use of soft-decision decoding, the bit-error rate can be reduced considerably without increasing transmission power. These coding techniques have proved very useful in the past decade, and many of them have been standardized for modems and other communication devices.

CCITT recommendation V.32 is one such standard that uses trellis-coded modulation and Viterbi decoding to achieve forward error correction at a data transmission rate of 9600 bits per second (bps). This application report deals with the general theory and implementation of the encoding and decoding algorithms required for the V.32 family of modems.

The architecture of the fifth generation of Texas Instruments digital signal processors (DSPs) is especially suited for soft-decision encoding and decoding algorithms. These dynamic programming algorithms often make use of looped code, conditional execution, min-max searches, and pointer-addressing techniques. The enhanced TMS320C5x core CPU allows zero-overhead looping, multiple-condition branches, delayed jumps and calls to minimize execution time, min-max instructions to implement efficient search algorithms, and postmodified indirect addressing (which includes indexed, circular, and bit-reversed addressing modes). These algorithms can be executed very rapidly since almost all 'C5x instructions take only one machine cycle (25 ns) to execute.

Introduction to the V.32 Standard

V.32 modems are designed for use on connections on GSTNs and on point-to-point 2-wire leased telephone-type circuits. The full-duplex mode of operation is supported using echo-cancelation techniques for channel separation. Each channel uses quadrature amplitude modulation (QAM) with a synchronous line-transmission rate of 2400 symbols per second (baud).

QAM is a modulation technique that allows two independent information channels to be modulated into a single carrier signal. These two channels are commonly referred to as *real* and *imaginary* (or *I* and *Q*)[1] components of the signal. A constellation diagram illustrates this concept (see Figure 1). Each point on the constellation has a unique set of real and imaginary components. For a 16-point constellation, four bits are required to uniquely represent each point.

If the input data stream is grouped into quad bits (also called symbols), each quad bit can be mapped to a constellation point, and corresponding I and Q values are modulated into a QAM signal. V.32 modems have a data-transmission rate of either 4800 bps or 9600 bps. At the rate of 9600 bps, either a 16-point or a 32-point constellation can be used (see Figure 1). Obviously, 5-bit-long symbols are required to map each point of a 32-point constellation.

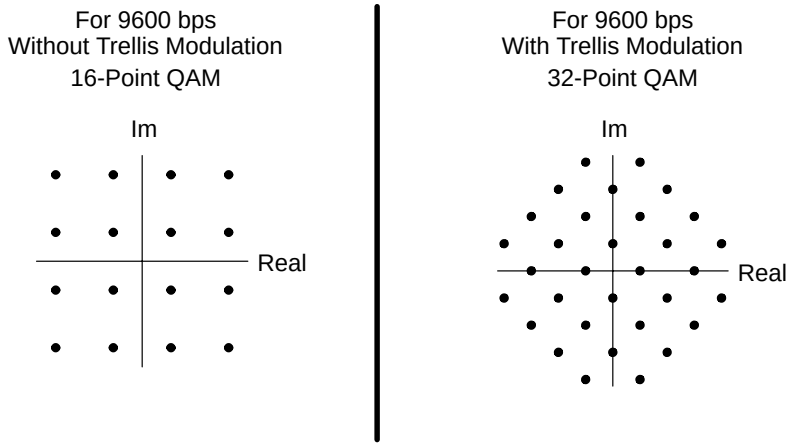
[1] I and Q components are also referred to as X and Y in literature. Both notations are used interchangeably in this paper.

The V.32 standard recommends two alternative modulation schemes at 9600 bps: one using a 16-point constellation, and the other using trellis (convolutional) coding with a 32-point constellation. When using the trellis coding, the input data stream to be transmitted is divided into groups of four consecutive data bits. The first two bits of each group are first differentially encoded and then convolutionally encoded to generate a set of three bits. The other two bits are not encoded but are passed to the output stage. Thus, each output group consists of five bits. These five bits are then mapped into a 32-point (diamond-type) constellation. On the receiver end, a maximum-likelihood decoding algorithm (due to Viterbi) is used to estimate the transmitted data.

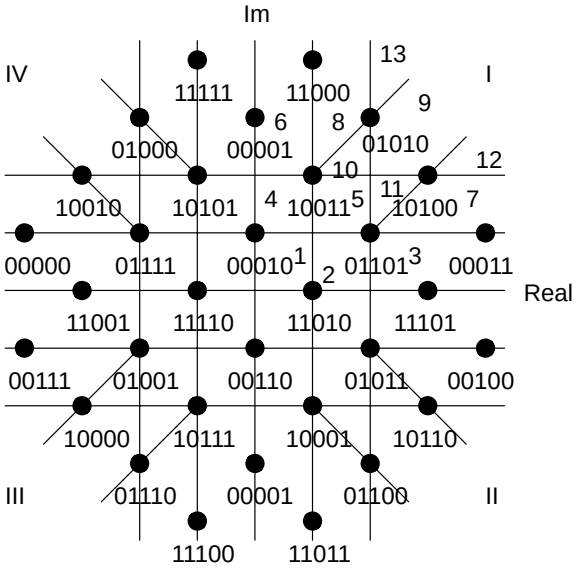
This report deals with the encoding and decoding algorithms as required for the 9600-bps 32-point constellation transmission. The basic encoding algorithm is known as a convolutional encoding scheme, and the decoding algorithm scheme is based on the Viterbi algorithm. Although the 32-point constellation is used extensively to help decode the signals, the actual modulation/demodulation scheme is not implemented in software.

Figure 1. V.32 Modems

(a) V.32 Modems Constellations



(b) V.32 Modems Constellation Regions



Each point corresponds to a 5-bit output symbol.

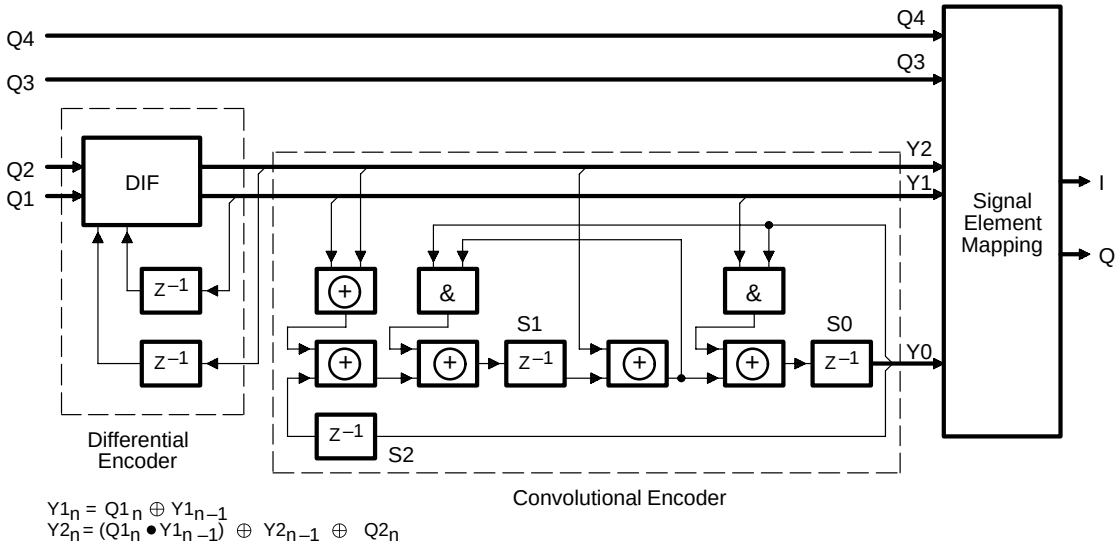
Standard V.32 Encoder

The V.32 encoder (see Figure 2) is divided into two functional blocks:

- Differential encoder
- Convolutional encoder

The input data stream to the encoder is divided into 4-bit long symbols (Q1, Q2, Q3, Q4). Each symbol is processed by the encoder, and the resulting output symbol is 5 bits long (Y0, Y1, Y2, Q3, Q4). The output symbol is larger than the input symbol because it contains error-correction information in addition to the transmit data.

Figure 2. V.32 Encoder



The V.32 standard recommends the QAM technique to transmit data over the channel. Without any error correction information, each symbol has four bits, requiring a 16-point constellation as shown in Figure 2. If a convolutional encoding scheme is employed, each symbol has five bits, and a 32-point constellation is required.

In general, for the same average power, a modulation scheme using a 32-point constellation has higher bit-error rate (BER) when compared with a 16-point constellation scheme. This is because the minimum Euclidean distance between any two points on a 32-point constellation is relatively small, which decreases the noise margin. However, convolutional encoding introduces constraints in transforming an input symbol to a 5-bit output symbol. Specifically, it does not allow two consecutive output symbols to be in the eight neighborhood positions of each other, as seen on the constellation diagram. The minimum distance between two consecutive output symbols is thereby increased, thus providing an overall performance gain of 3 dB.

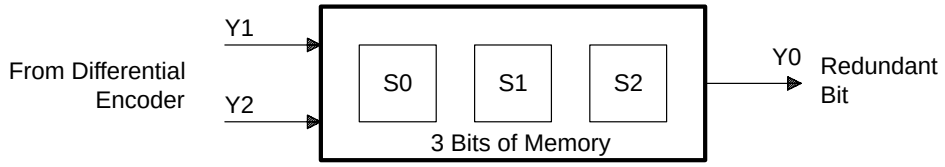
The differential encoder provides protection against 180° phase ambiguity in the channel. The following two equations describe the differential encoding algorithm:

$$Y1_n = Q1_n \oplus Y1_{n-1} \quad (1)$$

$$Y2_n = (Q1_n \bullet Y1_{n-1}) \oplus Y2_{n-1} \oplus Q2_n \quad (2)$$

Notice in Figure 3 that only two input bits are differentially encoded. Because of differential encoding, errors caused by phase reversal in the channel are not allowed to propagate, and the information sequence is reconstructed by the receiver except for the errors at points where phase reversal has occurred [1].

Figure 3. Viterbi Encoder — Convolutional Encoding Scheme



Convolutional Encoder

Definitions:

– S0, S1, and S2 are called delay states
– Y0, Y1, and Y2 are called path states

Constraint condition:

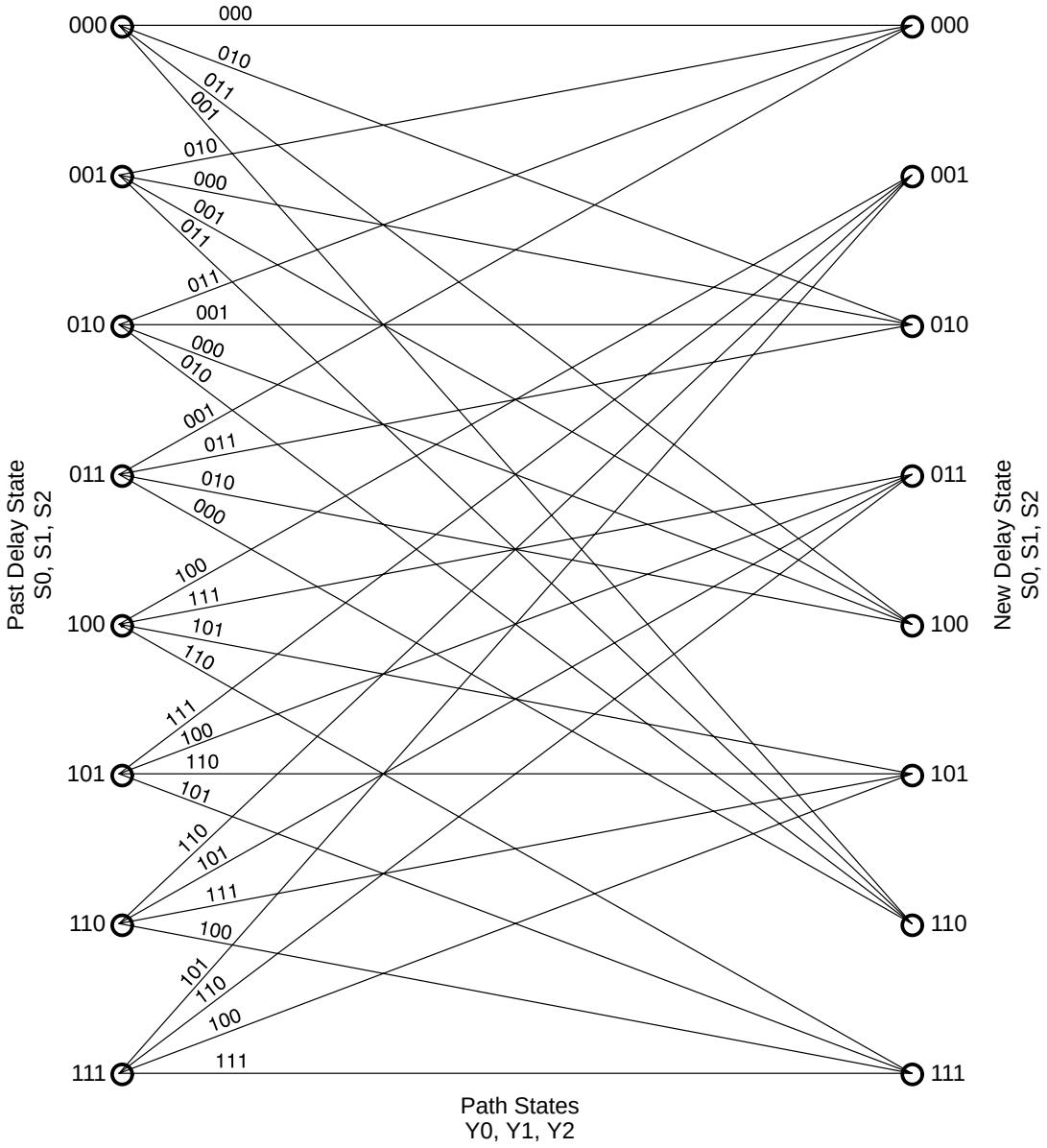
– Given a particular set of delay states (S0, S1, S2), not all path states (Y0, Y1, Y2) are possible.

The convolutional encoder takes the two differentially encoded bits (Y1, Y2) and generates an output bit Y0. Y0 is often called the redundant bit because it carries only the forward error-correction information. Functionally, the convolutional encoder is a 3-bit shift register interconnected by AND and XOR logic. A simplified diagram of a convolutional encoder is shown in Figure 3. By convention, the three bits of encoder memory (S0, S1, and S2) are called delay states, and the set of output bits (Y0, Y1, and Y2) are known as path states. The idea behind this terminology will become obvious later when the trellis structure is considered. The size of encoder memory is sometimes referred to as its constraint length.

One important constraint is imposed by the encoder. Given a particular set of delay states (S0, S1, and S2), not all path states are possible in that time interval. For instance, given a delay state (0, 0, 1) for the encoder, only four path states (0, 0, 0), (0, 1, 0), (1, 0, 0), and (1, 1, 0) are allowed in next time interval.

This leads to the concept of trellis structure. Since the encoder is essentially a finite-state machine, a finite-state diagram may be used to represent it. There are eight possible delay states of the encoder. At any given time, only one delay state (S0, S1, or S2) represents the encoder. In the next instant, only four delay states are possible instead of eight. The particular path chosen at that time depends on the current path state of the encoder (hence, the name path state). The trellis diagram (Figure 4) concisely illustrates all possible transformations from one delay state to another, along with their corresponding path states.

Figure 4. V.32 Modem Trellis Diagram



NOTE: Finite-state diagram for the convolutional encoder showing the relationship between delay and path states. Not all delay states can be reached from a previous delay state.

Viterbi Decoder

The Viterbi algorithm is based on a soft-decision maximum-likelihood decoding technique. The main function of any decoder is to select the most likely output. A simple hard-decision decoder selects a code word that differs from the received sequence in the smallest number of positions. In other words, the code word is chosen that minimizes distance between the received signal and the code word. A soft-decision decoding scheme makes use of past history and reliability information to decode incoming data. A necessary ingredient of any soft-decision decoder is a suitable distance (or cost) function.

A cost function may be unique to each modulation technique. Two widely used cost functions are the Hamming distance and the Euclidean distance functions [2]. The standard Viterbi algorithm does not specify any particular cost function. The Hamming distance function is suitable for binary signals. For PSK and QAM signals, the Euclidean distance function on their respective constellations is appropriate. For an added white gaussian noise (AWGN) channel, the farther the received signal from a point on the constellation, the less likely that it corresponds to that point. Therefore, the distance between the received signal (as it is mapped on the constellation) and a hypothesized output point on the constellation makes a good cost function for any QAM signal. Since V.32 uses QAM modulation, the distance estimate on its constellation is used as the cost function.

Figure 5. Viterbi Decoding — Output Tracking and Cost Function

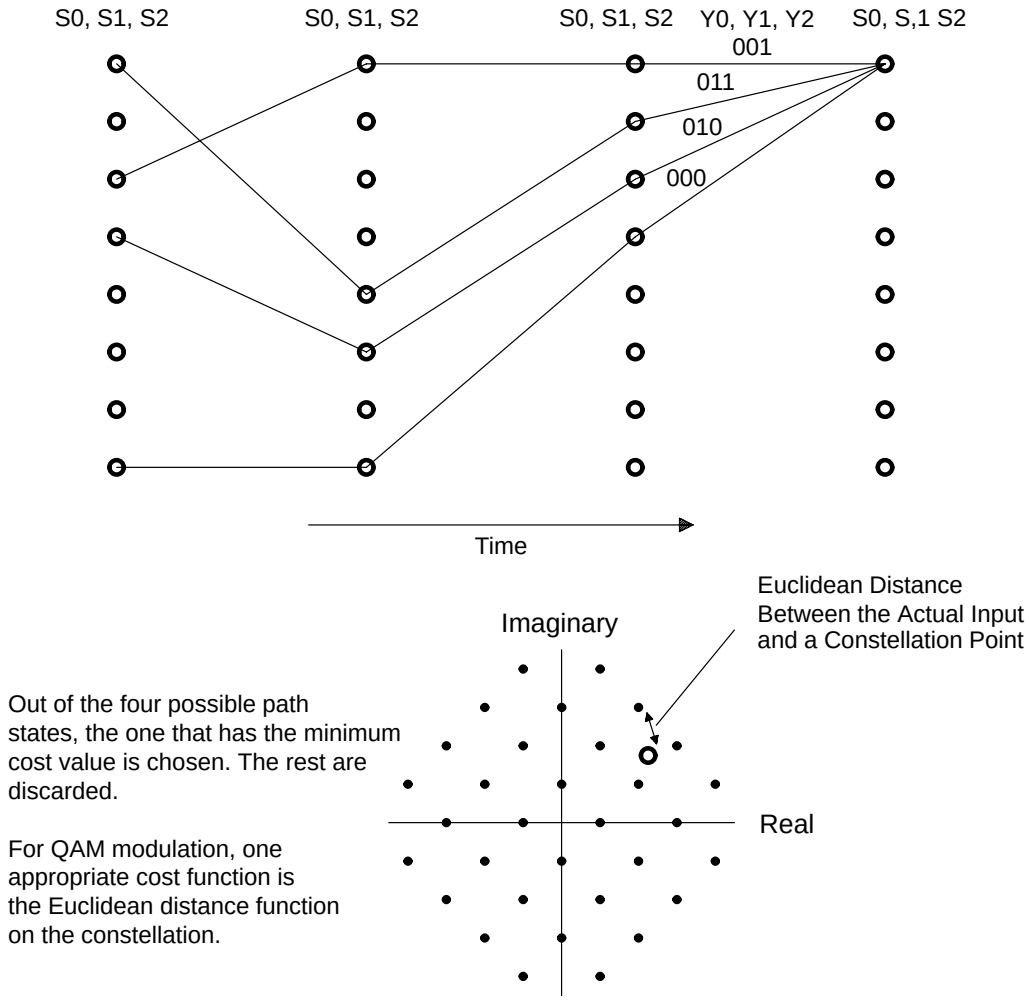


Figure 7 shows an expanded trellis diagram over several symbol time intervals with the x axis representing time and the y axis representing the eight possible delay states of the encoder. The encoder may attain only *one* delay state at any given time, but the decoder keeps track of all the possible states until it decides which one to select. This is the essence of soft-decision algorithms in which the actual decision is delayed until more information is available. Ideally, the maximum-likelihood method looks at the entire stream of input before making any decision about the output. Clearly, this approach is not feasible for real-time applications due to two factors:

- Prohibitive memory requirements, even for relatively small blocks of data
- Inherent time delay before the decoder selects an output

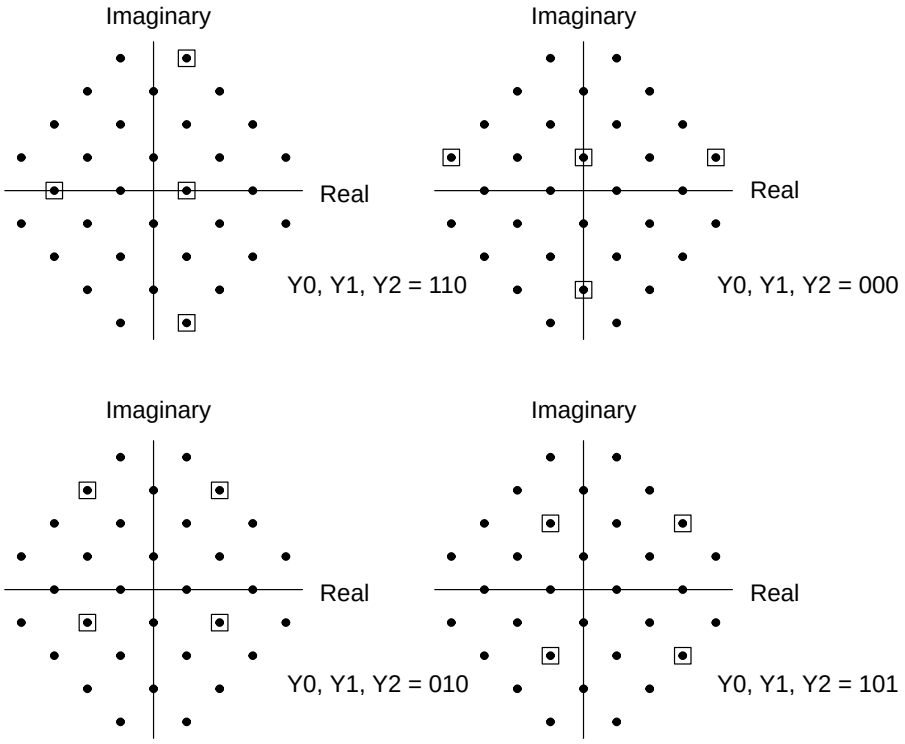
The more practical approach taken by Viterbi is to consider only a finite length of input data before making a decision about the output. The decision-making process relies heavily on the cost function.

To understand this algorithm, consider the expanded trellis diagram as shown in Figure 7. At each time interval, there are eight possible delay states. Since the decoder must keep an “open mind” until it is time

to select the most likely output, all eight states are considered as possible representations of the encoder in that time interval. A particular delay state can be approached only by four states from the previous time interval (see Figure 5). The decoder selects only one of these four states so as to establish a link between the previous time interval and the current one. Note that each link is identified by the path state it represents.

Each path state consists of three bits of a 5-bit symbol. Therefore, one path state uniquely identifies a set of four constellation points. The V.32 signal space mapping is defined in such a way that each set of four points is symmetrically arranged and equally spaced on the constellation, as shown in Figure 6. Furthermore, each set of points is spaced as far apart as possible on the constellation. At the beginning of each sample interval, the decoder compares the received signal with each set and selects the point from each set that is closest to the signal. Essentially, this is a form of hard decoding, but its effect on the quality of the decoder performance is not significant. This is because each set of four points is widely spaced on the constellation so that any noise perturbation is less likely to affect these estimates.

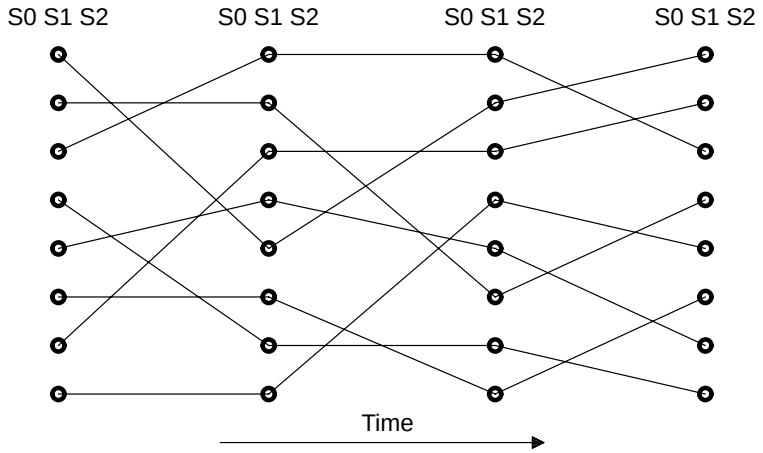
Figure 6. V.32 Modem — Signal Element Mapping



Eight constellation points are selected, and their respective distances from the received signal are computed. Each point corresponds to a different path state. Since each link in Figure 5 is identified by a path state, these computed distance values are associated with each link.

By selecting all eight links, connections are established between the delay states at the current time and the previous time (see Figure 7). In this way, eight independent path traces are stored in memory. The cost function is now updated for each of these path traces. The cost function is the sum of distances associated with each link of a path trace.

Figure 7. Viterbi Decoding — Dynamic Programming



- For every time increment, the minimum cost line is chosen for each of the eight delay states.
- Eight independent path traces are stored in memory.
- For each track, current cost is accumulated as it hops over the delay states.
- The state with the minimum accumulated distance is selected to receive output.

Of the eight path traces, the one that has minimum cost (or accumulated distance) is selected as the most likely path to receive the output. The selected path is traced back, and the 3-bit path state value (Y0, Y1, Y2) that is associated with the last link stored in memory is the result of the Viterbi algorithm. Note that this 3-bit result does not uniquely identify a 5-bit output symbol. The four constellation points that correspond to the 3-bit result are compared with the input corresponding to that time interval, and the 5-bit value associated with the point that is closest to the input is the output of the decoder. Since the output of the decoder corresponds to the time period of the last link, it lags the input of the decoder by the length of the path history maintained by the decoder. It is experimentally determined that the optimal length of a Viterbi decoder is four or five times the constraint length of the convolutional encoder [1]. The V.32 encoder has a constraint length of 3, and the decoder keeps a path history of the past 16 time intervals.

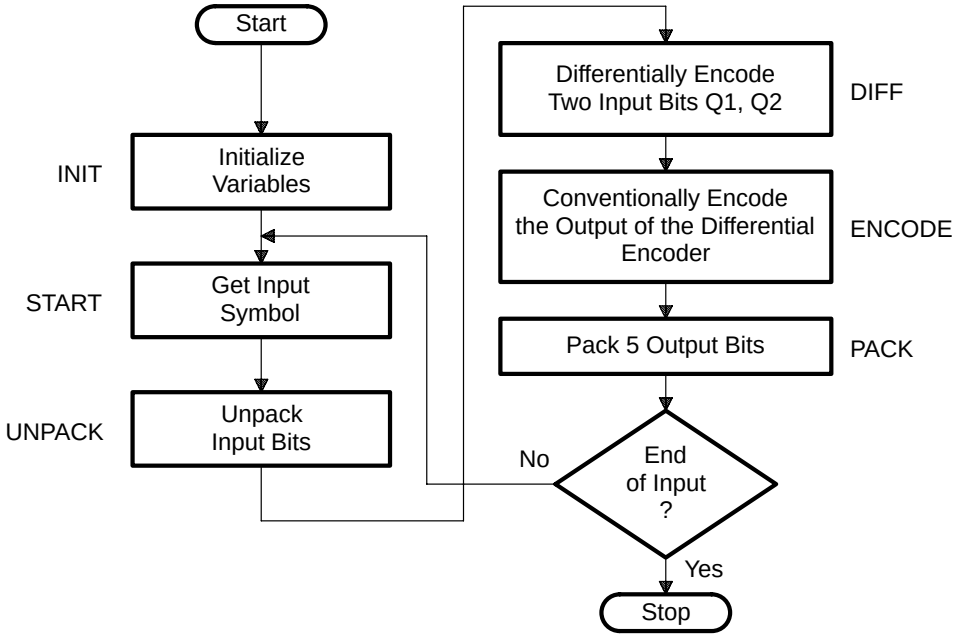
Algorithm Implementation on the TMS320C5x

The three most useful features of TMS320C5x for the Viterbi algorithm are circular buffers, minimum-maximum instructions, and zero-overhead loops. Circular addressing is used extensively throughout the decoder algorithm to access the distance tables, stepping through the path and delay states, and tracing back the past path states to get output. Minimum-maximum value instructions are used in search algorithms to compute minimum Euclidean distance for each state and to find minimum accumulated distance at each time interval. Since the algorithm is based on a dynamic programming technique, it tends to have a multiple looped structure. The zero-overhead loops of TMS320C5x are frequently used by the decoder program.

Encoder Implementation

The V.32 encoder block diagram is shown in Figure 2. As previously explained, it has two functional blocks: the differential encoder and the convolutional encoder. The encoder program flow is shown in Figure 8.

Figure 8. V.32 Encoder Program Flow



The initialization routine INIT sets up auxiliary registers to point to input and output tables and resets the delay states (S0, S1, S2) to 0. This ensures that the initial state of the encoder is known beforehand. It is useful from the decoder point of view because the decoder initializes the cost of delay state 0 to 0 so that this state is always selected in the very first time interval.

The encoder expects the input symbols to be stored in the table PCKD_IP with each element of the table containing a right-justified 4-bit symbol. The table input method is employed because of its simplicity. For real-time applications, other techniques can easily replace the default method. If the input data is coming from an ADC, a simple approach is to create two buffers. One is read by the encoding algorithm, while the other is filled with incoming data by an interrupt service routine. In case the encoding process is required to be synchronous with incoming data, no data buffer is needed. At every symbol time, the input symbol is read from a peripheral device, and the resulting 5-bit output symbol is sent to another external device.

The encoding algorithm operates on binary inputs. Therefore, each input symbol is unpacked into four words (which correspond to each bit) before any processing is done. The UNPACK section uses a zero-overhead block-repeat loop and PLU instructions to perform the unpacking operation.

```
UNPACK:     LACC   LOCATE          ;Get packed input bits
            RPTB   LOOP1           ;For i=0;i<=3;++i
            SACL   *               ;Save the word
            APL    * –             ;Keep LSB only
LOOP1:      SFR                    ;Shift right to get next bit
```

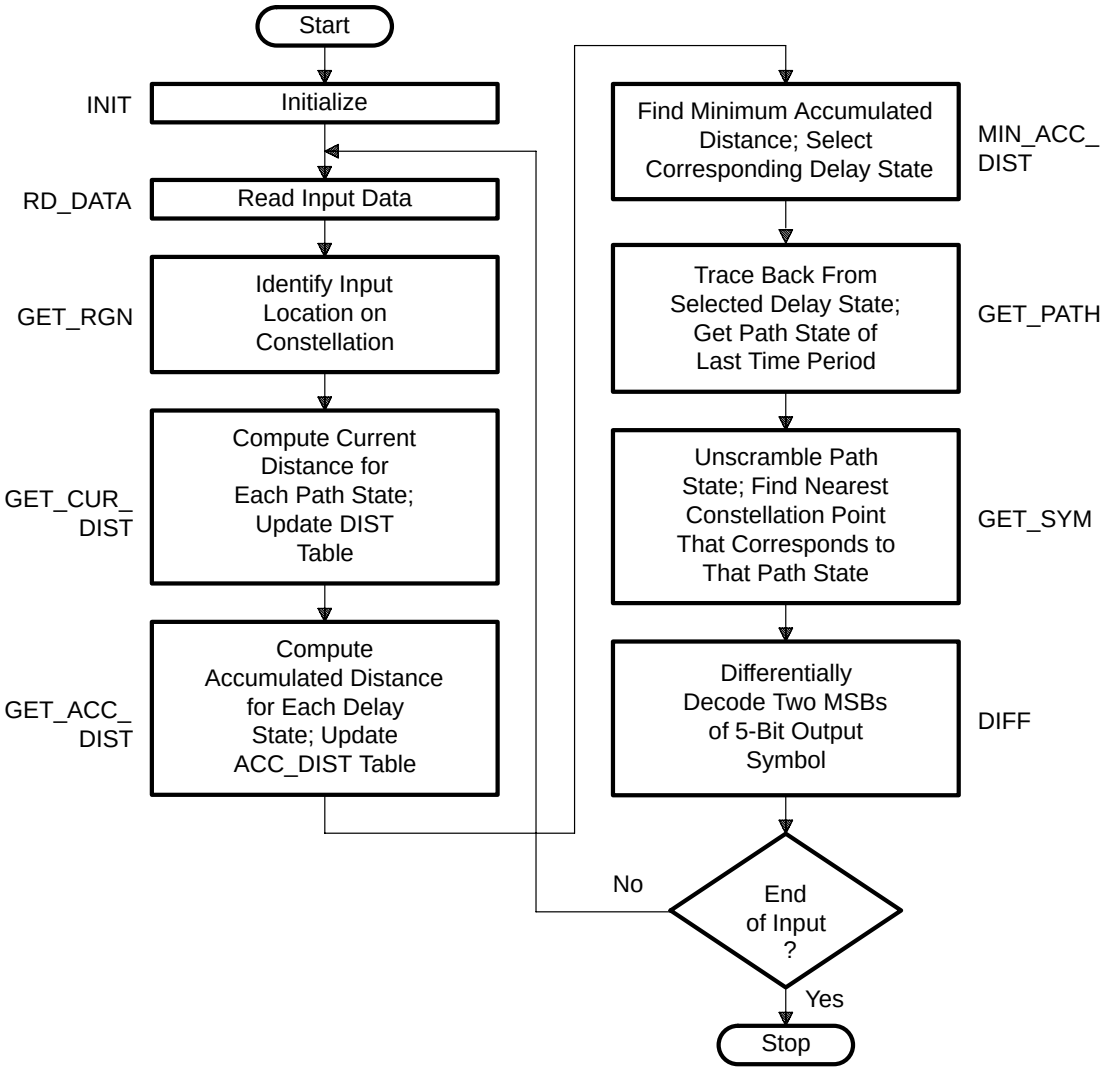
The DIFF function differentially encodes two input bits according to Equation (1) and Equation (2) on page 83. Its output overwrites the original two input bits located in INPUT table. Next, the convolutional encoder processes these two bits and generates a redundant bit Y0. The encoder state (S0, S1, S2) is stored in the STATMEM table, and it is updated each time a new redundant bit is generated.

Finally, the resulting five output bits (OUTPUT + INPUT) are packed into a single word by the PACK function. This output word contains five right-justified bits (Y0, Y1, Y2, Q3, Q4), and it is stored in the output buffer PCKD_OP in sequential order. Note that these five output bits could be sent to a DAC or a front-end modulator instead.

Viterbi Decoder Implementation

In contrast with the convolutional encoding algorithm, the Viterbi decoding algorithm is computationally more complex and numerically more intensive. In general, the execution time of the decoding algorithm is significantly greater than the execution time of the encoder algorithm. This section describes the algorithm in detail as it is implemented on the TMS320C5x. Although the code presented here is designed for the V.32 modem standard, it could easily be transformed for any other application of the Viterbi algorithm.

Figure 9. Decoder Flowchart



The decoder program flowchart is shown in Figure 9. Each process block in the flowchart corresponds to an independent function. The modularity of each block is sacrificed somewhat to gain execution efficiency. In other words, each block is integrated, to a certain extent, with the block that precedes it. The results of a block are frequently passed in internal registers to the next block. However, all system variables are defined explicitly in the beginning, and the line-by-line comments in the source code help identify where the results are being stored.

The initialization routine INIT is called to set up tables and variables. The ACCDIST table, which holds eight accumulated distance values for each delay state, is initialized by this function. As discussed in the *Standard V.32 Encoder* section on page 82, the first state of the encoder is always (0,0,0) (that is, state 0). To ensure that the decoder always chooses state 0 in the first time interval, the initial accumulated cost of state 0 is set to 0 while the rest of the states are set to a cost of 0.5.

The routine RD_DATA is called once every symbol interval to read new data. This is the only routine that needs to be rewritten to suit each application. The code presented here is not designed for any specific hardware. It assumes that some test data has already been stored in the TST_INP table before the decoder is invoked. The input is in the form of 5-bit symbols output by the encoder. Two look-up tables, XLOC and YLOC, convert each symbol to its equivalent real and imaginary axis values (also called XY or IQ values). The channel noise and distortion effects may be added to the I and Q channels independently. The resulting values are saved in variables CURR_X and CURR_Y for later use. This approach is taken so that test data and channel noise data may be computed independently of each other and stored in respective tables before the decoder is invoked. Obviously, this is not a real-time approach. The front-end demodulator can provide I and Q values directly to the device. In that case, RD_DATA is required to save only those values in CURR_X and CURR_Y locations. Each I and Q (or X and Y) input can have a maximum resolution of 16-bits.

Once the current input is located on the constellation by X and Y values, eight constellation points corresponding to the eight path states that are closest to this input point must be identified. Note that each path state corresponds to four unique constellation points (see Figure 6). The brute force method of determining these constellation points is to consider each group of four points individually, compute the distance from each point to the input, and select the closest one. This requires all 32 points that compose the V.32 constellation to be considered for each input symbol. Another way to make the selection is to use a look-up table. Since the locations of the constellation points are known beforehand, it is simpler to identify the region where the input lies and use a table to determine the eight points that are closest to that region. As shown in quadrant I in Figure 1(b), there are 13 distinct regions in each quadrant of the constellation. Each region has a unique set of eight constellation points (corresponding to eight path states). A table called REGION is set up in data memory that contains 13 macro elements, each element having four subelements corresponding to four quadrants of the constellation. Each subelement is a set of eight pointers to the closest constellation points.

To identify the region where the current input lies, the following decision algorithm is used, where X,Y is the location of the current input on the constellation shown in Figure 1(b).

```
If |X| <= 1 Then
   If |Y| <= 1 Then
      Region#1
   Else
      If |Y|<= 2 Then
         Region#4
      Else
         Region#6
Else
   If |X| <= 2 Then
      If |Y| <=1 Then
         Region#2
      Else
         If |Y| <=2 Then
            Region#5
         Else
            If |Y| <= |X|+1 Then
               Region#10
            Else
               Region#8
   Else
      If |Y| <= 1 Then
         Region#3
      Else
         If |Y| > |X|+1 Then
            Region#13
         Else
            If |Y| <= |X|-1 Then
               If |Y| <= 2 Then
                  Region#7
               Else
                  Region#12
            Else
               If |Y| <= 2 Then
                  Region#11
               Else
                  Region#9
```

After identifying a region, a quadrant is selected according to the polarities of X and Y.

Refer to the GET_RGN function of the decoder source code for implementation details. Note the use of delayed conditional branches and the XC instruction to avoid flushing the pipeline. The result of the GET_RGN function is a pointer to the REGION table.

The current cost of each path state is defined as the distance from the current input to the respective constellation point. The result of the GET_RGN function points to a set of eight constellation points. If (X,Y) is the input for a given time interval, and (Xk,Yk) are eight constellation points that correspond to state k (where k = 0...7), then the current distance table is defined as:

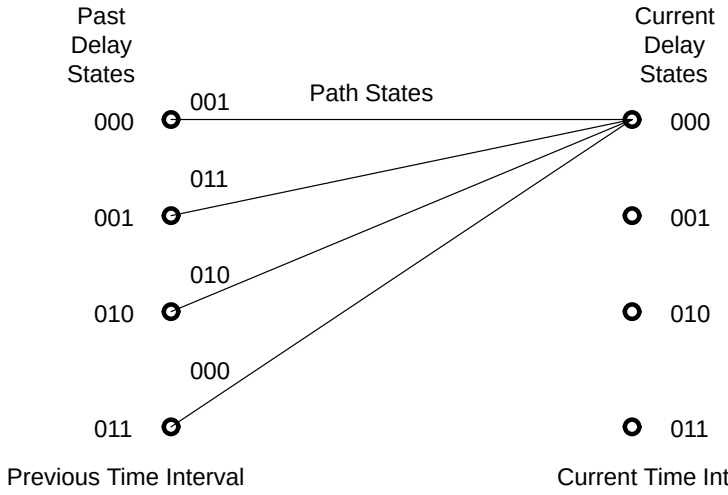
$$\text{DIST}\,[k] = (Xk - X)^2 + (Yk - Y)^2 \;; \quad k = 0 \ldots 7 \tag{3}$$

The square root operation is not performed because it is time-consuming. Although the square root function is not linear, distance values without the square root operation work well because the relationship between x and sqrt(x) is one-to-one and monotonic. The GET_CUR_DIST routine performs this computation for each path state.

```
STATE0:
       LAR    AR2,*+,AR2          ;Get address of 1st point out of 8
       MAR    *0+                 ;Add XLOC, AR2 points inside XLOC
       LACC   *                   ;Get x value of 1st point
       SUB    CURR_X              ;Subtract current x value
       SACL   DIFF_X              ;Save (Xc–Xi)
       SQRA   DIFF_X              ;P=(Xc–Xi) ^2
       ADRK   #32                 ;Now AR2 points inside YLOC
       LACC   *,0,AR0             ;Get Y value of 1st point
       SUB    CURR_Y              ;Subtract current y value
       SACL   DIFF_Y              ;Save (Yc–Yi)
       LACL   #0
       SQRA   DIFF_Y              ;P=(Yc–Yi) ^2, ACC=(Xc–Xi) ^2
       LTA    SMALL               ;ACC=(Xc–Xi) ^2+(Yc–Yi) ^2
       SACH   DIST,4              ;Save acc. distance*2^4
       MPY    DIST
       SPH    DIST                ;Save distance*0.1 in 1st location
```

The distance or cost values are stored in an 8-word DIST table. Each element of the DIST table corresponds to a path state. The order of storage in the table shown in Figure 12 is not a simple ascending or descending form. The reason for this scrambled order is explained later.

Figure 10. Delay State Linking



The next step is to accumulate the cost (or distance) for each delay state at the current time. As previously explained, at every time interval there are eight delay states (S0, S1, S2). Each delay state at the current time interval is linked to four delay states from the previous time interval, as shown in Figure 10. The minimum cost link is identified, and the distance value of the selected link is added to the accumulated cost of the delay state from which it originates. This gives the accumulated cost of the current delay state.

In addition to the accumulated cost, the following information needs to be stored for each delay state:

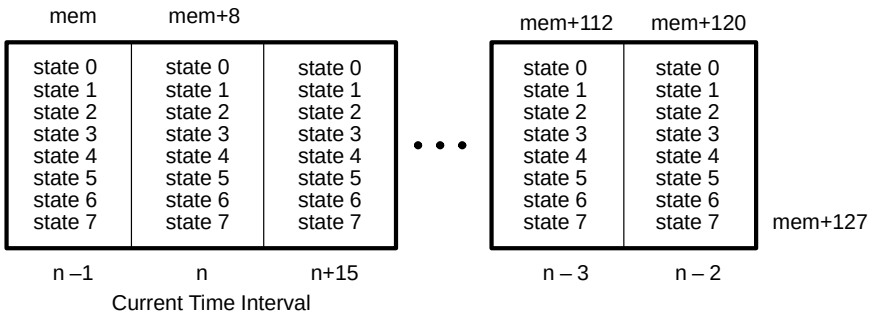
- The path state that identifies the link selected
- The delay state of the previous time interval that is linked to the current delay state

The code to perform these functions is:

```
STATE0:
       RPTB   ENDB0-1             ;For (i=0;i<=3;++i)
       LACC   *,0,AR2             ;Get prev. accumulated distance
       ADD    *,AR5               ;Add current distance
       CRLT                       ;If acc < prev. largest
       NOP                        ;Then
       XC     2,C                 ;Update PAST_DLY & PAST_PTH locations
       SAR    AR1,*,AR6           ;Pointer to ACCDIST --> PAST_DLY
       SAR    AR2,*,AR1           ;Pointer to DIST --> PAST_PTH
       MAR    *,AR1               ;ARP = AR1
       MAR    *+,AR2              ;AR1++ (circular addressing)
       MAR    *+,AR1              ;AR2++ (circular addressing)
ENDB0:
```

Pointers to the past path and delay states are stored in the PAST_PTH and the PAST_DLY tables. Since the decoder bases its decision on the path history of the previous 15 time periods, these two tables span 16 time periods (including the current time period). The length of each table is 128 words (16 time periods $\times$ 8 states). At each time interval, the GET_ACC_DIST routine adds new information to the tables and discards the oldest eight states. The format of these tables is shown below.

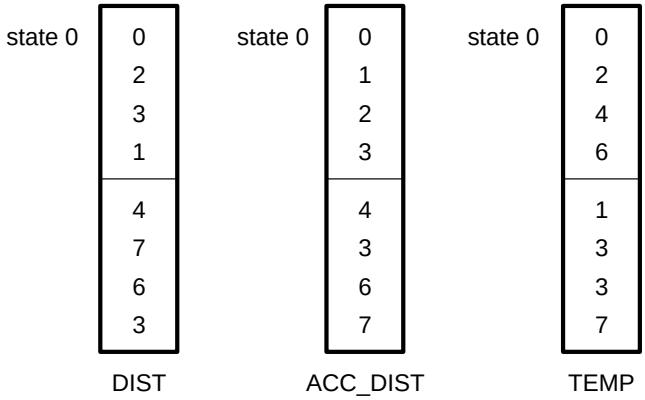
Figure 11. 128-Word Circular Buffers — Format of PAST_PATH and PAST_DLY Tables



Both tables are set up as 128-word circular buffers. Each of them is divided into 16 macro elements corresponding to 16 time intervals. Each macro element stores the state history of one time interval. A pointer is set up to indicate the location of the current time interval. By stepping through each macro element, a path can be traced backward in time.

Consider the V.32 trellis diagram again (see Figure 4). Notice that all even-numbered delay states of the current time interval have links to the first four delay states of the previous time interval. Similarly, all odd-numbered new delay states have links to the last four delay states. For instance, the new delay state 0 can be reached from the past delay states 0 – 3, and the new delay state 1 can be reached from the past delay states 4 – 7. So it is relatively simple to process even- and odd-numbered states in two groups. Furthermore, even-numbered delay states can be reached only by the first four path states, and odd-numbered delay states can be reached only by the last four path states.

Figure 12. DIST Table Structure



If the elements of the DIST table are set up as shown in Figure 12, all the path-state sequences can be generated from the same table. Four-word circular buffers are set up, comprising upper and lower halves of the DIST and ACC_DIST tables. By incrementing or decrementing through these circular buffers, path and delay-state sequences can be generated for each new delay state. (See the GET_ACC_DIST routine in the source code.) For each new delay state, only four past delay states and path states need to be accessed. The table for past delay states (ACC_DIST) is set up as a circular buffer so that after accessing four elements of the table, the pointer is automatically reset to the first element for the next iteration.

Once least-cost links to the eight delay states are identified and stored in appropriate tables by the MIN_ACC_DIST routine, the accumulated distance table ACC_DIST is updated with new accumulated distances. To avoid overflow, new accumulated distance is computed according to the following equation:

$$\text{new acc dist} = 0.9 \times \text{old acc dist} + 0.1 \times \text{dist} \qquad (4)$$

Note that this is a simple IIR implementation of a low-pass filter. The coefficients of Equation (4) can be modified to control the decay time of this low-pass filter.

There are eight independent tracks whose path histories are maintained in the PAST_PTH and PAST_DLY tables. The track that has the least accumulated cost (or distance) at this point is traced back for 16 time periods to determine the decoder output at that time. This task is performed by the GET_PATH routine as shown below. After 15 iterations, the delay state that corresponds to oldest link of the track is found.

```
   RPTB      TLOOP-1          ;for i=0,i<=15,i++
   MAR       *0+              ;offset by state for prev. time period
   LACC      *0-              ;get next pointer & reset AR0 to state 0
   SUB       #ACCDIST         ;subtract #ACCDIST to get next state
   SAMM      INDX             ;save next state
   SBRK      7                ;move AR0 7 locs back to avoid skipping CBER1
   SBRK      1                ;now AR0 is correctly positioned 1 time period
TLOOP:                        ;back (circular addressing)
```

The format of the PAST_PTH table is identical to the PAST_DLY table except that it contains previous path states instead of previous delay states. Also, the two tables are contiguous in data memory. Hence, by adding 128 to the pointer of the PAST_DLY table, corresponding path states can be accessed in the PAST_PTH table. The 3-bit path state (Y0, Y1, Y2) that corresponds to the oldest link is the output of the decoder. Since the path-state table DIST is not in a simple order, a short table look-up routine performs the descrambling of the output.

The 3-bit path state output by the Viterbi algorithm identifies a set of four points on the V.32 constellation. Of these four points, the one that is closest to the actual input (at that time period) should be selected. A

table must be set up in memory that stores the decoder input for the last 16 time periods so that the oldest input can be compared with these four constellation points. Fortunately, this cycle-consuming function can be avoided entirely by recalling that this comparison operation was done earlier (16 time periods back, to be exact) using the REGION table. If the pointer to the REGION table that identifies the eight closest constellation points (for each one of the path states) is available for that time interval, it is a simple matter to select a constellation point according to the path state number 0–7.

A 16-word circular table PATH_TBL is set up that stores pointers to the REGION table for the last 16 time periods. Since this table is always accessed sequentially (as opposed to randomly), the bit-reversed addressing mode is used to implement this circular buffer. The resulting 5-bit symbol (Y0, Y1, Y2, Q3, Q4) is the actual output. Obviously, Y0, the redundant bit, does not contain useful information (as it has already served its purpose) and can be discarded now.

Finally, the differential decoding algorithm (DIFF routine) converts Y1 and Y2 to Q1 and Q2. The following equations describe this decoding process:

$$Q1_n = Y1_n \oplus Y1_{n-1} \tag{5}$$

$$Q2_n = (Q1_n \bullet Y1_{n-1}) \oplus Y2_{n-1} \oplus Y2_n \tag{6}$$

A table look-up approach is taken here to decrease the execution time of this routine. A 16-word table DIFF_TBL is set up in memory. Each element of this table corresponds to a unique combination of bits $[Y1_{n-1}\ Y2_{n-1}\ Y1_n\ Y2_n]$, and it contains resulting decoded bits $Q1_nQ2_n$. Refer to the source code listing; see the *Code Availability* section on page 100. These two bits combined with $Q3_n$ and $Q4_n$ result in a 4-bit output symbol (Q1, Q2, Q3, Q4).

Performance Analysis

The V.32 encoder/decoder performance is evaluated on the TMS320C5x Software Development System (SWDS)[2]. The code benchmarks are also computed with the help of TMS320C5x SWDS. The transmission channel characteristics are simulated using the MATLAB software.

The input to the V.32 encoder is a binary data stream. As previously discussed, the stream is divided into 4-bit contiguous blocks called symbols. From the encoder standpoint, the input data is random, but the resulting 5-bit output symbols are not entirely random. Due to the convolutional encoding done on two bits of each 4-bit input symbol, output symbols are restricted within a subset of 32 symbols, depending on past symbol history.

The QAM modulator modifies the amplitude and the phase angle of the transmitted carrier signal according to each 5-bit symbol it receives. The communication channel imperfections distort the transmitted signal. White noise, impulse noise, and phase reversals are the most commonly encountered sources of channel distortion in telephony.

2 Since the writing of this paper, the ’C5x SWDS has been replaced with the ’C5x evaluation module (EVM) for code development.

The information is carried by the amplitude/phase of the transmitted carrier or, equivalently, by the I and Q components of it.

$$S(t) = amplitude \times cos\ (\omega t + phase) \tag{7}$$
$$= I \times cos\ (\omega t) + Q \times sin\ (\omega t) \tag{8}$$

The I and Q components of the signal received by a V.32 modem are corrupted with channel noise. If the channel is modeled as an AWGN-type channel, it is simple to simulate its effect on the signal by adding controlled Gaussian noise to the I and Q components independently. If N(t) is the zero-mean white noise signal, the signal-to-noise ratio (SNR) of QAM modulated signal S(t) is given by

$$SNR\ (dB) = 10 \times log_{10}\ [\frac{variance\ of\ S(t)}{variance\ of\ N(t)}] \tag{9}$$
$$= 10 \times log_{10}\ [\frac{E\ [S^2(t)]}{E\ [N^2(t)]}] \tag{10}$$

With the assumption that the I and Q inputs are statistically independent of each other, the SNR equation for the QAM modulated signal can be simplified as

$$SNR\ (dB) = 10 \times log_{10}\ [\frac{variance\ of\ I}{variance\ of\ N_i}] \tag{11}$$
$$= 10 \times log_{10}\ [\frac{variance\ of\ Q}{variance\ of\ N_q}] \tag{12}$$

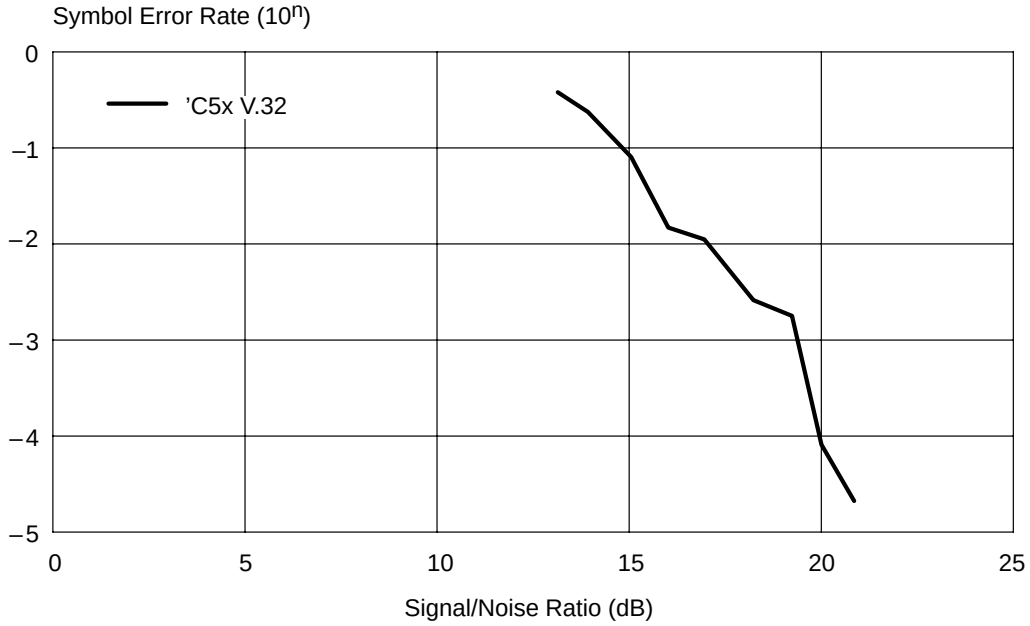
where N_i and N_q are additive noise signals for the I and Q input signals, respectively. Fixed-length sequences of I and Q are generated, and their sample variances are computed using the MATLAB software. For each desired value of SNR, required variances of N_i and N_q are calculated using Equations (9) through (12). Once the variances of N_i and N_q are determined, zero-mean Gaussian noise sequences N_i and N_q are generated by MATLAB. The input to the decoder program consists of I and Q data added to the respective noise sequences, N_i and N_q. This allows measuring the SNR performance of the decoder.

Figure 13 illustrates the performance of V.32 encoder/decoder code for various SNRs. These results are based on an input data sequence length of 4000 symbols. The yardstick for the performance measurement is symbol error rate (SER), which is defined as:

$$SER = \frac{total\ number\ of\ symbol\ errors}{total\ number\ of\ input\ symbols\ received} \tag{13}$$

Note that each input symbol consists of four bits.

Figure 13. White-Noise Impairment — Simulation Results



There are several factors that affect the performance of a Viterbi decoder in the presence of noise. One is the length of the path history analyzed by the decoder before selecting the most likely output. In general, it should be four or five times the encoder constraint length. Further increase in path history length gives only marginal improvement in performance.

Another performance factor is the decay time of the low-pass filter that is used to accumulate distance. By decreasing its time constant, the decoder can be made to respond to short noise bursts in the channel.

The table of eight accumulated distance values provides a convenient way of monitoring the performance of the decoder (and noise activity in the channel) in the absence of any prior knowledge of incoming data. Recall that these eight accumulated distance values allow the selection of minimum cost path at every symbol time interval. These values are also updated as new data is processed. During the relatively noise-free periods of transmission, it is observed that only one of the eight distance values remains significantly smaller than the rest. This in turn forces the decoder to select one particular path at every time interval. As the signal deteriorates, the difference between the minimum value and the rest of the table contents decreases. At some point, all distance values become so much alike that the decoder can no longer identify the correct path. This is the stage in which the BER increases considerably.

Table 1. Program Benchmarks

	Speed And Memory Requirements		
	Code Size (in Words)	Data Size (in Words)	CPU Loading per Symbol, Excluding Initialization (in Machine Cycles)
V.32 Encoder	79	10	90
V.32 Decoder	768	837	963-973

Table 1 shows the code size, data size, and CPU loading of the V.32 encoder/decoder program. This is by no means a fully optimized implementation of V.32 on the TMS320C5x. This code is written with the basic aims of demonstrating the capabilities of the TMS320C5x digital signal processor family and providing system designers with a head start on V.32 modem design. Table 2 and Table 3 present memory and speed requirements for various modules of the encoder and decoder. There are several speed-vs.-memory issues that can best be resolved by the system designer. The following paragraphs highlight some of them.

Table 2. V.32 Encoder Code

No	Function Name	Code Words	Machine Cycles
1	START	8	9
2	UNPACK	6	15
3	DIFF	11	12
4	ENCODE	20	21
5	PACK	12	20

Table 3. V.32 Decoder Code

No	Function Name	Code Words	Machine Cycles
1	RD_DATA	17	22
2	GET_RGN	108	80 - 112
3	GET_CUR_DIST	136	142
4	GET_ACC_DIST	228	489
5	MIN_ACC_DIST	36	65
6	GET_PATH	12	132
7	GET_SYM	11	15
8	DIFF	21	24

The approach that should be taken wherever speed-vs.-memory tradeoffs exist is to optimize for speed. For instance, the GET_RGN function uses a 416-word table to identify the eight closest constellation points. As discussed in the *Algorithm Implementation on the TMS320C5x* section on page 88, an alternate approach is to compute the distance between each constellation point and the current input and select the minimum distance point.

In the GET_CUR_DIST routine, distances corresponding to eight path states are computed by inline code, as opposed to looped code. This is done to facilitate the scrambled order of storage in the DIST table (see Figure 12). A considerable amount of program space may be released (approximately 100 words) if looped code is used here at the cost of additional machine cycles required to set up the loop and to access the DIST table.

In contrast with the GET_CUR_DIST routine, the GET_ACC_DIST routine is very difficult to implement in loop form. Each delay state computation itself makes use of iterative code. Furthermore, path-state sequences are unique for each delay state.

Summary

The TMS320C5x provides a powerful DSP engine for data-communication applications. This application report presents an efficient implementation of data encoding and decoding algorithms for V.32 modems on the TMS320C5x.

The encoder and decoder source code is designed with a generic hardware interface in mind. System designers can modify the input/output modules to suit their hardware requirements. The encoder algorithm is fairly straightforward. Most of the number crunching is required by the decoder algorithm. Although the code is written for the V.32 modem standard, a conscious effort is made to point out the V.32-specific and general-purpose Viterbi functions for adaptation of the code to any other Viterbi decoding scheme. For the same reason, the program flow is discussed in considerable detail.

Assembly code can be run on TMS320C50/1 in real time, without requiring any external memory. On a 35-ns TMS320C5x, the entire code only takes approximately 8% of the CPU time.

Code Availability

The associated program files are available from Texas Instruments TMS320 Bulletin Board System (BBS) at (713) 274-2323. Internet users can access the BBS via anonymous ftp at *ti.com.*

References

1. Michelson, A. M., and Levesque, A. H., *Error-Control Techniques for Digital Communications*, John Wiley & Sons, 1985.

2. Clark, G. C., and Cain, J. B., *Error Correction Coding for Digital Communications*, Plenum Press, 1981.

3. Proakis, J. G., *Digital Communications*, McGraw-Hill Book Company, 1983.

4. Forney, G. D., Jr., "The Viterbi Algorithm", *Proceedings Of The IEEE*, March 1973.

5. Viterbi, A. J., "Error Bounds for Convolutional Codes and An Asymptomatically Optimum Decoding Algorithm", *IEEE Transactions, Infinity Theory*, April 1967.

6. Lin, S., and Costello, D., *Error-Control Coding: Fundamentals and Applications*, Prentice-Hall, 1983.

7. *TMS320C5x User's Guide*, Texas Instruments, 1993.

8. "Report on the Work of Study Group XVII During the Period 1981–1984 Part III: Proposed New or Revised Recommendations in V-Series", *CCITT*, Malaga-Torremolinos, 1984.

9. *MATLAB User's Guide*, The Math Works, Inc., 1989.

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.