4-9-2002

Implementation of Viterbi decoder on Xilinx XC4005XL FPGA

Nabil Abu-Khader
Wayne State University

IMPLEMENTATION OF VITERBI DECODER ON XILINX XC4005XL FPGA

by

NABIL ABU-KHADER

THESIS

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

In partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

2002

MAJOR: ELECTRICAL ENGINEERING

Approved by:

4/29/02

Advisor Date

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Pepe Siy for his encouragements and guidance during my graduate studies. His trust in my capabilities allowed me to pursue the research of my choice. I thank him for encouraging me to dwell into the dual topic of Viterbi Decoding and FPGA's.

I would also like to thank Wayne State University for providing all the necessary support in completing this work. Last but not the least I would like to acknowledge the constant encouragement and support by my mother throughout the course of my master's studies.

TABLE OF CONTENTS

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation:

The use of error-correcting codes has proven to be an effective way to overcome data corruption in digital communication channels. The Viterbi decoding algorithm is used to decode convolutional codes and is found in many systems that receive digital data that might contain errors. Viterbi decoding, also known as maximum-likelihood decoding, is comprised of the two main tasks of updating the trellis and trace-back. The trellis used in Viterbi decoding is essentially the convolutional encoder state transition diagram with an extra time dimension. The Viterbi Algorithm (VA) was first described in 1967 by Andrew J. Viterbi as a method for efficiently decoding convolution codes. In most modern communication systems, channel coding is used to increase bandwidth, add error detection and correction capabilities, and provide a systematic way to translate logical bits of information to analog channel symbols used in transmission. Convolutional coding and block coding are the two major forms of channel coding used today. As their names imply, in convolutional coding the algorithms work on a few bits at a time while in block coding big chunks of data are processed together [SWA02]. Generally, convolutional coding is better suited for processing continuous data streams with relatively small latencies. Also, since convolutional forward error correction (FEC) works well with data streams affected by the atmospheric and environmental noise (Additive White Gaussian Noise) encountered in satellite and cable communications, they have found widespread use in many advanced communication systems. Viterbi

decoding is one of the most popular FEC techniques used today and is therefore the main focus here.

Viterbi decoding and sequential decoding are the two main types of algorithms used with convolutional codes. Although sequential decoding performs very well with long-constraint based convolutional codes, it has a variable decoding time and is less suited for hardware implementations. On the other hand, the Viterbi decoding algorithm has fixed decoding times and is well suited for hardware implementations.

On the trellis of a convolutional code, the VA finds the shortest path that leads to a particular state. Metrics are associated with each branch and they can be calculated as the Hamming distance of the corresponding code word over the received word for hard decoding or as the Euclidean distance (using a quantizer) in the case of soft decoding. Many paths can lead to the same state. The VA selects the path whose summation of all metrics is the lowest. This refers to the add-compare-select (ACS) operation [GAR00].

At the receiver, the stream of data (which may now contain errors) is passed through a Viterbi decoder, which attempts to extract the most likely sequence of the transmitted data.

Projects and researches previously done did not follow the new IEEE 802.16 specifications, but here, we followed the specifications as can be seen in chapter (5).

1.2 Approach to the solution:

Viterbi decoders are generally implemented using programmable digital signal processors (DSPs) or special purpose chip sets and application-specific integrated circuits (ASICs) [DU00]. Here, we aim to implement such decoder on an FPGA.

The specific aims are:

– Understand the principles of convolutional coding and Viterbi decoders.

– Design and code parameterisable, behavioural VHDL models for a Viterbi decoder according to the new IEEE 802.16 specifications.

– Test the VHDL code then Synthesise the decoder onto an FPGA.

Programmable Logic is ideal for implementing error control coding (ECC) functions for two main reasons. First, PLDs are flexible, easing the modification of coding methods and improving algorithms. Second, the performance and density of PLDs align optimally with industry requirements. Unlike Application Specific Standard Products (ASSPs) designed specifically for ECC, programmable logic devices offer the designers the speed of hardware and the flexibility of software while implementing ECC functions.

ECC is a methodology that detects and in some cases corrects errors induced in digital data during transmission over a noisy channel (digital video/audio broadcast, satellite communications) or during storage in an unreliable medium (compact disc, digital tape). Some of the common ECC functions ideally suited for programmable logic are Reed-Solomon, Viterbi, Trellis Coded Modulation (TCM), etc.

FPGAs can speed time to market for the particular design of telecommunication applications because of their quick turnaround time [ALT01].

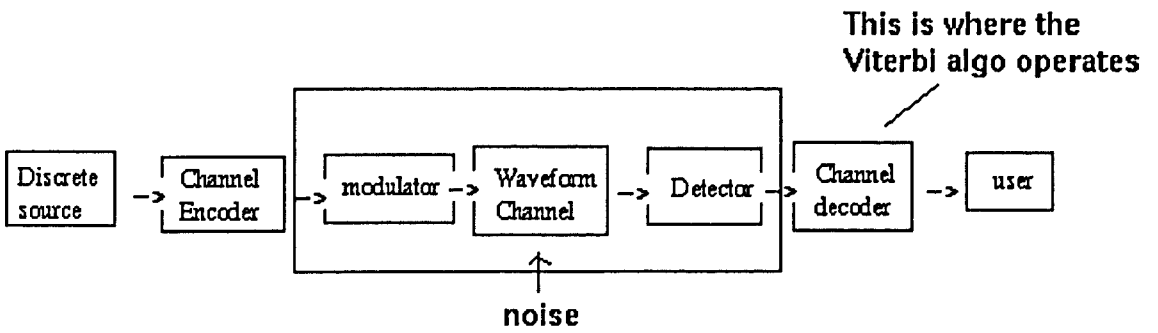
CHAPTER 2

PRELIMINARIES

In the following two sections we describe some basics towards our implementation. The first section describes a model of digital communication system and where the Viterbi algorithm is applied. The second section describes the convolutional encoding process.

2.1 A model of digital communication system:

A model of digital communication system is shown in fig (1) [NETCOM]:



Fig(1) A model of digital communication system

The discrete source generates information in the form of binary symbols. The channel encoder adds redundancy to it according to a prescribed rule before transmission. The channel decoder in the receiver uses this redundancy to decide which message bits were actually transmitted. The combined goal of the encoder and decoder is to minimize the effect of channel noise. There are many different error-correcting codes. They have been classified into "block codes", or "convolutional

codes". In block coding, an encoder generates a n - bit code word with a k - bit message block, so code words are produced on a block-by-block basis. But if we want to process the incoming bits serially rather than in large blocks, we will use convolutional coding.

2.2 Convolutional codes

Convolutional codes are used to add redundancy to a stream of data. The addition of redundancy (extra bits) allows for the detection and possible correction of incorrectly transmitted data.

The input bits are convolved in such a way that each bit influences the output more than once. Each input bit enters a shift register and the output of the encoder is derived by combining the bits in the shift register in a way determined by the structure of the encoder in use. Therefore, every bit transmitted will influence as many of the outputs as there are stages in the shift register. For example if a three bit shift register is used then, each encoded bit will influence three output bits of the encoder. The input bits could be fed in more than once at a time and the encoder could produce more than one output at each step [MEN98].

The output of a convolution encoder is dependent upon the current data and previously transmitted data. A convolutional code is characterized by its rate, constraint length, and its generator polynomials.

The code rate (k/n) of a convolution code is the ratio of the size of the input stream to the size of the output stream. The convolutional code used here has a rate of ½, which means that each input bit produces two output bits.

Rate = In /OUT_{high} OUT_{low}

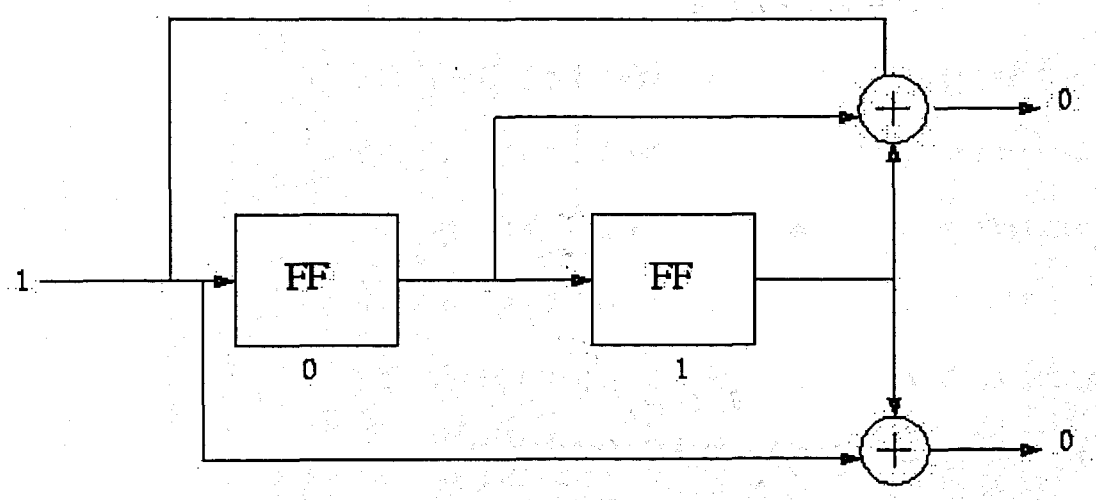
The constraint length (K) indicates the number of previous input data that must be examined along with the current input data to determine the output data, or simply, The number of stages in the shift register. The convolution code used here has a constraint length of 3. We need 2^(K-1) states. Also, The constraint length K is directly related to the number of registers in the encoder. These shift registers hold the previous data values that are systematically convolved with the incoming data bits. This redundancy of information in the final transmission stream is the key factor enabling the error correction capabilities that are necessary when dealing with transmission errors. Systems with higher constraint lengths are generally more robust. However, the complexity of the Viterbi decoder increases exponentially with the constraint length, so it is unusual to find constraint lengths greater than nine [MEN98].

The generator polynomials describe how past and current input data are used to determine the output data. The generating polynomials denote the convolutional encoder state bits, which are mathematically combined to produce an encoded bit. There is one generating polynomial per encoded bit. The convolution code used here uses the following generator polynomials:

$OUT_{low} = In(t) \oplus In(t-2)$

$OUT_{high} = In(t) \oplus In(t-1) \oplus In(t-2)$

Fig(2) shows an example of a ½ rate convolutional encoder with K=3 (4 states).

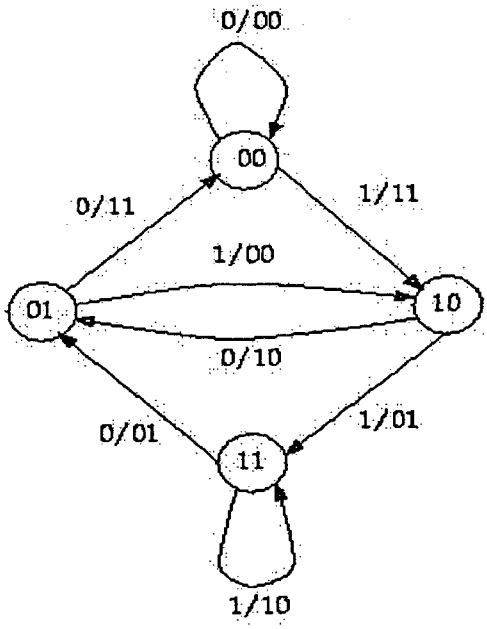


Fig(2) An example of a ½ rate convolutional encoder with K=3 (4 states)

As shown in the figure inputs enter from the left end and two outputs are generated for every input bit. The incoming bit as well as the bits in the two flipflops together form the "shift register". One output is produced by XORing all the bits of the shift register and the other is XOR output of two of them. Initially we can assume that the flipflops contain zeroes and subsequently they take on values depending on the input bits. The bit values represented by the flipflops is called the "state" of the system. In the example shown, the state of the encoder is "01".

In fig(2) [NETCOM], a particular situation is shown where the incoming bit is a 1 and the two previous bits were 0 and 1. The output of this situation would be 0 at both the outputs. The output bits are transmitted through a communication channel and are decoded by employing the Viterbi decoder at the receiving end.

The polynomials described above can be represented as a state machine (state transition diagram). The state transition diagram for the K = 3, r = 1/2 encoder is shown in fig(3).



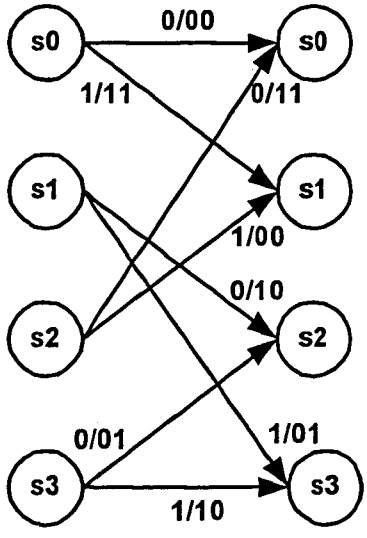
Fig(3) The state transition diagram for the K = 3, r = 1/2 convolutional encoder

Table 2.1 shows a random bit stream of data along with the output stream and the next state.

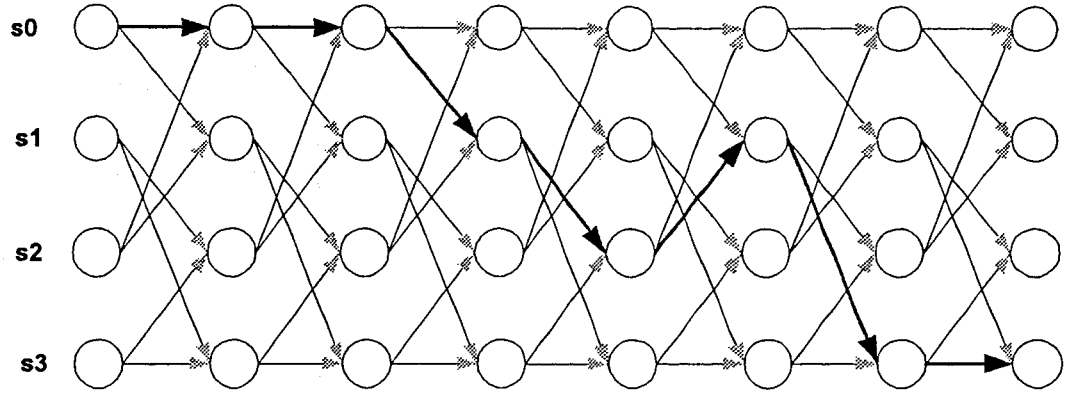
Input Stream	0	0	1	0	1	1	1
Next State	S0	S0	S1	S2	S1	S3	S3
Output Stream	0 0	0 0	1 1	1 0	0 0	0 1	1 0

Table 2.1 A random bit stream of data

The values inside the circle of fig(3) represent the current state. The values along the arrow represent the input and output bits of the encoder. For example the top circle indicates State 00. If the next input bit is a 0 then the output of the encoder will be 00 and the next state will again be 00. If the next input was a 1 instead, then the output of the encoder would be 11 and the next state would be 10. Each arrow in the above figure represents a possible transition from one state to another and is called a "branch". The Viterbi decoder will generate a branch metric value for each of the possible branches based on the input actually received by the encoder. This branch metric is the Hamming distance between the received value at the decoder and the output value associated with the branch. A trellis diagram can be used to represent the operation of a state machine over time. In a trellis diagram, the states are listed vertically; time advances from left to right. Fig(4) shows the same input data of table 2.1, represented in a trellis diagram [MEN98]. Note that upper branches are followed when the input is '0'; lower branches are followed when the input is '1'.



Fig(4a) A window of the trellis diagram showing a sample of data stream



Fig(4b) Trellis diagram showing a sample of data stream

2.3 Mapping the Channel Symbols to Signal Levels

Mapping the one/zero output of the convolutional encoder onto an antipodal baseband signaling scheme is simply a matter of translating zeroes to +1 and ones to -1. This can be accomplished by performing the operation y = 1 - 2x on each convolutional encoder output symbol [ACT97].

CHAPTER 3

VITERBI ALGORITHM

3.1 How Viterbi decoder works?

The Viterbi algorithm is based on the fact that there are only a finite number of possible states of the encoder, and that given two consecutive states we can predict the input bit(s) that would have caused that state transition.

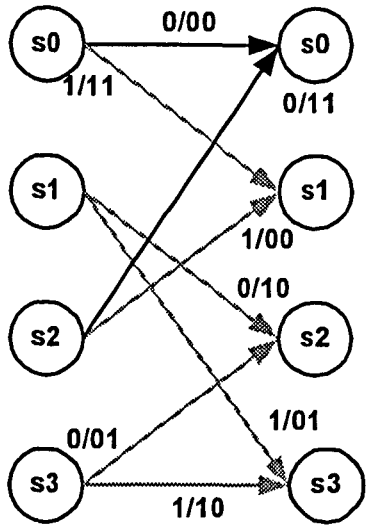
The Viterbi decoder comprises of three major parts, namely the branch metric generator (BMG), the add-compare-select unit (ACS) and the survivor memory unit.

The branch metric generator, which is used to calculate the branch metrics for every stage, is a single unit which is repeatedly used by all the stages of the trellis. The add-compare-select unit calculates the path metrics of all the states in a stage and the number of ACS units depends on the constraint length [SWA02]. The survivor memory unit is used to store the path history of all the surviving paths and is finally used to retrieve the original input sequence.

A Viterbi decoder attempts to reconstruct a path through a trellis diagram based on a potentially corrupted stream of data. This is accomplished by selecting the most likely path through the trellis. In the decoding process, legal transitions are considered much more likely than illegal transitions. Not all transitions between states are legal. For example there is no legal transition from state S0 to state S2.

When new input data arrives, a probability value is calculated for every path between two sets of states. The probability is determined by calculating the distance between state, which is the number of bits that would have to be incorrect for the path to be taken. For example, consider the state S0 in fig(5) [MEN98]. There are two possible

paths to this state: one from state S0 and one from state S2. These paths are highlighted in fig(5).



Fig(5) Possible transitions into state S0

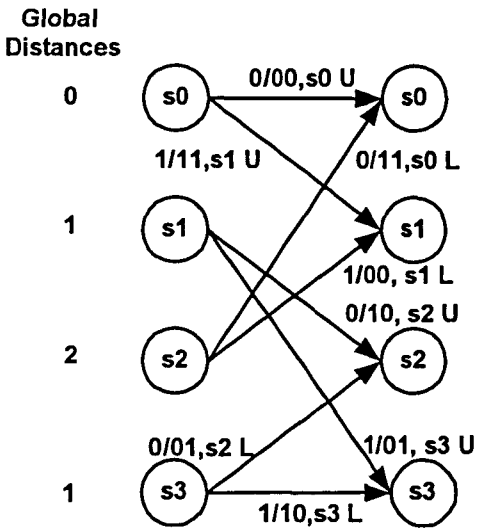
If the input pattern is "00", the distance for the upper branch is 0, since the pattern associated with this branch is the same as the input pattern. The distance associated with the lower branch is 2, since the pattern associated with the branch differs from the input pattern for both bits. In this manner, distances are calculated for all 8 possible branches.

These distances are added to a set of global distances that are maintained for each state [MEN98]. These values represent the likelihood of being in a particular state at the time the new input data arrives.

The global distances are added to the branch distances in order to select, for each state, the most likely branch that would reach that state. The most likely branch is the branch that has the lowest distance – a larger the distance number indicates that a

greater number of errors would have to occur for the path to be the correct one. This process is best illustrated by an example.

For example, given a set of current global distances and an input pattern, fig(6) shows the 8 branch distances, the 8 total distances, and the 4 selected branches [MEN98].



input pattern = 01

Branch Distances	Total distances (Branch+global)	Selected Branches
s0 upper (from 00) = 1	s0 upper = 1	
s0 lower (from 11) = 1	s0 lower = 3	upper
s1 upper (from 11) = 1	s1 upper = 1	
s1 lower (from 00) = 1	s1 lower = 3	upper
s2 upper (from 10) = 2	s2 upper = 3	
s2 lower (from 01) = 0	s2 lower = 1	lower
s3 upper (from 01) = 0	s3 upper = 1	
s3 lower (from 10) = 2	s3 lower = 3	upper

Fig(6) Branch selection based on global and branch distances

If the metrics of two paths are the same, then we choose one of them arbitrarily (based on chance).

Once the total distances have been calculated and branches have been selected, the branch selections are stored in a survivor memory.

The survivor memory stores the most likely branches for some number of previous input data. The number of previous branch selections stored in the survivor

memory is typically 4 to 5 times the constraint length. This is usually sufficient to adequately correct errors.

After the branches have been selected, the survivor memory is backtracked to determine the most likely error free input value. For each state, the survivor memory is used to determine the most likely previous state.

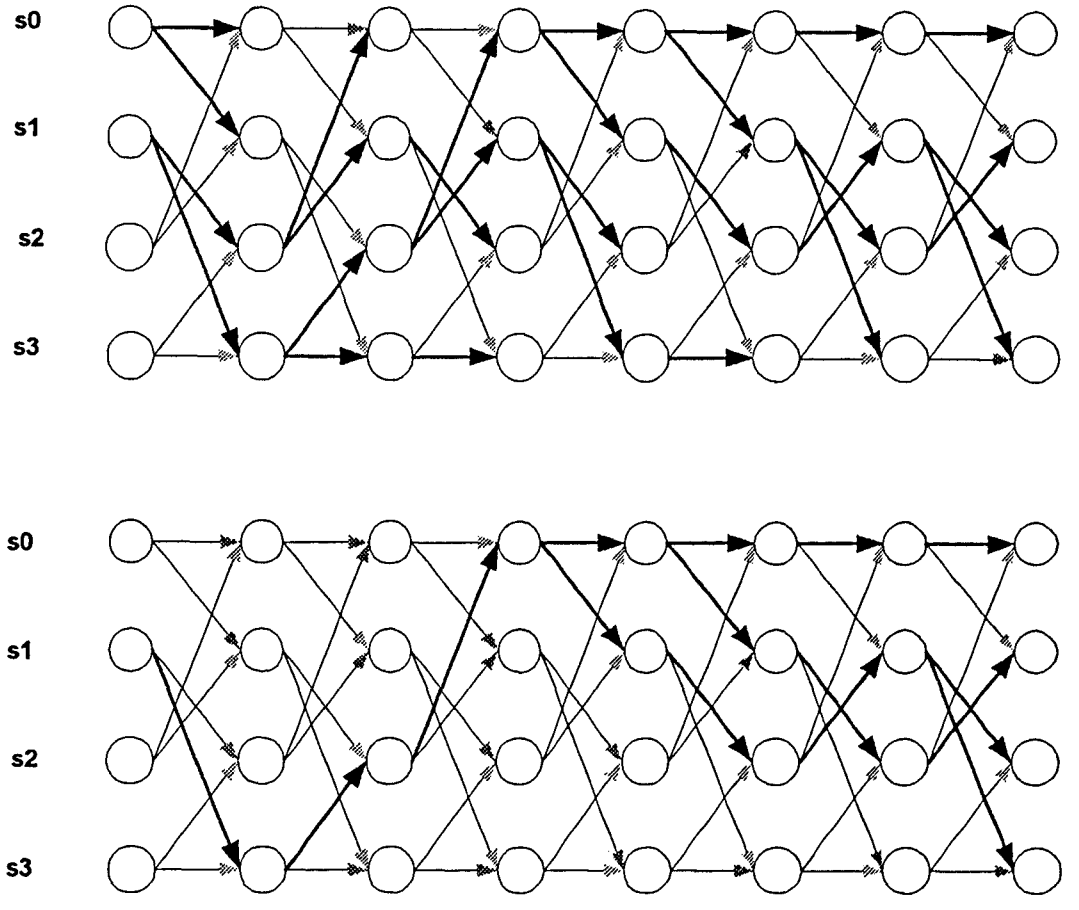
Since the survivor memory indicates the most likely branch that was used to get to each state, it similarly indicates the most likely previous state. The Viterbi algorithm asserts that the process of backtracking will tend to correct input sequences that contain errors because the paths tend to converge as the survivor memory is traversed.

The backtracking process continues until the beginning of the survivor memory is reached. Assuming the paths had all converged in one path, the direction of the last branch determines the decoded value. If the last branch was an upper branch, the output is '0', otherwise the output is '1'. If the paths have not converged, then some other decision-making process must be used to select the output value. At this point, we might see errors at the output [SCI02].

The backtracking process is best illustrated by an example. For simplicity, assuming a survivor window depth of 7. There are four values stored for each entry in the survivor window. These values represent the most likely path (upper or lower) that was used to reach that state. Backtracking begins at the end of the survivor window and uses this information to trace backwards.

Fig(7) shows two representations of a trellis [MEN98]. The bottom diagram shows the backtracked paths and how those paths tend to converge.

Fig(7) also shows that all the paths have converged by the forth step in backtracking. The final branch represents a lower branch, so the output would be a '1'.



Fig(7) Backtracking through the survivor window

Note that the backtracking process introduces a delay between the input data stream and the output data stream. This delay is equal to the depth of the survivor memory. In the previous example, the decoded output is delayed by 7 data samples.

After backtracking, the global distance values are updated to reflect the new total distances to be used for the next input data. Since the global distances represents a

running total that steadily increases as data arrives, the values must be periodically normalized to avoid overflow. This is accomplished by determining the smallest global distance and subtracting that value from all of the global distances.

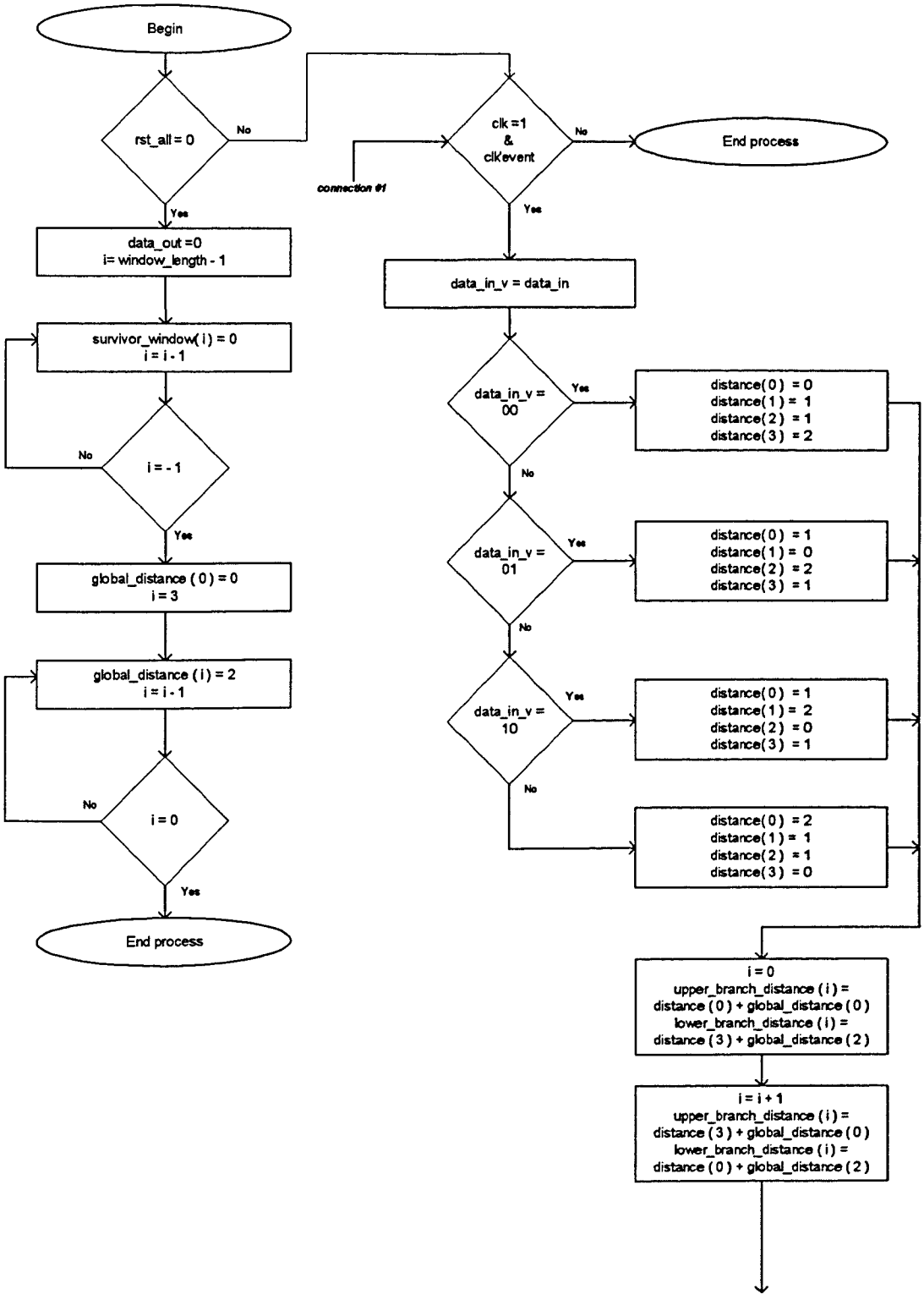
For example, if the global distances are 1,2,3, and 2, they would become 0,1,2, and 1 after normalization. Normalization maintains the relative difference between the values, but avoids overflow.

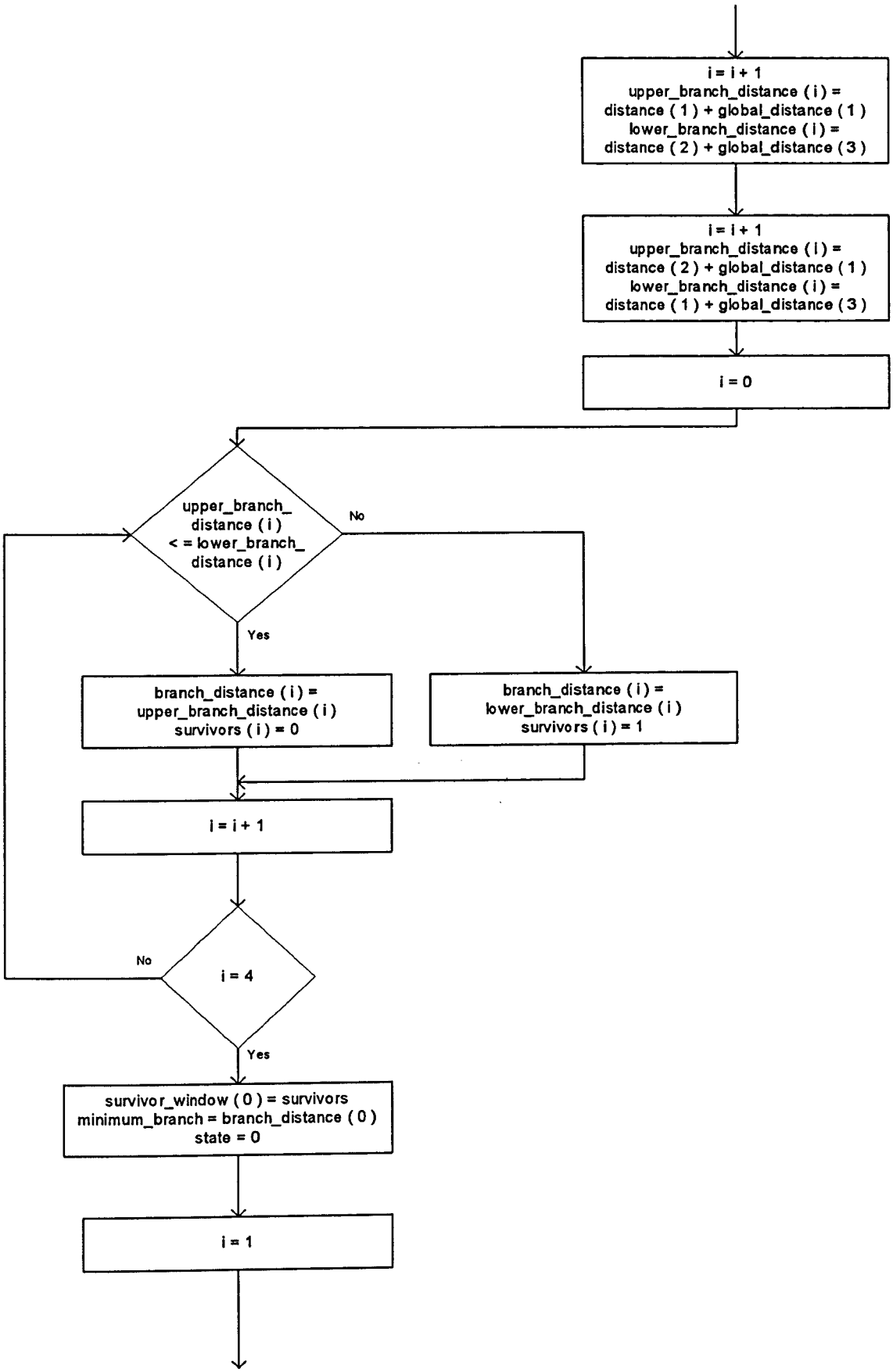
Finally, the contents of the survivor memory is shifted to the left in preparation for the next input data that will enter the survivor memory from the right.

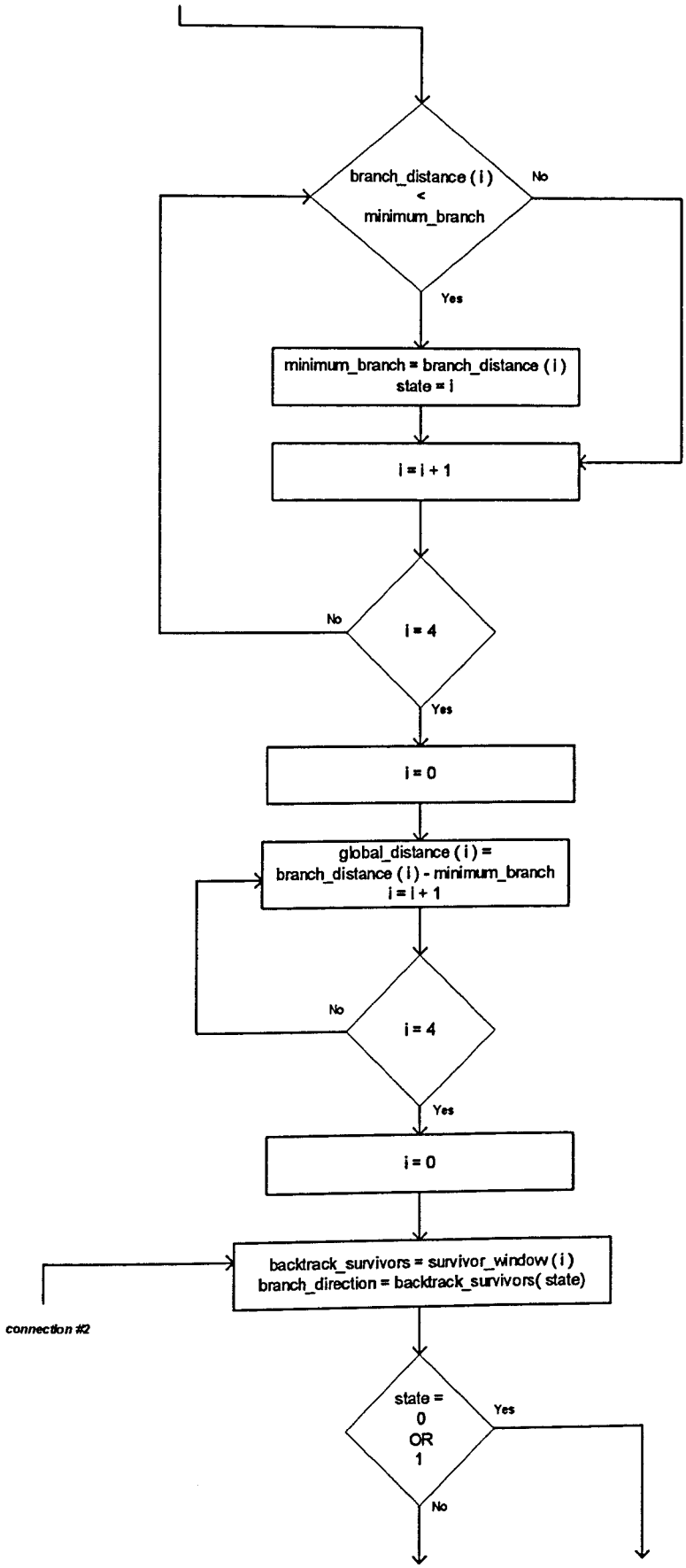
3.2 Viterbi Decoder Features:

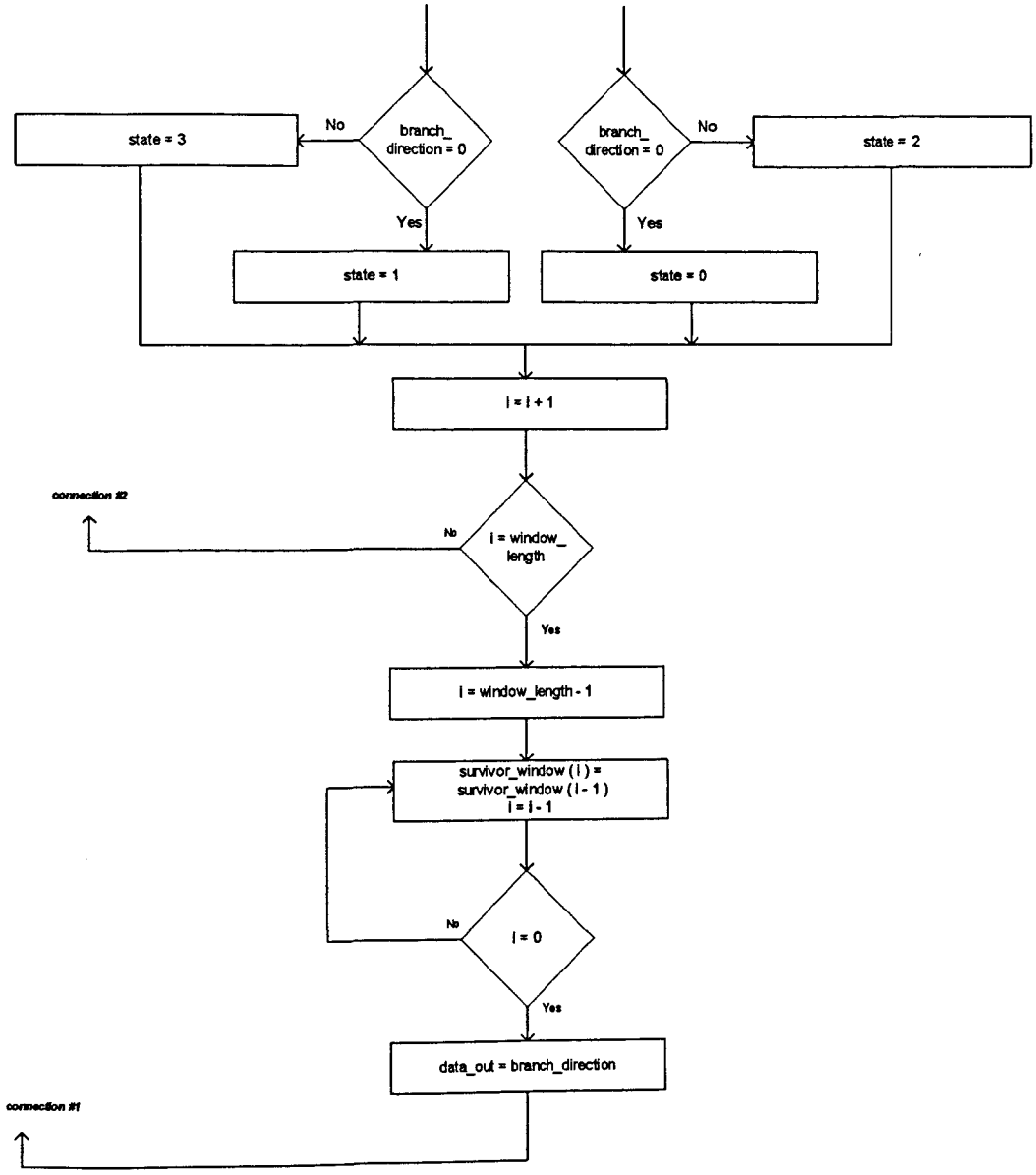
- Hard Decision Decoder: This means that the incoming data symbol will be '0' or '1', i.e. there are two quantization levels in A/D converter (normally the received data is analog, and it will be converted to digital). In a soft decision Viterbi decoder there are more than two quantization levels. Here we implemented the hard decision approach [XIL01].
- Trace-back method for survivor memory: This is the method to select the smallest path. Traceback method means that all the possible paths are stored in RAM, and this method selects the most likely path.
- Branch Metrics computations can be added for different applications: Branch metric computation makes the error calculation between the received symbol (2 bits in our case) and the symbol in the table. In hard-decision Viterbi, the Hamming distance is calculated.

3.3 Design Flow Chart




i = i + 1
upper_branch_distance (i) =
distance (1) + global_distance (1)
lower_branch_distance (i) =
distance (2) + global_distance (3)
i = i + 1
upper_branch_distance (i) =
distance (2) + global_distance (1)
lower_branch_distance (i) =
distance (1) + global_distance (3)
i = 0
upper_branch_
distance (i)
< = lower_branch_
distance (i)
No
Yes
branch_distance (i) =
upper_branch_distance (i)
survivors (i) = 0
branch_distance (i) =
lower_branch_distance (i)
survivors (i) = 1
i = i + 1
No
i = 4
Yes
survivor_window (0) = survivors
minimum_branch = branch_distance (0)
state = 0
i = 1


branch_distance (i)
<
minimum_branch
No
Yes
minimum_branch = branch_distance (i)
state = i
i = i + 1
No
i = 4
Yes
i = 0
global_distance (i) =
branch_distance (i) - minimum_branch
i = i + 1
No
i = 4
Yes
i = 0
backtrack_survivors = survivor_window (i)
branch_direction = backtrack_survivors(state)
connection #2
state =
0
OR
1
Yes
No


branch_ direction = 0
No
state = 3
Yes
state = 1
branch_ direction = 0
No
state = 2
Yes
state = 0
i = i + 1
connection #2
No
i = window_ length
Yes
i = window_length - 1
survivor_window (i) = survivor_window (i - 1)
i = i - 1
No
i = 0
Yes
data_out = branch_direction
connection #1

3.4 The VHDL code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity viterbi is
  port (
     clk        :in std_logic;
     rst_all    :in std_logic;
     data_in    :in std_logic_vector (1 downto 0);
     data_out   :out std_logic
    );

end viterbi;

architecture behavioral of viterbi is

begin

  process  (clk, rst_all)

    constant window_length : integer :=32;

    -- IEEE802.16 specifies a window length of 32

    subtype   survivor_elements is std_logic_vector (0 to 3);
    type      survivor_window_type is array (integer range <> ) of survivor_elements;
    variable  survivor_window     : survivor_window_type (window_length-1 downto 0);
    variable  survivors           : survivor_elements;
    variable  backtrack_survivors : survivor_elements;

    type      distance_array_type is array (0 to 3) of integer range 0 to 3;
    variable  distance            : distance_array_type;
    variable  global_distance     : distance_array_type;

    variable  data_in_v           : std_logic_vector (1 downto 0);

    type      branch_distance_array_type is array (0 to 3) of integer range 0 to 7;
    variable  upper_branch_distance : branch_distance_array_type;
    variable  lower_branch_distance : branch_distance_array_type;
    variable  branch_distance       : branch_distance_array_type;
    variable  minimum_branch        : integer range 0 to 7;

    subtype   state_type is integer range 0 to 3;
    variable  state                 : state_type;
    variable  branch_direction      : std_logic;

  begin

   if rst_all = '0' then
      data_out <='0';
      for i in window_length -1 downto 0 loop
```

```
    survivor_window(i) := (others => '0' );
    end loop;

    for i in 3 downto 1 loop
    global_distance(i) := 2;
    end loop;
    global_distance(0) := 0;

else

    if (clk'event and clk = '1' ) then

      data_in_v := data_in;

    -- calculate distances

    case data_in_v is
       when "00" =>
            distance (0) := 0;
            distance (1) := 1;
            distance (2) := 1;
            distance (3) := 2;
       when "01" =>
            distance (0) := 1;
            distance (1) := 0;
            distance (2) := 2;
            distance (3) := 1;
       when "10" =>
            distance (0) := 1;
            distance (1) := 2;
            distance (2) := 0;
            distance (3) := 1;
       when "11" =>
            distance (0) := 2;
            distance (1) := 1;
            distance (2) := 1;
            distance (3) := 0;
       when others =>
         null;
    end case;

    -- Add-compare-select (ACS)

    for i in 0 to 3 loop

     -- calculate distances for upper and lower branches

     case i is

          when 0 => upper_branch_distance(i) := distance(0) + global_distance(0);

                lower_branch_distance (i) := distance(3) + global_distance(2);

          when 1 => upper_branch_distance(i) := distance(3) + global_distance(0);

                lower_branch_distance (i) := distance(0) + global_distance(2);
```

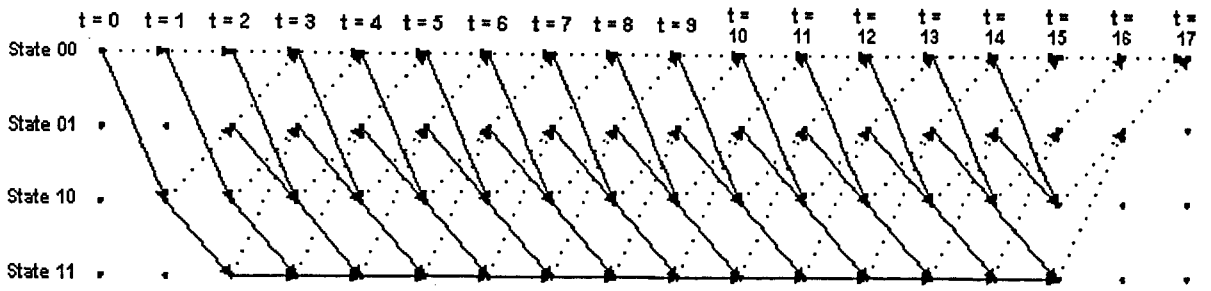
```
        when 2 => upper_branch_distance(i) := distance(1) + global_distance(1);

                lower_branch_distance (i) := distance(2) + global_distance(3);

        when 3 => upper_branch_distance(i) := distance(2) + global_distance(1);

                lower_branch_distance (i) := distance(1) + global_distance(3);


    end case;

    -- select the surviving branch and fill appropriate value
    -- into the survivor window

    if ( upper_branch_distance(i) <= lower_branch_distance(i) ) then

        branch_distance(i) := upper_branch_distance(i);
        survivors(i) := '0';

    else

        branch_distance(i) := lower_branch_distance(i);
        survivors(i) := '1';

    end if;

end loop;

    survivor_window(0) := survivors;

    -- find the minimum branch distance and the ending state

    minimum_branch := branch_distance(0);
    state :=  0 ;

    for i in 1 to 3 loop

        if (branch_distance(i) < minimum_branch) then
        minimum_branch := branch_distance(i);
        state := i;
        end if;

    end loop;

    -- to avoid overflow we substract the min branch

    for i in 0 to 3 loop

       global_distance(i) := abs (branch_distance(i) - minimum_branch);

    end loop;

    -- backtrack the survivor window

    for i in 0 to window_length -1 loop
```

```
            backtrack_survivors := survivor_window(i);
            branch_direction := backtrack_survivors( state );

            case state is

               when 0 | 1 => if ( branch_direction ='0' ) then
                               state := 0;
                             else
                               state := 2;
                             end if;
               when 2 | 3 => if ( branch_direction ='0' ) then
                               state := 1;
                             else
                               state := 3;
                             end if;
            end case;

          end loop;

          -- shifting the survivor window values to allow the next input
          -- to enter the trellis from the right hand

          for i in window_length -1 downto 1 loop

             survivor_window(i) := survivor_window(i - 1);

          end loop;

          -- output generation

          data_out <= branch_direction;

        end if;

      end if;

    end process;

end behavioral;
```

3.5 An Example:

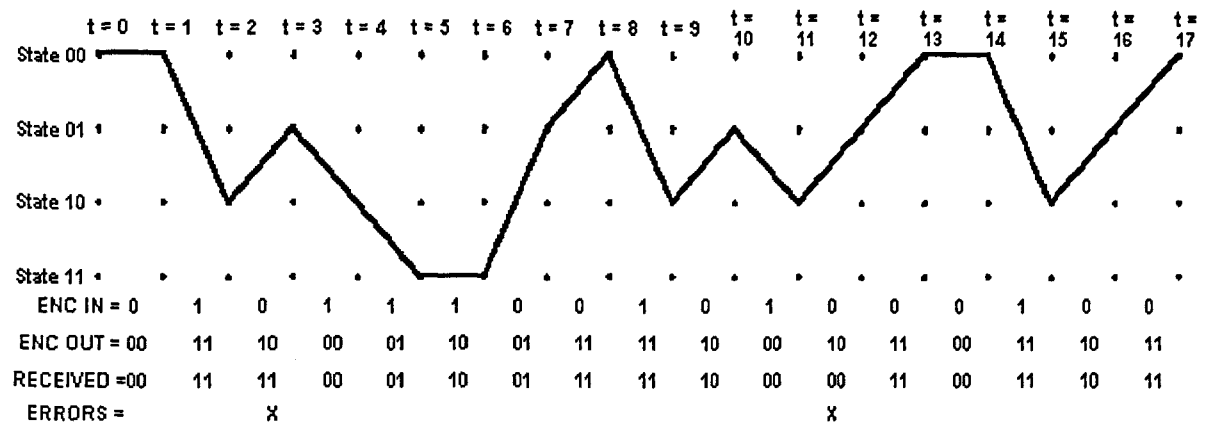
Now let's start looking at how the Viterbi decoding algorithm actually works [NETCOM]. The main goal of illustrating this example is to show that even if we received a wrong sequence at the input of the Viterbi decoder, we still get the correct

output sequence. We will review the error rate insertion matter in chapter (5). A trellis diagram is shown in fig(8).



Fig(8) A trellis diagram

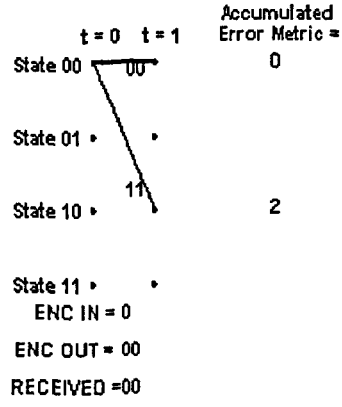
For our example, we're going to use hard-decision decoding. Suppose we received the following encoded message with a couple of bit errors:



Fig(9) Encoded message with a couple of bit errors

Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, and 11_2. That's because we know the convolutional encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These

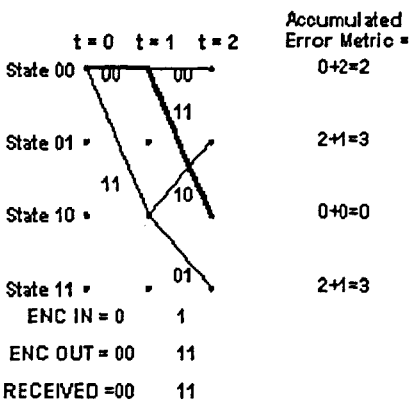
possible outputs of the encoder are 00_2 and 11_2. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics. At t = 1, we received 00_2. The only possible channel symbol pairs we could have received are 00_2 and 11_2. The Hamming distance between 00_2 and 00_2 is zero. The Hamming distance between 00_2 and 11_2 is two. Therefore, the branch metric value for the branch from State 00_2 to State 00_2 is zero, and for the branch from State 00_2 to State 10_2 is two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00_2 and for State 10_2 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. The figure below illustrates the results at t = 1:



Fig(10) The trellis diagram between t = 0 and t = 1

At each time instant t, we will store the number of the previuos state that led to each of the current states at t.

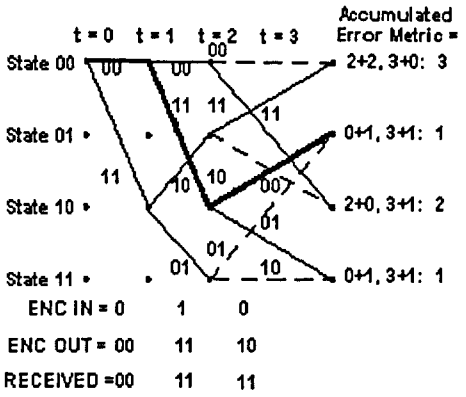
Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 going from State 10_2 to State 11_2. The Hamming distance between 00_2 and 11_2 is two, between 11_2 and 11_2 is zero, and between 10_2 or 01_2 and 11_2 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At t = 1, we could only be at State 00_2 or State 10_2. The accumulated error metric values associated with those states were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.



Fig(11) The trellis diagram between t = 0 and t = 2

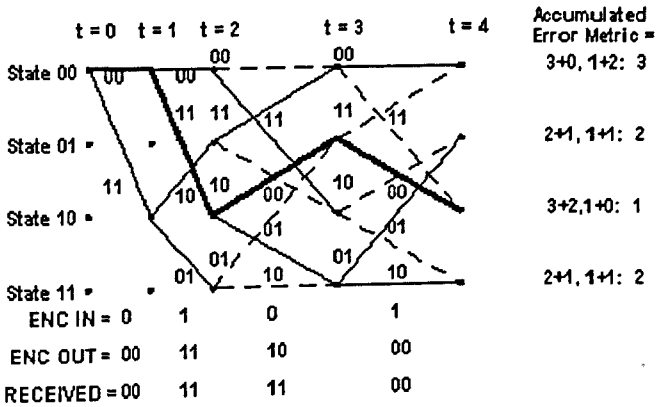
Now lets look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. We compare the accumulated

error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state.



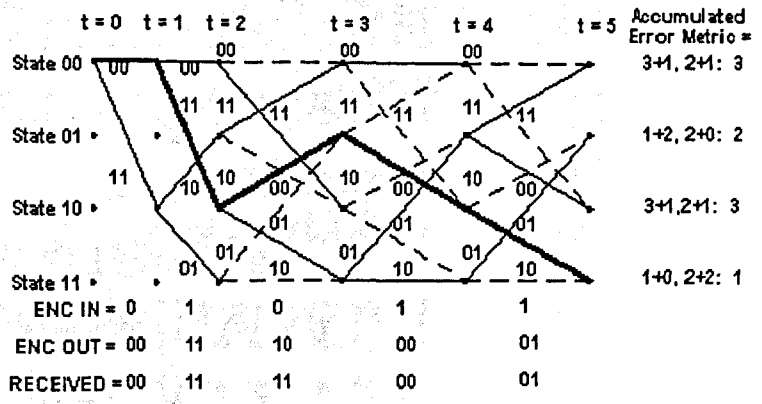
Fig(12) The trellis diagram between t = 0 and t = 3

Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these. Let's see what happens now at t = 4 [NETCOM]. The processing is the same as it was for t = 3. The results are shown in the figure:



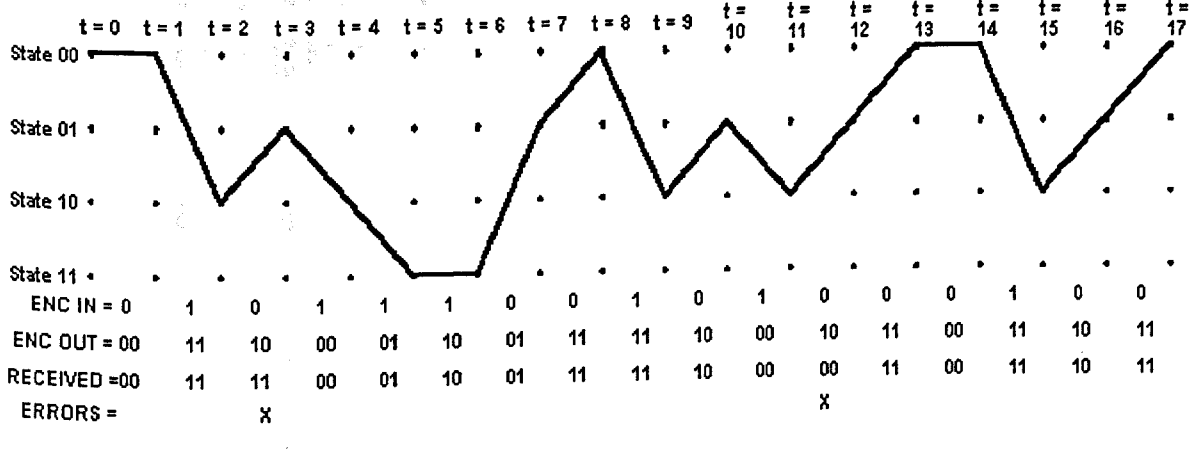
Fig(13) The trellis diagram between t = 0 and t = 4

Notice that at t = 4, the path through the trellis of the actual transmitted message, shown in bold, is again associated with the smallest accumulated error metric. Let's look at t = 5:



Fig(14) The trellis diagram between t = 0 and t = 5

At t = 5, the path through the trellis corresponding to the actual message, shown in bold, is still associated with the smallest accumulated error metric. This is the thing that the Viterbi decoder exploits to recover the original message. Now, Let's skip to the end. At t = 17, the trellis looks like this, with the clutter of the intermediate state history removed:



Fig(15) The trellis diagram showing the whole message

Here's an insight into how the traceback algorithm eventually finds its way onto the right path even if it started out choosing the wrong initial state. This could happen if more than one state had the smallest accumulated error metric, for example. We will use the figure for the trellis at t = 3 again to illustrate this point. We saw how at t = 3, both States 01_2 and 11_2 had an accumulated error metric of 1. The correct path goes to State 01_2 -notice that the bold line showing the actual message path goes into this state. But suppose we choose State 11_2. The previous state for State 11_2 , which is State 10_2 , is the same as the previous state for State 01_2! This is because at t = 2, State 10_2 had the smallest accumulated error metric. So after a false start, we are almost immediately back on the correct path. For the 15-bit message example, we built the trellis up for the entire message before starting traceback. For longer messages, or continuous data, this is neither practical or desirable, due to memory constraints and decoder delay. Research has shown that a traceback depth of (K x 5) is sufficient for Viterbi decoding with the type of codes we have been discussing [MEN98]. Any deeper traceback increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. The exception is punctured codes, which is not in our scope. They require deeper traceback to reach their final performance limits.

According to IEEE 802.16 specifications [ALT01], the traceback depth should equal 32, and that is more than (K x 5) = 15. Here, we tried to follow the specifications. We also compiled our design with a traceback of 16, and it was sufficient for correcting the errors.

CHAPTER 4

FIELD PROGRAMMABLE GATE ARRAYS (FPGAs)

4.1 What is an FPGA?

Before the advent of programmable logic, custom logic circuits were built at the board level using standard components, or at the gate level in expensive application-specific integrated circuits. The FPGA is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. Each logic cell can independently take on any one of a limited set of personalities. The individual cells are interconnected by a matrix of wires and programmable switches. A user's design is implemented by specifying the simple logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells and interconnect form a fabric of basic building blocks for logic circuits. Complex designs are created by combining these basic blocks to create the desired circuit [XILINX].

4.2 What does a logic cell do?

The logic cell architecture varies between different device families. Generally speaking, each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a Boolean logic function specified in the user program. In most families, the user also has the option of registering the combinatorial output of the cell, so that clocked logic can be easily implemented. The cell's combinatorial logic may be physically implemented as a small look-up table memory (LUT) or as a set of

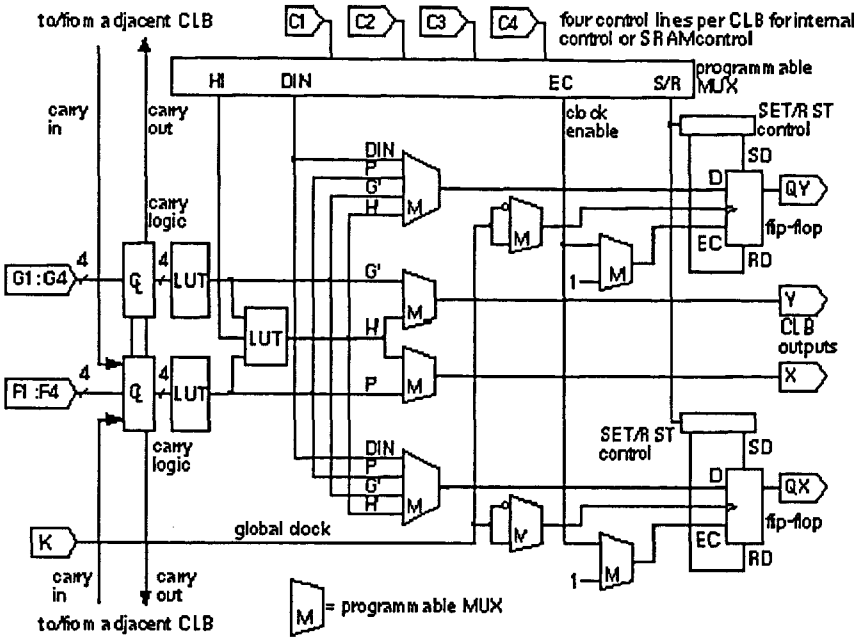
multiplexers and gates. LUT devices tend to be a bit more flexible and provide more inputs per cell than multiplexer cells at the expense of propagation delay.

4.3 What does 'Field Programmable' mean?

Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. A typical integrated circuit performs a particular function defined at the time of manufacture. In contrast, the FPGA's function is defined by a program written by someone other than the device maufacturer. Depending on the particular device, the program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up. This user programmability gives the user access to complex integrated designs without the high engineering costs associated with application specific integrated circuits.

4.4 Xilinx XC4000 family Logic Block

Figure (16) shows the CLB (Configurable Logic Block) used in the XC4000 series of Xilinx FPGAs. This is a fairly complicated basic logic cell containing 2 four-input LUTs that feed a three-input LUT. The XC4000 CLB also has special fast carry logic hard-wired between CLBs. MUX control logic maps four control inputs (C1–C4) into the four inputs: LUT input H1, direct in (DIN), enable clock (EC), and a set / reset control (S/R) for the flip-flops. The control inputs (C1–C4) can also be used to control the use of the F' and G' LUTs as 32 bits of SRAM.



Fig(16) The Xilinx XC4000 family CLB

The Xilinx CLB is the functional element from which user logic is constructed in an FPGA. I/O blocks (IOB) serve as the interface between the external package pin of the device and the internal user logic.

4.5 More about XC4005XL board

The XS4005XL Board is perfect for experimenting with FPGA designs, microcontroller programming, or hardware/software codesign [XESS]. The 9,000-gate XC4005XL FPGA operates at 5V so we can connect it to commonly available TTL chips. Digital logic designs can be loaded into the FPGA. The microcontroller can use the FPGA as a coprocessor. The 32-KByte SRAM can store microcontroller programs/data or serve as general-purpose storage for FPGA-based designs.

4.6 How to program the FPGA board?

Individually defining the many switch connections and cell logic functions would be a hard task. Fortunately, this task is handled by special software. The software translates a user's schematic diagrams or textual hardware description language code then places and routes the translated design. Most of the software packages have hooks to allow the user to influence implementation, placement and routing to obtain better performance and utilization of the device [XILINX].

Using the appropriate software tool, a (.bit) file was generated. This file contains all the information that the FPGA needs to be programmed. Using XSTOOLS software package, we should drag the (.bit) file into the GXSLOAD area. Also, In the "build" process, we should specify the pin assignment file, which defines the pins that the FPGA will use to connect to the outside world. Our FPGA has 84 pins. The pin assignment file is stored as (.ucf) file. Our (.ucf) file is shown below:

```
net clk loc =p13;
net rst_all loc=p44;
net data_in(0) loc=p45;
net data_in(1) loc=p46;
net data_out loc=p25;
```

This file means that the clock input is pin number 13 of the PFGA, and the rst_all pin is number 44. Also, the encoded data will enter the FPGA through pins 45 and 46, and the output is pin number 25 of the FPGA.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 What did we do?

We used VHDL hardware description language to implement the algorithm [SKA96]. We also used OrCAD Capture V9.1 to implement and find the simulation results. In our design, we tried to meet with IEEE 802.16 standard. IEEE 802.16 standard specifies the air interface of fixed point-to-multipoint broadband wireless access (BWA) systems providing multiple services. This standard is intended to enable rapid worldwide deployment of broadband wireless access products. The new IEEE 802.16 specifications requires a Viterbi decoder with constraint length of 3, traceback length of 32, and minimum throughput requirement of 44.8 Mbps.

Our specifications are a constraint length of 3, traceback length of 32, and a throughput of 45.4 Mbps.

We wrote down a random bit stream of data, then using the generator polynomials mentioned in chapter 2, we encoded these bits to get a randomly encoded input for our design. Random errors were then inserted into the encoded data stream. This represents the errors that might occur during transmission of the data through the wires or through the airwaves. The data stream, with errors, was processed by the Viterbi decoder. Finally, we compared the results to check whether or not the output of our Viterbi decoder (data_out) is the same as the original signal.

5.2 Using OrCAD Capture to implement Viterbi Decoder:

The Build tool automates the translation of schematics and/or VHDL into the programming files required for the programmable device and provides an interface to Xilinx Alliance Series software. Placement and routing may be run at anytime after design entry is complete. The Build tool detects source VHDL or schematic files that may be "out of date" relative to the post-route simulation model file and if necessary, reruns logic synthesis to refresh the EDIF 2 0 0 (Electronic Data Interchange Format) netlist [ORCAD].

The Build tools uses the settings specified by the dialog box to build a Xilinx Alliance Series command file (with the extension .CMD) and series of command lines for the Xilinx Alliance Series NGDBuild program. The Build tool creates a subdirectory called TIMED below the subdirectory of the OrCAD project file (OPJ), copies all EDIF 2 0 0 netlist files from the COMPILED subdirectory, then runs NGDBuild. All results and reports are saved into the TIMED subdirectory and referenced by the project.

Each Xilinx place and route stage automated by the Build tool is described here:

Translate – converts the EDIF 2 0 0 files created by logic synthesis into a binary database used by the Xilinx software. This option causes the build process to stop after the Xilinx EDIF2NGD netlist translation program and writes one or more NGO files into the .\TIMED subdirectory. The translation report is saved as a .BLD file.

MAP – structures the incoming logical design to the target device we specify on the Part Type field of the NGDBuild tab. This option causes the build process to stop

after the Xilinx MAP program allocates CLBs, IOBs, and other Xilinx FPGA resources to logical elements of a design. Results are reported in the MRP file.

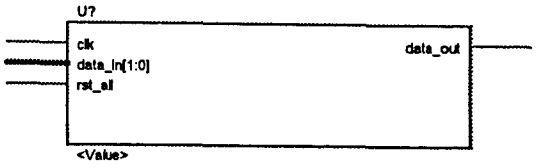
Place and Route – places and routes the physical design. This option causes the build process to stop after the Xilinx PAR program optimizes the layout using the timing constraints we may specify. Results of the timing analysis performed by PAR are available in the TIMED.DLY file and pin assignments in the TIMED.PAD file.

Re-entrant Place and Route – runs the Xilinx PAR program independently of Translate, MAP, and Implement. This option is useful to experiment with logic optimization and guide modes to arrive at a superior implementation.

Implement – generates a bitstream (.bit) file, which is required to program the FPGA. This option causes the build process to stop after the Xilinx BitGen program creates a configuration bitstream from the fully routed NCD (Native Circuit Description) created by PAR. Results from the BitGen run are available in the BGN and DRC files.

The Build tools uses the Xilinx Alliance Series program NGD2VHDL to create simulation models for timing simulation and adds them to the Timed folder of the Simulation Resources after place and route. The files required for a timing simulation include the netlist and standard delay format (with extension .SDF) file. Delay characteristics and performance rules of the SDF are applied by the simulator to emulate the propagation delays introduced by routing and logic block delays.

At this stage we simulate the post-place and route state of the project (Timed) to confirm that the PLD will meet our performance specifications. We apply input stimuli to the design. Our Viterbi decoder looked as follows after gererating the part:



Fig(17) Our Virerbi decoder symbol

The applied random stream of data is shown below as a table:

Time (ns)	0...	82	104	126	148	170	192	214	236	258	280	302	324	346	368
Transmitted Data	0	0	1	0	1	1	1	0	0	1	0	1	0	0	0
Encoded Data	0 0	0 0	1 1	1 0	0 0	0 1	1 0	0 1	1 1	1 1	1 0	0 0	1 0	1 1	0 0

cont.

Time (ns)	390	412	434	456	478	500	522	544	566	588	610	632	654	676	698
Transmitted Data	1	1	0	1	1	1	0	0	0	0	1	1	0	1	1
Encoded Data	1 1	0 1	0 1	0 0	0 1	1 0	0 1	1 1	0 0	0 0	1 1	0 1	0 1	0 0	0 1

cont.

Time (ns)	720	742	764	786	808	830	852	874	896	918	940	962	984	1006	1028
Transmitted Data	1	0	1	0	0	1	1	0	0	0	1	1	0	1	1
Encoded Data	1 0	0 1	0 0	1 0	1 1	1 1	0 1	0 1	1 1	0 0	1 1	0 1	0 1	0 0	0 1

cont.

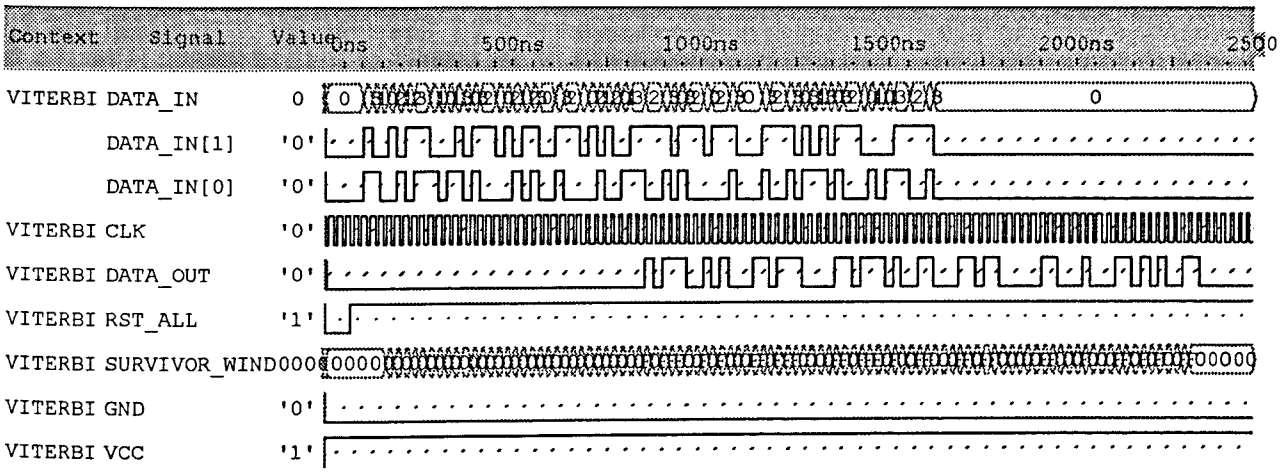
Time (ns)	1050	1072	1094	1116	1138	1160	1182	1204	1226	1248	1270	1292	1314	1336	1358
Transmitted Data	0	0	0	0	0	1	1	0	0	0	1	0	0	0	1
Encoded Data	0 1	1 1	0 0	0 0	0 0	1 1	0 1	0 1	1 1	0 0	1 1	1 0	1 1	0 0	1 1

cont.

Time (ns)	1380	1402	1424	1446	1468	1490	1512	1534	1556	1578	1600	1622
Transmitted Data	1	0	1	0	1	0	0	1	1	0	0	0
Encoded Data	0 1	0 1	0 0	1 0	0 0	1 0	1 1	1 1	0 1	0 1	1 1	0 0

Table 5.1 The applied random stream of data

The clock period was set to 22ns (45.4 MHz). Also the rst_all input was set to be 0 until 70ns to reset the system, then we forced it to be 1 after that time to skip the reset operation. The applied random stream of data is shown below with the most important signals in the system:



Fig(18) The simulation result after applying the random stream of data

As it appears, the output data is delayed by the depth of the memory. The first bit to come out from the traceback window is at time 807ns.

[32 (traceback window) +1 (clock cycle to generate the output)] x 22 (period) +76ns (delay to allow reset) + 5ns (output delay shown in fig(20)) = 807ns. Another graph showing how the data moves through the survivor window:

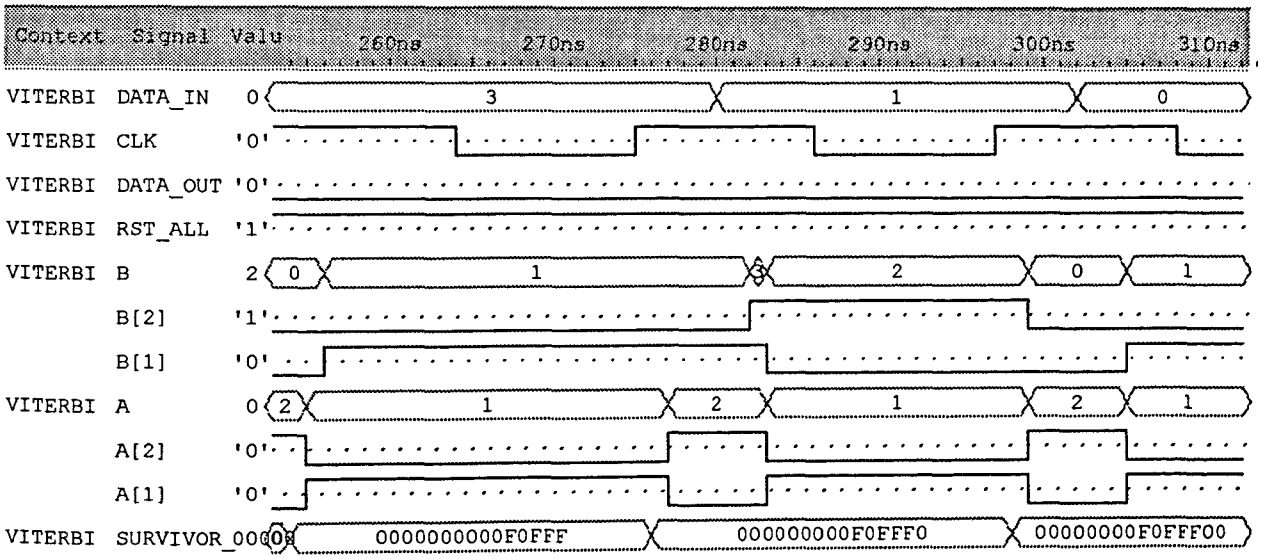
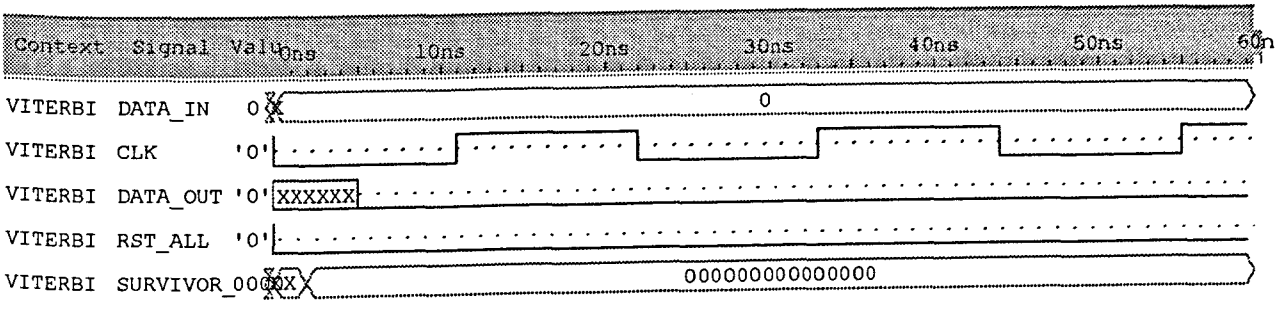


Fig (19) Data moves in the survivor window

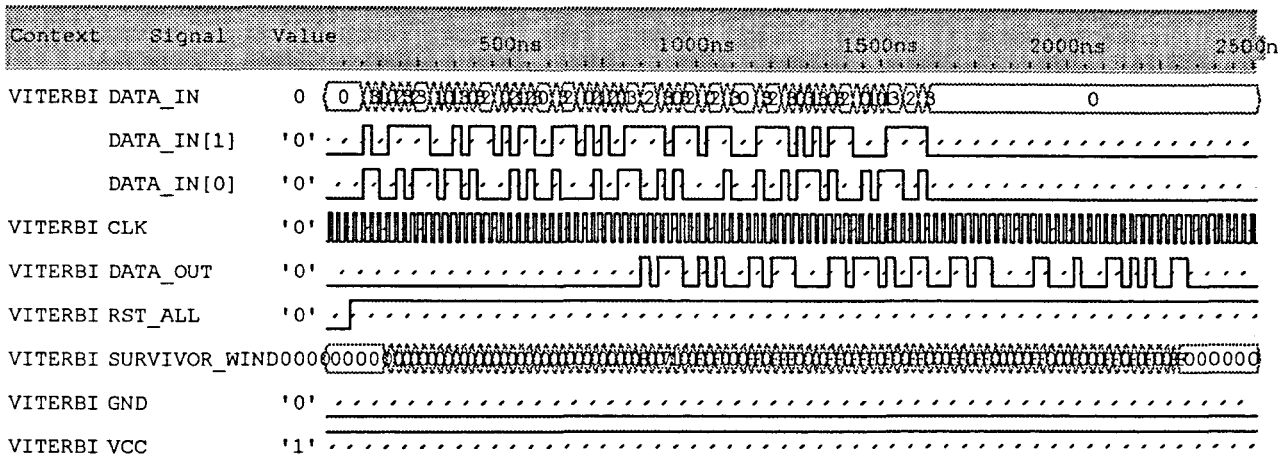
Now showing the delay for the whole system:



Fig(20) The delay for the whole system

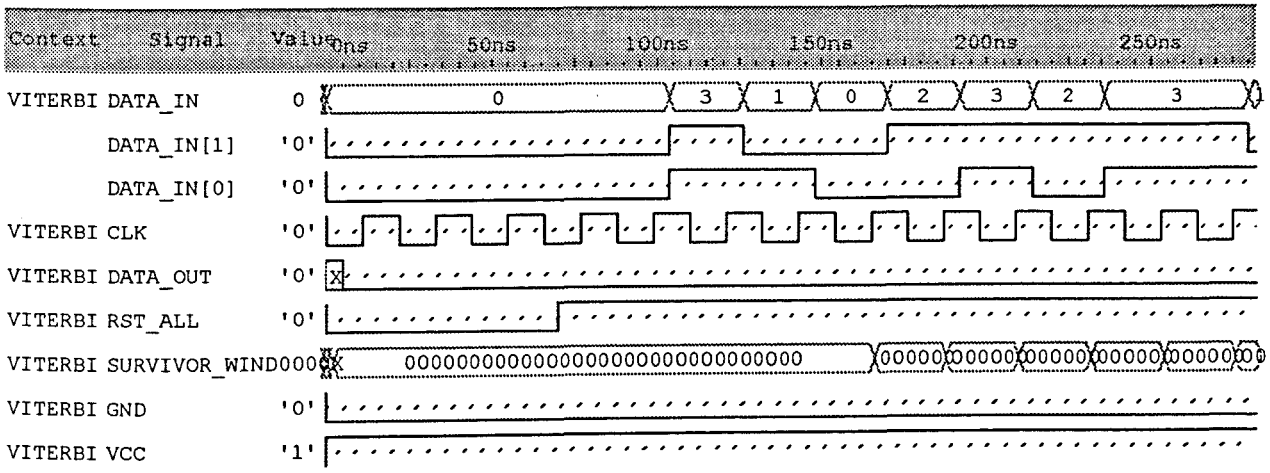
We did the following cases:

1. Inserting a noise to the signal (Data_in(1)) at 192ns (from 0 to 1):



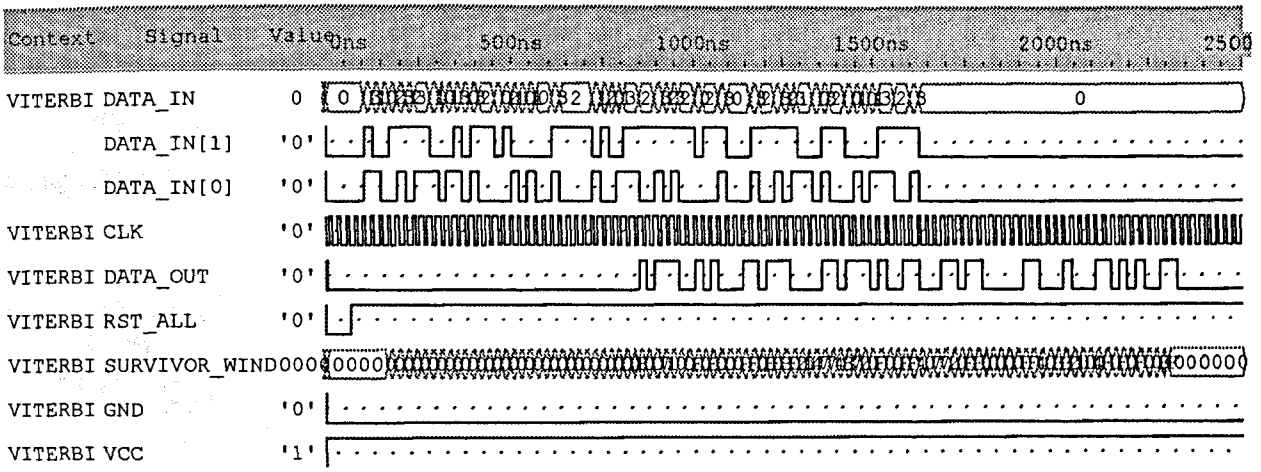
Fig(21) A noise inserted at time 192ns

We observe no change at the output. Zooming the change that we did:



Fig(22) Showing the previous change

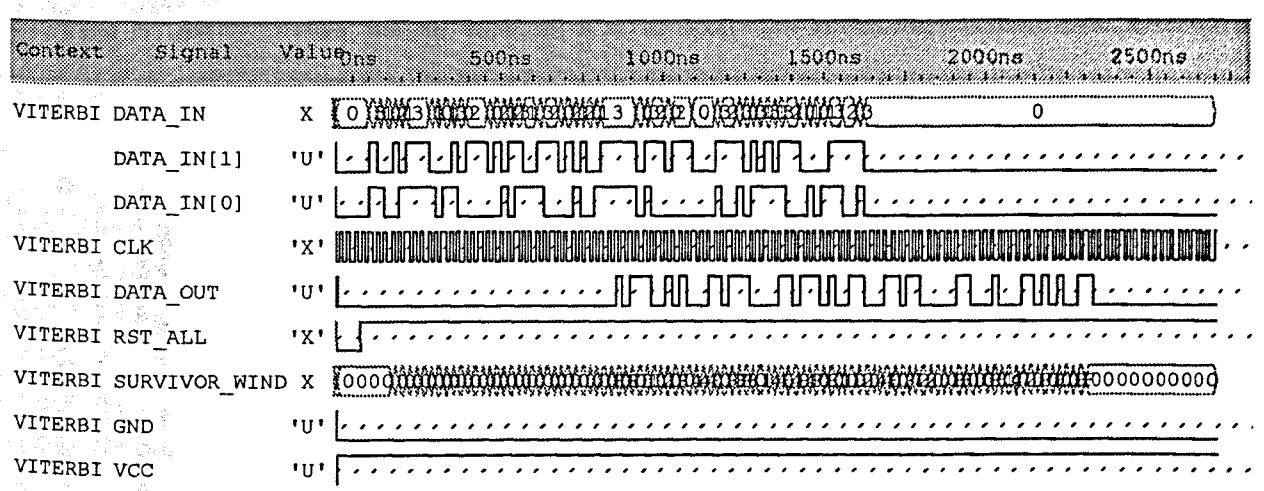
2. Changing the status at the following times for data_in (1) : To 1 @ 192ns, to 0 @ 522ns, to 1 @ 676ns, to 1 @ 918ns, to 1 @ 1248 ns.



Fig(23) Inserting more noise

We observe no change at the output.

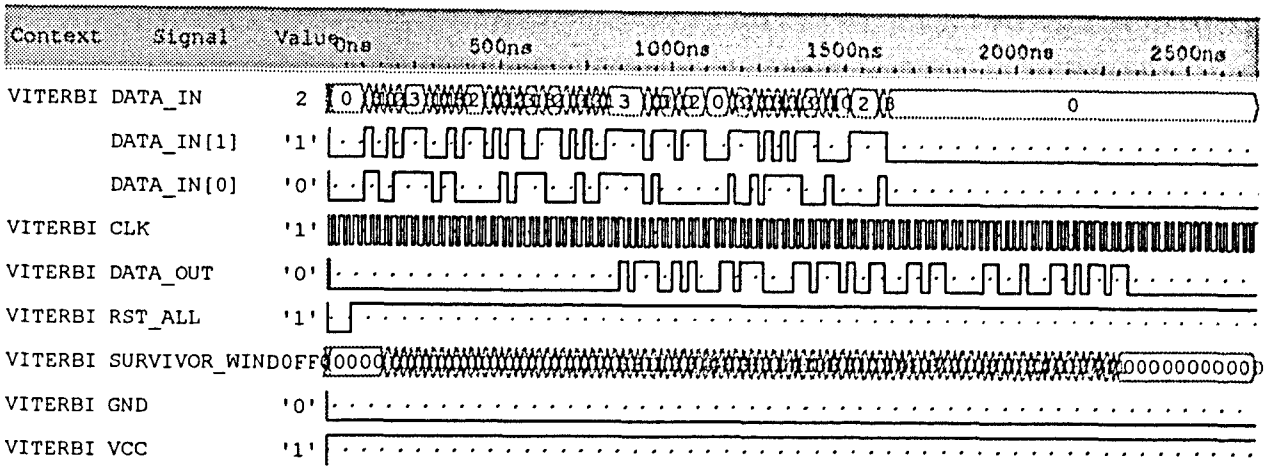
3. Now, inserting up to 8 errors in data_in (0) by Changing the status at the following times : To 1 @ 214ns, to 0 @ 390ns, to 1 @ 566ns, to 1 @ 588ns, to 1 @ 852ns, to 1 @ 874ns, to 0 @ 1072ns, to 1 @1336ns.



Fig(24) Inserting more noise

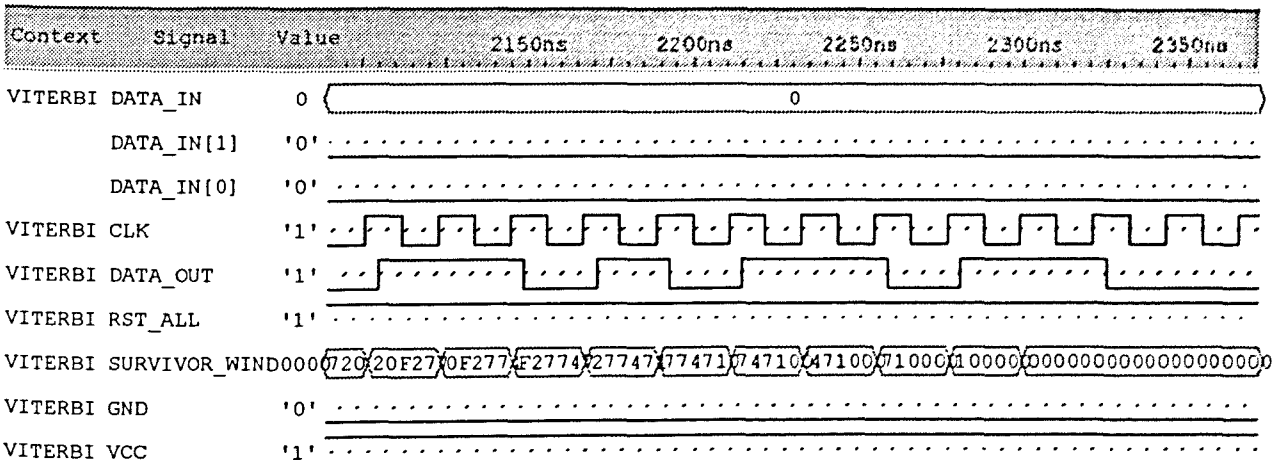
As we can see the output signal still keeping its shape, even though we changed the input signal.

4. Now, Adding 3 more errors to the previous change ; to 0 @ 1490ns, to 0 @ 1512ns, to 0 @ 1534ns (Total number of errors introduced = 11 error). In this case we observe a 1 bit change in the output at time 2237ns.



Fig(25) Observing a change at the output

Zooming in the change that happened to the output signal:



Fig(26) Showing the change that happened at the output

As can be seen above in the simulation samples, the Viterbi decoder does an adequate job at correcting errors that were introduced into the data stream. The error rate in the last sample was set to more than 15% (= 15.27%), so for 72 data points, 11 errors were introduced. Of those errors, all but 1 were fixed. This means that 90.9% of the errors that were introduced could be fixed by Viterbi decoding. Of course, different error rates would produce different results as can be seen in the following table:

No. of data points	No. of errors introduced	Error Rate	No. of errors at the output	Accuracy %
72	1	1%	0	100
72	2	3%	0	100
72	3	4%	0	100
72	4	6%	0	100
72	5	7%	0	100
72	6	8%	0	100
72	7	10%	0	100
72	8	11%	0	100
72	9	12.50%	0	100
72	10	13.80%	0	100
72	11	15%	1	90.9

Table 5.2 Our final optimum results

Some of the output files in the (Timed) directory are shown below:

Timed.dly

```
Wed Mar 20 02:50:53 2002
```

The 20 Worst Net Delays are:

Max Delay (ns)	Netname
5.972	nx3706
5.929	nx3702
5.760	nx3704
5.140	nx3708
4.893	D_dup_4473
4.788	D_dup_4467
4.493	a(2)_dup_3400
4.260	rst_all_int
4.135	a(1)_dup_3353
3.890	global_distance(2)(1)
3.672	a(0)_dup_3502
3.671	global_distance(3)(0)
3.506	survivor_window(1)(1)
3.432	global_distance(1)(0)
3.431	global_distance(2)(0)
3.343	global_distance(0)(0)
3.320	a(2)_dup_3364
3.314	a(1)_dup_3764
3.296	D_dup_4470
3.107	b(0)_dup_3417

viterbi.drc

DRC detected 0 errors and 0 warnings.

Viterbi.mrp

```
Xilinx Mapping Report File for Design 'viterbi'
Copyright (c) 1995-1999 Xilinx, Inc.  All rights reserved.

Design Information
------------------
Command Line   : m1map -cm area -gm exact -oe normal -p 4005XLPC84 VITERBI
Target Device  : x4005xl
Target Package : pc84
Target Speed   : -09
Mapper Version : xc4000xl -- C.16

Design Summary
--------------
   Number of errors:        0
   Number of warnings:      4
   Number of CLBs:            123 out of   196   62%
      CLB Flip Flops:     132
      CLB Latches:          0
      4 input LUTs:       123
      3 input LUTs:       106 (60 used as route-throughs)
   Number of bonded IOBs:       5 out of    65    7%
      IOB Flops:            1
      IOB Latches:          0
   Number of clock IOB pads:    1 out of    12    8%
   Number of BUFGLSs:           1 out of     8   12%
   Number of startup:           1 out of     1  100%
Total equivalent gate count for design: 1743
Additional JTAG gate count for IOBs:    240
```

Timed.pad

PAR: Xilinx Place And Route C.16.

Wed Mar 20 02:50:54 2002

Xilinx PAD Specification File

****

Input file: VITERBI.ncd

Output file: Timed.ncd

Part type: xc4005xl

Speed grade: -09

Package: pc84

Wed Mar 20 02:50:54 2002

Pinout by Pin Name:

Pin Name	Direction	Pin Number
clk	INPUT	P13
data_in(0)	INPUT	P45
data_in(1)	INPUT	P46
data_out	OUTPUT	P25
rst_all	INPUT	P44

Dedicated or Special Pin Name	Pin Number
/PROG	P55
CCLK	P73

DONE	P53
GND	P76
GND	P31
GND	P64
GND	P1
GND	P43
GND	P12
GND	P21
GND	P52
M0	P32
M1	P30
M2	P34
TDO	P75
TDO	TDO
VCC	P22
VCC	P2
VCC	P42
VCC	P11
VCC	P63
VCC	P33
VCC	P74
VCC	P54

viterbi.pcf

```
SCHEMATIC START ;
// created by map version C.16 on Wed Mar 20 02:49:47 2002
COMP "rst_all" LOCATE = SITE "P44" LEVEL 1;
COMP "data_out" LOCATE = SITE "P25" LEVEL 1;
COMP "data_in(1)" LOCATE = SITE "P46" LEVEL 1;
COMP "data_in(0)" LOCATE = SITE "P45" LEVEL 1;
```

```
COMP "clk" LOCATE = SITE "P13" LEVEL 1;
SCHEMATIC END ;
```

Viterbi.pin

```
-- Xilinx signal/pin map file produced by ngd2vhdl, version C.16
-- Options: -w -pf -ti UUT
-- Date: Wed Mar 20 02:51:55 2002
-- Part 4005xl
-- Package pc84
-- Speed -09

CLK                     INPUT       13
RST_ALL                 INPUT       44
DATA_IN(1)              INPUT       46
DATA_IN(0)              INPUT       45
DATA_OUT                OUTPUT      25
```

Viterbi.sum

```
*****************************************************

Cell: viterbi    View: behavioral    Library: work

*****************************************************

Total accumulated area :
 Number of FG Function Generators :    119
 Number of H Function Generators :      46
 Number of Packed CLBs :                60
```

```
Number of CLB Flip Flops :            132
Number of STARTUP :                     1
Number of IBUF :                        3
Number of IOB Output Flip Flops :       1

Number of ports :                       5
Number of nets :                      311
Number of instances :                 308
Number of references to this view :     0
```

Cell	Library	References	Total Area
F1_LUT	xi4xl	4 x 1	4 F1_LUT
F3_LUT	xi4xl	86 x 1	86 FG Function Generators
H3_LUT	xi4xl	46 x 1	46 H Function Generators
F4_LUT	xi4xl	31 x 1	31 FG Function Generators
F2_LUT	xi4xl	2 x 1	2 FG Function Generators
BUFG	xi4xl	1 x 1	1 BUFG
FDPE	xi4xl	3 x 1	3 CLB Flip Flops
FDCE	xi4xl	129 x 1	129 CLB Flip Flops
OFDX	xi4xl	1 x 1	1 IOB Output Flip Flops
INV	xi4xl	1 x 1	1 INV
IBUF	xi4xl	3 x 1	3 IBUF
STARTUP	xi4xl	1 x 1	1 STARTUP

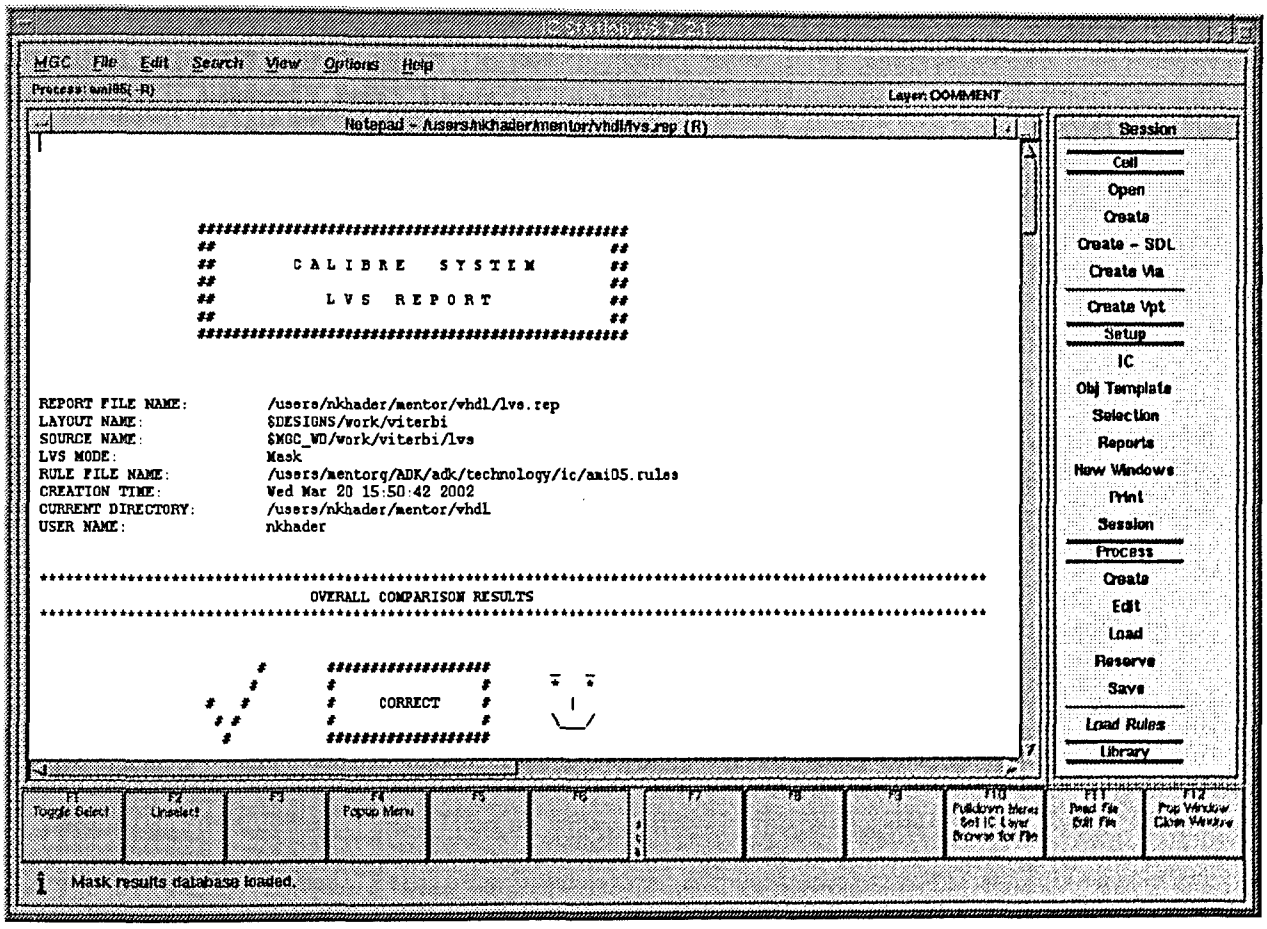
```
************************************************
Device Utilization for 4005xlPC84
************************************************
Resource                Used    Avail    Utilization
------------------------------------------------
IOs                     5       61        8.20%
```

```
FG Function Generators  119     392     30.36%
H Function Generators   46      196     23.47%
CLB Flip Flops          132     616     21.43%

-----------------------------------------------

Using default wire table: 4000xl-default
```

Viterbi.ucf

```
net clk loc=p13;
net rst_all loc=p44;
net data_in(0) loc=p45;
net data_in(1) loc=p46;
net data_out loc=p25;
```

We also used Mentor Graphics software to generate schematic and layout for our Viterbi decoder. Fig(27) illustrates the IC station LVS check result.



Fig(33) IC-station LVS check result

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this work, we implemented a Viterbi decoder into an FPGA under IEEE 802.16 specifications. A noisy transmission channel causes bit errors at the receiver. The Viterbi algorithm finds the most likely sequence of bits that is closest to the actual received sequence. The Viterbi decoder uses the redundancy, which the convolutional encoder imparted, to decode the bit stream and remove the errors.

With the increasing need for the transmission of digital data over a wide variety of mediums, the need for error control coding will increase significantly as well. Viterbi coding provides a robust error control method for many common types of data transfer mediums, particularly those that are one-way or that are noisy and sure to produce errors. Error Control Codes are already the standard for a large number of satellite transmission specifications as well as for compact disc technology. It is unlikely that the needs will decrease in our increasing technological society. The trend will continue.

Programmable logic also has a trend that will continue: bigger, faster, and cheaper. Even today it can be seen that for many large complex functions such as RS and Viterbi codes, programmable logic is not only a viable solution, but is the best solution. The huge strides that have been made in only the last five years can be expected to continue, with the ability to incorporate even larger and more complex functions into programmable logic at increased levels of performance. These larger, faster devices, combined with the increasing availability of intellectual property targeted at programmable logic, such as the functions evaluated in this paper, is a win not only

for the PLD manufacturer and the designer, but also the end customer who should see increased performance at lower cost in the end item product as well.

Our results showed that our design can correct errors up to 14% of the number of data points without any problem. When we increased the error rate up to15%, we saw that the output started to change. The accuracy of our design depends on the error rate inserted in the input data points as was shown before.

Some directions to continue this work are the following:

- Changing the parameters of the decoding algorithm, like the number of states per stage, and observing the effect on the hardware implementation.
- Combining Viterbi Algorithm with other decoding methods to improve the error correction rate.

BIBLIOGRAPHY

[GAR00]. Garrett D., Stan M. "Lower power Architecture of the soft output Viterbi Algorithm", ACM ISBN 1-58113-059-7/98/08, 2000.

[ALT01]. Altera Corporation, "Viterbi Compiler, MegaCore Function", user guide, November, 2001.

[MEG01]. Meguerdichian S., Koushanfar F., Mogre A., Petranovic D., Potkonjak M., "MetaCores: Design and Optimization Techniques", ACM ISBN 1-58113-297-2/01/0006, 2001.

[ACT97]. Actel Corporation, "Designing Telecommunication Applications Using Digital Signal Processing Functions With FPGAs", Actel 5192622-0, pages (1-8), 1997.

[XILINX] www.xilinx.com

[CHIPCENTER] www.chipcenter.com

[XESS] www.xess.com

[ORCAD] www.orcad.com

[DU00]. Du J., Falkowski J., Aziz A., Lane J., " Implementation of Viterbi Decoding On StarCore SC140 DSP", CMP Media Inc., DSP conference proceedings, 2000.

[MEN98]. Some decoding techniques in this work were taken from: Mentor Graphics Corporation, "Viterbi Decoder Algorithm ", Copyright 1998.

[SWA02]. Swaminathan S., Tessier R., Goeckel D., Burleson W., "A Dynamically reconfigurable Adaptive Viterbi Decoder ", ACM 1-58113-452-5/02/0002, 2002.

[SKA96]. Skahill K., "VHDL for Programmable Logic", Addison-Wesley Publishing, 1996.

[SCI02]. SCIworx, "Viterbi Decoder", Doc. No.:VC01B1.002HO, Rev 01.00.02. Sciworx.GmbH, 2002.

[XIL01] XILINX Inc., "Viterbi Decoder", LogicCore V1.0, 2001.

ABSTRACT

IMPLEMENTATION OF VITERBI DECODER
ON XILINX XC4005XL FPGA

by

NABIL ABU-KHADER

May 2002

Advisor: Dr. Pepe Siy

Major: Electrical Engineering

Degree: Master of Science

The Viterbi decoding algorithm is used to decode convolutional codes and is found in many systems that receive digital data that might contain errors. The use of error-correcting codes has proven to be an effective way to overcome data corruption in digital communication channels. In previous works, researchers describe the Viterbi Algorithm, but the accuracy does not exceed 10% of data points. Also, a lot of previous works do not follow IEEE 802.16 new specifications. Viterbi decoders are generally implemented using programmable digital signal processors (DSPs) or special purpose chip sets and application-specific integrated circuits (ASICs). Here, we aim to implement such decoder on an FPGA.

In This paper, we provide a more accurate Viterbi decoder according to IEEE 802.16 specifications. We used VHDL hardware description language to implement the algorithm. We also used OrCAD Capture V9.1 to compile, synthesize, and simulate our code. IEEE 802.16 standard specifies the air interface of fixed point-to-multipoint

broadband wireless access (BWA) systems providing multiple services. This standard is intended to enable rapid worldwide deployment of broadband wireless access products. The new IEEE 802.16 specifications require a Viterbi block decoder with constraint length of 3, traceback length of 32, and minimum throughput requirement of 44.8 Mbps.

The Implementation parameters for the decoder have been determined through simulation and the decoder has been implemented on a Xilinx XC4005XL FPGA.

AUTOBIOGRAPHICAL STATEMENT

NABIL ABU-KHADER

Education

- **Towards the Ph.D. in ASIC/VLSI**

  Wayne State University; Detroit, Michigan

- **M.S. in Electrical Engineering, 2002**

  Wayne State University; Detroit, Michigan

- **B.S. in Electrical Engineering, 1999**

  Al-Balqa Applied University; Amman, Jordan

Experience

- **Graduate Teacher Assistant,** VLSI Lab, WSU, Detroit, MI. 2000-2002
- **Electrical Engineer,** Barada Electrical Company, United Arab Emirates. 1999-2000
- **Electrical Engineer,** General Plastic Industrial Company, Jordan. 6/1999-9/1999