High-speed Viterbi Decoder Design And Implementation With FPGA

By
Jian Lin

A Thesis
Submitted to the Faculty of Graduate Studies
In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

0-612-57555-1

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION PAGE

High-speed Viterbi Decoder Design and Implementation with FPGA

BY

Jian Lin

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University

of Manitoba in partial fulfillment of the requirements of the degree

of

Master of Science

JIAN LIN © 2001

Abstract

This thesis describes a design and implementation of a Viterbi decoder using FPGA technology.

We use the sliding block filtering concept, the pipeline interleaving technique and the forward processing method to construct the design. We use VHDL to describe the design, Synopsys tools to synthesize it and Xilinx tools to target the design to an XVC300-8 device.

Besides the above, the principle of the Viterbi Algorithm, two kinds of structures of the Viterbi decoder, VHDL coding style, a high level synthesis strategy and the methodologies of FPGA design are briefly discussed.

We also present complete source code, scripts and reports for this design in appendixes.

Contents

Chapter 1

Introduction

1.1 Motivation

With the growing use of digital communication, there has been an increased interest in high speed Viterbi decoding design within a single chip. Advanced *field programmable gate array* (FPGA) technologies and well developed *electronic design automatic* (EDA) tools have made it possible to realize a Viterbi decoder with the throughput at the order of Giga-bit per second without using off-chip processor(s) or memory. The motivation of this thesis is to use VHDL, Synopsys synthesis and simulation tools to realize a 1 Gbits/s Viterbi decoder targeting Xilinx FPGA technology.

1.2 Overview

In communication systems, error control coding techniques play a very important role. Described in its simplest terms, error-control coding involves the addition of redundancy to transmitted data to provide the means for detecting and correcting errors that inevitably occur in any real-world communication system.

Convolutional coding is one of these techniques. The convolutional encoder operates on the source data stream at the bit level and produces a continuous stream of encoded symbols. Each information bit will affect a finite number of consecutive symbols in the output stream.

Convolutional encoding with Viterbi decoding is a *Forward Error Correction* (FEC) technique that is particularly suited to a channel in which the transmitted signal is mainly corrupted by *Additive White Gaussian Noise* (AWGN).

The classical method to realize the Viterbi decoder is an iterative calculation and memory trace-back [3], which is effective for a moderate decoding speed and long constrain length. For a high-speed decoder, iterative calculation and memory trace back becomes the bottlenecks for the throughput.

This thesis presents a concurrent method to realize the Viterbi decoder, in which sliding block input, concurrent calculations between consecutive blocks and pipeline processing techniques are used. With these techniques, the throughput can reach 1Gb/s.

1.3 Outline

The organization of this thesis is as following: In Chapter 2, Viterbi algorithm and its properties are reviewed. Chapter 3 describes the principle of the design, the structure of the decoder and detailed component design. In Chapter 4, the verification method, simulation environment and performance evaluations are presented. Chapter 5 covers implementation of the design including VHDL coding, compiling strategy, placement and routing as well as back-annotation for the timing simulation. In chapter 6, comparisons are made with some existing designs. Chapter 7 provides some conclusions.

Chapter 2

Viterbi Algorithm

Viterbi decoding was developed by Andrew J. Viterbi in his seminal paper [1] in 1967. Since then, other researchers have expanded on his work by finding good convolutional codes, exploring the performance limits of the technique, and varying decoder design parameters to optimize the implementation of the technique in hardware and software.

This chapter first gives a brief description of convolutional encoding. Then the Viterbi algorithm is discussed. Finally two properties of the Viterbi algorithm are presented for use in subsequent chapters.

2.1 Convolutional Encoding

Convolutional codes are usually characterized by two parameters and the patterns of n modulo-2 adders. The two parameters are the *code rate* and *constraint length*. The code rate, k/n, is expressed as a ratio of the number of bits into the convolutional encoder (k) to the number of channel symbols output by the convolutional encoder (n) in a given encoder cycle. The constraint length, K, denotes the "length" of the convolutional encoder, i.e. how many k-bit stages are available to feed n modulo-2 adders that produce the channel symbols. Figure-2.1 shows a general convolutional encoder. An alternative parameter to K is v ($=K$-1), which indicates there are 2^v states in the encoder [2]. The content of the K - 1 least significant bits of the shift register is denoted as the *state*.

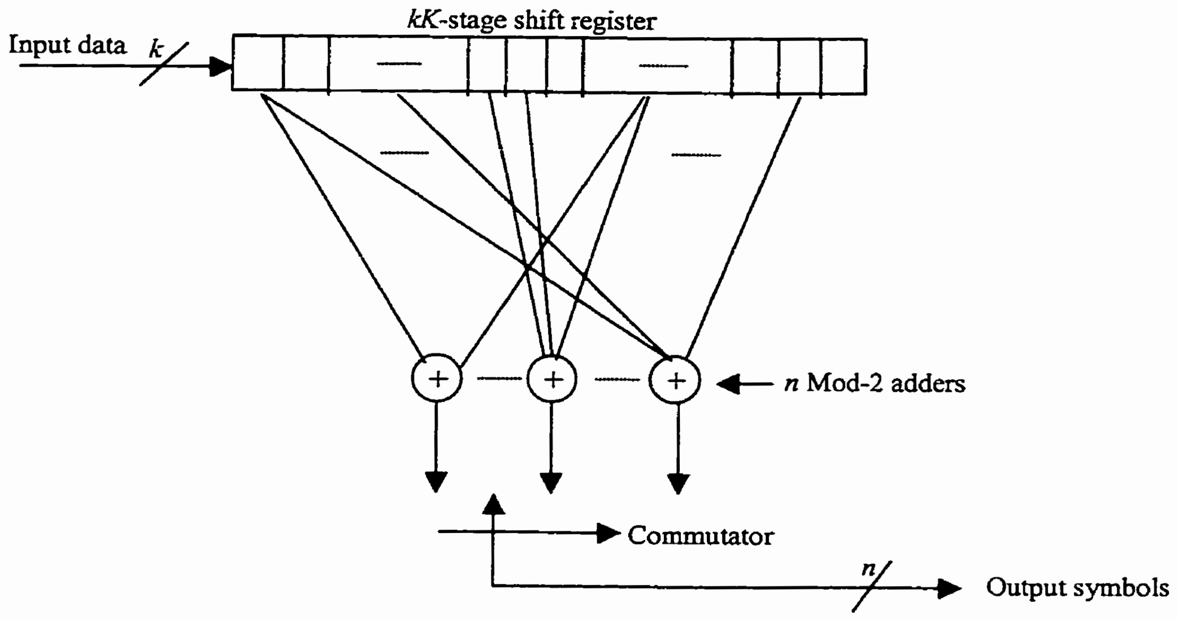


Figure-2.1 A general convolutional encoder

In this thesis, we take $R = 1/2$, $K = 3$, the two modulo-2 adders are: : $g_1 = x^2 + x^1 + 1$ (or expressed as: $g_1 = 111$) , $g_0 = x^2 + 1$ (or $g_0 = 101$), This encoder has been determined to be one of the best codes for $R = 1/2$, $K = 3$, or (2, 1, 3) code for short [3]. The corresponding logic diagram is shown as Figure-2.2. There are four possible states S ($= S_1S_0$): 00,01,10,11.

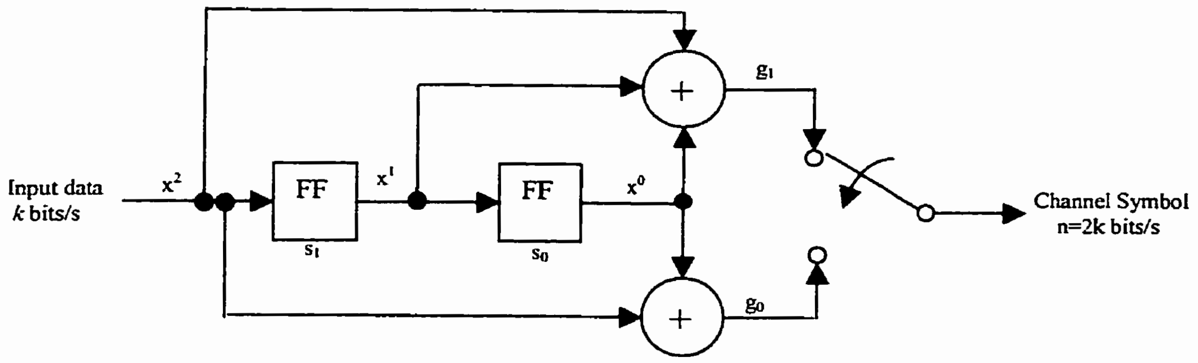


Figure-2.1. Convolutional Encoder with $R = 1/2$, $K = 3$, $g_1 = 111$, $g_0 = 101$

In this encoder, input bits should be stable during the encode cycle. The encoding cycle starts when the shift register clock edge occurs. The output of the left-hand flip-flop (FF) is clocked into the right-hand flip-flop, while the previous input bit is clocked into the left-hand flip-flop, and a new input bit becomes available. Then the outputs of the upper and lower modulo-2 adders become stable. The selector clock, which is double in frequency to the register clock, triggers out g_1, g_0 alternatively, forming the channel symbol sequence.

If the input sequence is

010111001010001,

then the output sequence will be:

00 11 10 00 01 10 01 11 11 10 00 10 11 00 11,

assuming that the output of both of the flip-flops in the shift register are initially cleared.

From the above, we can see that the mechanism of convolutional encoding is spreading the single bit information in several consecutive bits.

2.2 Viterbi Algorithm

The Viterbi Algorithm is known to be the maximum-likelihood decoding method for convolution codes. We use the (2, 1, 3) code with $g_1 = 111$, $g_0 = 101$ to explain it.

The evolution of this four-state encoder can be described using the trellis diagram shown in Figure-2.3. The trellis is a time-indexed version of the state diagram. Each box corresponds to a state at a given time index, and each branch corresponds to a state transition. Associated with each branch is the input bit and the output symbol corresponding to the state transition. Given a known starting state, every input sequence corresponds to a unique path through the trellis. In the trellis, each branch is assigned a weight, referred to as the *branch metric*, which is a measure of the likelihood of the correspoding transition given the noisy channel symbol sequence. Branch metrics are typically calculated using Hamming distance, so that the more likely path (shortest path) through the trellis corresponds to the most likely input sequence. From this point of view, the Viterbi algorithm is an efficient method for finding the shortest path through a trellis.

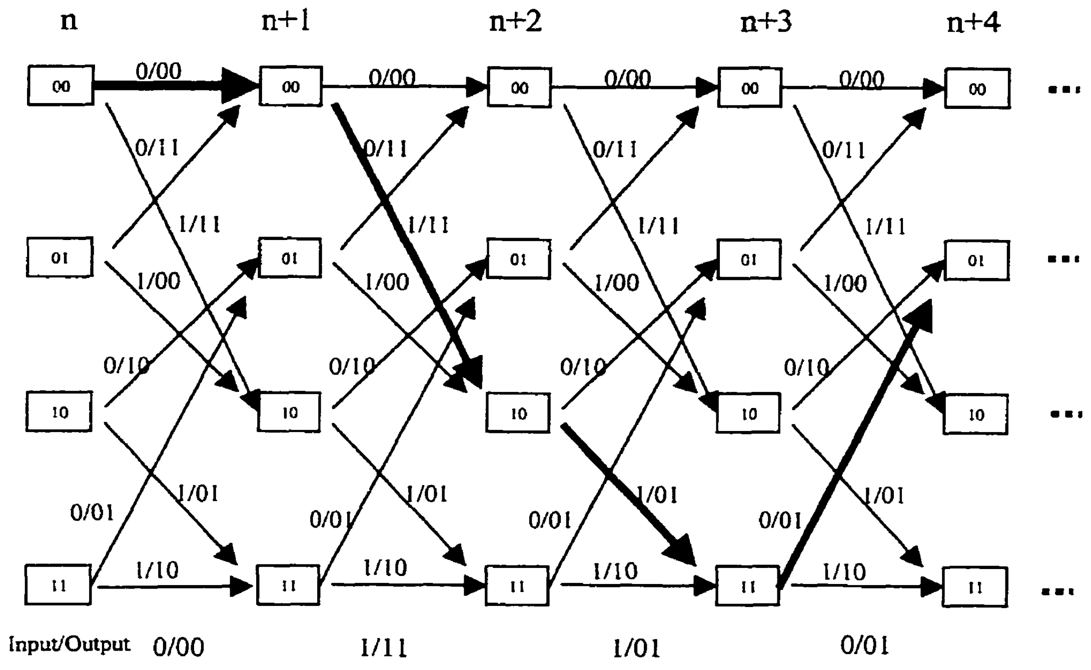


Figure-2.3. Trellis for convolutional code with 4 states

The first phase of the Viterbi algorithm is to recursively compute the shortest path from time n to time $n + 1$. At time n each state i is assigned a *path metric* Γ_n^i which is defined as the accumulated metric along the shortest path leading to that state. The path metrics at time $n + 1$ can be recursively calculated in terms of the path metrics of the previous iteration as follows:

$$\Gamma^{j}_{n+1} = \min_{i} \{ \Gamma^{i}_{n} + \lambda_{n}^{i,j} \} \qquad (1)$$

where i is a predecessor state of j and $\lambda_{n}^{i,j}$ is the branch metric on the transition from state i to state j. The qualitative interpretation of this expression is as follows. The shortest path into state j must pass through a predecessor state by definition. If the shortest path into j passes through i, then the path metric for the path must be given by the path metric for i plus the branch metric for the state transition from i to j. The final path metric for j is given by the minimum of all candidate paths.

The equation given in (1) is the well-known *add-compare-select* (ACS) operation. The hardware that implements this is referred to as a *two-way ACS unit.* For example, the ACS unit for state 00 in the four-state trellis is shown in Figure-2.4. It outputs the updated path metric and a 1-bit decision d^{j}_{n+1}, which identifies the entering path of minimum metric. All the decision bits should be stored for trace-back use.

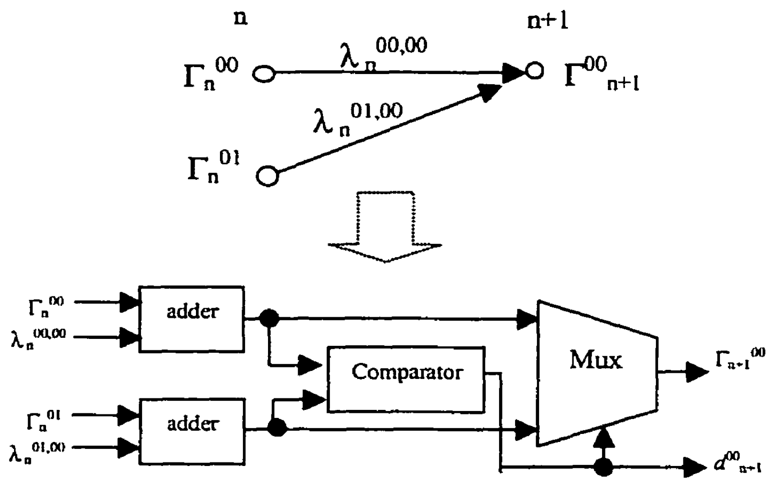


Figure-2.4. the two-way ACS units of state-00

Connecting four two-way ACS units together, the four-state trellis transition can be implemented as shown in Figure-2.5.

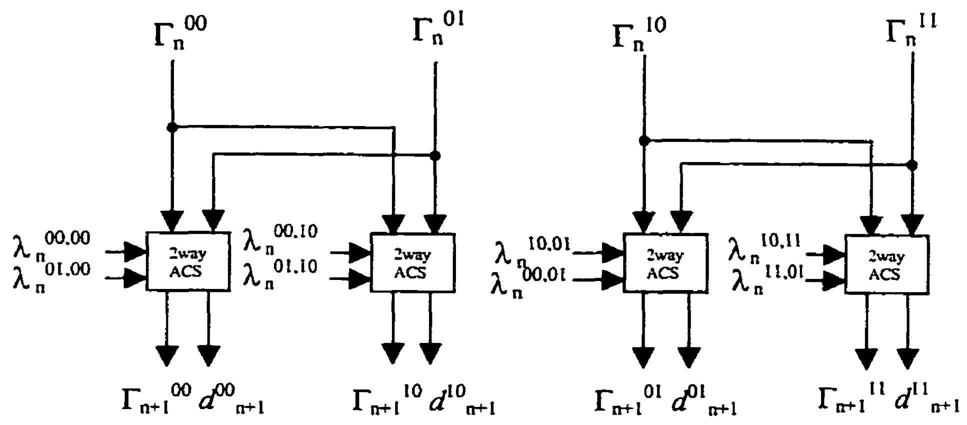


Figure-2.5 four-state trellis transition

The second phase of the Viterbi algorithm involves tracing back and finding the shortest path through the trellis, The shortest path leading to a state is referred to as *the survivor path* for that state. It can be recursively found using the stored decision bits. Given the current state S_n and the current decision bit, the previous state S_{n-1} can be estimated according to the following trace back recursion:

$$S_{n-1} = (S_n << 1)\, d^S \qquad (2)$$

Which corresponds to a 1-bit left shift of the current state register with input equal to the current state decision bit.

After finding the survivor path, the original message can be re-created using a table that maps state transitions to the inputs that caused them,. Table-2.1 is the table for the (2, 1, 3) code with $g_1 = 111$, $g_0 = 101$.

	Input was, Given Next State =			
Current State	00	01	10	11
00	0	X	1	X
01	0	X	1	X
10	X	0	X	1
11	X	0	X	1

X = Impossible

Table-2.1 State transition map

2.3 Two Properties of The Viterbi Algorithm

A property of the trellis which is used for survivor path decoding is that if the survivor paths from all possible states at time n are traced back, then with high probability, all the paths merge at time $n-L$, where L is the *survivor path length* and is typically $5v$ [4]. Once the survivor paths have merged, the traced path is unique independent of the starting state and future ACS iterations.

Similarly, when starting with unknown initial state metrics (typically set to zero), the state metrics after J trellis iterations are independent of the initial metrics, or equivalently, the survivor path will merge with the true survivor path as if the initial metrics had been known. The parameter J is the *synchronization length* and is also typically $5v$ [5].

2.4 Summary of The Viterbi Algorithm

Based on the Viterbi Algorithm and survivor path merge properties, we can summarize the Viterbi decoding process as follows:

1. For each received channel symbol, calculate the branch metric (i.e. Hamming distance) between the channel symbol and every possible channel symbol in the trellis.

2. At each decoding cycle, iterate the ACS process for each state to find the path metrics and store all the produced decision bits.

3. When the stored decision vectors are long enough (normally $> L$), start the trace back process using the formula (2) and recover the original message using the state transaction table as shown in Table-2.1.

Chapter 3

Hardware Design of The Viterbi Decoder

In principle, the Viterbi algorithm can be easily realized in hardware. The complexity of doing so it is determined by the constraint length *K* and the decoding speed [6]. To realize a Viterbi decoder with high throughput and long constraint length, the hardware scale will be very large. With the development of *very large-scale integration* (VLSI) technology, more and more topologies have been proposed to implement high speed Viterbi decoders using *application specific integrated circuits* (ASICs) or FPGAs. In this thesis, the concurrent calculation and the pipeline, interleaving techniques are introduced into the Viterbi decoder. This structure makes it possible to speedup the throughput with only a linear increase in hardware complexity.

In this chapter, we first briefly outline the design of a sequential Viterbi decoder, and then give a detailed description of our parallel Viterbi decoder: sliding block decoder. Finally, we provide detailed component design.

3.1 Sequential Decoder

There are many kinds of structures that can be used to implement the Viterbi Algorithm. Sequential decoding is the traditional one. Sequential decoding uses several memory banks to store, trace-back and decode the decision bits. The typical architecture of this kind of decoder is shown in Figure-3.1.

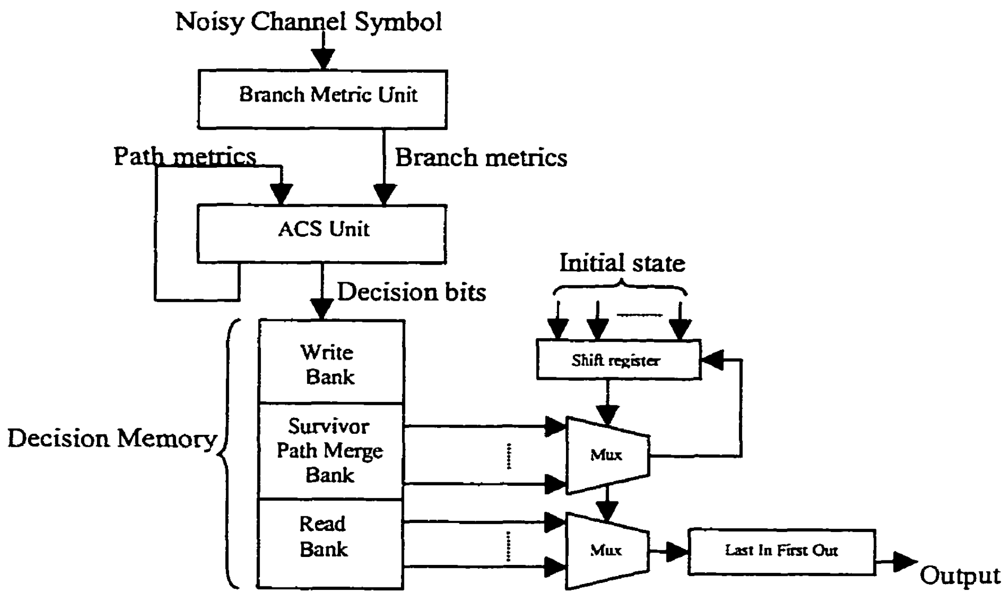


Figure-3.1. Typical architecture of a sequential Viterbi decoder using memory

Decision memory is usually composed of at least three separate banks for writing, survivor path merging and reading (decoding) at the same time to match the throughput of the ACS process rate. While the write bank accepts the decision vectors, the survivor path merge bank is for finding the survivor path and the read bank is for producing the output stream. The three memory banks will change roles in the trace-back process in this way:

Write Bank → Merge Bank → Read Bank → Write Bank ...

This sequential architecture is suitable for low throughput requirements and more area-efficient for long constraint length codes, The total memory size in depth and width is determined by the constraint length of the code. For example, if $K = 15$, the least total memory size is $3 X L X N = 3 X 5 (15 - 1) X 2^{(15-1)} =$ 3,440,640 bits. So, for moderate speed decoders with long constraint lengths, using the sequential structure is a practical method.

Increasing the throughput in this architecture needs more memory banks working in parallel and this increases the hardware cost dramatically.

3.2 Sliding Block Decoder

For high-speed practical implementations of the Viterbi algorithm, architectures are desired that at worst lead to a linear increase in hardware complexity for a linear speedup in the throughput rate. The sliding block decoder is one such architecture that utilize the concurrent calculation and pipeline processing [7].

Based on the first property of the survivor path, it is proposed in [7] that the state at time n can be decoded using only information from the interval $n - L$ to $n + L$. A decoder is called a *sliding block Viterbi decoder* (SBVD). It uses a block of received channel symbols as an address to access a lookup table, and the table output is the decoder output. Unfortunately, this implementation is impractical, because even for the simplest code (2,1,3) with 3-bit soft decision input, the number of the address bits rquired for the table is $2L \times 3 = 2 \times 5\,(3 - 1) \times 3 = 60$ bits! However, variations on the concept of the sliding block decoder have been used to realize the concurrent Veterbi Algorithm.

The *minimized method* in [8] and the *equal forward and backward method* in [10] are two practical applications of the sliding block decoder concept. These two methods are based on the survivor path property of the Viterbi algorithm and make use of the pipeline and concurrent structures to implement the Viterbi decoder for the (2, 1, 3) code.

The throughputs of these two implementations can reach up to 600 Mbits/sec and 1Gbits/sec, respectively.

The design in this thesis is also based on the sliding block decoding concept and pipeline technique, but the structure is different and some improvements are made in area-efficiency. The design uses *forward processing* and a *pipeline sharing* architecture.

Since the block length are finite, the ACS and trace-back recursions can be unfolded and pipelined to yield the systolic architecture shown in Figure-3.2.

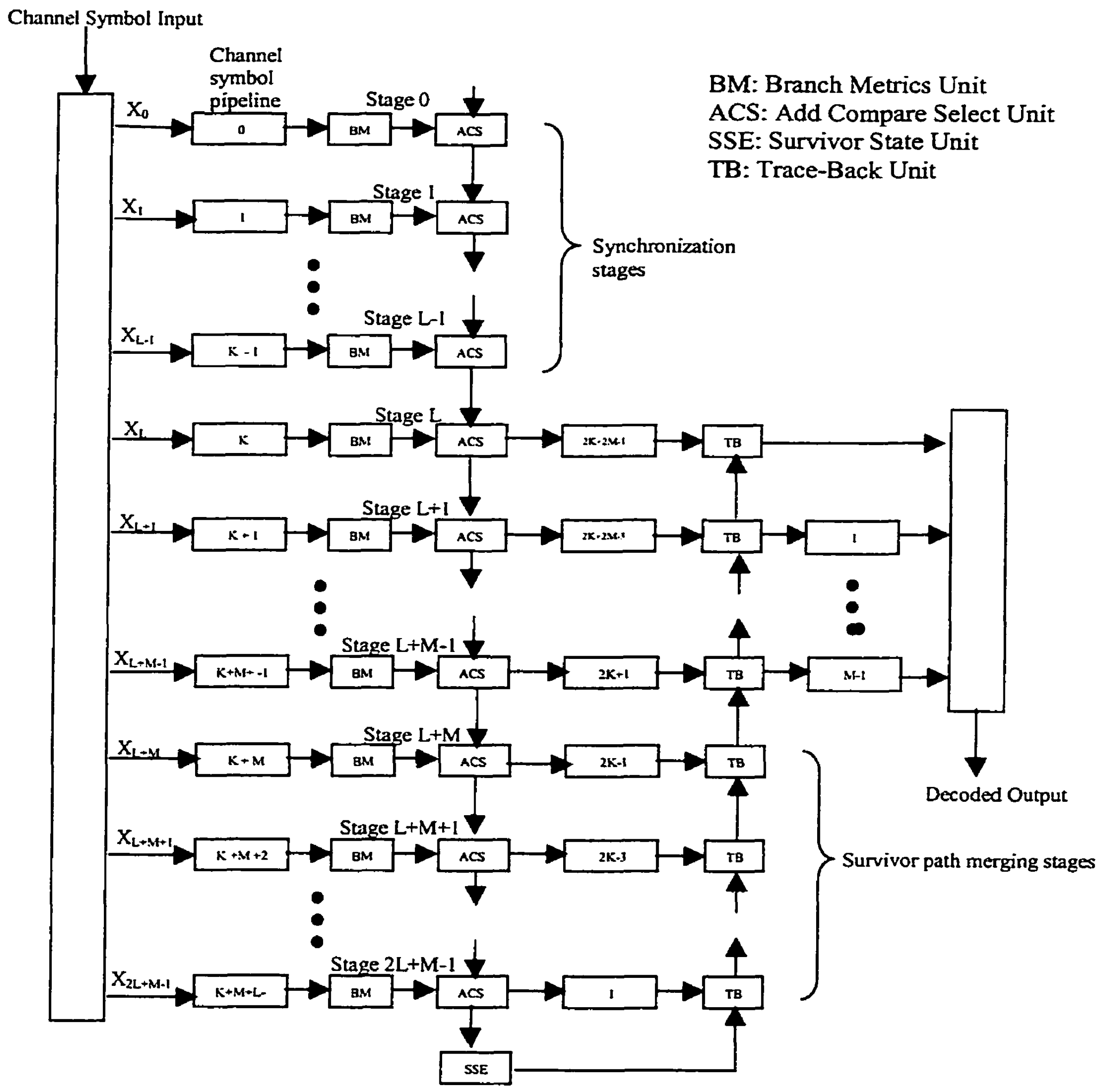


Figure-3.2 Forward processing sliding block decoding

For each block, starting from Stage 0, do the ACS calculation for the first channel symbol pair X_0, get the path metric for each state. Then feed these path metrics into Stage 1, do the ACS again with the second channel symbol, X_1, get the new path metrics, repeat this procedure until the end of the block. Then, compare the four state metrics in the *survivor state estimation* (SSE) block, get the minimum path metric state which is the starting state for the trace-back process. From the trace-back process, the original message can be retrieved. The SBVD method is equivalent to the best state survivor path decoding, hence the survivor path length and synchronization length can be reduced to $L = 2.5v$ [4]. So for one block of length $2L + M$, the decoded bits from L to $L + M$ are the most likely original message.

The performance of this method depends on the value of L. The throughput is determined by the decoding length M. Although the number of ACS units scales with M, the number of buffers in the pipelines scales with M^2. Therefor, it is desirable to achieve a given throughput by minimizing the decoding length and maximizing the clock rate. In this design, the aim is to achieve the throughput of 1Gbit/s. Since the minimum operating clock period can reach 12 ns in this design (see later chapters) , if we take $M = 12$, we can get 1Gbit/s throughput. In addition, since the decoder outputs M bits per clock cycle, M channel symbol pairs need to be read from the input stream during the same clock cycle. So only M new channel symbol pairs need to be fed into the decoder each cycle. The rest of the $2L$ symbols have been buffered at the previous cycle. If we let $M = 2L$ (i.e. $L = 6 > 2.5v$), we can make full use of the pipeline buffer resources through sharing. In this way, decoding of continuous input blocks with *M=2L* is analogous to pipeline filtering with *2L* overlap as shown in Figure-3.3.

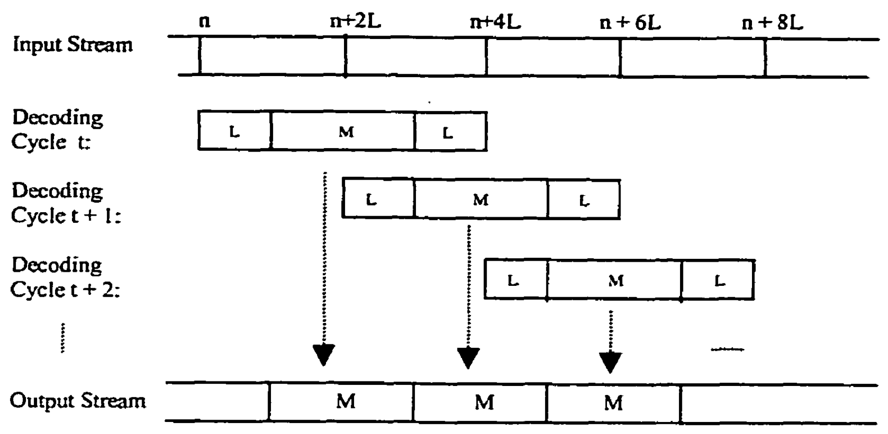


Figure-3.3. The pipeline processing flow

By overlapping these 2*L* channel symbols, we can reduce the number of channel symbol pipeline buffers from Figure-3.4 (a), which is a detailed form of Figure-3.2 with L=3 for example, to Figure-3.4 (b). By sharing the pipeline skew buffers which contain the same channel symbols, the number of buffers can be further reduced as shown in Figure-3.4 (c).

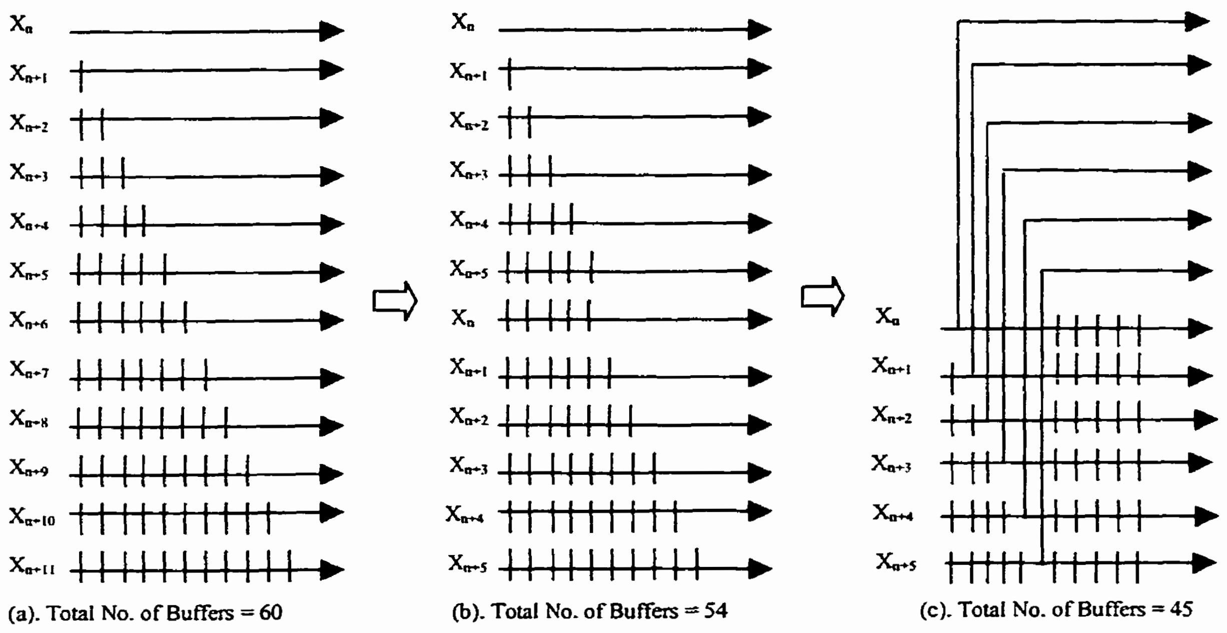

Xn
Xn+1
Xn+2
Xn+3
Xn+4
Xn+5
Xn+6
Xn+7
Xn+8
Xn+9
Xn+10
Xn+11
(a). Total No. of Buffers = 60
(b). Total No. of Buffers = 54
(c). Total No. of Buffers = 45

Figure-3.4. Channel symbol improvements

A more detailed view of the forward processing SBVD architecture for the simplified case of $L = 3$ and $M = 6$ is shown in Figure-3.5. For illustration, it is divided into two parts: the channel symbol pipelines and 6-bit decoder unit.

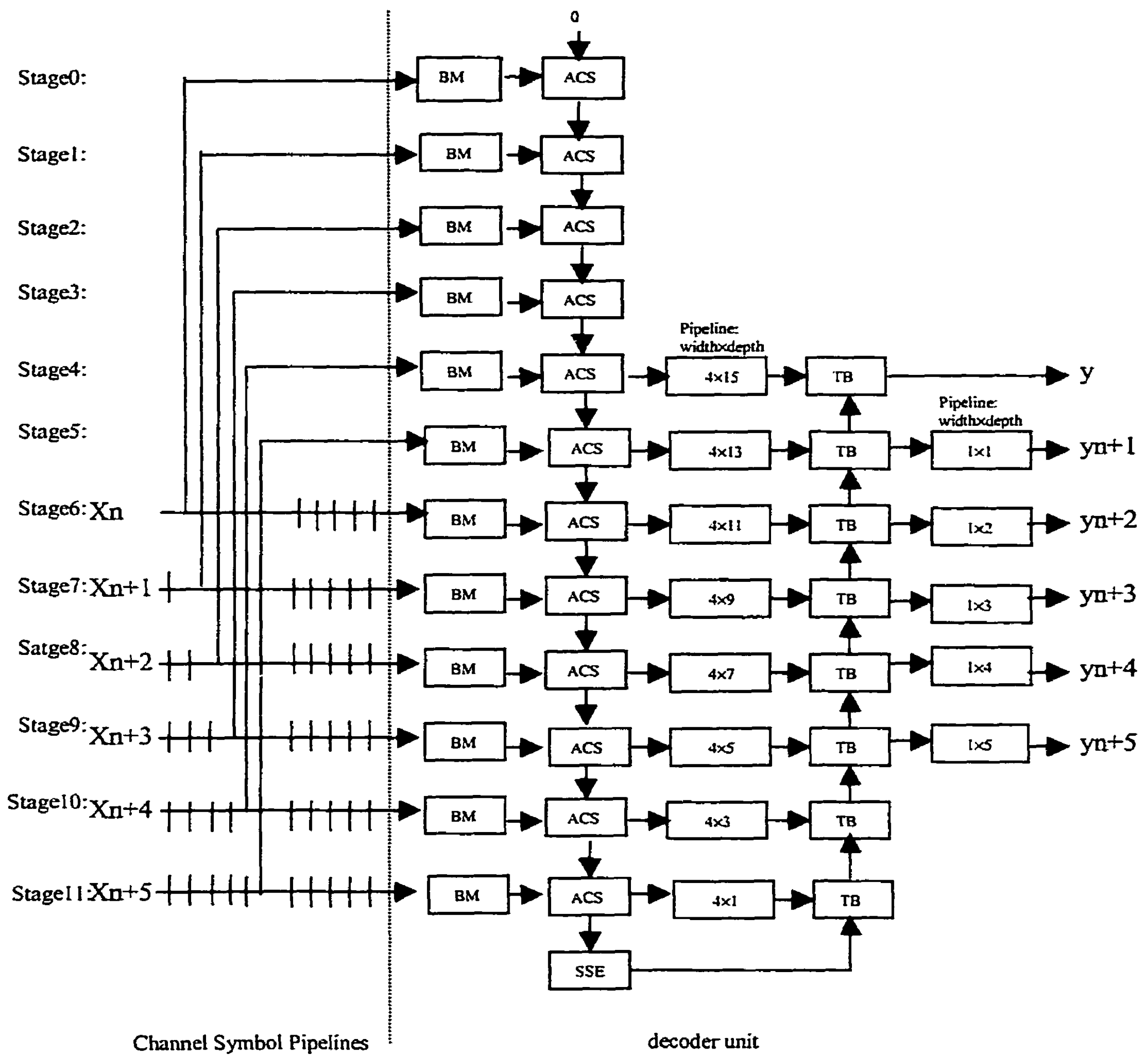
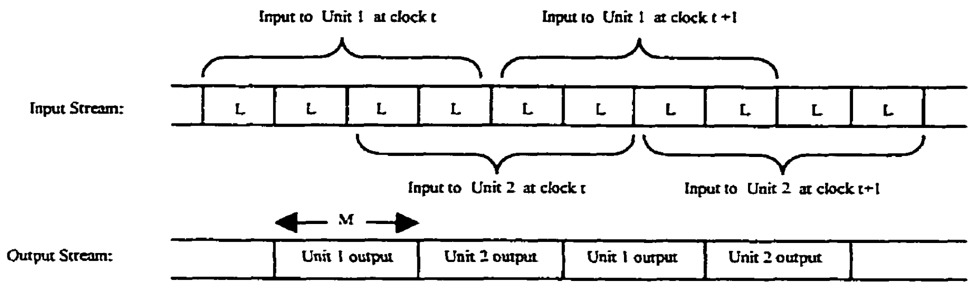


Figure-3.5. Simplified Forward Processing Sliding Block Viterbi Decoder

Given that the 12-bit decoder works at a clock rate f_{clk}, its throughput rate is $12f_{\mathrm{clk}}$. Theoretically, using N of these decoders connected in parallelly, the throughput of the composite Viterbi decoder can reach $12Nf_{\mathrm{clk}}$, as long as the serial-to-parallel and parallel-to-serial shift registers can operate at this speed. this realizes the linear increase in hardware complexity for a linear speedup in the throughput rate. For example, with N = 2 the two M-bit decoder units process the input stream blocks alternately as shown in Figure-3.6.



Input to Unit 1 at clock t
Input to Unit 1 at clock t+1
Input Stream:
L
Input to Unit 2 at clock t
Input to Unit 2 at clock t+1
M
Output Stream:
Unit 1 output
Unit 2 output

Figure-3.6. Decoding flow for the 2-Unit decoder

We can see that, at each decoding cycle, there are $2L$ channel symbols will be input to both units. As such their pipelines can be shared with each other. Figure-3.7 shows the structure of a 2-unit 6-bit decoder, that forms a 12-bit decoder. The throughput of such a decoder can reach $12f_{\mathrm{clk}}$.

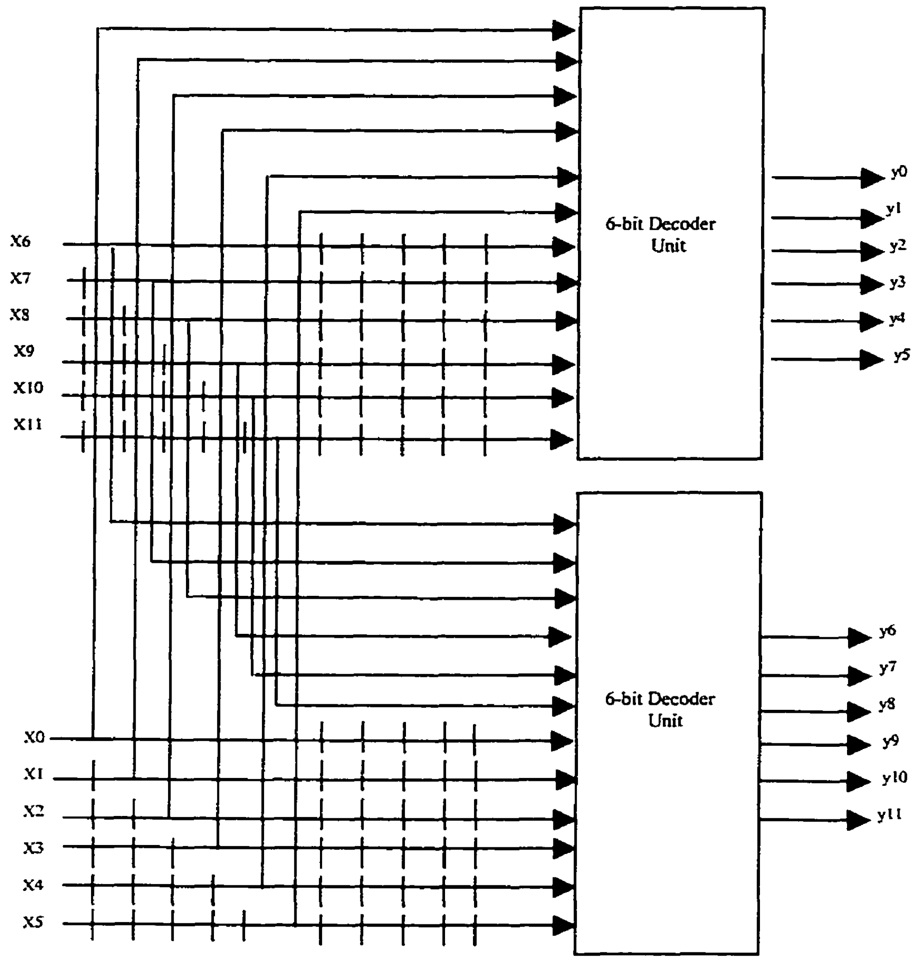


Figure-3.7. The structure of a 12-bit decoder built using two 6-bit decoders

3.4 Component Design

Implementation of the Forward Processing SBVD decoder is relatively straightforward given the high level architecture shown in Figure-3.5. The design consists of the following five basic functional components:

- Branch Metric unit (BM).
- ACS unit (ACS).

- Survivor State Estimate unit (SSE).
- Trace-Back units (TB).
- Pipeline buffers.

In each unit, there is a register storing all outputs. So one pipeline stage is accounted per unit. A single clock is used to synchronize all units and pipeline skew buffers. The critical path between the units determines the maximum clock rate. The following subsections discuss the details of each unit.

3.4.1 Branch Metric Units

For one trellis iteration, each branch metric (BM) unit accepts two 3-bit quantified symbols and produces four branch metrics λ_n^{00}, λ_n^{01}, λ_n^{10}, λ_n^{11}, corresponding to the four possible encoder outputs 00, 01, 10, 11, respectively. The branch metrics are calculated using the Hamming Distance measure as shown in Table-3.1 (where x is the hypothesized encoded symbol and y is the received symbol) .

Y	000	001	010	011	100	101	110	111
X=0	0	1	2	3	4	5	6	7
X=1	7	6	5	4	3	2	1	0

Table-3.1 Hamming Distance measure for branch metric

The branch metric is the sum of 2 symbol metrics, and thus it lies in the interval (0, 14), and therefore 4 bits are needed to express the branch metric. From Figure-2.3 we can see that $\lambda^{00,00} = \lambda^{01,10}$, $\lambda^{01,00} = \lambda^{00,10}$, $\lambda^{10,01} = \lambda^{11,11}$, $\lambda^{10,11} = \lambda^{11,01}$. We re-define these λs as

λ^{00}, λ^{11}, λ^{10}, λ^{01} , respectively. The branch metrics unit can be implemented using inverters and adders as shown in Figure-3.8.

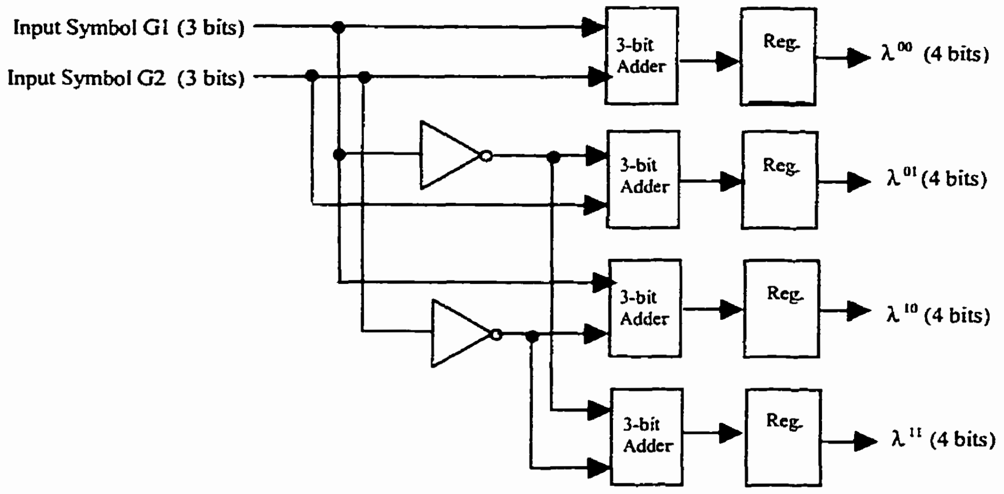


Figure-3.8. Branch metric unit

3.4.2 ACS Units

In section 2.2, the 2-way ACS unit and 4-way ACS units were constructed as shown in Figure-2.4 and Figure-2.5 respectively. In this section, we discuss the detail design of the addition, comparison and selection operations.

The recursive path metric update results in unbounded word growth due to the addition of branch metrics, which are always nonnegative. We avoid normalization using the modulo arithmetic approach proposed in [11]. The modulo arithmetic approach exploits the fact that the Viterbi Algorithm inherently bounds the maximum dynamic range Δ_{max} of the path metric to be:

$$\Delta_{max} \leq \lambda_{max} \log_2 N \qquad (4)$$

where N is the number of states and λ_{max} is maximum branch metric among state transitions [12].

Given two numbers a and b such that $| a - b | < \Delta$, which are to be compared using subtraction, a result from number theory states that the comparison can be evaluated as $(a - b) \bmod 2\Delta$ without ambiguity. Hence the path metrics can be updated and compared using modulo $2\Delta_{max}$ arithmetic. The modulo arithmetic is implicitly implemented by ignoring the path metric overflow. The number of bits for the path metric is:

$$\Gamma_{bits} = \lceil \log_2 (2\Delta_{max}) \rceil + 1. \qquad (5)$$

For the design of this 4-state decoder, $\Gamma_{bits} = \lceil \log_2 (2 \times 28) \rceil + 1 = 7$ bits.

Area-efficient ripple carry arithmetic is used to implement the 2-way ACS units since the look-ahead carry adder structure offers little speed advantage at the required 7-bit precision and increases additional area overhead, especially when the combined delay of the add and comparison are considered. By implementing the comparison using subtraction, the adder and subtractor carry chains run in parallel from LSB to MSB, resulting in an add-comparison delay that is only one full adder bit delay longer than the 7-bit ripple carry add delay alone as shown in Figure-3.9.

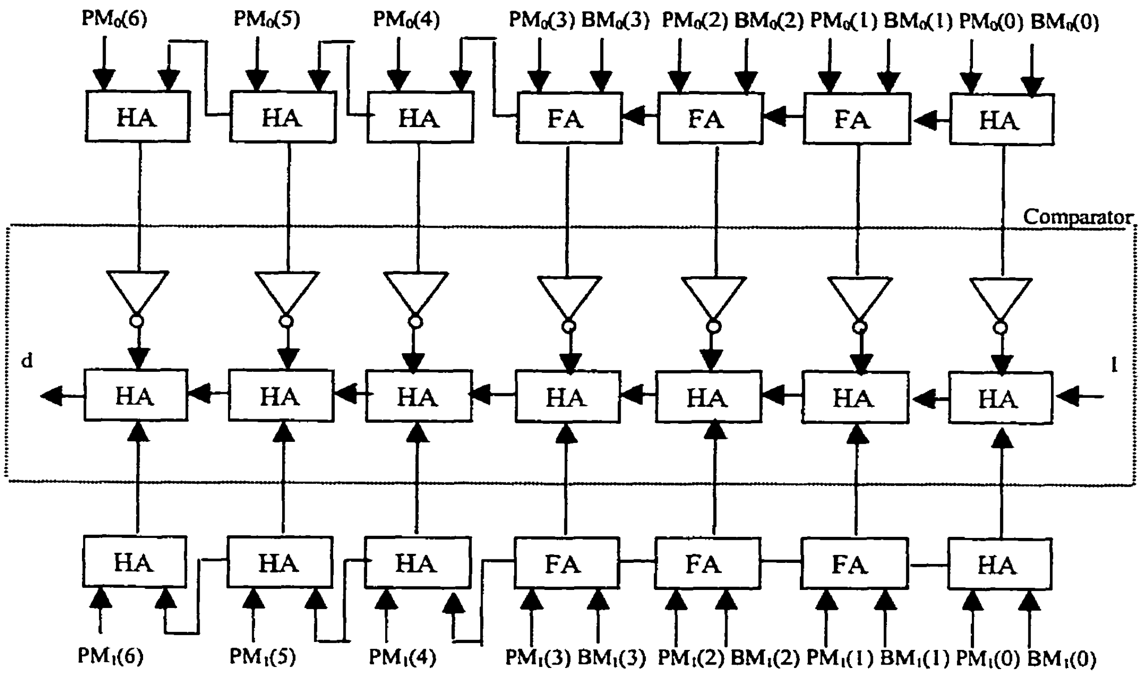

PM0(6)
PM0(5)
PM0(4)
PM0(3) BM0(3) PM0(2) BM0(2) PM0(1) BM0(1) PM0(0) BM0(0)
HA
FA
Comparator
d
1
PM1(6)
PM1(5)
PM1(4)
PM1(3) BM1(3) PM1(2) BM1(2) PM1(1) BM1(1) PM1(0) BM1(0)

Figure-3.9. Block diagram for 7-bit adder and 7-bit comparator.

The four-state ACS unit updates path metrics for a single iteration of the trellis. Each unit consists of four two-way ACS units. On each clock cycle, four path metrics from the previous stage are input and four updated path metrics are output. Each updating also generates a vector of four 1-b decisions that are output to the trace-back unit. All outputs are registered in flip-flops. After adding the flip-flop, the four-way ACS unit can be constructed as shown in Figure-3.10.

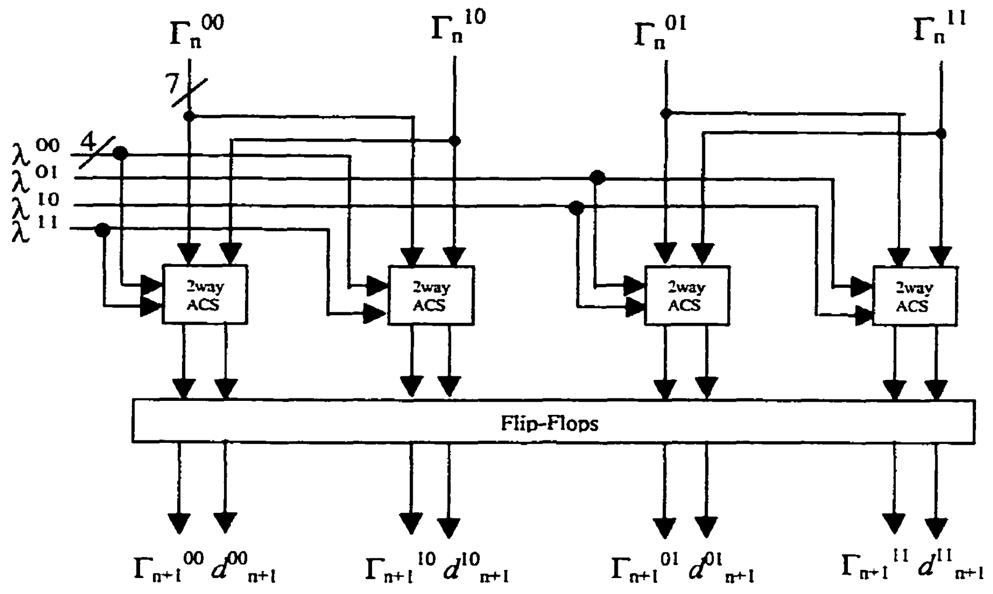


Figure-3.10. Four-way ACS

3.4.3 SSE Unit

To avoid the existence of the critical time path in the SSE unit, four path metrics are compared by generating six possible pair-wise comparisons and combining the comparison results to form the minimum path metric selection. The state with minimum path metric is registered in flip-flops as one stage of the buffer. The output of the flip-flops are the starting state for the trace-back processing. The logic block of the SSE unit is shown in Figure-3.11.

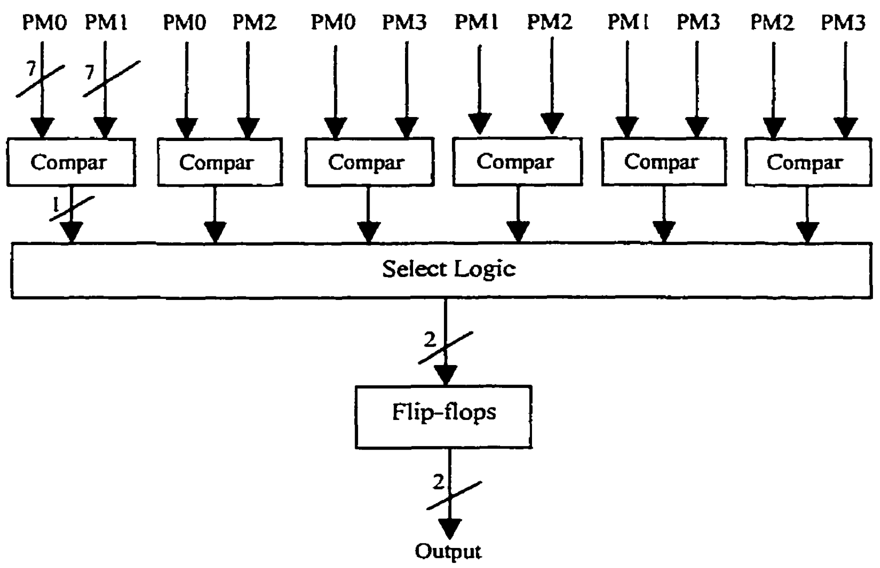


Figure-3.11. Logic block of the SSE unit

3.4.4 Trace-Back Unit

The trace-back (TB) unit implements a single trace-back recursion based on the formula $S_{n-1} = (S_n << 1) d^S$ and the state transition map shown in Table-2.1. The current estimated state S_n from the previous trace-back unit stage or SSE unit is used to select the decision of the current state from the input decision vector. The selected 1-b decision and the 2-b current state are appropriately combined to produce the estimated state S_{n-1} for the next stage and S_{n-1} (1) is the decoded output. S_{n-1} will be registered in Flip-flops to implement a one stage pipeline. The logic block diagram of the trace-back unit is shown in Figure-3.12.

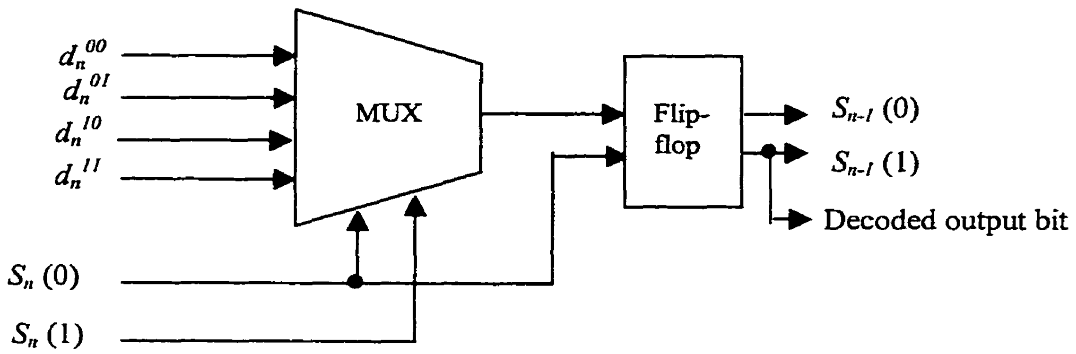


Figure-3.12. The logic block of trace-back unit

3.4.5 Pipeline Buffers

Unfolding and pipelining the recursive ACS and trace-back calculations requires re-timing of the input and output streams via skew buffers, which are implemented using flip-flops. The general *WXD* (widthXdepth) pipeline buffer is shown in Figure-3.13.

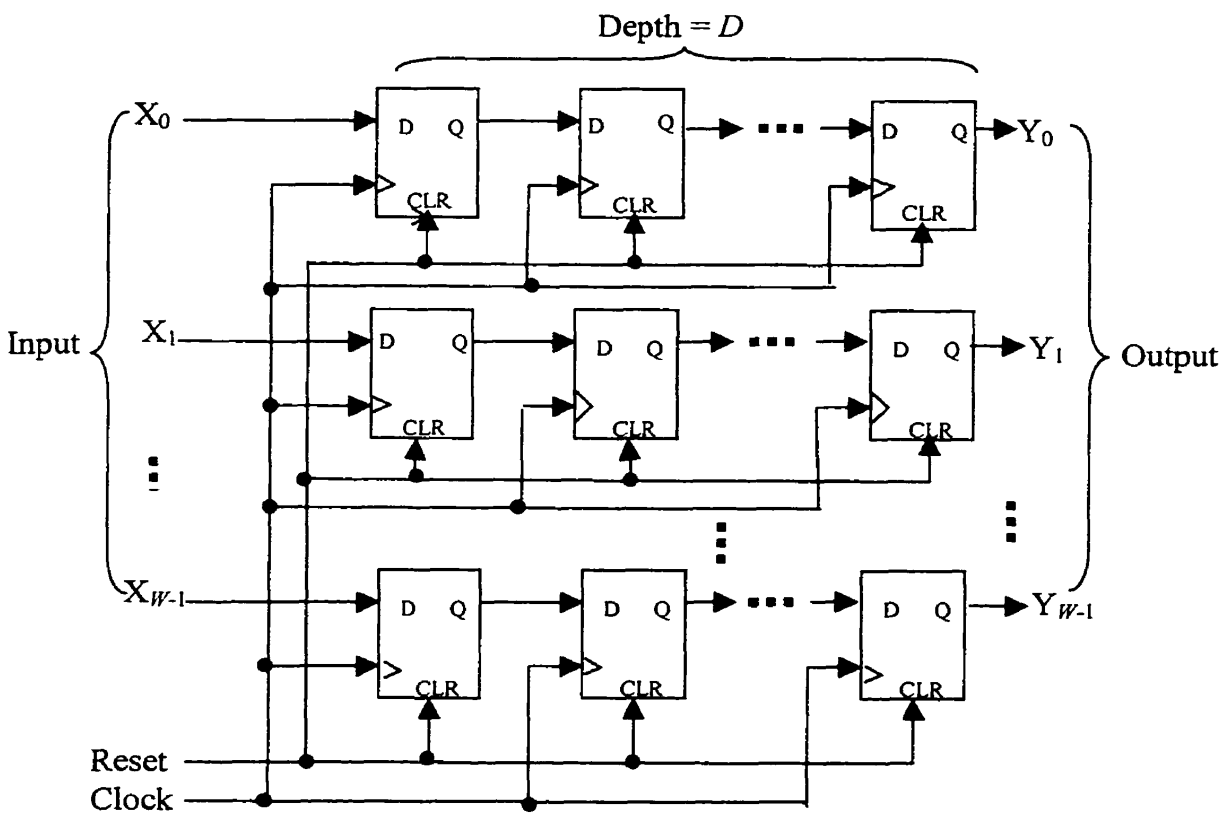


Figure-3.13. General *W*X*D* pipeline buffer

Chapter 4

Design Verification

4.1 Verification Environment

To verify the design, a simplified digital communication model was created as shown in Figure-4.1. At the transmitting end, the original messages are encoded into channel symbols, and then the digital signal should be converted into the analog signal. In the communication channel, the *additive white Gaussian noise* (AWGN) should be added. At the receiving end, the noisy analog signals are quantified into two 3-bit digital channel symbols. Through the Viterbi decoder, the original message can then be recovered. The performance of the Viterbi decoder can be evaluated through comparing the recovered message with the original message and calculating the *bit error rate* (BER) at a specific *energy per symbol to noise density ratio*, E_s/N_0.

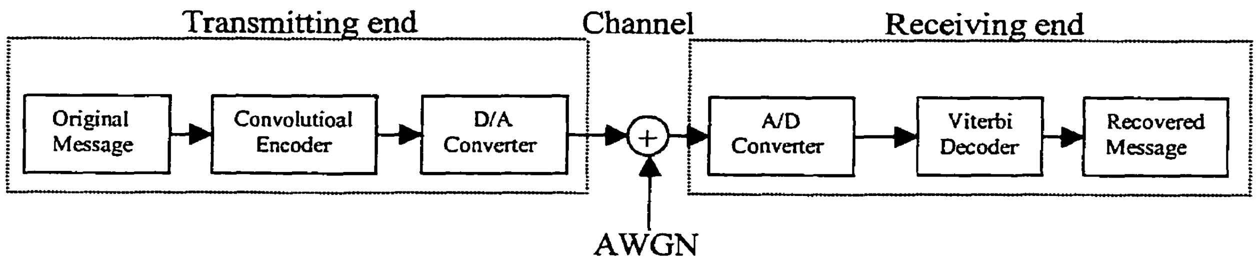


Figure-4.1. A Simplified Digital Communication Model

4.2 Verification Procedures

The verification procedure is shown in Figure-4.2. It consists of 3 steps. Step 1 is creating the simulation environment, including generating a random message, convolutional encoding, adding noise and quantization. Step2 is VHDL simulation and production of the decoded message. Step 3 is comparison of the original message with the decoded message, accumulating the message length and the error numbers and evaluating the BER for each specific E_s/N_0, if the error numbers are greater than 200. (Selecting 200 errors to calculate the BER is for the purpose of statistical accuracy.) Step1 and step 3 are implemented with C/C++ programs. Synopsys "vhdlsim" performs step 2 with a VHDL testbench. A Perl program under Unix integrates these three steps, controlling the executions of the C/C++ programs and vhdlsim.

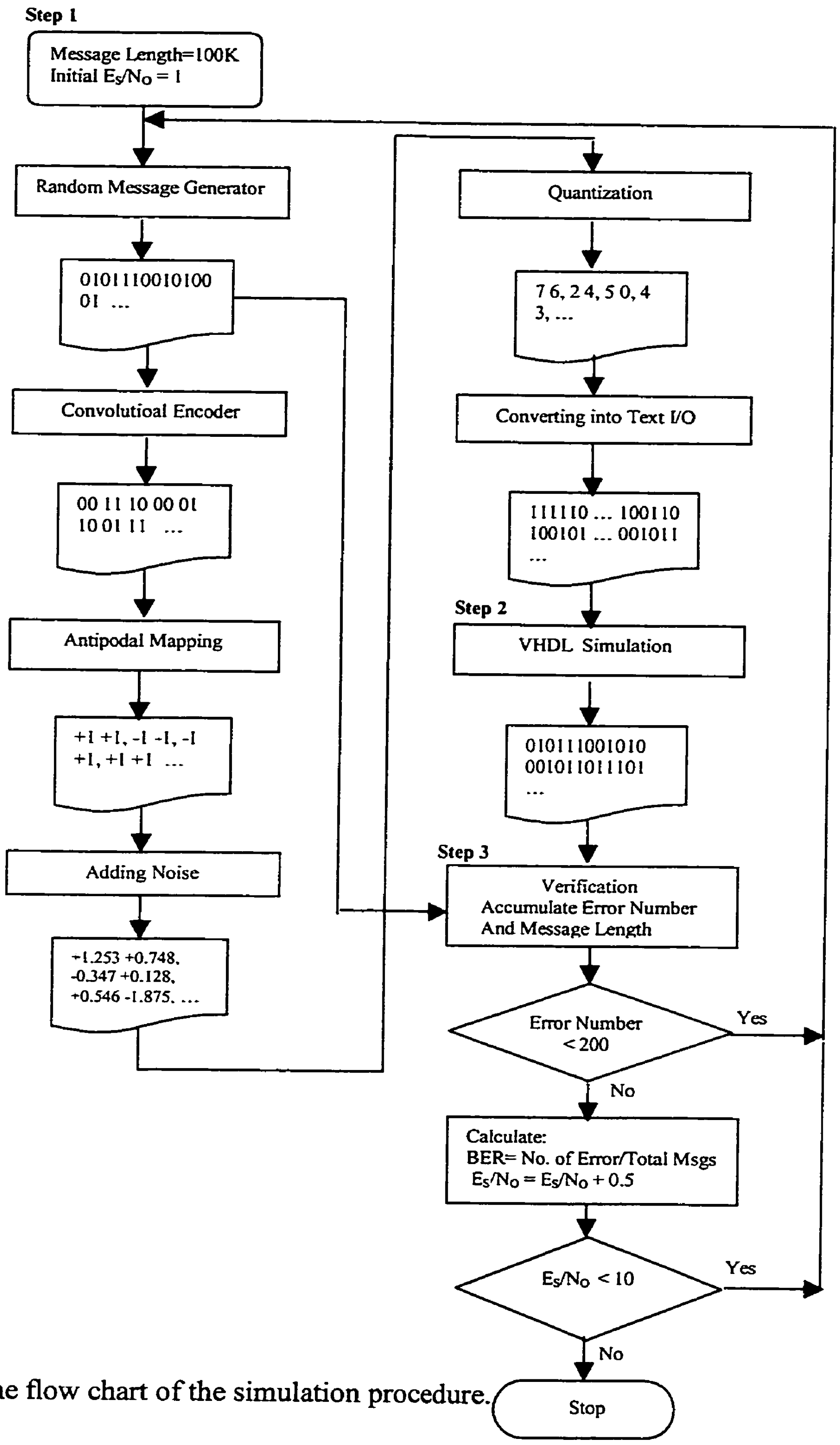


Figure-4.2 .The flow chart of the simulation procedure.

4.3 Simulation Environment

The *Random Message Generator* creates a stream of random binary data. Its length can be changed by an input argument. This stream of data is written into a "data.dat" file in ASCII code format for later comparison. The generation of the random binary data is as follows: Use the `rand()` function in C++ to generate a uniformly distributed random number between 0 and RAND_MAX. If the random number is greater than half of the range of the random variable, we assume it to be 1, otherwise, 0.

The *Convolutional Encoder* performs the specified convolutional encoding. In this design, it converts the message into the channel symbols of the (2, 1, 3) code with $g_1 = 111$ and $g_0 = 101$.

Antipodal Mapping converts ones and zeroes of channel symbols into antipodal base-band signals. Here, we assume that it translate zeroes to +1s and ones to -1s.

Adding Noise to the antipodal signals involves generating Gaussian random numbers, scaling the numbers according to the desired energy per symbol to noise density ratio, E_s/N_0, and adding the scaled Gaussian random numbers to the antipodal signals.

Since the C++ library only provides a uniform random number generator, `rand()`, we had to make use of the relationships among the uniform, Rayleigh, and Gaussian distributions[13], Given a uniform random variable $\boldsymbol{U}$ in (0, 1), a Rayleigh random variable $\boldsymbol{R}$ can be obtained by:

$$\boldsymbol{R} = \sqrt{2\sigma^2 \ln(1/(1-\boldsymbol{U}))} = \sigma\sqrt{2\ln(1/(1-\boldsymbol{U}))},$$

where σ^2 is the variance of the Rayleigh random variable, and Gaussian random variable $\boldsymbol{G}$ can be obtained by:

$$G = R \cos U',$$

where U' is another uniform random variable in $(0, 2\pi)$.

In the AWGN channel, the channel symbols are corrupted by additive noise, $n(t)$, which has the power spectrum $No/2$ watts/Hz. The variance σ^2 of this noise is equal to $No/2$. If we set the energy per symbol E_S equal to 1, then $E_S/N_O = 1/2\sigma^2$ and $\sigma = \sqrt{1/(2(E_S/N_O))}$. thus, given the desired Es/No, the standard deviation of the additive white Gaussian noise (AWGN) can be found. This standard deviation of the AWGN can be used to generate Gaussian random variables to simulate the noise. Adding this noise to the antipodal signal produces the noisy signal.

An ideal Viterbi decoder would work with infinite precision, or at least with floating-point numbers. In practical systems, the received channel symbols are usually quantized with one or a few bits of precision in order to reduce the complexity of the hardware. If the received channel symbols are quantized to one-bit precision ($< 0\text{V} = 1$, $\geq 0\text{V} = 0$), the result is called *hard-decision* data. If the received channel symbols are quantized with more than one bit of precision, the result is called *soft-decision* data. A Viterbi decoder with soft decision data inputs quantized to three or four bits of precision can perform about 2 dB better than one working with the hard-decision input [14]. The usual quantization precision is three bits. More bits provide little additional improvement [15]. In our design, every noisy signal is mapped into a 3-bit digital symbol. We assume the received signal levels in the absence of noise are $-1\text{V} = 1$, $+1\text{V} = 0$. Since the channel is modeled as additive white noise with Gaussian distribution and the power spectrum $No/2$ watts/Hz, the received signal has the mean and standard deviation:

$$\mu_n = \pm 1\ , \quad \sigma_n{}^2 = No/2.$$

A uniform, three-bit soft decision quantizer has best performance if the decision regions are given by:

$$D = q \times \sigma_n ,$$

The relationship of D to the quantizer decision regions is shown in Figure-4.3. The q value should be in the range of 0.45 to 0.71 [14]. If we take $q = 0.5$, then

$$D = 0.5 \times \sqrt{1/2\, E_S/No.}$$

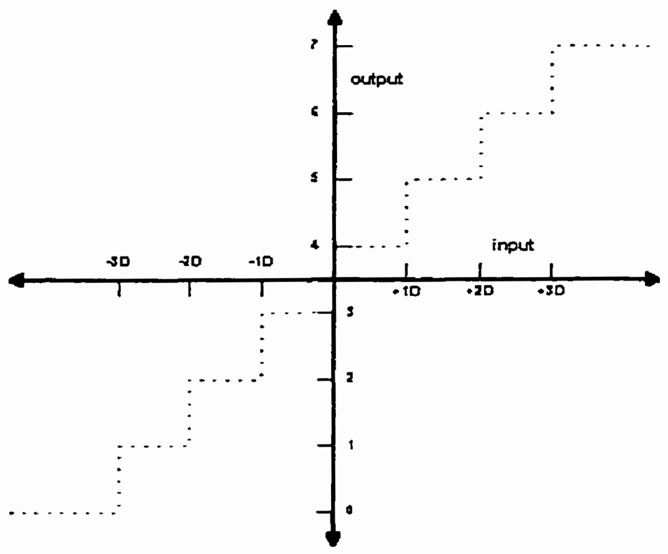


Figure-4.3. The uniform quantization

For the quantified channel symbols to be read by the VHDL testbench, they must be converted into a text file in ACSII code format. The file is named "quntzd.dat".

The C/C++ source code performing from the generating of the random message (in data.dat) to producing the quantified channel symbols file (in quntzd.dat) is provided in Appendix A. Appendix B is an example of the "data.dat" file. There is no return character between the rows in this file. Appendix C is an example of the "quntzd.dat" file. The length of both files is the number of bits and can be controlled by an input argument.

4.4 VHDL Simulation

The Viterbi decoder and its testbench are described in VHDL. They can be simulated with Synopsys "vhdlsim". The testbench reads the "quntzd.dat" file as the stimuli for the Viterbi decoder simulation. The simulation result with Synopsys "vhdlsim" is the decoded data which is output to a file, "decoded.dat", an example of which is shown in Appendix D. Before synthesizing, only functional simulation is performed. After placement and routing, timing simulation can also be done for the back-annotated VHDL file.

4.5 Evaluating the Performance

Evaluation includes comparing the "data.dat" file with "decoded.dat" file, accumulating the message length and error numbers, and if the error number is greater than 200, calculating the BER at the current E_S/N_O with the formula:

BER = Error number / total message length.

Then the E_S/N_O is increased by 0.5 and the process returns to step1 in Figure-4.2. After obtaining every BER versus E_S/N_O from 1 to 5, the curve in Figure-4.4 can be plotted using MatLab.

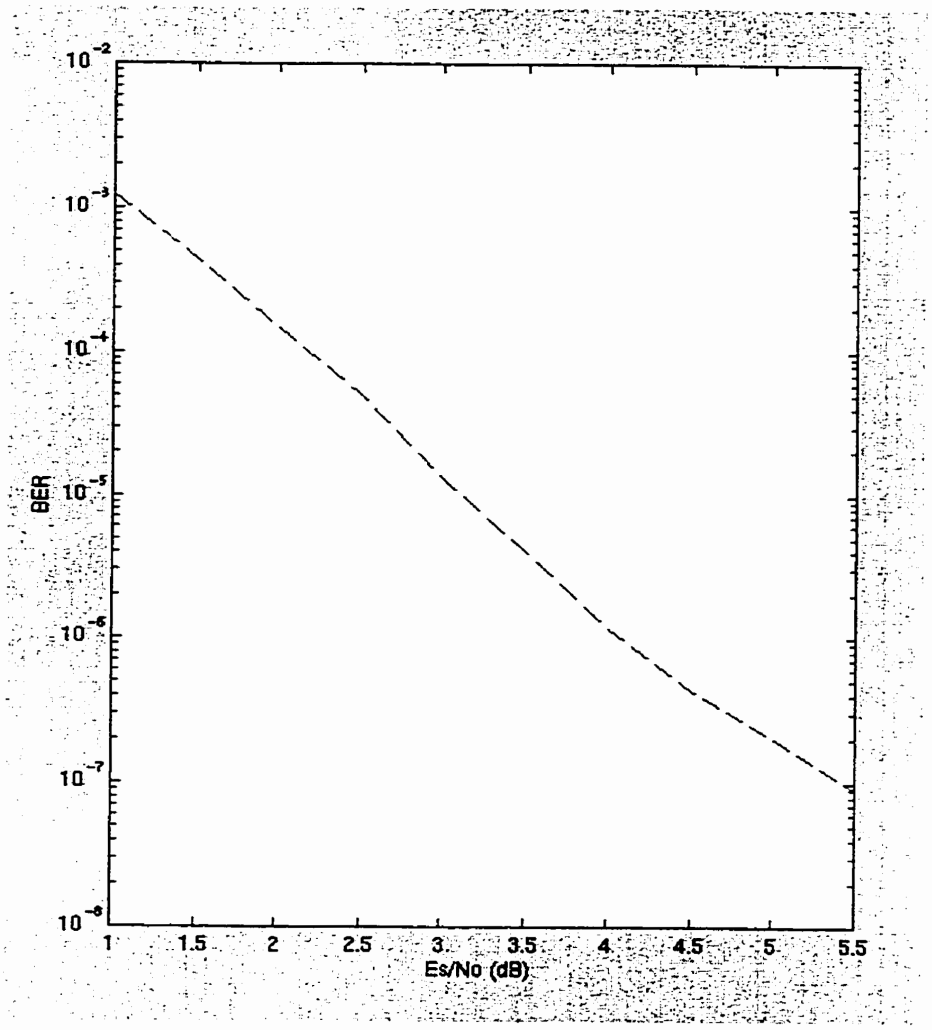


Figure-4.4 BER versus E_S/N_O

Chapter 5

Implementation of the Design

5.1 Implementation Design Flow

The implementation design flow is shown in Figure-5.1 and consists of the following steps:

1. Start with a functional VHDL description of the design.

2. Using Design Analyzer or Design Compiler in Synopsys check if there are any errors in the VHDL file and determine if the description can be synthesized.

3. After determining that the circuit can be synthesized, simulate to verify that the VHDL description performs the desired function. If it does not work as desired, modify the VHDL code.

Repeat steps 2 and 3 until the VHDL source code is functionally correct and can be synthesized. All the VHDL source code is provided in Appendix D.

4. Before synthesis, the compiling strategy and technology libraries must be defined. In this design, obtaining the fastest speed was chosen as the compilation strategy. The Xilinx-Virtex series is selected as the technology library. In Figure-5.1, these tasks are represented as inputs to the synthesis step.

5. Synthesize the VHDL description into a technology-specific gate-level netlist.

6. After synthesizing the design, obtain the timing report and FPGA area report as (shown in Appendix E). If the reports do not meet the design goals, they must be analyzed to

determine the modifications to be made. This process is iterative and might require modifying the original VHDL code or trading-off between the circuit speed and the area.

7. Save the synthesized design as a SEDIF file, which can be recognized by NGDBuild in Xilinx Alliance. Use the DC2NCF program to translate the Synopsys constraint DC file to a Netlist Constraints File (NCF). Step 2 to 7 are performed through a script file provided in Appendix F.

8. Run NGDBuild on the SEDIF file to create an NGD file. At the meantime, input the UCF (User Constraint File), which limits the longest delay between the stages not exceeding 12 ns, to NGDBuild.

9. Run the MAP program on the NGD file to create a mapped NCD file.

10. Run PAR on the NCD file to place and route the design.

11. Run NGDAnno on the routed NCD and NGM files to create an NGA file.

12. Run NGD2VHDL on the NGA file to create a VHDL file for simulation with back annotation. This step also creates a Standard Delay Format (SDF) file containing timing information. Step 8 to 12 are performed by a script provided in Appendix G.

13. Analyze VHDL code created in step 12 using Synopsys "vhdlan" command and then use "vhdlsim" to simulate the back-annotated design. The back-annotated simulation is run by executing a script provided in Appendix H.

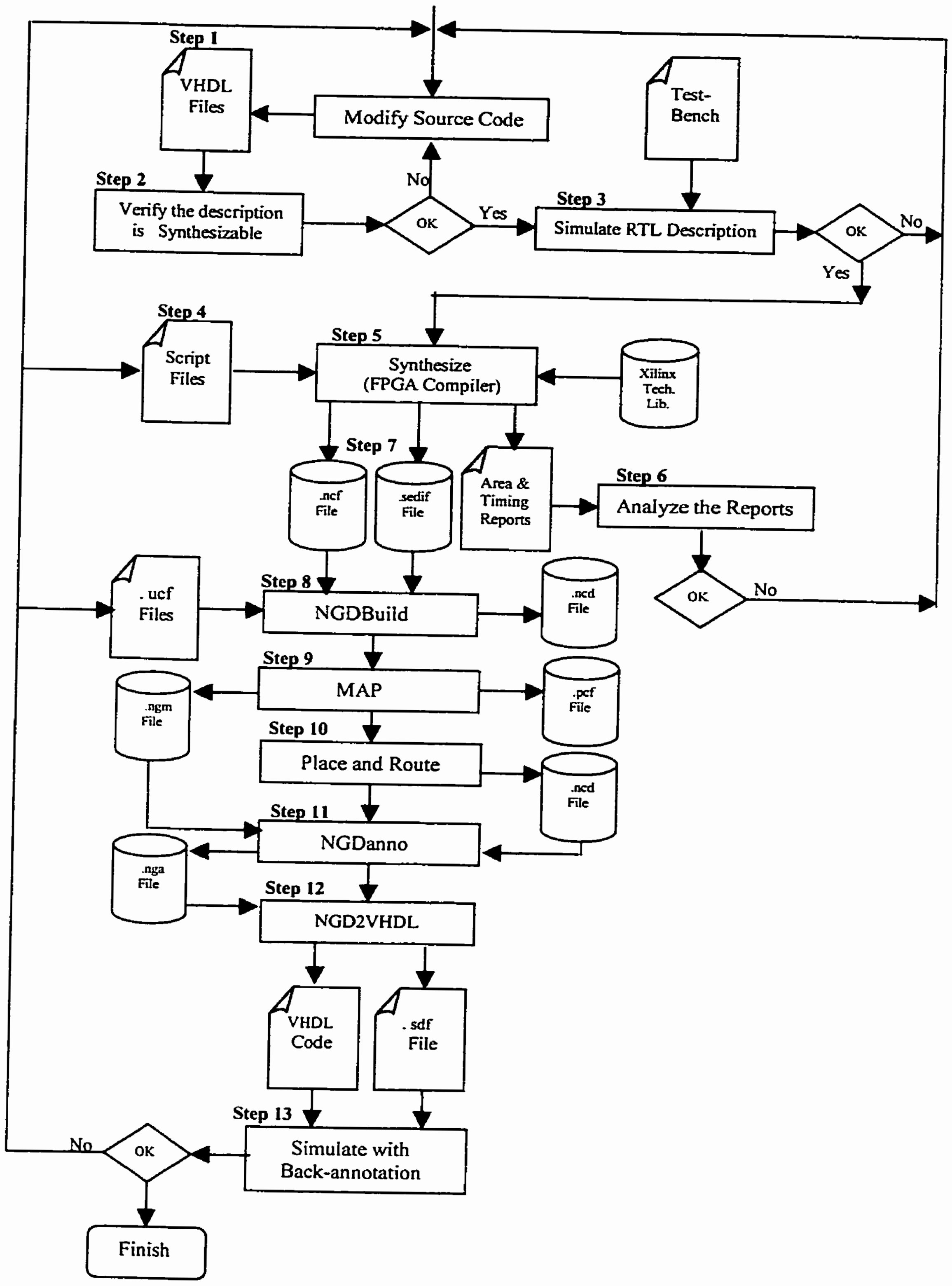


Figure-5.1 Implementation flow

5.2 VHDL Hierarchy

Hierarchical design in VHDL can make the source code reusable and easier to read and debug. Based on the required functional units and pipeline stages, the hierarchical structure for the design (as shown in Figure-5.2) was constructed. Each block represents a VHDL entity, that can be independently tested and synthesized to obtain the timing and FPGA area reports.

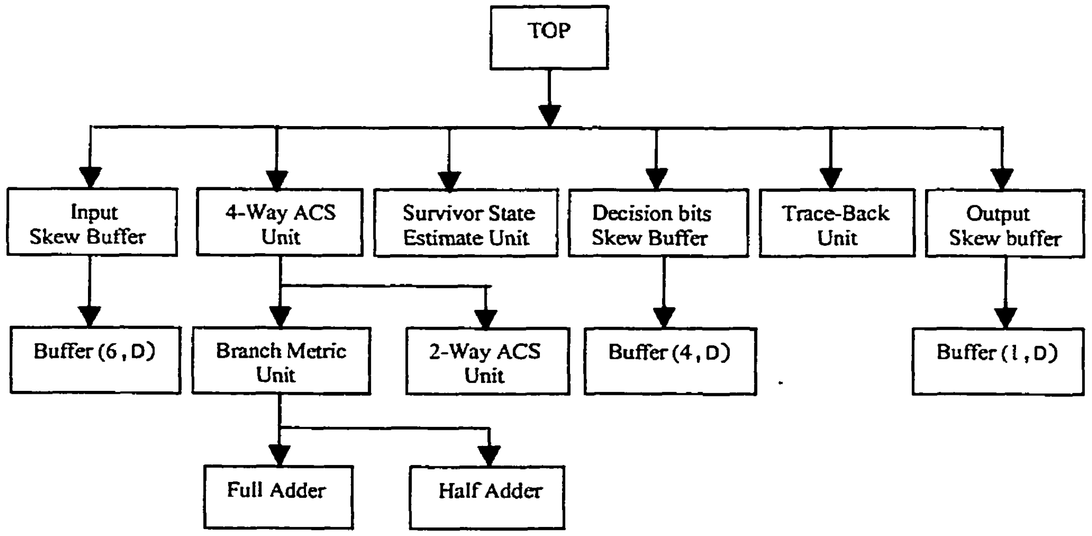


Figure-5.2 VHDL hierarchy

TOP is the top-level block of the Viterbi decoder. It is a pure structural modeling architecture, which is mainly composed of three VHDL statements: component declaration, signal declaration, and component instantiation. Through compiling TOP with the Synopsys Design Compiler, the critical timing path can be found and the total area and power consumption can be estimated for the design. If these parameters do not meet the requirements, the component that effects these parameters can in turn be modified.

The 4-way ACS unit is also a pure structural modeling architecture. It instantiates the branch metric unit and the 2-way ACS unit, combining Figure-3.9 and Figure-3.11 together. There are two pipeline stages in the 4-way ACS unit. Putting the branch metric unit and the 2-way ACS unit in an entity makes the top level architecture simple and readable.

The buffer (width, depth) component is a parameterized entity. It is instantiated as pipeline buffers for channel symbols, decision bits and output bits by giving the specific width and depth parameters required.

The branch metric unit instantiates the half adder component and the full adder component to realize the arithmetic addition instead of just using "+" to perform it. The reason for this is explained in section 5.4 under coding style consideration.

The rest of the blocks are behavioral modeling structures in which concurrent signal assignment statements and logic algebra expressions are used. These expressions will automatically instatiate Xilinx primitives or macros to implement the functions.

5.3 Xilinx Virtex architecture

To get the best VHDL coding style, the Virtex architecture must be understood. The basic building block of the Virtex CLB is the logic cell (LC). In Figure-5.3, the CLB contains four LCs organized as two slices. Figure-5.4 shows a more detailed view of a single slice (i.e. half the CLB).

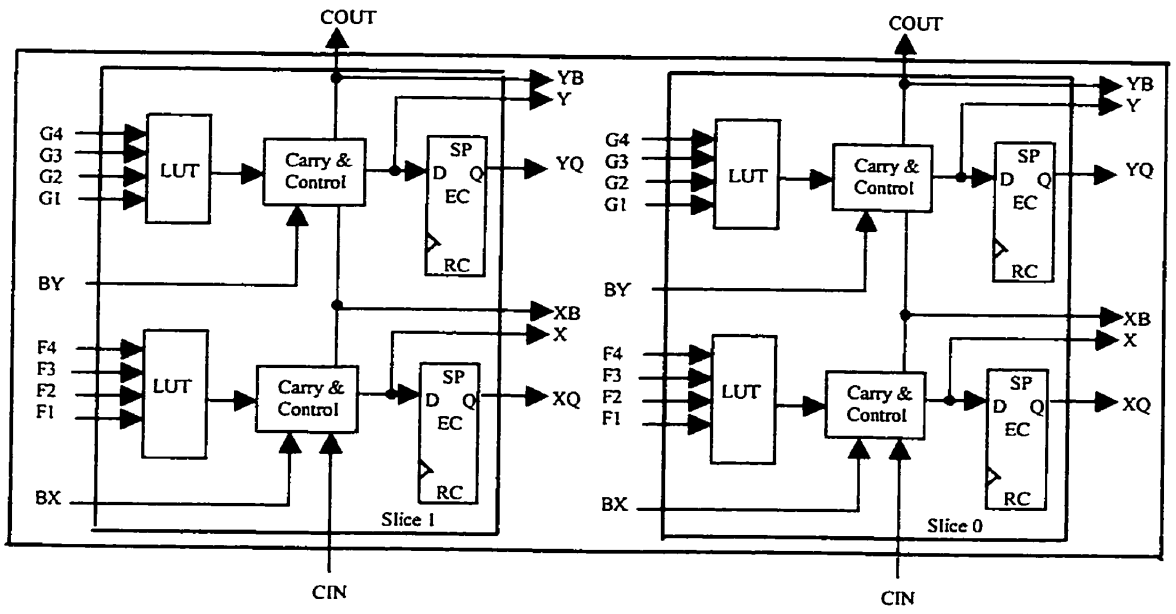


Figure-5.3 2-Slice Virtex CLB

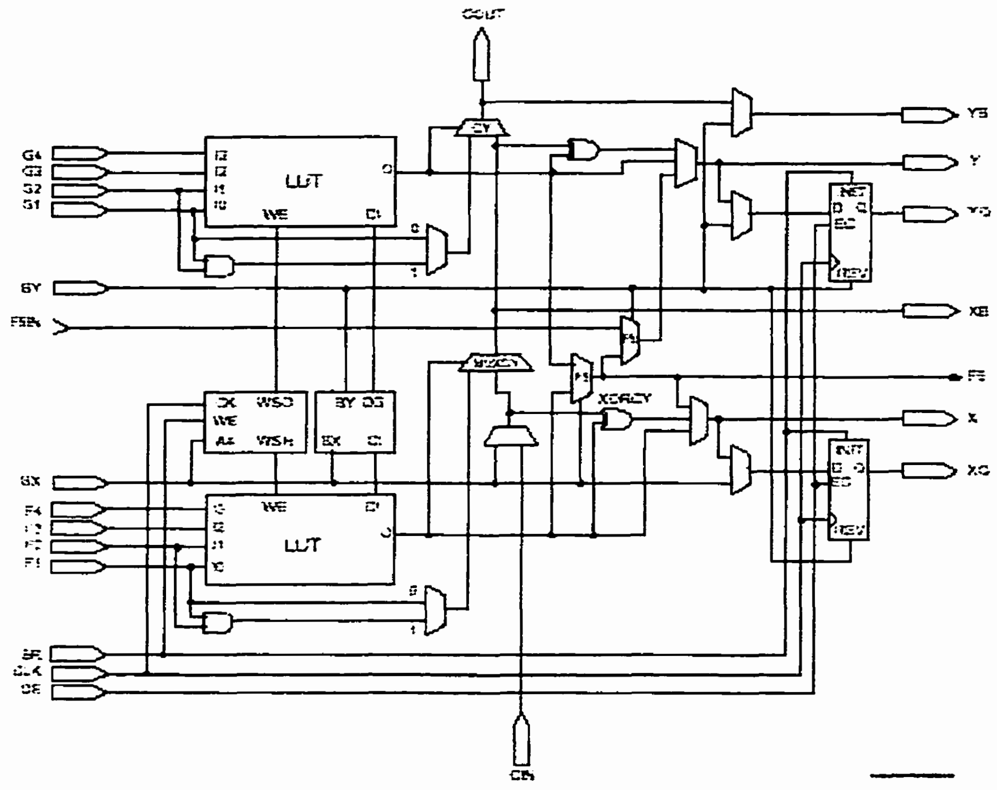


Figure-5.4 Block Diagram of a single-slice Virtex CLB

An LC includes a 4-input function generator, carry logic and a storage element. The function generator is implemented as a 4-input look-up table (LUT) and can implement any 4-input logic function. The output from the 4-input LUT in each LC drives both the CLB output and the D-input of the flip-flop. Each additional 2-input dedicated AND gate per LUT implements an efficient 1-bit multiplier.

The most relevant feature of the CLB is the dedicated carry logic which is required to implement fast, efficient arithmetic functions shown in Figure-5.5. There are two separate carry chains in the Virtex CLB, one per slice. The height of the carry chain is two bits per CLB. The logic consists of a 2-input MUX (MUXCY) and an XOR (XORCY) gate. The XOR gate allows a 1-bit full adder to be implemented within a logic cell (LC). The dedicated carry path is used to cascade LUT functions for implementing wide logic functions. This reduces logic delays due to the decreased number of logic levels even for very high fan-in function [16].

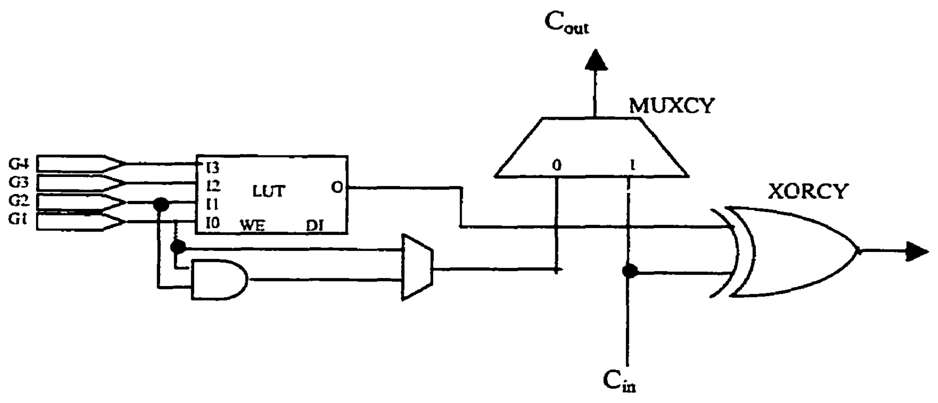


Figure-5.5 Carry Logic Diagram

5.4 Coding Style Considerations

When designing with VHDL, it is important to consider the coding style. Different coding styles will produce different synthesis results for a specific technology. The Xilinx tool has a technology specific library that takes advantage of specialized logic on their devices. The Synopsys tool recommends that you use this library because it provides improved performance and increases the accuracy of the area and timing predictions in the Synopsys environment. But you still should take each particular application into consideration. For example, in this design, if we use the VHDL code shown in Figure-5.7 to describe the branch metric unit, after synthesizing we will get the area and timing report shown in Figure-5.8 and schematic diagram shown in Figure-5.9. Although the "add" operation symbol "+" will automatically instantiate a macro for a 4-bit adder in Xilinx library, the synthesized results is not the minimum area and fastest speed. The reason is that there are 6 inverters before the add operation. They are implemented with 6 separate LUTs and can not be combined into the LUTs that implement the "+" macros because macros are untouchable. As a result, the area and the delay increase.

Figure-5.7

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity bm is
   port (
          sym: in std_logic_vector (5 downto 0);
          bm00: out std_logic_vector (3 downto 0);
          bm11: out std_logic_vector (3 downto 0);
          bm10: out std_logic_vector (3 downto 0);
          bm01: out std_logic_vector (3 downto 0);
          clk, reset: in std_logic);
end bm;

architecture arch_bm of bm is

signal nsym: std_logic_vector(5 downto 0);
signal bm00t,bm01t,bm10t,bm11t:std_logic_vector(3 downto 0);


begin
getnsym: for i in 0 to 5 generate
        nsym(i)<=not sym(i);
end generate getnsym;

bm00t<= ('0'&sym(5 downto 3)) + ('0'&sym(2 downto 0));
bm01t<= ('0'&sym(5 downto 3)) + ('0'&nsym(2 downto 0));
bm10t<= ('0'&nsym(5 downto 3)) + ('0'&sym(2 downto 0));
bm11t<= ('0'&nsym(5 downto 3)) + ('0'&nsym(2 downto 0));

process(clk,reset)
 begin
  if reset='1' then
  bm00<="0000";
  bm11<="0000";
  bm10<="0000";
  bm01<="0000";
 elsif clk'event and clk='1' then  --CLK rising edge
  bm00<=bm00t;
  bm01<=bm01t;
  bm10<=bm10t;
  bm11<=bm11t;
 end if;
 end process;

end arch_bm;
```

Figure-5.8

```
****************************************
Report : fpga
Design : bm
Version: 2000.05
Date   : Wed Oct  4 16:57:54 2000
****************************************

  LUT FPGA Design Statistics

  * Core Cell Statistics *
  Number of 2-input LUT cells:      6
  Number of Core Flip Flops:       16
  Number of Core 3-State Buffers:   0
  Number of Other Core Cells:      16
  Total Number of Core Cells:      70

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : bm
Version: 2000.05
Date   : Wed Oct  4 16:57:54 2000
****************************************

Operating Conditions: WCCOM   Library: xfpga_virtex-6
Wire Load Model Mode: top

 Startpoint: sym<3> (input port)
 Endpoint: bm10_reg<3>
        (rising edge-triggered flip-flop clocked by clk)
 Path Group: clk
 Path Type: max

 Des/Clust/Port     Wire Load Model      Library
 ------------------------------------------------
 bm                 xcv300-6_avg         xfpga_virtex-6

 Point                                              Incr      Path
 -----------------------------------------------------------------
 clock (input port clock) (rise edge)               0.00      0.00
 input external delay                               0.00      0.00 f
 sym<3> (in)                                        0.00      0.00 f
 U54/O (LUT2)                                       2.08      2.08 f
 add_28/plus/plus/A<0> (bm_xdw_add_4_1)             0.00      2.08 f
 add_28/plus/plus/A_LUT/LO (DWLUT2_L)               0.58      2.66 f
```

```
add_28/plus/plus/A_CY/LO (MUXCY_L)           0.90    3.56 f
add_28/plus/plus/A_CY_1/LO (MUXCY_L)         0.05    3.61 f
add_28/plus/plus/A_CY_2/LO (MUXCY_L)         0.05    3.66 f
add_28/plus/plus/A_XOR_3/O (XORCY)           1.13    4.79 r
add_28/plus/plus/S<3> (bm_xdw_add_4_1)       0.00    4.79 r
bm10_reg<3>/D (FDC)                          0.00    4.79 r
data arrival time                                    4.79
```

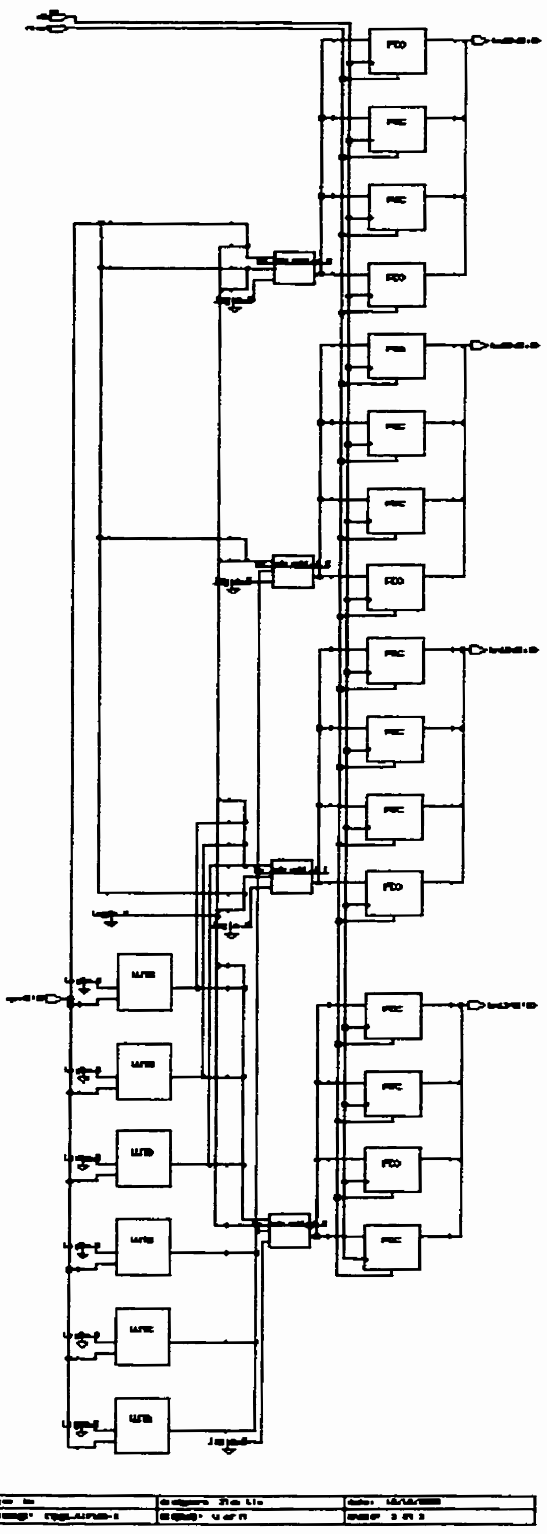


Figure-5.9 The schematic diagram for BM using "+" for the add operation

If we use the VHDL code provided in Appendix D to describe the branch metric unit in which a 3-bit adder is composed of a 1-bit half-adder and two 1-bit full-adders, after synthesizing we will get a better area report and timing report as shown in figure-5.10. The corresponding schematic diagram is shown in Figure-5.11. The reason is that 6 inverters will be combined with the half-adders and the full-adders into LUTs in the process of compiling since all of the instantiations can be ungrouped.

Figure-5.10

```
****************************************
Report : fpga
Design : qunt2bm
Version: 2000.05
Date   : Sun Sep  3 11:22:36 2000
****************************************

 LUT FPGA Design Statistics

 * Core Cell Statistics *
 Number of 2-input LUT cells:      5
 Number of 3-input LUT cells:      6
 Number of 4-input LUT cells:     10
 Number of Core Flip Flops:       16
 Number of Core 3-State Buffers:   0
 Number of Other Core Cells:       0
 Total Number of Core Cells:      37


****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : qunt2bm
Version: 2000.05
Date   : Sun Sep  3 11:22:36 2000
****************************************

Operating Conditions: WCCOM   Library: xfpga_virtex-6
Wire Load Model Mode: top

 Startpoint: sym<1> (input port)
 Endpoint: bm00_reg<2>
          (rising edge-triggered flip-flop clocked by clk)
 Path Group: clk
 Path Type: max
```

```
Des/Clust/Port   Wire Load Model    Library
-----------------------------------------------
qunt2bm          xcv300-6_avg       xfpga_virtex-6

Point                                   Incr    Path
------------------------------------------------------
clock (input port clock) (rise edge)    0.00    0.00
input external delay                    0.00    0.00 r
sym<1> (in)                             0.00    0.00 r
U52/O (LUT4)                            1.78    1.78 r
U64/O (LUT3)                            1.28    3.06 r
bm00_reg<2>/D (FDC)                     0.00    3.06 r
data arrival time                               3.06
```

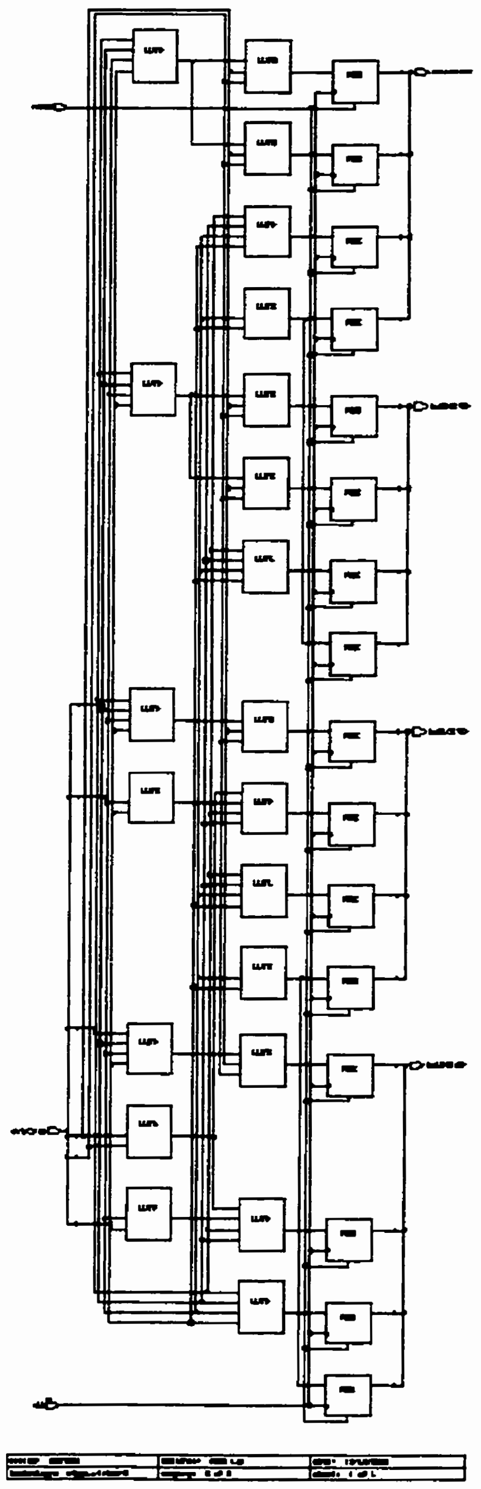


Figure-5.11 The schematic diagram for BM without using "+" for the add operation

For the 2-way ACS unit, Since there is not any logic gates needed before add algorithm or between add and compare operations, we can use "+" and " <=" to describe these two operations respectively. The synthesized result is better than that described with half adders and full adders.

5.5 Choosing the Xilinx Part

After compiling the TOP entity, the FPGA area and timing report shown in Figure-5.12 generated. Normally the area report is optimistic because the layout tool uses additional CLBs as feedthroughs for routing. Although the Virtex device XCV200E has 4704 flip-flops, which is more than we need in the design (3598 flip-flops) , after mapping the design into Xilinx CLB, we can see the number of slices are not enough for the design. So the Virtex device XCV300E (it has 3072 slices while XCV200E has only 2352 slices) had to be selected to implement this design.

Figure-5.12

```
****************************************
Report : fpga
Design : top
Version: 2000.05
Date   : Mon Oct  9 11:27:25 2000
****************************************

   LUT FPGA Design Statistics
   ---------------------------

   * Core Cell Statistics *
   Number of 2-input LUT cells:       137
   Number of 3-input LUT cells:       817
   Number of 4-input LUT cells:       276
   Number of Core Flip Flops:         3598
   Number of Core 3-State Buffers:    0
   Number of Other Core Cells:        2346
```

```
    Total Number of Core Cells:         11261

    * Port Statistics *
    Number of Input Ports:              74
    Number of Output Ports:             12
    Number of Bi-directional Ports:     0
    Total Number of Ports:              86

    * Pad Cell Statistics *
    Number of Input Pads:               74
    Number of Output Pads:              12
    Number of Clock Pads:               0
    Total Number of Pads Cells:         86


****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : top
Version: 2000.05
Date   : Sun Nov 12 16:56:56 2000
****************************************

Operating Conditions: WCCOM   Library: xfpga_virtexe-7
Wire Load Model Mode: top

  Startpoint: stage_0/acs3/sm_reg<0>
              (rising edge-triggered flip-flop clocked by clock)
  Endpoint: stage_1/acs1/sm_reg<0>
            (rising edge-triggered flip-flop clocked by clock)
  Path Group: clock
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  top                xcv300e-7_avg         xfpga_virtexe-7

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clock (rise edge)                                 0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  stage_0/acs3/sm_reg<0>/C (FDC)                          0.00       0.00 r
  stage_0/acs3/sm_reg<0>/Q (FDC)                          2.30       2.30 f
  stage_0/acs3/sm<0> (ACS)                                0.00       2.30 f
  stage_0/gm1_out<0> (acs4)                               0.00       2.30 f
  stage_1/gm1_in<0> (acs4)                                0.00       2.30 f
  stage_1/acs1/sm1<0> (ACS)                               0.00       2.30 f
  stage_1/acs1/add_23/plus/plus/A<0> (ACS_xdw_add_8_0)
                                                          0.00       2.30 f
  stage_1/acs1/add_23/plus/plus/A_LUT/LO (DWLUT2_L)       0.43       2.73 f
  stage_1/acs1/add_23/plus/plus/A_CY/LO (MUXCY_L)         0.43       3.16 f
  stage_1/acs1/add_23/plus/plus/A_XOR_1/O (XORCY)         2.02       5.18 r
  stage_1/acs1/add_23/plus/plus/S<1> (ACS_xdw_add_8_0)
                                                          0.00       5.18 r
  stage_1/acs1/lte_30/leq/leq/A<1> (ACS_xdw_comp_uns_8_0)
                                                          0.00       5.18 r
  stage_1/acs1/lte_30/leq/leq/A_LUT_1/LO (DWLUT2_L)       0.43       5.61 r
  stage_1/acs1/lte_30/leq/leq/A_CY_1/LO (MUXCY_L)         0.43       6.04 r
  stage_1/acs1/lte_30/leq/leq/A_CY_2/LO (MUXCY_L)         0.07       6.11 r
```

```
stage_1/acs1/lte_30/leq/leq/A_CY_3/LO (MUXCY_L)          0.07       6.18 r
stage_1/acs1/lte_30/leq/leq/A_CY_4/LO (MUXCY_L)          0.07       6.25 r
stage_1/acs1/lte_30/leq/leq/A_CY_5/LO (MUXCY_L)          0.07       6.32 r
stage_1/acs1/lte_30/leq/leq/A_CY_6/LO (MUXCY_L)          0.07       6.39 r
stage_1/acs1/lte_30/leq/leq/A_CFY/O (MUXCY)              1.97       8.36 r
stage_1/acs1/lte_30/leq/leq/A_GE_B (ACS_xdw_comp_uns_8_0)
                                                         0.00       8.36 r
stage_1/acs1/U43/O (LUT3)                                1.33       9.69 r
stage_1/acs1/sm_reg<0>/D (FDC)                           0.00       9.69 r
data arrival time                                                   9.69
```

The timing values reported are the pre-layout values. Pre-layout delays are evaluated by a statistical model, which is an approximation. The pre-layout results usually are pessimistic and typically differ from post-layout by 10 to 15 percent if the average wire load model is used [17]. From the timing report we can see that the data arrival time for the longest path (i.e. critical path) is 9.69 ns, which is smaller than our design aim of 12 ns. This means the selection of the speed grade "-8" is reasonable. So in the next step, we use this synthesized result to place and route targeting XVC300E-8.

5.6 Placement and Routing

Before place and route, two other processes need to be done. First, the netlist file needs to be converted into an NGD (Native Generic Database) file. NGDBuild performs this step. It reduces all components in the design to NGD primitives, checks the design by running a Logical DRC (Design Rule Check) on the converted design and writes an NGD file as output. In this step, the User Constraints File (.ucf file) needs to be input to NGDBuild. In this design we specify a timing constraint that limits the longest time delay to12 ns in "top.ucf" file. The second step is mapping the NGD file to a Xilinx FPGA. MAP executes this step. It maps the logic in the design to the components (logic cells, I/O cells, and other components) in the target device. The output is an NCD (Native Circuit

Description) file – a physical representation of the design in terms of the components in the Xilinx Virtex chip. The NCD file can then be placed and routed.

There are two different design flows that can be used to implement the placement and routing:

1. First run the PAR to place and route. If there are a few paths do not meet your requirement, use Floorplanner and/or FPGA Editer to modify them manually. If there are too many paths that do not meet your requirement, modify the user constraint file and run the PAR again or choose a higher speed grade part.
2. First run Floorplanner to manually placing the selected logic into the resources of the target device. Next, run MAP and PAR to fit the design into the target FPGA using the Floorplan constraints.

In our design, the first method was used, because there were too many critical paths existing between the consecutive stages so that the manual placement and routing through Floorplanner is hard to reach the expected result. After placement and routing, the PAR report is obtained as shown in Appendix I, which indicated that the maximum delay between the flip-flops, is 10.140 ns. Theoretically this means the chip can work at the clock cycle of 12 ns in the standard environment. The layout is shown in Figure-5.13.

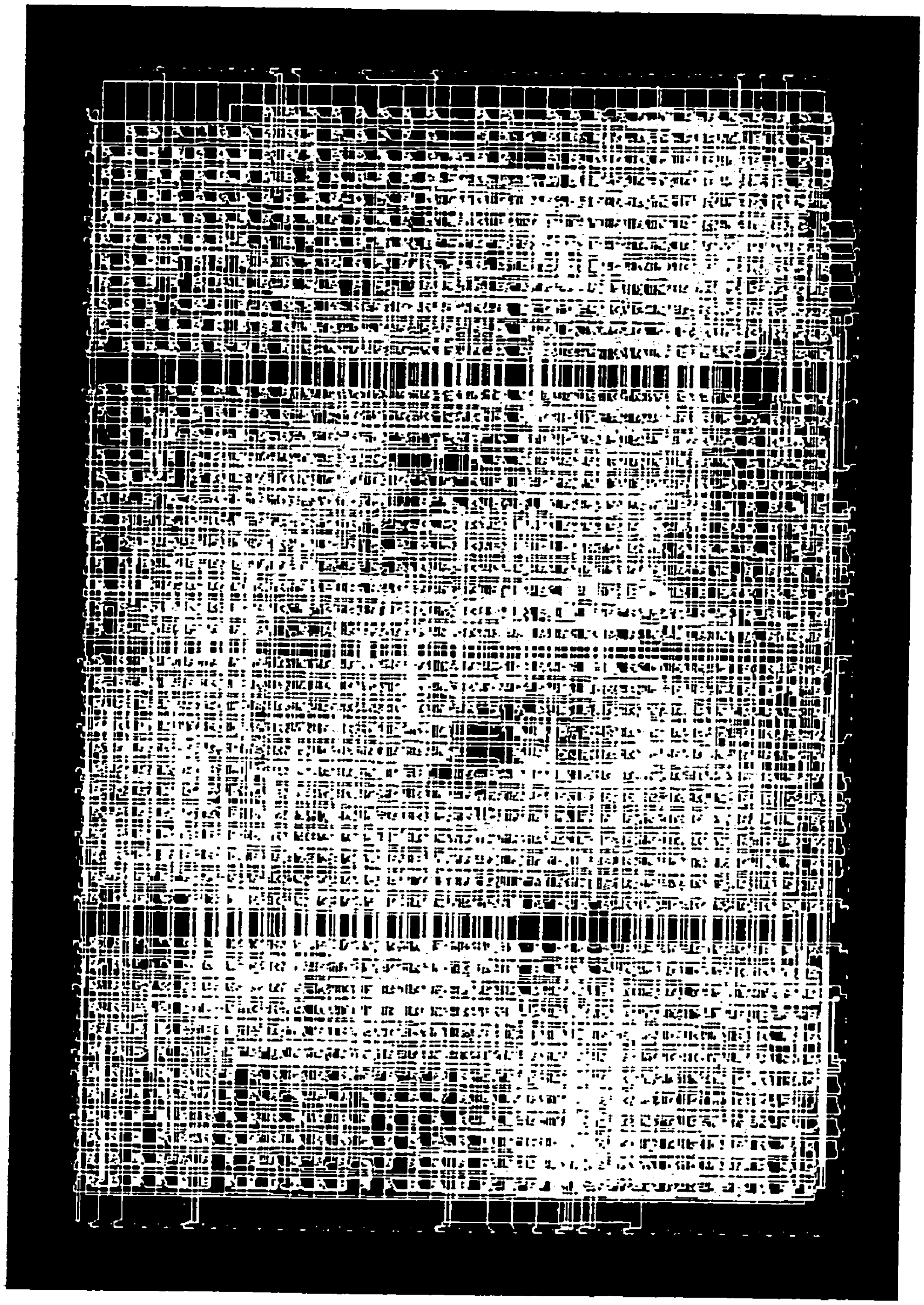


Figure-5.13 The layout of the FPGA for the Viterbi decoder

5.7 Back Annotation and Translation to VHDL

The back-annotation process generates a generic timing simulation model. In the Xilinx Development System, NGDAnno back-annotates timing information using the NCD file produced by PAR, and the NGM file produced by MAP. The NCD file, represents the physical design. The NGM file represents the logical design. NGDAnno distributes timing information associated with placement, routing, and block configuration from the physical NCD design file into the logical design represented in the NGM file. NGDAnno outputs an annotated logical design that has a .nga (Native Generic Annotated) extension. The NGA file then is input to NGD2VHDL, which converts the back-annotated file in Xilinx format into VHDL format for simulation. NGD2VHDL also produces the SDF (Standard Delay Format) file which is used by Synopsys simulation.

5.8 Simulation with Timing Delays

After NGD2VHDL produces the VHDL file and SDF file, we again use Synopsys "vhdlan" to compile the VHDL file and "vhdlsim" to simulate it with the SDF file. From Figure-5.12, we can see that the longest path is located between two adjacent ACS stages, which can be simplified as in Figure-5.14. The simulation waves in Figure-5.15 shows the longest path between two stages is about 10.8 ns. This again verifies that the design can work at 1/12ns= 83.3 MHz and therefore the throughput can reach 1 Gbit/s since each clock cycle 12 bits of output will be produced.

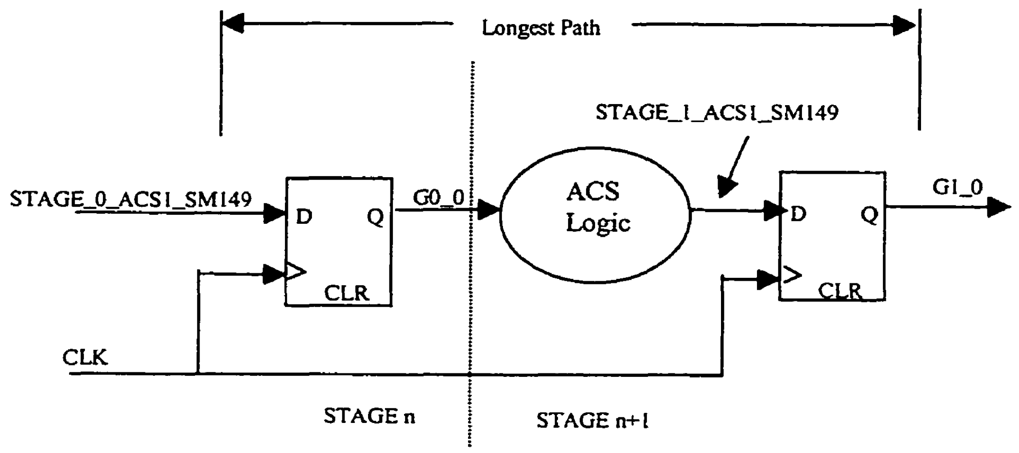
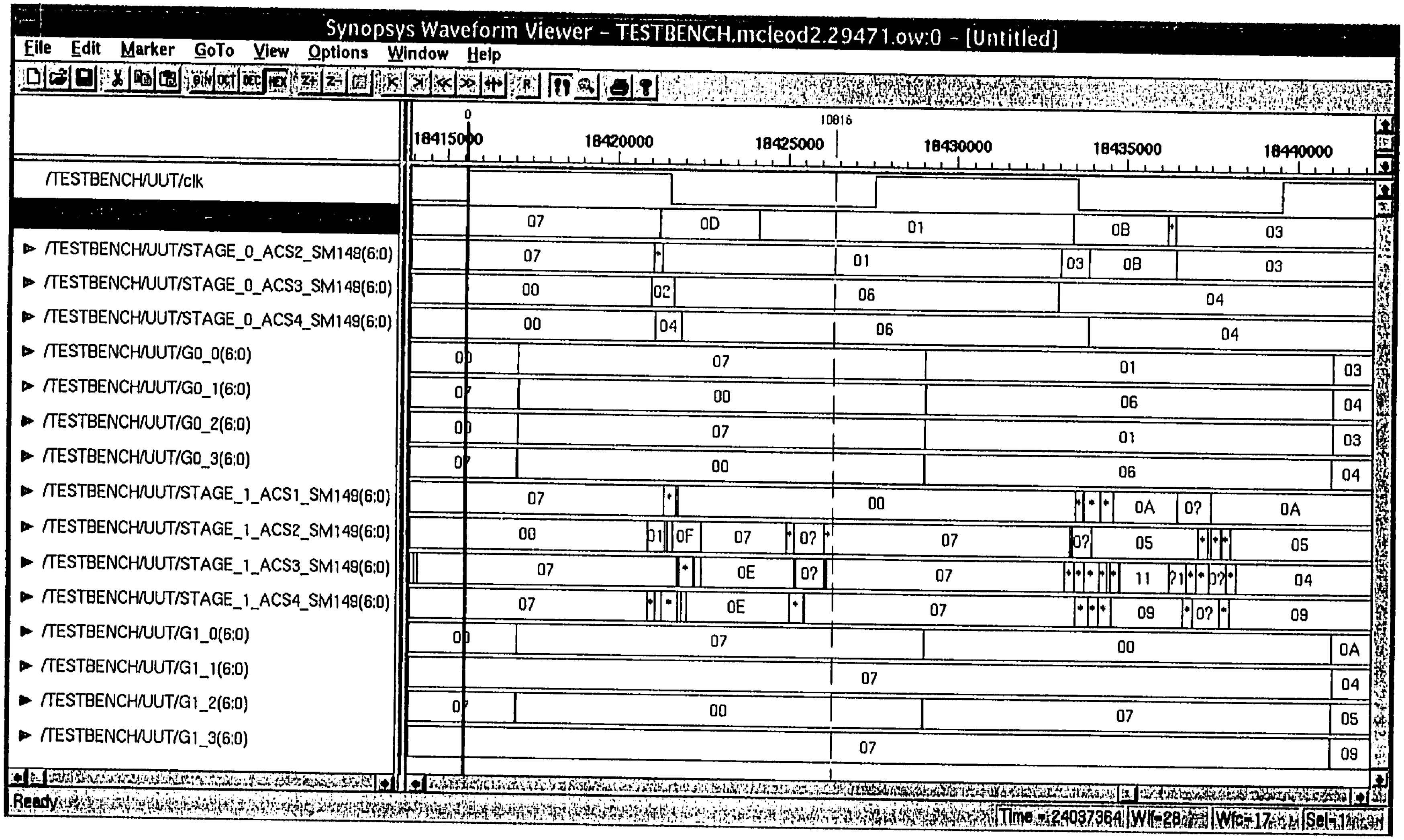


Figure-5.14 The longest path between two stages


Synopsys Waveform Viewer - TESTBENCH.mcleod2.29471.ow:0 - [Untitled]

File Edit Marker GoTo View Options Window Help

/TESTBENCH/UUT/clk
/TESTBENCH/UUT/STAGE_0_ACS2_SM149(6:0)
/TESTBENCH/UUT/STAGE_0_ACS3_SM149(6:0)
/TESTBENCH/UUT/STAGE_0_ACS4_SM149(6:0)
/TESTBENCH/UUT/G0_0(6:0)
/TESTBENCH/UUT/G0_1(6:0)
/TESTBENCH/UUT/G0_2(6:0)
/TESTBENCH/UUT/G0_3(6:0)
/TESTBENCH/UUT/STAGE_1_ACS1_SM149(6:0)
/TESTBENCH/UUT/STAGE_1_ACS2_SM149(6:0)
/TESTBENCH/UUT/STAGE_1_ACS3_SM149(6:0)
/TESTBENCH/UUT/STAGE_1_ACS4_SM149(6:0)
/TESTBENCH/UUT/G1_0(6:0)
/TESTBENCH/UUT/G1_1(6:0)
/TESTBENCH/UUT/G1_2(6:0)
/TESTBENCH/UUT/G1_3(6:0)

Ready Time = 24037364 Wf=28 Wfc=17 Sel=1

Chapter 6

Comparison with Existing Designs

Two designs with the same constraint length have been selected for comparison. Although differences in technology and design style make the comparison somewhat misleading, it still can be seen that doing this design is worthwhile. The comparison is summarized in Table-6.1.

Design	Constraint Length	Throughput (Mbit/s)	Coding gain @10^{-5} BER	Technology
[8]	3	600	Less than 3.4 dB	1.2μ CMOS
[10]	3	1000	3.4 dB	1.2μ CMOS
This thesis	3	1000	6. 2 dB	Xilinx Virtex XCV300E (0.18 μm CMOS)

Table-6.1 Comparisons with other Viterbi decoder designs

Gerhard Fettweis [8] designed a R=1/2, K=3 Viterbi decoder using the minimized method with 1.2μ CMOS technology in 1990. Its throughput is 600Mbit/s. The chip area is 170mm^2. In comparison, the minimized method is not a maximum likelihood algorithm because the estimates of the states at either end of the decode block are not based on all of the available data. A true maximum likelihood estimate is based on the entire observation interval and hence the coding gain of the sliding block Viterbi decoder method always upper bounds the minimized method for the same interval parameters.

Peter J. Black and Teresa H.-Y. Meng [10] designed a Viterbi decoder of (2,1,3) code using the hybrid (forward and backward) processing method with 1.2μ double-metal

CMOS technology in1996. Its throughput is 1Gbit/s. The chip area is 81mm^2. It has 3.4 dB coding gain at 10^{-5} BER. In comparison, the hybrid processing method accumulates the path metrics in the trellis through n-L to n-1 (forward) and through n+L to n+1 (backward) and at n selects the state which has the minimized path metric as a trace back state. In addition, there is no synchronization stage in the trace back process in the hybrid method. So the survivor path length is actually one half shorter than the forward processing method. Hence the coding gain of the former is smaller than that of the latter.

The disadvantage of the forward processing method is that it has to use 2402 flip-flops for the pipeline buffers compared with only 1188 flip-flops in the forwad and backward method [10].

Chapter 7

Conclusion

7.1 Summary

A sliding block Viterbi decoder was designed that combines the filtering characteristics of the sliding block decoder with the computation efficiency of the Viterbi algorithm. The finite memory length ($4L$) of the Viterbi algorithm allows decoding of the interval n-$M/2$ to $n+M/2$ based only on the input symbols over the interval n-$M/2$-L to $n+M/2+L$. Using the forward trellis processing method with the pipeline interleave structure unfolds the trellis iteration algorithm into concurrent calculations. Therefore the decoding speed is Mf_{clk} and can be further linearly increased with a linear increase in the hardware complexity.

The design was described in hierarchical VHDL, synthesized with the Synopsys tools and implemented using Xilinx Virtex. After the timing simulation with back annotation, the design was targeted onto an XCV300E-8 working at a frequency of 83.3MHz, hence providing 1Gbit/s of throughput.

7.2 Conclusion

Based on the design, simulation, implementation and evaluation for the sliding block Vitervi decoder, the following conclusions are drawn:

1. The sliding block method to implement the Viterbi decoder allows the use of hardware with a limited processing speed to achieve a very high throughput rate. It is a linear scale solution.

2. The performance of the forward processing with the sliding block method is better than that of hybrid processing, but the area of the former is larger than the latter.

3. To achieve the best synthesized result from the VHDL coding style, both the technology structure of the targeting device and your application logic should be taken into consideration.

7.3. Further Work

To make this design to work in real world, there are still some peripheral circuits that need to be designed such as high-speed parallel-to-serial/serial-to-parallel shift registers and a synchronization circuit.

Using similar components, other types of sliding block Viterbi decoder with longer constraint lengths can also be constructed, but the survivor path length, the path metric width and processing structure would have to be changed.

Appendix A: C/C++ Source Codes

// ****** This is the main program: encoder.cc   ******//

```
#include <stdlib.h>
#include <malloc.h>
#include <stream.h>
#include <math.h>
#include <stdio.h>
#include <time.h>
#include <iostream.h>
#include <fstream.h>

#include "vdsim.h"
#include "cnv_encd.cc"
#include "addnoise.cc"
#include "quantization.cc"

extern void cnv_encd(int g[2][4], long data_len, int *in_array, int
*out_array);
extern void addnoise(float es_ovr_n0, long data_len, int
*in_array,float *out_array);
extern void quantization(int g[2][3], float es_ovr_n0, long
channel_length, float *channel_output_vector, int
*decoder_output_matrix);
main(int argc,char *argv[])
{
    FILE *fileptr;
    long t,msg_length=MSG_LEN,channel_length, ltime;
    int *onezer, *encoded, *quantizationout;
    char *charptr;
    int  m, stime, FR=2, SN=1;
    float *splusn;
    float es_ovr_n0;
    int g[2][K] = {{1, 1, 1},{1, 0, 1}};
    m = K - 1;
    es_ovr_n0 = float (atof(argv[1]));
    msg_length=msg_length+(2*width-(2+msg_length)%(2*width));
    channel_length = ( msg_length + m ) * 2;

    onezer = (int *)malloc( msg_length * sizeof( int ) );
    encoded = (int *)malloc( channel_length * sizeof(int) );
    splusn = (float *)malloc( channel_length * sizeof(float) );
    quantizationout = (int *)malloc( msg_length * sizeof( int ) );

   //****************** generate random data ******************//

  ltime = time(NULL);
    stime = (unsigned int) ltime/2;
    srand(stime);

    /* generate the random data and write it to the output array    */
    for (t = 0; t < msg_length; t++)
       *( onezer + t ) = (int)( rand() / (RAND_MAX / 2) > 0.5 );
  /*************** Write the random dada to "data1.dat" *********/

   charptr =(char *)malloc( channel_length * sizeof( char ) );
   for (t=0;t<msg_length; t++)
    *(charptr+t)=0x30+*(onezer+t);
   fileptr = fopen("data1.dat","wb" );
   fwrite ( charptr, sizeof(char), msg_length, fileptr );
   fclose(fileptr);
   free(charptr);

   /**************** Convolutional Encoding ****************/

    cnv_encd(g, msg_length, onezer, encoded);

   /***************** Add Noise *******************/

   addnoise(es_ovr_n0, channel_length, encoded, splusn);

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

quantization(g, es_ovr_n0, channel_length, splusn,   quantizationout);

   free(onezer);
   free(encoded);
   free(splusn);
   free(quantizationout);

 }
```

```
//********************* cnv_encd.cc  ********************//

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "vdsim.h"

void cnv_encd(int g[2][K], long input_len, int *in_array, int
*out_array) {

    int m;                      /* K - 1 */
    long t, tt;                 /* bit time, symbol time */
    int j, k;                   /* loop variables */
    int *unencoded_data;        /* pointer to data array */
    int shift_reg[K];           /* the encoder shift register */
    int sr_head;                /* index to the first elt in the sr */
    int p, q;               /* the upper and lower xor gate outputs */

    m = K - 1;

    /* allocate space for the zero-padded input data array */
    unencoded_data = (int *)malloc((input_len + m)*sizeof(int));
    if (unencoded_data == NULL) {

        printf("\ncnv_encd.c: Can't allocate enough memory for
unencoded data!  Aborting...");

        exit(1);

    }


    /* read in the data and store it in the array */
    for (t = 0; t < input_len; t++)
        *(unencoded_data + t) = *(in_array + t);

    /* zero-pad the end of the data */
    for (t = 0; t < m; t++) {
        *(unencoded_data + input_len + t) = 0;
    }

    /* Initialize the shift register */
    for (j = 0; j < K; j++) {
        shift_reg[j] = 0;
    }

    sr_head = 0;

    /* initialize the channel symbol output index */
    tt = 0;

    /* Now start the encoding process */
 /*compute upper and lower mod-two adder outputs,one bit at a time */

    for (t = 0; t < input_len + m; t++) {
        shift_reg[sr_head] = *( unencoded_data + t );
        p = 0;
        q = 0;
        for (j = 0; j < K; j++) {
            k = (j + sr_head) % K;
            p ^= shift_reg[k] & g[0][j];
            q ^= shift_reg[k] & g[1][j];
        }

    /* write the upper and lower xor gate outputs as channel symbols */
        *(out_array + tt) = p;
        tt = tt + 1;
        *(out_array + tt) = q;
        tt = tt + 1;
        sr_head -= 1;     /* equivalent to shifting everything right one
place */
        if (sr_head < 0) /* but make sure we adjust pointer modulo K */
            sr_head = m;
    }

    /* free the dynamically allocated array */
    free(unencoded_data);

}
```

```
//********************* addnoise.cc  *********************//

#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "vdsim.h"


float gngauss(float mean, float sigma);


void addnoise(float es_ovr_n0, long channel_len, int *in_array, float
*out_array)
{
    long t;

    float mean, es, sn_ratio, sigma, signal;

    mean = 0;

    es = 1;

    sn_ratio = (float) pow(10, ( es_ovr_n0 / 10) );

    sigma =  (float) sqrt (es / ( 2 * sn_ratio ) );

    /* transform the data from 0/1 to +1/-1 and add noise */

    for (t = 0; t < channel_len; t++) {

        /*if the binary data value is 1, the channel symbol is -1; if
the binary data value is 0, the channel symbol is +1. */
        signal = 1 - 2 * *( in_array + t );

        /*  now generate the gaussian noise point, add it to the
channel symbol,
            and output the noisy channel symbol */

        *( out_array + t ) = signal + gngauss(mean, sigma);
    };
}

float gngauss(float mean, float sigma) {
double u, r;          /* uniform and Rayleigh random variables */

    /* generate a uniformly distributed random number u between 0 and 1
- 1E-6*/
    u = (double)rand() / RAND_MAX;
    if (u == 1.0) u = 0.999999999;

    /* generate a Rayleigh-distributed random number r using u */
    r = sigma * sqrt( 2.0 * log( 1.0 / (1.0 - u) ) );

    /* generate another uniformly-distributed random number u as
before*/
```

```
    u = (double)rand() / RAND_MAX;
    if (u == 1.0) u = 0.999999999;

    /* generate and return a Gaussian-distributed random number using r
and u */
    return( (float) ( mean + r * cos(2 * PI * u) ) );
}
```

```
//******************** quantiztion.cc  *****************//

#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
#include "vdsim.h"

#undef SLOWACS
#define FASTACS
#undef NORM
#define MAXMETRIC 128

void deci2bin(int d, int size, int *b);
int bin2deci(int *b, int size);
int nxt_stat(int current_state, int input, int *memory_contents);

void init_adaptive_quant(float es_ovr_n0);
char soft_quant(float channel_symbol);
int soft_metric(int data, int guess);

char quantizer_table[256];   /*  lj */

void quantization(int g[2][K], float es_ovr_n0, long int
channel_length,
          float *channel_output_vector, int *decoder_output_matrix)
 {
    FILE *fileptr;
    int i, j, l;
    long int t;
    int memory_contents[K];
    int input[TWOTOTHEM][TWOTOTHEM];
    int output[TWOTOTHEM][2];
    int nextstate[TWOTOTHEM][2];
    int accum_err_metric[TWOTOTHEM][2];
    int state_history[TWOTOTHEM][K * 5 + 1];
    int state_sequence[K * 5 + 1];
    int *channel_output_matrix;
    char *chptr;
    char *str = "1 000000 000000 000000 000000 000000 000000 000000
000000 000000 000000 000000 000000\n";
    int binary_output[2];
    int branch_output[2];
    int m, n, number_of_states, depth_of_trellis, step;
    int branch_metric, qunt_length,
        sh_ptr, sh_col, x, xx, h, hh, next_state,count,tmp;
    /* n is 2^1 = 2 for rate 1/2 */
    n = 2;

    /* m (memory length) = K - 1 */
    m = K - 1;

    /* number of states = 2^(K - 1) = 2^m for k = 1 */
    number_of_states = (int) pow(2, m);
    depth_of_trellis = 3 * 5;
    /* initialize data structures */
    for (i = 0; i < number_of_states; i++) {
        for (j = 0; j < number_of_states; j++)
            input[i][j] = 0;

        for (j = 0; j < n; j++) {
            nextstate[i][j] = 0;
            output[i][j] = 0;
        }

        for (j = 0; j <= depth_of_trellis; j++) {
            state_history[i][j] = 0;
        }

        /* initial accum_error_metric[x][0] = zero */
        accum_err_metric[i][0] = 0;
        /* by setting accum_error_metric[x][1] to MAXINT, we don't need
a flag */
        accum_err_metric[i][1] = MAXINT;

    }

    /* generate the state transition matrix, output matrix, and input
matrix
       - input matrix shows how FEC encoder bits lead to next state
       - next_state matrix shows next state given current state and
input bit
       - output matrix shows FEC encoder output bits given current
presumed
       encoder state and encoder input bit--this will be compared to
actual
       received symbols to determine metric for corresponding branch of
trellis
    */

    for (j = 0; j < number_of_states; j++) {
        for (l = 0; l < n; l++) {
            next_state = nxt_stat(j, l, memory_contents);
            input[j][next_state] = l;

            /* now compute the convolutional encoder output given the
current
               state number and the input value */
            branch_output[0] = 0;
            branch_output[1] = 0;

            for (i = 0; i < K; i++) {
                branch_output[0] ^= memory_contents[i] & g[0][i];
                branch_output[1] ^= memory_contents[i] & g[1][i];
            }

            /* next state, given current state and input */
            nextstate[j][l] = next_state;
            /* output in decimal, given current state and input */
            output[j][l] = bin2deci(branch_output, 2);
            } /* end of l for loop */

        } /* end of j for loop */


        channel_output_matrix = (int *)malloc( channel_length * sizeof(int)
);
        if (channel_output_matrix == NULL) {
            printf(
            "\nquantization.c: Can't allocate memory for
channel_output_matrix! Aborting...");
            exit(1);
        }

        /* now we're going to rearrange the channel output so it has n
rows,
           and n/2 columns where each row corresponds to a channel symbol
for
           a given bit and each column corresponds to an encoded bit */
        qunt_length=(channel_length/n)*6;
        channel_length = channel_length / n;

         chptr = (char *)malloc( qunt_length * sizeof( char ) );
        if (chptr == NULL) {
            printf("\n testquantization.c:  error allocating onezer array,
aborting!");
            exit(1);
        }

        init_adaptive_quant(es_ovr_n0);

        printf("\n");
        /* quantize the channel output--convert float to short integer */
        /* channel_output_matrix = reshape(channel_output, n,
channel_length) */
         fileptr = fopen("quantzd.dat","wb" );
         fwrite ((str), sizeof(char),2+7*width, fileptr );
         count=1;

        for (t = 0; t < (channel_length * n); t += n) {
            for (i = 0; i < n; i++)
             {   tmp =(soft_quant( *(channel_output_vector + (t + i) ) ));
                 *(channel_output_matrix+(t/n)+(i*channel_length)) = tmp;
                  if (count==1)
                      {
                       *(chptr+t+i)='0';
                       *(chptr+t+i+1)=' ';
                       if ( tmp & 0x04) *(chptr+t+i+2)='1';
                       else *(chptr+t+i+2)='0';
                       if ( tmp & 0x02) *(chptr+t+i+3)='1';
                       else *(chptr+t+i+3)='0';
                       if ( tmp & 0x01) *(chptr+t+i+4)='1';
                       else *(chptr+t+i+4)='0';
                       count += 1;

                       fwrite ((chptr+t+i), sizeof(char),5, fileptr );
```

```
                        }
                       else if (count<2*width)
                        { if (count % 2 != 0) {
                            *(chptr+t+i)=' ';
                            if ( tmp & 0x04) *(chptr+t+i+1)='1';
                            else *(chptr+t+i+1)='0';
                            if ( tmp & 0x02) *(chptr+t+i+2)='1';
                            else *(chptr+t+i+2)='0';
                            if ( tmp & 0x01) *(chptr+t+i+3)='1';
                            else *(chptr+t+i+3)='0';
                            count += 1;
                            fwrite ((chptr+t+i), sizeof(char),4, fileptr );

                            }
                            else {
                            if ( tmp & 0x04) *(chptr+t+i)='1';
                            else *(chptr+t+i)='0';
                            if ( tmp & 0x02) *(chptr+t+i+1)='1';
                            else *(chptr+t+i+1)='0';
                            if ( tmp & 0x01) *(chptr+t+i+2)='1';
                            else *(chptr+t+i+2)='0';
                            count += 1;
                            fwrite ((chptr+t+i), sizeof(char),3, fileptr );

                            }
                        }
                        else
                        {

                         if ( tmp & 0x04) *(chptr+t+i)='1';
                          else *(chptr+t+i)='0';
                         if ( tmp & 0x02) *(chptr+t+i+1)='1';
                          else *(chptr+t+i+1)='0';
                         if ( tmp & 0x01) *(chptr+t+i+2)='1';
                          else *(chptr+t+i+2)='0';
                         *(chptr+t+i+3)=0x0d;
                         *(chptr+t+i+4)=0x0a;
                         fwrite ((chptr+t+i), sizeof(char),5, fileptr );
                         count=1;
                        };

};
    } /* end t for-loop */

        /* write data to file */
    fclose(fileptr);
    free(chptr);

}
/* *************************************************************** */

/* this initializes a quantizer that adapts to Es/No */
void init_adaptive_quant(float es_ovr_n0) {

    int i, d;
    float es, sn_ratio, sigma;
```

```
     es = 1;
     sn_ratio = (float) pow(10.0, ( es_ovr_n0 / 10.0 ) );
     sigma =  (float) sqrt( es / ( 2.0 * sn_ratio ) );

     d = (int) ( 32 * 0.5 * sigma );

     for (i = -128; i < ( -3 * d ); i++)
         quantizer_table[i + 128] = 7;

     for (i = ( -3 * d ); i < ( -2 * d ); i++)
         quantizer_table[i + 128] = 6;

     for (i = ( -2 * d ); i < ( -1 * d ); i++)
         quantizer_table[i + 128] = 5;

     for (i = ( -1 * d ); i < 0; i++)
         quantizer_table[i + 128] = 4;

     for (i = 0; i < ( 1 * d ); i++)
         quantizer_table[i + 128] = 3;

     for (i = ( 1 * d ); i < ( 2 * d ); i++)
         quantizer_table[i + 128] = 2;

     for (i = ( 2 * d ); i < ( 3 * d ); i++)
         quantizer_table[i + 128] = 1;

     for (i = ( 3 * d ); i < 128; i++)
         quantizer_table[i + 128] = 0;
}


/* this quantizer assumes that the mean channel_symbol value is +/- 1,
   and translates it to an integer whose mean value is +/- 32 to
address
   the lookup table "quantizer_table". Overflow protection is included.
*/
 char soft_quant(float channel_symbol)
{
    int x;

    x = (int) ( 32.0 * channel_symbol );
    if (x < -128) x = -128;
    if (x > 127) x = 127;

    return(quantizer_table[x + 128]);
}


/* this metric is based on the algorithm given in Michelson and
Levesque,
   page 323. */
int soft_metric(int data, int guess) {

    return(abs(data - (guess * 7)));
}
```

```
/* this function calculates the next state of the convolutional
encoder, given
   the current state and the input data.  It also calculates the memory
   contents of the convolutional encoder. */
int nxt_stat(int current_state, int input, int *memory_contents) {

    int binary_state[K - 1];               /* binary value of current
state */
    int next_state_binary[K - 1];          /* binary value of next state
*/
    int next_state;                        /* decimal value of next
state */
    int i;                                 /* loop variable */

    /* convert the decimal value of the current state number to binary
*/
    deci2bin(current_state, K - 1, binary_state);

    /* given the input and current state number, compute the next state
number */
    next_state_binary[0] = input;
    for (i = 1; i < K - 1; i++)
        next_state_binary[i] = binary_state[i - 1];

    /* convert the binary value of the next state number to decimal */
    next_state = bin2deci(next_state_binary, K - 1);

    /* memory_contents are the inputs to the modulo-two adders in the
encoder */
    memory_contents[0] = input;
    for (i = 1; i < K; i++)
        memory_contents[i] = binary_state[i - 1];

    return(next_state);
}


/* this function converts a decimal number to a binary number, stored
   as a vector MSB first, having a specified number of bits with
   leading zeroes as necessary */

void deci2bin(int d, int size, int *b) {
    int i;

    for(i = 0; i < size; i++)
        b[i] = 0;

    b[size - 1] = d & 0x01;

    for (i = size - 2; i >= 0; i--) {
        d = d >> 1;
        b[i] = d & 0x01;
    }
}
```

```
/* this function converts a binary number having a specified
   number of bits to the corresponding decimal number */
int bin2deci(int *b, int size) {
    int i, d;

    d = 0;

    for (i = 0; i < size; i++)
        d = d + (int)pow(2, size - i - 1) * b[i];

    return(d);
}
```

```
//****************** vdsim.h *************************//

#define K 3               /* constraint length */
#define TWOTOTHEM 4        /* 2^(K - 1) -- change as required */
#define PI 3.141592654     /* circumference of circle divided by diameter
*/
#define MSG_LEN 100000  /* how many bits in each test message */
#define DOENC 1            /* test with convolutional encoding/Viterbi
decoding */
#undef  DONOENC            /* test with no coding */
#define LOESNO 0.0         /* minimum Es/No at which to test */
#define HIESNO 3.5         /* maximum Es/No at which to test */
#define ESNOSTEP 0.5       /* Es/No increment for test driver */
#define width 12           /* Decoder's width */
```

```
//******  This program execute step 3 in Figure-4.2 *********//

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#define decode_length 12
#define max_err_No 100

main()
{ char msg,dcd;
  long number_error=0,total_error,msg_length=0,total_length;
  float es_ovr_n,BER;
  int errposition ;
  long j,cnt[decode_length+1];

  ifstream in_msg ;
  ifstream in_dcd ;
  fstream io_tmp;
  fstream io_tpos;
  ofstream out_result;
  ofstream out_pos;
//************ open data.dat *******************//

  in_msg.open ("data.dat",ios::in | ios::nocreate );
  if (!in_msg) {
    cout<< "Cannot open data1.dat \n";
    return 1;
  }

// ****************open decoded.dat ***********//

  in_dcd.open ("decoded.dat",ios::in | ios::nocreate );
  if (!in_dcd) {
    cout<< "Cannot open decoded.dat \n";
    return 1;
  }
// *************** open tpos.dat **************//
   io_tpos.open ("tpos.dat",ios::in|ios::out);
   if ( !io_tpos) {
        cout<<" Cannot open tpos.dat \n";
        return 1;
        }
   io_tpos.setf(ios::showpoint);
   for (j = 0; j < decode_length; j++) io_tpos>>cnt[j];

// ******** compare the simulation result ***********//
  in_dcd.seekg(42*(decode_length+1)+6,ios::beg);

  while(in_msg.get(msg))
   { msg_length++;
     in_dcd>>dcd;

     if (msg!=dcd)
     { number_error++;
```

```
          errposition=(int(msg_length)+decode_length/2)%decode_length;
          if (errposition==0) errposition = decode_length;
          cout<<msg_length<<"-"<<errposition<< " ";
          cnt[errposition-1]++;
        }
      }
      io_tpos.seekp(0,ios::beg);
      for (j = 0; j < decode_length; j++)
        io_tpos<<cnt[j]<<" ";

      io_tpos<<"\n";
      io_tpos.close();
   // *************** open tmp.dat **************//

     io_tmp.open ("tmp.dat",ios::in|ios::out|ios::nocreate);
     if ( !io_tmp) {
       cout<<" Cannot open tmp.dat \n";
       return 1;
     }
     io_tmp.setf(ios::showpoint);

   // **** update total_lengh, total_error and es_ovr_n ****//

      io_tmp>>total_length>>total_error>>es_ovr_n;

      total_error = total_error + number_error;
      total_length = total_length + msg_length;

      if (total_error<200) {
       cout<<total_length<<" "<<total_error<<" "<<es_ovr_n;
      // cout<<es_ovr_n;
       io_tmp.seekp(ios::beg);
       io_tmp<<total_length<<" "<< total_error<<" "<< es_ovr_n<<" ";
       io_tmp.close();
       }
      else {
       io_tmp.seekp(ios::beg);
       io_tmp << 0 << " " << 0 << " " << es_ovr_n + 0.5<<" ";
       io_tmp.close();

       BER = float(total_error)/float(total_length);

  // *************** update result.dat **************//

       out_result.open ("result.dat", ios::out | ios::app );
       if ( !out_result) {
       cout<<" Cannot open result.dat \n";
       return 1;
       }

       printf(" BER = %E \n ",BER);
       cout<< "The number of error bits is " << total_error
         << " in " << total_length << " bits.\n";
       // cout<<es_ovr_n+0.5;
       out_result.setf(ios:: scientific);
       out_result<< BER << ",";
       out_result.setf(ios::fixed);
     out_result << es_ovr_n << "\n";
     out_result.close();
 // *************** update pos.dat **************//
    out_pos.open ("pos.dat", ios::out | ios::app );
     if ( !out_result) {
     cout<<" Cannot open pos.dat \n";
     return 1;
     }

     for (j = 0; j < decode_length; j++)
      out_pos<<cnt[j]<<" ";
     out_pos<<es_ovr_n<<"\n";
     out_pos.close();

     for (j = 0; j < decode_length; j++) cnt[j]=0;
     // *************** open tpos.dat **************//
    io_tpos.open ("tpos.dat",ios::in|ios::out);
    if ( !io_tpos) {
        cout<<" Cannot open tpos.dat \n";
        return 1;
      }
    io_tpos.setf(ios::showpoint);
    for (j = 0; j < decode_length; j++) io_tpos<<cnt[j]<<" ";
    io_tpos.close();

   }

  in_msg.close();
  in_dcd.close();

  return 0;
}
```

//** This Perl program coordinate encoder, vhdlsim and comp **//
// ** It makes the flow in Figure-4.2 repeat work **//

```
#!/usr/local/bin/perl

$LOGFILE = "tmp.dat";
open(LOGFILE) or die("Could not open log file.");
read(LOGFILE, $line,30);
close(LOGFILE);
($msg, $err, $esovrn) =split(' ',$line);
$esovrn = substr($esovrn,0,3);
while ($esovrn<=8.5) {
 print("$esovrn");

 system("nice -19 encoder $esovrn");
 system("nice -19 vhdlsim -nc conf_testbench -e my");
 system("comp");

 $LOGFILE = "tmp.dat";
 open(LOGFILE) or die("Could not open log file.");
 read(LOGFILE, $line,30);
 close(LOGFILE);
 ($msg, $err, $esovrn) = split(' ',$line);
 $esovrn = substr($esovrn,0,3);
};
print("The test is over! \n");
```

Appendix B. Data.dat file Format

```
01111101101111001101100000110001011110100100101000110101111011010100100
01110110110010011100100001110000011010101001000010111111100010100010011
10101001110010111100111010110110011010001110000001101110100010101110001
00010011011101111110000011011100001000001101001010001010000000010100010
10101000000101110011010000000110001010000101101111110101101000111110001
11101011110011010001110001101100001011010010101001000011010110111011001
01110010111000001111101111100101010111101001001100110101001101011011010
00010110000100001010000000011110101001001011111111110100010000011001100
11000100010100001101011110111011100100010101000011001011011001101110100
00001010010111111111011001111100001010100001110010000100110101100100010
00101001010010000100001101110110010011111011101110001010101101101101 00
11001011001010110101111111010111001111100101000101001111110101100000100
00011100101100101001101101001011010101101001100101100000000010111100100
11001010110010000000101010100110001111100100111001110100101100001111111
101110010011
......
```

Appendix C quntzd.dat file format

```
1 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000 000000
0 000000 111111 000111 110010 011010 111000 000111 000000 000110 000111 000010 000110
0 111000 111001 010110 111111 111111 000111 000111 000000 000111 000111 111110 000000
0 000000 000001 111111 000111 000111 101100 000000 111111 110000 011000 010101 110001
0 101000 000111 001000 111001 111111 111111 111000 111111 111111 110000 000000 100000
0 110111 000000 111111 000111 010110 000000 100000 000000 000110 111001 111000 000111
0 000000 000111 010110 000000 111000 000000 111000 111111 100111 111000 111110 000000
0 101110 000111 111000 000111 000000 000111 000111 000000 000110 000111 011111 111111
0 101000 111111 111110 000111 101000 000111 101111 101111 111000 111111 000000 000000
0 101111 000110 111000 000111 011111 000010 000000 001000 111110 000111 000111 000001
0 111000 000010 111101 001000 110010 101111 111111 111001 111111 000000 000000 111101
0 111000 000000 000111 110010 111001 111000 111001 111000 000101 111111 000001 110111
0 100010 000000 110000 111111 000000 111111 111001 111111 111111 001111 111000 000111
0 000000 110000 000000 111000 111111 111111 000111 111000 000111 111111 111011 111001
0 010000 000101 110000 111000 000111 111111 101110 000100 111100 000111 000000 111000
0 001000 000111 000111 000001 000110 010111 101110 111111 000111 000111 000000 111000
0 111111 000000 111101 000111 101000 000110 111111 000001 000000 000000 000000 111111
0 010111 000111 000000 000111 111000 000111 000000 111001 111111 000000 111111 111000
0 001000 011000 001000 000111 111001 000111 111101 000000 110111 110000 111011 000010
0 110111 101000 111111 011110 000111 000111 000001 000110 111000 000111 010000 000111
0 111000 111000 111001 111000 000101 110111 000001 000000 010000 111111 000111 000111
0 000000 000110 111000 000111 100111 000000 011000 110111 110000 111100 000000 000000
0 000000 110110 011111 000111 000000 111000 111111 111111 111000 000011 100010 111111
0 000000 111111 111000 000000 111000 100111 000000 000010 000000 000001 010000 000010
0 111111 111001 000000 111000 111111 000001 111111 111000 000001 110000 000000 111001
0 001011 100001 111110 000000 000011 000000 000000 110101 111001 000010 000110 111000
0 100111 111111 110111 000111 000111 000000 111000 111100 000000 011000 000000 010000
0 000000 111111 100111 001111 111111 000000 111111 111000 000000 111000 010111 000000
0 000000 111110 111000 000000 000111 000111 000000 001111 111010 111000 110000 111000
0 010111 000000 111000 001000 000111 001110 000000 111010 111111 001000 101111 000101
0 110000 111000 111000 000111 111111 000000 111011 000111 111010 111000 000110 011000
0 110000 000000 000110 111000 111000 000111 111111 111101 000110 000111 000000 111000
0 011111 000001 111111 000111 011000 000111 111111 000000 111111 000111 001111 000000
0 000111 000111 110111 001100 000001 111111 111001 000000 000101 000110 000000 111000
0 111111 111111 111000 001000 100000 010001 111100 111111 111111 111000 111111 010000
................
```

Note:

First bit in each row is the reset signal.

Two 3-bit quantized symbol compose of a channel symbol pair.

Each row represents a half block.

Appendix D. decoded.dat File Format

```
000000000000 ← Reset period.
000000000000 ⎫
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎬  The latency period (41 clock cycles) after reset.
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎪
000000000000 ⎭
000000011111 ← The decoded data begins from the seventh bit of this row.
011011110011
011000001100
010111101001
001010001101
011110110101
001000111011
011001001110
010000111000
```

```
001101010100
100001011111
110001010001
001110101001
110010111100
111010110110
011010001110
000001101110
100010101110
001000100110
111011111100
000110111000
010000011010
010100010100
000000101000
101010100000
010111001101
000000011000
101000010110
111111010110
100011111000
111101011110
011010001110
001101100001
011010010101
001000011010
110111011001
011100101110
000011111011
111001010101
111010010011
001101010011
010110110100
001011000010
000101000000
001111010100
100101111111
111010001000
001100110011
000100010100
001101011110
111011100100
010101000011
001011011001
101110100000
010100101111
111110110011
111000010101
000011100100
001001101011
.........
```

Appendix E. VHDL Source Code

```
library IEEE;
library UNISIM;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use UNISIM.all;

entity top is
   port (
        x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11:  in std_logic_vector (5 downto 0);
        clock,reset:  in STD_LOGIC;
        y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11:  out STD_LOGIC);
end top;

architecture arch_top of top is

component BUFGDLL
   port (I : in STD_LOGIC;
         O : out STD_LOGIC);
end component;

component BUFG
  port (I : in std_logic;
        O : out std_logic);
end component;

component FDC
  port (Q: out std_logic;
        D,C,CLR: in std_logic);
end component;

component bufflx
        generic (depth:integer);
        port (din: in std_logic;
              dout: out std_logic;
              clk,reset:in std_logic);
end component;


component buffx1
      generic(width:integer);
      port (din:  in std_logic_vector(width-1 downto 0);
            dout: out std_logic_vector(width-1 downto 0);
            clk,reset: in std_logic);
```

```
end component;

component buffxx
        generic (width: integer; depth:integer);
        port (din: in std_logic_vector (width-1 downto 0);
                dout: out std_logic_vector (width-1 downto 0);
                clk,reset:in std_logic);
end component;

component  acs4
     port (sym: in std_logic_vector(5 downto 0);
             gm0_in,gm1_in,gm2_in,gm3_in: in std_logic_vector (6 downto 0);
             gm0_out,gm1_out,gm2_out,gm3_out: out std_logic_vector (6 downto 0);
             d: out STD_LOGIC_vector (3 downto 0);
             clk: in STD_LOGIC; reset:in std_logic);
end component;

component cs port (gm0,gm1,gm2,gm3: in std_logic_vector (6 downto 0);
                        selec: out std_logic_vector (1 downto 0);
                        clk: in STD_LOGIC; RESET: in std_logic);
end component;

component tb port (state_in: in std_logic_VECTOR(1 downto 0);
                        d: in STD_LOGIC_vector (3 downto 0);
                        state_out: out std_logic_vector(1 downto 0);
                        clk: in std_logic;
                         reset: in std_logic);
end component;

signal oscout,clk: STD_LOGIC;
signal g0_0,g0_1,g0_2,g0_3,
        g1_0,g1_1,g1_2,g1_3,
        g2_0,g2_1,g2_2,g2_3,
        g3_0,g3_1,g3_2,g3_3,
        g4_0,g4_1,g4_2,g4_3,
        g5_0,g5_1,g5_2,g5_3,
        g6_0,g6_1,g6_2,g6_3,
        g7_0,g7_1,g7_2,g7_3,
        g8_0,g8_1,g8_2,g8_3,
        g9_0,g9_1,g9_2,g9_3,
        g10_0,g10_1,g10_2,g10_3,
        g11_0,g11_1,g11_2,g11_3,
        g12_0,g12_1,g12_2,g12_3,
        g13_0,g13_1,g13_2,g13_3,
        g14_0,g14_1,g14_2,g14_3,
        g15_0,g15_1,g15_2,g15_3,
        g16_0,g16_1,g16_2,g16_3,
        g17_0,g17_1,g17_2,g17_3,
        g18_0,g18_1,g18_2,g18_3,
        g19_0,g19_1,g19_2,g19_3,
        g20_0,g20_1,g20_2,g20_3,
        g21_0,g21_1,g21_2,g21_3,
        g22_0,g22_1,g22_2,g22_3,
        g23_0,g23_1,g23_2,g23_3,zero:
        std_logic_vector (6 downto 0 );

signal b4_1i,b4_1o,b4_3i,b4_3o,b4_5i,b4_5o,b4_7i,b4_7o,b4_9i,
        b4_9o,b4_11i,b4_11o,b4_13i,b4_13o,b4_15i,b4_15o,b4_17i,
        b4_17o,b4_19i,b4_19o,b4_21i,b4_21o,b4_23i,b4_23o,b4_25i,
        b4_25o,b4_27i,b4_27o,b4_29i,b4_29o,b4_31i,b4_31o,b4_33i,
        b4_33o: std_logic_vector (3 downto 0);

signal s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,
        s17,s18:std_logic_vector (1 downto 0);

signal b1_1,b1_2,b1_3,b1_4,b1_5,b1_6,b1_7,b1_8,b1_9,
        b1_10,b1_11: std_logic;

signal x1_1,x2_2,x3_3,x4_4,x5_5,x6_6,x7_7,x8_8,x9_9,x10_10,
        x11_11,x0_11,x1_12,x2_13,x3_14,x4_15,x5_16,x6_17,x7_18,
        x8_19,x9_20,x10_21,x11_22: std_logic_vector (5 downto 0);

begin

DLL: BUFGDLL port map(I=>clock,O=>oscout);
CLOCKBUF:BUFG port map(I=>oscout,O=>clk);

pipex0_11:buffxx
       generic map (width=>6,depth=>11)
       port map  (din=>x0,dout=>x0_11,clk=>clk,reset=>reset);
----------------------------
pipex1_1:buffx1
       generic map (width=>6)
       port map  (din=>x1,dout=>x1_1,clk=>clk,reset=>reset);
pipex1_12:buffxx
       generic map (width=>6,depth=>11)
       port map  (din=>x1_1,dout=>x1_12,clk=>clk,reset=>reset);
----------------------------
pipex2_2:buffxx
       generic map (width=>6,depth=>2)
       port map  (din=>x2,dout=>x2_2,clk=>clk,reset=>reset);
pipex2_13:buffxx
```

```
        generic map (width=>6,depth=>11)
        port map  (din=>x2_2,dout=>x2_13,clk=>clk,reset=>reset);
----------------------------
pipex3_3:buffxx
        generic map (width=>6,depth=>3)
        port map  (din=>x3,dout=>x3_3,clk=>clk,reset=>reset);

pipex3_14:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x3_3,dout=>x3_14,clk=>clk,reset=>reset);
----------------------------
pipex4_4:buffxx
        generic map (width=>6,depth=>4)
        port map  (din=>x4,dout=>x4_4,clk=>clk,reset=>reset);
pipex4_15:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x4_4,dout=>x4_15,clk=>clk,reset=>reset);
----------------------------
pipex5_5:buffxx
        generic map (width=>6,depth=>5)
        port map  (din=>x5,dout=>x5_5,clk=>clk,reset=>reset);
pipex5_16:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x5_5,dout=>x5_16,clk=>clk,reset=>reset);
----------------------------
pipex6_6:buffxx
        generic map (width=>6,depth=>6)
        port map  (din=>x6,dout=>x6_6,clk=>clk,reset=>reset);
pipex6_17:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x6_6,dout=>x6_17,clk=>clk,reset=>reset);
----------------------------
pipex7_7:buffxx
        generic map (width=>6,depth=>7)
        port map  (din=>x7,dout=>x7_7,clk=>clk,reset=>reset);
pipex7_18:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x7_7,dout=>x7_18,clk=>clk,reset=>reset);
----------------------------
pipex8_8:buffxx
        generic map (width=>6,depth=>8)
        port map  (din=>x8,dout=>x8_8,clk=>clk,reset=>reset);
pipex8_19:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x8_8,dout=>x8_19,clk=>clk,reset=>reset);
----------------------------
```

```
pipex9_9:buffxx
        generic map (width=>6,depth=>9)
        port map  (din=>x9,dout=>x9_9,clk=>clk,reset=>reset);
pipex9_20:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x9_9,dout=>x9_20,clk=>clk,reset=>reset);
-------------------------
pipex10_10:buffxx
        generic map (width=>6,depth=>10)
        port map  (din=>x10,dout=>x10_10,clk=>clk,reset=>reset);
pipex10_21:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x10_10,dout=>x10_21,clk=>clk,reset=>reset);
-------------------------
pipex11_11:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x11,dout=>x11_11,clk=>clk,reset=>reset);
pipex11_22:buffxx
        generic map (width=>6,depth=>11)
        port map  (din=>x11_11,dout=>x11_22,clk=>clk,reset=>reset);
------------------------------------------------------
zero<="0000000";
stage_0: acs4
   port map (sym=>x0,
      gm0_in=>zero,gm1_in=>zero,gm2_in=>zero,gm3_in=>zero,
      gm0_out=>g0_0,gm1_out=>g0_1,gm2_out=>g0_2,gm3_out=>g0_3,
      clk=>clk,reset=>reset);
------------------------------------------------------
stage_1:acs4
   port map (sym=>x1_1,
      gm0_in=>g0_0, gm1_in=>g0_1, gm2_in=>g0_2, gm3_in=>g0_3,
      gm0_out=>g1_0,gm1_out=>g1_1,gm2_out=>g1_2,gm3_out=>g1_3,
      clk=>clk,reset=>reset);
------------------------------------------------------
stage_2: acs4
   port map (sym=>x2_2,
      gm0_in=>g1_0, gm1_in=>g1_1, gm2_in=>g1_2, gm3_in=>g1_3,
      gm0_out=>g2_0,gm1_out=>g2_1,gm2_out=>g2_2,gm3_out=>g2_3,
      clk=>clk,reset=>reset);
------------------------------------------------------
stage_3:acs4
   port map (sym=>x3_3,
      gm0_in=>g2_0,  gm1_in=>g2_1, gm2_in=>g2_2,  gm3_in=>g2_3,
      gm0_out=>g3_0,gm1_out=>g3_1, gm2_out=>g3_2, gm3_out=>g3_3,
      clk=>clk,reset=>reset);
------------------------------------------------------
```

```
stage_4:acs4
   port map (sym=>x4_4,
      gm0_in=>g3_0, gm1_in=>g3_1, gm2_in=>g3_2, gm3_in=>g3_3,
      gm0_out=>g4_0,gm1_out=>g4_1,gm2_out=>g4_2,gm3_out=>g4_3,
      clk=>clk,reset=>reset);
----------------------------------------------------------------
stage_5:acs4
   port map (sym=>x5_5,
      gm0_in=>g4_0, gm1_in=>g4_1, gm2_in=>g4_2, gm3_in=>g4_3,
      gm0_out=>g5_0,gm1_out=>g5_1,gm2_out=>g5_2,gm3_out=>g5_3,
      clk=>clk,reset=>reset);
----------------------------------------------------------------
stage_6:acs4
   port map (sym=>x6_6,
      gm0_in=>g5_0, gm1_in=>g5_1, gm2_in=>g5_2, gm3_in=>g5_3,
      gm0_out=>g6_0,gm1_out=>g6_1,gm2_out=>g6_2,gm3_out=>g6_3,
      clk=>clk,reset=>reset);
----------------------------------------------------------------
stage_7: acs4
   port map (sym=>x7_7,
      gm0_in=>g6_0, gm1_in=>g6_1, gm2_in=>g6_2, gm3_in=>g6_3,
      gm0_out=>g7_0,gm1_out=>g7_1,gm2_out=>g7_2,gm3_out=>g7_3,
      d=>b4_33i, clk=>clk,reset=>reset);
buff4_33: buffxx
   generic map (width=>4,depth=>33)
   port map  (din=>b4_33i,dout=>b4_33o,clk=>clk,reset=>reset);
TraceBack17: TB
  port map (state_in=>s17,d=>b4_33o,state_out=>s18,
         clk=>clk,reset=>reset);
y0<=s18(1);
----------------------------------------------------------------
stage_8:acs4
   port map (sym=>x8_8,
      gm0_in=>g7_0, gm1_in=>g7_1, gm2_in=>g7_2, gm3_in=>g7_3,
      gm0_out=>g8_0,gm1_out=>g8_1,gm2_out=>g8_2,gm3_out=>g8_3,
      d=>b4_31i,clk=>clk,reset=>reset);
buff4_31: buffxx
   generic map (width=>4,depth=>31)
   port map  (din=>b4_31i,dout=>b4_31o,clk=>clk,reset=>reset);
TraceBack16: TB
  port map (state_in=>s16,d=>b4_31o,state_out=>s17,
         clk=>clk,reset=>reset);
buff1_1:FDC port map (Q=>y1,D=>s17(1),C=>clk,CLR=>reset);
----------------------------------------------------------------
stage_9:acs4
   port map (sym=>x9_9,
        gm0_in=>g8_0, gm1_in=>g8_1, gm2_in=>g8_2, gm3_in=>g8_3,
        gm0_out=>g9_0,gm1_out=>g9_1,gm2_out=>g9_2,gm3_out=>g9_3,
        d=>b4_29i,clk=>clk,reset=>reset);
 buff4_29: buffxx
    generic map (width=>4,depth=>29)
    port map  (din=>b4_29i,dout=>b4_29o,clk=>clk,reset=>reset);
 TraceBack15: TB
   port map (state_in=>s15,d=>b4_29o,state_out=>s16,
         clk=>clk,reset=>reset);
 buffl_2:bufflx
    generic map (depth=>2)
    port map  (din=>s16(1),dout=>y2,clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_10:acs4
    port map (sym=>x10_10,
       gm0_in=>g9_0, gm1_in=>g9_1, gm2_in=>g9_2, gm3_in=>g9_3,
       gm0_out=>g10_0,gm1_out=>g10_1,gm2_out=>g10_2,gm3_out=>g10_3,
       d=>b4_27i,clk=>clk,reset=>reset);
 buff4_27: buffxx
    generic map (width=>4,depth=>27)
    port map  (din=>b4_27i,dout=>b4_27o,clk=>clk,reset=>reset);
 TraceBack14: TB
   port map (state_in=>s14,d=>b4_27o,state_out=>s15,
         clk=>clk,reset=>reset);
 buffl_3:bufflx
    generic map (depth=>3)
    port map  (din=>s15(1),dout=>y3,clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_11:acs4
    port map (sym=>x11_11,
       gm0_in=>g10_0, gm1_in=>g10_1, gm2_in=>g10_2, gm3_in=>g10_3,
       gm0_out=>g11_0,gm1_out=>g11_1,gm2_out=>g11_2,gm3_out=>g11_3,
       d=>b4_25i,clk=>clk,reset=>reset);
 buff4_25: buffxx
 generic map (width=>4,depth=>25)
    port map  (din=>b4_25i,dout=>b4_25o,clk=>clk,reset=>reset);
 TraceBack13: TB
   port map (state_in=>s13,d=>b4_25o,state_out=>s14,
         clk=>clk,reset=>reset);
 buffl_4:bufflx
    generic map (depth=>4)
    port map  (din=>s14(1),dout=>y4,clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_12:acs4
    port map (sym=>x0_11,
       gm0_in=>g11_0, gm1_in=>g11_1, gm2_in=>g11_2, gm3_in=>g11_3,
        gm0_out=>g12_0,gm1_out=>g12_1,gm2_out=>g12_2,gm3_out=>g12_3,
        d=>b4_23i,clk=>clk,reset=>reset);

 buff4_23: buffxx
 generic map (width=>4,depth=>23)
    port map  (din=>b4_23i,dout=>b4_23o,clk=>clk,reset=>reset);
 TraceBack12: TB
   port map (state_in=>s12,d=>b4_23o,state_out=>s13,
           clk=>clk,reset=>reset);
 buff1_5:buff1x
    generic map (depth=>5)
    port map  (din=>s13(1),dout=>y5,clk=>clk,reset=>reset);
 --------------------------------------------------------
 stage_13:acs4
    port map (sym=>x1_12,
        gm0_in=>g12_0,  gm1_in=>g12_1,  gm2_in=>g12_2,  gm3_in=>g12_3,
        gm0_out=>g13_0,gm1_out=>g13_1,gm2_out=>g13_2,gm3_out=>g13_3,
        d=>b4_21i,clk=>clk,reset=>reset);
 buff4_21: buffxx
 generic map (width=>4,depth=>21)
    port map  (din=>b4_21i,dout=>b4_21o,clk=>clk,reset=>reset);
 TraceBack11: TB
   port map (state_in=>s11,d=>b4_21o,state_out=>s12,
           clk=>clk,reset=>reset);
 buff1_6:buff1x
    generic map (depth=>6)
    port map  (din=>s12(1),dout=>y6,clk=>clk,reset=>reset);
 --------------------------------------------------------
 stage_14:acs4
    port map (sym=>x2_13,
        gm0_in=>g13_0,  gm1_in=>g13_1,  gm2_in=>g13_2,  gm3_in=>g13_3,
        gm0_out=>g14_0,gm1_out=>g14_1,gm2_out=>g14_2,gm3_out=>g14_3,
        d=>b4_19i,clk=>clk,reset=>reset);
 buff4_19: buffxx
 generic map (width=>4,depth=>19)
    port map  (din=>b4_19i,dout=>b4_19o,clk=>clk,reset=>reset);
 TraceBack10: TB
   port map (state_in=>s10,d=>b4_19o,state_out=>s11,
           clk=>clk,reset=>reset);
 buff1_7:buff1x
    generic map (depth=>7)
    port map  (din=>s11(1),dout=>y7,clk=>clk,reset=>reset);
 --------------------------------------------------------
 stage_15:acs4
    port map (sym=>x3_14,
        gm0_in=>g14_0,  gm1_in=>g14_1,  gm2_in=>g14_2,  gm3_in=>g14_3,
        gm0_out=>g15_0,gm1_out=>g15_1,gm2_out=>g15_2,gm3_out=>g15_3,
        d=>b4_17i,clk=>clk,reset=>reset);

 buff4_17: buffxx
 generic map (width=>4,depth=>17)
    port map  (din=>b4_17i,dout=>b4_17o,clk=>clk,reset=>reset);
 TraceBack9: TB
   port map (state_in=>s9,d=>b4_17o,state_out=>s10,
           clk=>clk,reset=>reset);
 buff1_8:buff1x
    generic map (depth=>8)
    port map  (din=>s10(1),dout=>y8,clk=>clk,reset=>reset);
-------------------------------------------------------
 stage_16:acs4
    port map (sym=>x4_15,
       gm0_in=>g15_0,  gm1_in=>g15_1,  gm2_in=>g15_2,  gm3_in=>g15_3,
       gm0_out=>g16_0,gm1_out=>g16_1,gm2_out=>g16_2,gm3_out=>g16_3,
       d=>b4_15i,clk=>clk,reset=>reset);
 buff4_15: buffxx
 generic map (width=>4,depth=>15)
    port map  (din=>b4_15i,dout=>b4_15o,clk=>clk,reset=>reset);
 TraceBack8: TB
   port map (state_in=>s8,d=>b4_15o,state_out=>s9,
           clk=>clk,reset=>reset);
 buff1_9:buff1x
    generic map (depth=>9)
    port map  (din=>s9(1),dout=>y9,clk=>clk,reset=>reset);
-------------------------------------------------------
 stage_17:acs4
    port map (sym=>x5_16,
       gm0_in=>g16_0,  gm1_in=>g16_1,  gm2_in=>g16_2,  gm3_in=>g16_3,
       gm0_out=>g17_0,gm1_out=>g17_1,gm2_out=>g17_2,gm3_out=>g17_3,
       d=>b4_13i,clk=>clk,reset=>reset);
 buff4_13: buffxx
 generic map (width=>4,depth=>13)
    port map  (din=>b4_13i,dout=>b4_13o,clk=>clk,reset=>reset);
 TraceBack7: TB
   port map (state_in=>s7,d=>b4_13o,state_out=>s8,
           clk=>clk,reset=>reset);
 buff1_10:buff1x
    generic map (depth=>10)
    port map  (din=>s8(1),dout=>y10,clk=>clk,reset=>reset);
-------------------------------------------------------
 stage_18:acs4
    port map (sym=>x6_17,
       gm0_in=>g17_0,  gm1_in=>g17_1,  gm2_in=>g17_2,  gm3_in=>g17_3,
        gm0_out=>g18_0,gm1_out=>g18_1,gm2_out=>g18_2,gm3_out=>g18_3,
        d=>b4_11i,clk=>clk,reset=>reset);

 buff4_11: buffxx
 generic map (width=>4,depth=>11)
    port map  (din=>b4_11i,dout=>b4_11o,clk=>clk,reset=>reset);
 TraceBack6: TB
   port map (state_in=>s6,d=>b4_11o,state_out=>s7,
          clk=>clk,reset=>reset);
 buff1_11:buff1x
    generic map (depth=>11)
    port map  (din=>s7(1),dout=>y11,clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_19:acs4
    port map (sym=>x7_18,
       gm0_in=>g18_0, gm1_in=>g18_1, gm2_in=>g18_2, gm3_in=>g18_3,
       gm0_out=>g19_0,gm1_out=>g19_1,gm2_out=>g19_2,gm3_out=>g19_3,
       d=>b4_9i,clk=>clk,reset=>reset);
 buff4_9: buffxx
 generic map (width=>4,depth=>9)
    port map  (din=>b4_9i,dout=>b4_9o,clk=>clk,reset=>reset);
 TraceBack5: TB
   port map (state_in=>s5,d=>b4_9o,state_out=>s6,
          clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_20:acs4
    port map (sym=>x8_19,
       gm0_in=>g19_0, gm1_in=>g19_1, gm2_in=>g19_2, gm3_in=>g19_3,
       gm0_out=>g20_0,gm1_out=>g20_1,gm2_out=>g20_2,gm3_out=>g20_3,
       d=>b4_7i,clk=>clk,reset=>reset);
 buff4_7: buffxx
 generic map (width=>4,depth=>7)
    port map  (din=>b4_7i,dout=>b4_7o,clk=>clk,reset=>reset);
 TraceBack4: TB
   port map (state_in=>s4,d=>b4_7o,state_out=>s5,
          clk=>clk,reset=>reset);
 ----------------------------------------------------------
 stage_21:acs4
    port map (sym=>x9_20,
       gm0_in=>g20_0, gm1_in=>g20_1, gm2_in=>g20_2, gm3_in=>g20_3,
       gm0_out=>g21_0,gm1_out=>g21_1,gm2_out=>g21_2,gm3_out=>g21_3,
       d=>b4_5i,clk=>clk,reset=>reset);
 buff4_5: buffxx
 generic map (width=>4,depth=>5)
    port map  (din=>b4_5i,dout=>b4_5o,clk=>clk,reset=>reset);
 TraceBack3: TB
```

```
  port map (state_in=>s3,d=>b4_5o,state_out=>s4,
         clk=>clk,reset=>reset);
------------------------------------------------------
stage_22:acs4
   port map (sym=>x10_21,
      gm0_in=>g21_0, gm1_in=>g21_1, gm2_in=>g21_2, gm3_in=>g21_3,
      gm0_out=>g22_0,gm1_out=>g22_1,gm2_out=>g22_2,gm3_out=>g22_3,
      d=>b4_3i,clk=>clk,reset=>reset);
buff4_3: buffxx
generic map (width=>4,depth=>3)
   port map  (din=>b4_3i,dout=>b4_3o,clk=>clk,reset=>reset);
TraceBack2: TB
  port map (state_in=>s2,d=>b4_3o,state_out=>s3,
         clk=>clk,reset=>reset);
------------------------------------------------------
stage_23:acs4
   port map (sym=>x11_22,
      gm0_in=>g22_0, gm1_in=>g22_1, gm2_in=>g22_2, gm3_in=>g22_3,
      gm0_out=>g23_0,gm1_out=>g23_1,gm2_out=>g23_2,gm3_out=>g23_3,
      d=>b4_1i,clk=>clk,reset=>reset);
buff4_1: buffx1
generic map (width=>4)
   port map  (din=>b4_1i,dout=>b4_1o,clk=>clk,reset=>reset);
TraceBack1: TB
  port map (state_in=>s1,d=>b4_1o,state_out=>s2,
         clk=>clk,reset=>reset);
------------------------------------------------------
csmap: cs
  port map (gm0=>g23_0,gm1=>g23_1,gm2=>g23_2,gm3=>g23_3,
         selec=>s1,clk=>clk,reset=>reset);
------------------------------------------------------
end arch_top;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity bm is
   port (
    sym: in std_logic_vector (5 downto 0);
        bm00: out std_logic_vector (3 downto 0);
        bm11: out std_logic_vector (3 downto 0);
        bm10: out std_logic_vector (3 downto 0);
        bm01: out std_logic_vector (3 downto 0);
```

```
          clk: in STD_LOGIC;
          reset: in STD_LOGIC);
end bm;

architecture bm_arch of bm is

component hadder port(
               a,b:  in std_logic;s,
               cout: out std_logic);
end component;

component fadder port(
               a,b,cin: in std_logic;s,
               cout:    out std_logic);
end component;

signal nsym:std_logic_vector(5 downto 0);
signal bm00t,bm01t,bm10t,bm11t:
     std_logic_vector(3 downto 0);
signal cout0,cout1,cout2,cout3:
     std_logic_vector(1 downto 0);
begin
getnsym: for i in 0 to 5 generate
       nsym(i)<=not sym(i);
end generate getnsym;

bm00_0: hadder port map
 (a=>sym(3),b=>sym(0),s=>bm00t(0),cout=>cout0(0));
bm00_1: fadder port map
(a=>sym(4),b=>sym(1),cin=>cout0(0),s=>bm00t(1),
 cout=>cout0(1));
bm00_23: fadder port map (a=>sym(5),b=>sym(2),cin=>cout0(1),s=>bm00t(2),
 cout=>bm00t(3));
bm01_0: hadder port map (a=>sym(3),b=>nsym(0),s=>bm01t(0),cout=>cout1(0));
bm01_1: fadder port map (a=>sym(4),b=>nsym(1),cin=>cout1(0),s=>bm01t(1),
 cout=>cout1(1));
bm01_23: fadder port map (a=>sym(5),b=>nsym(2),cin=>cout1(1),s=>bm01t(2),
 cout=>bm01t(3));
bm10_0: hadder port map (a=>nsym(3),b=>sym(0),s=>bm10t(0),cout=>cout2(0));
bm10_1: fadder port map (a=>nsym(4),b=>sym(1),cin=>cout2(0),s=>bm10t(1),
 cout=>cout2(1));
bm10_23: fadder port map (a=>nsym(5),b=>sym(2),cin=>cout2(1),s=>bm10t(2),
 cout=>bm10t(3));
bm11_0: hadder port map (a=>nsym(3),b=>nsym(0),s=>bm11t(0),cout=>cout3(0));
bm11_1: fadder port map (a=>nsym(4),b=>nsym(1),cin=>cout3(0),s=>bm11t(1),
 cout=>cout3(1));
 bm11_23: fadder port map (a=>nsym(5),b=>nsym(2),cin=>cout3(1),s=>bm11t(2),
  cout=>bm11t(3));

  process(clk,reset)
   begin
   if reset='1' then
    bm00<="0000";
    bm11<="0000";
    bm10<="0000";
    bm01<="0000";
   elsif clk'event and clk='1' then  --CLK rising edge
    bm00<=bm00t;
    bm01<=bm01t;
    bm10<=bm10t;
    bm11<=bm11t;
  end if;
 end process;
end bm_arch;

------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ACS is
   port (bm0: in std_logic_vector (3 downto 0);
           bm1: in std_logic_vector (3 downto 0);
           sm0: in std_logic_vector (6 downto 0);
           sm1: in std_logic_vector (6 downto 0);
           sm: out std_logic_vector (6 downto 0);
           d:  out std_logic;
           clk: in std_logic;
           reset: std_logic);
end ACS;

architecture arch_ACS of ACS is

signal sm0t,sm1t:std_logic_vector(7 downto 0);

begin
   sm0t<=("0"&sm0) + ("0000"&bm0);
   sm1t<=("0"&sm1) + ("0000"&bm1);

process (clk,reset)
```

```
begin
  if reset='1' then
   sm <= (others=>'0');
   d<='0';
  elsif clk'event and clk='1' then
   if (sm0t<=sm1t) then
       sm <= sm0t(6 downto 0); d<='0';
    else sm <= sm1t(6 downto 0);d<='1';
   end if;
  end if;
end process;
end arch_ACS;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity acs4 is

 port (sym: in std_logic_vector (5 downto 0);
     gm0_in,gm1_in,gm2_in,gm3_in:
     in std_logic_vector (6 downto 0);
     gm0_out,gm1_out,gm2_out,gm3_out:
     out std_logic_vector (6 downto 0);
     d: out STD_LOGIC_vector (3 downto 0);
     clk: in STD_LOGIC; reset:in std_logic);
end acs4;

architecture arch_acs4 of acs4 is

component qunt2bm port (sym: in std_logic_vector (5 downto 0);
        bm00,bm11,bm10,bm01: out std_logic_vector (3 downto 0);
        clk,reset: std_logic);
end component;

component ACS port (bm0: in std_logic_vector (3 downto 0);
                   bm1: in std_logic_vector (3 downto 0);
                   sm0: in std_logic_vector (6 downto 0);
                   sm1: in std_logic_vector (6 downto 0);
                   sm: out std_logic_vector (6 downto 0);
                   d: out std_logic;
                   clk: in std_logic;
                   reset : in std_logic);
end component;
```

```
signal b00,b11,b01,b10: std_logic_vector (3 downto 0);

begin

bm1:qunt2bm port map (sym=>sym,
                      bm00=>b00,bm11=>b11,bm10=>b10,bm01=>b01,
                      clk=>clk,reset=>reset);

acs1: ACS port map (bm0=>b00,bm1=>b11,sm0=>gm0_in,sm1=>gm1_in,
                    d=>d(0),sm=>gm0_out,clk=>clk,reset=>reset);
acs2: ACS port map (bm0=>b11,bm1=>b00,sm0=>gm0_in,sm1=>gm1_in,
                    d=>d(2),sm=>gm2_out,clk=>clk,reset=>reset);
acs3: ACS port map (bm0=>b10,bm1=>b01,sm0=>gm2_in,sm1=>gm3_in,
                    d=>d(1),sm=>gm1_out,clk=>clk,reset=>reset);
acs4: ACS port map (bm0=>b01,bm1=>b10,sm0=>gm2_in,sm1=>gm3_in,
                    d=>d(3),sm=>gm3_out,clk=>clk,reset=>reset);
end arch_acs4;

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity cs is
   port ( gm0,gm1,gm2,gm3:in std_logic_vector(6 downto 0);
          selec:    out std_logic_vector (1 downto 0);
          clk: in STD_LOGIC; RESET:in std_logic);
end cs;

architecture arch_cs of cs is

signal a,b,c,d,e,f: std_logic;
signal sel: std_logic_vector(1 downto 0);

begin
a<='0' when gm0<=gm1 else '1';
b<='0' when gm0<=gm2 else '1';
c<='0' when gm0<=gm3 else '1';
d<='0' when gm1<=gm2 else '1';
e<='0' when gm1<=gm3 else '1';
f<='0' when gm2<=gm3 else '1';
sel(1)<=((a or b or c) and ( not a or d or e)) or (b and d and not f) or (c and e and f);
sel(0)<=((a or b or c) and (not b or not d or f)) or(a and not d and not e) or (c and e and f);
```

```
process(clk,reset)
  begin
   if reset='1' then
    selec<="00";
   elsif clk'event and clk='1' then
    selec<=sel;
  end if;
end process;
end arch_cs;
------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity TB is
   port (state_in: in STD_LOGIC_VECTOR (1 downto 0);
          d: in STD_LOGIC_vector (3 downto 0);
          state_out: out std_logic_vector (1 downto 0);
          clk: in std_logic;
          reset: in std_logic);
end TB;

architecture TB_arch of TB is
signal tmp: std_logic_vector (1 downto 0);
begin
 tmp(1)<=state_in(0);
 with state_in select
 tmp(0)<= d(0)  when "00",
        d(1)  when "01",
        d(2)  when "10",
        d(3)  when "11",
        'X' when others;

process (clk,reset)
 begin
  if reset='1' then
    state_out<= "00";
   elsif clk'event and clk='1' then
    state_out <= tmp;
  end if;
 end process;
end TB_arch;

------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity buffl1 is
   port (din:  in std_logic;
           dout: out std_logic;
           clk: in std_logic;
           reset:in std_logic);
end buffl1;

architecture arch_buffl1 of buffl1 is
begin
  process(clk,reset)
   begin
    if reset='1' then
     dout<='0';
    elsif clk'event and clk='1' then
      dout<=din;
    end if;
  end process;
end arch_buffl1;

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

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.all;

entity bufflx is
generic(depth:integer);
   port (din:  in  std_logic;
           dout: out std_logic;
           clk: in std_logic;
           reset:in std_logic);
end bufflx;

architecture arch_bufflx of bufflx is

component buffl1
  port (din:  in std_logic;
        dout: out std_logic;
        clk: in std_logic;
        reset:in std_logic);
end component;
```

```
signal x: std_logic_vector(1 to depth);

begin
  cascade: for i in 1 to depth generate

   first_stage: if i=1 generate
    firststagemap: buffl1
              port map (din=>din,dout=>x(i),clk=>clk,reset=>reset);
   end generate first_stage;

   mid_stages: if (i>1 and i<depth) generate
    midstagesmap: buffl1
             port map (din=>x(i-1),dout=>x(i),clk=>clk,reset=>reset);
   end generate mid_stages;

   last_stage: if (i=depth) generate
    laststagemap:buffl1
             port map (din=>x(i-1),dout=>dout,clk=>clk,reset=>reset);
   end generate last_stage;

  end generate cascade;
end arch_bufflx;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity buffx1 is
generic(width:integer);
   port (
      din:  in std_logic_vector(width-1 downto 0);
      dout: out std_logic_vector(width-1 downto 0);
      clk: in std_logic;
      reset:in std_logic);
end buffx1;

architecture arch_buffx1 of buffx1 is
begin
  process(clk,reset)
   begin
    if reset='1' then
     dout<=(others=>'0');
   elsif clk'event and clk='1' then
```

```
    dout<=din;
   end if;
  end process;
 end arch_buffx1;

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

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.all;

entity buffxx is
generic(width:integer;depth:integer);
  port (
     din:  in  std_logic_vector(width-1 downto 0);
     dout: out std_logic_vector(width-1 downto 0);
     clk: in std_logic;
     reset:in std_logic);
end buffxx;

architecture arch_buffxx of buffxx is

component buffx1
  generic (width:integer);
  port (din:  in std_logic_vector(width-1 downto 0);
      dout: out std_logic_vector(width-1 downto 0);
      clk: in std_logic;
      reset:in std_logic);
end component;

type vctr is array (1 to depth) of std_logic_vector (width-1 downto 0);
signal x: vctr;

begin
 cascade: for i in 1 to depth generate

  first_stage: if i=1 generate

   firststagemap: buffx1
              generic map (width=> width)
              port map (din=>din,dout=>x(i),clk=>clk,reset=>reset);
  end generate first_stage;

  mid_stages: if (i>1 and i<depth) generate
   midstagesmap: buffx1
```

```
            generic map (width=> width)
            port map (din=>x(i-1),dout=>x(i),clk=>clk,reset=>reset);
  end generate mid_stages;

  last_stage: if (i=depth) generate
   laststagemap:buffx1
            generic map (width=> width)
            port map (din=>x(i-1),dout=>dout,clk=>clk,reset=>reset);
  end generate last_stage;

 end generate cascade;
end arch_buffxx;

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

library IEEE;
use IEEE.std_logic_1164.all;

entity fadder is
  port (a: in std_logic;
        b: in std_logic;
        cin: in std_logic;
        s: out std_logic;
        cout: out std_logic);
end fadder;

architecture arch_fadder of fadder is
signal axorb: std_logic;
begin
 axorb <= a xor b;
 cout <= (a and b)
or (axorb and cin);
 s <= cin xor axorb;

end arch_fadder;

configuration cfg_fadder of fadder is
 for arch_fadder
 end for ;
end cfg_fadder ;

-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity hadder is
  port (a: in std_logic;
          b: in std_logic;
          s: out std_logic;
          cout: out std_logic);
end hadder;

architecture arch_hadder of hadder is
signal axorb: std_logic;
begin
 s<= a xor b;
 cout <= a and b;
end arch_hadder;

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

package CONSTANTS is
  constant PERIOD : time := 12 ns ;
  constant HALF_PERIOD : time := PERIOD / 2 ;
end CONSTANTS ;

library STD ;
library IEEE ;
use std.textio.all ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_textio.all ;
use Work.constants.all ;
use IEEE.std_logic_arith.all;
use Work.all ;

entity testbench is
end testbench;

architecture arch_testbench of testbench is

  component top
port (
      x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11: in std_logic_vector(5 downto 0);
      clock,reset: in STD_LOGIC;
      y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11: out STD_LOGIC);
  end component;

signal x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11: std_logic_vector (5 downto 0);
signal clock,reset: STD_LOGIC;
```

```
 signal y: STD_LOGIC_vector (0 to 11);

begin

  UUT : top
    Port Map ( x0=>x0,x1=>x1,x2=>x2,x3=>x3,x4=>x4,x5=>x5,
              x6=>x6,x7=>x7,x8=>x8,x9=>x9,x10=>x10,x11=>x11,
              clock=>clock,reset=>reset,
              y0=>y(0),y1=>y(1),y2=>y(2),y3=>y(3),
              y4=>y(4),y5=>y(5),y6=>y(6),y7=>y(7),y8=>y(8),
              y9=>y(9),y10=>y(10),y11=>y(11) );

  STIMULUS : process
  file TVin : TEXT is in "quantzd.dat" ;
  file TVout : TEXT is out "decoded.dat" ;
  variable INline , OUTline : LINE ;
  variable reseti: std_logic;
  variable x0i,x1i,x2i,x3i,x4i,x5i,x6i,x7i,x8i,x9i,x10i,x11i:
        std_logic_vector(5 downto 0);

  begin
  readline( TVin , INline ) ;
  read( INline , reseti ) ;
  read( INline , x0i );  read( INline , x1i );
  read( INline , x2i );  read( INline , x3i );
  read( INline , x4i );  read( INline , x5i );
  read( INline , x6i );  read( INline , x7i );
  read( INline , x8i );  read( INline , x9i );
  read( INline , x10i );  read( INline , x11i );

    clock <= '0';
    reset <=reseti;

    x0 <= x0i; x1 <= x1i; x2 <= x2i;
    x3 <= x3i; x4 <= x4i; x5 <= x5i;
    x6 <= x6i; x7 <= x7i; x8 <= x8i;
    x9 <= x9i; x10 <= x10i; x11 <= x11i;
    wait for HALF_PERIOD;

    clock <= '1';
    wait for HALF_PERIOD;

 for i in 1 to 12 loop
   clock<='0';
   wait for half_period;
   clock<='1';
```

```
      wait for half_period;
    end loop;
      clock<='0';
      wait for half_period;

      while not endfile( TVin) loop
       --Get a vector
       readline( TVin , INline ) ;
       read( INline , reseti ) ;
       read( INline , x0i );  read( INline , x1i );
       read( INline , x2i );  read( INline , x3i );
       read( INline , x4i );  read( INline , x5i );
       read( INline , x6i );  read( INline , x7i );
       read( INline , x8i );  read( INline , x9i );
       read( INline , x10i );  read( INline , x11i );

       clock <= '0';
       reset <=reseti;

       x0 <= x0i; x1 <= x1i; x2 <= x2i;
       x3 <= x3i; x4 <= x4i; x5 <= x5i;
       x6 <= x6i; x7 <= x7i; x8 <= x8i;
       x9 <= x9i; x10 <= x10i; x11 <= x11i;
       wait for HALF_PERIOD;

       clock <= '1';
       wait for HALF_PERIOD;
       --Write output values
       write(OUTline , y) ;
       writeline( TVout , OUTline );

    end loop;

    for i in 1 to 42 loop
     clock<='0';
     wait for half_period;
     clock<='1';
     wait for half_period;
     write(OUTline , y) ;
      writeline( TVout , OUTline );
    end loop;

  assert false report "test complete";
  end process ;
end arch_testbench;
```

Appendix F. Area and Timing Report for the TOP Entity

```
****************************************
Report : fpga
Design : top
Version: 2000.05
Date   : Mon Oct  9 11:27:25 2000
****************************************

   LUT FPGA Design Statistics
   ---------------------------

   * Core Cell Statistics *
   Number of 2-input LUT cells:          137
   Number of 3-input LUT cells:          817
   Number of 4-input LUT cells:          276
   Number of Core Flip Flops:            3598
   Number of Core 3-State Buffers:       0
   Number of Other Core Cells:           2346
   Total Number of Core Cells:           11261

   * Port Statistics *
   Number of Input Ports:                74
   Number of Output Ports:               12
   Number of Bi-directional Ports:       0
   Total Number of Ports:                86

   * Pad Cell Statistics *
   Number of Input Pads:                 74
   Number of Output Pads:                12
   Number of Clock Pads:                 0
   Total Number of Pads Cells:           86

   FPGA Specific Attributes
   -------------------------


****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : top
Version: 2000.05
Date   : Mon Oct  9 11:30:57 2000
****************************************

Operating Conditions: WCCOM   Library: xfpga_virtex-6
Wire Load Model Mode: top

  Startpoint: stage_0/acs3/sm_reg<0>
              (rising edge-triggered flip-flop clocked by clock)
  Endpoint: stage_1/acs1/sm_reg<0>
            (rising edge-triggered flip-flop clocked by clock)
```

```
Path Group: clock
Path Type: max

Des/Clust/Port      Wire Load Model      Library
------------------------------------------------
top                 xcv300-6_avg         xfpga_virtex-6

Point                                                       Incr       Path
-----------------------------------------------------------------------------
clock clock (rise edge)                                     0.00       0.00
clock network delay (ideal)                                 0.00       0.00
stage_0/acs3/sm_reg<0>/C (FDC)                              0.00       0.00 r
stage_0/acs3/sm_reg<0>/Q (FDC)                              2.60       2.60 f
stage_0/acs3/sm<0> (ACS)                                    0.00       2.60 f
stage_0/gm1_out<0> (acs4)                                   0.00       2.60 f
stage_1/gm1_in<0> (acs4)                                    0.00       2.60 f
stage_1/acs1/sm1<0> (ACS)                                   0.00       2.60 f
stage_1/acs1/add_23/plus/plus/A<0> (ACS_xdw_add_8_0)
                                                            0.00       2.60 f
stage_1/acs1/add_23/plus/plus/A_LUT/LO (DWLUT2_L)           0.58       3.18 f
stage_1/acs1/add_23/plus/plus/A_CY/LO (MUXCY_L)             0.90       4.08 f
stage_1/acs1/add_23/plus/plus/A_CY_1/LO (MUXCY_L)           0.05       4.13 f
stage_1/acs1/add_23/plus/plus/A_CY_2/LO (MUXCY_L)           0.05       4.18 f
stage_1/acs1/add_23/plus/plus/A_CY_3/LO (MUXCY_L)           0.05       4.23 f
stage_1/acs1/add_23/plus/plus/A_CY_4/LO (MUXCY_L)           0.05       4.28 f
stage_1/acs1/add_23/plus/plus/A_XOR_5/O (XORCY)             1.73       6.01 r
stage_1/acs1/add_23/plus/plus/S<5> (ACS_xdw_add_8_0)
                                                            0.00       6.01 r
stage_1/acs1/lte_30/leq/leq/A<5> (ACS_xdw_comp_uns_8_0)
                                                            0.00       6.01 r
stage_1/acs1/lte_30/leq/leq/A_LUT_5/LO (DWLUT2_L)           0.58       6.59 r
stage_1/acs1/lte_30/leq/leq/A_CY_5/LO (MUXCY_L)             0.90       7.49 r
stage_1/acs1/lte_30/leq/leq/A_CY_6/LO (MUXCY_L)             0.05       7.54 r
stage_1/acs1/lte_30/leq/leq/A_CFY/O (MUXCY)                 2.05       9.59 r
stage_1/acs1/lte_30/leq/leq/A_GE_B (ACS_xdw_comp_uns_8_0)
                                                            0.00       9.59 r
stage_1/acs1/U43/O (LUT3)                                   1.28      10.87 r
stage_1/acs1/sm_reg<0>/D (FDC)                              0.00      10.87 r
data arrival time                                                     10.87

clock clock (rise edge)                                    50.00      50.00
clock network delay (ideal)                                 0.00      50.00
stage_1/acs1/sm_reg<0>/C (FDC)                              0.00      50.00 r
library setup time                                         -0.40      49.60
data required time                                                    49.60
-----------------------------------------------------------------------------
data required time                                                    49.60
data arrival time                                                    -10.87
-----------------------------------------------------------------------------
slack (MET)                                                           38.73
```

Appendix G. The script for Synopsys Compiler

```
TOP = top
edifout_design_name = top

designer = "Jian Lin"
company  = "U of M"
part     = "XCV300PQ240-6"

analyze -format vhdl \
{./HDLs/buff.vhd ./HDLs/bitcomp.vhd \
./HDLs/hadder.vhd ./HDLs/fadder.vhd \
./HDLs/comparex.vhd ./HDLs/qunt2bm.vhd \
./HDLs/tb.vhd \
./HDLs/acs_dw.vhd ./HDLs/cs.vhd \
./HDLs/topNosmgrs1.vhd}

elaborate qunt2bm
ungroup -all
remove_constraint -all
remove_clock -all
create_clock  "clk" -period 50
group_path -critical_range 10000 -default
compile -map_effort high
report_fpga
report_timing

elaborate acs
remove_constraint -all
remove_clock -all
create_clock  "clk" -period 50
group_path -critical_range 10000 -default
compile -map_effort high
report_fpga
report_timing

elaborate acs4
set_dont_touch {bm1,acs1,acs2,acs3,acs4}
remove_constraint -all
remove_clock -all
create_clock  "clk" -period 50
group_path -critical_range 10000 -default
compile -map_effort high
report_fpga
report_timing

elaborate cs
remove_constraint -all
remove_clock -all
create_clock  "clk" -period 50
group_path -critical_range 10000 -default
compile -map_effort high
report_fpga
report_timing
```

```
elaborate buff11
compile

elaborate buff1x -param "depth = 2"
uniquify
compile

elaborate buff1x -param "depth = 3"
uniquify
compile

elaborate buff1x -param "depth = 4"
uniquify
compile

elaborate buff1x -param "depth = 5"
uniquify
compile

elaborate buff1x -param "depth = 6"
uniquify
compile

elaborate buff1x -param "depth = 7"
uniquify
compile

elaborate buff1x -param "depth = 8"
uniquify
compile

elaborate buff1x -param "depth = 9"
uniquify
compile

elaborate buff1x -param "depth = 10"
uniquify
compile

elaborate buff1x -param "depth = 11"
uniquify
compile

elaborate buffx1 -param "width = 6"
uniquify
compile

elaborate buffx1 -param "width = 4"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 2"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 3"
uniquify
```

```
compile

elaborate buffxx -param "width = 6, depth= 4"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 5"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 6"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 7"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 8"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 9"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 10"
uniquify
compile

elaborate buffxx -param "width = 6, depth= 11"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 33"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 31"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 29"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 27"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 25"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 23"
uniquify
compile
```

```
elaborate buffxx -param "width = 4, depth= 21"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 19"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 17"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 15"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 13"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 11"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 9"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 7"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 5"
uniquify
compile

elaborate buffxx -param "width = 4, depth= 3"
uniquify
compile

elaborate tb
compile -map_effort high
report_fpga
report_timing

elaborate top -arch "arch_top"
set_dont_touch { DLL,CLOCKBUF,pipex1_1, \
                 pipex1_12,pipex2_2,pipex3_3, \
                 pipex4_4,pipex5_5,pipex6_6, \
                 pipex7_7,pipex8_8,pipex9_9, \
                 pipex10_10,pipex11_11, \
                 pipex0_11,pipex2_13,pipex3_14, \
                 pipex4_15,pipex5_16,pipex6_17, \
                 pipex7_18,pipex8_19,pipex9_20, \
                 pipex10_21,pipex11_22, \
                 stage_0, stage_1, stage_2, \
                 stage_3, stage_4, stage_5, \
                 stage_6, stage_7, stage_8, \
                 stage_9, stage_10, stage_11, \
                 stage_12, stage_13, stage_14, \
                 stage_15, stage_16, stage_17, \
                 stage_18, stage_19, stage_20, \
                 stage_21, stage_22, stage_23, \
                 buff4_33, buff4_31, buff4_29, \
                 buff4_27, buff4_25, buff4_23, \
                 buff4_21, buff4_19, buff4_17, \
                 buff4_15, buff4_13, buff4_11, \
                 buff4_9, buff4_7, buff4_5, \
                 buff4_3, buff4_1, \
                 TraceBack1, TraceBack2, TraceBack3, \
                 TraceBack4, TraceBack5, TraceBack6, \
                 TraceBack7, TraceBack8, TraceBack9, \
                 TraceBack10, TraceBack11, TraceBack12, \
                 TraceBack13, TraceBack14, TraceBack15, \
                 TraceBack16, TraceBack17, \
                 csmap}

set_port_is_pad "*"
remove_attribute find(port,"clock")port_is_pad

set_pad_type -exact IBUF {x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11}
set_pad_type -exact OBUF {y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11}
set_pad_type -slewrate HIGH {y0,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11}

insert_pads

remove_constraint -all
remove_clock -all
create_clock  "clock" -period 50
group_path -critical_range 10000 -default
compile -map_effort high
report_fpga
report_timing

write -format db -hierarchy -output top + ".db"

set_attribute TOP "part" -type string part

write -format edif -hierarchy -output TOP + ".sedif"

write_script > TOP + ".dc"

sh dc2ncf -w TOP + ".dc"

exit
```

Appendix H. The script for Placement and Routing

```
#!/bin/csh -f
ngdbuild -p xcv300-6-pq240 -uc top.ucf top.sedif top.ngd
map -u -o top_m.ncd top.ngd top.pcf
par -w -ol 2 -d 0 top_m.ncd  top_r.ncd top.pcf
ngdanno -s 6 -o top_anno.nga top_r.ncd top_m.ngm
ngd2vhdl top_anno.nga -w top_time.vhd
```

Appendix I. The script for Timing Simulation

```
#!/usr/local/bin/perl

$LOGFILE = "tmp.dat";
open(LOGFILE) or die("Could not open log file.");
read(LOGFILE, $line,30);
close(LOGFILE);
($msg, $err, $esovrn) =split(' ',$line);
$esovrn = substr($esovrn,0,3);
while ($esovrn<=8.5) {
 print("$esovrn");

 system("nice -19 encoder $esovrn");
 system("nice -19 vhdlsim -nc -sdf_top /testbench/uut -sdf top_time.sdf
conf_testbench -e my");
 system("comp");

 $LOGFILE = "tmp.dat";
 open(LOGFILE) or die("Could not open log file.");
 read(LOGFILE, $line,30);
 close(LOGFILE);
 ($msg, $err, $esovrn) = split(' ',$line);
 $esovrn = substr($esovrn,0,3);
};
print("The test is over! \n");
```

Appendix J. The Report for Placement and Routing

```
Release 3.1.01i - Par D.19

Mon Nov 13 14:41:08 2000

par -w -ol 5 -d 0 map.ncd top.ncd top.pcf


Constraints file: top.pcf

Loading device database for application par from file "map.ncd".
   "top" is an NCD, version 2.32, device xcv300e, package pq240, speed -8
Loading device for application par from file 'v300e.nph' in environment
/CMC/tools/xilinx.
Device speed data version:  PREVIEW 1.33 2000-06-16.


Device utilization summary:

   Number of External GCLKIOBs         1 out of 4      25%
   Number of External IOBs            85 out of 158    53%

   Number of SLICEs                 2924 out of 3072   95%

   Number of DLLs                      1 out of 8      12%
   Number of GCLKs                     2 out of 4      50%


Overall effort level (-ol):   5 (set by user)
Placer effort level (-pl):    5 (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    5 (set by user)

Starting initial Timing Analysis.  REAL time: 25 secs
Finished initial Timing Analysis.  REAL time: 44 secs

Starting initial Placement phase. REAL time: 48 secs
Finished initial Placement phase. REAL time: 52 secs
Starting the placer. REAL time: 53 secs
Placement pass 1
.......................................................................
Placer score = 408090
Optimizing ...
Placer score = 343675
Improving the placement. REAL time: 2 mins 3 secs
Placer stage completed in real time: 5 mins 44 secs

Optimizing ...
Starting IO Improvement.  REAL time: 6 mins 31 secs
Placer score = 312010
Finished IO Improvement.  REAL time: 6 mins 31 secs

Placer completed in real time: 6 mins 31 secs

Writing design to file "top.ncd".

Total REAL time to Placer completion: 6 mins 50 secs
Total CPU time to Placer completion: 6 mins 38 secs
```

```
0 connection(s) routed; 14510 unrouted.
Starting router resource preassignment
Completed router resource preassignment. REAL time: 7 mins 13 secs
Starting iterative routing.
Routing active signals.
.........
End of iteration 1
14510 successful; 0 unrouted; (0) REAL time: 10 mins 48 secs
Constraints are met.
Total REAL time: 11 mins 2 secs
Total CPU  time: 9 mins
End of route.  14510 routed (100.00%); 0 unrouted.
No errors found.
Completely routed.

Total REAL time to Router completion: 11 mins 19 secs
Total CPU time to Router completion: 9 mins 12 secs

Generating PAR statistics.

   The Delay Summary Report

   The Score for this design is: 162


The Number of signals not completely routed for this design is: 0

   The Average Connection Delay for this design is:        1.146 ns
   The Maximum Pin Delay is:                               3.974 ns
   The Average Connection Delay on the 10 Worst Nets is:   2.386 ns

   Listing Pin Delays by value: (ns)

    d < 1.00   < d < 2.00  < d < 3.00  < d < 4.00  < d < 5.00  d >= 5.00
   ---------   ---------   ---------   ---------   ---------   ---------
        8196        4303         191        1820           0           0

Timing Score: 0

Asterisk (*) preceding a constraint indicates it was not met.

--------------------------------------------------------------------------------
  Constraint                                  | Requested    | Actual       | Logic
                                              |              |              | Levels
--------------------------------------------------------------------------------
  TS01 = MAXDELAY FROM TIMEGRP "FFS" TO TIM    | 12.000ns     | 10.145ns     | 8
  EGRP "FFS" 12 nS                            |              |              |
--------------------------------------------------------------------------------
  TS_DLL_CLK0 = PERIOD TIMEGRP "DLL_CLK0"     |              |              |
  50 nS   HIGH 50.000 %                       |              |              |
--------------------------------------------------------------------------------
  TS_DLL_CLK0_0 = PERIOD TIMEGRP "DLL_CLK0_   |              |              |
  0"  50 nS   HIGH 50.000 %                   |              |              |
--------------------------------------------------------------------------------


All constraints were met.
Writing design to file "top.ncd".


All signals are completely routed.
```

```
Total REAL time to PAR completion: 12 mins 12 secs
Total CPU time to PAR completion: 9 mins 43 secs

Placement: Completed - No errors found.
Routing: Completed - No errors found.
Timing: Completed - No errors found.

PAR done.
```

Reference

[1] A. Viterbi, "Error bounds for convolutional coding and an asymptotically optimum decoding algorithm", IEEE trans. Inform. Theory, vol. IT-13, pp260-269, Apr. 1967.

[2] A.J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding.* New York: McGraw- Hill, 1979, pp. 229-230.

[3] A. M. Michelson and /a. /h. Levesque, *Error-Control Techniques for Digital Communication.* John Wily & sons, 1985, pp. 15, 29.

[4] G. C. Clark and J. B. Cain, Error-Correction Coding for Digital Communication. NewYork: Plenum, 1981, pp. 227-264.

[5] A.J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding.* New York: McGraw- Hill, 1979, pp. 258-261

[6] H. F. Lin and D. G. Messerschmitt, " Algorithms and architectures for concurrent Viterbi decoding," in Proc. ICC'89, June 1989, vol.2, pp. 836-840.

[7] K. -H Tzou and J. G. Dunham, " Sliding block decoding of convolutional codes," IEEE Trans. Commun., vol. COM-29,pp. 1401-1403, Sept. 1981.

[8] G. Fettweis, H. Dawid, and H. Meyr, " Minimized method Viterbi decoding: 600 Mb/s per chip", in *Proc. GLOBECOM 90*, VOL. 3, Dec.1990, pp. 1712-1716.

[9] Peter J. Black and Teresa H.-Y. Meng, " Hybrid Suvivor Architecture for Viterbi Decoders" *IEEE J. Solid-State Circuits*, vol.

[10] Peter J. Black and Teresa H.-Y. Meng, "A 1-Gb/s, Four-State, Sliding Block Viterbi Decoder" *IEEE Trans. Commun.*, vol. COM-32, pp. 797-805, June 1997.

[11] A. P. Hekstra, " An alternative to metric rescaling in Viterbi decoders," IEEE Trans. Commun., vol.37, no. 11, pp. 1220-1222, Nov. 1989.

[12] A.J. Viterbi and J. K. Omura, *Principles of Digital Communication and Coding.* New York: McGraw-Hill, 1979, pp. 258-261.

[13] J. G. Proakis and M. Salehi, *Contemporary Communication Systems USING MATLAB*. PWS Publishing Company, 1998, pp 49-50.

[14] J. A. Heller and I. M. Jacobs, "*Viterbi decoding for satellite and space communication*" IEEE Trans. Commun. Techhnol., vol. COM-19, pp. 835-848, Oct. 1971.

[15] I. M. Jacobs, " Sequential Decoding for effient communication from deep space," *IEEE Trans. Commun. Technol.*, vol. COM-15, Aug. 1967, pp. 492-501.

[16] *Xilinx Data Book* : http://www.xilinx.com.

[17] *FPGA Compiler User Guide* from Synopsys Online Document. pp. 7-21.