Chapter 4

Design of High Speed Re-configurable Co-Processor for Convolution-encoder and Viterbi-decoder Operations

4.1 Introduction

Convolutional-codes are used expansively in many applications in order to accomplish accurate and reliable data transfer without any overhead, these convolution encoder including radio, digital video, satellite communication and mobile communication [29][30]. These codes are repeatedly implemented in concatenation with a Reed-Solomon, particularly hard-decision code. Prior to turbo-codes, such constructions were the most efficient, coming closest to the Shannon. Convolutional codes are used extensively applications in communication system where it finds numerous applications. It exhibits good performance with high accuracy, low implementation cost, reliability in transfer of data. Data transfer includes radio, video, mobile and satellite communication. The Re-configurable convolution encoders operate on a serial bit stream of data used at transmitter side addition of this extra redundancy (parity) bits to it. Reconfigurable Viterbi-decoder most commonly uses the Viterbi-algorithm (VA) which is then uses the redundant bits to identify and/or correct as many serial bit stream errors at the receiver end to reconstruct the original sequence of data without any overhead[31][32] . There are other encoding algorithms for corresponding convolution encoded serial bit stream (for example, the sequential decoder) the Viterbi algorithm is a maximum –likelihood algorithm decoding [33].

Contrary to the hamming and RS codes seen above, which operate with blocks of bits or symbols, convolutional codes operate over a serial bit stream, processing one bit (or a small group of bits) at time.

An \((n, k, K)\) convolutional code converts \(k\) information bits into \(n\) channel bits, where, as before, \(m=n-k\) is the number of redundancy (parity) bits. In most applications, \(K=1\) and \(N=2\) are employed (That is, for every bit that enters the encoder, two bits are produced at the output), so the code rate is \(r=1/2\). The third parameter, \(K\), is called constraint length because it specifies the length of the convolved vectors (that is, the number of bits from \(x\), the input stream that are used in the computations). Consequently in spite of the encoding being serial, it depends on \(K\)-1 past values of \(x\), which are kept in memory.
The name “convolutional” comes from the fact that the encoder computes
the convolution between its impulse response (generator polynomial) and the input bit stream (see
equation 1.1 below). This can be observed with the help of the general architecture for
convolutional encoders depicted in Figure 4.1 (a).

Figure 4.1.(a) General architecture for \((m, k=1, K)\) convolutional-encoder showing a \(K\)-stage shift register at the top,
\(K\) encoder co-efficient \(h_{ij} = \text{‘0’ or ‘1’}\) per row , \(n\) rows, an array of modulo-2 adders (XOR gates), and an output
switch that assigns \(n\) values to \(y\) for every bit of \(x\) \((k=1\) in this case \); (b) A popular implementation with \(k=1, n=2,\)
and \(K=7\).

The upper part shows a shift register, whose input is \(x\), in which the last \(K\) values of \(x\) are
stored. the encoder’s output is \(y\). note the existence of a switch at the output, which assign \(n\)
values to \(y\) for every \(k\) \((=1\) in this case\) bits of \(x\), resulting in a code with rate \(k/n\).the \(K\) past
values of \(x\) ( stored in the shift register ) are multiplied by the encoder’s co-efficients \(h_{ij}\), which
are either ‘0’ or ‘1’, and then added together to produce \(y_i\) \((i=1,\ldots,n)\), that is :

\[
y_i = \sum_{j=1}^{k} h_{ij} \cdot X_j
\]

A popular implementation, with \(k=1, n=2\) and \(K=7\) (used by NASA in the voyager
spacecraft) is shown in Figure 4.1(b). Note again that \(K\) (Constraint length) is the number of bits
from x that participate in the computations of y (the larger K, the more errors the code can correct). The co-efficient s in this example are $h_1=(h_{11}, h_{12}, \ldots, h_{17}) = (1,1,1,0,0,1)$ and $h_2=(h_{21}, h_{22}, \ldots, h_{27}) = (1,0,1,0,1,1)$. Therefore

$$Y_1 = h_{11}x_1 + h_{12}x_2 + \ldots + h_{17}x_7 = x_1 + x_2 + x_3 + x_4 + x_7 \quad (1.1)$$

$$Y_2 = h_{21}x_1 + h_{22}x_2 + \ldots + h_{27}x_7 = x_1 + x_3 + x_4 + x_6 + x_7 \quad (1.2)$$

Note that the leftmost stage in both diagrams of Figure 4.1 can be removed, resulting in a shift register with K-1 flip-flops instead of K. in this case, the inputs to the adders must come from $(x, x_1, x_2 \ldots x_{(k-1)})$, that is , the current value of x must participate in the computations . This produces exactly the same results as the circuit with K stages, just one clock cycle earlier. However, it is unusual in synchronous systems to store only part of a vector, so the option with K stages might be preferred.

Both options (with K-1 and K flip-flops) are depicted in Figure 4.2, which shows the smallest convolutional code of interest , with $k=1, n=2, k=3$, $h_1=(1,1,1)$, and $h_2=(1,0,1)$. As an example, suppose that the sequence “101000…” is presented to this encoder (from left to right ). The resulting output is then y=”11 10 00 10 11 00…”

Any convolution-encoder can be represented by an FSM (finite state machine), so the state transition diagram for the encoder of Figure 4.2 (a) was included in Figure 4.2 (c) (the machine has $2^{k-1} = 4$ states, called A, B, C and D). In this case, the current value of x must be treated as an input to the FSM instead of simply a state-control bit, causing each state to exhibit two possible values for y; for example, when in state A, the output can be y=”00” (if x=’0’) or y=”11” (if x=’1’).
Figure 4.2 (a), (b) convolutional-encoder with $k=1$, $n=2$, and $K=3$ implemented with two and three flip-flop, respectively; (c) state transition diagram for the FSM in (a).

The FSM model can be used to determine the number of bits that the code can correct, given by $[(d_{\text{free}} - 1)/2]$, where $d_{\text{free}}$ is the code’s minimum free distance, that is, the minimum hamming distance between the encoded codewords. This parameter is determined by calculating the hamming weight ($w=$number of 1’s) in $y$ for all paths departing from $A$ and returning to $A$, without repetition. Due to the reason mentioned above (dual output values), examining all paths in the diagram of Figure 4.2 (c) is not simple, so either the extended version of the FSM diagram (Figure 4.2 (b)) is used (so all outputs have a fixed value) or a trellis diagram is employed.

The latter is shown in Figure 4.3 (a). Inside the circle are the state values (contents of the K-1=2 shift register stages), and outside are the values of $y (=y_1y_2)$ in each branch. Note that
each state has (as mentioned above) two possible output values represented by two lines that depart from each state; when $x='0'$, the upper line (transition) occurs, while for $x='1'$ the lower transition happens. From A to A, without repetition, are marked with thicker lines in figure 4.3 (b), where dashed lines indicate dead ends (repetitions).

Note that “repetition” means a transition that has already occurred, like from node B to node C ($B \rightarrow C$), not the repetition of a node (because each node allows two paths or two pairs of values for $y$). The shortest path in figure 4.3 (b) is also the one with the smallest $w$; the path is $A \rightarrow B \rightarrow C \rightarrow A$, which produces $y='11 10 11'$, so $w=5(=d_{\text{free}})$. Therefore, this code can correct $[(5-1)/2]=2$ errors. However, for the code to exhibit this free distance, a minimum run of bits is needed in $x$; for example, this minimum is $2K-1$ bits when the last $K-1$ bits are a tail of zeros.
Figure 4.3. General trellis diagram for the K=3 convolutional-encoder of Figure 4.2 (a), constructed with K-1=2 flip-flops in the shift register; (b) single-pass paths departing from A and returning to A (dashed lines indicated a dead end – a repetition).

As mentioned above, the other option to examine the paths from A to A is to use the expanded version (Figure 4.1 (b)) of the FSM. This case is depicted in Figure 4.3 (c), and the corresponding trellis is in figure 4.3(a). The system starts in A, with all flip-flops reset, progressing to B at the next clock transition if x=’1’ or remaining in an otherwise, with a similar reasoning applicable to the other states. Note that the output (y) in each state is now fixed, so following the machine paths is easier. Starting from A, and returning to A without repeating any node (now we look simply at nodes instead of transitions between nodes), we find that the following seven paths exist (the corresponding hamming weights are also listed):

Path 1: ABCEA (w=5)
Path 2: ABDGEA (w=6)
Path 3: ABDHGEA (w=7)
Path 4: ABDGFCEA (w=7)
Path 5: ABCFDGEA (W=7)
Path 6: ABDHGFC EA (w=8)
Path 7: ABCFDHGEA (w=8)
The shortest path, one of the longest paths, and also a dead path are marked with thicker lines in the trellis of figure 4.3 (b). Note that, as expected, the smallest \( w \) is still 5. In this version of the FSM the longest path (without repetition) cannot have more than eight transitions because there are only eight states, so \( w \leq 8 \). Likewise, because there are \( k=3 \) flip-flops in the shift register, at least \( K+1=4 \) transitions (clock cycles) are needed for a bit to completely traverse the shift register, so the shortest path cannot have less than \( K+1=4 \) transitions (note that path 1 above has only four transitions , while paths 6-7 have eight).

Even though convolution encoders are extremely simple circuits, the corresponding decoders are not. The two main types of decoders are called viterbi decoder [viterbi67] and sequential decoder (of which the fano algorithm [Fano63] is the most common). In the former, the size grows exponentially with \( K \) (which is consequently limited to \( \approx 10 \)), but the decoding time is fixed. The latter is more appropriate for large \( K \), but the decoding time is variable, causing large signal latencies , so the viterbi algorithm is generally preferred moreover, in some applications small convolutional codes have been concatenated with reed-solomon codes to provide very small error rates .in such cases , the decoder is referred to as a concatenated RSV ( reed- solomon- viterbi)decoder.

4.2 Viterbi-decoder

As mentioned above, the viterbi decoder is the most common decoder for convolutional codes. It is maximum-likelihood algorithm (the codewords with the smallest hamming distance to the received word is taken as the correct one), and it can be easily explained using the trellis diagram of figure 4.4.

![Trellis Diagram](image)

Figure 4.4. Example of sequence processed by the \( K=3 \) convolutional-encoder of Figure 4.1 (a) with two errors introduced in the communications channel.
Suppose that the K=3 encoder of Figure 4.1 (a) is employed and receives the bit stream \(x = "0110000..."\) (Starting from the left) depicted in Figure 4.4. Using the trellis of Figure 4.2 (a), we verify that the sequence \(y = "00 11 01 00 11 00 00..."\) is produced by the encoder. However, as shown in Figure 4.4 let us assume that two errors are introduced by the channel, causing \(y^* = "00 11 00 00 11 00 00..."\) to be actually received by the decoder.

The decoding procedure is illustrated in figure 4.5 where hard decision is employed (hard decision is called so because it is based on a single decision bit, that is, zero or one; viterbi decoders can also operate with soft decision, which is based on integers rather than on a zero-or-one value, leading to superior performance).

The encoder of figure 4.5 was assumed to be initially in state A. when the first pair of values (\(y^* = "00"\)) is received (figure 4.5(a)), the hamming distance (which constitutes the code’s metric) between it and the only two branches of the trellis allowed so far is calculated and accumulated. Then the next pair of values (\(y^* = "11"\)) is received (figure 4.5 (b)), and a similar calculation is developed with the new accumulated metric now including all four trellis nodes. The actual decisions will begin in the next iteration. When the next pair of values (\(y^* = "00"\)) is presented (figure 4.5 (c)), each node is reached by two paths. Adding the accumulated metrics allows each node to decide which node is reached by two paths. Adding the accumulated metrics allows each node to decide which of the two paths to keep (the path with the smallest metric is chosen). This procedure is repeated of values is presented. For the whole sequence, finally reaching the situation depicted in figure 4.5 (g), where the last pair of values is presented. Note the accumulated metrics in the four possible paths (one for each node). because A is the node with the smallest metric, the path leading to A is taken as the “correct” codewords. Tracing back (see the thick line in Figure 4.5 (h)), \(y^{**} = "00 11 01 01 11 00 00..."\) results, which coincides with the transmitted codeword \(y\), so the actual \(x\) can be recovered without errors in spite of the two errors introduced by the channel ( recall that the K=3 code can correct two errors even if they are close to each other, or it can correct various groups of up to two errors if the groups are far enough apart ).

To recover the original sequence during trace back, all that is needed is to observe whether the upper or lower branch was taken in each transition. The first traceback step in figure 4.5 (h)) is from A to A; note that, when going forward, from node A the encoder can only move to node A (if input =‘0’) or B (if ‘1’), so the first traceback step is from A to C; because node C is connected to nodes A (for input =‘0’) and B (for ‘1’), the third bit is again ‘0’. Following this reasoning for all seven traceback steps, the sequence “0110000” results (where the first traced-back bit is the rightmost), which coincides with the original sequence.
A fundamental issue regards the depth (L) of the decoder. The more the decoder iterates before deciding to trace back to find the (most likely) encoded sequence, the better. It was shown [viterbi67]; however, that little gain occurs after \( \approx 5K \) iterations. For the \( K=3 \) encoder, this means that \( L \approx 15 \) iterations should occur before any output is actually made available. Likewise, for the \( K=7 \) code of Figure 4.1(b)) about 35 iteration are required. Therefore, \( L \) determines the amount of delay between the received and the decoded codewords, as well as the amount of memory needed to store the trellis states. These two parameters (delay and memory) are therefore the ultimate deciding factors in the choice of \( K \) in an actual application.

Another observation regards the traceback procedure. If the sequence is simply truncated, then the starting node for traceback is the output with the smallest metric (nose A in the example of figure 4.5 (h)). However, determining the node with the smallest metric requires additional computation effort, which can be avoided with the inclusion of a trail of \( K-1 \) zeros at the end of the bit sequence, which forces the final node to always be A, so traceback can always start from it without any previous calculation.
Figure 4.5. Viterbi-decoding procedure for the sequence shown in figure 3.4 (with two errors) The last trellis shows the traceback, which produces the corrected code word.
4.3 Re-configurable structure for Convolution-encoder and Viterbi-decoder Operations

The Viterbi-decoder finds application in most of the basic data storage applications and also for communications systems. It is usually used for decoding convolutional codes, forward error correction (FEC) and in detection of baseband information for wireless systems.

There are usually three processes in a Viterbi-decoding: Add-Compare-Select (ACS), survivor path management and branch-metric calculations. The branch-metric is produced by the branch metric calculation and the distance between code word through the trellis diagram and the received sequence of data is measured. The general Viterbi-branch-metric equations are shown in figure.4.6. For the code-rate = 1/2 and code-rate = 1/3.

The branch-metric calculation making use of proposed structure is shown in figure.4.7. It consist a series of adder/subtractor operation. Execution of branch-metric calculation for code-rate value of 1/2 can be done with the operations illustrated in figure 3.9.6 and figure 3.9.7. Six adder/subtractor units are made use of (as shown in figure 4.6 and figure 4.7) for generation of 8 branch metrics with a code-rate of 1/3 as shown in figure. 4.7.

\[
\begin{align*}
\text{Bm00} &= (-x) + (-y) \\
\text{Bm01} &= (-x) + y \\
\text{Bm10} &= x + (-y) \\
\text{Bm11} &= x + y \\
\text{Code Rate} = 1/2
\end{align*}
\]

\[
\begin{align*}
\text{Bm000} &= (-x) + (-y) + (-z) \\
\text{Bm001} &= (-x) + (-y) + z \\
\text{Bm010} &= (-x) + y + (-z) \\
\text{Bm011} &= (-x) + y + z \\
\text{Bm100} &= x + (-y) + (-z) \\
\text{Bm101} &= x + (-y) + z \\
\text{Bm110} &= x + y + (-z) \\
\text{Bm111} &= x + y + z \\
\text{Code Rate} = 1/3
\end{align*}
\]

Figure. 4.6. General Viterbi-branch-metric equations

Figure.4.7. Proposed Branch-metric calculation.
The general ACS operation is shown in figure 4.8. The general trellis butterfly is illustrated in figure 4.9. The values of path-metric are added to the values of branch metric by the ACS operation. The minimum value between the two is selected by the ACS module. The proposed structure of trellis butterfly module is shown in figure 4.10. A trellis butterfly module operation has been performed for adding the branch-metric value by using the operations shown in figure 4.6 and for comparing of two path-metric values by using the operation in figure 3.9.8. Saving of the new path-metric to the memory is done based upon the sign of subtraction results.

Figure 4.8. Path-metric

Figure 4.9. trellis-butterfly module

Figure 4.10. Trellis-butterfly manipulation implemented using the proposed structure
A 32-bit adder and 16x16 multiplication operations can be made use of to model the proposed structure. About 5600 gate counts and 1 GHz operation clock/cycle are used to design the multiplier with the help of SEC-0.12 nm standard-Cell-Library. Therefore the proposed structure can support the high speed Re-configurable co-processor applicable for the next generation communication systems.

### 4.4 Implementation results and comparisons

VHDL has been used to model the proposed design of high speed re-configurable co-processor for Viterbi decoder and also convolution-encoder. Its synthesis is done with model-sim. The design of adder includes 7100 gates and an operation clock of 2 GHz using SEC – 0.16nm Standard-Cell-Library. Similarly the design of multiplier includes 5600 gates and an operation clock of 1 GHz using SEC – 0.16 nm Standard-Cell-Library. The waveforms obtained from model-sim for the proposed design of high speed re-configurable co-processor for Viterbi-decoder and convolutional-encoders are shown in figure 4.11 and figure 4.12.

![Waveforms obtained from model-sim software for the proposed design of high speed Re-configurable Co-processor for convolution encoder.](image)

Figure 4.11 Waveforms obtained from model-sim software for the proposed design of high speed Re-configurable Co-processor for convolution encoder.
The model-sim simulator has been used for the evaluation of the proposed Design of High Speed Re-configurable Co-processor for Viterbi Decoder Operations and Convolution Encoder for different existing communication algorithms using. Figure 4.13 and table 4.1 shows the performance comparisons between the conventional Star-Cores SC-140, TMS320C6x, TMS320C55x and proposed co-processor.

Table 4.1 Performance Comparisons between proposed co-processor and existing system

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC140 (Operations/Cycle)</th>
<th>Proposed (Operations/Cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MAC</td>
<td>2.5</td>
<td>4</td>
</tr>
<tr>
<td>Complex multiply</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>Complex MAC</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>FFT butterfly</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>Branch metric calculation (code rate : 1/2)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Branch metric calculation (code rate : 1/3)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Trellis butterfly</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Convolutional encoding (cycle), K=10,1/3,205bits</td>
<td>N.A</td>
<td>12X106</td>
</tr>
</tbody>
</table>
Figure 4.13. Graphical performance comparison between proposed co-processor and existing system.