How to Implement a Viterbi Decoder on the StarCore SC140

Application Note

Abstract

The application note describes how to implement an efficient Viterbi decoder on the StarCore SC140. It begins with an overview of convolutional encoding and Viterbi decoding. The overview is followed by a description of the StarCore SC140 special instructions that allow you to program an efficient Viterbi decoder. The note describes how to efficiently compute trellis butterflies, the basic computations in Viterbi decoding. Optimized code for portions of the Viterbi algorithm, including the branch metric calculation and the kernel, is provided. The note concludes with optimized assembly code for a complete Viterbi decoder according to the GSM TCH/FS standard, including performance benchmarks for the decoder running on the StarCore SC140.

ANSC140VIT/D
Alpha Release 7/18/00

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Table of Contents

Introduction

This chapter describes the topic, purpose, intended audience, and organization of this application note.

1.1 Topic and Purpose

Convolutional encoding with Viterbi decoding is one of the most popular forward-error-correction (FEC) techniques for error correction in communication systems. This application note describes how to implement the Viterbi decoder portion of this communication scheme on the StarCore SC140.

The StarCore SC140 Instruction Set contains several special instructions that greatly enhance the implementation and performance of the Viterbi decoding algorithm. These instructions take advantage of the unique bus structure and parallel arithmetic capabilities of the StarCore SC140. This application note will show you how to use the special instructions of the StarCore SC140 to program an efficient Viterbi decoder.

1.2 Audience

This application note is written specifically for application developers and programmers who want to learn how to implement an efficient Viterbi decoder on the StarCore SC140.

1.3 Organization

This application note contains the following chapters:

1. Introduction
2. Convolutional Encoding and Viterbi Decoding
3. Special Viterbi Instructions
4. Computing Trellis Butterflies
5. Optimizations to the Branch Metric Calculation
6. Optimizations to the Viterbi Decoder Kernel
7. Implementation in Endian Modes
8. Viterbi Decoder for the StarCore SC140
9. References

Introduction describes the topic, purpose, audience, and organization of this application note. You are reading the introduction right now.

Convolutional Encoding and Viterbi Decoding provides an overview of convolutional encoding and Viterbi decoding. It also describes the basic computational steps of the Viterbi decoding algorithm. Peruse this chapter before continuing to subsequent chapters.

Special Viterbi Instructions describes the special instructions of the StarCore SC140 that allow you to program an efficient Viterbi decoder. Spend some time to become familiar with these powerful instructions.

Computing Trellis Butterflies provides optimized assembly code for computing two trellis butterflies in parallel. This chapter begins by describing the trellis butterfly, the basic computation in Viterbi decoding, and the Add-Compare-Select (ACS) function, the principal function for computing trellis butterflies.

Optimizations to the Branch Metric Calculation describes how to optimize the Branch Metric (BM) calculation.

Optimizations to the Viterbi Decoder Kernel provides the memory map, optimized assembly code, pointer descriptions, and performance benchmarks for the Viterbi decoder kernel running on the StarCore SC140. The chapter also describes how to modify the Viterbi decoding algorithm for convolutional codes with constraint lengths greater than 5.

Endian Modes Support describes how the endian modes support of the StarCore SC140 works for Viterbi decoding.

Viterbi Decoder for the StarCore SC140 provides complete optimized assembly code for implementing a Viterbi decoder on the StarCore SC140 according to the Global System for Mobile Communications (GSM) Traffic Channel/Full-Rate Speech (TCH/FS) standard.

References lists sources where you can obtain more detailed information about Viterbi decoding and the StarCore SC140.

Convolutional Encoding and Viterbi Decoding

This chapter provides an overview of convolutional encoding and Viterbi decoding and describes the basic computational steps in the Viterbi decoding algorithm. It starts with a description of a typical forward-error-correction (FEC) communication scheme.

For a list of sources where you can obtain detailed information about convolutional encoding and Viterbi decoding, see “References” on page 67.

2.1 FEC Communication Scheme

In a typical communication scheme, a vector of symbols is transmitted by a source, travels over a communication channel where it is corrupted by noise and interference, and is received at a destination. The purpose of a FEC communication scheme is to add redundancy to the transmitted data so that any errors introduced by the communication channel can be corrected at the receiver. One of the most popular FEC techniques is convolutional encoding and Viterbi decoding (see Figure 1):

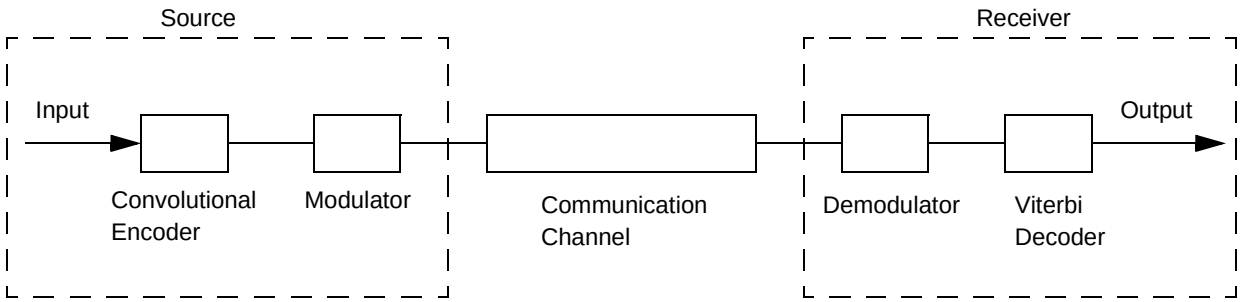

Figure 1. Typical FEC Communication Scheme

As illustrated in Figure 1, a convolutional encoder operates on an input data stream to generate encoded data that consists of the input data plus some redundant symbols, called parity symbols. A Viterbi decoder attempts to recreate the original stream of input data at its output by using the parity symbols to correct any errors introduced by the communication channel.

This application note focuses on the Viterbi decoder portion of this FEC communication scheme. To understand Viterbi decoding, you need a basic understanding of convolutional encoding.

2.2 How Convolutional Encoding Works

A convolutional encoder operates on an input stream of data symbols to generate an output stream of data symbols consisting of the original input data plus some redundant symbols, called parity symbols. A typical convolutional encoder comprises K stages of shift registers and one or more modulo-2 adders (see Figure 2).

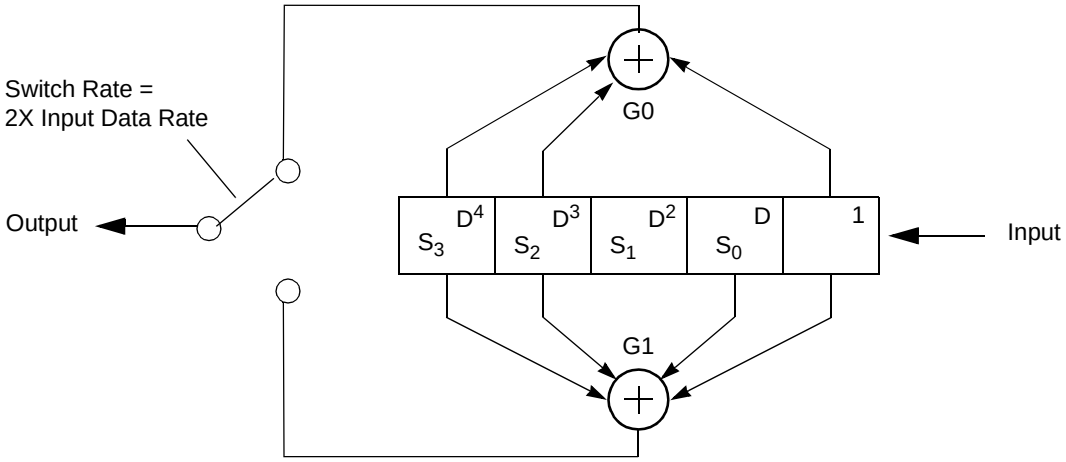

Figure 2. Convolutional Encoder for K=5 and R=1/2

As illustrated in Figure 2, the input symbols enter the shift register stages one symbol at a time on the right. The modulo-2 adders, or parity generators, use modulo-2 arithmetic to add the symbols of selected shift register stages to generate an encoded output symbol. Every time a symbol shifts into the input, the modulo-2 adders generate two encoded output symbols. Because there are more output symbols than input symbols, the output contains redundant symbols. The decoder can use the redundant symbols to correct any errors that occur during the transmission of the data over a communication channel.

Convolutional codes have a standard notation. A convolutional encoder with a code rate of R=m/n transmits n output symbols for every m input symbols. Each output symbol is a function of the current input symbol and the K-1 previous input symbols, where K is referred to as the constraint length of the convolutional encoder. The word (S_3,S_2,S_1,S_0) is the encoder state. The remaining symbol is referred to as the input symbol.

The convolutional encoder illustrated in Figure 2 has a constraint length of K=5 and a code rate of R=1/2. A constraint length of K=5 means that each output symbol depends on the current input symbol and the four previous input symbols. A code rate of R=1/2 means that for each input symbol, the modulo-2 adders generate two encoded output symbols.

The mathematical connections between the shift register stages and the modulo-2 adders of a convolutional encoder are typically described by generator polynomials. For the convolutional encoder illustrated in Figure 2, the upper and lower connections are represented by the following generator polynomials:

- $G0 = 1 + D^3 + D^4$
- $G1 = 1 + D + D^3 + D^4$

Each factor D represents one clock delay for the corresponding modulo-2 adder input.

2.3 How Viterbi Decoding Works

A Viterbi decoder attempts to recreate as its output the original stream of data that entered the convolutional encoder. To understand how a Viterbi decoder works, let's perform an exercise where we create some sample encoder input and use the encoder illustrated in Figure 2 to generate the corresponding encoder output. Then, let's attempt to decode the encoder output to recreate the original encoder input.

Note: The purpose of this exercise is to demonstrate the basic concept of Viterbi decoding. The actual details involved in a specific implementation of a Viterbi decoder may differ.

2.3.1 Generating Sample Encoder Inputs and Outputs

To get started, let's create some sample encoder input and generate the corresponding encoder output. Sample encoder inputs and the corresponding generated outputs appear in Table 1, where we assume that the encoder is initially filled with 0s.

Table 1: Sample Encoder Inputs and Outputs

Encoder Input	Encoder Output
0	00
1	11
1	10
0	01
0	11

2.3.2 Decoding the Encoder Output

To decode the encoder output, a Viterbi decoder attempts to recreate the set of encoder state changes, or transitions, that most likely generated the output. To see how a Viterbi decoder might go about this, let's try decoding the sample encoder output using the same assumption that the encoder used to generate it. That is, let's assume that the initial state of the encoder is 0000.

Before we can begin decoding, we need some sort of criterion to determine how well the outputs of a set of recreated encoder states compare with the decoder inputs. As a criterion, let's track the agreements between the recreated encoder outputs and the actual decoder inputs. The cumulative agreement of these inputs and outputs for a set of state changes, or path, leading to a particular recreated encoder state is called the *path metric* for that path. Incremental agreements for each state change, or branch, along a path are called *branch metrics*.

An encoder state tree defines the possible states and transitions of an encoder. Figure 3 shows a state in an encoder state tree and one of its possible transitions.

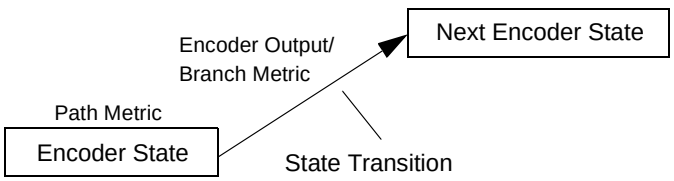

Figure 3. One State of the Encoder State Tree

Figure 4 shows the encoder state tree of our example encoder for the first four decoder input pairs. The boxes in the figure, called state boxes, contain the encoder states. Next to each arrow, or state transition, appears a number pair showing the encoder output for that state transition (that is, the output produced immediately before the state changes) and the branch metric associated with the transition.

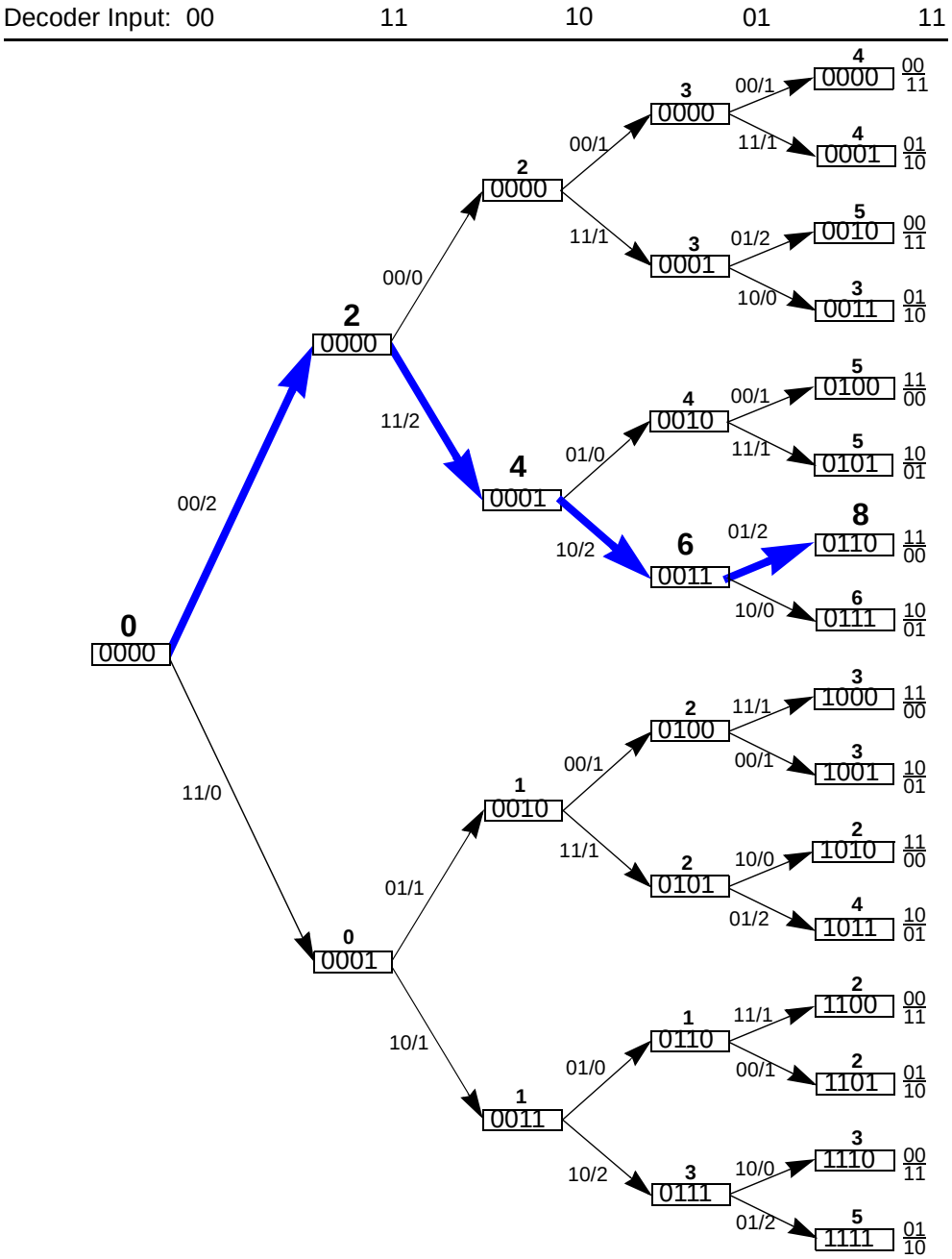

Figure 4. First Four Levels of the Encoder State Tree

The encoder state tree is structured such that transitions corresponding to an encoder input of 0 are always the upper path and transitions corresponding to an encoder input of 1 are always the lower path.

Determining the encoder output for any transition is straightforward: Load the encoder of Figure 2 with the state that appears in the state box just before the transition. Then, enter 0 or 1 for the input bit, depending on whether the transition is along an upper or a lower path. The encoder output is determined by the generator polynomials of the encoder.

For each transition, compare the recreated encoder output with the decoder input to determine the number of agreements between the bits. These agreements, referred to as branch metrics, are shown next to each branch in the encoder state tree. For each encoder state, keep track of the cumulative branch metrics to calculate the path metrics. The path metrics appear above each state box.

To select the encoder state sequence that most likely generated the decoder input, choose the path with the largest cumulative path metric. In Figure 4, the path with the largest path metric displays in bold.

The last step in decoding is to obtain the decoder output. Trace the path with the largest path metric back through the encoder state tree to its beginning. Then, follow the same path forward to generate the decoder output by sending 0 to the decoder output each time you traverse an upper transition and 1 to the decoder output each time you traverse a lower transition. Using this method yields a decoder output of 01100, which agrees with the sample encoder input.

This decoding method would work great and we could end our decoding exercise here if it weren't for a problem: As Figure 4 illustrates, the number of paths that the decoder must track doubles for each decoder input pair, which means that for any reasonable number of decoder inputs, the processing power and memory required to track all the paths is far too large for any practical decoder.

2.3.3 Collapsing the Encoder State Tree

To solve the problem of the ever-growing encoder state tree, consider that a decoder might not need to track all the possible paths in the tree. All it really needs to do is find the path with the largest path metric. If the decoder can somehow conclude, as it processes its way through the encoder state tree, that certain paths cannot ever have the largest path metric, it can ignore those paths in its future calculations.

To learn how the decoder might be able to eliminate certain paths in the encoder state tree, let's extend the encoder tree for one more pair of decoder inputs. The total number of states then doubles from 16 to 32. As it happens, the decoder can eliminate half of these states and maintain the number of states at 16 indefinitely.

Consider the selected pair of encoder states in Figure 5. Notice how extending the encoder tree to accommodate one more pair of decoder inputs yields states with five bits. However, our example encoder only requires five bits—four state bits and one input bit—to determine its output bits. Therefore, the fifth, or leading, encoder state bit is superfluous. To emphasize this in the figure, we separate the leading bits from the first four bits in each state box.

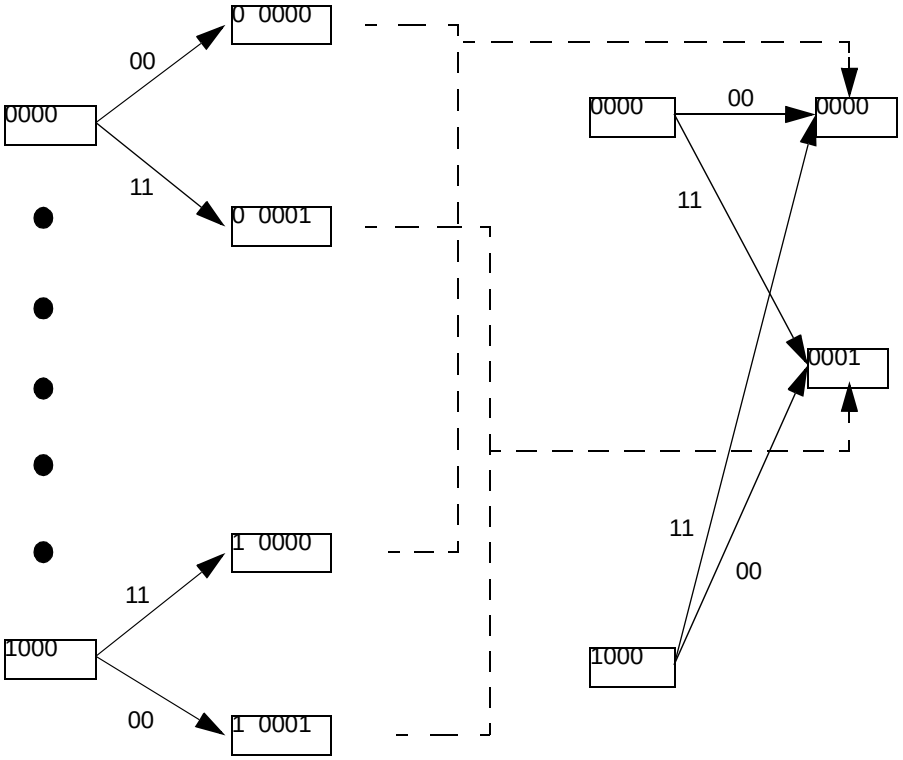

Figure 5. Collapsing the State Tree

Because the leading bits do not affect the encoder outputs, the decoder can ignore them. As a result, the 32 states introduced by extending the encoder tree collapse back to 16 states. Figure 5 illustrates how the states collapse.

However, we now have a new problem: Because each state now has two entering paths rather than one, the decoder must determine which path to keep for each state.

2.3.4 Choosing the Correct Path

To learn how the decoder chooses the input path for each state, consider the additive property of path metrics. Suppose you have two paths that extend from time i to time k and you want to find the path with the largest path metric. Also, suppose you are interested in the path metrics of these paths at another time j such that i < j < k. For a given encoder state S at time j, consider two paths P1 and P2 that extend from time i to time j and enter state S at time j. To get from state S at time j to another state at time k, consider two more paths, Q1 and Q2. This setup is illustrated in Figure 6:

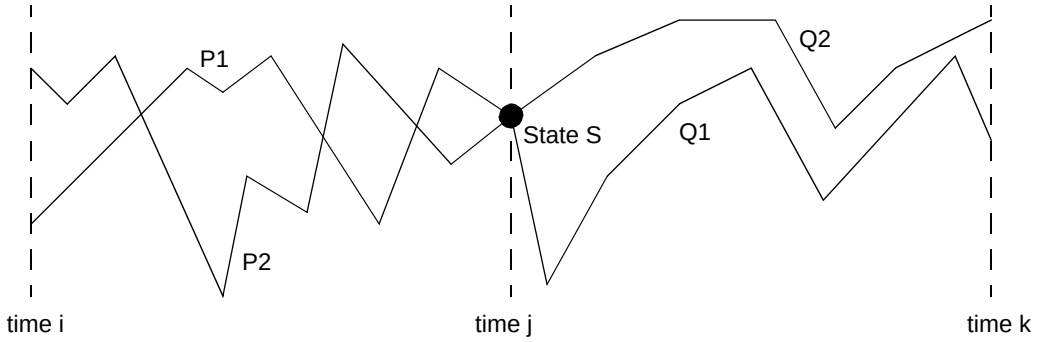

Figure 6. Choosing the Correct Path

Let the path metric for path P1 up to time j be PM1. Let the path metric for path P2 up to time j be PM2. Define QM1 and QM2 to be the partial path metrics for paths Q1 and Q2, respectively. A partial path metric is the contribution due to the partial path only. That is, QM1 is the path metric for path Q1 at time k minus the value of the path metric at state S at time j.

Finally, assume that PM1 > PM2. Then, any path containing path P2 cannot be the path with largest path metric for any time after time j. To understand why, note that the path metric for $P_1 \cup Q_1$ is PM1 + QM1. Similarly, $P_1 \cup Q_2$ has path metric PM1 + QM2, $P_2 \cup Q_1$ has path metric PM2 + QM1, and $P_2 \cup Q_2$ has path metric PM2 + QM2. Suppose $P_2 \cup Q_2$ is a candidate for largest path metric. Then, $P_1 \cup Q_2$ has a larger path metric because PM1 + QM2 > PM2 + QM2 (remember, we assumed that PM1 > PM2). This result holds true for *any* Q2. Therefore, if PM1 > PM2, we can eliminate path P2 from further consideration at time j—without waiting for the “future” decoder inputs from time j to time k!

This result has great significance for our decoding problem. As Figure 5 illustrates, each state is a junction for two entering paths. The number of paths doubles each time the decoder receives another pair of inputs, but the decoder can eliminate half of the paths each time as well. This is the basis of the Viterbi algorithm.

A Viterbi decoder keeps track of the path metrics for each state up to the current time. By eliminating the paths that cannot ever have the largest path metric, the decoder maintains the processing power and memory required to decode its input at a practical level. Because the decoder need only track each encoder state at any time, another type of diagram better illustrates the Viterbi decoding process: a trellis diagram.

2.3.5 Trellis Diagram

For Viterbi decoding, the most meaningful way to visualize the relationship between the input and output data sequences of a convolutional encoder is a trellis diagram (see Figure 7):

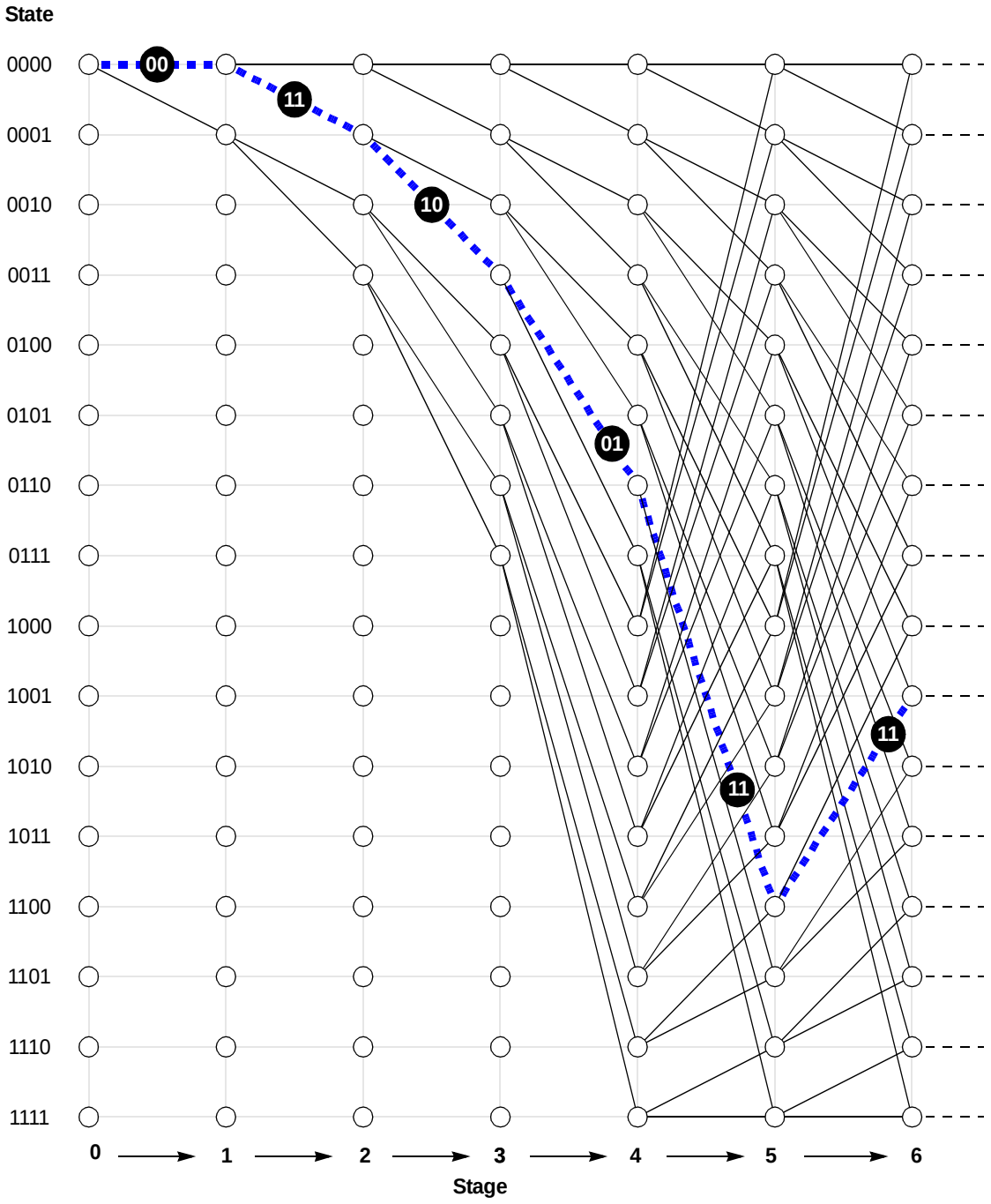

Figure 7. Trellis Diagram for a Convolutional Code with K=5 and R=1/2

As Figure 7 illustrates, a trellis diagram consists of nodes and branches. Each node in the trellis is labeled from 0000 to 1111. The label corresponds to the encoder state at that node in the trellis. The number of states of a convolutional encoder is $s = 2^{K-1}$, where K is the constraint length of the encoder.

Each branch in the trellis diagram represents a state transition, or a single input to the encoder. An input of 0 corresponds to the upper branch. An input of 1 corresponds to the lower branch.

To determine the encoder output for a branch, load the encoder with the state corresponding to the node just before the branch and then enter 0 or 1 for the input bit, depending on whether the branch is an upper or lower branch. The encoder output is determined by the generator polynomials of the encoder.

Any encoder input sequence traces a particular path through the trellis, yielding a corresponding output sequence. For example, if the input sequence to our example encoder is 011001, the corresponding output sequence is 001110011111. This particular input sequence is indicated by the dotted path in Figure 7. The corresponding encoder output symbols are shown above the branches of the path.

The structure of the trellis diagram clarifies the basic mission of a Viterbi decoder. Unlike the encoder state tree, the number of states in the trellis is bounded; therefore, the decoder only needs to keep track of a finite number of states. Also, after a certain stage in the trellis, each state has two entering paths. The decoder must choose which path to keep for each state at each stage of the trellis.

2.3.6 Computing Trellis Butterflies

As Figure 7 illustrates, the number of states that the Viterbi decoder must track reaches a constant after a certain trellis stage. When this happens, the trellis is said to have reached steady state. At steady state, each state has two entering paths. From this point on, the decoder must select which path to keep for each state at each trellis stage. This process of selection is referred to as updating the states.

For program efficiency, Viterbi decoders are typically programmed to update states in pairs. Updating the states in pairs is referred to as computing a trellis butterfly (see Figure 8). The states that the decoder updates are called the new, or destination, states. The states leading to the destination states are called the old, or source, states. To update a pair of destination states, the decoder needs the path metrics of two source states and the branch metrics associated with the branches leading to the destination states.

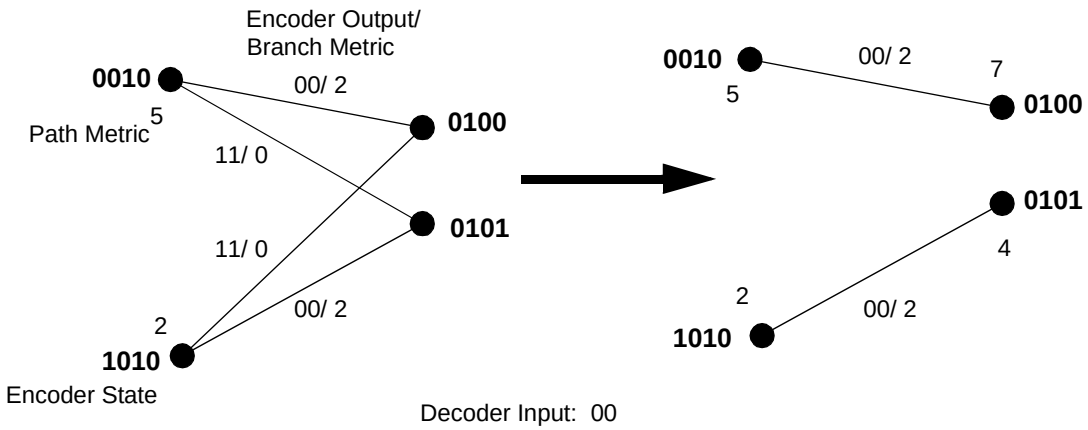

Figure 8. Trellis Butterfly Computation

Let’s consider how a Viterbi decoder updates the pair of states in Figure 8. The decoder first computes the branch metric for each branch leading from the source states to the destination states. For example, the lower branch from state 0010 has a recreated encoder output of 11. Since the decoder input is 00, none of the recreated encoder output bits agrees with the decoder input bits. Therefore, the decoder assigns a branch metric of 0 to this branch. The decoder performs a similar computation for the other three branches.

To select which path to keep for each destination state, the decoder adds the branch metric of each branch to the path metric of its source state, compares the resulting path metrics, and selects the paths with the largest path metrics as the survivor paths. The path metrics of the two surviving paths become the path metrics for the two destination states. The decoder updates all the destination state pairs at each trellis stage.

Note: In general, the source state pairs are not the same as the destination state pairs. For our example decoder, the source state pair 0ABC and 1ABC provides the path metrics that the decoder needs to update the destination state pair ABC0 and ABC1, where ABC is fixed for each update.

2.4 Viterbi Decoding with Soft Decision Inputs

To introduce Viterbi decoding in the most straightforward manner, previous sections of this chapter assumed that the inputs to the decoder were hard decision inputs. However, a major advantage of a Viterbi decoder is its ability to handle soft decision inputs. This ability greatly improves the fidelity of the decoder output relative to the encoder input.

This section describes how Viterbi decoders work with soft decisions:

- Differences between hard and soft decisions
- Why soft decisions more accurately cater to a typical communication channel
- How to compute branch metrics for soft decisions
- How a trellis works with soft decisions

2.4.1 Differences between Hard and Soft Decisions

Let's consider a hypothetical communication channel in which the transmitter generates a pulse of -1.0 volt for one time unit to signify a bit value of 1 and a pulse of +1.0 volt for one time unit to signify a bit value of 0. Of course, channel corruption can distort transmitted voltages, resulting in slightly altered to grossly distorted received voltages. Regardless of the level of channel corruption, the receiver must interpret the incoming voltage signals as either a bit value of 1 or 0. To interpret the incoming signals, receivers use a set of pre-programmed rules. In general, these rules fall into one of two categories:

- Hard decision rules
- Soft decision rules

Hard decision rules. Continuing with our hypothetical communication channel, let's take a look at the situation whereby the receiver operates according to a set of hard decision rules. According to this set of rules, the receiver interprets received voltages with positive values as a 0 and received voltages with negative values as a 1. That is, the receiver bases its decision on whether the received voltage represents a 1 or 0 solely on the sign of the voltage.

Although hard decision rules are simple, they suffer a major drawback: They don't consider the probability that the received voltage is the voltage that was actually transmitted. For example, we can easily intuit that a received voltage of -0.9 volts was most likely caused by a transmitted voltage of -1.0 volts with a channel corruption of +0.1 volts and that a received value of +1.1 volts was most likely caused by a transmitted value of +1.0 volts with a channel corruption of +0.1 volts. In other words, we can say that -0.9 volts most likely represents a transmitted bit value of 1 and that +1.1 volts most likely represents a transmitted bit value of 0.

However, for received voltages with magnitudes closer to 0.0 volts, such as -0.1 volts or +0.2 volts, we cannot be so confident which transmitted bit values the voltages represent, for it is almost equally probable that a received voltage of, say, -0.1 volts was caused by a transmitted voltage of -1.0 volts with a corruption of +0.9 volts as it was caused by a transmitted voltage of +1.0 volts with a corruption of -1.1 volts. In one case, the correct interpretation of the transmitted bit value is a 1. In the other case, the correct interpretation is a 0.

To summarize the above discussion, we can say that received voltages having magnitudes close to 1.0 volts are more likely to represent their intended bit values than received voltages having magnitudes close to 0.0 volts, and hard decision rules don't consider that likelihood.

Soft decision rules. A receiver that operates according to a set of soft decision rules makes use of the sign *and magnitude* information of a received voltage to estimate the transmitted bit value based on the probabilities of the received voltage falling within certain voltage ranges for each transmitted bit value.

2.4.2 Soft Decisions and a Typical Communication Channel

Because communication channels are generally subject to additive noise corruption, a number of transmissions at a given voltage level typically results in a range, or distribution, of received voltages. If the corruption is random, the range of received voltages will follow a normal distribution. To learn about the concepts behind soft decision decoding, let's consider a hypothetical communication channel that has the transmission characteristics illustrated in Figure 9:

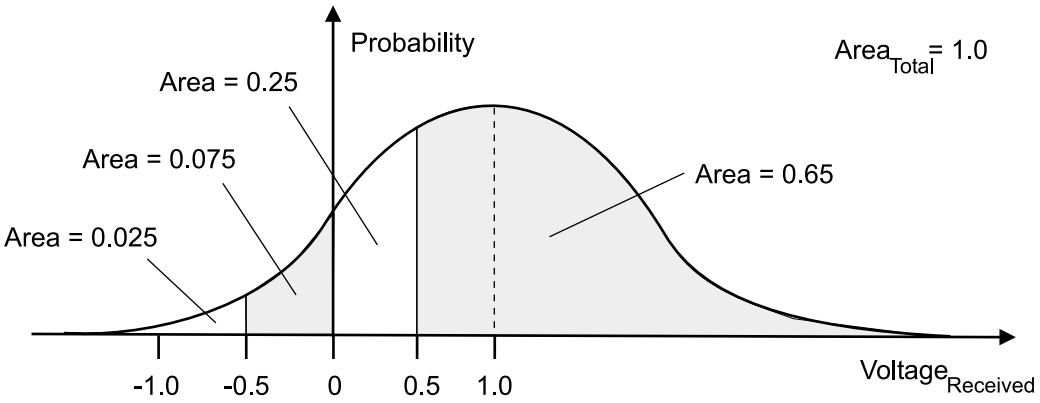

Figure 9. Received Voltage Distribution (for a transmitted voltage of +1.0 volts)

As Figure 9 illustrates, the received voltage distribution can be divided into a number of regions, the area of each region representing the probability that a received voltage will fall within its range. For example, for a transmitted voltage of +1.0 volts, the probability that the received voltage will be greater than +0.5 volts is 0.65.

For our hypothetical communication channel, the received voltage distribution for a transmitted voltage of -1.0 volts is a mirror image about the y-axis of the distribution shown in Figure 9. A communication channel that exhibits this type of symmetry in its transmission characteristics is referred to as a *discrete memoryless channel*.

The transmission characteristics of our communication channel have some adverse consequences for a receiver that operates according to hard decision rules. As Figure 9 illustrates, for a transmitted voltage of +1.0 volts, the received voltage will be positive with a probability of 0.9, and negative with a probability of 0.1. For a hard decision receiver, this means that the probability of a bit error is 0.1. Such an error rate is far too high for any practical decoding scheme.

Note: The transmission characteristics of a real communication channel may exhibit a tighter received voltage distribution (i.e., smaller standard deviation) than the one shown in Figure 9. Also, in a real channel, the received voltage distribution for a transmitted bit of 1 may not be symmetrical to the received voltage distribution for a transmitted bit of 0. Nevertheless, the concepts presented in this section are valid.

We can greatly reduce the bit error rate by exploiting our knowledge of the transmission characteristics of the communication channel to design a receiver. To do this, we need to design the receiver to make decisions about the received voltage according to a set of rules that considers the probability that the received voltage is correct. These rules are called *soft decision rules*. Here's a set of soft decision rules based on dividing the distribution of Figure 9 into four regions:

- If the received voltage is greater than +0.5 volts, the receiver interprets the voltage as a bit value of 0 with high confidence. We will designate this decision as 0H.
- If the received voltage is between 0.0 and +0.5 volts, the receiver interprets the voltage as a bit value of 0 with low confidence. We will designate this decision as 0L.

- If the received voltage is between -0.5 and 0.0 volts, the receiver interprets the voltage as a bit value of 1 with low confidence. We will designate this decision as 1L.
- If the received voltage is less than -0.5 volts, the receiver interprets the voltage as a bit value of 1 with high confidence. We will designate this decision as 1H.

For this set of soft decision rules, the possible soft decision outputs of the receiver for each transmitted bit are illustrated in Figure 10:

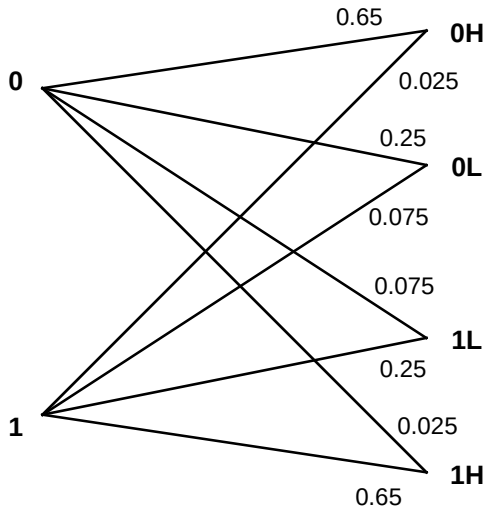

Figure 10. Binary to Quartenary Discrete Memoryless Channel

In Figure 10, the numbers adjacent to each path correspond to the probability of each receiver soft decision (i.e., 0H, 0L, 1L, 1H) for each transmitted bit (i.e., 0 or 1). These probabilities are also shown in Table 2.

Table 2: Transition Probabilities of the Discrete Memoryless Channel

	0H	**0L**	**1L**	**1H**
0	0.65	0.25	0.075	0.025
1	0.025	0.075	0.25	0.65

In Table 2, each entry represents the probability of a specific receiver decision given a certain transmitted voltage. For example, the entry in row 1 column 1 represents the probability that the receiver will interpret the received voltage as a bit value of 0 with high confidence (i.e., 0H) given that a bit value of 0 was actually transmitted.

Note: For simplicity of discussion, we chose to divide the received voltage distribution of Figure 9 into only four regions, which resulted in four soft decision rules. We could have chosen to divide the received voltage distribution into a larger number of regions, which would have resulted in a larger—and more accurate!—set of soft decision rules. In fact, Viterbi decoders typically receive a finely discretized set of soft decision inputs, each input representing the confidence that the receiver has in the received signal. For example, the confidence that a receiver has in a bit value transmission of a 1 or a 0 can be discretized into the following range of fixed point binary inputs to the decoder: FF to 7F.

2.4.3 Computing Branch Metrics for Soft Decisions

To apply the knowledge of the characteristics of our communication channel to a Viterbi decoder, we need to find a way to work with the soft decision inputs in our trellis. Remember, in Viterbi decoding, we are trying to find the most likely path through the trellis. That is, we are interested in the path whose sequence of transitions has the highest probability.

To find the probability of a sequence of transitions, we must find the probability for each transition and then compute their product. However, by working with the logarithms of the probabilities instead, we can take advantage of the following math property:

$$\log(a \times b) = \log a + \log b$$

From a computational standpoint, it is easier to work with the logarithms of the probabilities than with the probabilities themselves, because we can then add the probabilities for each transition to compute the probability for each path.

Note: The branch metric of a transition is the probability of the transition. The path metric of a path is the probability of the sequence of transitions that constitute the path. These statements are the key to understanding soft decision decoding!

The logarithms of the transition probabilities for our hypothetical communication channel are shown in Table 3.

Table 3: Logarithmic Values of the Transition Probabilities

	0H	0L	1L	1H
0	-0.19	-0.60	-1.12	-1.60
1	-1.60	-1.12	-0.60	-0.19

We are now in the position to begin the decoding process. We first compute the branch metrics for each trellis transition at each stage of the trellis. To do this, we work with the logarithms of the transition probabilities as follows:

For each trellis transition:

1. Determine the encoder output.
2. Using Table 3, find the transition probability of each decoder input (i.e., receiver soft decision) for its corresponding encoder output.
3. Add the transition probabilities for the pair of decoder inputs to compute the branch metric of the trellis transition.

The branch metric calculation for a pair of trellis transitions and a sample pair of soft decoder inputs is illustrated in Figure 11.

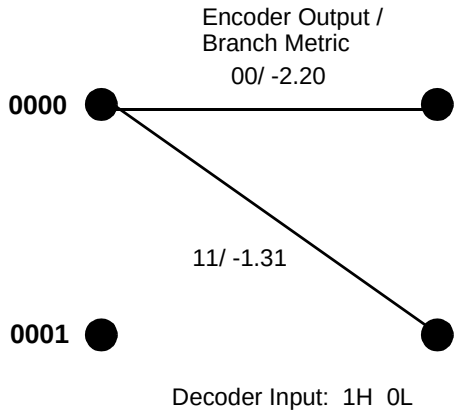

Figure 11. Branch Metric Calculation for Soft Decision Inputs (for K=5)

The branch metrics in Figure 11 are calculated as follows:

Upper transition:

For the upper transition, the encoder output is 00.

1. In Table 3, find the transition probability for a decoder input of 1H given an encoder output of 0. The entry is -1.60.
2. In Table 3, find the transition probability for a decoder input of 0L given an encoder output of 0. The entry is -0.60.
3. Add the transition probabilities of the two decoder inputs to calculate the branch metric. The result is -2.20.

Lower transition:

For the lower transition, the encoder output is 11.

1. In Table 3, find the transition probability for a decoder input of 1H given an encoder output of 1. The entry is -0.19.
2. In Table 3, find the transition probability for a decoder input of 0L given an encoder output of 1. The entry is -1.12.
3. Add the transition probabilities of the two decoder inputs to calculate the branch metric. The result is -1.31.

2.4.4 How a Trellis Works with Soft Decisions

The rest of the decoding process is the same as for the hard decision example presented earlier in this chapter:

1. Compute the branch metric for each transition at each trellis stage.
2. Add the branch metrics for each path to compute the path metrics.
3. Select the path with the largest path metric.
4. Trace the best path back to its beginning, and then follow the path forward, outputting 0 each time you traverse an upper transition, and 1 each time you traverse a lower transition.

Note: Because the branch metrics are logarithms with negative values, the path metric of the best path has the smallest magnitude.

The results of the decoding process for part of the trellis and a sample sequence of soft decoder inputs are illustrated in Figure 12.

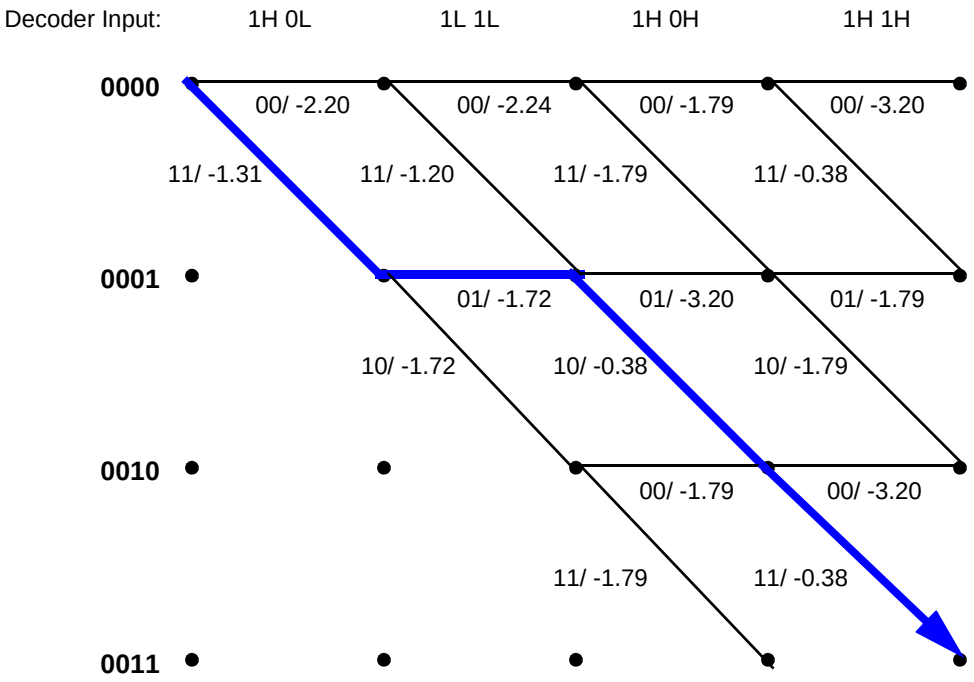

Figure 12. Partial Trellis for Soft Decision Inputs (for K=5)

In Figure 12, the encoder outputs and the branch metrics display next to each transition. The best path appears in bold.

Now that you understand how Viterbi decoding works in theory, let's explore how it works in practice. The next section describes the Viterbi decoder implementation presented in this application note.

2.5 Viterbi Decoding Algorithm

To decode a convolutional code, a decoder must use some type of decoding rule to find a path through the trellis, where each path in the trellis defines a unique decoder output sequence. Ideally, the decoder chooses a path that minimizes any discrepancies between the encoder output sequence derived from the trellis and the actual input to the decoder. Such a path most likely represents the actual encoder input, which is the data that the decoder is attempting to reproduce as its output.

The Viterbi decoding algorithm is a maximum-likelihood decoding algorithm for convolutional codes. Maximum-likelihood decoding produces an output sequence that maximizes the log-likelihood function, which represents the probability that the decoder output sequence matches the encoder input sequence.

Viterbi decoding is one of the most popular forward-error-correction (FEC) techniques because it is simple to implement and offers a large coding gain. The large coding gain results mainly from the ease with which the Viterbi decoding algorithm handles soft decision inputs received from the demodulator.

The Viterbi decoding algorithm consists of the following modules (see Figure 13):

- Initialization
- Branch metric calculation
- Viterbi decoder kernel
- Sub-blocks save
- Traceback

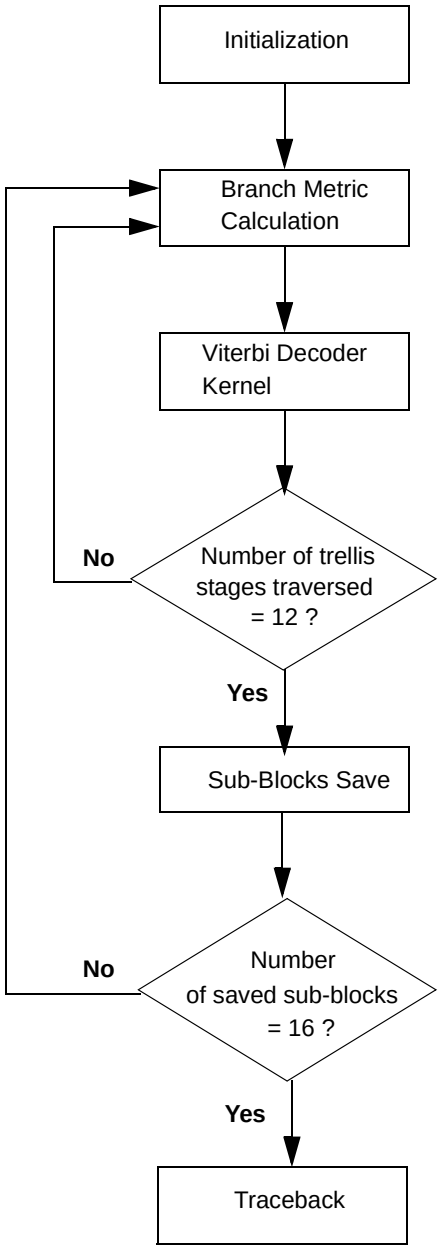

Figure 13. Flow Chart of the Viterbi Decoding Algorithm

2.5.1 Initialization

The initialization portion of the Viterbi decoding algorithm establishes the types of variables used by the algorithm and sets their initial values.

2.5.2 Branch Metric Calculation

The Viterbi decoding algorithm starts at state 0000 and works its way through the trellis. At each trellis node, the algorithm performs a branch metric calculation. For each branch entering the node, the algorithm calculates the number of agreements between the encoder output derived from traversing the branch and the current input to the decoder. The number of agreements is referred to as the branch metric.

Note: The Viterbi decoding algorithm provided for GSM TCH/FS integrates the branch metric calculation and the Viterbi decoder kernel.

The branch metric calculation used by the Viterbi algorithm presented in this paper is based on a well known method called *The Manhattan Metric*. According to this method, the algorithm for detemining the branch metric for each trellis transition is as follows:

To calculate the branch metric of a trellis transition:

1. Determine the encoder output bits.

   Note: The number of encoder output bits is 1/R, where R is the encoder rate.

2. For each encoder output bit, perform the following:

If . . .	Then . . .
The encoder output bit is 0	Add the respective decoder input (i.e., the receiver soft decision).
The encoder output bit is 1	Subtract the respective decoder input.

3. Accumulate the soft values of the decoder input bits according to their signs in step 2.

   Note: The result is the branch metric of the transition.

According to this algorithm, the branch metrics of any two competing transitions can be shown to have the same magnitude but opposite signs. Because of this, a trellis butterfly calculation can use a single branch metric magnitude (different signs) for all four state transitions. That is, when processing a trellis butterfly, we can calculate a single branch metric magnitude and determine its sign according to the state transition as follows:

A transition from this state . . .	To this state . . .	Yields this branch metric . . .
0XXX	XXX0	+BM
1XXX	XXX0	-BM
0XXX	XXX1	-BM
1XXX	XXX1	+BM

The rationale underpinning the Manhattan Metric is straightforward: The decoder input (i.e., the received signal) can have any fixed point binary value FF81 to 007F, where FF81 represents a strong 1 (-a) and 007F represents a strong 0 (+a). The value of 0000 is considered neutral.

For example, let's assume that the encoder output bits are 0 and 1, and that the receiver samples the values of +a and -a respectively (i.e., correct levels with highest probability). The branch metric is then calculated as +a - (-a) = +2a, which provides the highest positive branch metric (representing the highest probability) for the transition. Any other value for the received signal will result in a smaller branch metric for the transition. In other words, the larger the corrupting noise, the smaller the resulting branch metric.

In summary, the branch metric represents the probability of a transition, and it is calculated by simply manipulating the received signal values to the decoder, without translations into Gaussian probability tables, as are performed in traditional soft decision decoding (see "Viterbi Decoding with Soft Decision Inputs" on page 13).

2.5.3 Viterbi Decoder Kernel

The Viterbi decoder kernel represents the heart of the Viterbi decoding algorithm, determining how to proceed through the trellis to find a path that best represents the encoder input. To determine the best path, the kernel performs the following tasks for every node at each trellis stage:

1. Updates the path metrics for each of the two paths leading to the node.
2. Saves the path with the largest path metric and discards the other path.

Updating the path metrics. The kernel uses the branch metrics to determine the path metrics for each path entering a trellis node. The path metric of a path is the cumulative number of agreements between the encoder output derived from traversing that path through the trellis and the inputs to the decoder up to that stage in the trellis. To compute the path metric of a path entering a node, the kernel performs the following:

1. Retrieves the previous path metric of the path from memory.
2. Adds the previous path metric to the branch metric for that path from the latest branch metric calculation.
3. Updates the previous path metric in memory with the current path metric.

Saving the survivor paths. After the kernel computes the path metrics for each path entering a node, it compares the path metrics of the two paths and saves the path with the largest path metric. The other path is discarded. Retained paths are referred to as survivor paths. The number of survivor paths is equal to the number of encoder states.

2.5.4 Sub-Blocks Save

The Viterbi decoder kernel updates the path metrics and survivor paths at each trellis stage. A survivor path corresponds to each of the 2^{K-1} encoder states, where K is the constraint length of the convolutional code.

The kernel updates the survivor paths at each trellis stage by using the VSL instruction (see “Special Instructions for Viterbi Decoding” on page 27). To update the survivor paths, the VSL instruction shifts left the register holding the previous survivor path and inserts 0 or 1 in the least significant bit (LSB), depending on which state the new survivor path belongs to.

The sub-blocks save module assigns each of the 2^{K-1} survivor paths to a 16-bit word in either the high or low portion of a 32-bit register (see Figure 14).

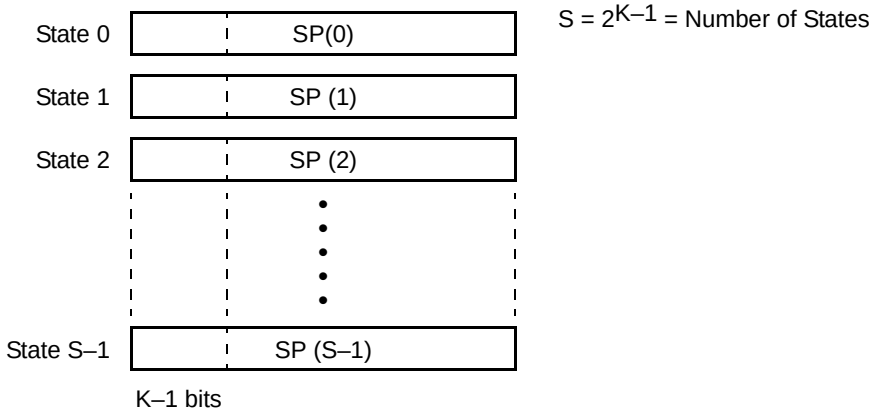

Figure 14. Sub-Blocks Save

As Figure 14 shows, the sub-blocks save module reserves some of the two bytes of memory—K-1 bits of it—for pointing to the previous state number. The module uses the remaining bits—16-(K-1) bits—for the actual data.

After the decoder traverses 16 stages of the trellis, the sub-blocks save module must save the registers that contain the survivor paths in memory so they will not be lost as the decoder continues to traverse forward through the trellis. In practice, however, the sub-blocks save module actually saves the survivor paths more often—every 16-(K-1) stages. The saved survivor paths are referred to as a sub-block.

2.5.5 Traceback

The sub-blocks save module works in tandem with the Viterbi decoder kernel to produce a series of sub-blocks, each sub-block containing a portion of the survivor paths for each state in the trellis (see Figure 15). There are 2^{K-1} partial survivor paths in each sub-block.

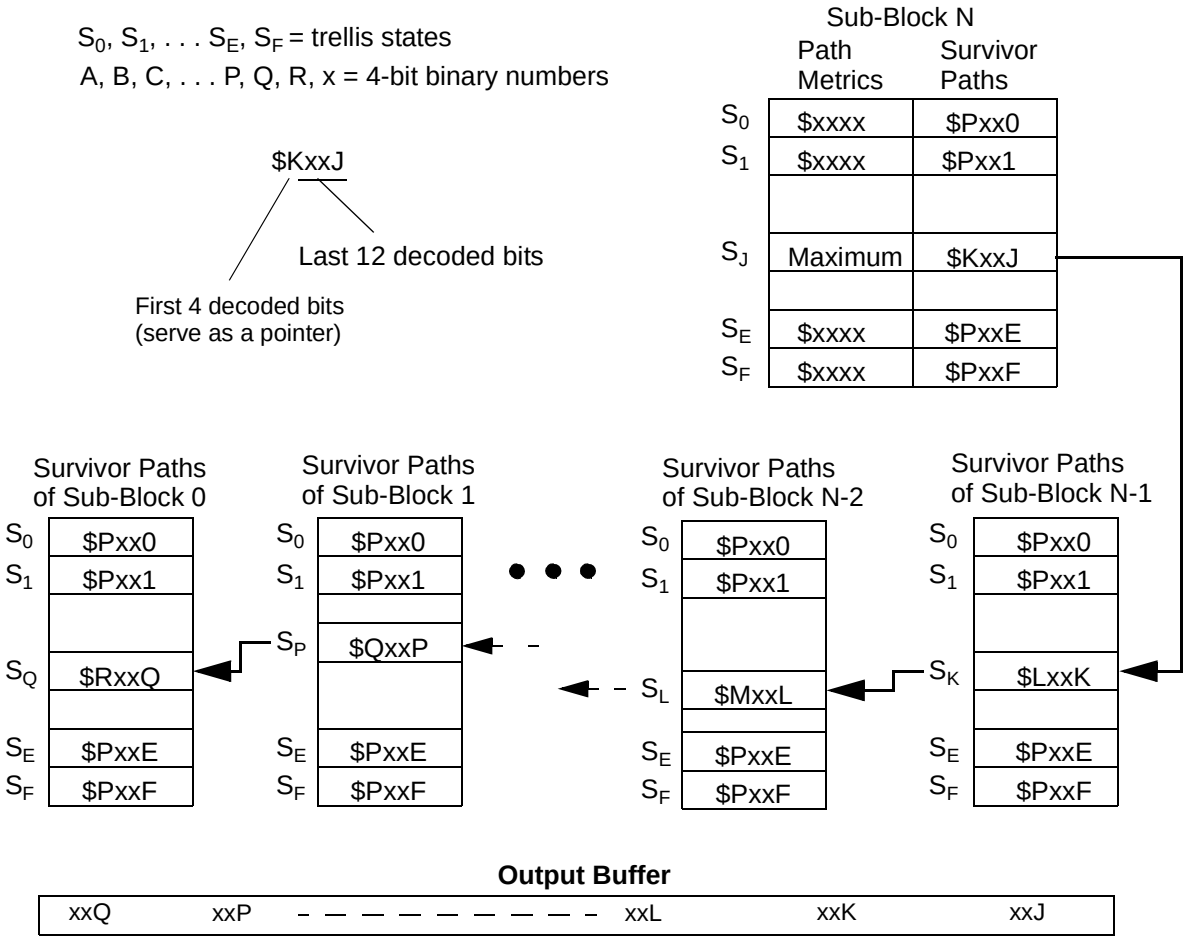

Figure 15. Traceback Function

The traceback module builds the maximum-likelihood survivor path by selecting as the final state of the trellis the state that corresponds to the survivor path with the largest path metric. The module then performs a traceback of the corresponding paths in each sub-block to build the maximum-likelihood survivor path. For GSM channels, the final state is specified as the zero state. The Viterbi algorithm for GSM TCH/FS produces a final state of zero by appending K-1 zeros, called tail bits, to each block of encoded bits.

As Figure 15 illustrates, the traceback module extracts a portion —12 bits—of the maximum-likelihood survivor path from each sub-block. The module can easily move from a word in a sub-block to the appropriate word in the previous sub-block because of the way the Viterbi algorithm is structured: Because a sub-block is saved every 16-(K-1) trellis stages, the last K-1 bits in each word are the same as the first K-1 bits of the appropriate word in the previous sub-block.

The first K-1 bits of each word correspond to the address of the next appropriate word. That is, the first word in a sub-block contains 0000 in its last K-1 bits; the second word in a sub-block contains 0001 in its last K-1 bits; and so forth. In short, the first K-1 bits of each word serve as a pointer to the appropriate word—that is, the one that belongs to the maximum likelihood survivor path—in the previous sub-block.

The traceback function comprises two steps:

1. Reads the last 16-(K-1) bits from the selected word in the current sub-block and copies them to the output buffer.
2. Reads the first K-1 bits from the selected word to determine the address of the appropriate word in the previous sub-block.

The traceback function repeats these two steps until it reaches the last sub-block, which is labeled sub-block 0. After the traceback function reaches the sub-block 0, it has copied the maximum-likelihood survivor path to the output buffer.

In summary, the traceback function extracts 12 bits of data from each sub-block and copies them to the output buffer, where they appear as a continuous stream of decoded bits.

Special Instructions for Viterbi Decoding

Viterbi decoding makes extensive use of the add-compare-select (ACS) function, which you will learn more about in the next chapter. The ACS function consumes the majority of the cycles during Viterbi decoding. To reduce the cycle counts consumed by the ACS function, the StarCore SC140 instruction set contains several special instructions that allow you to program an efficient ACS function. Spend some time to become familiar with these powerful instructions:

- ADD2
- SUB2
- MAX2VIT
- VSL.4F
- VSL.4W

To understand these instructions and how they help you to program an efficient Viterbi decoder, you need some basic information about the StarCore SC140 architecture. The StarCore SC140 architecture contains . . .

- Four data arithmetic logic units (DALUs)
- Two address generation units (AGUs)
- 16 general purpose 40-bit data registers (D0 through D15)
- 16 address registers (R0 through R15)

In a single clock cycle, the StarCore SC140 can perform four DALU operations on 32-bit operands and two AGU operations for transferring up to 64 bits of data.

Several instructions further increase the capabilities of the StarCore SC140 by dividing a 32-bit data register into two 16-bit registers, which are denoted by D.L for the low portion of the register and D.H for the high portion of the register. Each data register has an overflow byte.

For detailed information about the StarCore SC140 architecture and instruction set, refer to the *StarCore SC140 Core Reference Manual.*

3.1 ADD2

The ADD2 instruction performs two add operations on 16-bit operands in one DALU unit (see Figure 16).

3.1.1 Operation and Assembler Syntax

The following table provides the operation and the assembler syntax for the ADD2 instruction.

Operation	Assembler Syntax
Da.H + Dn.H → Dn.H Da.L + Dn.L → Dn.L	ADD2 Da,Dn

3.1.2 Description

The ADD2 instruction performs the following:

1. Adds the source 16-bit operand Da.L to the destination operand Dn.L and stores the result in the destination Dn.L.
2. Adds the source 16-bit operand Da.H to the destination operand Dn.H and stores the result in the destination Dn.H.

Note: Carry is disabled between bits 15 and 16. The extension Dn.E of the result is undefined.

3.1.3 Instruction Fields

{Da} All data registers [D0 . . . D15].

{Dn} All data registers [D0 . . . D15].

3.1.4 Use in Viterbi Decoding

The Viterbi decoder kernel uses the ADD2 and SUB2 instructions to update the path metrics for each state at each trellis stage (see Figure 16).

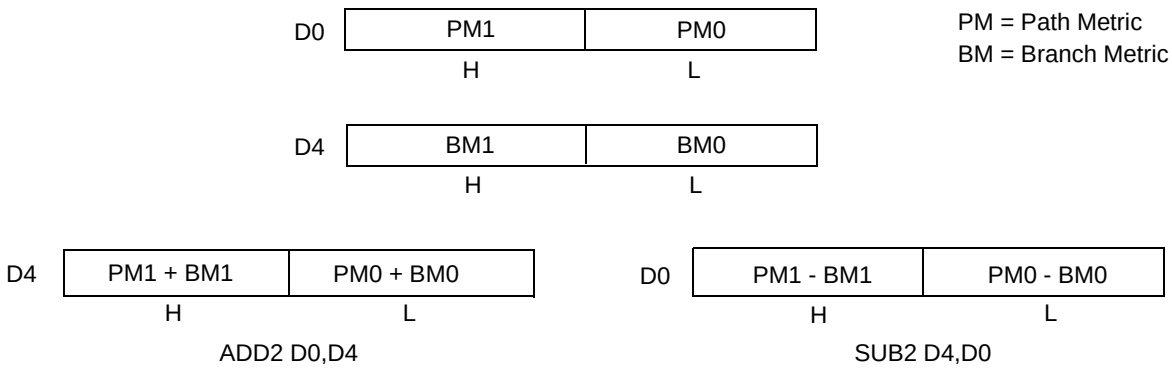

Figure 16. Register Schematic for ADD2 d0,d4 and SUB2 d4,d0

3.2 SUB2

This SUB2 instruction performs two subtract operations of 16-bit operands in one DALU unit (see Figure 16).

3.2.1 Operation and Assembler Syntax

The following table provides the operation and assembler syntax for the SUB2 instruction.

Operation	Assembler Syntax
Dn.H – Da.H → Dn.H Dn.L – Da.L → Dn.L	SUB2 Da,Dn

3.2.2 Description

The SUB2 instruction performs the following:

1. Subtracts the source 16-bit operand Da.L from the destination operand Dn.L and stores the result in the destination Dn.L.
2. Subtracts the source 16-bit operand Da.H from the destination operand Dn.H and stores the result in the destination Dn.H.

Note: Carry is disabled between bits 15 and 16. The extension Dn.E of the result is undefined.

3.2.3 Instruction Fields

{Da} All data registers [D0...D15].

{Dn} All data registers [D0...D15].

3.2.4 Use in Viterbi Decoding

The Viterbi decoder kernel uses the ADD2 and SUB2 instructions to update the path metrics for each state at each trellis stage (see Figure 16).

3.3 MAX2VIT D4,D2 and MAX2VIT D12,D10

The section describes these MAX2VIT instructions:

- MAX2VIT D4,D2
- MAX2VIT D12,D10

Use a prefix with the MAX2VIT D12,D10 instruction to encode the high bank registers.

3.3.1 Operation and Assembler Syntax

The following table provides the operation and the assembler syntax for these MAX2VIT instructions. VF0 and VF1 are Viterbi flags in the status register, SR.

Operation	Assembler Syntax
If D4.L > D2.L, then D4.L → D2.L and VF0 = 0 Else VF0 = 1 If D4.H > D2.H, then D4.H → D2.H and VF1 = 0 Else VF1 = 1	MAX2VIT D4,D2

3.3.2 Description

The MAX2VIT instruction performs the following (see Figure 17):

1. Subtracts the signed value of D4.L from the signed value of D2.L.
2. If the difference is negative (D4.L > D2.L), then VF0 = 0 and D4.L is transferred to D2.L. Otherwise, D2.L is not changed and VF0 = 1.
3. Subtracts the signed value of D4.H from the signed value of D2.H.
4. If the difference is negative (D4.H > D2.H), then VF1 = 0 and D4.H is transferred to D2.H. Otherwise, D2.H is not changed and VF1 = 1.

Note: The instruction transfers the extension Dn.E with the high portion according to the status of Viterbi flag VF1. The extension Dn.E does not influence subtractions. Only bits 15 and 31 determine the sign.

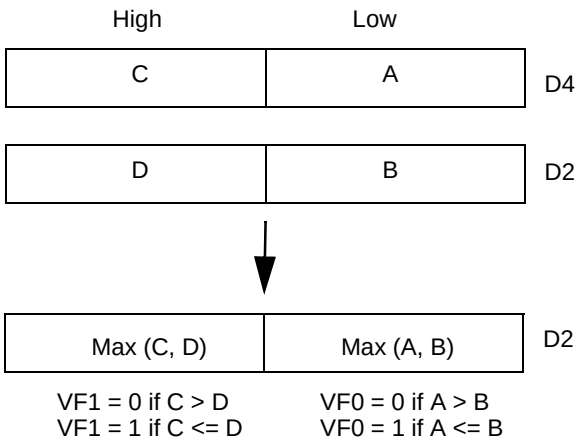

Figure 17. Register Schematic for the MAX2VIT Instruction

3.3.3 Use in Viterbi Decoding

The MAX2VIT instruction performs the *Compare* and part of the *Select* for an add-compare-select (ACS) function:

- Compares and selects the paths with the largest path metrics for two new states.
- Sets Viterbi flags VF0 and VF1, one for each new state, to indicate that the survivor paths have been updated.

For example, if D4.L is greater than D2.L, the instruction clears VF0. If the reverse is true, the instruction sets VF0. Viterbi flag VF0 indicates whether the survivor path corresponding to state 2j comes from old state J or old state J+S/2 (see “Computing Trellis Butterflies” on page 39).

3.4 MAX2VIT D0,D6 and MAX2VIT D8,D14

The section describes these MAX2VIT instructions:

- MAX2VIT D0,D6
- MAX2VIT D8,D14

Use a prefix with the MAX2VIT D8,D14 instruction to encode the high bank registers.

Note: The MAX2VIT D0,D6 and MAX2VIT D8,D14 instructions are similar to the MAX2VIT D4,D2 and MAX2VIT D12,D10 instructions. The only difference is that the instructions operate on different sets of source registers and Viterbi flags.

3.4.1 Operation and Assembler Syntax

The following table provides the operation and the assembler syntax of these MAX2VIT instructions. VF2 and VF3 are Viterbi flags in the status register, SR.

Operation	Assembler Syntax
If D0.L > D6.L, then D0.L → D6.L and VF2 = 0 Else VF2 = 1 If D0.H > D6.H, then D0.H → D6.H and VF3 = 0 Else VF3 = 1	MAX2VIT D0,D6

3.4.2 Description

The MAX2VIT instruction performs the following (see Figure 18):

1. Subtracts the signed value of D0.L from the signed value of D6.L.
2. If the difference is negative (D0.L > D6.L), then VF2 = 0 and D0.L is transferred to D6.L. Otherwise, D6.L is not changed and VF2 = 1.
3. Subtracts the signed value of D0.H from the signed value of D6.H.
4. If the difference is negative (D0.H > D6.H), then VF3 = 0 and D0.H is transferred to D6.H. Otherwise, D6.H is not changed and VF3 = 1.

Note: The instruction transfers the extension Dn.E with the high portion according to the status of Viterbi flag VF3. The extension Dn.E does not influence subtractions. Only bits 15 and 31 determine the sign.

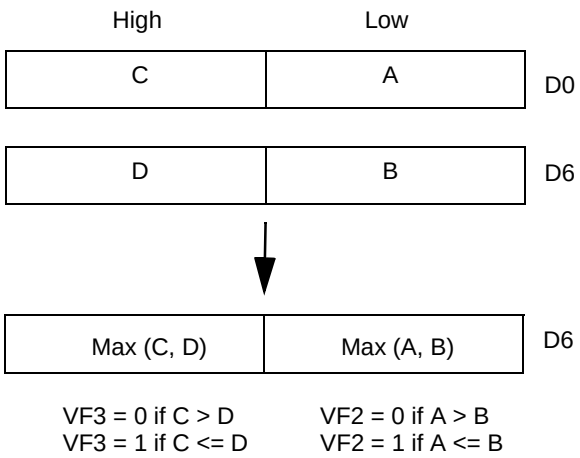

Figure 18. Register Schematic for MAX2VIT Instruction

3.4.3 Use in Viterbi Decoding

The MAX2VIT instruction performs the *Compare* and part of the *Select* for an add-compare-select (ACS) function:

- Compares and selects the paths with the largest path metrics for two new states.
- Sets Viterbi flags VF2 and VF3, one for each new state, to indicate that the survivor paths have been updated.

For example, if D0.L is greater than D6.L, the instruction clears VF2. If the reverse is true, the instruction sets VF2. Viterbi flag VF2 indicates whether the survivor path corresponding to state 2j comes from old state J or old state J+S/2 (see “Computing Trellis Butterflies” on page 39).

3.5 VSL.4F

The section describes the VSL.4F instructions:

- VSL.4F D2:D6:D1:D3, (Rn)+N0
- VSL.4F D10:D14:D9:D11, (Rn)+N0

Use a prefix with VSL.4F D10:D14:D9:D11, (Rn)+N0 to encode the high bank registers.

3.5.1 Operation and Assembler Syntax

The following table provides the operation and the assembler syntax of the VSL.4F instructions.

Operation	Assembler Syntax
D2.H → (address 0) D6.H → (address 1) If VF1 == 1, then (D3.H << 1,0) → (address 2) Else (D1.H << 1,0) → (address 2) If VF3 == 1, then (D3.H << 1,1) → (address 3) Else (D1.H << 1,1) → (address 3)	VSL.4F D2:D6:D1:D3, (Rn) + N0

In this table . . .

- The term << 1,0 means shift left 1 bit and fill the least significant bit (LSB, bit 16) with 0.
- The term << 1,1 means shift left 1 bit and fill the LSB with 1.

The memory word locations to which the addresses 0, 1, 2, and 3 refer depend on the endian mode:

Address	Big Endian Mode	Little Endian Mode
0	(Rn+2)	(Rn)
1	(Rn)	(Rn+2)
2	(Rn+6)	(Rn+4)
3	(Rn+4)	(Rn+6)

3.5.2 Description

The VSL.4F instruction writes four consecutive words taken from the high portion of the source data registers to the memory (see Figure 19). The instruction does not change the values in the registers.

The instruction first writes D2.H and D6.H to the location of the first two words in the memory. The order in which the instruction writes the words to memory depends on the endian mode.

The next two words that the instruction writes are . . .

- A left-shifted value of D1.H or D3.H, according to the status of Viterbi flag VF1. If VF1 is set, the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen. The LSB is filled with 0.
- A left-shifted value of D1.H or D3.H, according to the status of Viterbi flag VF3. If VF3 is set, the left-shifted D3.H is chosen. Otherwise, the left-shifted D1.H is chosen. The LSB is filled with 1.

The order in which the instruction writes these words to memory also depends on the endian mode.

For the address expression (Rn)+N0, the instruction multiplies the value in N0 by 8 to yield the actual address increment. For example, if N0=1, the instruction allocates eight bytes of memory for storing four 16-bit words.

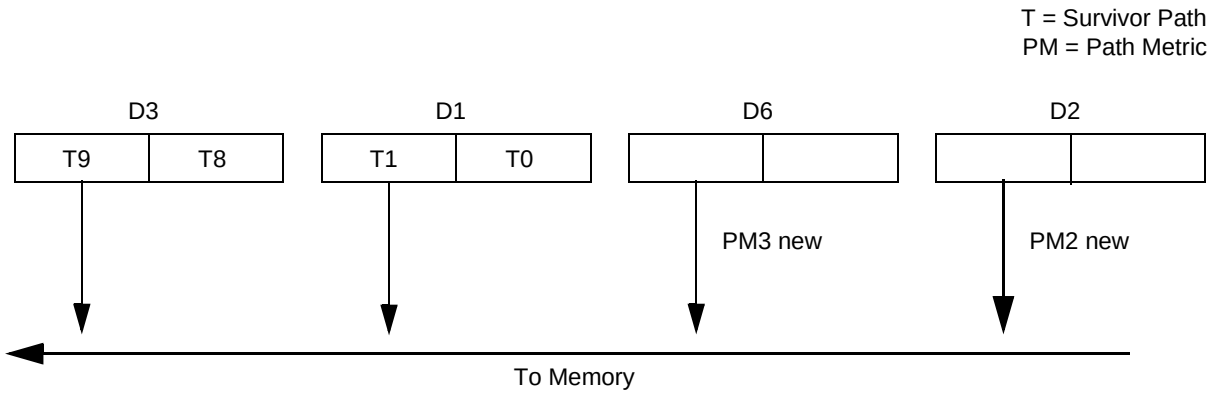

Figure 19. Register Schematic for the VSL.4F Instruction

3.5.3 Instruction Fields

{Rn} All address registers [R0 . . . R15].

3.5.4 Use in Viterbi Decoding

The VSL.4F instruction performs the *Select* part of the ACS function:

- Stores the new path metrics in memory.
- Updates and stores the survivor paths in memory.

For example, the instruction stores the new path metrics D2 and D6 in memory. The instruction also reads the Viterbi flags and determines which survivor path to select for each new state. The instruction updates the survivor paths by shifting left the register that contains the old path and inserting either 0 or 1 in the least significant bit (LSB), depending on which state the survivor path comes from.

3.6 VSL.4W

The section describes the VSL.4W instructions:

- VSL.4W D2:D6:D1:D3, (Rn)+N0
- VSL.4W D10:D14:D9:D11, (Rn)+N0

Use a prefix with VSL.4W D10:D14:D9:D11, (Rn)+N0 to encode the high bank registers.

Note: The VSL.4W and VSL.4F instructions are similar except for the following: The VSL.4W instruction operates on the low portions of the source registers, whereas the VSL.4F instruction operates on the high portions. Also, the operations of the instructions depend on different Viterbi flags.

3.6.1 Operation and Assembler Syntax

The following table provides the operation and the assembler syntax for the VSL.4W instruction.

Operation	Assembler Syntax
D2.L → (address 0) D6.L → (address 1) If VF0 == 1, then (D3.L << 1,0) → (address 2) Else (D1.L << 1,0) → (address 2) If VF2 == 1, then (D3.L << 1,1) → (address 3) Else (D1.L << 1,1) → (address 3)	VSL.4W D2:D6:D1:D3, (Rn) + N0

In this table . . .

- The term << 1,0 means shift left 1 bit and fill the least significant bit (LSB, bit 16) with 0.
- The term << 1,1 means shift left 1 bit and fill the LSB with 1.

The memory word locations that addresses 0, 1, 2 and 3 refer to depend on the endian mode:

Address	Big Endian Mode	Little Endian Mode
0	(Rn+2)	(Rn)
1	(Rn)	(Rn+2)
2	(Rn+6)	(Rn+4)
3	(Rn+4)	(Rn+6)

3.6.2 Description

The VSL.4W instruction writes four consecutive words from the lower portion of the source data registers to the memory (see Figure 20). The instruction does not change the values in the registers.

The instruction writes D2.L and D6.L to the location of the first two words in the memory. The order in which the instruction writes the words to memory depends on the endian mode.

The next two words that the instruction writes are . . .

- A left-shifted value of D1.L or D3.L, according to the status of Viterbi flag VF0. If VF0 is set, the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen. The LSB is filled with 0.
- A left-shifted value of D1.L or D3.L, according to the status of Viterbi flag VF2. If VF2 is set, the left-shifted D3.L is chosen. Otherwise, the left-shifted D1.L is chosen. The LSB is filled with 1.

The order in which the instruction writes these words to memory also depends on the endian mode.

For the address expression (Rn)+N0, the instruction multiplies the value in N0 by 8 to yield the actual address increment. For example, if N0=1, the instruction allocates eight bytes of memory for storing four 16-bit words.

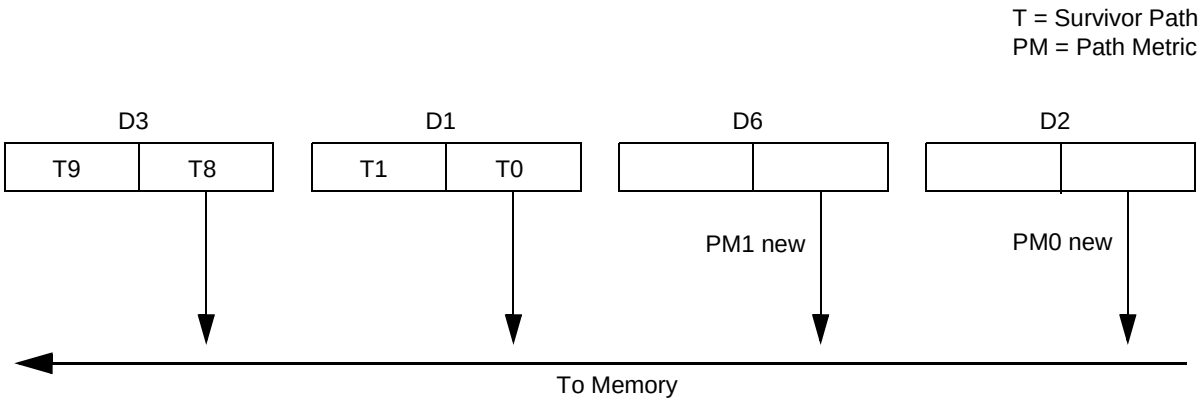

Figure 20. Register Schematic for the VSL.4W Instruction

3.6.3 Instruction Fields

{Rn} All address registers [R0 . . . R15].

3.6.4 Use in Viterbi Decoding

The VSL.4W instruction performs the *Select* part of the ACS function:

- Stores the new path metrics in memory.
- Updates and stores the survivor paths in memory.

For example, the instruction stores the new path metrics D2 and D6 in memory. The instruction also reads the Viterbi flags and determines which survivor path to select for each new state. The instruction updates the survivor paths by shifting left the register that contains the old path and inserting either 0 or 1 in the least significant bit (LSB), depending on which state the survivor path comes from.

Computing Trellis Butterflies

To design an efficient Viterbi decoder, programmers typically design the decoder to incorporate as much parallel processing as the processor will allow. Doing this increases processing speed, because many of the tasks that would be performed sequentually are instead performed simultaneously. One opportunity for taking advantage of parallel processing in Viterbi decoding is to design the algorithm to compute more than one trellis butterfly in parallel.

This chapter shows you how to use the special instructions of the StarCore SC140 to compute two trellis butterflies in parallel. The chapter begins by describing the trellis butterfly, the basic computation performed in Viterbi decoding, and the Add-Compare-Select (ACS) function, the principal function for computing trellis butterflies.

Before reading this chapter, review the special instructions of the StarCore SC140 that help you program an efficient Viterbi decoder (see “Special Instructions for Viterbi Decoding” on page 27).

4.1 Trellis Butterfly

A trellis diagram is a simple way to visualize the input and output sequences of a convolutional code (see “Convolutional Encoding and Viterbi Decoding” on page 3). The trellis diagram for a R=1/n convolutional code can be subdivided into a number of basic modules (see Figure 21).

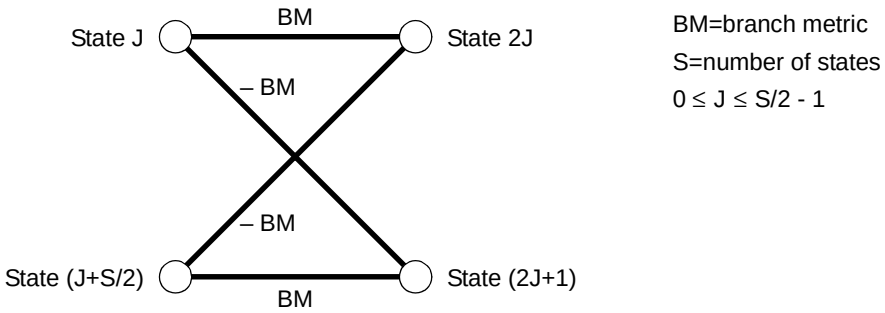

Figure 21. Trellis Butterfly

These modules, called trellis butterflies, illustrate the transitions between two old states at trellis stage i and two new states at trellis stage i+1.

Notice that the branch metrics for the upper and lower paths from each old state are equal and opposite. That is, the branch metric of the upper path from state J is equal and opposite the branch metric of the upper path from state J+S/2. The same is true for the lower paths from the two old states.

The symmetry of the branch metrics for the upper and lower paths arises from the nature of the trellis structure and the generator polynomials. The negative branch metrics arise from soft decision inputs that can be positive or negative and that span a range that is symmetrical about zero.

Each stage of a trellis diagram contains $2^{K-1}/2$ trellis butterflies, where K is the constraint length of the convolutional code. To visualize this feature of a trellis diagram, consider one stage of the trellis diagram for an R=1/n convolutional code with a constraint length of K=3 (see Figure 22).

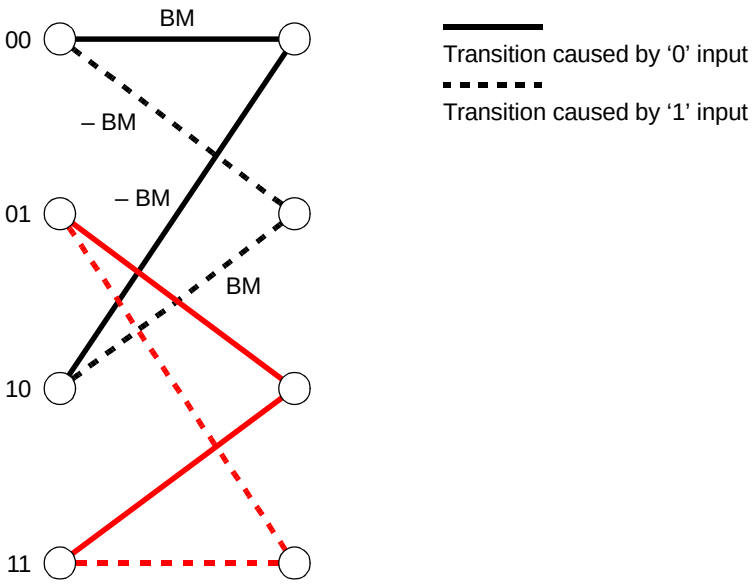

Figure 22. One Stage of a Trellis Diagram for K=3 and R=1/n

4.2 Add-Compare-Select Function

The Viterbi decoder kernel computes trellis butterflies by using a basic mathematical function called the Add-Compare-Select (ACS) function (see Figure 23).

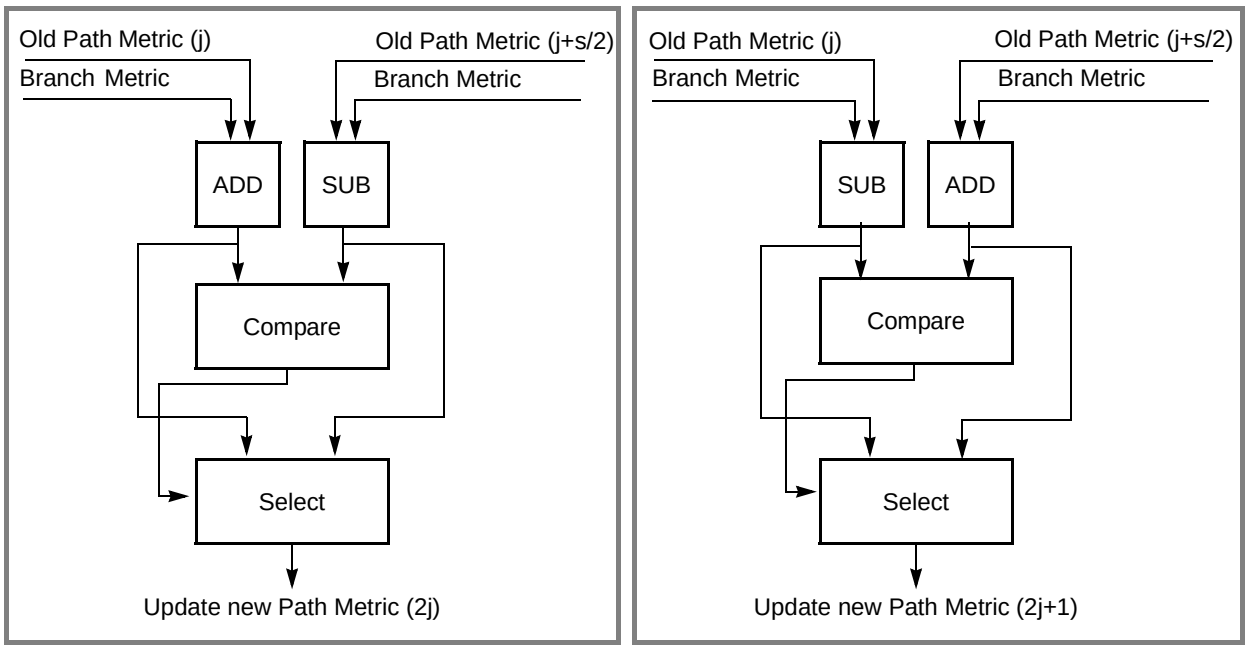

Figure 23. Trellis Butterfly Computation Using Two ACS Functions

Since one ACS function selects the transition to one new state, the Viterbi decoder kernel must perform two ACS functions to compute one trellis butterfly. Computing a trellis butterfly comprises these operations:

1. Reading the path metrics and survivor paths of states j and j+s/2 at stage i.
2. Computing the path metrics of states 2j and 2j+1 at stage i+1.
3. Comparing the two path metrics and selecting the path with the largest path metric.
4. Storing the updated path metrics and survivor paths.

4.3 Computing Two Trellis Butterflies

For efficient processing, the Viterbi decoder kernel presented in this application note is programmed to compute two trellis butterflies in parallel. To compute two trellis butterflies in parallel, the kernel uses the parameters defined in Figure 24.

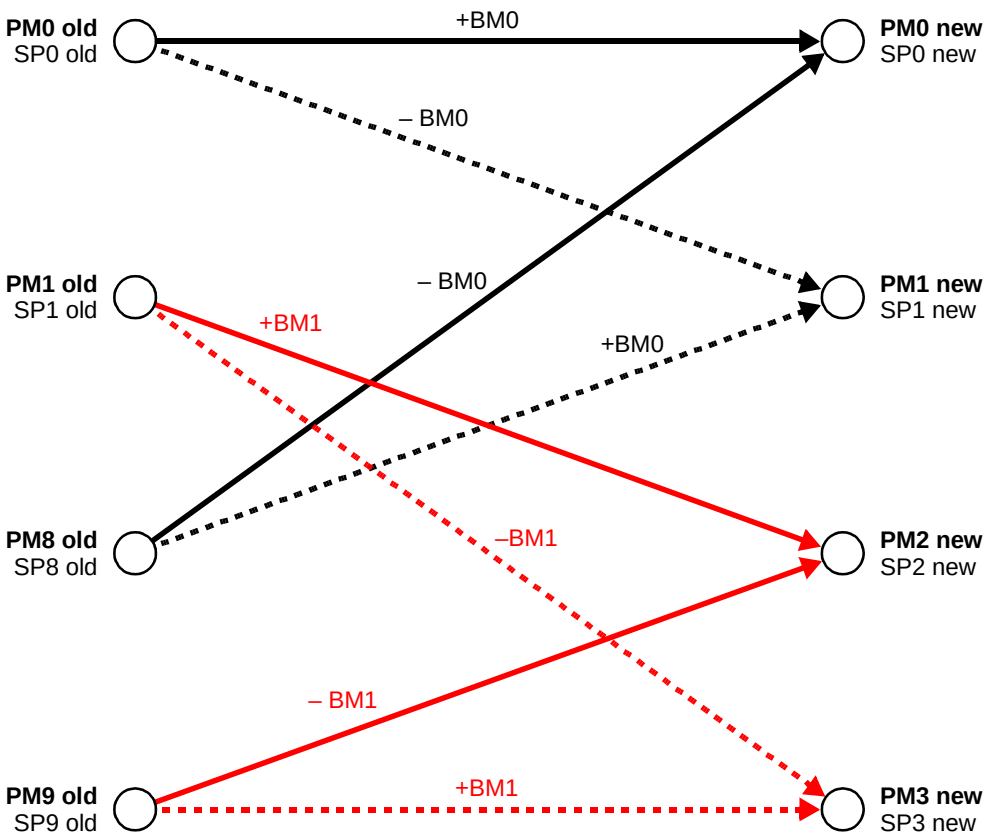

Figure 24. Computing Two Trellis Butterflies (for K=5 and R=1/n)

To design the Viterbi decoder kernel to efficiently compute two butterflies in parallel, you have two methods:

- In-place method
- Dedicated register method

4.3.1 In-Place Method

To use the in-place method to compute two trellis butterflies in parallel, design the Viterbi decoder kernel such that the registers that hold the branch metrics (BMs) also hold the new path metrics (PMs). Figure 25 shows the initial StarCore SC140 register map for computing two trellis butterflies using the in-place method:

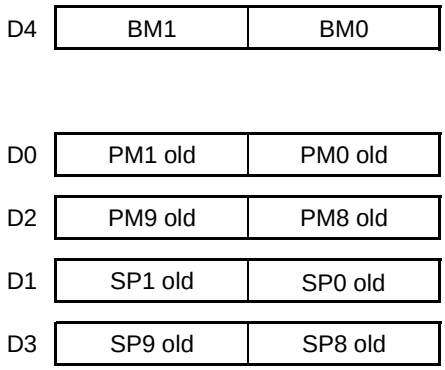

Figure 25. Initial Register Map for Two Trellis Butterflies - In-Place Method

In this figure . . .

- Register D4 contains the BMs.
- Registers D0 and D2 contain the PMs from the previous trellis stage for states 0, 1, 8, and 9.
- Registers D1 and D3 contain the survivor paths (SPs) from the previous trellis stage for states 0, 1, 8, and 9.

The optimized StarCore SC140 assembly code for computing two trellis butterflies using the in-place method is shown in Example 1. If you design the Viterbi decoding algorithm such that the BMs, SPs, and PMs are stored in the StarCore SC140 data registers shown in Figure 25, the code of this example computes two trellis butterflies in three cycles. On average, the number of cycles required to compute each trellis butterfly is 1.5 cycles.

Note: Because the loading of the data from memory can be performed in parallel with the processing of the previous trellis butterfly, the cycles required for loading the data don't contribute to the cycle count.

Example 1. Assembly Code for Computing Two Butterflies - In-Place Method

```
; Read from memory the pre-calculated BMs.
   move.l(r1)+,d4
;
; Read the old SPs and PMs from memory.
   [ tfr d4,d6  move.2l(r2)+,d0:d1
   move.2l (r3)+,d2:d3
   ]
; Add and subtract the BMs from the old PMs.
   [ add2 d0,d4  sub2 d6,d2
   sub2 d4,d0  add2 d2,d6
   ]
```

```
; Compare the PMs for each path and select the paths with the
; largest PMs as the SPs.
   max2vit d4,d2  max2vit d0,d6
;
; Move the new SPs and PMs to memory.
   [ vsl.4w d2:d6:d1:d3,(r4)+n0  vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
```

4.3.2 Dedicated Register Method

You can also design the Viterbi decoder kernel to process two butterflies in parallel using the dedicated register method. This method is similar to the in-place method, except that it stores the BMs in a dedicated register. Because the BMs stored in the dedicated register are not overwritten during the trellis butterfly computation, the algorithm can reuse them to compute other trellis butterflies that have the same BMs. Figure 26 shows the initial StarCore SC140 register map for computing two trellis butterflies using the dedicated register method:

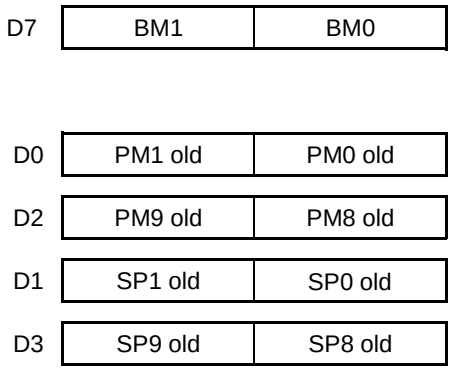

Figure 26. Initial Register Map for Two Trellis Butterflies - Dedicated Register Method

In this figure . . .

- Register D7 contains the BMs.
- Registers D0 and D2 contain the PMs from the previous trellis stage for states 0, 1, 8, and 9.
- Registers D1 and D3 contain the SPs from the previous trellis stage for states 0, 1, 8, and 9.

The StarCore SC140 assembly code for computing two trellis butterflies using the dedicated register method is shown in Example 2.

Example 2. Assembly Code for Computing Two Butterflies - Dedicated Register Method

```
; Read the old SPs and PMs from memory.
   [ move.2l (r2)+,d0:d1  move.2l (r3)+,d2:d3
   ]
; Read the pre-calculated BMs from memory.
   [ tfr d0,d4  tfr d2,d6
   move.l (r1)+,d7
   ]
```

```
; Add and subtract the BMs from the old PMs.
   [ add2 d7,d4  sub2 d7,d2
   sub2 d7,d0  add2 d7,d6
   ]
; Compare the PMs for each path and select the paths with the
; largest PMS as the SPs.
   max2vit d4,d2  max2vit d0,d6
;
; Move the new SPs and PMs to memory.
   [ vsl.4w d2:d6:d1:d3,(r4)+n0  vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
```

4.3.3 Advantages and Disadvantages

Each method for computing two trellis butterflies in parallel has its advantages and disadvantages. The in-place method reduces the number of register transfers because the registers that hold the BMs also hold the new PMs.

The dedicated register method uses a dedicated register for the BMs and is therefore more flexible for calculating the new PMs than the in-place method. For example, the dedicated register method computes two butterflies in parallel using BM1 and BM0 and two additional butterflies in parallel using -BM1 and -BM0. The kernel performs these computations using the same pre-determined BM1 and BM0 stored in register D7. To perform the computations, the kernel simply alternates use of the ADD2 and SUB2 instructions. However, the computational flexibility associated with this method comes at a cost: This method requires more register transfers.

Fortunately, you don’t have to choose one method or the other. You can design the Viterbi decoder kernel to capitalize on the advantages of both methods. The optimized assembly code for the GSM TCH/FS Viterbi Decoder provided in this application note incorporates both methods to achieve better performance (e.g., cycle counts and code size) than it would be capable of if it used either method alone (see “Viterbi Decoder for the StarCore SC140” on page 57).

Optimizations to the Branch Metric Calculation

This chapter describes how to optimize the branch metric (BM) calculation.

5.1 Branch Metric Calculation

The Viterbi decoder kernel requires branch metrics (BMs) to calculate the path metrics (PMs). The input data for calculating the BMs are the soft-decision encoded symbols that the decoder receives from the demodulator.

The decoder calculates the BMs as a function of . . .

- Constraint length, K
- Code rate, R
- Generator polynomials, G0, G1, . . Gn
- Soft-decision encoded symbols

5.2 Optimized Assembly Code

Example 3 shows the optimized assembly code for the branch metric calculation. This code is targeted for the StarCore SC140 and conforms to the Global System for Mobile Communications (GSM) Traffic Channel/Full-Rate Speech (TCH/FS) standard.

The assembly code is for decoding convolutional codes with a constraint length of K=5, a code rate of R=1/2, and the following generator polynomials:

- $G0 = 1 + D^3 + D^4$
- $G1 = 1 + D + D^3 + D^4$

The code of Example 3 stores the soft decision encoded symbols in registers D13 and D15. The code stores the BMs in the high and low portions of registers D5 and D7. The BMs in register D7 are used to calculate the first set of four butterflies. The BMs in register D5 are used to calculate the second set of four butterflies.

Example 3. Branch Metric calculation for GSR TCH/FS

```
; Get the soft decision inputs for the first stage.
    move.w (r0)+n1,d15  move.w (r1)+n1,d13
;
; Calculate the BM for the first stage of the trellis.
   [ add d15,d13,d7  sub d13,d15,d13
   clr d5
   ]
; d7 = BM1||BM0 is used for the first four trellis butterflies.
   [ insert #16,#16,d13,d7  neg d13
   sub d7,d5,d5
   ]
; d5 = BM3||BM2 is used for the last four trellis butterflies.
   insert #16,#16,d13,d5
```

5.3 Further Optimizations

Example 3 shows that the BMs for calculating eight trellis butterflies in GSM TCH/FS have a special feature: The low and high portions of register d5 contain the negatives of the BMs in the low and high portions of register d7, respectively. You can use this feature of the algorithm to further optimize the assembly code as follows:

- Determine the BMs in register d7 only.
- Calculate trellis butterflies according to the dedicated register method (see “Dedicated Register Method” on page 43).

The assembly code of the GSM TCH/FS Viterbi decoder incorporates the branch metric calculation into the main kernel of the trellis butterfly computation and makes use of this optimization technique to achieve improved performance (see “Viterbi Decoder for the StarCore SC140” on page 57).

Optimizations to the Viterbi Decoder Kernel

This chapter provides optimized assembly code for the Viterbi decoder kernel. The kernel updates the path metrics and saves the survivor paths at each trellis stage. Basically, the kernel is the heart of Viterbi decoding, determining the path through the trellis that most likely represents the encoder input.

The Viterbi decoder kernel is the most computationally intensive part of the Viterbi decoding algorithm, performing 2^{K-1} add-compare-select (ACS) computations for each decoded bit. To reduce the cycle counts consumed by the ACS computation, the kernel assembly code capitalizes on the variable-length-execution-set (VLES) capabilities of the StarCore SC140.

Because of the VLES capabilities of the StarCore SC140, the Viterbi decoder kernel presented in this application note performs two ACS calculations (for two different trellis butterflies) in parallel. In addition, the kernel incorporates pipelining techniques for computing the $2^{K-1}/2$ trellis butterflies at each trellis stage. These techniques maximize the parallelism of the assembly code, reducing the overall cycle count.

The StarCore SC140 instruction set contains special instructions that reduce the cycle counts consumed by the Viterbi decoder kernel (see “Special Instructions for Viterbi Decoding” on page 27).

This chapter includes the following:

- Memory map illustrating where the kernel stores its results
- Optimized assembly code for the kernel
- Description of the pointers in the assembly code
- Performance benchmarks for the kernel running on a StarCore SC140
- How to modify the kernel for convolutional codes with constraint lengths greater than 5

6.1 Memory Map of the Kernel

The memory map for the Viterbi decoder kernel illustrates where the kernel stores the updated path metrics and survivor paths at each trellis stage (see Figure 27).

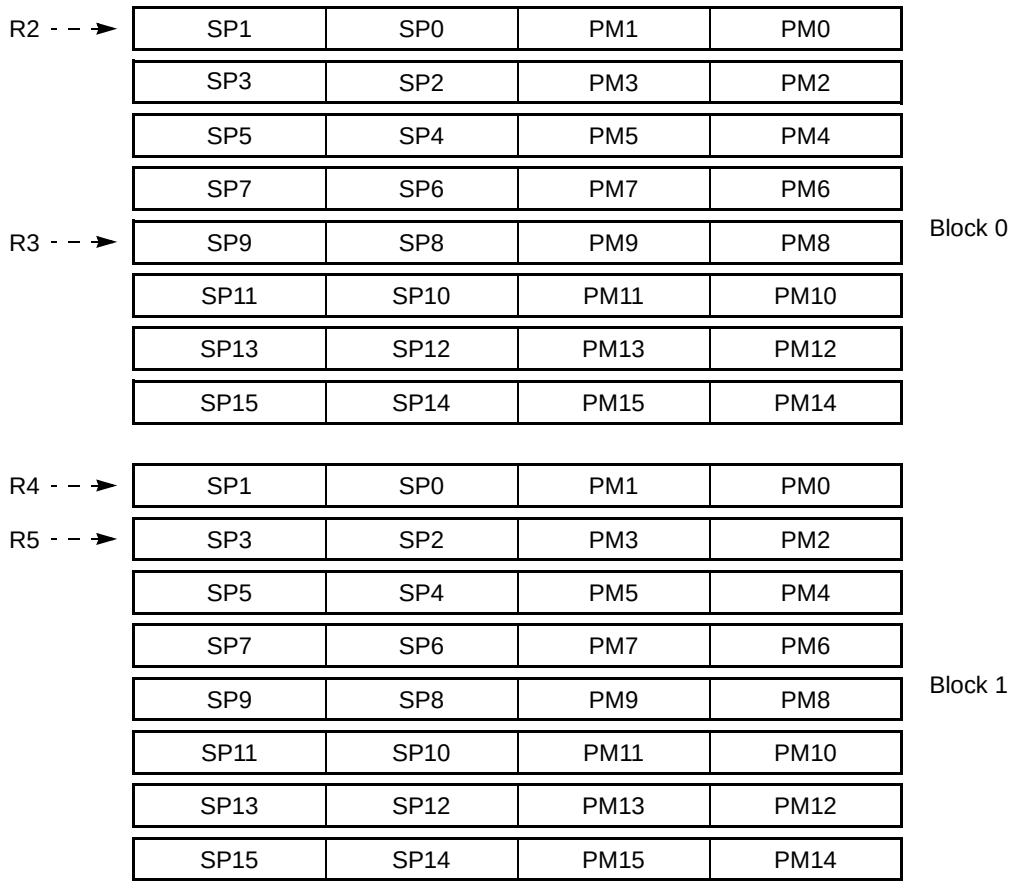

Figure 27. Memory Map for the Viterbi Decoder Kernel

This memory map applies to convolutional codes with a constraint length of K=5 and a code rate of R=1/*n*. A similar memory map can be derived for convolutional codes with other constraint lengths.

The data buffer illustrated by the memory map is a modulo buffer. In a modulo buffer, one block—either Block 0 or Block 1—saves the path metrics and survivor paths from the previous trellis stage. The other block saves the path metrics and survivor paths from the current trellis stage. The roles of these blocks alternate every trellis stage.

The data in each block are the path metrics and the survivor paths for each encoder state. Each cell in a block represents 2 bytes of data. Each block contains 2^{K-1}*2*2=64 bytes.

6.2 Kernel Assembly Code

This section provides the assembly code for the Viterbi decoder kernel. This code applies to convolutional codes with a constraint length of K=5 and a code rate of R=1/n. The code comprises an initialization step followed by 10 execution sets.

```
; ********************************************************************
;                          Kernel Initialization
; ********************************************************************

; Read the first 4 branch metrics (BMs) from the registers.
   move.2l (r1)+,d4:d5

; ********************************************************************
;                          Viterbi Decoder Kernel
; ********************************************************************

; Execution Set 1
; Read the path metrics (PM) and survivor path (SP) values for the first two
; trellis butterflies.
   [ tfr d4,d6  move.2l (r2)+,d0:d1
   move.2l (r3)+,d2:d3
   ]
; Execution Set 2
; Perform the add operation for the first two trellis butterflies.
; Read next 4 BMs.
   [ add2 d0,d4  sub2 d6,d2
   sub2 d4,d0  add2 d2,d6
   move.2l (r1)+,d12:d13
   ]
; Execution Set 3
; Perform the compare and select operation for the first two trellis
; butterflies. Read the second pair of PMs and SPs.
   [ max2vit d4,d2  max2vit d0,d6
   tfr d12,d14  move.2l (r2)+,d8:d9
   move.2l (r3)+,d10:d11
   ]
; Execution Set 4
; Process the second pair of trellis butterflies. Move selected PMs and SPs
; to memory.
   [ add2 d8,d12  sub2 d14,d10
   sub2 d12,d8  add2 d10,d14
   vsl.4w d2:d6:d1:d3,(r4)+n0  vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
```

```
; Execution Set 5
; Perform the compare and select operation for the second pair of
; trellis butterflies. Read the third pair of PMs and SPs.
   [ max2vit d12,d10  max2vit d8,d14
   tfr d5,d6  tfr d5,d4
   move.2l (r2)+,d0:d1  move.2l (r3)+,d2:d3
   ]
; Execution Set 6
; Process the third pair of trellis butterflies. Move selected PMs and SPs
; to memory.
   [ add2 d0,d4  sub2 d6,d2
   sub2 d4,d0  add2 d2,d6
   vsl.4w d10:d14:d9:d11,(r4)+n0  vsl.4f d10:d14:d9:d11,(r5)+n0
   ]
; Execution Set 7
; Perform the compare and select operation for the third pair of trellis
; butterflies. Read the fourth pair of PMs and SPs.
   [ max2vit d4,d2  max2vit d0,d6
   tfr d13,d12  tfr d13,d14
   move.2l (r2)+,d8:d9  move.2l(r3)+,d10:d11
   ]
; Execution Set 8
; Process the fourth pair of trellis butterflies. Move selected PMs and SPs
; to memory.
   [ add2 d8,d12  sub2 d14,d10
   sub2 d12,d8  add2 d10,d14
   vsl.4w d2:d6:d1:d3,(r4)+n0  vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
; Execution Set 9
; Perform the compare and select operation for the fourth pair of trellis
; butterflies. Read the next 4 BMs for the next stage.
   [ max2vit d12,d10  max2vit d8,d14
   move.2l (r1)+,d4:d5
   ]
; Execution Set 10
; Move selected PMs and SPs to memory.
   [ vsl.4w d10:d14:d9:d11,(r4)+n0  vsl.4f d10:d14:d9:d11,(r5)+n0
   ]
```

6.3 Pointers Used in the Kernel

The following table describes the pointers in the assembly code:

The address in this register . . .	Points to . . .
R1	Branch metric (BM) table
R2	State j of the previous trellis stage
R3	State (j+l/2) of the previous trellis stage
R4	State 2j of the current trellis stage
R5	State (2j+1) of the current trellis stage

6.4 Kernel Cycle Count

Running on the StarCore SC140, the Viterbi decoder kernel processes eight trellis butterflies in 10 cycles (for convolutional codes with a constraint length of K=5). On average, the processing of each trellis butterfly consumes 1.25 cycles.

6.5 How to Modify the Kernel for K > 5

You can modify the Viterbi decoding algorithm for a convolutional code with a constraint length of K greater than 5. All you need to do is program the kernel to run in a loop of 2^{K-5} times per trellis stage.

After you program the kernel to run in a loop, you can further optimize the kernel assembly code for GSM channels by concatenating execution set 10 to execution set 2 such that the last execution set of a trellis stage is performed on the next trellis stage.

To do this, make the following changes to the Viterbi decoder kernel:

1. Delete `move.2l(r2)+,d0:d1` from execution set 1 and append it to the kernel initialization.
2. Append `move.2l(r2)+,d0:d1` to execution set 9.
3. Delete `move.2l(r1)+,d12:d13` from execution set 2 and append it to execution set 1.
4. Append execution set 10 to execution set 2.

Endian Modes and Viterbi Decoding

The StarCore SC140 supports little and big endian computer architectures. Little endian is a computer architecture in which the least significant byte of a multi-byte numeric word has the lowest address. That is, each word is stored little byte first. In a big endian architecture, the most significant byte of a multi-byte numeric word has the lowest address. That is, each word is stored big byte first.

This section describes the memory maps of the kernel and the sub-blocks save modules of the Viterbi decoding algorithm for little and big endian architectures.

For detailed information about the endian modes support of the StarCore SC140, refer to the *StarCore SC140 Core Reference Manual*.

7.1 Little Endian Mode

For little endian mode, the memory maps of the Viterbi decoder kernel and the sub-block save is illustrated in Figure 28.

Viterbi decoder kernel

(Rn + 6)	(Rn + 4)	(Rn + 2)	(Rn + 0)
SP1	SP0	PM1	PM0
(Rn + 14)	(Rn + 12)	(Rn + 10)	(Rn + 8)
SP3	SP2	PM3	PM2
(Rn + 22)	(Rn + 20)	(Rn + 18)	(Rn + 16)
SP5	SP4	PM5	PM4
(Rn + 30)	(Rn + 28)	(Rn + 26)	(Rn + 24)
SP7	SP6	PM7	PM6
(Rn + 38)	(Rn + 36)	(Rn + 34)	(Rn + 32)
SP9	SP8	PM9	PM8
(Rn + 46)	(Rn + 44)	(Rn + 42)	(Rn + 40)
SP11	SP10	PM11	PM10
(Rn + 54)	(Rn + 52)	(Rn + 50)	(Rn + 48)
SP13	SP12	PM13	PM12
(Rn + 62)	(Rn + 60)	(Rn + 58)	(Rn + 56)
SP15	SP14	PM15	PM14

sub-block save

(Rn + 2)	(Rn + 0)
SP1	SP0
(Rn + 6)	(Rn + 4)
SP3	SP2
(Rn + 10)	(Rn + 8)
SP5	SP4
(Rn + 14)	(Rn + 12)
SP7	SP6
(Rn + 18)	(Rn + 16)
SP9	SP8
(Rn + 22)	(Rn + 20)
SP11	SP10
(Rn + 26)	(Rn + 24)
SP13	SP12
(Rn + 30)	(Rn + 28)
SP15	SP14

Figure 28. Memory Map for Little Endian Mode

7.2 Big Endian Mode

For big endian mode, the memory map of the Viterbi decoder kernel and the sub-block save is illustrated in Figure 29.

Viterbi decoder kernel

(Rn + 0)	(Rn + 2)	(Rn + 4)	(Rn + 6)
PM1	PM0	SP1	SP0
(Rn + 8)	(Rn + 10)	(Rn + 12)	(Rn + 14)
PM3	PM2	SP3	SP2
(Rn + 16)	(Rn + 18)	(Rn + 20)	(Rn + 22)
PM5	PM4	SP5	SP4
(Rn + 24)	(Rn + 26)	(Rn + 28)	(Rn + 30)
PM7	PM6	SP7	SP6
(Rn + 32)	(Rn + 34)	(Rn + 36)	(Rn + 38)
PM9	PM8	SP9	SP8
(Rn + 40)	(Rn + 42)	(Rn + 44)	(Rn + 46)
PM11	PM10	SP11	SP10
(Rn + 48)	(Rn + 50)	(Rn + 52)	(Rn + 54)
PM13	PM12	SP13	SP12
(Rn + 56)	(Rn + 58)	(Rn + 60)	(Rn + 62)
PM15	PM14	SP15	SP14

sub-block save

(Rn + 0)	(Rn + 2)
SP1	SP0
(Rn + 4)	(Rn + 6)
SP3	SP2
(Rn + 8)	(Rn + 10)
SP5	SP4
(Rn + 12)	(Rn + 14)
SP7	SP6
(Rn + 16)	(Rn + 18)
SP9	SP8
(Rn + 20)	(Rn + 22)
SP11	SP10
(Rn + 24)	(Rn + 26)
SP13	SP12
(Rn + 28)	(Rn + 30)
SP15	SP14

Figure 29. Memory Map for Big Endian Mode

7.3 Differences between Endian Modes

Figure 28 and Figure 29 show that in big endian mode, pairs of survivor paths and path metrics are switched in 16-bit word address locations. As a result, the big endian version of the Viterbi code requires one more operation for determining the correct offset to the survivor path during traceback than the little endian version of the code. After the traceback module calculates the offset from the sub-block base address, it inverts the least significant bit (LSB) of the offset in big endian mode. Although this offset correction adds one cycle to the processing of every 12 decoded bits, it does not significantly affect the total cycle count.

The traceback module for the GSM TCH/FS Viterbi decoder contains assembler directives that instruct the assembler which portions of the code to convert to opcode depending on whether the target architecture is little or big endian:

- IF
- ELSE
- ENDIF

The assembler uses these directives to include or exclude certain portions of the assembly code for conversion into opcode as requried by the target architecture.

You must specify the target architecture to the assembler. To do this, you have two choices:

- Specify the target architecture on a command line when you invoke the assembler.
- Specify the target architecture in the assembly source file.

If you specify the target architecture in the assembly source file, the instruction that you use must precede the assembly directives.

Viterbi Decoder for the StarCore SC140

This chapter provides optimized assembly code for implementing a complete Viterbi decoder on the StarCore SC140. This code complies with the Global System for Mobile Communications (GSM) Traffic Channel/Full-Rate Speech (TCH/FS) standard. This standard is the most widely used digital cellular communications standard in the world.

This chapter also provides performance statistics for this implementation of a Viterbi decoder running on the StarCore SC140.

8.1 Viterbi Decoder Inputs

As defined in GSM Standard 05.03, the Viterbi decoder for GSM TCH/FS receives 378 soft decision symbols in each input block.

Note: The size of the buffer is set to twice the size of the input frame (i.e., $2 \times 189 = 378$).

8.2 Code Parameters

The convolutional code for this implementation of a Viterbi decoder has a constraint length of K=5 and a code rate of R=1/2.

8.3 Generator Polynomials

The generator polynomials for the convolutional code are . . .

- $G0 = 1 + D^3 + D^4$
- $G1 = 1 + D + D^3 + D^4$

8.4 Assembly Code

This section provides the assembly code for implementing the Viterbi decoder for GSM TCH/FS on the StarCore SC140. The code comprises four main parts:

- Initialization
- Viterbi decoder kernel (includes branch metric calculation)
- Sub-blocks save
- Traceback

```
;*********************************************************************
;                   Viterbi Decoder for GSM TCH/FS
;*********************************************************************
; Copyright Motorola Inc., 2000. All rights reserved.
; Copyright Lucent Technologies Inc., 2000. All rights reserved.
; Reverse engineering is prohibited.
;
; Purpose: GSM TCH/FS Viterbi decoding - including traceback
; Code rate R = 1/2
; Constraint length K = 5
; Generator Polynomials: G0=1+D^3+D^4
;                        G1=1+D+D^3+D^4
;
; Inputs: EQU_OUTPUTS - 378 soft decisions buffer.
;
; Outputs: PACKED_TRACEBACK_OUTPUTS - Viterbi decoding outputs buffer
; Packed in 16-bit contiguous words
;
; Registers used: d0-d15, r0-r9, b2-b5, n0-n3, m0, mctl
;
;*********************************************************************
NO_OF_STATES define "16"  ; Number of trellis encoder states
NO_OF_STAGES define "189"  ; Number of Viterbi trellis stages
PM_INIT define "0"  ; Path metric initialization for all states

; Path metric initialization of state 0. The purpose of the following
; initialization is to increase the weight of state 0.
ZERO_INIT define "1000"

;*********************************************************************
;           Implementation of GSM TCH/FS Viterbi Decoder
;*********************************************************************
; The remainder = 9 for 189 decoded bits.
RESIDUE equ NO_OF_STAGES-(((NO_OF_STAGES-1)/12)*12)
```

```
;*********************************************************************
;                          Initialization
;*********************************************************************
   org p:$4000
   align 4
EQU_OUTPUTS
   INCLUDE 'viterbi_input.in'  ; ds NO_OF_STAGES*2*2
   align 8
PACKED_TRACEBACK_OUTPUTS  ; Output bits are packed in 16-bit words.
   ds ((NO_OF_STAGES-1)/16+1)*2 ; = 24 for 189 decoded bits
   align 8
PM_RAM_SP_RAM
; The argument of DSM contains 128 bytes for GSM TCH/FS. There are 16 states
; and 2 words per state for SP||PM. Each word contains 2 bytes and
; is double buffered.
   dsm 2*(NO_OF_STATES*2)*2
; The PM_RAM_SP_RAM and SAVED_SP buffers must be in different groups
; to prevent stalls.
   org p:$BD00
   align 8
SAVED_SP
   ds (((NO_OF_STAGES-1)/12+1)*NO_OF_STATES)*2  ; = 512 for 189 decoded bits.

   org p:
   jmp $1000

   org p:$1000

   move.l #EQU_OUTPUTS,r0  ; r0,r1 are pointers to soft decision inputs.
   move.l #PM_RAM_SP_RAM,d4  ; r2,r3 are old state pointers.
   move.l #SAVED_SP,r6  move.l d4,r2  ; r6,r7 are sub-blocks pointers.
   adda #2,r0,r1  move.l d4,b2
   move.l #PM_RAM_SP_RAM+64+48,r4  ; r4,r5 are new state pointers.
   doensh0 #NO_OF_STATES/2-1  adda #8,r2,r3  ; path metric init.
   adda #8,r4,r5  move.w #11,n3
   move.w #2,n1  move.w #-2,n0  ; n0 advances in the reverse order.
   [ clr d1  clr d3
   move.w #ZERO_INIT,d2  move.w #PM_INIT,d0
   ]
   insert #16,#16,d0,d0  insert #16,#16,d0,d2
   move.2l d2:d3,(r2)+n1  move.2l d0:d1,(r3)+n1
   loopstart0
   move.2l d0:d1,(r2)+n1  move.2l d0:d1,(r3)+n1
   loopend0
   move.l d4,b3  move.l #$00888800,MCTL
```

```
    [
; Last butterflies are processed first.
    [ move.l #PM_RAM_SP_RAM+24,r2
; Set base address for r2,r3,r4,r5. All registers use M0 modulo addressing.
    move.l d4,b4
    ]
    move.l d4,b5  move.w #128,m0

;*********************************************************************
;                       Butterfly Initialization
;*********************************************************************
; d4 is loop count for the Viterbi kernel.
    move.w #<12,d4  dosetup0 _out_loop

    adda #32,r2,r3  adda #8,r6,r7  ; r7 is the sub-blocks pointer.
    [ clr d5
; Get soft decision inputs for the first trellis stage.
    move.w (r0)+n1,d7  move.w (r1)+n1,d6
    ]
    [ add d7,d6,d7  sub d6,d7,d0  ; Calculate BM for the first trellis stage.
    move #<4,n2  dosetup1 _viterbi_kernel
    ]
    [ insert #16,#16,d0,d7  add #<(NO_OF_STAGES-1)/12,d5  ; d7 = BM1 || BM0
    move.2l (r2)-,d8:d9  move.2l (r3)-,d10:d11  ; d5 = 15 for 189 decoded bits
    ]
    [ tfr d8,d12  tfr d10,d14
    doen0 #(NO_OF_STAGES-1)/12+1
    ]
    loopstart0
_out_loop:
;*********************************************************************
;                     Main Loop (NO_OF_STAGES-1)/12+1
;*********************************************************************
    [ sub d0,d0,d0  sub d1,d1,d1  ; dummy NOPs to align start of loop
    sub d2,d2,d2  sub d3,d3,d3
    doen1 d4
    ]
    falign
    loopstart1
```

```
_viterbi_kernel:
;*************************************************************************
;                          Viterbi Decoder Kernel
;*************************************************************************
   [ sub2 d7,d12  add2 d7,d10
   add2 d7,d8  sub2 d7,d14
   move.2l (r2)-,d0:d1  move.2l (r3)-,d2:d3
   ]
   [ max2vit d12,d10  max2vit d8,d14
   tfr d0,d4  tfr d2,d6
   move.w (r0)+n1,d15  move.w (r1)+n1,d13
   ]
   [ sub2 d7,d4  add2 d7,d2
   add2 d7,d0  sub2 d7,d6
   vsl.4w d10:d14:d9:d11,(r4)+n0
   vsl.4f d10:d14:d9:d11,(r5)+n0
   ]
   [ max2vit d4,d2  max2vit d0,d6
   tfr d7,d12  tfr d7,d14
   move.2l (r2)-,d8:d9  move.2l (r3)-,d10:d11
   ]
   [ add2 d8,d12  sub2 d14,d10
   sub2 d12,d8  add2 d10,d14
   vsl.4w d2:d6:d1:d3,(r4)+n0
   vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
   [ max2vit d12,d10  max2vit d8,d14
   tfr d7,d4  tfr d7,d6
   move.2l (r2)+n3,d0:d1  move.2l (r3)+n3,d2:d3
   ]
   [ add2 d0,d4  sub2 d6,d2
   sub2 d4,d0  add2 d2,d6
   vsl.4w d10:d14:d9:d11,(r4)+n0
   vsl.4f d10:d14:d9:d11,(r5)+n0
   ]
   [ max2vit d4,d2  max2vit d0,d6
   add d15,d13,d7  sub d13,d15,d13
   move.2l (r2)-,d8:d9  move.2l (r3)-,d10:d11
   ]
   [ tfr d8,d12  tfr d10,d14
   insert #16,#16,d13,d7
   vsl.4w d2:d6:d1:d3,(r4)+n0
   vsl.4f d2:d6:d1:d3,(r5)+n0
   ]
   loopend1
```

```
;*********************************************************************
;                              Sub-Blocks Save
;*********************************************************************
   [ clr d4  clr d6  ; r8 points to SP1||SP0
; r9 points to SP3||SP2 in the PM_RAM_SP_RAM table.
   adda #-12,r2,r8  adda #-4,r2,r9
   ]
   [ add #<12,d4  deceq d5  ; T-bit set indicates the last sub-block.
   move.l (r8)+n2,d0  move.l (r9)+n2,d1
   ]
   [ add #<RESIDUE,d6
; Last sub-block contains the remainder = 9 decoded bits.
   move.l (r8)+n2,d2  move.l (r9)+n2,d3
   ]
   [ tfrt d6,d4  ; d4 = 9 when processing the last sub-block.
   move.2l d0:d1,(r6)+n1  move.2l d2:d3,(r7)+n1
   ]
   move.l (r8)+n2,d0  move.l (r9)+n2,d1
   move.l (r8)+n2,d2  move.l (r9)+n2,d3
   move.2l d0:d1,(r6)+n1  move.2l d2:d3,(r7)+n1
   loopend0

   move.l #$00000000,MCTL

;*********************************************************************
;                               Traceback
;*********************************************************************
   IF BIG_ENDIAN
; r0 points to the last sub-block.
   move.l #SAVED_SP+NO_OF_STATES*((NO_OF_STAGES-1)/12)*2+2,r0
   ELSE
   move.l #SAVED_SP+NO_OF_STATES*((NO_OF_STAGES-1)/12)*2,r0
   ENDIF
   [ dosetup0 _traceback_loop
; Every four sub-blocks, traceback generates three 16-bit words of
; packed decoded bits.
   doen0 #((NO_OF_STAGES-1)/12+1)/4
   ]
   [ clr d2
   IF BIG_ENDIAN
   move.w (r0)-,d1  ; Start traceback from state zero.
   ELSE
   move.w (r0),d1
   ENDIF
```

```
    move.l #PACKED_TRACEBACK_OUTPUTS+((NO_OF_STAGES-1)/16)*2,r1
    ]
    [ extractu #4,#RESIDUE,d1,d2  ; Last sub-block contains 9 decoded bits.
    asll #<(12-RESIDUE),d1  ; align RESIDUE to left
    move.w #-2*NO_OF_STATES,n1  ; offset to next sub-block
    move.w #1,d5  ; d5 is used for big endian only.
    ]
    loopstart0
_traceback_loop
    IF BIG_ENDIAN
    eor d5,d2  ; offset correction
    ENDIF
    move.l d2,n0  ; survivor path index in the next sub-block
    adda n1,r0  move.w d1,(r1)-  ; Write 16 decoded bits to the output buffer.
    move.w (r0+n0),d1
    extractu #4,#12,d1,d2  asrr #<4,d1
    IF BIG_ENDIAN
    eor d5,d2  ; offset correction
    ENDIF
    move.l d2,n0
    adda n1,r0
    move.w (r0+n0),d3
    extractu #4,#12,d3,d4
    IF BIG_ENDIAN
    eor d5,d4  ; offset correction
    ENDIF
    [ insert #8,#8,d3,d  asrr #<8,d3
    move.l d4,n0
    ]
    adda n1,r0  move.w d1,(r1)-
    move.w (r0+n0),d1
    extractu #4,#12,d1,d2
    IF BIG_ENDIAN
    eor d5,d2  ; offset correction
    ENDIF
    insert #12,#4,d1,d3  move.l d2,n0
    adda n1,r0  move.w d3,(r1)-
    move.w (r0+n0),d1
    extractu #4,#12,d1,d2
    loopend0

END
```

8.5 How to Assemble the Viterbi Algorithm

This section describes how to assemble the Viterbi algorithm provided in this application note:

- How to handle input data
- Which assembly commands to use for big and little endian architectures

To assemble the Viterbi algorithm, you must have the SC100 Assembler installed on your computer. You may also want to refer to the following materials:

- *SC100 Assembly Language Tools User's Manual*
- *Getting Started with the SC100 Tools*

These materials are publications of Lucent Technologies and Freescale You can access them by clicking the Documentation link at www.starcore-dsp.com.

Note: The procedure provided in this section applies when you are using the SC100 Assembler in the command-line mode. However, even if you are using the SC100 Assembler within an integrated development environment (IDE), you may still find the information contained in this procedure helpful.

8.5.1 How to Handle the Input Data

When you assemble the Viterbi algorithm, the assembler looks for a file entitled, "viterbi_input.in." This file consists of 378 soft decision inputs structured as follows:

```
dc $ffaa
dc $ffc1
. . .
. . .
```

For the Viterbi algorithm to assemble correctly, you must place the input file in the same directory folder as the algorithm.

8.5.2 Which Assembly Commands to Use

After you have placed the Viterbi algorithm and the input file in the same directory folder, you are ready to assemble the algorithm. To assemble the Viterbi algorithm using the SC100 Assembler, use one of the following commands:

For . . .	Use this command . . .
Little endian architecture	asmsc100 -a -b -l -g -s all -d BIG_ENDIAN 0 viterbi.asm
Big endian architecture	asmsc100 -a -b -l -g -s all -d BIG_ENDIAN 1 -obe viterbi.asm

When invoked with one of these commands, the SC100 Assembler will produce an executable file and store it in the same directory folder as the algorithm. You can then run the executable on the SC100 Simulator to test the application (see "How to Test the Viterbi Algorithm" on page 65).

8.6 How to Test the Viterbi Algorithm

You typically test an algorithm by running it on its target processor or simulator and comparing the output to a reference output that is known to be correct for the input data. This section describes how to test the Viterbi algorithm using the SC100 Simulator:

- Which simulator commands to use for big and little endian architectures
- How to check the decoder output

To test the Viterbi algorithm, you must have the SC100 Simulator installed on your computer. Also, you must have already assembled the algorithm using the SC100 Assembler (see “How to Assemble the Viterbi Algorithm” on page 64).

Note: The procedure provided in this section applies when you are using the SC100 Simulator in the command-line mode. However, even if you are using the SC100 Simulator within an integrated development environment (IDE), you may still find the information contained in this procedure helpful.

8.6.1 Which Simulator Commands to Use

To test the Viterbi application using the SC100 Simulator, perform the following:

1. Make sure that the executable file produced by the SC100 Assembler and the input file are in the same directory folder.
2. Enter the following command:
   simsc100
3. Run one of the following sequences of commands:

For . . .	Run this sequence of commands . . .
Little endian architecture	reset d m0 load viterbi.cld break END go display p:PACKED_TRACEBACK_OUTPUTS..PACKED_TRACEBACK_OUTPUTS+$16
Big endian architecture	reset d m1 load viterbi.cld break END go display p:PACKED_TRACEBACK_OUTPUTS..PACKED_TRACEBACK_OUTPUTS+$16

8.6.2 How to Check the Decoder Output

When run on the SC100 Simulator, the Viterbi decoder generates 189 bits of data structured as follows:

10010001011001011 . . . .

Check to be sure that the actual output of the simulator agrees with the reference output corresponding to the input data.

8.7 Performance

This section provides performance benchmarks for the GSM TCH/FS Viterbi decoder running on the StarCore SC140:

- Cycle count
- Data memory consumption

8.7.1 Cycle Count

The cycles consumed by the Viterbi decoder is governed by the following formula:

$$C = [(\lfloor (B-1)/12 \rfloor + 1) \times 8 + B \times 9 + 28] + [(\lfloor (B-1)/12 \rfloor + 1) \times (5 + E) + 3]$$

Where . . .

- C is the number of cycles consumed
- B is the block size, or the number of decoded bits in each block
- E is the endian mode (E=1 for big endian; E=0 for little endian)

This formula applies to GSM channels, convolutional codes with a constraint length of K=5 and a code rate of R=1/2, and a Viterbi decoder algorithm that saves a sub-block every 12 trellis stages.

The first term of the formula represents the cycles consumed by the following modules: branch metric (BM) calculation, add-compare-select (ACS) function, and sub-blocks save. The second term represents the cycles consumed by the traceback module.

The number of convolutional encoded bits in the GSM TCH/FS block is B=189. Therefore, in little endian mode, the cycle count for Viterbi decoding of a GSM TCH/FS block is 1940 cycles. In big endian mode, the traceback routine consumes 16 additional cycles, resulting in a total cycle count of 1956 cycles.

8.7.2 Data Memory Consumption

The data memory consumed by the Viterbi decoder is governed by the following formula:

$$M = [2^{(K+2)}] + [(\lfloor (B-1)/12 \rfloor + 1) \times 2^{K}]$$

Where . . .

- M is number of bytes of data memory consumed
- K is the constraint length of the convolutional code
- B is the block size, or the number of decoded bits in each block

The first term of the formula represents the bytes of memory consumed by the add-compare-select (ACS) function. The second term represents the bytes of memory consumed by the sub-blocks save module.

For the GSM TCH/FS Viterbi decoder, B=189 and K=5. Therefore, the data memory consumed is 640 bytes.

8.7.3 Program Memory Consumption

The Viterbi decoder consumes 476 bytes of program memory.

References

This chapter provides sources where you can obtain detailed information about Viterbi decoding and the StarCore SC140.

9.1 Viterbi Decoding

For additional information about the Viterbi algorithm, refer to the following:

- Clark, G. & Cain, J. *Error Correction Coding for Digital Communications*, Plenum Press, 1982.
- Forney, G. *The Viterbi Algorithm*, Proceedings of the IEEE 61 pp. 268-278, March 1973.
- GSM 05 Series, *Physical layer on Radio Path*, ETSI, V7, Release 1998.
- Lin, S. & Costello, D. J. *Error Control Coding*, Prentice-Hall, 1983.
- Rorabaugh, C. Britton *Error Control Cookbook*, McGraw-Hill, 1996.
- Wicker, Stephen B. *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, 1995.

9.2 StarCore SC140

For additional information about the StarCore SC140 and its associated tools, refer to the following:

- *StarCore SC140 Core Reference Manual*
- *SC100 Assembly Language Tools User's Manual*
- *Getting Started with the SC100 Tools*

These materials are publications of Lucent Technologies and Freescale. You can access them by clicking the Documentation link at www.starcore-dsp.com.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor