An Efficient Implementation of Punctured Rate 1/4 Viterbi Decoder on an Analog Devices Blackfin DSP

By Yaniv Sapir and Yosef Stein, Analog Devices, Inc.

Abstract

In this paper we will present an implementation of a punctured rate 1/4 Viterbi decoder on Analog Devices Blackfin DSP. The implementation utilizes a few of the Blackfin's specialized instructions, allowing us to achieve a high efficiency and a fast implementation.

Introduction

The Viterbi Algorithm (VA) is a common means for forward error correction in digital communications. Due to its strength, it is applied to most digital communications standards today.

Error correction using VA consists on a convolutional encoder in the transmitter side and a Viterbi decoder (VD) on the receiver side. At the encoder side, a choice can be made to encode the incoming data stream into streams of higher rates. These are referenced as code rate 1/2, 1/3, 1/4, etc. This means that for every incoming message symbol (bit) there are 2, 3 or 4 encoded symbols, respectively. The encoded bits are mapped to the values of 1 and -1, representing logical value of 0 and 1 respectively. These values are used to modulate the transmission signal.

In order to reduce the transmission bandwidth requirement, puncturing of the encoded stream can be performed. In this process, some of the encoded symbols are omitted from the transmitted stream. At the receiver side, the omitted symbols are replaced by a value of 0 (indefinite value, with the same probability of being -1 or 1) which is used to fill in the omitted places. By its nature, a replaced punctured symbol is neutral and does not effect the VD's decision.

Viterbi decoders are commonly implemented as hardware modules or as software functions in a DSP embedded application. A software Viterbi decoder implementation is comprised of 2 major sections. These are the forward path section, where the most probable state transitions are chosen and stored in a survivors table, and a traceback section, where the original message is being decoded out of the survivors information.

The heart of the forward path is the Add-Compare-Select (ACS) operation. A typical implementation maintains a table of accumulators, one for each possible state. For each message symbol group, the distance (Hamming, if hard symbols, Euclidean if soft ones) between the received values and the theoretical, ideal values is calculated. This distance
is the current error (Branch Metric, BM) and is added to the state's error accumulator (State Metric, SM).

Following we will present an efficient implementation of a punctured, rate 1/4 Viterbi decoder on the Analog Devices Blackfin DSP. This DSP provides specialized instructions for the implementation of the VD forward path, making it very efficient in decode – almost as low as 1 cycle per state per bit.

We will examine an example case, the implementation of a punctured, rate 1/4 Viterbi decoder, with constraint length of 5. We assume the reader has some familiarity with the VA (many good available sources exist for VA theory elaboration, including Wikipedia).

The Problem at Hand

For code constraint length of \( K = 5 \), a convolutional encoder consisting of a LFSR of 4 taps is generating the output stream through a number of functions, or generator polynomials, which are linear combinations of the LFSR's taps and the current input. Thus, the system has \( 2^4 = 16 \) states. In our example, these polynomials are:

\[
\begin{align*}
G_1 &= S_0 + S_{-1} + S_{-4} \\
G_2 &= S_0 + S_{-2} + S_{-3} + S_{-4} \\
G_3 &= S_0 + S_{-1} + S_{-2} + S_{-4} \\
G_4 &= S_0 + S_{-1} + S_{-3} + S_{-4}
\end{align*}
\]

where \( S_{-j} \) is the value of LFSR tap \( j \), such that \( S_0 \) is the current input and \( S_{-4} \) is the least recent one. In order to save bandwidth, the implementers have chosen to use a puncturing scheme which essentially eliminates the fourth function, \( G_4 \). Thus, we effectively have a code rate of 1/3. This encoder is seen in Figure 1:

![Figure 1: Rate 1/3 convolutional encoder](image)

A careful selection of generator polynomials should yield a state transition table which is well balanced and distributed in a way that, for a given current state and current input, the generated code symbols enable the efficient restoration of the input sequence.
Given a received sequence (which is 3 times longer than the original message), we use a trellis diagram to calculate the maximum likelihood message string. The trellis diagram represents the possible state transitions and the ideal code symbols for these transitions. We then calculate the distance between these theoretical ideal values and the actual received values. The accumulation of these differences is the required Path Metric.

A one-stage trellis diagram for our case is seen in Figure 2. In the figure, state numbers are such that the lsb is the most recent bit (corresponding to $S_{-1}$) and the msb is the least recent bit (corresponding to $S_{-4}$). Thus, if we are at state #7 ($0111_b$) and the input $S_0$ is 1, the new state is #15 ($1111_b$). Similarly, state #10 with input bit of 0 leads to state #4. We see that there are exactly two source states leading to a given new state, based on the input message bit. Since the msb of the old state is pushed out and thus doesn't affect the new state, the two states are exactly 8 positions apart.

Below, the $p$ and $q$ values are the theoretical code symbols for each transition. $p_{ij}$ and $q_{ij}$ are the symbols for input bit $i (i \in \{0,1\})$ and polynomial $G_j (j \in \{1,2,3\})$. $P$'s are for the lower old state and $q$'s are for the upper old state. In the following analysis, we will designate the actual received values by $\rho_{ij}$.

![Figure 2: Valid state transitions](image)

When decoding the incoming symbol stream, the current state is unknown. So, we correlate the incoming values with the theoretical values for each transition by means of the Euclidian distance. This distance is the Branch Metric (BM). Iterating through the new state column, we have two BM values for each state, corresponding to two source states. We then add these BM's to the SM's of the old states, and compare the two results. The result with the lower sum (i.e., closer to the ideal values) is considered more likely to
be the actual state transition, and is then selected and stored as the new SM. The decision itself (as a bit – 0 for the low old state and 1 for the high old state, called survivor bit) is being stored in a dedicated Survivors array for use in the traceback stage.

We designate the two BM's as \( \beta_0 \) for the metric from the lower old state and \( \beta_1 \) for the metric from the high old state. They are calculated as follows:

\[
\beta_0 = (p_{01} - \rho_j)^2 + (p_{02} - \rho_j)^2 + (p_{03} - \rho_j)^2 = \sum_{j=1}^{3}(p_{0j} - \rho_j)^2 = \sum_{j=1}^{3}(2p_{0j}\rho_j) + \sum_{j=1}^{3}(\rho_j)^2
\]

\[
\beta_1 = (p_{11} - \rho_j)^2 + (p_{12} - \rho_j)^2 + (p_{13} - \rho_j)^2 = \sum_{j=1}^{3}(p_{1j} - \rho_j)^2 = \sum_{j=1}^{3}(2p_{1j}\rho_j) + \sum_{j=1}^{3}(\rho_j)^2
\]

These metrics are added to the old states' SM's. Then we compare and select the smaller of the two sums:

\[
\sigma_0 + \beta_0 < \sigma_1 + \beta_1
\]

Remembering that \( p_{0j}, p_{1j} \) are mapped to \( \pm 1 \), then \( \sum( p_{0j} )^2 = \sum( p_{1j} )^2 = 3 \). Additionally, the \( \sum(\rho_j)^2 \) term is the same in both sides of the equation. These terms repeat for each message bit as the forward path process advances. Hence, these two terms can be eliminated from the inequation. Doing so we are left with:

\[
\tilde{\sigma}_0 - \sum_{j=1}^{3}(2p_{0j}\rho_j) < \tilde{\sigma}_1 - \sum_{j=1}^{3}(2p_{1j}\rho_j)
\]

We can further eliminate the -2 multiplier by inverting the equation sign. We now look for the greater of the two terms:

\[
\tilde{\sigma}_0 + \sum_{j=1}^{3}(p_{0j}\rho_j) \quad > \quad \tilde{\sigma}_1 + \sum_{j=1}^{3}(p_{1j}\rho_j)
\]

Since \( p_{ij} = \pm 1 \), we essentially have the simple sum and difference of the received values \( \rho_j \). For the rate 1/3 code, we have 8 possible terms, corresponding to the 8 combinations of \( p_{0j} \) and \( p_{1j} \):

\[
\delta_0 = +\rho_1 + \rho_2 + \rho_3 \quad \delta_1 = +\rho_1 + \rho_2 - \rho_3 \quad \delta_2 = +\rho_1 - \rho_2 + \rho_3 \quad \delta_3 = +\rho_1 - \rho_2 - \rho_3
\]

\[
\delta_4 = -\rho_1 - \rho_2 - \rho_3 \quad \delta_5 = -\rho_1 - \rho_2 + \rho_3 \quad \delta_6 = -\rho_1 + \rho_2 - \rho_3 \quad \delta_7 = -\rho_1 + \rho_2 + \rho_3
\]

On the Blackfin processor these combinations are effectively calculated using the Add-on-Sign instruction. It is easy to see, though, that the terms in the second row above are just the opposites of the terms in the first row, and we can take this fact to our advantage in the implementation of the "Add" part of the ACS operation, utilizing the Quad-Add-
Subtract instruction. The comparison and selection of the greater metric is performed by the Blackfin's VIT_MAX instruction.

Implementing the Forward Path Section

For the forward path, a loop is iterated for each received message symbol (comprised of 3 encoded soft values plus one de-punctured value). For each iteration, we need to calculate the various $\delta$ terms and use them to compute the BM's. These are added to the SM's of previous states and the pairs are compared and survivors are selected. In order to manage the required data, we maintain a couple of circular buffers. Two buffers store the 'previous' and 'current' state metric values. The two buffers are alternating in their function in every iteration. A pair of pointers ($i_1$, $i_2$) point to the low and high source state leading to the current state number. These two are always half-array-size apart. A third pointer points ($i_3$) to the current resulting state number, where the surviving SM will be saved. This structure is seen in Figure 3:

![Structure of State Metric buffers](image)

Figure 3: Structure of State Metric buffers
A third buffer is defined to store the decision bits (0 for the low branch and 1 for the high branch), referenced to by \( p_5 \). The data structure definitions for the decoder is seen in Listing 1:

```c
// Memory bank A
.section L1_data_a;
.align 2
.byte2 _VD_Quantized_input[VD_N_bits * 4]
.align 2;
.byte2 _VD_Decision_History_Buffer[VD_N_bits * VD_N_states / 16 + 1];

// Memory bank B
.section L1_data_b;
.align 4096;
.byte2 _VD_SM_ping[VD_N_states];
.align 4096; // preventing stalls by directing mem accesses to different sub-banks
.byte2 _VD_SM_pong[VD_N_states];
.align 4;
.byte2 _VD_Branch_Metric_Multipliers[10] =
  0x0001, 0x0001, // ++
  0x0001, 0xffff, // +- 
  0xffff, 0x0001, // -+
  0xffff, 0xffff, // --
  0x0001, 0x0001; // ++
```

Listing 1: Data structures for the decoder

The pointers are initialized as seen in Listing 2:

```c
// Register assignments are given here in a short form to save space. The
// .l/.h notation should be used in actual implementation.
p1 = _VD_Quantized_input;
p5 = _VD_Decision_History_Buffer;

// branch metric multipliers array
i0 = _VD_Branch_Metric_Multipliers;
l0 = 0;
m0 = -16;

// Setup for initial and alternate state circular buffers
b1 = _VD_SM_ping;
li = b1;

b2 = b1;
li2 = _VD_SM_ping + VD_N_states_Buf_Sz / 2;

b3 = b1;
li3 = _VD_SM_pong;

// init state arrays with zeros here...
li1 = 2*(_VD_SM_pong - _VD_SM_ping);
li2 = li1;
li3 = li1;
m1 = _VD_SM_pong - _VD_SM_ping - 16;
m3 = _VD_SM_pong - _VD_SM_ping - (16 x 2 - 4);
li3 = m3;
```

Listing 2: Pointers initialization
When pointers and arrays are initialized, the forward path loop can take place. The loop iterates for \( N \) iterations, where \( N \) is the original message length. In each iteration, the first step is to compute the various \( \delta \) differences. For this step we use the Blackfin's special Add-on-Sign instruction. The code is seen in Listing 3:

```
.section L1_code;
// i0 = _VD_Branch_Metric_Multipliers
// p1 = _VD_Quantized_input
// p5 = _VD_Decision_History_Buffer; p4 = 2
// r7 contains 2 input symbols rho2 and rho3 loaded at previous iteration
// r2 contains decision bits from previous iteration
r0.h = r0.l = sign(r5.h) * r7.h + sign(r5.l) * r7.l
  || r5 = [i0++]  || w[p5+p4] = r2.h;         // rho2 + rho3
r1.h = r1.l = sign(r5.h) * r7.h + sign(r5.l) * r7.l
  || r5 = [i0++] ;                                // rho2 - rho3
r2.h = r2.l = sign(r5.h) * r7.h + sign(r5.l) * r7.l
  || r5 = [i0++] ;                                // -rho2 + rho3
r3.h = r3.l = sign(r5.h) * r7.h + sign(r5.l) * r7.l
  || r5 = [i0+m0] || r7 = [p1++];                 // -rho2 - rho3
r7.h = r5.l * r7.l, r7.l = r5.l * r7.l (IU);
// states 0,1
r0 = r0 + r7 ;                                // delta0 = rho1 + rho2 + rho3
r1 = r1 + r7 ;                                // delta1 = rho1 + rho2 - rho3
// states 2,3
r2 = r2 + r7 ;                                // delta2 = rho1 - rho2 + rho3
r3 = r3 + r7 ;                                // delta3 = rho1 - rho2 - rho3
```

Listing 3: Calculation of the 4 \( \delta \) differences

For the ACS operation we utilize the Blackfin's special Quad 16-bit Add-Subtract and Dual 16-bit VIT_MAX instructions. For our case of code constraint length of 5, there are 16 possible states in the state machine.

For each state we need to compute the two possible accumulated SM's, one from each possible source state (this is the Add operation) and then compare and select the most probable of the two to be the surviving path and new SM. The Quad-Add-Subtract instruction comes in our help in enabling us to take advantage on the \( \delta \) symmetry mentioned above. The VIT_MAX instruction allows us to perform 2 C-S operations at once and it is used to compute SM of two consecutive even and odd states. The code for the ACS step is seen in Listing 4:

```
// i1 = _VD_Old_State_Metric_Buffer_Low_Half
// i2 = _VD_Old_State_Metric_Buffer_High_Half
// i3 = _VD_New_State_Metric_Buffer
// m0, m3 are the state buffers loop offsets
// states 0,1
r7=r4++r0, r6=r4--;r0 || [i3+m3]=r6 || r4.h=w[i2++];
r6=vit_max(r7,r6) (asr) || r4.l=w[i2++];
```

Listing 4: ACS operation
Note that because the ACS specialized instructions deal with 16-bit entities, the implementer needs to make the trade-off between the soft-bit resolution and the length of the decoded frame, such that there is no overflow in the SM accumulators.

Implementing the Traceback Section

After building the decision history table and accumulating the State Metrics we can now start the traceback section for the actual message buildup. Since the accumulated SM's are calculated for the last received symbol, we now iterate through the decision table in a reverse order. First step is to find the state with the minimal accumulated path error, or in our case, the maximal accumulated correlation value. This is done by a simple max value search on the state buffer, as seen in Listing 5:

```
// p4 = _VD_New_State_Metric_Buffer
r2 = 0;                       // State # counter
r6 = 0;                       // Current state # of maximum metric
p2 = 15;                      // Number of states - 1
r0 = [fp-0x08];               // message length
r0 = W [p4++] (z);            // r0 holds the current max metric
// loop through state arrays to find maximum accumulated metric
loop _TB_FIND_MAX lc1 = p2;
loop_begin _TB_FIND_MAX;
    r2 += 1;
    r1 = W [p4++] (z);
    cc = r0 < r1;
    if cc r0 = r1;
    if cc r6 = r2;
loop_end _TB_FIND_MAX;
```

Listing 5: Finding state with maximum metric

Now that we found the state with maximum metric, this will be our starting point for the traceback. We now iterate through the table from its last column to the first, using the decision bit to comprise the old state number from the new state number. In order to make this as efficient as possible, we pack the current state number in the high half of a register and as we iterate, we shift the register to the right, so the lsb of the current state number is pushed to the lower half, thus building the decoded message. After 16
iterations, we get 16 message bits in the lower half, which are saved to the output buffer. Iterations continue until the whole decision buffer is processed. This structure is seen in Figure 4:

Figure 4: Traceback register data structure

Listing 6 shows the implementation of the traceback loop:

```
Listing 6: Traceback for decoding the message
```

**Conclusion**

In this article we demonstrated how to implement a punctured rate 1/4 (effectively 1/3) Viterbi decoder. Utilizing the specialized instructions of the Analog Devices Blackfin processor, Add-on-Sign, Quad-Add-Subtract and VIT_MAX, we managed to reduce the cycle count of the forward path to 26 cycle/bit. Then, with a careful selection of the data
structure used for the traceback, we could implement this section in a cycle count of about 4 cycles/bit.

Note that the program can be easily simplified and converted to decode a rate 1/2 code. This merely requires just the change of the BM calculations part, to have only two δ values instead of four.

Additionally, the traceback section is agnostic to the code's rate and can be reused to decode other rates as well.

About the Authors

**Yaniv Sapir** is a Senior DSP Engineer in the Advanced Technologies team in the GPDSP division of Analog Devices, developing embedded media algorithms and software for ADI's Blackfin processor. Yaniv holds a B.Sc. in aerospace engineering from the Technion, Israel Institute of Technology. He may be reached at yaniv.sapir@analog.com.

**Yosef Stein** is the principal manager of the GPDSP division Advanced Technologies team. He is working on the development of automotive and medical signal processing algorithms using Analog Devices' fixed point DSPs. Yosef holds a B.Sc. in Electrical Engineering, Technion. He may be reached at yosi.stein@analog.com.