Implementing Viterbi Decoders Using the VSL Instruction on DSP Families DSP56300 and DSP56600

by

Dana Taipale

This application report describes how to generate, from a set of convolutional code polynomials, the assembly code needed for implementation of a Viterbi decoder.

Order this document by number APR40/D
(Revision 0, May 1998)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EXAMPLES

SECTION 1

INTRODUCTION

Introduction

1.1 INTRODUCTION

Today's communication systems typically make considerable use of signal processing to improve their performance. Two common functions that use signal processing to improve performance are channel equalization and error correction coding. For equalization, maximum likelihood sequence estimation is among the most popular schemes; for error correction, convolutional coding with Viterbi decoding is a method of choice. Both communication functions make use of the Viterbi algorithm to accomplish their task. To simplify the explanations presented here, the focus is on using the algorithm for error correction.

This application report is written with the intent of instructing the reader who, with a set of convolutional code polynomials, can generate the assembly code needed to implement a Viterbi decoder on the DSP56300 and DSP56600 families of digital signal processors (DSPs).

The DSP56300 and DSP56600 families of DSPs are designed to implement communication functions. Because the Viterbi algorithm occupies such a prominent position in many communications systems, these processor families have a special instruction, Viterbi shift left (VSL). This instruction is designed to improve the performance of the processor when doing Viterbi algorithm operations. One goal of this application note is to explain the use of the VSL instruction when implementing the Viterbi algorithm.

1.2 VITERBI ALGORITHM

Communication applications that use convolutional coding or sequence estimation often use the Viterbi algorithm to efficiently process the received data. These applications occur so often that the DSP56300 and DSP56600 families have a special instruction to enhance the algorithm's implementation. This application note begins by introducing the Viterbi algorithm, then proceeds step by step through an implementation of that algorithm for a specific error correction example. In addition, use of the new VSL instruction is explained for obtaining efficient coding for applications requiring the Viterbi algorithm.

1.3 MANUAL ORGANIZATION

Section 2 of this manual covers many basics of the Viterbi algorithm and introduces much of the terminology used in this application note. Those already acquainted with the algorithm may wish to proceed directly to **Section 3**.

Section 3 shows how to develop assembly language routines for the Viterbi algorithm. This is done by means of an example, taking a set of encoding polynomials from the wireless interim standard IS-136 and developing the assembly routines needed to implement the corresponding decoder using the Viterbi algorithm.

Section 4 discusses some code modifications that are useful for generalizing and optimizing the example code. The code-to-cover generalized branch metrics is extended, and an efficient method for normalizing the path metrics is introduced. In addition, two additional sets of code are given for more efficient processing of data periodically forced through a known state.

Section 5 summarizes the application report and presents interesting data and statistics arrived at by comparing the basic Viterbi algorithm code of **Section 2** with that of the modified program code presented in **Section 3** and **Section 4**.

Appendix A contains the complete program listing for **Section 3**.

Appendix B contains the complete program listing for **Section 4**.

Appendix C contains the complete 24-bit program listing for **Section 4**.

SECTION 2

THE VITERBI ALGORITHM

2.1 IS-136

This section introduces the Viterbi algorithm through an example from industry: the convolutional error correcting code used in IS-136, the TIA/EIA interim wireless standard. Here it is shown how the algorithm is derived, and some typical computational enhancements are introduced.

2.2 CONVOLUTIONAL ENCODING

The Viterbi algorithm can be usefully illustrated by implementing a convolutional code. A convolutional encoder implementation in hardware is illustrated schematically in **Figure 2-1**. In order to illustrate the steps needed to implement Viterbi decoders in general, this application report develops the assembly code needed to implement a Viterbi algorithm decoder for this encoder.

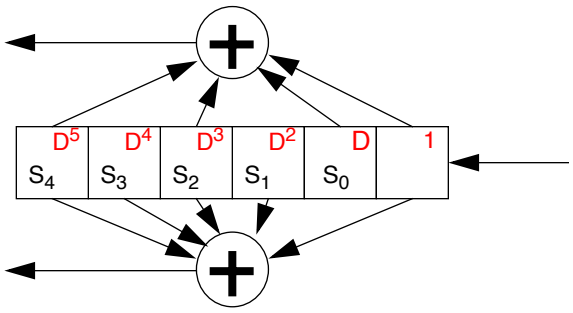

Figure 2-1 Example Rate 1/2 Convolutional Encoder

In this figure, the square boxes represent a one-bit-wide shift register. Some shift register outputs are connected to modulo 2 adders (i.e., parity generators). Every time one bit is shifted into the input, two output bits are generated by the adders. Because we have more output bits than input bits, we have redundancy. The redundant information can be used to correct errors at the receiver.

Error correcting codes have some standard notation. This example, used in the wireless standard IS-136, is a rate 1/2 code, meaning that for each processing period, one information bit is taken in and two bits of output are generated. The word (S_4,S_3,S_2,S_1,S_0) is the encoder state. The remaining bit of the encoder we call the input bit. There is a convenient notation that describes the encoder using polynomials. This encoder can be described with the encoding polynomials ($1+D+D^3+D^5$,$1+D^2+D^3+D^4+D^5$). Each factor D corresponds to a one clock delay for that adder input. Given the encoding polynomials, we can begin to design the Viterbi decoder.

2.3 VITERBI DECODER

To design a Viterbi decoder, begin by taking some example encoder input and generating the corresponding output. An example appears in **Table 2-1**, where it is assumed that the encoder is 0 filled at the start. For proper decoding, we need to recreate the encoder states and find the set of state changes, or transitions, that produce an encoder output that best agrees with our decoder input data. We begin using the same assumption we used in generating the encoder output in **Table 2-1**. We assume that the initial state of the encoder is 00000.

Table 2-1 Input/Output Mapping

Encoder Input	Encoder Output
1	11
0	10
1	10
1	10
0	10

To compare our recreated encoder state with the decoder input, we keep track of the agreements between the recreated encoder outputs and the decoder inputs. The cumulative agreement for a path leading to one particular recreated encoder state is called a path metric for that path. Incremental agreements are called branch metrics.

Figure 2-2 shows a prototype for one state of the state tree for this decoder. **Figure 2-3** shows the entire tree used by the decoder to track the first five input pairs. The state of the recreated encoder appears in the box. Along each arrow (a state transition) appears a number pair showing the corresponding encoder output bit pair for that state transition (the output produced immediately before the state changes). This figure is ordered so that transitions corresponding to an input 0 are always the upper path and transitions corresponding to an input 1 are always the lower path. To determine the encoder output for a given transition, load the encoder with the state and use a 0 or 1 for the input bit, according to the transition chosen. The encoder polynomials then determine the encoder output, as shown in **Figure 2-1**.

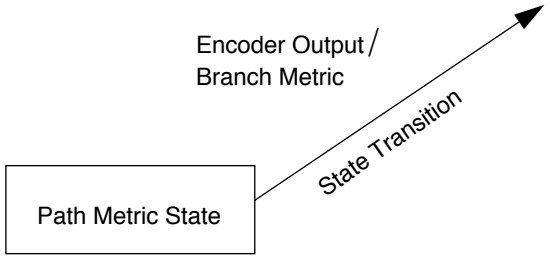

Figure 2-2 Single State of Prototype Encoder Tree Diagram

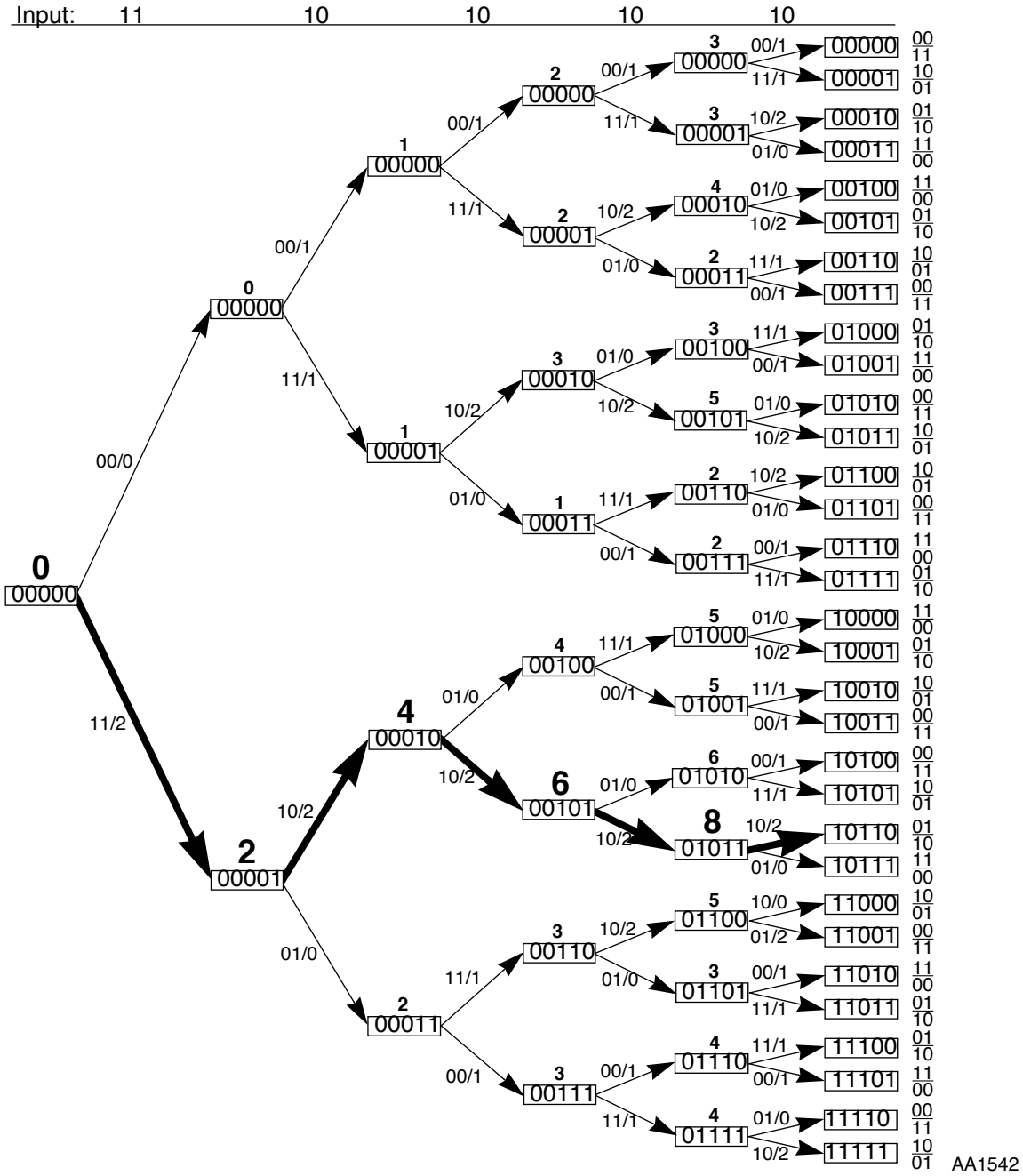

Figure 2-3 First Five Levels of the Encoder State Tree

Comparing the recreated encoder output with the decoder input, we determine the number of agreements, shown in **Figure 2-3** as the branch metric. We find a branch metric for each transition. For each state, we track the cumulative branch metrics to form the path metrics. These are shown as a number appearing above each state box.

To recreate the correct input sequence, we choose the recreated encoder path that best agrees with the decoder input data. In this case, the best path is the one with the largest final path metric. For clarity, the path metrics for the best path are distinguished with a larger font size and bolder arrows. To recreate the input sequence (so far), we can use two methods. The easiest is to use the state as the decoder output. Unfortunately, this method will not work after we finish the development of the decoder. The second method will work when we are done. To obtain the decoder output, trace the best path (the one with the largest final path metric) back to the beginning. Now, follow the same path forward again to obtain the input sequence by placing a 0 at the decoder output each time we choose an upper transition, and a 1 output each time we choose the lower transition. Using this method on the tree in **Figure 2-3** gives us 10110, which agrees with the encoder input example.

The most troublesome aspect of this decoder is that the number of states we have to track for each decoder input is actually the number of possible paths. For this coding example, the number of states doubles for each input. For any reasonable number of inputs, the amount of work and storage needed for this decoder is far too large to be practical. To solve this problem, begin by noting that we don't really need all the data generated by the decoder. In particular, all the work goes toward finding the path that best agrees with the input. We only need the path that gives us the largest path metric. If we determine that a path cannot ever have the largest path metric, we can ignore that path for all future calculations.

To collapse the ever-growing tree in **Figure 2-3**, consider what happens if we continue the tree for one more pair of decoder inputs. The total number of states would be 64 for the next input pair. To keep the diagram manageable, only a pair of specially chosen states appears in **Figure 2-4**. When we extend the tree to the next state, we get states with six bits. Note, however, that the encoder we are attempting to trace only needs five bits to determine its output bits. To emphasize this, the sixth (leading) bit is separated in the state boxes. Because the extra (leading) bits do not affect the recreated encoder outputs, we can ignore them. As a result, the 64 states collapse into 32 states again. The only resulting complication is that each state now has two rather than one entering paths. **Figure 2-4** shows this state collapse as well as each state's multiple input paths in two example states. To correctly process the decoder input, we must next determine which of the input paths to keep for each state.

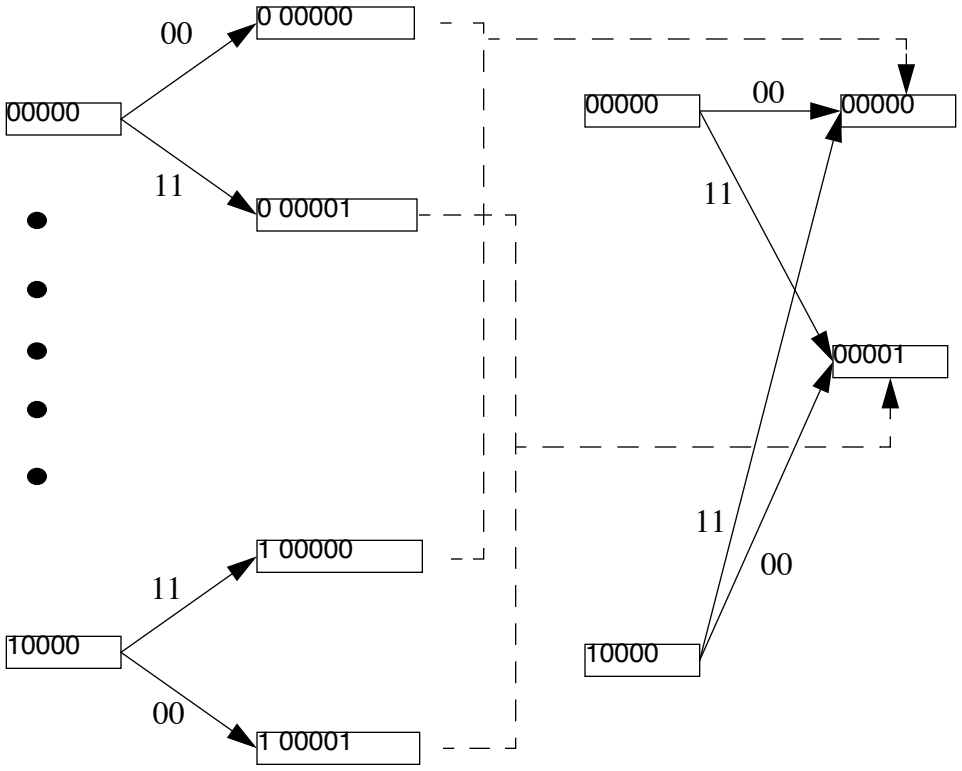

Figure 2-4 Using Multiple Input Paths to Collapse the State Tree

To find out how to choose the correct input branch for each state, we use the additivity of the path metrics. Suppose we want to find the path with the largest metric that extends from time i to time k (i<k). Also, suppose we are interested in another time j such that i<j<k. For a given state (encoder state) S at time j, consider two paths P1 and P2, extending from time i to time j, both entering state S at time j. To get from state S at time j to time k, consider two more paths Q1 and Q2. Our setup so far is illustrated in **Figure 2-5**.

Let the path metric for P1 up to time j be PM1, and let the path metric for P2 up to time j be PM2. Define QM1 and QM2 to be the partial path metrics for their respective paths. By partial path metrics we mean the contribution due to the partial path only, that is, QM1 is the path metric for path Q1 minus the value of the path metric of state S at time j. Finally, assume that PM1>PM2. Then any path containing P2 cannot be the path with largest path metric for any time after time j.

To understand why this is so, note that the total 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 $P_1 \cup Q_2$ is still larger, as PM1+QM2 >PM2+QM2 (remembering we assumed that PM1 >PM2). This is true for *any* Q2. So, if PM1 > PM2, we can eliminate P2 from further consideration at time j, *without waiting for the "future" inputs from time j to time k*.

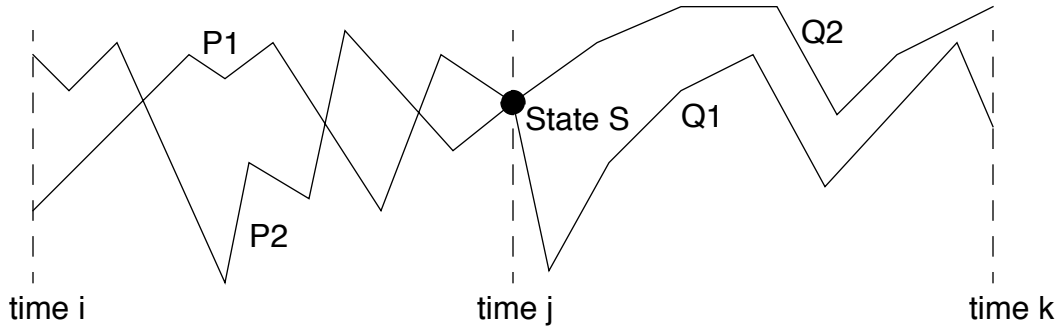

Figure 2-5 Using Intermediate States to Eliminate Partial Paths

In our example, each state will have two paths entering it at any time. The number of paths doubles each time, but we can now eliminate half of the paths each time as well. This is the basis of the Viterbi algorithm. To use it, we must keep track of the best path metric for each state up to the current time. By eliminating paths that can never have the largest path metric, however, we keep the amount of computation constant with time. To diagram what is going on, we need a different figure than the tree in **Figure 2-3**. We need only keep track of each (encoder) state at each time. The result is called a trellis because it resembles one. A diagram of a trellis for our IS-136 code appears in **Figure 2-6**.

The beginning of the trellis does not look the same for each time (each decoder input is a *stage* in the diagram). This is because we assumed the encoder started in state 00000. In general, the encoder can be in any state, and the trellis repeats. For our figure, this can be seen in stages 6 and 7 (after 5 stages, our encoder can be in any state, so we get a "steady state" in our trellis diagram). Each line in the trellis is a transition, just like the ones in the tree. For the trellis, however, the number of states is bounded. The number of states for this trellis is determined by the encoder which has five state bits. This means there are $2^5 = 32$ states in the trellis.

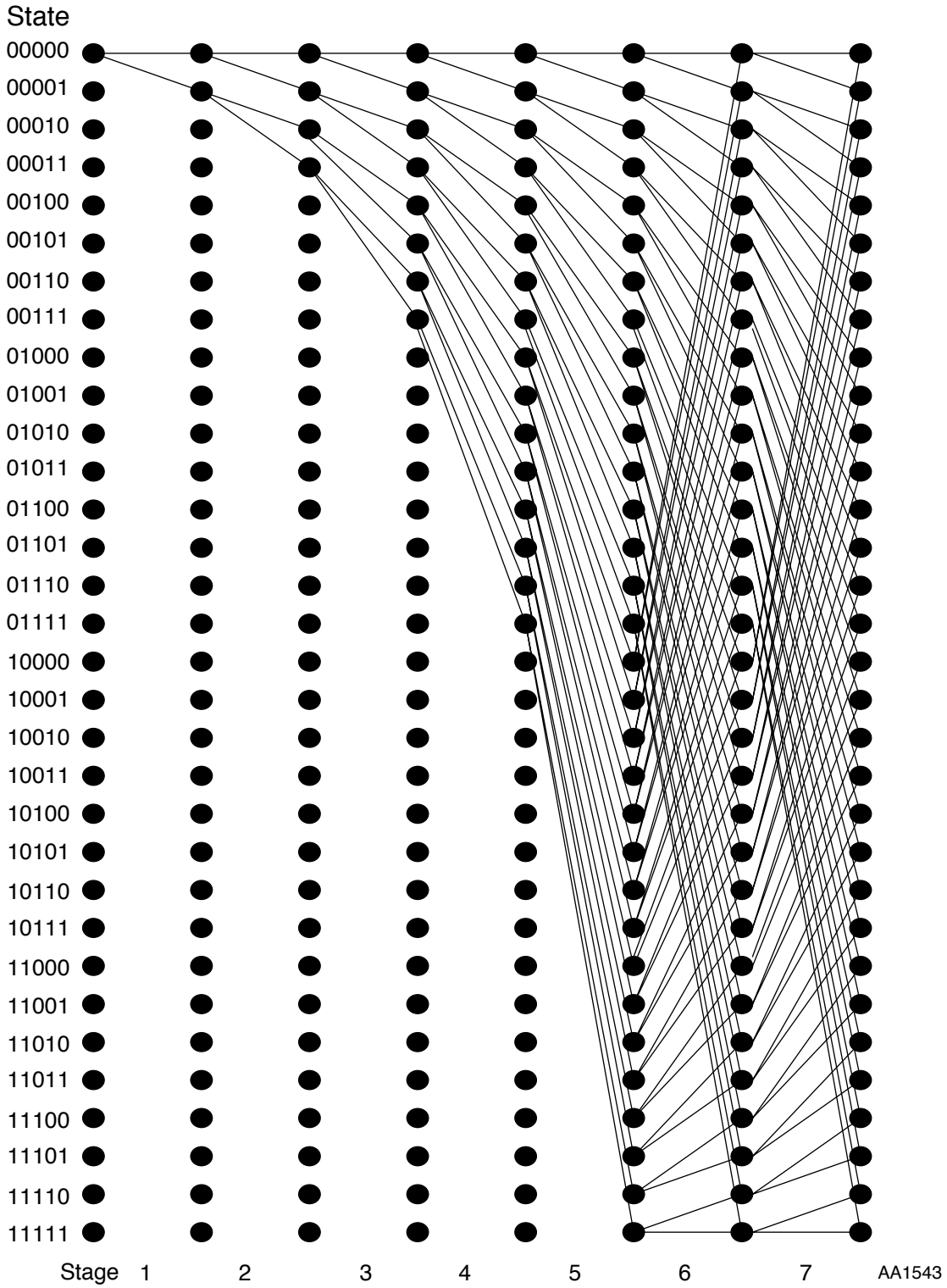

Figure 2-6 Trellis Structure for Viterbi Decoding

Fortunately, we do not have to examine all of this trellis at once! For an idea of the basic processing, we can examine states in pairs. In particular, note that we only need path metrics of two source states to update a pair of destination states. Note that the source state pairs are not the same as the destination state pairs.

For our example code, the source state pair 0ABCD and 1ABCD provides the path metrics needed to update the destination state pair ABCD0 and ABCD1, where ABCD is fixed for each state pair update. Let's examine a single pair update. This is sometimes called a Viterbi butterfly. An example butterfly is shown in **Figure 2-7**.

.

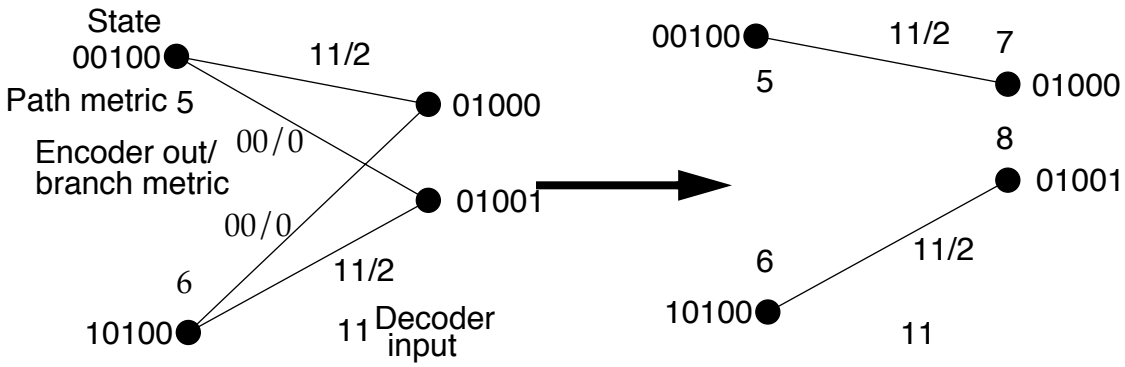

Figure 2-7 Example Viterbi Butterfly

To compare the input paths to the state being updated, we must take the path metric for each state on connecting input branches and add the branch metric associated with that path. In the example shown in **Figure 2-7**, the lower branch from state 00100 has a recreated encoder output of 00. As noted before, this can be determined by finding the encoder output for a state of 00100, along with an input bit of 1 because we are taking the lower branch. Assuming a decoder input of 11, we find that neither bit of the recreated encoder output agrees with the decoder input. We assign a branch metric of 0 (no agreements). We can do similar assignments for the other branches, as shown in **Figure 2-7**.

To update the states, we add the path metrics to the branch metrics. Then we compare the sums and choose the larger as the surviving branch. The sum of the surviving path becomes the path metric for the updated state. We must do this update for all state pairs, for each decoder input. When implementing a Viterbi decoder in software, this update is the most computation-intensive and creates a do loop that should be optimized to obtain good performance. In the next section, we will develop DSP56300 assembly code to implement this algorithm.

2.4 ALGORITHMIC ENHANCEMENTS

There are a number of modifications that can be made to the algorithm in order to improve its effectiveness as well as our ability to implement it. In our example, we count agreements. This produces branch metrics of 0, 1, and 2. In our example butterfly, however, the branch metrics into the state pair are all 0's and 2's. This is not entirely a coincidence. For this code, the encoding polynomials both have taps at both extremes. What this means is that a 1/0 at the end of the encoder will invert/not invert the encoder output. For our trellis, this means that the pair of branches into a given state will always have complementary branch metrics (0 and 2, or 1 and 1). By combining this observation with one more property, we can simplify the Viterbi butterfly computation.

The additional property for this code (shown in **Figure 2-8**) is that a 1/0 at the input bit will invert/not invert the encoder output. For our trellis, this means that the two states of a state pair being updated will have the same branch metrics, but with their order reversed (0 and 2 will become 2 and 0, etc.). These polynomial conditions are not true in general but do occur quite often. To take advantage of them, we subtract 1 from every branch metric we produce. This gives branch metric pairs of (-1,1) and (0,0).

As shown in **Figure 2-9**, each pair now has odd symmetry in the two components. Hence, we need not compute branch metrics for both branches in an update. Instead, we can find just the uppermost branch metric, and add it as usual for the upper branch. Next, subtract rather than add it to do the lower branch processing. To process the lower state update, we subtract the branch metric for the upper path and add to update the lower path. The total path metrics change, but their relative values do not. Since we are only looking for the path with the maximum path metric, the results are the same.

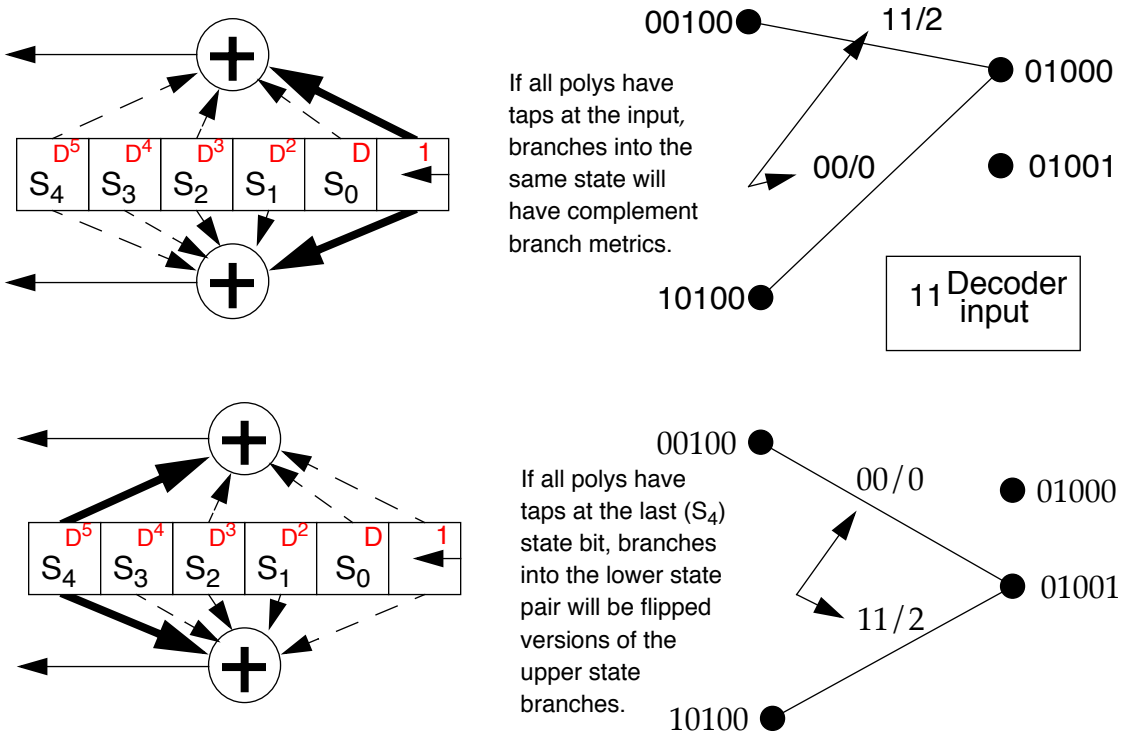

Figure 2-8 Encoder Tap Symmetry to Reduce Calculations

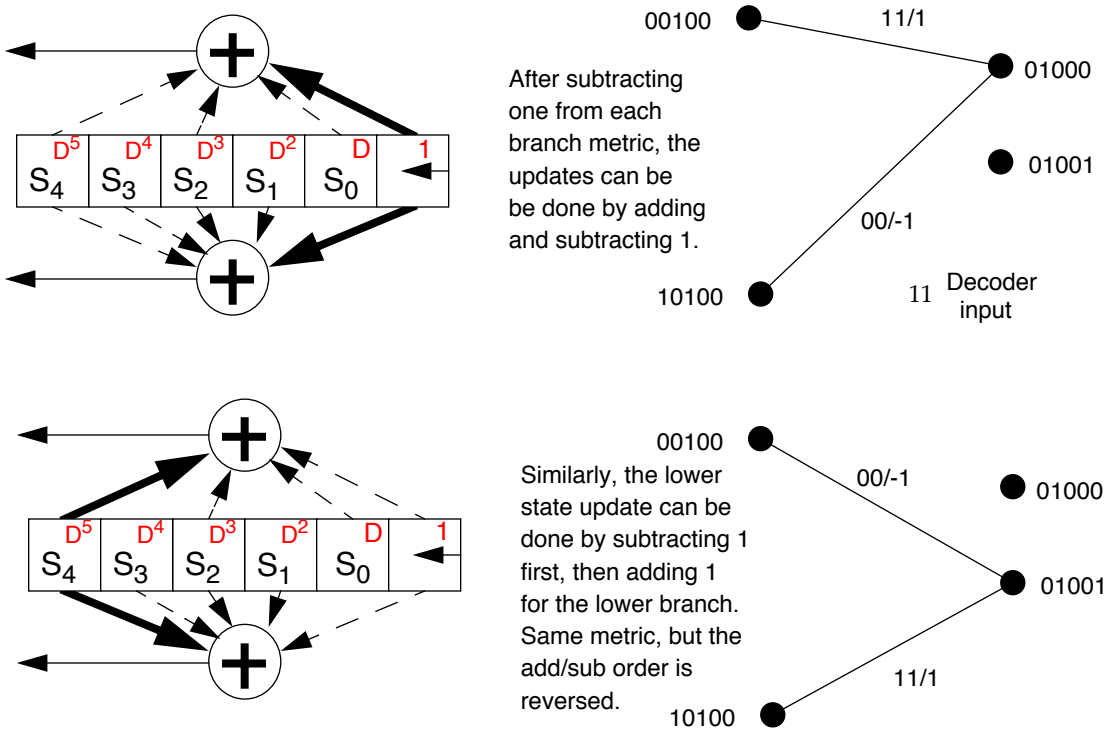

Figure 2-9 Applying an Offset to Obtain Odd Symmetry in the Branch Metrics

A second modification to the algorithm is the ability to use soft decisions. By this we mean that the input data need not be just a 0/1 decision on a receiver statistic. We can assign intermediate values to indicate the reliability of the decision. In this way, a 0.001 can be a very reliable 0, a .75 can be a moderately reliable one, and a .49 is just barely a zero. The Viterbi algorithm generalizes to handle soft decisions with no trouble; the additions for the metric updates simply have more bits. By using soft decisions, the ability of the decoder to correct errors is greatly improved.

Note that from an implementation on our DSP view, this modification is almost invisible, as 16-bit or 24-bit arithmetic will be done regardless of the number of bits in the branch metrics. Of course, provisions to protect path metric values from overflow *may* have to change for some systems.

SECTION 3

EXPANDING THE VITERBI ALGORITHM

3.1 INTRODUCTION

Having introduced the principal concepts involved in the Viterbi algorithm, we can proceed to generate code to implement it. There are, however, a number of details that need attention. We have concentrated on the Viterbi butterfly.The next step involves, among other things, generating the branch metrics and allowing for multiple word traceback to recover the output data. Up to now, we have not even discussed how the path decision might be stored.

The example code in this section shows a basic implementation of the Viterbi algorithm implemented on the DSP56600 assembly language. These examples decode data encoded using the convolutional code for IS-136.

3.2 PARTITIONING THE TASK

In this section, we develop code to implement the entire algorithm. We will start with the most critical section, the Viterbi butterfly code. We then work outward, developing branch metric storage code and traceback instructions.

All code assumes 16-bit registers and will run as-is on the DSP56600 DSP family. It also runs correctly on the DSP56300 family of DSPs when they are set in 16-bit arithmetic mode.

A complete code listing of the modules presented here appears in **Appendix A**. The code can be easily modified for 24-bit operation by changing one line of code, as shown in the detailed traceback code discussion. Note that the test data provided is 16-bit data, and would have to be changed to 24-bit data to test correctly in 24-bit arithmetic mode. Also, **Appendix C** has a 24-bit version of the code discussed in **Section 4**.

3.3 THE INNER LOOP: VITERBI BUTTERFLIES

We begin by discussing the code used to do a Viterbi butterfly (updating the states). The easiest way to present the code for this section is to display the instructions, then discuss their function. The code utilizes the Viterbi shift left (VSL) instruction, designed for DSP56300 and DSP56600 DSP families to enhance their ability to execute the Viterbi algorithm. The code appears in **Example 3-1**.

Example 3-1 Using the VSL Instruction in a Viterbi Butterfly

```
;*******************viterbi add, compare, select butterfly macro***
;       FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;               doing an add, compare, select update for state pairs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should offset addresses by NUMSTATES/2
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,x0,y1,r2,r4,r5,n5 r2 unchanged (modulo required)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;               x: path metric, y: path, states ordered assuming
;               bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;               as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C /
;        D \  /
;           \/
;           /\
;        D /  \
;         /    \
;       SB------NSB
;           C
;*****************************************************************
;
ACS     macro
;
;       move    #BRY,r2                                 ;r2 points to branch metrics
        move    y:(r2)+,y1                              ;get first branch metric
        move    l:(r5)+n5,a                             ;load 1st metric/path pair
;
        do      #NoOfAcsButt,_P_NextStage               ;update each state
        sub     y1,a    l:(r5)-n5,b                     ;sub pt,br met,get next pair
        add     y1,b                                    ;update metrics
        max     a,b     l:(r5)+n5,a                     ;pick max,reload 1st pair
        vsl     b,0,l:(r4)+                             ;store survivor, end top half
        add     y1,a    l:(r5)-n5,b                     ;add pt,br met, reload next pair
        sub     y1,b    x:(r5)+,x0      y:(r2)+,y1      ;inc st ptr,ld nxt br met
        max     a,b     l:(r5)+n5,a                     ;pick max met,load next pair
        vsl     b,1,l:(r4)+                             ;st survivor,end 2nd half
_P_NextStage
        nop                                             ;needed to separate do loop ends
        endm
```

We now explain the routine line by line. Begin by moving the address of the precomputed branch metric table to the r2 address register. In our final, combined routine, this is unnecessary because the desired value already exists in r2, but we include it as a comment here to make it clear that this value is needed. Branch metric creation will be discussed in **Section 4.2**.

Next, we update the address register that points to the states used to update. The memory that stores the path metrics is divided into two parts. In this routine, r5 points to the old path metrics, used to update the new ones. Address register r4 points to the new states, those being updated. For each new decoder input, we swap the memory used to store the updated path metrics with the memory pointing to the old path metrics. To do this, we initialize address registers r4 and r5 to be modulo registers, with a modulus of twice the number of states. Address incrementing for the butterfly is so designed that at the end of the loop, r4 and r5 have automatically swapped pointer values. Thus, no instructions need be dedicated to swapping the memory.

The path metrics are stored in X memory. Collocated with them in Y memory are the paths. By paths, we mean a word that contains the bit decisions describing the recreated encoder input data that would produce that path in the decoder. How we get these will become clear below as we go through this code. By collocating the path metrics with their respective paths, we can get both by doing long memory moves. The next two moves load the first branch metric, and the first (state 00000) path metric/path pair.

Now we are ready for the loop. It is necessary to update every state, and we update states in pairs. Hence, the number of loop passes is equal to the number of states divided by 2. In our example, this is the value of NoOfAcsButt (i.e., 16).

Because we have polynomials with taps at both ends of the encoder, we can use one branch metric, and add and subtract to update our states. We begin by subtracting the branch, loading the second path metric/path pair at the same time.

Next, we add the branch metric to the path metric we just fetched. Note that for both the subtract and add of the metric, the path metrics in A1 and B1 are affected, but because we are not performing a long word add, the paths in A0 and B0 are not.

To compute an updated metric, choose the largest result of the subtract/add operations. The MAX instruction puts the updated result in B. Note that all of A is transferred if A1 is the survivor path, which means that B0 holds the path bits that represent the survivor path. This instruction also reloads the first path metric/path pair for use in updating the lower state later in the loop.

The next instruction is the VSL. VSL is a mnemonic for Viterbi Shift Left, a new instruction tailored for Viterbi algorithm updates. The action of this instruction is to take an accumulator a or b, store the mid-register (a1 or b1) in X memory, shift the accumulator left, append a 0 or 1, as indicated by the instruction arguments, and store the low register (a0 or b0) in Y memory. The net result is that path bits are updated and the path metric/path pair stored in memory.

This VSL puts a 0 in the LSB of B0. In this way we store the input bit of this transition (which is 0 regardless of the path chosen because the input bit is 0 as long as the destination state is the upper state). Because we are storing recreated encoder bits, this path string will be a recreation of the encoder input, which is what we want for the decoder output. Of course, we can only store 16 or 24 bits of the path at a time (16 for DSP56600 or DSP56300 in 16-bit arithmetic mode, 24 for DSP56300 otherwise).

Next, repeat the operations for the lower state update. Note we subtract and add, rather than add and subtract, as required by the lower state update. The add instruction reloads the second path metric/path pair, while the sub instruction increments the path metric fetch pointer and loads the branch metric from the branch metric table, both for use in the next loop iteration. The increment of the path metric fetch pointer is a dummy read into x0 (i.e., x0 is not used). This allows us to use a second parallel move to increment r5.

The max instruction finds the survivor path metric/path pair for the lower state, and preloads the path metric/path pair for the next loop iteration. The last VSL instruction shifts the survivor path left 1 bit, appends a 1 to represent the recreated encoder state for the lower state, and stores the lower state path metric.

We now iterate this loop by the number of butterflies needed, and exit the macro.

3.4 CREATING THE BRANCH METRICS

To present the branch metric routine, we start with the encoding polynomials. For this example, we start with the encoding polynomials. We then show how to create the branch metrics we need (in the order required) to do the butterfly correctly. Recall that the encoding polynomials are $1+D+D^3+D^5$ and $1+D^2+D^3+D^4+D^5$. There are 32 states in the decoder, so we need 32 branch metrics. In general, we would access these metrics in pairs in the butterfly routine, updating the states in pairs. As noted above, we can save the work involved in half of these, because our polynomials induce a symmetry in the branch metrics. By this we mean that the branch metrics for this code have the property that the upper and lower input branches to any state are *complementary*.

Thus, by using a shifted version of the branch metrics, we only need one branch metric for each state pair update. This metric is the one that gets subtracted from one path and added into the other. We need only generate and store 16 branches.

Any given branch will, for a specified input, have one of four values. The four values will represent how well the decoder input compares to the recreated encoder outputs of 00, 01, 10, 11. For our example, we assume some soft decision input. The decoder input data is shifted and scaled so that 1 is changed to 16, and 0 is changed to -16. We obtain two partial branch metrics B_0 and B_1, where B_0 is the partial branch metric for the encoder polynomial $1+D+D^3+D^5$ output, and B_1 is the partial branch metric for the $1+D^2+D^3+D^4+D^5$ polynomial output. We can then compute the four branches as B_0+B_1, B_0-B_1, $-B_0+B_1$, and $-B_0-B_1$, corresponding to recreated encoder values of 11, 10, 01, and 00.

The butterfly loop needs the branches in a specific order dictated by the encoding polynomials. As seen in **Figure 2-2**, we update states 2i and 2i+1 using states i and i+16. We only need the 16 branches for the 16 state pairs. The state gives us five of the six values needed to determine the encoder output. The remaining value can be found by examining the butterfly loop assembly code. In the loop, note that we subtract the branch metric first (from the upper branch), then add it to the lower branch. The butterfly loop is written assuming that the single branch metric value has the correct sign to be added in the lower branch. This is equivalent to assuming the encoder input bit is a 1 (for the purposes of computing the branch metric table). We account for this in the code by choosing positive 16 for 1's and minus 16 for 0's (as noted in the preceding paragraph). The resulting recreated encoder outputs for the first 16 states are shown in the following table.

Table 3-1 Recreated Encoder Outputs

State	Input Bit	Encoder Output	Map Encoder Output to Branch Metric	Branch Metric for Example Decoder 10 Input
00000	1	11	B_0+B_1	16 + (-16) = 0
00001	1	01	$-B_0+B_1$	(-16) + (-16) = -32
00010	1	10	B_0-B_1	32
00011	1	00	$-B_0-B_1$	0
00100	1	00	$-B_0-B_1$	0
00101	1	10	B_0-B_1	32
00110	1	01	$-B_0+B_1$	-32
00111	1	11	B_0+B_1	0
01000	1	10	B_0-B_1	32
01001	1	00	$-B_0-B_1$	0
01010	1	11	B_0+B_1	0
01011	1	01	$-B_0+B_1$	-32
01100	1	01	$-B_0+B_1$	-32
01101	1	11	B_0+B_1	0
01110	1	00	$-B_0-B_1$	0
01111	1	10	B_0-B_1	32

The code needed to implement the values in this table appears in **Example 3-2**.

Example 3-2 Find Branch Metrics Code

```
;*****************BRANCH METRIC MACRO*************************************
;       FUNCTION: Input data and generate branch metrics.
;               For this decoder, the metric is a scaled
;               sum or difference of the real and imag inputs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r1 should point to the next input XY data pair
;       OUTPUTS:
;               Branch metrics are stored at BRX in XY memory
;       REGISTERS USED:
;               a,b,x01,y01,r1,r2,n2, r2 unchanged (modulo req'd)
;*************************************************************
;
FindMetrics  macro
        move    l:(r1)+,y               ;grab dec input
        move    #-16,x1                 ;sign for real component, 0 sent.
        mpy     x1,y1,a                 ;a has 0x partial branch
        move    #BRY+3,r2               ;storage for generated branch metrics
        mac     x1,y0,a         a,b     ;a gets 00 branch
        mac     -x1,y0,b                ;b has 01 branch
;

        neg     a       a,x1    a,y:(r2)+n2 ;mv 00 to x1,11 to a, st 00 in location 3
        neg     b       b,x0    b,y:(r2)+n2 ;mv 01 to x0,10 to b, st 01 in location 6
;*************************************************************************
;         AT this point X1 has 00, X0 has 01,
;         A1 has 11, B1 has 10, needed for quick storage in Y memory
;*************************************************************************
        move    x1,y:(r2)+n2            ;store 00 in location 9
        move    x0,y:(r2)+n2            ;store 01 in location 12
        move    b,y:(r2)+n2             ;store 10 in location 15
        move    b,y:(r2)+n2             ;store 10 in location 2
        move    b,y:(r2)+n2             ;store 10 in location 5
        move    b,y:(r2)+n2             ;store 10 in location 8
        move    x0,y:(r2)+n2            ;store 01 in location 11
        move    x1,y:(r2)+n2            ;store 00 in location 14
        move    x0,y:(r2)+n2            ;store 01 in location 1
        move    x1,y:(r2)+n2            ;store 00 in location 4
        move    a,y:(r2)+n2             ;store 11 in location 7
        move    a,y:(r2)+n2             ;store 11 in location 10
        move    a,y:(r2)+n2             ;store 11 in location 13
        move    a,y:(r2)                ;store 11 in location 0. r2-> BRY
        endm
```

The FindMetrics macro is for the most part straightforward. We begin by loading the decoder input data to compute the branch metrics. Next, load the scaling factor, and multiply to obtain the partial branch metric with one of the inputs. The next line initializes address register r2 as a pointer to the table location used by state 3, where the branch metrics are to be stored. We then finish the computation of the branch metric for a branch with encoder output 00, followed by the metric for 01. With these two metrics, we begin to load the branch metric table.

We can load the branch metric table in any order. To minimize the number of cycles needed, we apply a few constraints. First, it is easier if we load the table in some consistent manner such as using a constant address offset each time. Second, it is convenient if we end up at the address used by state 0, because then we are initialized for the butterfly state update. Also, we must finish generating the metric values for 11 and 10. It is most efficient if we can do this in parallel with the moves to load the branch metric table.

The easiest way to store the branch metrics might be to start at state 15 and count down. If we do this, it turns out that a stall is unavoidable: we cannot generate all of the required branch metric values in time to use them with this ordering. Other loading orders are easy to obtain by using an address register increment that is relatively prime to the table length (and using a modulo addressing mode to wrap around). For our table length, any odd number increment will ensure that we cover the entire table with constant increments. Our first candidate is 3. If we start at state 3, and increment by 3's, we will cover the entire table and end up at the address for state 0. These first three branch metric values are associated with recreated encoder outputs of 00, 01 and 00. Because the repeat of 00 gives us time to generate the extra branch metric values with no stall, we use increments of 3.

We begin by writing the branch metric for state 3. At the same time, we negate A to compute the metric for branches with encoder output 11, saving the metric for 00 in x1. For the next write, we negate B to compute the metric for 10, saving the metric for 01 in x0. We write to the branch metric table at the same time.

What follows are writes to the branch metric table. Using modulo addressing, we increment the address by 3's until the entire table is filled. We write the branch metrics determined by the encoder polynomials as they appear in **Table 3-1**.

Finally, note that this code has been constructed so that the address register r2 points to the beginning of the branch metric table at the end of this routine. Because of this, the butterfly loop is automatically initialized to load branch metrics by the end of this routine.

3.5 STORING THE PATHS

For some applications of the Viterbi algorithm, the paths will not need to be stored. This occurs when the memory of the encoder (or channel, for maximum likelihood sequence estimation) is small enough for the paths in the algorithm to merge quickly. In other situations, this may not be adequate. To clarify the storage process, we introduce the idea of traceback.

Assume the decoder has been processing data for some time. If we examine the paths up to the time of the current input, it is not clear what path should be chosen. In fact, choosing a path to decide what decoder output should correspond to the current input would lead to poor performance. This is because we do not have all the data we can get. Because the current encoder input has an effect on the decoder inputs for some time into the future, future decoder inputs can be used to increase the reliability of current decoder output decision values. As a result, we introduce delay into the decoder, waiting for future inputs to be processed before making a decision.

For the Viterbi algorithm, this means that while the current array of paths may not have reliable decisions about the present, they do contain reliable data about the past. In practice, there is a traceback depth for the decoder, the number of states we need to look back before the decision data is assumed to be reliable. For this depth, it is highly likely that the paths have converged. This means that all current paths have the same data up to this time in the past. If this depth is less than the register size used for the paths (i.e., 16 bits for the DSP56600 and the DSP56300 in 16-bit arithmetic mode, 24 bits for DSP56300 otherwise), we can choose a path and output the most significant bits of the register as decisions.

If this is not the case, we must make some provision to store the paths. For our example, a traceback depth of 16 bits is inadequate. A depth of 24 bits is extremely marginal for this example, while 35 bits is probably more than needed. Hence, we implement traceback storage. For this example, we actually store paths for the entire sequence length (assumed to be 168 bits here), and produce the entire output at the end. It should be easy to modify the code to produce partial output data in stages if desired.

The critical concept in storing paths is to arrange the path data so that we can connect the paths for consecutive storage times. The path bits of the survivor path bits at the end must be joined with their preceding path bits. Because we stored the preceding path bits in memory, we must know where in memory to look for the survivor path. Where in memory do we look?

To see how this is done, consider an example path in register B0 just after a Viterbi butterfly update. Assume there are 13 pairs of decoder inputs processed so far.

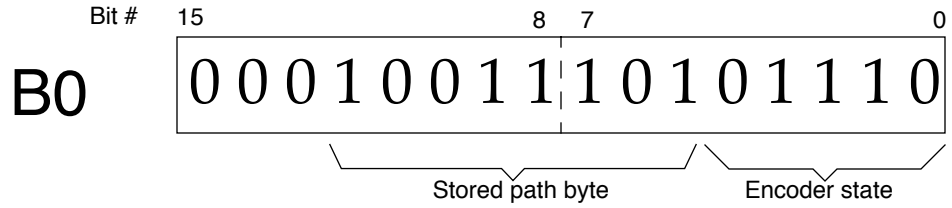

Figure 3-1 Stored B0 Path

Note that the current encoder state is part of the path! In retrospect, this must be so, as both the state (the recreated encoder state) and its path (the potential decoder output—the recreated encoder input) represent the same data stream. To use this, store the path byte in a memory location with address defined by the time the path bytes are stored, offset by the current encoder state. Then, make sure the saved partial path does not contain the current encoder state. Instead, save the current encoder state part of the path to be stored the next time we save paths.

To do the traceback, start with the survivor path at the end, and trace the path bits back in time to recover the decoder output. When we recover the stored path as part of our traceback, the final bits in that path point to the offset memory location where we need to read to continue the traceback.

Hence, if we omit the most recent 5 bits of the path, we can potentially store up to 11 bits of path (assuming 16-bit registers). We have chosen to store 8 in this example because reconstruction is a little easier if we have data in bytes. There is a trade-off. Storing more path bits at a time means we don't need to store paths as often (fewer instructions executed, less memory used), but the traceback reconstruction is more difficult because we have to piece together 11-bit data into 16-bit words.

The code in our example code makes one more assumption. We assume that the data is sent in a block, and that the transmitter ends the block by 0 filling the encoder. The effect of 0 filling the encoder is to end everything at state 0. Hence, for traceback, start everything at state 0.

For other codes that do not use this assumption, traceback can either begin at state 0 anyway (relying on the convergence of paths to give the correct output at the traceback depth) or the decoder can search for the state with the maximum path metric to begin the traceback.

3.6 TRACEBACK: OBTAINING THE DECODER OUTPUT

This section presents the code needed to obtain the decoder output from the stored path data. For this example, we have data organized in blocks of 168 bits. Because the encoder is flushed to the zero state at the end, and the data for all 168 bits is stored, the traceback can obtain the entire block at once. The most complicated portion of this code is the need to obtain the pointer to the next previous state from the current path data.

The code listings for storage and traceback appear in **Example 3-3** and **Example 3-4**, respectively.

Example 3-3 Partial Path Storage Code Listing

```
;***********************STORE PARTIAL PATH METRICS MACRO***************
;       FUNCTION: The storage is somewhat twisted.  The stored paths are current
;                 up to the most recent input bit, which is NOT convenient for
;                 traceback.  I process the data as follows:  I pre
;                 loaded the path with 6 bits.  Thereafter, I process 8 path bits
;                 so the path has 14 bits.  The most significant 8, I save.
;                 the remaining bits are the current encoder values for that path.
;                 The 5 lsb's are the current state.
;       INPUTS:
;                 n0 should point to memory where the path bits are to be stored
;                 r5 should point to the latest path metric for state 0
;       OUTPUTS:
;                 Updated paths storage X memory, n0 points to next path storage
;       REGISTERS USED:
;                 a,b,x1,r0,n0,r3,r5
;*********************************************************************
;
;       Store off path data in bytes to avoid overflow in path reg's
;
STOREPATHS macro LPCNT
;
        move    n0,r3                           ;n0 stores path data pointer
        move    n0,r0
        move    #>$1f,x1                        ;mask for 5 lsb's(NUMENCBITS of 1's)
        do      LPCNT,_PSTORE1                  ;store paths for each state
        move    y:(r5),b                        ;grab path
        asr     #5,b,a                          ;align bits 5-12 with a1
;                                                (the 8 bits beyond enc)
        and     x1,b                            ;mask off 5 lsb's to return to path
        move    a1,x:(r3)+                      ;store 8 ms path bits
        move    b1,y:(r5)+                      ;return 5 ls path bits
_PSTORE1
        lua     (R4+NUMSTATES),R5               ;r5 points to latest states
        lua     (r0+NUMSTATES),n0               ;update path data pointer
        endm
```

To store the paths, begin by recalling the path storage address pointer. In the example code, this is stored in n0. We place copies into r3 (to use as the address pointer for data writes) and r0 (to update the pointer to the next *page* at the end of the routine). We then mask off the 5 LSBs, and proceed to store the paths. We have designed this macro to process a variable number of states, controlled by the macro argument LPCNT. This variability will be used in **Section 4**.

For each path stored, then, we start by reading in the path. We move the path 5 bits to the right, so that A1 has the byte of path data to store right aligned. We mask off all but the 5 rightmost bits in B1 to clear the stored path bits. The loop finishes by writing both the path data to be stored and the updated path for the path metric.

The macro finishes by setting r5 for use in the next butterfly, as well as updating N0 to point to the next page of stored path data during the next storage interval.

Example 3-4 Traceback Output Path Code Listing

```
;***********************TRACEBACK OUTPUT PATH MACRO******************
;       FUNCTION: To output the correct data, we begin at the end. We take the
;               output path of the survivor state (0), and place its associated
;               output path in memory as the last output data byte. Then we use
;               bits 3-7 of that data as an offset pointer to the correct traceback
;               data of the next previous path data memory. We continue this until
;               we have traced the data back to the beginning.
;       INPUTS:
;               n0 should point to memory where the path bits are to be stored
;               r5 should point to the latest path metric for state 0
;       OUTPUTS:
;               Decoder output data in Y memory
;       REGISTERS USED:
;               a,b,y,r0,n0,r2,r5
;
;*********************************************************************
;
;       Store off path data in bytes to avoid overflow in path Registers
;
TRACEBACK macro
;
        move    N0,R0                           ;path ptr to r0
        move    #0,n0                           ;prep for traceback
        move    #DECOUT+(NUMINPUTS/8/2),r2      ;point to end to trace data
        move    y:(r5),b                        ;recall last path
        move    #-1,m2                          ;r2 now linear
        move    b1,x:(r0)                       ;save off last path data
        move    #$513,x0                        ;control word for extract, 16 bit
;       move    #$501d,x0                       ;control word for extract, 24 bit
;
;*****************BEGIN TRACEBACK*****************************
;
        IF (EVEN==0)
        move    x:(r0+n0),a                     ;recall last path
        extractu x0,a,b                         ;bits 3-7 of a1 point to next data
        lua     (r0-NUMSTATES),r0               ;dec r0 to next earlier state set
        lsl     #8,a                            ;move to upper byte
        move    b0,n0                           ;load as offset for traceback
        move    a1,y:(r2)-                      ;save off
        ENDIF
;
```

Example 3-4 Traceback Output Path Code Listing (Continued)

```
            do      #NUMINPUTS/8/2,TRCBK          ;once for each byte pair
            move    x:(r0+n0),a                   ;recall last path
            extractu x0,a,b                       ;get ptr to next earlier path
            lua     (r0-NUMSTATES),r0             ;point r0 to next earlier states
            move    b0,n0                         ;save ptr as offset
            move    a1,x1                         ;save out byte in x1
            move    x:(r0+n0),a                   ;do it all again!
            extractu x0,a,b
            lua     (r0-NUMSTATES),r0
            move    b0,n0
            lsl     #8,a                          ;move to upper byte
            or      x1,a                          ;or in last byte to get 16 bit word
            move    a1,y:(r2)-                    ;store result
    TRCBK
            endm
```

We begin the traceback in **Example 3-4** by moving the path storage pointer to r0 and initializing n0 for its use as an offset. Next we position r2 at the *end* of the buffer that will contain the decoder output. Again, this is because we will traceback the path from the end survivor path to the beginning. For our example, we can start at state 0. Start by taking the current path for state 0 (r5 points to this) and reading into B. We put r2 in linear addressing mode, then store the last path (the loop will read it again). Finally, we load x0 with the control code ($513) to extract the pointer to the next traceback location from the path.

The next section of code is used if the number of bytes of data is odd. It decodes the last byte, and places it in the most significant byte of decoder output. As this code is like the last half of the following do loop, discussion of this code is deferred.

The do loop processes the traceback two bytes at a time. In this way, it can assemble the decoder output into 16-bit words for more efficient storage. Begin by reading the current path into A1. Then, extract the pointer to the next (actually previous) path using extract. Recall the control register contains $513, which means we take bits $13-$17 of A or, equivalently, bits 3-7 of A1. Note that we can load $501d instead and the program will work in 24-bit mode, although the example data provided would require 00 appended to every data symbol to test correctly. As noted above, these most significant bits of the path byte point to the storage address of the path continuation in the previous “page” of memory. By page, we mean the block of stored path bits for each state. There is one page written each time we store paths (every 8 decoder input periods).

Next, move r0 to point to the previous page, then move the extracted path pointer bits to n0 to be used in the next traceback read. We end processing of this least significant byte by moving it to x1.

Again, we read the next (previous) path into A1 using r0, which points to the previous page, and n0, which points to the correct path in that page. We extract the new path pointer information, update r0 to the next previous page, and move the path pointer to n0. Move the path data to the upper byte, OR in the lower byte from x1, and move the resulting word into memory using r2. The loop continues until we have traced back to the beginning, and y:DECOUT points to the start of the decoder output.

3.7 MAIN: GLUING THE PIECES TOGETHER

The only thing left is to put the pieces together. In **Examples 3-5** through **3-7**, we present the main routine. Its function is to initialize registers so everything starts properly, and to invoke the macros at the needed time.

Example 3-5 Main Viterbi Decoding Routine: Initialization

```
;*************************MAIN*********************************************
;
NUMSTATESequ    32
ENCBITS         equ     6               ;most cases=log2(NUMSTATES)+1
NoOfAcsButt     equ         NUMSTATES/2
NUMINPUTSequ    168
EVEN            equ     1-(NUMINPUTS/8)%2  ;EVEN SET TO 1/0 IF NUMINPUTS IS
;                                           EVEN/ODD #BYTES
        org     p:$400
VITDEC  move    #NUMSTATES/2-1,m2       ;r2 points to branch metric table
        move    #>3,n2                  ;increment for branch metric storage
        move    #STATE1,r5              ;r5 points to current state metric
        move    #STATE2,r4              ;r4 points to updated state metric
        move    #NUMSTATES*2-1,m4       ;both modulo to flip loc each sym
        move    #NUMSTATES*2-1,m5
        move    #NUMSTATES/2,n5         ;input metrics spacing for each butterfly
        move    #PATHOUT,n0             ;n0 points to storage for output paths
        move    #-1,m3                  ;set linear mode, traceback ptr
;
        move    #INDATA,r1              ;r1 points to input data
;
```

Example 3-5 begins the Viterbi decoding routine by setting equates for the encoder size and input data length, and initializing address registers. NUMSTATES is set to the number of encoder states, which for our example is 32. ENCBITS is the number of bits used to encode. This includes the input bit as well as the state bits. NoOfAcsButt sets the number of ACS butterflies for the butterfly loop. For our rate 1/2 code, this is automatically set to half the number of states. The EVEN flag is used by the assembler to include an extra half-byte of traceback if the number of input bytes of data is odd.

Next, we begin the executable code. The first two lines initialize address registers m2 and n2 so that r2 addressing wraps around and increments by 3's—both for branch metric storage. The next 5 lines initialize address registers r4 and r5 to wrap around for path metric/path storage to begin them pointing to the correct places in the storage table. Offset register n5 is set to access path metrics spaced halfway apart in the storage table.

The initialization finishes by pointing n0 to the path storage table start, making sure r3 is in linear mode for traceback, and pointing r1 to the input data. Note that the r1 addressing mode is not set. We assume that an outside calling routine does this, and permit circular input buffers if desired.

Example 3-6 Main Viterbi Decoding Routine: Patch Metric Update

```
;
;***********************PREPARATION LOOP************************
;       This loop iterates by the number of bits used in the
;       encoder to pre load the bit decisions. Thereafter,
;       the paths are updated and stored off in bytes. We need
;       the preload bits so that the stored path metrics point
;       correctly to their previous paths for traceback
;***************************************************************
;
        do      #ENCBITS-1,PRELP        ;once for each encoder bit
;
        FindMetrics
        ACS
PRELP
;*****************MAIN LOOP--PROCESS BYTES OF DATA*****************
        do      #NUMINPUTS/8-1,DATALP   ;process bytes of output
        do      #8,SYMLP                ;8 bits per byte
;
        FindMetrics
        ACS
SYMLP
        STOREPATHS#NUMSTATES
DATALP
```

These next two sections do much of the decoding work. The first loop processes ENCBITS of input data. We process 1 pair of decoder inputs for each iteration of the macros FindMetrics and ACS. This preprocessing loads ENCBITS-1 of data into the paths. Thereafter, we process the decoder input pairs in groups of eight, producing 8 path bits for each state. For every 8 decoder input pairs processed, we invoke the STOREPATHS macro to store the paths.

By preloading ENCBITS-1 and storing the leftmost eight every time we store paths, we ensure that the paths always have (ENCBITS-1)+8 bits at the time we store paths.The extra ENCBITS-1 bits are, as noted above, the current state, and are kept till the next path store time to allow us to traceback. The main loop processes all but the last 8-ENCBITS+1 sets of decoder input data.

Example 3-7 Main Viterbi Decoding Routine: Termination and Traceback

```
;
;********POSTPROCESSING,LAST INFO BYTE ******************************
;
; ENCBITS PREPROCESSED, SO WE HAVE 8-ENCBITS BITS LEFT.
; FOR THIS EXAMPLE, WE HAVE 3INPUTS LEFT TO PROCESS
;*********************************************************************
;
;  THE LAST THREE BITS------
;
        do      #8-ENCBITS+1,FLSH1;process last 3 bits to get last byte
;
        FindMetrics
        ACS
FLSH1
;*******Traceback the path data to obtain the decoder output**********
        TRACEBACK
FIN     nop
```

Example 3-7 shows the final processing for the data block. At this time, we have processed all but the last byte, i.e., 8 bits less the ENCBITS-1 processing we did in preprocessing. This means we have 3 bits of data left to process. After processing the last decoder input, the TRACEBACK macro is invoked to obtain the decoder output. The decoded data is memory at y:DECOUT, in 16-bit word form.

3.8 MEMORY ORGANIZATION

Example 3-8 shows all memory organization except the input data defines used to test the code. The first reserved spaces are dedicated to the storage for the path metrics (in X memory) and their respective paths (in Y memory). Note that these must be collocated. For this code, we initialized the metrics for that state 00 has a large path metric, and the rest are 0. We do not initialize the path metrics in executable code! This allows the decoder to operate on data over multiple invocations if desired. If the decoder is operating on independent data blocks, each started assuming the encoder starts in the 0 state, then this memory will have to be initialized accordingly. Another alternative in this case would be to modify the butterfly loop in the preprocessor to use the fact that the encoder starts in the 0 state. This option is explored in **Section 4**.

Example 3-8 Memory Organization Code

```
;**********************MEMORY ORGANIZATION*****************************
;       MOST OF THE MEMORY LOCATION IS IMPORTANT--THE FIRST TWO
;       LABELS OF THE X AND Y DATA MEMORY ARE PAIRED AND MUST
;       BE CO LOCATED (AT THE SAME ADDRESSES).  IN ADDITION, STATE1
;       AND STATE2 MUST BE LOCATED ON A 0 MOD 2*NUMSTATES BOUNDARY,
;       BRY MUST BE ON A 0 MOD NUMSTATES/2 BOUNDARY.
;       X and Y memory for input data must be paired --GOOD LUCK.....
;**************************************************************
;
        org     x:$0
STATE1  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
STATE2  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
;
PATHOUT DS      NUMSTATES*(NUMINPUTS/8+1)
INDATA  ;THIS DATA ENCODES
;$1234,$5678,$9abc,$4973,$7925,$3491,$ad43,$ff21,$7ebb,$0100,$00 ;....
;I didn't include the test data in this listing...
        org     y:$0
PATH1   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
PATH2   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
BRY     DSM     NUMSTATES/2
DECOUT  DS      NUMINPUTS/8
        org     y:@cvs(y,INDATA)
YDATA   DC      $a000,$a000,$a000,$6000;....
```

The path metric/path memory must start on a 0 mod 2*NUMSTATES boundary for the modulo addressing to work properly. In addition, the Branch metric storage BRY must start on a 0 mod NUMSTATES/2 boundary to operate correctly. Finally, the storage of the input data must be paired in X and Y memory, because we do long moves to get the decoder input data pairs into the branch metric calculation.

SECTION 4

ALGORITHMIC EXTENSIONS

4.1 INTRODUCTION

This section shows how to adjust the code of **Section 3** to include several enhancements. Generalized branch metrics are allowed, as well as optimizing the code to reduce the computations when data occurs in blocks.

The code in this section introduces several modifications to the code discussed in **Section 3**. **Section 4.2** modifies the handling of branch metrics to allow nonsymmetrical branch metrics. **Section 4.3** and **Section 4.4** develop two additional macros useful if the data is blocked and the encoder is periodically forced through a known state. We put all of these routines together to produce a modified Viterbi algorithm decoder that has autonormalized path metrics and efficient coding for blocked data. Note that a complete listing of this code appears in **Appendix B**.

4.2 ALLOWING MORE GENERAL BRANCH METRICS

The code introduced so far is efficient for algorithms that have special branch metrics. In particular, the branch metrics are assumed to be inverses of each other and flipped between state pairs. Although this is true for many popular codes in use today, it is not necessarily the case. If a polynomial set is specified where one of the encoding polynomials does not have a tap at one end, the branch metrics lose some of their symmetry. A simple example is the polynomial set (1,1+D), whose trellis appears in **Figure 4-1**.

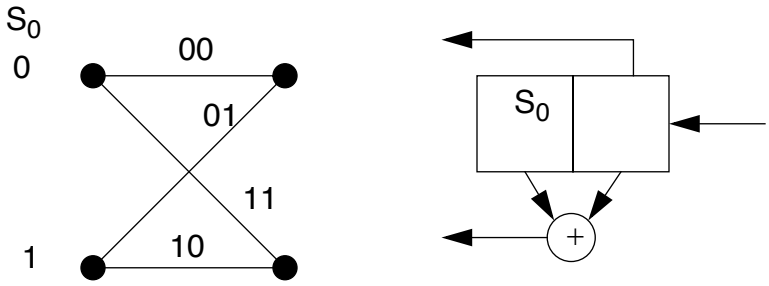

Figure 4-1 Polynomial (1,1+D) Trellis

Note that the input branches to State 0 are not complements of each other! We might accommodate this with a fractional branch offset to make branch 00 and branch 01 negatives, but note that the offset for the lower state would have to be different. Another approach is to compute separate branch offsets for the upper and lower state updates. This requires a different version of the branch metric generation as well as the butterfly loop. We consider modified code for each of these functions in the examples that follow.

4.2.1 Modify Viterbi Butterfly

Begin modification with the butterfly loop. The code from **Section 3** reads in the branch metric from y memory near the loop end (the y:(r0)+,y1 that appears two lines from _P_NextStage). The easiest way to get two branch metrics is to do long reads on the branch metrics. For the code in **Section 3**, however, finding the space to do the extra branch read is harder. Most of the data movement is tightly controlled and cannot be moved in the code without disrupting the data flow. Instead, we can make use of the pipeline stall in the Viterbi butterfly loop. The modified code appears in **Example 4-1**.

Example 4-1 Modified Viterbi Butterfly

```
;*******************viterbi add, compare, select butterfly macro***
;               FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;                       doing an add,compare,select update for state pairs.
;       INPUTS:
;                       r2 should point to the beginning of the branch metric table
;                       r2 should point to the beginning of the branch metric table
;                       r5 should point to the latest path metric for state 0
;                       r4 should point to the storage location for updated state 0
;                       n5 should offset addresses by NUMSTATES/2
;       OUTPUTS:
;                       Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;                       a,b,y01,r2,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;                       x: path metric, y: path,states ordered assuming
;                       bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;                       as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C /
;        D \  /
;           \/
;           /\
;        D /  \
;         /    \
;       SB------NSB
;           C
```

Example 4-1 Modified Viterbi Butterfly (Continued)

```
;*************************************************************
;
ACS     macro
;
;       move    #BRY,r2                 ;r2 points to branch metrics
        move    l:(r2)+,y               ;get first branch metric
        move    l:(r5)+n5,a             ;load 1st metric/path pair
;
        do      #NoAcs,_P_NextStage     ;update each state
        add     y0,a    l:(r5)-n5,b     ;sum pt,br met,get next pair
        add     y1,b                    ;update metric
        max     a,b     l:(r5)+n5,a     ;pick max, reload 1st pair
;       move    l:(r2)+,y               ;load next branch metrics
        vsl     b,0,l:(r4)+             ;save surivior, end top half
        add     y1,a    l:(r5)-n5,b     ;sum pt,br,reload next pair
        add     y0,b    (r5)+           ;sum again,inc pt mt read ptr
        max     a,b     l:(r5)+n5,a     ;pick max, load next pair
        move    l:(r2)+,y               ;load next branch metrics
        vsl     b,1,l:(r4)+             ;write survivor,end 2nd half
_P_NextStage
        nop                             ;needed to separate do loop ends
        endm
```

This code is almost identical to the butterfly code in **Section 3**, with the following differences. The first executable line is a long move to register Y. This allows us to grab two branch metrics at the same time. For this example, we show both branch metrics being added to the path metrics to update the states. Line 9 in the loop (the assembly line move l:(r2)+,y) is a long read of two more branch metrics. This is an extra line of code, but it executes in the same number of cycles as the original code because it takes the place of a pipeline stall. Finally, note that address register 5 is still incremented, but without a dummy read into a register.

The loop uses the same branch metrics for both upper and lower path metric updates. This is what is needed for **Example 4-1**, where we still have some symmetry because both encoding polynomials have a tap on the input bit. If this is not the case, the code can accommodate other conditions. A more general setup is to uncomment the long move that reads additional branch metrics in line 4 of the loop. This line will also take the place of a pipeline stall, and so does not increase the number of cycles needed to execute the butterfly.

4.2.2 Modify Branch Metric Generation

The branch metric generation needs modification also. **Example 4-2** shows the user a different motivation for noncomplementary branch metrics. In this example, we use convolutional code from **Section 3**, but take the path metric for state 0 and subtract it from all branch metrics. This is a computationally inexpensive way to automatically normalize the path metrics so that we need not worry about arithmetic saturation of path metrics when decoding very long streams of data. It does, however, destroy the symmetry of the branch metrics.

Example 4-2 Modified Branch Metric Generation Code

```
;*****************BRANCH METRIC MACRO************************************
;       FUNCTION: Input data and generate branch metrics.  This function
;               subtracts the path metric for state 0 from all branch metrics
;               to provide autonormaliztion. For this decoder, the metric is a scaled
;               sum or difference of the real and imag inputs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r1 should point to the next input XY data pair
;       OUTPUTS:
;               Branch metrics are stored at BRX in XY memory
;       REGISTERS USED:
;               a,b,x01,y01,r1,r2,r5, r5 unchanged,r2 unchanged (modulo req'd)
;*************************************************************
;
FindMetrics macro
        move            x:(r5),a                ;st. metric scaling, read last st. 0
        neg             a               l:(r1)+,y ;negate metric|grab dec input
        move            #-16,x1                 ;sign for real component, upper br.
        mac             x1,y1,a         a,b     ;comp 0x partial br. path mt. to
        move            y1,x0                   ;mv real input to x0
        mac             x1,y0,a         a,y1    ;a gets 00 br. y1 gets 0x partial br
        subl            a,b             b,x0    ;a has 00 br, b has 11 br, save path
        tfr             y1,a            a,y1    ;swap 0x and 00 to compute 01 branch
        mac             -x1,y0,a        b,x1    ;a gets 01 br
        tfr             x0,b            b,y0    ;swap path metric,11 branch to y0
        subl            a,b             y1,x0   ;b gets 10 br, x0 has copy of 00 br.

;*************************************************************************
;       AT this point, X is configured with br 11|00, y with 00|11, and
;       AB with 01|10, BA with 10|01 all needed for quick storage in XY
;       memory.
;*************************************************************************
```

Example 4-2 Modified Branch Metric Generation Code (Continued)

```
move            x,l:(r2)+               ;store 11 in location 0.
move            ab,l:(r2)+              ;store 01 in location 1
move            ba,l:(r2)+              ;store 10 in location 2
move            y,l:(r2)+               ;store 00 in location 3
move            y,l:(r2)+               ;store 00 in location 4
move            ba,l:(r2)+              ;store 10 in location 5
move            ab,l:(r2)+              ;store 01 in location 6
move            x,l:(r2)+               ;store 11 in location 7
move            ba,l:(r2)+              ;store 10 in location 8
move            y,l:(r2)+               ;store 00 in location 9
move            x,l:(r2)+               ;store 11 in location 10
move            ab,l:(r2)+              ;store 01 in location 11
move            ab,l:(r2)+              ;store 01 in location 12
move            x,l:(r2)+               ;store 11 in location 13
move            y,l:(r2)+               ;store 00 in location 14
move            ba,l:(r2)+              ;store 10 in location 15 r2-> BRX
endm
```

For this example, we are using the same code as in **Section 3** to make the unnormalized branch metrics symmetrical. For easy storage in memory, we want to generate metric pairs. Each pair should have the values needed for the upper and lower branch metric value needed for the butterfly. For this example, the branch metric for encoder output 00 will be paired with the branch metric value for 11 and the value for 01 paired with the value for 10. All branch metrics will have the previous path metric for state 0 subtracted from them. This means that after the Viterbi butterfly update, each path metric value will have been normalized. It can be shown that this limits the worst case path metric values to be no more than 12 times the maximum branch metric value (for this code example).

The branch metric macro begins by reading in the last path metric for state 0. It then negates that value in accumulator a and reads the decoder input into register y. The data in y is assumed to be in the form y1:y0 mapped to real:imaginary data input. Next, the scaling factor -16 is placed in register x1.

The mac instruction finds a partial branch metric by multiplying the real data by the scaling factor and adding the result to the negated path metric value in a. We also save the negated path metric value to b for later use.

Next, we move the real decoder input from y1 to x0. The next mac instruction multiplies imaginary input by the scaling factor and adds it to a. The accumulator a now has the normalized branch metric for encoder 00 outputs. At the same time, the partial branch metric that was in a is saved in y1 for later use.

The subl a,b instruction does a shift left and subtract operation on b and a to obtain the branch metric for encoder 11 outputs. The negated path metric value is saved in x0. The transfer instruction and its parallel move swap y1 with a to save the 00 branch metric and place the partial branch metric for the next computation.

The next mac instruction scales the imaginary part and subtracts it from the partial branch metric to obtain the branch metric for encoder 01 outputs. The 11 branch metric is saved in register x1. The negated path metric is placed in b, and the 11 branch metric is move to y0.

The final subl instruction shift left and subtracts the negated path metric from the 01 branch to get the 10 branch metric in b. The 00 branch metric is copied to x0.

After all these moves, we find that register x1:x0 contains branches 11:00, register y1:y0 contains branches 00:11, and a1:b1 has branches 01:10. Note that we can access accumulators as ab or ba to get branches 01:10 or 10:01. All we have to do now is to place the branch metric pairs in memory to correspond with their states for the Viterbi butterfly loop.

Unlike the case in **Section 3**, our branch metric generation allows us to use a simple memory placement scheme. We begin with the memory location for state 0, and increment through the states until the r2 address pointer rolls over to 0 after location 15 (r2 is set to be a modulo 16 address pointer). Because we have all possible needed branch metric pairs in some register, we just code each move with the branch metric pair for that state.

4.3 STARTING FROM 0: THE PRE ACS MACRO

For data that is coded in packets, it is not unusual for the encoder to start in a known state. The easiest case is when the encoder begins each data packet starting in state zero. This is what is assumed for this example. For this case, the beginning trellis is shown in **Figure 4-1**. Notably, the number of states to be updated changes from 2 to 4 to 8, etc., doubling until the full 32 states is reached. Until that time, we have fewer computations to do because the number of states is smaller and because no comparison of competing branches is needed until 32 states of the trellis are being used. We can reduce the computation needed for a packet by treating the first five decoder inputs separately.

The code to do this appears in the PreACS macro in **Example 4-3**. The first two moves save the state address pointers to be restored at the routine's end. Next the branch metrics are loaded into the accumulators b and a. Finally, the path metric/path pair is loaded into x.

The do loop processes each state update. The branches are added to the state to obtain the state updates. The second of these adds also loads in the next path metric/path pair for the next loop iteration. The VSL instructions store the updated path metrics, and the move loads the next branch metrics.

After the loop, we double n5 (used to vary the number of loop iterations), restore r4 and r5 to swap path metric read/write addresses, and reinitialize the branch metric pointer for use by the branch metric update routine.

Example 4-3 Pre-ACS Macro

```
;*****************************;PreACS*************************************
;       FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;               ACS butterfly from assumed 0 state to start, and double
;               the number of states on each invocation until the full
;               trellis is used.  This routine CANNOT be used unless the
;               encoder is starting from an all 0's state.  Note this
;               means it cannot be used if the data is a continuation
;               of a previously processed data stream.  Use the ACS macro
;               instead.  However, for packetised data, or other data that
;               assumes the data starts with the encoder 0 filled, this
;               routine saves cycles, AND only state 0 needs to be initialised
;               before starting the decode.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is doubled each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;               x: path metric, y: path,states ordered assuming
;               bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;               as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C
;        D \
;           \
;            \
;             \
;              \
;       SB      NSB
;
```

Example 4-3 Pre-ACS Macro (Continued)

```
;*************************************************************
;
PreACSmacro
;
;   move#BRY,r2    ;r2 points to branch metrics
    mover5,r3      ;save r5 to init r4 later
    mover4,n3      ;save r4 to init r5 later
    movel:(r2)+,ba ;get first branch metrics
    movel:(r5)+,y  ;load 1st metric/path pair
;
    do  n5,_P_NextStageupdate each state
    add y,a        ;update metric
    add y,b l:(r5)+,y;updt met, ld nxt pair
    vsl a,0,l:(r4)+;end top half
;
    vsl b,1,l:(r4)+;end 2nd half
    movel:(r2)+,ba ;ld br met
_P_NextStage
    moven5,b       ;recall loop count
    asl b   n3,r5  ;mult it by 2
    mover3,r4
    moveb,n5       ;storage for loop count
    move#BRY,r2    ;reinit branch ptr
    endm
```

4.4 COLLAPSING THE STATES

For codes that 0 fill the encoder to create an end of data block, we can reduce the number of cycles needed to finish the Viterbi decoder processing. **Example 4-4** assumes the coder ends a block by 0 filling the encoder. For the decoder, this means that the last 5 inputs to a block are known beforehand to take the upper state. There are two cycle-saving consequences to this. First, we need not update the lower state in the butterflies. Second, each of the last five inputs creates only half the number of states from the previous input. For the last input, we update state 0 only, giving us a single starting point for traceback. The code appears in **Example 4-4**.

Example 4-4 The ACSFlush Macro

```
;**********************ACSFlush Macro*****************************
;       FUNCTION: This routine is very similar to the ACS macro, except that
;               the encoding shift register is now flushing back to 0.
;               This means that only the upper paths are taken, halving
;               the number of states we need to update.  The ACS code is
;               modified so that we don't compute the lower paths. In
;               addition, state storage is modified so that survivor
;               paths are stored in consecutive memory locations. Survivors
;               are even states on the first pass, then every fourth state,etc.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is halved each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,n2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;
;       SA------NSA
;           C  /
;             /
;            /
;          D/
;           /
;          /
;       SB
;
;*****************************************************************
ACSFlush macro
;**     move    #BRY,r2                 ;r2 points to branch metrics
       move    r5,r3                   ;save r5 to init r4 later
       move    r4,n3                   ;save r4 to init r5 later
       move    l:(r2)+n2,y             ;get first branch metrics
       move    l:(r5)+n5,a             ;load 1st metric/path pair
       do      n5,_NextStage           ;update each state
       add     y0,a    l:(r5)-n5,b     ;updt met,load next pair
       add     y1,b    l:(r2)+n2,y     ;updt met,ld nxt br met
       max     a,b     (r5)+           ;sel surv,save met,inc st ptr
       move    l:(r5)+n5,a             ;ld next pair
       vsl     b,0,l:(r4)+             ;end 2nd half
_NextStage
       move    n5,b                    ;recall flush count, loop count
       asr     b       n2,a            ;divide it by 2,prep double branch jump
       move    n3,r5
       move    b,n5                    ;storage for flush count
       asl     a       (r2)-n2         ;reinit branch ptr
       move    r3,r4
       move    a,n2                    ;next pass, skip more branches
       endm
```

We have placed a comment at the beginning, moving the branch metric table address to r2. Since this is done in the FindMetrics routine, we don't actually need to execute this, but it is included as a reminder that r2 needs to be set properly. The remainder of this routine is like the ACS butterfly, except that parts are removed, and the number of iterations varies.

We store the beginning values of r4 and r5 so that we can restore them at the end, swapped with each other. Because the number of loop iterations varies, we don't get the automatic swapping that the ACS macro has. The first branch metrics are fetched, and the updated count value is stored in memory. We then load the first path metric/path pair.

The loop is quite similar to the ACS except we have eliminated unneeded operations. We add a branch metric and load the next path metric/path pair (for the lower path). We then add a branch metric and read the branch metrics for the next loop iteration. Taking the MAX of the accumulators chooses the survivor path metric/path pair, and the path metric read pointer is incremented for the next loop in parallel. The move loads the path metric/path pair for the next loop. The VSL completes the current state update for the upper state.

Note that the state storage pointer is incremented only once, even though we only update the upper state. This means that state 2 will be written at the address *normally* used for state 1, state 4 at the address used for state 2, etc. The next invocation of the ACSFlush will read the data correctly, because the read pointer increment, r5 is halved for each invocation of this macro. During the next invocation, we will only have states 0, 4, 8, etc. These are exactly the states that are possible when the encoder is being 0 filled.

To end this macro, we restore r4 and r5 to the desired values, swapping old and update memories as needed. The loop count is read from n5, halved, and restored, so that we process half as many states on the next loop invocation.

4.5 MAIN: PUTTING THE PIECES BACK TOGETHER

There are a number of changes to the main program that are required for these new macros to operate properly. The code for MAIN appears in the following example.

Example 4-5 Main Program Code Changes

```
;*************************MAIN*************************************************
;
NUMSTATESequ    32
ENCBITS         equ     6               ;most cases=log2(NUMSTATES)+1
NoAcs   equ     NUMSTATES/2
NUMINPUTSequ    168
EVEN            equ     1-(NUMINPUTS/8)%2  ;EVEN SET TO 1/0 IF NUMINPUTS IS
;                                               EVEN/ODD #BYTES
        org     p:$400
VITDEC  move    #NUMSTATES/2-1,m2       ;r2 points to branch metric table
        move    #BRX,r2
        move    #STATE1,r5              ;r5 points to current state metric
        move    #STATE2,r4              ;r4 points to updated state metric
        move    #NUMSTATES*2-1,m4       ;both modulo to flip locations each sym
        move    #NUMSTATES*2-1,m5
        move    #>1,n5                  ;ctr/input metrics spacing for each butterfly
;       move    #NUMSTATES/2,n5         ;input metrics spacing for each butterfly
        move    #PATHOUT,n0             ;n0 points to storage for output paths
        move    #-1,m3                  ;set linear mode, traceback ptr
        move    #>1,n2
;
        move    #INDATA,r1              ;r1 points to input data
;
;**********************PREPARATION LOOP************************
;       This loop iterates by the number of bits used in the
;       encoder to pre load the bit decisions.  Thereafter,
;       the paths are updated and stored off in bytes.  We need
;       the preload bits so that the stored path metrics point
;       correctly to their previous paths for traceback.
;**************************************************************
;
        do      #ENCBITS-1,PRELP        ;preload decisions/trellis start
;
        FindMetrics
        PreACS
;       ACS
PRELP
        move    #NUMSTATES/2,n5         ;n5 now serves as offset between fetch
;                                       states
;
;*****************MAIN LOOP--PROCESS BYTES OF DATA*****************
        do      #NUMINPUTS/8-2,DATALP;process bytes of output
        do      #8,SYMLP                 ;8 bits per byte
;
        FindMetrics
        ACS
SYMLP
        STOREPATHS#NUMSTATES
DATALP
```

Example 4-5 Main Program Code Changes (Continued)

```
;********POSTPROCESSING,LAST 2 INFO BYTES & FLUSH ENCODER BACK TO 0****
;
; THESE SETUPS SHOULD ACCOMODATE ENCODERS OF 5 TO 8 BITS.
; ENCBITS-1 PREPROCESSED, SO WE HAVE 16-ENCBITS+1 BITS LEFT.
; WE DO ENCODER FLUSHING FOR #ENCBITS-1 BITS. SO WE HAVE
; (16-ENCBITS+1) - ENCBITS+1 DO DO BEFORE FLUSHING.
; NEXT, WE PROCESS THE ENCODER FLUSH BITS, STOPPING TO STORE
; PATHS AS NEEDED. WE STORE AFTER 8 BITS PROCESSED SO THIS MEANS
; STORE PATHS AFTER 16-2*ENCBITS+2 (from nonflush) +8-(16-2*ENCBITS+2)
; (these are the additional flush bits needed before storing paths).
; FINALLY, WE FINISH OUT THE DATA. THE BITS REMAINING ARE:
; 16-ENCBITS+1-(16-2*ENCBITS+2)-(8-(16-2*ENCBITS+2))
; = 8-ENCBITS.
;
; FOR THIS EXAMPLE, WE NONFLUSH PROCESS
; 6 MORE BITS, THEN FLUSH 2, STORE PATH, THEN FLUSH
; THE LAST THREE TO STATE 0.
; LAST 6 NONFLUSH BITS---
;*********************************************************************
        do      #16-(2*ENCBITS)+2,LAST6
        FindMetrics
        ACS
;
LAST6
;
;  ENCODER FLUSH-----
;
        move    #>NUMSTATES/2,n5                ;flush, process 16,8, 4, then 2,
;                                               then 1 state
        do      #8-(16-(2*ENCBITS)+2),FLSH;process 3 of last five bits to get
;                                                next to last byte
        FindMetrics
        ACSFlush
;       ACS
FLSH
        STOREPATHS#NUMSTATES/8
;
;  FLUSH THE LAST THREE BITS------
;
        do      #8-ENCBITS+1,FLSH3              ; do last 3 bits to get last byte
;
        FindMetrics
        ACSFlush
;       ACS
FLSH3
;
        TRACEBACK
;
FINISH  nop
```

To make the changes to the MAIN program code, we begin as we did in **Section 3**, by initializing the address registers. R2 is still used to address branch metrics. Registers r4 and r5 address the states for path metric storage. The differences needed for the initialization code are to set n5 to 1 (needed for tracking the number of states in the PreAcs code), and n2 is set to 1. This is different from the code in **Section 3** for two reasons. First, the branch metric generation is different, and second, using n2 to update register r2 for branch metric storage is no longer required. Instead, we use n2 to change the addresses to read branch metrics for the ACSFlush routine.

Next, we do ENCBITS-1 iterations of input processing to preload the path storage registers. This is different because we use the PreACS macro instead of the ACS macro. Using the PreACS macro means that the code uses the starting encoder 0 state to reduce execution cycles as well as to avoid the need of initializing the path metric storage for all the states.

The next major piece of code is the nested do loops that process bytes of decoder input and store off the resulting paths. This code is identical to the code in **Section 3**, except that we process one less byte of data.

The end processing is more complicated because there are two subprocesses occurring at the same time and their endings are not in phase. One process is the storing of the path data in bytes as the path storage registers fill up. The second process is the ACSFlush processing, which collapses the trellis states back to 0.

For this example, start with 168 decoder inputs. We preprocess five inputs, and then process 19*8=152 more inputs in bytes using the nested loops. We have 11 decoder inputs left to process. We need to do 8 more to get the next byte, but after processing 6 inputs, we have to change from using the ACS macro to using the ACSFlush macro (used to collapse states for the last five inputs).

Accordingly, we process the first 6 of the last 11 decoder inputs using the ACS macro. We then process the next two decoder inputs using the ACSFlush macro. Now we have processed eight more decoder inputs, and can store off the path data using the STOREPATHS macro. We then finish processing the last three decoder inputs using the ACSFlush macro. The final path data is in the collapsed state (as it was in the code in **Section 3**), and is access in the TRACEBACK macro as before. After doing the traceback using the TRACEBACK macro of **Section 3**, the decoder is done at the FINISH label.

SECTION 5

SUMMARY

Summary

5.1 SUMMARY

This application report began with an explanation of the Viterbi algorithm in **Section 2**, introduced a basic Viterbi decoder in **Section 3**, and offered some enhancements to the decoding assembly code in **Section 4**. The explanations in **Section 2** and **Section 3** are detailed enough for the ideas behind the code to be understood. The goal is to make it easy to take and modify this code to produce efficient code for any desired application of the Viterbi algorithm.

We chose a nontrivial industry standard code as our example, and there are some statistics that are worth noting. **Table 5-1** presents some notable statistics for code presented in this note. In the table, we compare the basic Viterbi decoder code of **Section 3** with the basic code in **Section 4**. The basic code requires about 215 clock cycles per decoder input to decoder the data. As an example, this means that at a received data rate of 13,350 per second (the IS-136 rate if all six slots in a frame are voice data to be decoded) 2.87 MIPS are required.

If we use generalized branch metrics, the numbers increase, because the branch metric calculation is more complicated. Note, however, that they lower again as the Pre ACS and ACSFlush routines are included. Once all the enhancements of **Section 4** are included, the computational load drops to 2.81 MIPS, more than 3% below what the rate would be if we just used the generalized branch metrics.

5.2 CONCLUSIONS

In conclusion, note that the relative improvement is dependent on the length of the data packet. Shorter packets show greater improvement. For example, an encoded voice data slot in IS-136 uses a packet size of only 89 decoder inputs instead of our example 168. For packets of this size, the percent change is 4.8%—large enough to make the extra code worth considering.

Table 5-1 Viterbi Decoder Code Statistics

Code Used	Total Clock Cycles	Clock Cycles per Information Bit	MIPS at 13,350 Data Rate	% change
Basic Viterbi	36127	215.04	2.87	—
Viterbi with generalized branch metrics	36668	218.26	3.31	+1.5%
+ACSFlush	36065	214.67	2.87	-0.172%
+Pre ACS	35459	211.07	2.81	-1.85%

Another statistic worth noting is the program memory required by each program. The basic Viterbi decoder code in **Section 3** takes 178 program words, and the code in **Section 4** with all enhancements added takes 311. The large increase is mostly due to the unrolling of some loops at the end, required to phase the ACSFlush and STOREPATHS macros correctly. Notably, the amount of program space required for either routine is quite small, and may not be a large consideration for most systems.

5.3 PROGRAM LISTINGS

Finally, note that there are three appendices to this application report. Each appendix contains a program listing for the program as described on one of the sections of this manual.

- **Appendix A** contains a listing for the complete code discussed in **Section 3**.
- **Appendix B** contains a complete listing of the code discussed in **Section 4**.
- **Appendix C** contains a complete listing of a 24-bit version of the **Section 4** code, suitable for implementation on a DSP56300 DSP operating in 24-bit arithmetic mode.

APPENDIX A

BASIC ALGORITHM PROGRAM LISTING

A.1 VITERBI ALGORITHM PROGRAM LISTING

This appendix contains the complete program listing for a 16-bit implementation of the Viterbi algorithm, as presented in **Section 3** of this manual.

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder

```
        OPT     mex
;***********************VITERBI DECODER***********************************
;  THIS ROUTINE IMPLEMENTS A CONVOLUTIONAL DECODER USING THE VITERBI ALG.
;  IT IS OPTIMIZED FOR SPEED, WHICH MEANS THAT EVERY ALU REGISTER
;  AND ALL OF THE R REGISTERS ARE USED.  A SIGNIFICANT AMOUNT OF
;  THESE CAN BE FREED BY STORING AND REUSING REGISTERS BETWEEN THE
;  FINDMETRICS ROUTINES AND THE ACS,ACSFlush ROUTINES.
;       INPUT is in INDATA real in x imag in y.  OUTPUT begins at y:DECOUT
;*****************BRANCH METRIC MACRO************************************
;       FUNCTION: Input data and generate branch metrics.
;               For this decoder, the metric is a scaled
;               sum or difference of the real and imag inputs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r1 should point to the next input XY data pair
;       OUTPUTS:
;               Branch metrics are stored at BRX in XY memory
;       REGISTERS USED:
;               a,b,x01,y01,r1,r2,n2, r2 unchanged (modulo req'd)
;*************************************************************
FindMetrics macro
        move    l:(r1)+,y               ;grab dec input
        move    #-16,x1                 ;sign for real component, 0 sent.
        mpy     x1,y1,a                 ;a has 0x partial branch
        move    #BRY+3,r2               ;storage for generated branch metrics
        mac     x1,y0,a         a,b     ;a gets 00 branch
        mac     -x1,y0,b                ;b has 01 branch
;

        neg     a       a,x1    a,y:(r2)+n2 ;mv 00 to x1,11 to a, st 00 in location 3
        neg     b       b,x0    b,y:(r2)+n2 ;mv 01 to x0,10 to b, st 01 in location 6
;*************************************************************************
;         AT this point X1 has 00, X0 has 01,
;         A1 has 11, B1 has 10, needed for quick storage in Y memory
;*************************************************************************
        move    x1,y:(r2)+n2            ;store 00 in location 9
        move    x0,y:(r2)+n2            ;store 01 in location 12
        move    b,y:(r2)+n2             ;store 10 in location 15
        move    b,y:(r2)+n2             ;store 10 in location 2
        move    b,y:(r2)+n2             ;store 10 in location 5
        move    b,y:(r2)+n2             ;store 10 in location 8
        move    x0,y:(r2)+n2            ;store 01 in location 11
        move    x1,y:(r2)+n2            ;store 00 in location 14
        move    x0,y:(r2)+n2            ;store 01 in location 1
        move    x1,y:(r2)+n2            ;store 00 in location 4
        move    a,y:(r2)+n2             ;store 11 in location 7
        move    a,y:(r2)+n2             ;store 11 in location 10
        move    a,y:(r2)+n2             ;store 11 in location 13
        move    a,y:(r2)                ;store 11 in location 0. r2-> BRY
        endm
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;
;*******************viterbi add, compare, select butterfly macro***
;       FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;               doing an add,compare,select update for state pairs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should offset addresses by NUMSTATES/2
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,x0,y1,r2,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;               x: path metric, y: path,states ordered assuming
;               bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;               as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C /
;        D \  /
;           \/
;           /\
;        D /  \
;         /    \
;       SB------NSB
;           C
;*****************************************************************************
ACS     macro
;
;       move    #BRY,r2                                 ;r2 points to branch metrics
        move    y:(r2)+,y1                              ;get first branch metric
        move    l:(r5)+n5,a                             ;load 1st metric/path pair
;
        do      #NoOfAcsButt,_P_NextStage               ;update each state
        sub     y1,a    l:(r5)-n5,b                     ;sub pt,br met,get next pair
        add     y1,b                                    ;update metrics
        max     a,b     l:(r5)+n5,a                     ;pick max,reload 1st pair
        vsl     b,0,l:(r4)+                             ;store survivor,end top half
        add     y1,a    l:(r5)-n5,b                     ;add pt,br met,reload next pair
        sub     y1,b    x:(r5)+,x0      y:(r2)+,y1      ;inc st ptr,ld nxt br met
        max     a,b     l:(r5)+n5,a                     ;pick max met,load next pair
        vsl     b,1,l:(r4)+                             ;st survivor,end 2nd half
_P_NextStage
        nop                                             ;needed to separate do loop ends
        endm
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;***********************STORE PARTIAL PATH METRICS MACRO***************
;       FUNCTION: The storage is somewhat twisted.  The stored paths are current
;                 up to the most recent input bit, which is NOT convenient for
;                 traceback.  I process the data as follows:  I pre
;                 loaded the path with 6 bits.  Thereafter, I process 8 path bits
;                 so the path has 14 bits.  The most significant 8, I save.
;                 the remaining bits are the current encoder values for that path.
;                 The 5 lsb's are the current state.
;       INPUTS:
;                 n0 should point to memory where the path bits are to be stored
;                 r5 should point to the latest path metric for state 0
;       OUTPUTS:
;                 Updated paths storage X memory, n0 points to next path storage
;       REGISTERS USED:
;                 a,b,x1,r0,n0,r3,r5
;*************************************************************************
;       Store off path data in bytes to avoid overflow in path reg's
;
STOREPATHS macro LPCNT
;
        move    n0,r3                                   ;n0 stores path data pointer
        move    n0,r0
        move    #>$1f,x1                                ;mask for 5 lsb's(NUMENCBITS of
1's)
        do      LPCNT,_PSTORE1                          ;store paths for each state
        move    y:(r5),b                                ;grab path
        asr     #5,b,a                                  ;align bits 5-12 with a1
;                                                        (the 8 bits beyond enc)
        and     x1,b                                    ;mask off 5 lsb's to return to
path
        move    a1,x:(r3)+                              ;store 8 ms path bits
        move    b1,y:(r5)+                              ;return 5 ls path bits
_PSTORE1
        lua     (R4+NUMSTATES),R5                       ;r5 points to latest states
        lua     (r0+NUMSTATES),n0                       ;update path data pointer
        endm
;
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;***********************TRACEBACK OUTPUT PATH MACRO******************
;       FUNCTION: To output the correct data, we begin at the end.  We take the
;                 output path of the survivor state (0), and place its associated
;                 output path in memory as the last output data byte.  Then we use
;                 bits 3-7 of that data as an offset pointer to the correct traceback
;                 data of the next previous path data memory.  We continue this until
;                 we have traced the data back to the begining.
;       INPUTS:
;                 n0 should point to memory where the path bits are to be stored
;                 r5 should point to the latest path metric for state 0
;       OUTPUTS:
;                 Decoder output data in Y memory
;       REGISTERS USED:
;                 a,b,y,r0,n0,r2,r5
;
;*********************************************************************
;       Store off path data in bytes to avoid overflow in path reg's
;
TRACEBACK macro
;
        move    N0,R0                                   ;path ptr to r0
        move    #0,n0                                   ;prep for traceback
        move    #DECOUT+(NUMINPUTS/8/2),r2              ;point to end to trace data
        move    y:(r5),b                                ;recall last path
        move    #-1,m2                                  ;r2 now linear
        move    b1,x:(r0)                               ;save off last path data
        move    #$513,x0                                ;control word for extract
;
;*****************************BEGIN TRACEBACK*****************************
        IF (EVEN==0)
        move    x:(r0+n0),a                     ;recall last path
        extractu x0,a,b                         ;bits 3-7 of a1 point to next data
        lua     (r0-NUMSTATES),r0               ;dec r0 to next earlier state set
        lsl     #8,a                            ;move to upper byte
        move    b0,n0                           ;load as offset for traceback
        move    a1,y:(r2)-                      ;save off
        ENDIF
;
        do      #NUMINPUTS/8/2,TRCBK            ;once for each byte pair
        move    x:(r0+n0),a                     ;recall last path
        extractu x0,a,b                         ;get ptr to next earlier path
        lua     (r0-NUMSTATES),r0               ;point r0 to next earlier states
        move    b0,n0                           ;save ptr as offset
        move    a1,x1                           ;save out byte in x1
        move    x:(r0+n0),a                     ;do it all again!
        extractu x0,a,b
        lua     (r0-NUMSTATES),r0
        move    b0,n0
        lsl     #8,a                            ;move to upper byte
        or      x1,a                            ;or in last byte to get 16 bit word
        move    a1,y:(r2)-                      ;store result
TRCBK
        endm
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;*************************MAIN*************************************************
NUMSTATESequ    32
ENCBITS         equ     6                       ;most cases=log2(NUMSTATES)+1
NoOfAcsButtequ  NUMSTATES/2
NUMINPUTSequ    168
EVEN            equ     1-(NUMINPUTS/8)%2       ;EVEN SET TO 1/0 IF NUMINPUTS IS
;                                                EVEN/ODD #BYTES
        org     p:$400
VITDEC  move    #NUMSTATES/2-1,m2               ;r2 points to branch metric table
        move    #>3,n2                          ;increment for branch metric storage
        move    #STATE1,r5                      ;r5 points to current state metric
        move    #STATE2,r4                      ;r4 points to updated state metric
        move    #NUMSTATES*2-1,m4               ;both modulo to flip loc each sym
        move    #NUMSTATES*2-1,m5
        move    #NUMSTATES/2,n5                 ;input metrics spacing for each butter-
fly
        move    #PATHOUT,n0                     ;n0 points to storage for output paths
        move    #-1,m3                          ;set linear mode, traceback ptr
        move    #INDATA,r1                      ;r1 points to input data
;***********************PREPARATION LOOP*************************
;       This loop iterates by the number of bits used in the
;       encoder to pre load the bit decisions.  Thereafter,
;       the paths are updated and stored off in bytes.  We need
;       the preload bits so that the stored path metrics point
;       correctly to their previous paths for traceback
;****************************************************************
        do      #ENCBITS-1,PRELP                ;once for each encoder bit
;
        FindMetrics
        ACS
PRELP
;*****************MAIN LOOP--PROCESS BYTES OF DATA*****************
        do      #NUMINPUTS/8-1,DATALP           ;process bytes of output
        do      #8,SYMLP                        ;8 bits per byte
;
        FindMetrics
        ACS
SYMLP
        STOREPATHS#NUMSTATES
DATALP
;********POSTPROCESSING,LAST INFO BYTE *********************************
;
; ENCBITS-1 PREPROCESSED, SO WE HAVE 8-ENCBITS+1 BITS LEFT.
; FOR THIS EXAMPLE, WE HAVE 3 INPUTS LEFT TO PROCESS
;*******************************************************************
;  THE LAST THREE BITS------
        do      #8-ENCBITS+1,FLSH1              ;process last 3 bits to get last byte
        FindMetrics
        ACS
FLSH1
;*******Traceback the path data to obtain the decoder output**********
        TRACEBACK
FIN     nop
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;**********************MEMORY ORGANIZATION*****************************
;       MOST OF THE MEMORY LOCATION IS IMPORTANT--THE FIRST TWO
;       LABELS OF THE X AND Y DATA MEMORY ARE PAIRED AND MUST
;       BE CO LOCATED (AT THE SAME ADDRESSES).  IN ADDITION, STATE1
;       AND STATE2 MUST BE LOCATED ON A 0 MOD 2*NUMSTATES BOUNDARY,
;       X and Y memory for input data must be paired --GOOD LUCK.....
;*************************************************************************
;*****************************************************************
;
        org     x:$0
STATE1  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
STATE2  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
;
PATHOUT DS      NUMSTATES*(NUMINPUTS/8+1)
INDATA  ;THIS DATA ENCODES
$1234,$5678,$9abc,$4973,$7925,$3491,$ad43,$ff21,$7ebb,$0100,$20
        DC      $a000,$a000,$a000,$6000,$6000,$a000,$a000,$6000
        DC      $6000,$6000,$6000,$6000,$6000,$a000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$a000,$6000,$a000,$6000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$6000,$a000
        DC      $6000,$a000,$6000,$a000,$6000,$6000,$6000,$6000
        DC      $a000,$a000,$6000,$6000,$a000,$a000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$6000,$a000,$6000,$a000
        DC      $a000,$a000,$6000,$a000,$a000,$a000,$a000,$6000
        DC      $6000,$6000,$6000,$a000,$6000,$6000,$6000,$6000
        DC      $6000,$6000,$a000,$a000,$a000,$a000,$6000,$6000
        DC      $a000,$a000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $6000,$a000,$6000,$a000,$6000,$6000,$a000,$a000
        DC      $6000,$6000,$6000,$a000,$a000,$6000,$a000,$6000
        DC      $6000,$6000,$a000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$a000,$6000
        DC      $6000,$a000,$6000,$a000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$6000,$6000,$a000,$6000,$a000,$6000
;*************************************************************************
```

Example A-1 Basic 16-Bit Implementation of a Viterbi Decoder (Continued)

```
;************************************************************************
;
        org     y:$0
PATH1   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
PATH2   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
BRY     DSM     NUMSTATES/2
DECOUT  DS      NUMINPUTS/8
        org     y:@cvs(y,INDATA)
YDATA   DC      $a000,$a000,$a000,$6000,$a000,$6000,$a000,$6000
        DC      $a000,$6000,$a000,$a000,$6000,$6000,$a000,$6000
        DC      $a000,$a000,$6000,$a000,$6000,$6000,$6000,$a000
        DC      $6000,$6000,$6000,$6000,$6000,$6000,$a000,$6000
        DC      $6000,$6000,$6000,$a000,$a000,$a000,$6000,$a000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$6000,$a000
        DC      $6000,$6000,$6000,$6000,$a000,$6000,$a000,$a000
        DC      $6000,$6000,$a000,$6000,$6000,$6000,$a000,$6000
        DC      $a000,$6000,$6000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$a000,$6000,$a000,$a000,$6000,$6000
        DC      $6000,$a000,$6000,$a000,$a000,$6000,$a000,$6000
        DC      $6000,$6000,$a000,$a000,$6000,$a000,$6000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$6000,$6000,$6000,$6000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$6000,$a000
        DC      $a000,$a000,$a000,$6000,$a000,$a000,$6000,$a000
        DC      $6000,$6000,$a000,$6000,$6000,$6000,$a000,$a000
        DC      $6000,$6000,$a000,$a000,$6000,$a000,$a000,$6000
        DC      $a000,$6000,$6000,$6000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$6000,$a000,$6000,$6000,$6000,$6000
        end
```

APPENDIX B

EXTENDED ALGORITHM PROGRAM LISTING

B.1 16-BIT ENHANCED VITERBI DECODER PROGRAM LISTING

This appendix contains the complete 16-bit program listing for **Section 4**. The code in this section introduces several modifications to the code discussed in **Section 3**. Generalized branch metrics are allowed, as well as the optimization of the code that reduces the computations when data occurs in blocks.

Example B-1 Extended Algorithm Program Listing

```
        OPT     mex
;****************56600**VITERBI DECODER***********************************
; THIS ROUTINE IMPLEMENTS A CONVOLUTIONAL DECODER USING THE VITERBI ALG. IT IS
; OPTIMIZED FOR SPEED, WHICH MEANS THAT EVERY ALU REGISTER AND ALL OF THE R REGISTERS
; ARE USED.  A SIGNIFICANT AMOUNT OF THESE CAN BE FREED BY STORING AND REUSING
; REGISTERS BETWEEN THE FINDMETRICS ROUTINES AND THE ACS,ACSFlush ROUTINES.
; INPUT is at INDATA: real in x imag in y.  OUTPUT begins at y:DECOUT
; GLOBAL REGISTER USE: a,b,x,y,r012345,n025,m245(are set to a modulo mode)
;*****************BRANCH METRIC MACRO**************************************
;       FUNCTION: Input data and generate branch metrics.  This function
;                 subtracts the path metric for state 0 from all branch metrics
;                 to provide autonormaliztion. For this decoder, the metric is a scaled
;                 sum or difference of the real and imag inputs.
;       INPUTS:
;                 r2 should point to the beginning of the branch metric table
;                 r5 should point to the latest path metric for state 0
;                 r1 should point to the next input XY data pair
;       OUTPUTS:
;                 Branch metrics are stored at BRX in XY memory
;       REGISTERS USED:
;                 a,b,x01,y01,r1,r2,r5, r5 unchanged,r2 unchanged (modulo req'd)
;*****************FINDMETRICS MACRO*****************************************
FindMetrics macro
        move    x:(r5),a                ;st. metric scaling, read last st. 0
        neg     a       l:(r1)+,y ;negate metric|grab dec input
        move    #-16,x1                 ;sign for real component, upper br.
        mac     x1,y1,a         a,b     ;comp 0x partial br. path mt. to
        move    y1,x0                   ;mv real input to x0
        mac     x1,y0,a         a,y1    ;a gets 00 br. y1 gets 0x partial br
        subl    a,b             b,x0    ;a has 00 br, b has 11 br, save path
        tfr     y1,a            a,y1    ;swap 0x and 00 to compute 01 branch
        mac     -x1,y0,a        b,x1    ;a gets 01 br
        tfr     x0,b            b,y0    ;swap path metric,11 branch to y0
        subl    a,b             y1,x0   ;b gets 10 br, x0 has copy of 00 br.
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;*************************************************************************
;       AT this point, X is configured with br 11|00, y with 00|11, and AB
;       with 01|10, BA with 10|01 all needed for quick storage in XY memory
;*************************************************************************
        move            x,l:(r2)+               ;store 11 in location 0.
        move            ab,l:(r2)+              ;store 01 in location 1
        move            ba,l:(r2)+              ;store 10 in location 2
        move            y,l:(r2)+               ;store 00 in location 3
        move            y,l:(r2)+               ;store 00 in location 4
        move            ba,l:(r2)+              ;store 10 in location 5
        move            ab,l:(r2)+              ;store 01 in location 6
        move            x,l:(r2)+               ;store 11 in location 7
        move            ba,l:(r2)+              ;store 10 in location 8
        move            y,l:(r2)+               ;store 00 in location 9
        move            x,l:(r2)+               ;store 11 in location 10
        move            ab,l:(r2)+              ;store 01 in location 11
        move            ab,l:(r2)+              ;store 01 in location 12
        move            x,l:(r2)+               ;store 11 in location 13
        move            y,l:(r2)+               ;store 00 in location 14
        move            ba,l:(r2)+              ;store 10 in location 15 r2-> BRX
        endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;*******************viterbi add, compare, select butterfly macro***
;               FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;                       doing an add,compare,select update for state pairs.
;       INPUTS:
;                       r2 should point to the beginning of the branch metric table
;                       r5 should point to the latest path metric for state 0
;                       r4 should point to the storage location for updated state 0
;                       n5 should offset addresses by NUMSTATES/2
;       OUTPUTS:
;                       Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;                       a,b,y01,r2,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;                       x: path metric, y: path,states ordered assuming
;                       bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;                       as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C /
;        D \  /
;           \/
;           /\
;        D /  \
;         /    \
;       SB------NSB
;           C
;*****************************************************************
;
ACS     macro
;
;       move    #BRY,r2                 ;r2 points to branch metrics
        move    l:(r2)+,y               ;get first branch metric
        move    l:(r5)+n5,a             ;load 1st metric/path pair
;
        do      #NoAcs,_P_NextStage     ;update each state
        add     y0,a    l:(r5)-n5,b     ;sum pt,br met,get next pair
        add     y1,b                    ;update metric
        max     a,b     l:(r5)+n5,a     ;pick max, reload 1st pair
        vsl     b,0,l:(r4)+             ;save surivior, end top half
        add     y1,a    l:(r5)-n5,b     ;sum pt,br,reload next pair
        add     y0,b    (r5)+           ;sum again,inc pt mt read ptr
        max     a,b     l:(r5)+n5,a     ;pick max, load next pair
        move    l:(r2)+,y               ;load next branch metrics
        vsl     b,1,l:(r4)+             ;write survivor,end 2nd half
_P_NextStage
        nop                             ;needed to separate do loop ends
        endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;*****************************;PreACS*************************************
;       FUNCTION: Update path metrics/paths for the Viterbi algorithm by ACS butterfly
;       from assumed 0 state to start, and double the number of states on each invocation
;       until the full trellis is used.  This routine CANNOT be used unless the encoder is
;       starting from an all 0's state.  Note this means it cannot be used if the data is
;       a continuation of a previously processed data stream.  Use the ACS macro instead. ;
However, for packetised data, or other data that assumes the data starts with the
;       encoder 0 filled, this routine saves cycles, AND only state 0 needs to be
;       initialised before starting the decode.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is doubled each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;               x: path metric, y: path,states ordered assuming
;               bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;               as x:C, y:D, CD,CD,CD, etc.
;       SA------NSA
;         \  C
;        D \
;           \
;            \
;             \
;              \
;       SB      NSB
;*****************************************************************
PreACS  macro
;       move    #BRY,r2                 ;r2 points to branch metrics
        move    r5,r3                   ;save r5 to init r4 later
        move    r4,n3                   ;save r4 to init r5 later
        move    l:(r2)+,ba              ;get first branch metrics
        move    l:(r5)+,y               ;load 1st metric/path pair
        do      n5,_P_NextStage         update each state
        add     y,a                     ;update metric
        add     y,b     l:(r5)+,y       ;updt met, ld nxt pair
        vsl     a,0,l:(r4)+             ;end top half
        vsl     b,1,l:(r4)+             ;end 2nd half
        move    l:(r2)+,ba              ;ld br met
_P_NextStage
        move    n5,b                    ;recall loop count
        asl     b       n3,r5           ;mult it by 2
        move    r3,r4
        move    b,n5                    ;storage for loop count
        move    #BRY,r2                 ;reinit branch ptr
        endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;***********************ACSFlush Macro*****************************
;       FUNCTION: This routine is very similar to the ACS macro, except that
;               the encoding shift register is now flushing back to 0.
;               This means that only the upper paths are taken, halving
;               the number of states we need to update.  The ACS code is
;               modified so that we don't compute the lower paths. In
;               addition, state storage is modified so that survivor
;               paths are stored in consecutive memory locations.
;               Survivors are even states on the first pass, then
;               every fourth state, etc.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is halved each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,n2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;
;       SA------NSA
;           C  /
;             /
;            /
;          D/
;           /
;          /
;       SB
;******************************************************************
ACSFlush macro
;**     move    #BRY,r2                 ;r2 points to branch metrics
       move    r5,r3                   ;save r5 to init r4 later
       move    r4,n3                   ;save r4 to init r5 later
       move    l:(r2)+n2,y             ;get first branch metrics
       move    l:(r5)+n5,a             ;load 1st metric/path pair
;
       do      n5,_NextStage           ;update each state
       add     y0,a    l:(r5)-n5,b     ;updt met,load next pair
       add     y1,b    l:(r2)+n2,y     ;updt met,ld nxt br met
       max     a,b     (r5)+           ;sel surv,save met,inc st ptr
       move    l:(r5)+n5,a             ;ld next pair
       vsl     b,0,l:(r4)+             ;end 2nd half
_NextStage
       move    n5,b                    ;recall flush count, loop count
       asr     b       n2,a            ;divide it by 2,prep double branch jump
       move    n3,r5
       move    b,n5                    ;storage for flush count
       asl     a       (r2)-n2         ;reinit branch ptr
       move    r3,r4
       move    a,n2                    ;next pass, skip more branches
       endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;***********************STORE PARTIAL PATH METRICS MACRO***************
;       FUNCTION: The storage is somewhat twisted.  The stored paths are
;                 current up to the most recent input bit, which is NOT
;                 convenient for traceback.  As a result, I process the data as
;                 follows:  I preloaded the path with 6 bits.  Thereafter, I
;                 process 8 path bits so the path has 14 bits.  The most
;                 significant 8, I save. The remaining bits are the current
;                 encoder values for that path. The 5 lsb's are the current
;                 state.
;       INPUTS:
;                 n0 should point to memory where the path bits are to be stored
;                 r5 should point to the latest path metric for state 0
;       OUTPUTS:
;                 Updated paths storage X memory, n0 points to next path storage
;       REGISTERS USED:
;                 a,b,y1,r0,n0,r3,r5
;*********************************************************************
;
;       Store off path data in bytes to avoid overflow in path reg's
;
STOREPATHS macro LPCNT
;
        move    n0,r3                   ;n0 stores path data pointer
        move    n0,r0
        move    #>$1f,y1                ;mask for 5 lsb's(NUMENCBITS-1 of 1's)
        do      LPCNT,_PSTORE1          ;store paths for each state
        move    y:(r5),b                ;grab path
        asr     #5,b,a                  ;align bits 5-12 w/ a1 (8 bits beyond enc)
        and     y1,b                    ;mask off 5 lsb's to return to path
        move    a1,x:(r3)+              ;store 8 ms path bits
        move    b1,y:(r5)+              ;return 5 ls path bits
_PSTORE1
        lua     (R4+NUMSTATES),R5       ;r5 points to latest states
                lua     (r0+NUMSTATES),n0       ;update path data pointer
        endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;***********************TRACEBACK OUTPUT PATH MACRO******************
;       FUNCTION: To output the correct data, we begin at the end.  We take
;               the output path of the survivor state (0), and place its
;               associated output path in memory as the last output data
;               byte.  Then we use bits 3-7 of that data as an offset pointer
;               to the correct traceback data of the next previous path data
;               memory.  We continue this until we have traced the data back
;               to the begining.
;       INPUTS:
;               n0 should point to memory where the path bits are to be stored
;               r5 should point to the latest path metric for state 0
;       OUTPUTS:
;               Decoder output data in Y memory
;       REGISTERS USED:
;               a,b,y,r0,n0,r2,r5
;
;*********************************************************************
;       Store off path data in bytes to avoid overflow in path reg's
TRACEBACK macro
;
        move    N0,R0                           ;path ptr to r0
        move    #0,n0                           ;prep for traceback
        move    #DECOUT+(NUMINPUTS/8/2),r2 ;point to end of output buffer
        move    y:(r5),b                        ;recall last path
        move    #-1,m2                          ;r2 now linear
        move    b1,x:(r0)                       ;save off last path data
        move    #$513,y0                        ;control word for extract
;
;*****************BEGIN TRACEBACK*****************************
        IF (EVEN==0)
        move    x:(r0+n0),a                     ;recall last path
        extractu y0,a,b                         ;bits 3-7 of a1 point to next data
        lua     (r0-NUMSTATES),r0               ;dec r0 to next earlier path set
        lsl     #8,a                            ;move to upper byte
        move    b0,n0                           ;load as offset for traceback
        move    a1,y:(r2)-                      ;save off
        ENDIF
;
        do      #NUMINPUTS/8/2,TRCBK            ;once for each byte pair
        move    x:(r0+n0),a                     ;recall last path
        extractu y0,a,b                         ;get ptr to next earlier path
        lua     (r0-NUMSTATES),r0               ;point r0 to next earlier states
        move    b0,n0                           ;save ptr as offset
        move    a1,y1                           ;save out byte in x1
        move    x:(r0+n0),a                     ;do it all again!
        extractu y0,a,b
        lua     (r0-NUMSTATES),r0
        move    b0,n0
        lsl     #8,a                            ;move to upper byte
        or      y1,a                    ;or in last byte to get 16 bit word
        move    a1,y:(r2)-              ;store result
TRCBK
        endm
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;*************************MAIN*************************************************
;
NUMSTATESequ    32
ENCBITS         equ     6               ;most cases=log2(NUMSTATES)+1
NoAcs   equ     NUMSTATES/2
NUMINPUTSequ    168
EVEN            equ     1-(NUMINPUTS/8)%2  ;EVEN SET TO 1/0 IF NUMINPUTS IS
;                                               EVEN/ODD #BYTES
        org     p:$400
VITDEC  move    #NUMSTATES/2-1,m2       ;r2 points to branch metric table
        move    #BRX,r2
        move    #STATE1,r5              ;r5 points to current state metric
        move    #STATE2,r4              ;r4 points to updated state metric
        move    #NUMSTATES*2-1,m4       ;both modulo to flip locations each sym
        move    #NUMSTATES*2-1,m5
        move    #>1,n5                  ;ctr/input metrics spacing for each butterfly
;       move    #NUMSTATES/2,n5         ;input metrics spacing for each butterfly
        move    #PATHOUT,n0             ;n0 points to storage for output paths
        move    #-1,m3                  ;set linear mode, traceback ptr
        move    #>1,n2
;
        move    #INDATA,r1              ;r1 points to input data
;
;***********************PREPARATION LOOP*************************
;       This loop iterates by the number of bits used in the
;       encoder to pre load the bit decisions.  Thereafter,
;       the paths are updated and stored off in bytes.  We need
;       the preload bits so that the stored path metrics point
;       correctly to their previous paths for traceback.
;****************************************************************
;
        do      #ENCBITS-1,PRELP        ;preload decisions/trellis start
;
        FindMetrics
        PreACS
;       ACS
PRELP
        move    #NUMSTATES/2,n5         ;n5 now serves as offset between fetch
;                                       states
;
;*****************MAIN LOOP--PROCESS BYTES OF DATA*****************
        do      #NUMINPUTS/8-2,DATALP;process bytes of output
        do      #8,SYMLP                 ;8 bits per byte
;
        FindMetrics
        ACS
SYMLP
        STOREPATHS#NUMSTATES
DATALP
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;********POSTPROCESSING,LAST 2 INFO BYTES & FLUSH ENCODER BACK TO 0****
;
; THESE SETUPS SHOULD ACCOMODATE ENCODERS OF 5 TO 8 BITS.
; ENCBITS-1 PREPROCESSED, SO WE HAVE 16-ENCBITS+1 BITS LEFT.
; WE DO ENCODER FLUSHING FOR #ENCBITS-1 BITS. SO WE HAVE
; (16-ENCBITS+1) - ENCBITS+1 DO DO BEFORE FLUSHING.
; NEXT, WE PROCESS THE ENCODER FLUSH BITS, STOPPING TO STORE
; PATHS AS NEEDED. WE STORE AFTER 8 BITS PROCESSED SO THIS MEANS
; STORE PATHS AFTER 16-2*ENCBITS+2 (from nonflush) +8-(16-2*ENCBITS+2)
; (these are the additional flush bits needed before storing paths).
; FINALLY, WE FINISH OUT THE DATA. THE BITS REMAINING ARE:
; 16-ENCBITS+1-(16-2*ENCBITS+2)-(8-(16-2*ENCBITS+2))
; = 8-ENCBITS.
;
; FOR THIS EXAMPLE, WE NONFLUSH PROCESS
; 6 MORE BITS, THEN FLUSH 2, STORE PATH, THEN FLUSH
; THE LAST THREE TO STATE 0.
; LAST 6 NONFLUSH BITS---
;*********************************************************************
        do      #16-(2*ENCBITS)+2,LAST6
        FindMetrics
        ACS
;
LAST6
;
;  ENCODER FLUSH-----
;
        move    #>NUMSTATES/2,n5                ;flush, process 16,8, 4, then 2,
;                                               then 1 state
        do      #8-(16-(2*ENCBITS)+2),FLSH;process 3 of last five bits to get
;                                                next to last byte
        FindMetrics
        ACSFlush
;       ACS
FLSH
        STOREPATHS#NUMSTATES/8
;
;  FLUSH THE LAST THREE BITS------
;
        do      #8-ENCBITS+1,FLSH3              ; do last 3 bits to get last byte
;
        FindMetrics
        ACSFlush
;       ACS
FLSH3
;
        TRACEBACK
;
FINISH  nop
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
;**********************MEMORY ORGANIZATION*****************************
;       MOST OF THE MEMORY LOCATION IS IMPORTANT--THE FIRST TWO
;       LABELS OF THE X AND Y DATA MEMORY ARE PAIRED AND MUST
;       BE CO LOCATED (AT THE SAME ADDRESSES).  IN ADDITION, STATE1
;       AND STATE2 MUST BE LOCATED ON A 0 MOD 2*NUMSTATES BOUNDARY,
;       X and Y memory for input data must be paired --GOOD LUCK.....
;**********************************************************************
;
        org     x:$0
STATE1  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
STATE2  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
BRX     DSM     NUMSTATES/2
;
PATHOUT DS      NUMSTATES*(NUMINPUTS/8+1)
INDATA  ;THIS DATA ENCODES
$1234,$5678,$9abc,$4973,$7925,$3491,$ad43,$ff21,$7ebb,$0100,$20
        DC      $a000,$a000,$a000,$6000,$6000,$a000,$a000,$6000
        DC      $6000,$6000,$6000,$6000,$6000,$a000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$a000,$6000,$a000,$6000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$6000,$a000
        DC      $6000,$a000,$6000,$a000,$6000,$6000,$6000,$6000
        DC      $a000,$a000,$6000,$6000,$a000,$a000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$6000,$a000,$6000,$a000
        DC      $a000,$a000,$6000,$a000,$a000,$a000,$a000,$6000
        DC      $6000,$6000,$6000,$a000,$6000,$6000,$6000,$6000
        DC      $6000,$6000,$a000,$a000,$a000,$a000,$6000,$6000
        DC      $a000,$a000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$a000,$a000,$a000
        DC      $6000,$a000,$6000,$a000,$6000,$6000,$a000,$a000
        DC      $6000,$6000,$6000,$a000,$a000,$6000,$a000,$6000
        DC      $6000,$6000,$a000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$a000,$6000
        DC      $6000,$a000,$6000,$a000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$6000,$6000,$a000,$6000,$a000,$6000
;
        org     y:$0
PATH1   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
PATH2   DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
```

Example B-1 Extended Algorithm Program Listing (Continued)

```
BRY     DSM     NUMSTATES/2
DECOUT  DS      NUMINPUTS/8
        org     y:@cvs(y,INDATA)
YDATA   DC      $a000,$a000,$a000,$6000,$a000,$6000,$a000,$6000
        DC      $a000,$6000,$a000,$a000,$6000,$6000,$a000,$6000
        DC      $a000,$a000,$6000,$a000,$6000,$6000,$6000,$a000
        DC      $6000,$6000,$6000,$6000,$6000,$6000,$a000,$6000
        DC      $6000,$6000,$6000,$a000,$a000,$a000,$6000,$a000
        DC      $a000,$a000,$6000,$6000,$6000,$a000,$6000,$a000
        DC      $6000,$6000,$6000,$6000,$a000,$6000,$a000,$a000
        DC      $6000,$6000,$a000,$6000,$6000,$6000,$a000,$6000
        DC      $a000,$6000,$6000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$a000,$6000,$a000,$a000,$6000,$6000
        DC      $6000,$a000,$6000,$a000,$a000,$6000,$a000,$6000
        DC      $6000,$6000,$a000,$a000,$6000,$a000,$6000,$a000
        DC      $a000,$6000,$6000,$a000,$a000,$6000,$a000,$a000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$a000,$6000
        DC      $a000,$6000,$a000,$6000,$6000,$6000,$6000,$6000
        DC      $a000,$a000,$a000,$a000,$a000,$6000,$6000,$a000
        DC      $a000,$a000,$a000,$6000,$a000,$a000,$6000,$a000
        DC      $6000,$6000,$a000,$6000,$6000,$6000,$a000,$a000
        DC      $6000,$6000,$a000,$a000,$6000,$a000,$a000,$6000
        DC      $a000,$6000,$6000,$6000,$6000,$a000,$a000,$a000
        DC      $a000,$a000,$6000,$a000,$6000,$6000,$6000,$6000
        end
```

APPENDIX C

24-BIT ALGORITHM PROGRAM LISTING

C.1 24-BIT ENHANCED VITERBI DECODER PROGRAM LISTING

This section contains a second complete program listing for **Section 4**. This program is nearly identical to the program listing in **Appendix B**. The difference is that the data is 24-bit data, meant to run on a 56300 family digital signal processor in 24-bit arithmetic mode. In addition, the path data is stored and recovered in 16-bit words instead of 8-bit bytes. The longer register allow us to reduce the memory and some complexity in the traceback.

Example C-1 24-bit Algorithm Program Listing

```
        OPT     mex
;***************56600**24 bit VITERBI DECODER*****************************
;  THIS ROUTINE IMPLEMENTS A CONVOLUTIONAL DECODER USING THE VITERBI ALG.
;  IT IS OPTIMIZED FOR SPEED, WHICH MEANS THAT EVERY ALU REGISTER
;  AND ALL OF THE R REGISTERS ARE USED.  A SIGNIFICANT AMOUNT OF
;  THESE CAN BE FREED BY STORING AND REUSING REGISTERS BETWEEN THE
;  FINDMETRICS ROUTINES AND THE ACS,ACSFlush ROUTINES.
;  INPUT is at INDATA: real in x imag in y.  OUTPUT begins at y:DECOUT
;  GLOBAL REGISTER USE: a,b,x,y,r012345,n025,m245(are set to a modulo mode)
;***********************************************************************
;
;*****************BRANCH METRIC MACRO***********************************
;       FUNCTION: Input data and generate branch metrics.  This function
;       subtracts the path metric for state 0 from all branch metrics
;       to provide autonormaliztion. For this decoder, the metric is a scaled
;       sum or difference of the real and imag inputs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r1 should point to the next input XY data pair
;       OUTPUTS:
;               Branch metrics are stored at BRX in XY memory
;       REGISTERS USED:
;               a,b,x01,y01,r1,r2,r5, r5 unchanged,r2 unchanged (modulo req'd)
;*************************************************************
;
FindMetricsmacro
        move            x:(r5),a                ;st. metric scaling, read last st. 0
        neg             a               l:(r1)+,y ;negate metric|grab dec input
        move            #-16,x1                 ;sign for real component, upper br.
        mac             x1,y1,a         a,b
        move            y1,x0                   ;cp metric to b|mv real in to x0
        mac             x1,y0,a         a,y1    ;y1 gets 0x partial branch
        subl            a,b             b,x0    ;a has 00 br., b has 11 branch, save st.
        tfr             y1,a            a,y1    ;swap 0x and 00 to compute 01 branch
        mac             -x1,y0,a        b,x1    ;a gets 01 br,x1 gets cp of 11 br.
        tfr             x0,b            b,y0    ;swap st metric,11 branch to y0
        subl            a,b             y1,x0   ;b gets 10 br, x0 has copy of 00 br.
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;*************************************************************************
;         AT this point, X is configured with br 11|00, y with 00|11, and
;         AB with 01|10, BA with 10|01 all needed for quick storage in XY memory
;*************************************************************************

        move            x,l:(r2)+               ;store 11 in location 0.
        move            ab,l:(r2)+              ;store 01 in location 1
        move            ba,l:(r2)+              ;store 10 in location 2
        move            y,l:(r2)+               ;store 00 in location 3
        move            y,l:(r2)+               ;store 00 in location 4
        move            ba,l:(r2)+              ;store 10 in location 5
        move            ab,l:(r2)+              ;store 01 in location 6
        move            x,l:(r2)+               ;store 11 in location 7
        move            ba,l:(r2)+              ;store 10 in location 8
        move            y,l:(r2)+               ;store 00 in location 9
        move            x,l:(r2)+               ;store 11 in location 10
        move            ab,l:(r2)+              ;store 01 in location 11
        move            ab,l:(r2)+              ;store 01 in location 12
        move            x,l:(r2)+               ;store 11 in location 13
        move            y,l:(r2)+               ;store 00 in location 14
        move            ba,l:(r2)+              ;store 10 in location 15 r2-> BRX
        endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;*******************viterbi add, compare, select butterfly macro***
;       FUNCTION: Update path metrics/paths for the Viterbi algorithm by
;               doing an add,compare,select update for state pairs.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should offset addresses by NUMSTATES/2
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as
;               x: path metric, y: path,states ordered assuming
;               bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged
;               as x:C, y:D, CD,CD,CD, etc.
;
;       SA------NSA
;         \  C /
;        D \  /
;           \/
;           /\
;        D /  \
;         /    \
;       SB------NSB
;           C
;*****************************************************************
;
ACS     macro
;
;       move    #BRY,r2                 ;r2 points to branch metrics
        move    l:(r2)+,y               ;get first branch metric
        move    l:(r5)+n5,a             ;load 1st metric/path pair
;
        do      #NoAcs,_P_NextStage     ;update each state
        add     y0,a    l:(r5)-n5,b     ;get next pair
        add     y1,b                    ;update metrics
        max     a,b     l:(r5)+n5,a     ;reload 1st pair
        vsl     b,0,l:(r4)+             ;end top half
        add     y1,a    l:(r5)-n5,b     ;reload next pair
        add     y0,b    (r5)+           ;inc st ptr
        max     a,b     l:(r5)+n5,a     ;load next pair
        move    l:(r2)+,y               ;load next branch metrics
        vsl     b,1,l:(r4)+             ;end 2nd half
_P_NextStage
        nop                             ;needed to separate do loop ends
        endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;*****************************;PreACS************************************
;FUNCTION: Update path metrics/paths for the Viterbi algorithm by the ACS
;       butterfly from assumed 0 state to start, and double the number of states
;       on each invocation until the full trellis is used.  This routine CANNOT
;       be used unless the encoder is starting from an all 0's state.  Note this
;       means it cannot be used if the data is a continuation of a previously
;       processed data stream.  Use the ACS macro instead. However, for packetised
;       data, or other data that assumes the data starts with the encoder 0 filled,
;       this routine saves cycles, AND only state 0 needs to be initialised
;       before starting the decode.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is doubled each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;       Registers:
;            r5, pointer to the path metric/path table, arranged as x: path metric,
;                y: path,states ordered assuming bits shift right to left.
;            r4, pointer to the output path metric/path table
;            r0, pointer to the branch metric table, arranged as x:C, y:D, CD,CD, etc.
;
;       SA------NSA
;         \  C
;        D \
;           \
;            \
;             \
;              \
;       SB      NSB
;*****************************************************************
PreACS  macro
;       move    #BRY,r2                 ;r2 points to branch metrics
        move    r5,r3                   ;save r5 to init r4 later
        move    r4,n3                   ;save r4 to init r5 later
        move    l:(r2)+,ba              ;get first branch metrics
        move    l:(r5)+,y                       ;load 1st metric/path pair
        do      n5,_P_NextStage                 ;update each state
        add     y,a                             ;update metric
        add     y,b     l:(r5)+,y               ;updt met, ld nxt pair
        vsl     a,0,l:(r4)+                     ;end top half
        vsl     b,1,l:(r4)+                     ;end 2nd half
        move    l:(r2)+,ba                      ;ld br met
_P_NextStage
        move    n5,b                            ;recall loop count
        asl     b       n3,r5                   ;mult it by 2
        move    r3,r4
        move    b,n5                            ;storage for loop count
        move    #BRY,r2                         ;reinit branch ptr
        endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;***********************ACSFlush Macro*****************************
;       FUNCTION: This routine is very similar to the ACS macro, except that
;               the encoding shift register is now flushing back to 0.
;               This means that only the upper paths are taken, halving
;               the number of states we need to update.  The ACS code is
;               modified so that we don't compute the lower paths. In
;               addition, state storage is modified so that survivor
;               paths are stored in consecutive memory locations.
;               Survivors are even states on the first pass, then
;               every fourth state, etc.
;       INPUTS:
;               r2 should point to the beginning of the branch metric table
;               r5 should point to the latest path metric for state 0
;               r4 should point to the storage location for updated state 0
;               n5 should be the number of input states/2 to process
;                       n5 is halved each time this macro is invoked
;       OUTPUTS:
;               Updated path metrics/paths stored in XY memory
;       REGISTERS USED:
;               a,b,y01,r2,n2,r3,n3,r4,r5,n5 r2 unchanged (modulo req'd)
;
;       SA------NSA
;           C  /
;             /
;            /
;          D/
;           /
;          /
;       SB
;
;******************************************************************
ACSFlush macro
;**     move    #BRY,r2                 ;r2 points to branch metrics
       move    r5,r3                   ;save r5 to init r4 later
       move    r4,n3                   ;save r4 to init r5 later
       move    l:(r2)+n2,y             ;get first branch metrics
       move    l:(r5)+n5,a             ;load 1st metric/path pair
;
       do      n5,_NextStage           ;update each state
       add     y0,a    l:(r5)-n5,b     ;updt met,load next pair
       add     y1,b    l:(r2)+n2,y     ;updt met,ld nxt br met
       max     a,b     (r5)+           ;sel surv,save met,inc st ptr
       move    l:(r5)+n5,a             ;ld next pair
       vsl     b,0,l:(r4)+             ;end 2nd half
_NextStage
       move    n5,b                    ;recall flush count, loop count
       asr     b       n2,a            ;divide it by 2,prep double branch jump
       move    n3,r5
       move    b,n5                    ;storage for flush count
       asl     a       (r2)-n2         ;reinit branch ptr
       move    r3,r4
       move    a,n2                    ;next pass, skip more branches
       endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;***********************STORE PARTIAL PATH METRICS MACRO***************
;       FUNCTION: The storage is somewhat twisted.  The stored paths are current
;                 up to the most recent input bit, which is NOT convenient for
;                 traceback.  As a result, I process the data as follows:  I pre
;                 loaded the path with 6 bits.  Thereafter, I process 8 path bits
;                 so the path has 14 bits.  The most significant 8, I save.
;                 the remaining bits are the current encoder values for that path.
;                 The 5 lsb's are the current state.
;       INPUTS:
;                 n0 should point to memory where the path bits are to be stored
;                 r5 should point to the latest path metric for state 0
;       OUTPUTS:
;                 Updated paths storage X memory, n0 points to next path storage
;       REGISTERS USED:
;                 a,b,y1,r0,n0,r3,r5
;***********************************************************************
;
;       Store off path data in wordsto avoid overflow in path reg's
;
STOREPATHS macro LPCNT
;
        move    n0,r3                   ;n0 stores path data pointer
        move    n0,r0
        move    #>$1f,y1                ;mask for 1 lsb's(NUMENCBITS-1 of 1's)
        do      LPCNT,_PSTORE1          ;store paths for each state
        move    y:(r5),b                ;grab path
        asr     #5,b,a                  ;align bits5-20 w/ a1 (16 bits beyond enc)
        and     y1,b                    ;mask off 5 lsb's to return to path
        move    a1,x:(r3)+              ;store 16 ms path bits
        move    b1,y:(r5)+              ;return 5 ls path bits
_PSTORE1
        lua     (R4+NUMSTATES),R5       ;r5 points to latest states
        lua     (r0+NUMSTATES),n0       ;update path data pointer
        endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;***********************TRACEBACK OUTPUT PATH MACRO*******************
;       FUNCTION: To output the correct data, we begin at the end.  We take the
;               output path of the survivor state (0), and place its associated
;               output path in memory as the last output data word.  Then we use
;               bits 11-15 of that data as an offset pointer to the correct traceback
;               data of the next previous path data memory.  We continue this until
;               we have traced the data back to the begining.
;       INPUTS:
;               n0 should point to memory where the path bits are to be stored
;               r5 should point to the latest path metric for state 0
;       OUTPUTS:
;               Decoder output data in Y memory
;       REGISTERS USED:
;               a,b,y,r0,n0,r2,r5
;
;*********************************************************************
;
;       Store off path data in words to avoid overflow in path reg's
;
TRACEBACK macro
;
        move    N0,R0                           ;path ptr to r0
        move    #0,n0                           ;prep for traceback
        move    #DECOUT+(NUMINPUTS/8/2),r2 ;point to end of output buffer
        move    y:(r5),b                        ;recall last path
        move    #-1,m2                          ;r2 now linear
        move    b1,x:(r0)                       ;save off last path data
        move    #$5023,y0                       ;control word for extract
;
;*****************BEGIN TRACEBACK*****************************
;
        IF (EVEN==0)
        move    x:(r0+n0),a                     ;recall last path
        extractu y0,a,b                         ;bits 11-15 of a1 point to next data
        lua     (r0-NUMSTATES),r0               ;dec r0 to next earlier state set
        lsl     #8,a                            ;move to upper byte
        move    b0,n0                           ;load as offset for traceback
        move    a1,y:(r2)-                      ;save off
        ENDIF
;
        do      #NUMINPUTS/8/2,TRCBK            ;once for each byte pair
        move    x:(r0+n0),a                     ;recall last path
        extractu y0,a,b                         ;get ptr to next earlier path
        lua     (r0-NUMSTATES),r0               ;point r0 to next earlier states
        move    b0,n0                           ;save ptr as offset
        move    a1,y:(r2)-                      ;store result
TRCBK
        endm
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;*************************MAIN*********************************************
;
NUMSTATESequ    32
ENCBITS         equ     6                       ;most cases=log2(NUMSTATES)+1
NoAcs   equ     NUMSTATES/2
NUMINPUTSequ    168
EVEN            equ     1-(NUMINPUTS/8)%2       ;EVEN SET TO 1/0 IF NUMINPUTS IS
;                                               EVEN/ODD #BYTES
        org     p:$400
VITDEC  move    #NUMSTATES/2-1,m2               ;r2 points to branch metric table
        move    #BRX,r2
        move    #STATE1,r5                      ;r5 points to current state metric
        move    #STATE2,r4                      ;r4 points to updated state metric
        move    #NUMSTATES*2-1,m4               ;modulo to flip locations each sym
        move    #NUMSTATES*2-1,m5
        move    #>1,n5                          ;ctr/input metrics spacing for each
;                                               butterfly
;       move    #NUMSTATES/2,n5                 ;input spacing for each butterfly
        move    #PATHOUT,n0                     ;n0 points to stor for output paths
        move    #-1,m3                          ;set linear mode, traceback ptr
        move    #>1,n2
;
        move    #INDATA,r1                      ;r1 points to input data
;
;***********************PREPARATION LOOP*************************
;       This loop iterates by the number of bits used in the
;       encoder to pre load the bit decisions.  Thereafter,
;       the paths are updated and stored off in bytes.  We need
;       the preload bits so that the stored path metrics point
;       correctly to their previous paths for traceback.
;****************************************************************
;
        do      #ENCBITS-1,PRELP                ;preload decisions/trellis start
;
        FindMetrics
        PreACS
;       ACS
PRELP
        move    #NUMSTATES/2,n5                 ;n5 now serves as offset between
;                                               fetch states
;*****************MAIN LOOP--PROCESS BYTES OF DATA*****************
        do      #NUMINPUTS/16-1,DATALP          ;process bytes of output
        do      #16,SYMLP                       ;16 bits per byte
;
        FindMetrics
        ACS
SYMLP
        STOREPATHS#NUMSTATES
DATALP
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;********POSTPROCESSING, FLUSH ENCODER BACK TO 0****
; ASSUMES that there are an odd number of bytes of data!!!
; This will need redoing if the number of data bytes is even.
; THESE SETUPS SHOULD ACCOMODATE ENCODERS OF 5 TO 8 BITS.
; ENCBITS-1 PREPROCESSED, 152 IN MAIN LOOP DONE
; SO WE HAVE 24-ENCBITS+1 BITS LEFT.
; WE DO ENCODER FLUSHING FOR #ENCBITS-1 BITS. SO WE HAVE
; (24-ENCBITS+1) - ENCBITS+1 DO DO BEFORE FLUSHING.
; NEXT, WE PROCESS THE ENCODER FLUSH BITS, STOPPING TO STORE
; PATHS AS NEEDED. WE STORE AFTER 16 BITS PROCESSED SO THIS MEANS
; STORE PATHS AFTER 24-2*ENCBITS+2 (from nonflush) +16-(24-2*ENCBITS+2)
; (these are the additional flush bits needed before storing paths).
; FINALLY, WE FINISH OUT THE DATA. THE BITS REMAINING ARE:
; 24-ENCBITS+1-(24-2*ENCBITS+2)-(16-(24-2*ENCBITS+2))
; = 8-ENCBITS+1.
;
; FOR THIS EXAMPLE, WE NONFLUSH PROCESS
; 14 MORE BITS, THEN FLUSH 2, STORE PATH, THEN FLUSH
; THE LAST THREE TO STATE 0.
; LAST 14 NONFLUSH BITS---
;*****************************************************************
        do      #24-(2*ENCBITS)+2,LAST14
        FindMetrics
        ACS
;
LAST14
;
;  ENCODER FLUSH-----
;
        move    #>NUMSTATES/2,n5                ;flush, process 16,8, 4, then 2,
;                                               then 1 state
        do      #16-(24-(2*ENCBITS)+2),FLSH;process 2 of last five bits to get
;                                                next to last byte
        FindMetrics
        ACSFlush
;       ACS
FLSH
        STOREPATHS#NUMSTATES/8
;
;  FLUSH THE LAST THREE BITS------
;
        do      #8-ENCBITS+1,FLSH3              ;process last 3 bits to get last
;                                               byte
        FindMetrics
        ACSFlush
;       ACS
FLSH3
;
        TRACEBACK
;
FINISH  nop
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
;**********************MEMORY ORGANIZATION*****************************
;       MOST OF THE MEMORY LOCATION IS IMPORTANT--THE FIRST TWO
;       LABELS OF THE X AND Y DATA MEMORY ARE PAIRED AND MUST
;       BE CO LOCATED (AT THE SAME ADDRESSES).  IN ADDITION, STATE1
;       AND STATE2 MUST BE LOCATED ON A 0 MOD 2*NUMSTATES BOUNDARY,
;       X and Y memory for input data must be paired --GOOD LUCK.....
;**********************************************************************
;
        org     x:$0
STATE1  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
STATE2  DC      $0ff,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
        DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
BRX     DSM     NUMSTATES/2
;
PATHOUT DS      NUMSTATES*(NUMINPUTS/8+1)
INDATA  ;THIS DATA ENCODES
$1234,$5678,$9abc,$4973,$7925,$3491,$ad43,$ff21,$7ebb,$0100,$20
        DC      $a00000,$a00000,$a00000,$600000,$600000,$a00000,$a00000,$600000
        DC      $600000,$600000,$600000,$600000,$600000,$a00000,$a00000,$600000
        DC      $a00000,$600000,$a00000,$600000,$a00000,$600000,$a00000,$600000
        DC      $a00000,$a00000,$600000,$600000,$600000,$a00000,$a00000,$a00000
        DC      $a00000,$a00000,$a00000,$a00000,$a00000,$a00000,$a00000,$a00000
        DC      $a00000,$600000,$600000,$a00000,$a00000,$a00000,$a00000,$a00000
        DC      $a00000,$a00000,$a00000,$a00000,$a00000,$600000,$600000,$a00000
        DC      $600000,$a00000,$600000,$a00000,$600000,$600000,$600000,$600000
        DC      $a00000,$a00000,$600000,$600000,$a00000,$a00000,$a00000,$600000
        DC      $a00000,$600000,$a00000,$600000,$600000,$a00000,$600000,$a00000
        DC      $a00000,$a00000,$600000,$a00000,$a00000,$a00000,$a00000,$600000
        DC      $600000,$600000,$600000,$a00000,$600000,$600000,$600000,$600000
        DC      $600000,$600000,$a00000,$a00000,$a00000,$a00000,$600000,$600000
        DC      $a00000,$a00000,$600000,$a00000,$a00000,$a00000,$a00000,$a00000
        DC      $a00000,$600000,$600000,$a00000,$a00000,$a00000,$a00000,$a00000
        DC      $600000,$a00000,$600000,$a00000,$600000,$600000,$a00000,$a00000
        DC      $600000,$600000,$600000,$a00000,$a00000,$600000,$a00000,$600000
        DC      $600000,$600000,$a00000,$a00000,$a00000,$600000,$a00000,$a00000
        DC      $a00000,$a00000,$600000,$600000,$600000,$a00000,$a00000,$600000
        DC      $600000,$a00000,$600000,$a00000,$600000,$a00000,$a00000,$a00000
        DC      $a00000,$a00000,$600000,$600000,$a00000,$600000,$a00000,$600000
;
```

Example C-1 24-bit Algorithm Program Listing (Continued)

```
          org     y:$0
PATH1     DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
          DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
PATH2     DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
          DC      $0,$0,$0,$0,$0,$0,$0,$0,0,0,0,0,0,0,0,0
BRY       DSM     NUMSTATES/2
DECOUT    DS      NUMINPUTS/8
          org     y:@cvs(y,INDATA)
YDATA     DC      $a00000,$a00000,$a00000,$600000,$a00000,$600000,$a00000,$600000
          DC      $a00000,$600000,$a00000,$a00000,$600000,$600000,$a00000,$600000
          DC      $a00000,$a00000,$600000,$a00000,$600000,$600000,$600000,$a00000
          DC      $600000,$600000,$600000,$600000,$600000,$600000,$a00000,$600000
          DC      $600000,$600000,$600000,$a00000,$a00000,$a00000,$600000,$a00000
          DC      $a00000,$a00000,$600000,$600000,$600000,$a00000,$600000,$a00000
          DC      $600000,$600000,$600000,$600000,$a00000,$600000,$a00000,$a00000
          DC      $600000,$600000,$a00000,$600000,$600000,$600000,$a00000,$600000
          DC      $a00000,$600000,$600000,$a00000,$a00000,$600000,$a00000,$a00000
          DC      $a00000,$a00000,$a00000,$600000,$a00000,$a00000,$600000,$600000
          DC      $600000,$a00000,$600000,$a00000,$a00000,$600000,$a00000,$600000
          DC      $600000,$600000,$a00000,$a00000,$600000,$a00000,$600000,$a00000
          DC      $a00000,$600000,$600000,$a00000,$a00000,$600000,$a00000,$a00000
          DC      $a00000,$a00000,$a00000,$a00000,$a00000,$600000,$a00000,$600000
          DC      $a00000,$600000,$a00000,$600000,$600000,$600000,$600000,$600000
          DC      $a00000,$a00000,$a00000,$a00000,$a00000,$600000,$600000,$a00000
          DC      $a00000,$a00000,$a00000,$600000,$a00000,$a00000,$600000,$a00000
          DC      $600000,$600000,$a00000,$600000,$600000,$600000,$a00000,$a00000
          DC      $600000,$600000,$a00000,$a00000,$600000,$a00000,$a00000,$600000
          DC      $a00000,$600000,$600000,$600000,$600000,$a00000,$a00000,$a00000
          DC      $a00000,$a00000,$600000,$a00000,$600000,$600000,$600000,$600000
          end
```

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