Viterbi Algorithms as a Stream Application

Final Project

John Davis, Andrew Lin, Njuguna Njoroge, Ayodele Thomas

EE482C Advanced Computer Organization: Stream Processor Architectures

Stanford University, Spring 2002

Table of Contents

Summary of Project and Findings

The Viterbi algorithm (VA) is a popular method used to decode convolutionally coded messages. The algorithm tracks down the most likely state sequences the encoder went through in encoding the message, and uses this information to determine the original message. Instead of estimating a message based on each individual sample in the signal, the convolution encoding and Viterbi decoding process packages and encodes a message as a sequence, providing a level of correlation between each sample in the signal [1,2].

Viterbi decoders are usually implemented using a DSP or with specialized hardware [3]. Because of the streaming nature of the encoding input, the Viterbi can also be implemented in a stream architecture like Imagine. We compare a DSP implementation of the Viterbi algorithm to an implementation of the Viterbi on the Imagine architecture. We also consider the benefits and limitations of using the Imagine architecture for generalized feedback/feedforward algorithms.

Because Viterbi implementations on DSP can take advantage of specialized hardware, they are able to extract a lot of parallelism. There is an abundant amount of performance that can be extracted from a streams implementation, but limitations of the Imagine architecture and compilation environment prevent it from reaching the full potential. We suggest several additions to the Imagine architecture motivated by the DSP implementations that would improve performance.

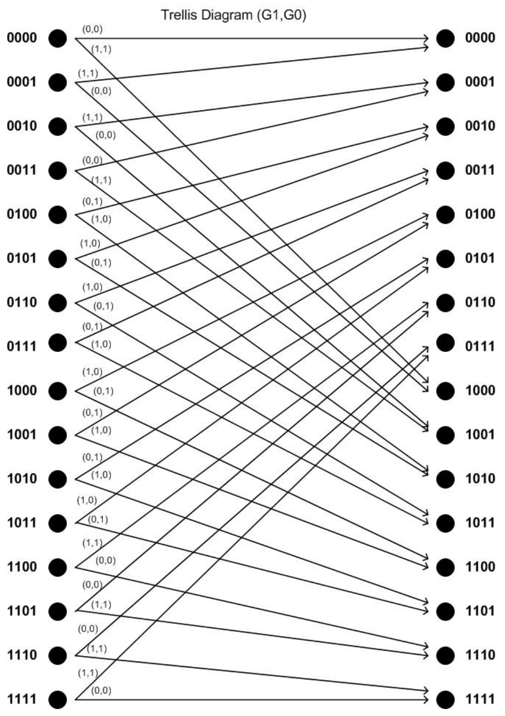
What we did

Description of the Viterbi Algorithm

The Viterbi algorithm is comprised of two routines - a metric update and a traceback. The metric update accumulates probabilities for all states based on the current input symbol using the state transitions represented by a trellis diagram (Figure 1). The traceback routine reconstructs the original data once a path through the trellis is identified.

The Viterbi that we implement is a 16-state, constraint length 5, ½ rate, radix-2 convolutional decoder. The rate is the ratio of input bits to output bits. Two bits are transmitted for each input bit for a coding rate of ½. In Figure 4, X(n) is the uncoded input and $G_0(n)$, $G_1(n)$ are the encoded outputs. Radix-2 means that there are only two possible next states for each current state. The current state is identified by the last 4 inputs to the encoder. Figure 2 shows the state transition trellis for this encoder. The state transition trellis can be reordered and visualized as a series of butterfly trellis' that indicate the state transitions and possible outputs (Figure 2).

There are generally two approaches to the Viterbi algorithm. The sliding window is the first approach. With this approach, the dependency chain is as long as the data stream and there is only a single statewide compare during the traceback. A second approach uses frames. With frames, the dependency chain is reduced to the length of the frame. In order to break the dependency chain, the input stream can be padded with zeros. That would allow the decoder to assume that the start state is 0000 when starting the metric update, and that the end state of the frame is 0000 when starting the traceback. This eliminates the need for the 16 traceback compares.



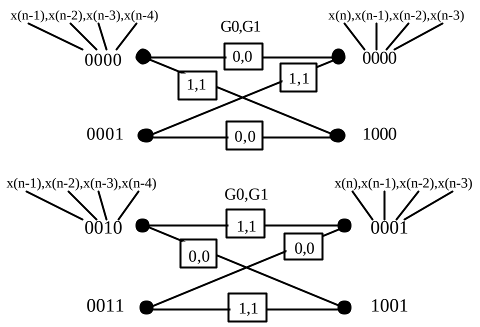

x(n-1),x(n-2),x(n-3),x(n-4)
G0,G1
x(n),x(n-1),x(n-2),x(n-3)
0000
0,0
1,1
1,1
0000
0001
0,0
1000
x(n-1),x(n-2),x(n-3),x(n-4)
G0,G1
x(n),x(n-1),x(n-2),x(n-3)
0010
1,1
0,0
0,0
0001
0011
1,1
1001

Figure 2: Reordered Butterfly Trellis' For paths originating from states 0000, 0001, 0010, and 0011.

$$G_0(n) = x(n) + x(n-1) + x(n-3) + x(n-4)$$
$$G_1(n) = x(n) + x(n-2) + x(n-3) + x(n-4)$$

Figure 3: System equations for 16-state rate ½ convolutional Encoder

Figure 1: 16-state, radix-2 State Transition Trellis

Viterbi Decoding on the Imagine Architecture

The Viterbi Algorithm as a Streaming Application

The Viterbi algorithm fits nicely into the streaming paradigm, although there are issues with its implementation on the Imagine architecture. For a ½ rate system, a stream of 3-bit integer pairs, where each integer in the pair can take on the values [-4,-3,-2,-1,0,1,2,3], is the input to the decoder. The output is a stream of single integers that can also be stored as packed bits. The decoding process involves two major steps, a metric update and a traceback, which can be thought of as kernels. An illustration of how these two kernels might interconnect with input and output streams is shown in Figure 4.

In this particular model, a stream that consists of the coded message and injected noise is input to the Viterbi decoder. Within the decoder, a metric update kernel is performed, which produces two streams – a state metric stream, which contains the accumulated state metrics for all delay states, and a transition stream, which contains the optimal path chosen for each delay state. These two streams are passed to a traceback kernel, which traverses the state metric stream and employs the transition stream to find the optimal path through the trellis.

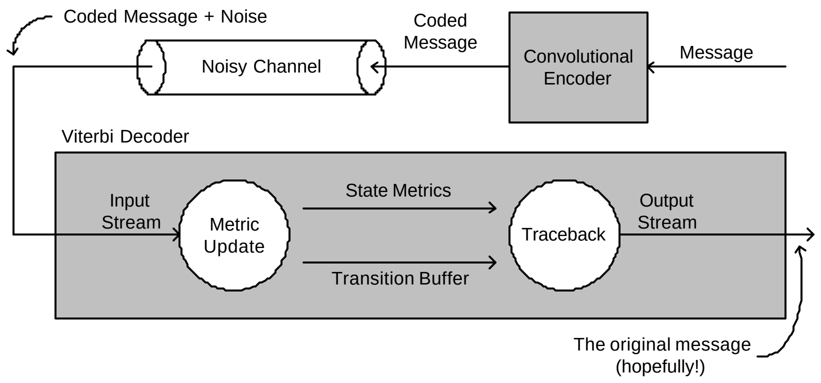


Figure 4: Streaming implementation of Viterbi algorithm

Exploiting Parallelism

An ILP approach to partitioning the Viterbi algorithm for Imagine sends elements from a single stream to all eight clusters (Figure 5). Although there is a lot of computation that has the potential to be exploited by the streams architecture, this algorithm is actually communication bound. The problem stems from the fact that each sample is correlated with adjacent samples. The calculation of possible branch metrics is completely independent from one data element to the next. However, the algorithm is serialized by the mechanism that determines the possible state metrics for a particular record of the input stream. Whether implemented using the sliding window or frame approach, intercluster communication is required.

In order for the state metrics for time T to be computed, the state metrics from the previous time period must first be communicated since the new state metrics are found from the sum of the old state metrics and the new local distance from the encoded input stream. In addition to the time waiting for the new state metrics to be computed, additional stall time is incurred by the actual communication of the metrics, which because of architectural limitations, must be done serially (Figure 5). Because this is a 16-state decoder, 16 values would have to be communicated between clusters for each input record, increasing the delay and minimizing the benefit of using a streams architecture. Communication and stall time have a detrimental effect on the performance that is achieved by the Viterbi in an ILP implementation. Therefore, it does not make sense to implement the algorithm using ILP for Imagine.

Although instruction level parallelism cannot be successfully exploited for either Viterbi approach (frame and sliding window), data level parallelism can be exploited for the frame approach. Although there is still a single input stream, for a DLP implementation each cluster would receive a different frame (Figure 6). Begin and end states are known for each frame and therefore frames can be decoded in parallel without intercluster dependencies or communication. The zero padding needed to create independent frames does lead to some reduced throughput and additional overhead. However, since the amount of padding required depends only on the constraint length (5), the cost will be amortized over the length of the frame without a noticeable impact on the SRF or throughput.

A third implementation of the Viterbi decoder uses a SIMD architecture to exploit thread level parallelism. In this implementation, each cluster receives a separate data stream and acts as a full Viterbi decoder (Figure 7). Because clusters receive separate data streams, the streams are completely independent and there is no communication between clusters, much like the DLP implementation. However, unlike the DLP implementation, there does not have to be additional overhead or reduced throughput since this approach can be used for either frames or sliding window. TLP will not be useful for applications that require only a single Viterbi decoder. Our implementation can represent the exploitation of either DLP or TLP.

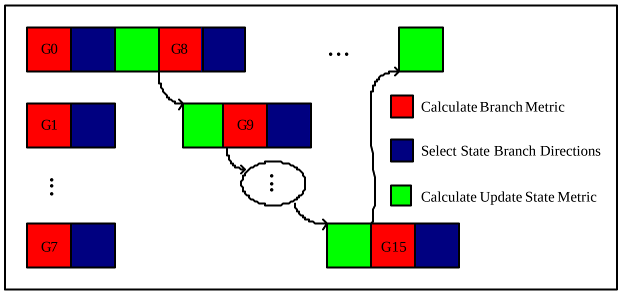


Figure 5: Viterbi exploiting ILP on Imagine

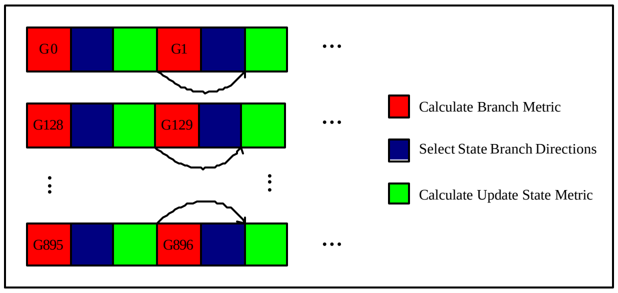


Figure 6: Viterbi exploiting DLP on Imagine

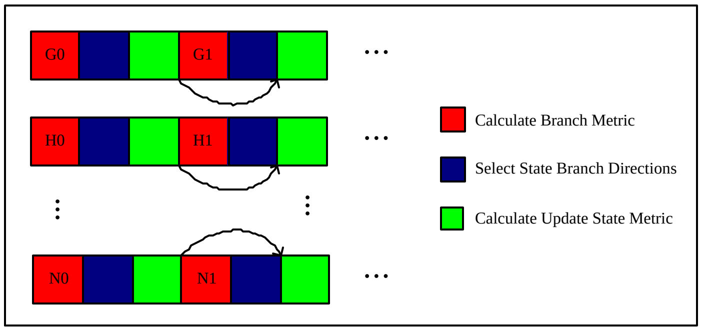


Figure 7: Viterbi exploiting TLP on Imagine

StreamC/KernelC Implementation

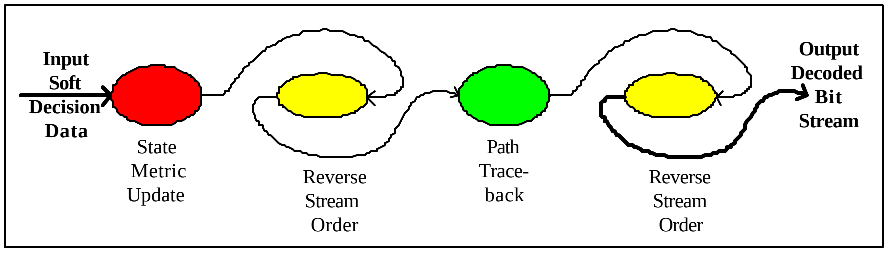


Figure 8: Imagine Kernels

The Imagine implementation of the Viterbi algorithm consists of three kernelC modules and a streamC module that invokes the three kernels. The implementation reads in an integer file of soft decision pairs into a stream of encodedG records. encodedG records have two integer fields, named g0 and g1. This stream is passed into the state metric update kernel, which we named statemetric in kernelC.

This module performs several key functions: It first creates the state trellis structure. This structure utilizes 32 trellisStruct type variables. trellisStruct records contain three fields, the direction (whether it is upper or lower), the next state and the g0 and g1 collapsed into one variable. After the trellis is formed, the kernel loops through each element of the input stream. For each element, the statemetric kernel computes the branch metric, the new state metric and chooses the best state metric for each of the 16 states for that time sample. The path direction is stored for each of the 16 states based on the best state metric for that state. This information is stored in a statesRecord record variable that is outputted in each iteration of the input stream loop. After the kernel has iterated through the whole stream, it also outputs the state metric values of the last time sample.

The outputs of the state metric kernel need to be reversed since the Viterbi algorithm commences the path trace back at the end of the stream. We reverse the stream by passing an index stream into a stream derivation function in streamC. We generate the reverse index stream in kernelC. Since the stream derivation function accesses the stream as 32-bit words, we had to tailor our reverse index stream so that it reversed the *records*, not the words. After the stream reversal, the state transition and the final state metric are passed into the path traceback kernel. This kernel performs a 16-way compare to determine which state is the best. From there, it traverses the transition stream, following the direction (upper or lower) indicated by the stateRecord entry. After each iteration, the kernel outputs the appropriate bit, which is encoded as either an integer 0 or an integer 1 in our implementation.

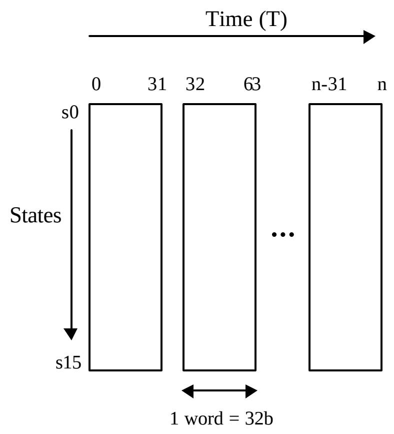
Finally, this stream is reversed (using the reverse index stream kernel) and is saved to a file. Note that we were simulating the situation that each cluster is a Viterbi decoder, so we replicated our input data 8 times for each cluster.

Integer vs. Bit Manipulation

The encoded input stream can be represented in one of two ways – as a stream of integers or as a stream of packed bits. For our initial Imagine implementation, we choose to use a stream of integers. This choice was made for ease of programming. However, we recognize that this choice has a negative impact on storage space. If the elements of the stream were represented as bits (a pair of two 3-bit integers), the soft decision value requires only 3/32 of the SRF space that is required with integers. In reality, this is best encoded using a 4-bit nibble. Thus, the reason reduction in SRF space is 1/8 of the current implementation. Since the streams that are decoded with the Viterbi are quite long, that space savings could be important in a sliding window implementation. It is not clear that using an integer representation will have a negative impact on performance since bit manipulation in Imagine would require repeated masking and shifting to extract bits and the use of temporary variables to perform the metric update and traceback.

Bit manipulation would have to take place during the state metric update and during the traceback. The intermediate data created by the state metric update could be stored in one of two ways. In the first approach, each word would represent the state transition information for a single state over 32 time periods (Figure 9). Therefore sixteen words would be streamed in and saved to a local array before traceback could begin (since there are 16 states). For each time period, a single bit would be shifted out of each array index. A compare would take place and the selected path could be indexed directly by the array. Because of the number of words that must be stored locally, this implementation could be scratchpad limited.

The second implementation has more complex bit manipulations, but would result in less scratchpad pressure. In this approach, data for all 16 states during a particular time period would be packed into a half word (Figure 10). In order to access the information, the word would be masked and shifted to extract the upper or lower bits. Then, once the appropriate state is determined, more shifting and masking would have to take place to extract the correct bit. Because of the additional shifting and masking steps, this implementation would be less efficient and may not afford any more performance that the previous scratchpad limited version.



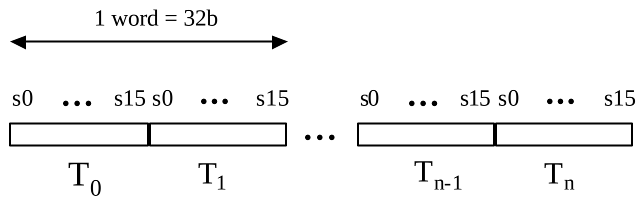


Figure 9: Packed by state

Figure 10: Packed by time

Performance Results

Table 2 provides the isim cycle accurate simulation results. The results are classified in terms of time spent in various section of the StreamC/KernelC code. There is significant StreamC overhead associated with loading the microcode and doing stream management. Over half of the StreamC overhead is associated with the initial program startup, before the first KernelC function is called, the second row of Table 2. We decided to use a kernel to generate the reverse index streams. This added 655 cycles for the two kernel calls. Finally, traceback was relatively short, taking about 22 cycles per output bit for each cluster.

Total StreamC Overhead	50533 cycles
Initialization	26562 cycles
Metric Update Loop	6255 cycles
Traceback Loop	2805 cycles
Reverse Index Gen.	1310 cycles
Total	87465 cycles

Table 2: Performance results Imagine code for 8 streams, each of 128 data samples.

Given a frame based Viterbi algorithm, the associated overhead due to padding would amount to about 6% throughput loss for a frame size of 128 and 8 bits of padding. These results do not take into account stream reconstruction cost for the frame-based algorithm that exploits DLP. These results can also be used for a streaming implementation that exploits the TLP of 8 data streams. This would require 8 'C54x DSPs to achieve the same throughput.

Optimizations

There are several optimizations that can be done using the Imagine architecture. Som were more successful than others. We do not use the segmented addition and subtraction because there would be no benefit given the limited number of comparators in the cluster. Further, for the smaller reverse index streams, arrays were allocated and statically initialized to prevent KernelC calls and the associated overhead. All KernelC loops were pipelined and loop unrolled when possible. However, as shown in Figure 11, the state metric kernel is computationally dense, but scratchpad limited, making loop optimizations difficult. Finally, "expand" was used when possible to place as many of the intermediate variables in local LRFs to reduce scratchpad pressure. However, not all arrays could be expanded due to dynamic indexing or iscd failures.

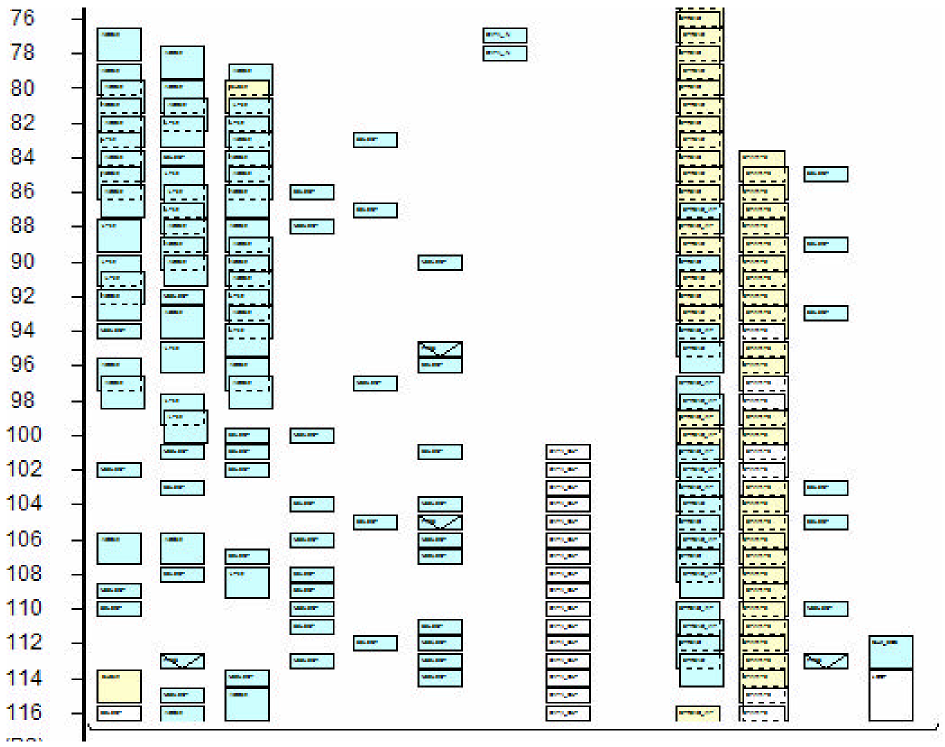


Figure 11. Statemetric update kernel schedule without the 30 cycles of scratchpad accesses.

Viterbi Decoding on the Texas Instruments 'C54x DSP

Architectural Features

The Texas Instruments TMS320C54x DSP utilizes separate program and data memory spaces, which allows instructions and data to be fetched in parallel. In addition, the DSP features four internal busses (one program memory bus, and three data memory busses), eight auxiliary registers, and address generation logic. This allows the DSP to execute multiple operand operations in parallel, reducing memory bottlenecks [4]. The 'C54x also features two 40-bit accumulators and a 40-bit adder. A multiply-accumulate (MAC) unit is also provided using one of these accumulators in combination with a 17x17-bit multiplier. The ‘C54x also includes a 40-bit ALU, which can operate in dual 16-bit mode – that is, instead of performing a single 40-bit operation, it can perform two 16-bit operations at that same time. A 40-bit barrel shifter and compare-select-store unit (CSSU) is also included. The CSSU is a piece of specialized hardware included specifically for Viterbi decoding. More details about this functional unit are presented in the next section.

The ‘C54x operates using a simple six stage pipeline, allowing six instructions to be in flight at a given time. The six stages of the pipeline are: Generate Program Address, Get Opcode, Decode Instruction, Generate Read Address, Read Operands, Generate Write Address, Execute Instruction, Write Result. The writeback operation is overlaid with the last two stages of the pipeline.

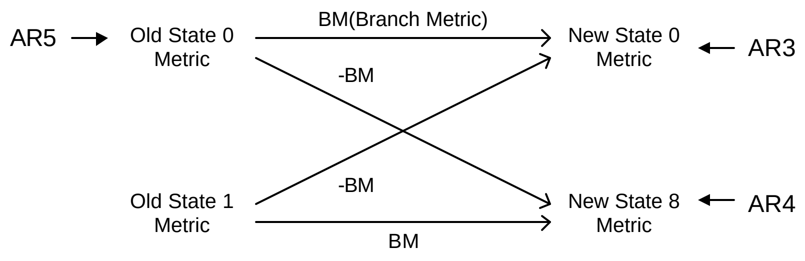
Viterbi Decoding on the ‘C54x

As mentioned in the previous section, the ‘C54x includes a compare-select-store unit for Viterbi decoding. This functional unit is used with the ALU to perform fast calculation and selection of the state metric and trellis path when updating the state metrics [4]. The CSSU unit compares the two 16-bit parts of a specified accumulator (one of the two 40-bit accumulators, which are called A and B), and shifts a 1 into a 16-bit transition register (TRN) if the 16-bit word stored in the upper 16 bits of the accumulator is larger than that which is stored in the lower 16 bits. Otherwise, a 0 is shifted into TRN. The larger value is then stored to a specified location in data memory.

In addition to the CSSU, the metric update utilizes the dual 16-bit ALU mode via the DADST and DSADT instructions [5]. These instructions perform dual add/subtract operations in parallel using the 40-bit ALU. This is best illustrated by example. A 4-state trellis, corresponding to butterfly structure that maps states 0 and 1 to states 0 and 8 in the 16-state ½ rate trellis used in this study, is shown in Figure 11. It is assumed that the lower half of auxiliary register 5 (AR5[15:0]) holds the address to the word in data memory that holds the old state metric for state 0. The upper half of AR5 points to the old state metric for state 1. AR3 points to the new state metric for state 0, and AR4 points to the new state metric for state 8. The following four lines of assembly code perform the metric update for this trellis [4]:

```
DADST *AR5, A   ; A(39:16) = *AR5(31:16) + T, A(15:0) = *AR5(15:0) - T
DSADT *AR5+%, B ; B(39:16) = *AR5(31:16) - T, B(15:0) = *AR5(15:0) + T
CMPS A, *AR3+%  ; *AR3 = max( A(31:16), A(15:0) )
CMPS B, *AR4+%  ; *AR4 = max( B(31:16), B(15:0) )
```

The first line of code computes the accumulated state metric for the paths from state 0 to state 0 and from state 1 to state 0. The next line of code computes the accumulated state metric for the paths from state 0 to state 8 and from state 1 to state 8. The third line of code selects the larger of the two paths that go to state 0, and the last line of code selects the larger of the two paths that go to state 8. The decision that results from the CMPS instruction is shifted into bit 0 of the 16-bit TRN register. Each instruction takes 1 cycle to execute.



New State 0 Metric = max(Old State 0 Metric + BM, Old State 1 Metric - BM)

New State 8 Metric = max(Old State 0 Metric - BM, Old State 1 Metric + BM)

Figure 11: The 4 state butterfly trellis mapping states 0 and 1 to states 0 and 8 for the 16 state Viterbi decoder.

The ‘C54x provides a number of features that simplify traceback. There is an instruction to extract individual bits out of a word (BITT) – that is, a bit in a word can be extracted by addressing its position in the word [5]. This is best illustrated by example. The result of the CMPS instruction is shifted into the TRN register, and this register is written to data memory after all state metrics are updated for a single timestep (or data sample). The transition buffer is thus organized as an array of 16 bit words, one word per timestep. If the input data stream has 128 data samples, then the transition buffer would be an array of 128 16-bit words. During traceback, the transition for the current sta te is extracted from the transition buffer by using the BITT instruction to address the correct bit in a word in the transition buffer array. After the bit is extracted, it is shifted into the word containing the state. The word containing the state is then masked so that the lower 4 bits remain. The state word then contains the next state, and this traceback routine repeats. This is illustrated in Figure 12.

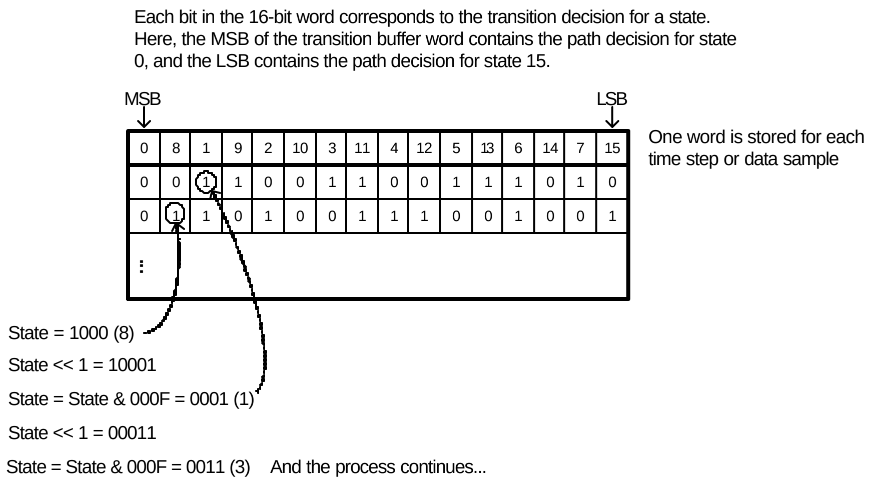

Each bit in the 16-bit word corresponds to the transition decision for a state. Here, the MSB of the transition buffer word contains the path decision for state 0, and the LSB contains the path decision for state 15.
MSB
LSB
One word is stored for each time step or data sample
0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15
0 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0
0 1 1 0 1 0 0 1 1 1 0 0 1 0 0 1
State = 1000 (8)
State << 1 = 10001
State = State & 000F = 0001 (1)
State << 1 = 00011
State = State & 000F = 0011 (3)
And the process continues...

Figure 12: A bit in the transition buffer is addressed by the state, and the next state is computed by shifting in the extracted transition bit and masking the result.

Initialization	308 cycles
Metric Update Loop	9604 cycles
Traceback Loop	1344 cycles
Total	11256 cycles

Table 3: Performance results ‘C54x Viterbi assembly code for 128 data samples.

Performance Results

The ‘C54x Viterbi assembly code was tested using a cycle accurate simulator supplied by Texas Instruments, sim500x. The performance results for 128 data samples are summarized in Table 3.

Thread level parallelism can be exploited by running several ‘C54x Viterbi decoders in parallel This would result in a total cycle time that is 7.8 times faster than what is shown in Table 2. Unlike Imagine, this would not be a single chip solution, and would require several ‘C54x chips and the board space to mount them. However, if only one ‘C54x DSP was to process the eight streams, the resulting total cycle time of 90048 cycles, 3 % more cycles than the Imagine process. Similar conclusions can be extrapolated if we consider the DLP case. In both situations, reconstructing the output stream from the individual frames was not considered.

What we learned

Generalized Viterbi in Streams

We considered a general Viterbi implementation and the impact of changing the parameters on the performance.

Changing the rate will have the greatest effect on the stream size. As the rate decreases (or similarly as the number of output bits increases) the size of the encoded stream will increase linearly. Changing the rate will also impact the number of stream elements that must be loaded before branch metric calculations can take place.

Increasing the number of available states affects the size of the intermediate streams and the amount of computation that must be done in computing them. The number of add-compare-selects that must be performed in selecting the state metric also increases. Furthermore, a larger number of states means that the trellis is larger and will take up more space in the scratchpad.

Increasing the radix will make the state transition trellis more complex. A more complex trellis increases the number of comparators that are needed to evaluate each state transition.

Therefore, performance limitations present in this implementation would be exacerbated if any of the parameters were increased.

Programming in StreamC/KernelC

Several architectural features limited the performance of the Imagine implementation. There are thirty-two intermediate values that must persist and be updated for each record of the stream. Because they overflow the LRF's and are stored in the scratchpad. Since these values must be loaded for each record, the loop becomes scratchpad limited. Multiple ports in the scratchpad would help the performance of this application. Imagine has a difficult time extracting performance for kernels that keep a large number of persistent intermediate values. Furthermore, LRF pressure is high because "expand" has limited usage for arrays within the kernels and iscd may not be able to schedule kernels with a large number of expanded arrays.

Code expansion is also a problem because nested loops are discouraged. Because there are 16 states, most operations must be repeated 16 times and must be programmed as separate lines in the source code.

More comparators would also be beneficial. Sixteen compares must take place to select the best path. Because there is only a single comparator per cluster and intercluster communication is expensive (due to the sequential access), performance is limited.

As mentioned previously, using packed bits would greatly decrease the size of intermediate streams in the Viterbi algorithm. However, bit manipulation can be costly since masking and shifting must be done repeatedly and each manipulation takes a cycle. Direct indexing of bits in a word similar to DSP implementation would allow much easier and better forming bit manipulation.

A programming limitation is the separation of the concept of structures and arrays. In KernelC, arrays of records and records composed of arrays are forbidden. Therefore, temporary arrays must be maintained to allow direct indexing by state. Being able to combine structures and arrays would make programming easier and reduce the number of intermediate variables. Furthermore, the ability to optimize nested loops would have greatly reduced the code size of the kernels. Finally, we found it difficult to index into streams of records. The documentation was incomplete, but we believe that it has improved greatly since we started the project.

Feedback/feedforward Algorithms for Streams Architectures

The Viterbi algorithm is just one example of a feedback or feedforward algorithm. Feedback/forward algorithms can take many different forms, but there are two basic types – loops that depend only on previous input values and feedback loops that rely on values generated in previous loops. Feed loops that depend on previous input data are easier to implement than feed loops that depend on values calculated during previous loops.

An IIR Filter is an example of the former type of feed loop. The filter contains a feedback loop, but only input values are fed back in the loop. Because there are no dependencies between input data, and the filter needs a finite number of previous values, edge data can be replicated so that data can be distributed across clusters with no communication. While breaking the dependency chain does increase the amount of data that must be transmitted and reduce throughput, the benefit of removing communication is more important as long the amount of replication is dominated by the size of the input stream chunk.

For the Viterbi implementation, we pad the data to allow chunking at the frame level. This is equivalent to the process of replication for the IIR filter. The main difference is the usefulness of the data being added. For the IIR, zero padding would pollute the data is should not be implemented. The data that is replicated is data that we actually care about. For the Viterbi, we pad with zeros, which is essentially junk data. However, the processes using strip-mining to chunk the data and using additional data (whether padded or replicated) to break dependency chains is a general technique that can be used for all feedback/ feedforward algorithms implemented in streams (Figure 13).

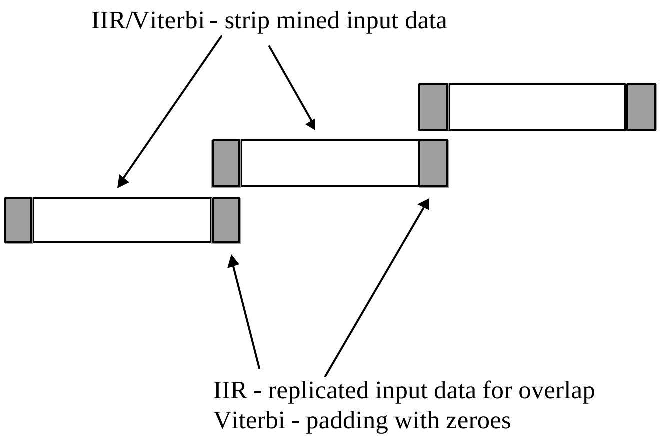

IIR/Viterbi - strip mined input data
IIR - replicated input data for overlap
Viterbi - padding with zeroes

Figure 13: Strip-Mining for Feedback/feedforward loops

Conclusions

The Viterbi algorithm (VA) is an example of a streaming algorithm that can be implemented on Imagine. The structure of the VA prevents us from exploiting all levels of parallelism, in particular ILP. From the results, it is clear that the time Imagine spends in the statemetric kernel provides slightly better than ideal speedup, based on an individual ALU cluster and DSP comparison, Imagine is 53% faster based on cycle count. The traceback does not perform as well due to the computational sparse nature of the kernel.

There are some clear improvements that can be made to the Imagine architecture that would improve its programmability and performance. The software features that the VA would have benefited most from involved bit manipulation, and memory management. Circular buffers and the ability to reverse index the SRF would have removed two kernels. Furthermore, more comparators or segmented comparators would have enabled the use of the segmented adders. Finally, bit indexing into a word would have made the traceback kernel more efficient.

There are many aspects of future work that can be explored based on this research. The majority of the overhead for the Viterbi decoding was spent in StreamC. Thus pipelining StreamC could reduce the overhead and initialization time. Improving the kernel scheduler would reduce the scratchpad pressure and use more of the LRF. This would have enabled better pipelining of the statemetric kernel. In addition, adding the ability to reverse index the SRF would be useful for this type of application. Overall, stream management and bit manipulation was the major barrier to this project, so any improvement would be beneficial.

References

1. A. J. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,: IEEE Trans. Information Theory, vol. IT-13, pp.260-269, April 1967

2. David Forney Jr., "The Viterbi Algorithm", Proceedings of the IEEE, Vol. 61, No. 3, March 1973.

3. P. Black, T. Meng, “A 140-Mb/s, 32-State, Radix-4 Viterbi Decoder,” *Journal of Solid State Circuits*, Vol. 27, No. 12, December 1992

4. “TMS320C54x DSP CPU and Peripherals: Reference Set, Volume 1,” Texas Instruments, 1997.

5. Hendrix, H., “Viterbi Decoding Techniques in the TMS320C54x Family,” Texas Instruments, June 1996.

Appendix A: Imagine Code

Stream C: viterbi_sc.cpp

```
#include "idb_streamc.hpp"
#include "viterbi_h.hpp"
#include "indexGen.hpp"
#include "unroll_depth.h"


#define NUM_CLUSTER 8
#define MAX_INDEX_ARR_SIZE 16384

STREAMPROG(my_viterbi);

extern String get_next_word(char** s);
extern int    get_next_num(char** s);


static void ReverseStream(StreamSchedulerInterface& scd, im_stream<statesRecord> &in, int stride,
                                               im_stream<statesRecord> &rev);
static void ReverseStreamInt(StreamSchedulerInterface& scd, im_stream<im_int> &in, int stride,
                                               im_stream<im_int> &rev);
static void ReverseStream_State(StreamSchedulerInterface& scd, im_stream<statesRecord> &in,
                                               im_stream<statesRecord> &rev);
static void ReverseStream_Int(StreamSchedulerInterface& scd, im_stream<im_int> &in,
                                               im_stream<im_int> &rev);
static void MakeReversedIndexStream(StreamSchedulerInterface& scd, int length, int record_size,
                                                    im_stream<im_int> out);


// defining stream program
void my_viterbi(StreamSchedulerInterface& scd, String args) {
 // read parameters (input file name, comparison file name)

  String in_fname;
  String out_fname;
  String comp_fname;
  int in_length;
  int sMetric [] = {112, 96, 80, 64, 48, 32, 16, 0};

  im_uc<im_int> loop_iter;

  //char i_str_buff[6];
  //String i_string;


  if (args.length() > 3) {
    char* a = args.char_ptr();

    in_fname = get_next_word(&a);
        in_length = get_next_num(&a);

    out_fname = get_next_word(&a);
        comp_fname = get_next_word(&a);

        cout<<"Input file:"<<in_fname<<" size: "<<in_length<<endl;
        cout<<"Output file: "<<out_fname<<endl;
        cout<<"Compare file: "<<comp_fname<<endl;
  }
  else {
    cerr << "Usage:<input_data_file> <length> <comparison_name>" << endl;
    return;
  }

        // We assume that the data files is divisable by the fir tap length and cluster.

        cout<< "Reading in streams!"<<endl;
        // Stream Declarations

        im_stream<encodedG> NAMED(in_s) = newStreamData<encodedG>(in_length / 2);

        im_stream<statesRecord> NAMED (transition_s) = newStreamData<statesRecord>(in_length / 
2);
        im_stream<statesRecord> NAMED (rev_transition_s) = newStreamData<statesRecord>(in_length 
/ 2);

        im_stream<statesRecord> NAMED (final_metric_s) = 
newStreamData<statesRecord>(NUM_CLUSTER);
        im_stream<statesRecord> NAMED (rev_final_metric_s) = 
newStreamData<statesRecord>(NUM_CLUSTER);

        im_stream<im_int> NAMED(rev_out_combined_s) = newStreamData<im_int>(in_length / 2 );
        im_stream<im_int> NAMED(out_combined_s) = newStreamData<im_int>(in_length / 2 );
        //im_stream<im_int> NAMED(out_s) = newStreamData<im_int>(in_length / NUM_CLUSTER);

        loop_iter = 16;

        // load the input data from the files
        streamLoadFile(in_fname.char_ptr(), "txt", "", in_s);


        statemetric(in_s, transition_s, final_metric_s);

        ReverseStream(scd, transition_s, 16, rev_transition_s);

        im_stream<im_int> &index_s = newStreamData<im_int>(NUM_CLUSTER);
        streamLoadBin(sMetric, NUM_CLUSTER, index_s);
        rev_final_metric_s = final_metric_s(0, NUM_CLUSTER, im_fixed, im_acc_index, index_s);

        bestpath(rev_transition_s, final_metric_s, rev_out_combined_s, loop_iter);

        ReverseStreamInt(scd, rev_out_combined_s, 1, out_combined_s);

        //streamSaveFile(out_fname.char_ptr(), "txt", "d", out_combined_s);
        //streamCompareFile(comp_fname.char_ptr(), rev_out_combined_s, 0, "a");

}



/* ReverseStream
 * --------------------
 * Reverses the order of the elements in a stateRecord type stream.
 */

static void ReverseStream(StreamSchedulerInterface& scd, im_stream<statesRecord> &in, int stride,
                                              im_stream<statesRecord> &rev)
{
        int in_length = in.getLength();
        String test_fname;
        im_uc<im_int>          start_Index;
        im_uc<im_int>          size;
        im_uc<im_int>          rec_stride;
        im_stream<im_int> &index_s = newStreamData<im_int>(in_length);

        start_Index = in_length;
        size = ((in_length)/(NUM_CLUSTER*UDEPTH));
        rec_stride = stride;

        index(start_Index, rec_stride, size, index_s);
        test_fname = "Viterbi\\vit_files\\state_index.out";
        streamSaveFile(test_fname.char_ptr(), "txt", "d", index_s);
        //MakeReversedIndexStream(scd, in_length, 16, index_s);
        rev = in(0, in_length, im_fixed, im_acc_index, index_s);

}

/* ReverseStreamInt
```

```
 * --------------------
 * Reverses the order of the elements in a stateRecord type stream.
 */

static void ReverseStreamInt(StreamSchedulerInterface& scd, im_stream<im_int> &in, int stride,
                                               im_stream<im_int> &rev)
{
       int in_length = in.getLength();
       String test_fname;
       im_uc<im_int>         start_Index;
       im_uc<im_int>         size;
       im_uc<im_int>         rec_stride;
       im_stream<im_int> &index_s = newStreamData<im_int>(in_length);

       start_Index = in_length;
       size = ((in_length)/(NUM_CLUSTER*UDEPTH));
       rec_stride = stride;

       index(start_Index, rec_stride, size, index_s);
       test_fname = "Viterbi\\vit_files\\int_index.out";
       streamSaveFile(test_fname.char_ptr(), "txt", "d", index_s);

       //MakeReversedIndexStream(scd, in_length, 16, index_s);
       rev = in(0, in_length, im_fixed, im_acc_index, index_s);

}
```

Kernel C: statemetric_kc.cpp

```
#include "idb_kernelc.hpp"
#include "viterbi_h.hpp"
#include "idb_kernelc2.hpp"
#include "unroll_depth.h"

// Path definitions

#define UPPER 0
#define LOWER 1


KERNELDEF(statemetric, "Viterbi/statemetric_kc.uc");

// in - input stream of encoder output values
// out - output stream of state transition values
// out_final_metric - output stream of state metric at final time

kernel statemetric(istream<encodedG> in, ostream<statesRecord> out, ostream<statesRecord>
out_final_metric)
{


       //int i;
       int temp1, temp2;

       encodedG g; // stream input variable
       int g0, g1;    // output vars

       //array<trellisStruct> trellis (16) (2);     // trellis[#states][radix]
                                            //    radix = 2; #states = 16
                                            //    trellis[x][y].state = next_state;
                                            //    trellis[x][y].dir = direction;

       expand<int> prev_state (16);  // State Metric for previous time period
       array<int> curr_state (16);   // State Metric for current time period , can't expand no
sched
       array<int> br_m (4);          // branch metric table , cannot expand line 286

       int state_transition;   // State Transition path
       statesRecord transitions; // State Transition record
       statesRecord final_metric; // Final State Metrics
       int br_m1, br_m2;     // branch metric options
```

```
    int path1, path2;     // path options

    cc comp;

    // Initialize the butterfly Trellis
    //
    // g0g1 = 0 -> g0,g1 = 0,0
    // g0g1 = 1 -> g0,g1 = 0,1
    // g0g1 = 2 -> g0,g1 = 1,0
    // g0g1 = 3 -> g0,g1 = 1,1

    // treliss[state][radix]--state from 0 to 15, path dir 0 or 1
//    radix = 2; #states = 16
//    trellis[x][y].state = next_state;
//    trellis[x][y].dir = direction;

    trellisStruct trellis00, trellis01;
    trellisStruct trellis10, trellis11;
    trellisStruct trellis20, trellis21;
    trellisStruct trellis30, trellis31;
    trellisStruct trellis40, trellis41;
    trellisStruct trellis50, trellis51;
    trellisStruct trellis60, trellis61;
    trellisStruct trellis70, trellis71;
    trellisStruct trellis80, trellis81;
    trellisStruct trellis90, trellis91;
    trellisStruct trellis100, trellis101;
    trellisStruct trellis110, trellis111;
    trellisStruct trellis120, trellis121;
    trellisStruct trellis130, trellis131;
    trellisStruct trellis140, trellis141;
    trellisStruct trellis150, trellis151;



    // Butterfly 1

    trellis00.state = 0;  // next state
    trellis00.dir = UPPER;        // 0 = upper path
    trellis00.g0g1 = 0;   // G0,G1 value

    trellis01.state = 8;
    trellis01.dir = LOWER; // 1 = lower path
    trellis00.g0g1 = 3;   // G0,G1 value

    trellis10.state = 0;
    trellis10.dir = UPPER;
    trellis10.g0g1 = 3;

    trellis11.state = 8;
    trellis11.dir = LOWER;
    trellis11.g0g1 = 0;

    // Butterfly 2
    trellis20.state = 1;
    trellis20.dir = UPPER;
    trellis20.g0g1 = 3;

    trellis21.state = 9;
    trellis21.dir = LOWER;
    trellis21.g0g1 = 0;

    trellis30.state = 1;
    trellis30.dir = UPPER;
    trellis30.g0g1 = 0;

    trellis31.state = 9;
    trellis31.dir = LOWER;
    trellis31.g0g1 = 3;

    // Butterfly 3
```

```
trellis40.state = 2;
trellis40.dir = UPPER;
trellis40.g0g1 = 1;

trellis41.state = 10;
trellis41.dir = LOWER;
trellis41.g0g1 = 2;

trellis50.state = 2;
trellis50.dir = UPPER;
trellis50.g0g1 = 2;

trellis51.state = 10;
trellis51.dir = LOWER;
trellis51.g0g1 = 1;

// Butterfly 4
trellis60.state = 3;
trellis60.dir = UPPER;
trellis60.g0g1 = 2;

trellis61.state = 11;
trellis61.dir = LOWER;
trellis61.g0g1 = 1;

trellis70.state = 3;
trellis70.dir = UPPER;
trellis70.g0g1 = 1;

trellis71.state = 11;
trellis71.dir = LOWER;
trellis71.g0g1 = 2;

// Butterfly 5
trellis80.state = 4;
trellis80.dir = UPPER;
trellis80.g0g1 = 2;

trellis81.state = 12;
trellis81.dir = LOWER;
trellis81.g0g1 = 1;

trellis90.state = 4;
trellis90.dir = UPPER;
trellis90.g0g1 = 1;

trellis91.state = 12;
trellis91.dir = LOWER;
trellis91.g0g1 = 2;

// Butterfly 6
trellis100.state = 5;
trellis100.dir = UPPER;
trellis100.g0g1 = 1;

trellis101.state = 13;
trellis101.dir = LOWER;
trellis101.g0g1 = 2;

trellis110.state = 5;
trellis110.dir = UPPER;
trellis110.g0g1 = 2;

trellis111.state = 13;
trellis111.dir = LOWER;
trellis111.g0g1 = 1;

// Butterfly 7
trellis120.state = 6;
trellis120.dir = UPPER;
trellis120.g0g1 = 3;
```

```
trellis121.state = 14;
trellis121.dir = LOWER;
trellis121.g0g1 = 0;

trellis130.state = 6;
trellis130.dir = UPPER;
trellis130.g0g1 = 0;

trellis131.state = 14;
trellis131.dir = LOWER;
trellis131.g0g1 = 3;

// Butterfly 8
trellis140.state = 7;
trellis140.dir = UPPER;
trellis140.g0g1 = 0;

trellis141.state = 15;
trellis141.dir = LOWER;
trellis141.g0g1 = 3;

trellis150.state = 7;
trellis150.dir = UPPER;
trellis150.g0g1 = 3;

trellis151.state = 15;
trellis151.dir = LOWER;
trellis151.g0g1 = 0;

// Initialize the state Metrics
prev_state[0] = 0;
curr_state[0] = 0;
prev_state[1] = 0;
curr_state[1] = 0;
prev_state[2] = 0;
curr_state[2] = 0;
prev_state[3] = 0;
curr_state[3] = 0;
prev_state[4] = 0;
curr_state[4] = 0;
prev_state[5] = 0;
curr_state[5] = 0;
prev_state[6] = 0;
curr_state[6] = 0;
prev_state[7] = 0;
curr_state[7] = 0;
prev_state[8] = 0;
curr_state[8] = 0;
prev_state[9] = 0;
curr_state[9] = 0;
prev_state[10] = 0;
curr_state[10] = 0;
prev_state[11] = 0;
curr_state[11] = 0;
prev_state[12] = 0;
curr_state[12] = 0;
prev_state[13] = 0;
curr_state[13] = 0;
prev_state[14] = 0;
curr_state[14] = 0;
prev_state[15] = 0;
curr_state[15] = 0;


loop_stream(in) pipeline(1) unroll(UDEPTH){
        in >> g;
        g0 = g.g0;
        g1 = g.g1;

        // Calculate branch metrics (local distance)
```

```
               br_m[0] = g0 + g1;
               br_m[1] = g0 - g1;
               br_m[2] = g1 - g0;
               br_m[3] = 0 - (g0 + g1);  // apparently Imagine can't do - g0 - g1 b/c of the
first
                                                              // negative sign.

               //
               // Find new State metrics and transitions for each state
               //

               // *********** State 0 *************

               // Get the possible branching options
               //  and state transition paths
               br_m1 = br_m[ trellis00.g0g1 ] ;
               path1 = trellis00.dir ;
               br_m2 = br_m[ trellis01.g0g1 ] ;
               path2 = trellis01.dir ;

               temp1 = prev_state[0] + br_m1;
               temp2 = prev_state[0] + br_m2;

               // Select the best option for the state metric
               //    and state transition path for this state
               comp = itocc(temp1 > temp2);
               curr_state[0] = select(comp, temp1, temp2);
               state_transition = select(comp, path1, path2);

               // output the state transitions to a stream
               transitions.state0 = state_transition;

               // Feedback
               prev_state[0] = curr_state[0];


           // *********** State 1 *************

               // Get the possible branching options
               //  and state transition paths
               br_m1 = br_m[ trellis10.g0g1 ] ;
               path1 = trellis10.dir ;
               br_m2 = br_m[ trellis11.g0g1 ] ;
               path2 = trellis11.dir ;

               temp1 = prev_state[1] + br_m1;
               temp2 = prev_state[1] + br_m2;

               // Select the best option for the state metric
               //    and state transition path for this state
               comp = itocc(temp1 > temp2);
               curr_state[1] = select(comp, temp1, temp2);
               state_transition = select(comp, path1, path2);

               // output the state transition to a stream
               transitions.state1 = state_transition;

               // Feedback
               prev_state[1] = curr_state[1];


           // *********** State 2 *************

               // Get the possible branching options
               //  and state transition paths
               br_m1 = br_m[ trellis20.g0g1 ] ;
               path1 = trellis20.dir ;
               br_m2 = br_m[ trellis21.g0g1 ] ;
               path2 = trellis21.dir ;

               temp1 = prev_state[2] + br_m1;
```

```
    temp2 = prev_state[2] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[2] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state2 = state_transition;

    // Feedback
    prev_state[2] = curr_state[2];


// *********** State 3 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis30.g0g1 ] ;
    path1 = trellis30.dir ;
    br_m2 = br_m[ trellis31.g0g1 ] ;
    path2 = trellis31.dir ;

    temp1 = prev_state[3] + br_m1;
    temp2 = prev_state[3] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[3] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state3 = state_transition;

    // Feedback
    prev_state[3] = curr_state[3];

// *********** State 4 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis40.g0g1 ] ;
    path1 = trellis40.dir ;
    br_m2 = br_m[ trellis41.g0g1 ] ;
    path2 = trellis41.dir ;

    temp1 = prev_state[4] + br_m1;
    temp2 = prev_state[4] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[4] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state4 = state_transition;

    // Feedback
    prev_state[4] = curr_state[4];
// *********** State 5 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis50.g0g1 ] ;
    path1 = trellis50.dir ;
    br_m2 = br_m[ trellis51.g0g1 ] ;
    path2 = trellis51.dir ;
    temp1 = prev_state[5] + br_m1;
    temp2 = prev_state[5] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[5] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state5 = state_transition;

    // Feedback
    prev_state[5] = curr_state[5];
// *********** State 6 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis60.g0g1 ] ;
    path1 = trellis60.dir ;
    br_m2 = br_m[ trellis61.g0g1 ] ;
    path2 = trellis61.dir ;

    temp1 = prev_state[6] + br_m1;
    temp2 = prev_state[6] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[6] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state6 = state_transition;

    // Feedback
    prev_state[6] = curr_state[6];
// *********** State 7 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis70.g0g1 ] ;
    path1 = trellis70.dir ;
    br_m2 = br_m[ trellis71.g0g1 ] ;
    path2 = trellis71.dir ;

    temp1 = prev_state[7] + br_m1;
    temp2 = prev_state[7] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[7] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state7 = state_transition;

    // Feedback
    prev_state[7] = curr_state[7];
// *********** State 8 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis80.g0g1 ] ;
    path1 = trellis80.dir ;
    br_m2 = br_m[ trellis81.g0g1 ] ;
    path2 = trellis81.dir ;

    temp1 = prev_state[8] + br_m1;
    temp2 = prev_state[8] + br_m2;
```

```
   // Select the best option for the state metric
   //    and state transition path for this state
   comp = itocc(temp1 > temp2);
   curr_state[8] = select(comp, temp1, temp2);
   state_transition = select(comp, path1, path2);

   // output the state transition to a stream
   transitions.state8 = state_transition;

   // Feedback
   prev_state[8] = curr_state[8];
// *********** State 9 *************

   // Get the possible branching options
   //  and state transition paths
   br_m1 = br_m[ trellis90.g0g1 ] ;
   path1 = trellis90.dir ;
   br_m2 = br_m[ trellis91.g0g1 ] ;
   path2 = trellis91.dir ;

   temp1 = prev_state[9] + br_m1;
   temp2 = prev_state[9] + br_m2;

   // Select the best option for the state metric
   //    and state transition path for this state
   comp = itocc(temp1 > temp2);
   curr_state[9] = select(comp, temp1, temp2);
   state_transition = select(comp, path1, path2);

   // output the state transition to a stream
   transitions.state9 = state_transition;

   // Feedback
   prev_state[9] = curr_state[9];
// *********** State 10 *************

   // Get the possible branching options
   //  and state transition paths
   br_m1 = br_m[ trellis100.g0g1 ] ;
   path1 = trellis100.dir ;
   br_m2 = br_m[ trellis101.g0g1 ] ;
   path2 = trellis101.dir ;

   temp1 = prev_state[10] + br_m1;
   temp2 = prev_state[10] + br_m2;

   // Select the best option for the state metric
   //    and state transition path for this state
   comp = itocc(temp1 > temp2);
   curr_state[10] = select(comp, temp1, temp2);
   state_transition = select(comp, path1, path2);

   // output the state transition to a stream
   transitions.state10 = state_transition;


   // Feedback
   prev_state[10] = curr_state[10];
// *********** State 11 *************

   // Get the possible branching options
   //  and state transition paths
   br_m1 = br_m[ trellis110.g0g1 ] ;
   path1 = trellis110.dir ;
   br_m2 = br_m[ trellis111.g0g1 ] ;
   path2 = trellis111.dir ;

   temp1 = prev_state[11] + br_m1;
   temp2 = prev_state[11] + br_m2;
```

```
    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[11] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state11 = state_transition;


    // Feedback
    prev_state[11] = curr_state[11];
// *********** State 12 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis120.g0g1 ] ;
    path1 = trellis120.dir ;
    br_m2 = br_m[ trellis121.g0g1 ] ;
    path2 = trellis121.dir ;

    temp1 = prev_state[12] + br_m1;
    temp2 = prev_state[12] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[11] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state12 = state_transition;


    // Feedback
    prev_state[12] = curr_state[12];
// *********** State 13 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis130.g0g1 ] ;
    path1 = trellis130.dir ;
    br_m2 = br_m[ trellis131.g0g1 ] ;
    path2 = trellis131.dir ;

    temp1 = prev_state[13] + br_m1;
    temp2 = prev_state[13] + br_m2;

    // Select the best option for the state metric
    //    and state transition path for this state
    comp = itocc(temp1 > temp2);
    curr_state[13] = select(comp, temp1, temp2);
    state_transition = select(comp, path1, path2);

    // output the state transition to a stream
    transitions.state13 = state_transition;


    // Feedback
    prev_state[13] = curr_state[13];
// *********** State 14 *************

    // Get the possible branching options
    //  and state transition paths
    br_m1 = br_m[ trellis140.g0g1 ] ;
    path1 = trellis140.dir ;
    br_m2 = br_m[ trellis141.g0g1 ] ;
    path2 = trellis141.dir ;

    temp1 = prev_state[14] + br_m1;
    temp2 = prev_state[14] + br_m2;
```

```
            // Select the best option for the state metric
            //    and state transition path for this state
            comp = itocc(temp1 > temp2);
            curr_state[14] = select(comp, temp1, temp2);
            state_transition = select(comp, path1, path2);

            // output the state transition to a stream
            transitions.state14 = state_transition;


            // Feedback
            prev_state[14] = curr_state[14];
        // *********** State 15 *************

            // Get the possible branching options
            //  and state transition paths
            br_m1 = br_m[ trellis150.g0g1 ] ;
            path1 = trellis150.dir ;
            br_m2 = br_m[ trellis151.g0g1 ] ;
            path2 = trellis151.dir ;

            temp1 = prev_state[15] + br_m1;
            temp2 = prev_state[15] + br_m2;

            // Select the best option for the state metric
            //    and state transition path for this state
            comp = itocc(temp1 > temp2);
            curr_state[15] = select(comp, temp1, temp2);
            state_transition = select(comp, path1, path2);

            // output the state transition to a stream
            transitions.state15 = state_transition;


            // Feedback
            prev_state[15] = curr_state[15];

            out << transitions;

       } // End loop_stream

       final_metric.state0 = curr_state[0];
       final_metric.state1 = curr_state[1];
       final_metric.state2 = curr_state[2];
       final_metric.state3 = curr_state[3];
       final_metric.state4 = curr_state[4];
       final_metric.state5 = curr_state[5];
       final_metric.state6 = curr_state[6];
       final_metric.state7 = curr_state[7];
       final_metric.state8 = curr_state[8];
       final_metric.state9 = curr_state[9];
       final_metric.state10 = curr_state[10];
       final_metric.state11 = curr_state[11];
       final_metric.state12 = curr_state[12];
       final_metric.state13 = curr_state[13];
       final_metric.state14 = curr_state[14];
       final_metric.state15 = curr_state[15];

       out_final_metric << final_metric;
}
```

bestpath_kc.cpp

```
#include "idb_kernelc.hpp"
#include "viterbi_h.hpp"
#include "idb_kernelc2.hpp"
#include "unroll_depth.h"
```

```
KERNELDEF(bestpath, "Viterbi/bestpath_kc.uc");

// in - input stream of state transition values
// in_state_metric - input stream of final state metric values
// out - output stream of decoded output

kernel bestpath(istream <statesRecord> in, istream <statesRecord> in_state_metric, ostream<int>
out, uc<int>& loop_cnt)
{

        int count = 0;
        //int tcount;
        int max_state = 0;
        int max_state_metric = 0;
        int current_state_metric;
        int state_transition;
        int path;
        int dir;

//      uc<int> cnt;

//      cnt = commclperm(8, count, loop_cnt);


        statesRecord final_state_metric;  // The final state metrics for input
        array<int> state_metric_arr (16); // Temp array to hold state metric values
        statesRecord transition;
        array<int> transition_arr (16);

        cc comp, comp2;

        in_state_metric >> final_state_metric;

        // How much less efficient is this than streaming 16 ints and looping through that
stream.
        state_metric_arr[0] = final_state_metric.state0;
        state_metric_arr[1] = final_state_metric.state1;
        state_metric_arr[2] = final_state_metric.state2;
        state_metric_arr[3] = final_state_metric.state3;
        state_metric_arr[4] = final_state_metric.state4;
        state_metric_arr[5] = final_state_metric.state5;
        state_metric_arr[6] = final_state_metric.state6;
        state_metric_arr[7] = final_state_metric.state7;
        state_metric_arr[8] = final_state_metric.state8;
        state_metric_arr[9] = final_state_metric.state9;
        state_metric_arr[10] = final_state_metric.state10;
        state_metric_arr[11] = final_state_metric.state11;
        state_metric_arr[12] = final_state_metric.state12;
        state_metric_arr[13] = final_state_metric.state13;
        state_metric_arr[14] = final_state_metric.state14;
        state_metric_arr[15] = final_state_metric.state15;

        loop_count(loop_cnt) pipeline(1) unroll(UDEPTH){

                // Select end point for best path
                ;
                current_state_metric = state_metric_arr[count];
                comp = itocc(current_state_metric > max_state_metric);
                max_state = select(comp, count, max_state);
                max_state_metric = select(comp, current_state_metric, max_state_metric);
                count = count + 1;
        }

        in >> transition;

        transition_arr[0] = transition.state0;
        transition_arr[1] = transition.state1;
        transition_arr[2] = transition.state2;
        transition_arr[3] = transition.state3;
        transition_arr[4] = transition.state4;
        transition_arr[5] = transition.state5;
        transition_arr[6] = transition.state6;
        transition_arr[7] = transition.state7;
        transition_arr[8] = transition.state8;
        transition_arr[9] = transition.state9;
        transition_arr[10] = transition.state10;
        transition_arr[11] = transition.state11;
        transition_arr[12] = transition.state12;
        transition_arr[13] = transition.state13;
        transition_arr[14] = transition.state14;
        transition_arr[15] = transition.state15;

        //tcount = 0;
        loop_stream(in) {
                dir = transition_arr[max_state];
                out << dir;
                max_state = shift(max_state, 1) + dir;       // This shifts state to left, then
                                                                                          //
tacks on the direction as LSB to get next state
                max_state = max_state & 15;                          // puts 0 in the upper half-
word of max_state
                in >> transition;
                transition_arr[0] = transition.state0;
                transition_arr[1] = transition.state1;
                transition_arr[2] = transition.state2;
                transition_arr[3] = transition.state3;
                transition_arr[4] = transition.state4;
                transition_arr[5] = transition.state5;
                transition_arr[6] = transition.state6;
                transition_arr[7] = transition.state7;
                transition_arr[8] = transition.state8;
                transition_arr[9] = transition.state9;
                transition_arr[10] = transition.state10;
                transition_arr[11] = transition.state11;
                transition_arr[12] = transition.state12;
                transition_arr[13] = transition.state13;
                transition_arr[14] = transition.state14;
                transition_arr[15] = transition.state15;
        //      tcount = tcount + 1;
        }
        //got output the last bit
        dir = transition_arr[max_state];
        out << dir;
}
```

indexGen_kc.cpp

```
#include "idb_kernelc.hpp"
#include "indexGen.hpp"
#include "idb_kernelc2.hpp"
#include "unroll_depth.h"
/*
        You may need to change the kernel definition
        (see wavelets.hpp)
*/

KERNELDEF(index, "Viterbi/indexGen_kc.uc");

kernel index(uc<int>& start, uc<int>& rec_stride, uc<int>& size, ostream<int> out)
{

        int myId = cid();
        int index;
        int stride;
        int tmp;

        index = commclperm(8,index, start);
        stride= commclperm(8,stride, rec_stride);

        index = index - 1;
        //tmp = myId*stride;
        //tmp2 = 8 * stride;

        loop_count(size) pipeline(1)unroll(UDEPTH){
                tmp = lo((index - myId)*stride);
                out<< tmp;
                index = index - 8;
        }
}
```

Header Files:

```
#ifndef _INDEXGEN
#define _INDEXGEN

#include "idb_types.hpp"
#include "idb_deftypes.hpp"

/*
        You may need to change the kernel declarations
        (including name) and add new kernels.
*/

kernel index(uc<int>& start, uc<int>& rec_stride, uc<int>& size, ostream<int> out);

KERNELDECL(index);

#define index KERNELCALL(index)

#include "idb_undeftypes.hpp"

#endif
/*******************************************/

#ifndef _UNROLL_DEPTH__H
#define _UNROLL_DEPTH__H

#define UDEPTH 1


#endif


/*********************************************/

#ifndef _VITERBI_H
#define _VITERBI_H

#include "idb_types.hpp"
#include "idb_deftypes.hpp"

#define NUM_STATES 16

record encodedG{
  int g0, g1;
        // may need to add 6 more dummy variables to make Imagine happy
};

// Trellis structure definition
record trellisStruct{
        int state;      // next state
        int dir;        // direction of butterfly - upper or lower path
        int g0g1;       // output 1 and output2 combined for butterfly
                        // g0g1 = 0, g0,g1 = 0,0
                        // g0g1 = 1, g0,g1 = 0,1
                        // g0g1 = 2, g0,g1 = 1,0
                        // g0g1 = 3, g0,g1 = 1,1
};
```

```
// State Record.
// Groups the 16 states together in a array of 16 integers.
record statesRecord{
	//int state[NUM_STATES];
	int state0;
	int state1;
	int state2;
	int state3;
	int state4;
	int state5;
	int state6;
	int state7;
	int state8;
	int state9;
	int state10;
	int state11;
	int state12;
	int state13;
	int state14;
	int state15;
};

kernel statemetric(istream<encodedG> in, ostream<statesRecord> out, ostream<statesRecord>
out_final_metric);
kernel bestpath(istream<statesRecord> in, istream<statesRecord> in_state_metric, ostream<int>
out, uc<int>& loop_cnt);

KERNELDECL(statemetric);
KERNELDECL(bestpath);

#define statemetric KERNELCALL(statemetric)
#define bestpath KERNELCALL(bestpath)

#include "idb_undeftypes.hpp"

#endif
```

Appendix B: C54x DSP Code

```
;;; Viterbi decoder: rate 1/2, 4 unit delays, 16 delay states, 64-bit
;;; frame, no puncturing.
;;; You're code will load the 3-bit soft decision decoder inputs into
;;; the data memory

        .mmregs                ; Use globally defined symbolic names for
; assembler related variables
      .def     start
      .ref     IN_BUF

;;; ***************************************************************
;;;
;;; Constants
IN_PORT .set    0h             ; Address of input port
OUT_PRT .set    1h             ; Address of output port
        ;; Define constants here

;;; ***************************************************************
;;;
;;; Uninitialized memory allocation
        ;; Define system variables and arrays here
        ;; Use .bss directive
        .bss Metric, 32, 1, 1 ;;Circular buffer for Metric storage
        .bss Trans_Buff, 128  ;;Transition Bufer for 128 pairs
        .bss Distance, 2      ;;Local Distance storage
        .bss Output_Buff, 8   ;;Output Buffer for 128 bits of data

        ;;; Traceback constants
T_Buff_End  .equ(Trans_Buff+127+4)   ;; END OF TRANSITION BUFFER
OUTWORDS    .equ 8

BFLY_FORW .macro
          ;; AR3 is the new ptr at 0
          ;; AR4 is the new ptr at 8
          ;; AR5 is the old metric ptr
          DADST *AR5,A   ;A = Old_Met(2*j)+T // Old_Met(2*j+1)-T
          DSADT *AR5+%,B ;B = Old_Met(2*j)-T // Old_Met(2*j+1)+T
          CMPS  A,*AR3+% ;New_Met(j)=Max(Old_Met(2*j+1)+T,Old_Met(2*j+1)-T)
                                 ;TRN=TRN << 1
                          ;If(Old_Met(2*j)+T =< Old_Met(2*j+1)-T) Then TRN[0]=1
           CMPSB,*AR4+%  ;New_Met(j+2^(k-2))=Max(Old_Met(2*j)+T,Old_Met(2*j+1)-T)
                          ;TRN=TRN << 1
                          ;If(Old_Met(2*j)-T =< Old_Met(2*j+1)+T) Then TRN[0]=1
          .endm

BFLY_REV .macro
         ;; AR3 is the new ptr at 0
         ;; AR4 is the new ptr at 8
         ;; AR5 is the old metric ptr
         DSADT  *AR5,B     ;A=Old_Met(2*j)-T // Old_Met(2*j+1)+T
         DADST  *AR5+%,A   ;B=Old_Met(2*j)+T // Old_Met(2*j+1)-T
         CMPS   B,*AR3+% ;New_Met(j)=(Max (Old_Met(2*j)+T,Old_Met(2*j+1)-T)
                          ;TRN=TRN<<1
                          ;If (Old_Met(2*j)+T =< Old_Met(2*j+1)-T) Then TRN[0]=1
         CMPS   A,*AR4+% ;New_Met(j+2^(k-2))=(Max (Old_Met(2*j) T,Old_Met(2*j+1)+T)
                          ;TRN=TRN<<1
                          ;If (Old_Met(2*j)+T =< Old_Met(2*j+1)-T) Then TRN[0]=1
         .endm

BFLY_END .macro
         ;; AR3 is the new ptr at 0
         ;; AR4 is the new ptr at 8
         ;; AR5 is the old metric ptr
         DADST  *AR5,A     ;A = Old_Met(2*j)+T // Old_Met(2*j+1)-T
         DSADT  *AR5+%,B   ;B = Old_Met(2*j)-T // Old_Met(2*j+1)+T
         CMPS   A,*AR3+0% ;New_Met(j) = Max(Old_Met(2*j+1)+T,Old_Met(2*j+1)-T)
                          ;TRN=TRN << 1
                          ;If(Old_Met(2*j)+T =< Old_Met(2*j+1)-T) Then TRN[0]=1
        CMPS    B,*AR4+0%  ;New_Met(j+2^(k-2)) = Max(Old_Met(2*j)+T,Old_Met(2*j+1)-T)
                          ;TRN=TRN << 1
                          ;If(Old_Met(2*j)-T =< Old_Met(2*j+1)+T) Then TRN[0]=1
        .endm

BIT_SHIFT .macro  ;;Computes bit position and next state
          SFTL A, -3, B
          AND  AR7, B
          ADD  A, #ONE, B
          STLM B, T

          NOP  ;; Stall between STLM and BITT

          BITT  *AR1-
          ROLTC A
          .endm

;Important constants for the viterbi decoder
WORD    .set    16
K       .set    5
ONE     .set    1
LOOP    .set    127

        .text
        NOP
start:
        ;; Initialize key bits: C16, OVM, AX0
        ;;SSBX SXM
        ;;SSBX C16
        ;;SSBX OVM
        STM #0X2B80, ST1 ;;ALL in one from A.Fernandez in class

        ;;Initialize circular buffer
        STM #32, BK
        STM #9,  AR0

        ;;LOAD input data
        STM #IN_BUF,   AR1
        RPT #(256-1)
        PORTR #IN_PORT, *AR1+

        ;;Initial State Metric table and ptr to input data
        STM #IN_BUF,      AR1
        STM #Distance,   AR2
        STM #(Metric+16), AR3
        STM #(Metric+8),  AR4
        STM #(Metric+16), AR5
        STM #Trans_Buff,  AR6
        STM #ONE, AR7  ;; STORE 1 FOR LATER USE

        ;;Init Metric Table
        ST  #0x0200, *AR3+%
        RPT #(WORD-2)
        ST  #0xFE00, *AR3+%

;;**************INITIALIZATION DONE TIME FOR VITERBI*****************

        STM #LOOP, BRC
        RPTB TraceBack-1

        ;;Local Distance calc, Load input data
        LD  *AR1+, 16, A
        SUB *AR1, 16, A, B
        STH B, *AR2+
        ADD *AR1+, 16, A, B
        STH B, *AR2

        LD *AR2-, T    ;; TEMP = LOCAL DISTANCE(0)
        BFLY_FORW      ;; STATES 0000 AND 1000
        BFLY_REV       ;; STATES 0001 AND 1001

        LD *AR2+, T    ;; TEMP = LOCAL DISTANCE(1)
        BFLY_FORW      ;; STATES 0010 AND 1010
        BFLY_REV       ;; STATES 0011 AND 1011
        BFLY_REV       ;; STATES 0100 AND 1100
        BFLY_FORW      ;; STATES 0101 AND 1101

        LD *AR2-, T    ;; TEMP = LOCAL DISTANCE(0)
        BFLY_REV       ;; STATES 0110 AND 1110
        BFLY_END       ;; STATES 0111 AND 1111

        ST TRN, *AR6+  ;; STORE PATH IN TRANSITION BUFFER

TraceBack:
        ;; PAD output with Zeroes for correct output
        ST #0, *AR6+
        ST #0, *AR6+
        ST #0, *AR6+
        ST #0, *AR6+

        RSBX   OVM     ;; overflow off

        STM #T_Buff_End, AR1   ;; END OF TRANSITION BUFFER
        STM #OUTWORDS-1, AR2   ;; # OUTPUT WORDS -1
        STM #Output_Buff+8-1, AR3 ;; end of output buffer
        NOP
        MVMM AR2, AR4
        LD #0, A               ;; Load Initial State in A: 0000

        ;;Time to do the traceback
        STM  #(15), BRC            ;; Loop 15 times
T_LOOP: RPTB #(TBEND-1)

        BIT_SHIFT

TBEND:  STL A, *AR3-
        BANZD T_LOOP, *AR2-

        ;; From TI Vitberi Doc.
        STM #15, BRC
        MAR *AR3+
        LD  *AR3, A

RVS:    SFTA A, -1, A
        STM  #15, BRC
        RPTB RVS2-1
        ROL B
        SFTA A, -1, A

RVS2:   BANZD RVS, *AR4-
        STL B, *AR3+
        LD  *AR3, A

        ;;; Write Output Data
        STM   #Output_Buff, AR2
        RPT   #(8-1)
        PORTW *AR2+, OUT_PRT

DONE:   B      DONE
```