On a Viterbi decoder design for low power dissipation

By

Samirkumar Ranpara

Thesis submitted to the Faculty
of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Electrical Engineering

Dr. Dong S. Ha, Chairman
Dr. Nathaniel J. Davis IV
Dr. James R. Armstrong

April, 1999
Blacksburg, Virginia

Keywords : Viterbi decoder, Low power, Full scan, Synopsys

On a Viterbi decoder design for low power dissipation.

Samirkumar Ranpara
Dr. Dong S. Ha, Chairman
Bradley Department of Electrical and Computer Engineering.

(Abstract)

Convolutinal coding is a coding scheme often employed in deep space communications and recently in digital wireless communications. Viterbi decoders are used to decode convolutional codes. Viterbi decoders employed in digital wireless communications are complex and dissipate large power. With the proliferation of battery powered devices such as cellular phones and laptop computers, power dissipation, along with speed and area, is a major concern in VLSI design. In this thesis, we investigated a low-power design of Viterbi decoders for wireless communications applications. In CMOS technology the major source of power dissipation is attributed to dynamic power dissipation, which is due to the switching of signal values. The focus of our research in the low-power design of Viterbi decoders is reduction of dynamic power dissipation at logic level in the standard cell design environment. We considered two methods, clock-gating and toggle-filtering, in our design. A Viterbi decoder consists of five blocks. The clock-gating was applied to the survivor path storage block and the toggle-filtering to the trace-back block of a Viterbi decoder. We followed the standard cell design approach to implement the design. The behavior of a Viterbi decoder was described in VHDL, and then the VHDL description was modified to embed the low-power design. A gate level circuit was obtained from the behavioral description through logic synthesis, and a full scan design was incorporated into the gate level circuit to ease testing. The gate level circuit was placed and routed to generate a layout of the design. Our experimental result shows the proposed design reduces the power dissipation of a Viterbi decoder by about 42 percent compared with the on without considering the low-power design.

Acknowledgements

Special thanks are due to my committee chairman and advisor Dr Dong S. Ha. It was through his patience, understanding and invaluable guidance that this work was accomplished. I would also like to express my appreciation for my committee members Dr Nathaniel J. Davis IV and Dr James R. Armstrong for serving as my committee members and commenting on this work.

I am extremely grateful for the VISC computing resources without which none of this work would have been possible. I am greatly indebted to Dr. Will Ebel (Visiting Faculty), for his expert advice and corrections on this work. I would also like to first thank Han Bin and then to all other friends for their advise, guidance and help.

My parents and my wife didn't know what I was doing, but they were always eager to help me out in all possible ways; without them it is hard to imagine accomplishing all this work.

Contents

List Of Figures.

List Of Tables.

Chapter 1

Introduction

Convolutional coding has been used in communication systems including deep space communications and wireless communications. It offers an alternative to block codes for transmission over a noisy channel. An advantage of convolutional coding is that it can be applied to a continuous data stream as well as to blocks of data. IS-95, a wireless digital cellular standard for CDMA (code division multiple access), employs convolutional coding. A third generation wireless cellular standard, under preparation, plans to adopt turbo coding, which stems from convolutional coding.

The Viterbi decoding algorithm, proposed in 1967 by Viterbi, is a decoding process for convolutional codes in memory-less noise [52]. The algorithm can be applied to a host of problems encountered in the design of communication systems [52]. The Viterbi decoding algorithm provides both a maximum-likelihood and a maximum a posteriori algorithm. A maximum a posteriori algorithm identifies a code word that maximizes the conditional probability of the decoded code word against the received code word, in contrast a maximum likelihood algorithm identifies a code word that maximizes the conditional probability of the received code word against the decoded code word. The two algorithms give the same results when the source information has a uniform distribution.

Traditionally, performance and silicon area are the two most important concerns in VLSI design. Recently, power dissipation has also become an important concern, especially in battery-powered applications, such as cellular phones, pagers and laptop computers. Power dissipation can be classified into two categories, static power dissipation and dynamic power dissipation. Typically, static power dissipation is due to various leakage currents, while dynamic power dissipation is a result of charging and discharging the parasitic capacitance of transistors and wires. Since the dynamic power dissipation accounts for about 80 to 90 percent of overall power dissipation in CMOS circuits; numerous techniques have been proposed to reduce dynamic

power dissipation. These techniques can be applied at different levels of digital design, such as the algorithmic level, the architectural level, the gate level and, the circuit level.

In this thesis, a low-power design of Viterbi decoders at the gate level in the standard cell design environment is proposed. In the standard cell design environment, the behavior of a design is described in a high-level hardware description language, such as VHDL or Verilog. The behavioral design is synthesized to generate a gate level design. The gate-level design is placed and routed to generate a layout of the design. The advantages of a standard cell based design over full custom design are -- faster turn around time for the design, ease in design verification and more accurate modeling of the circuit.

Low-power design of Viterbi decoders at the gate-level is focused here. Viterbi algorithms [50], [21], [5] and implementation of Viterbi decoders [6], [28], [32], [35], [36], [43] were investigated intensively in the past three decades. Most relevant works in low-power design of Viterbi decoders include [23], [27], [28], [33], [36] and [43]. Seki et al, [43] and Lang et al, [33] suggested use of a scarce state transition (SST) scheme [32]. The scheme uses a simple pre-decoder and a pre-encoder to minimize transitions at the input of a Viterbi decoder. This reduces dynamic power dissipation. Kang and Wilson [27] suggested partitioning major blocks at the system level and the reduction of spurious transitions at a lower level. Garrett and Stan [23] suggest a specialized SRAM cell structure that allows a sequential write update and parallel read access across the memory in such a way that reduces dynamic power dissipation. The above mentioned works showed that their designs substantially reduces power dissipation of Viterbi decoders.

Unlike the existing approaches, we introduce low-power design techniques into the behavior of Viterbi decoder. After the behavior of a Viterbi decoder was described in VHDL, we modified the behavior of the circuit to reduce dynamic power dissipation. Two major techniques, clock gating and toggle filtering, were investigated in this thesis. In addition, a full scan for easy testing of the circuit was employed. In a full scan design, all sequential elements are controllable and observable during testing. In our experiments, estimated power dissipation was estimated on the basis of the switching activity measured through behavioral simulation. Experimental results indicate that our methods effectively reduce the power dissipation of Viterbi decoders.

This section describes the organization of this thesis. The background on the operation of convolutional encoders and Viterbi decoders is provided in Chapter 2. A brief description of low-power design techniques investigated in this thesis is also covered in this chapter. Chapter 3 proposes a low-power design for Viterbi decoders. It also discusses design alternatives for power dissipation. Chapter 4 describes the environment for our experiments and lists power dissipation results of original and proposed Viterbi decoders. Chapter 5 summarizes the thesis.

Chapter 2
Viterbi decoding algorithm and low-power design techniques

2.1 Overview and introduction

In this chapter, we provide necessary background on our research for a low-power design of Viterbi decoders for convolutional codes. First, convolutional codes are discussed. Then, we explain the Viterbi decoding algorithm and the Viterbi decoder design. The goal of our design is to achieve low-power dissipation, therefore design concepts and techniques for low power dissipation are also discussed.

2.2 Convolutional codes

The Viterbi decoding algorithm proposed in 1967 is a decoding process for convolutional codes. Convolutional coding has been used in communication systems including deep space communications and wireless communications. Convolutional codes offer an alternative to block codes for transmission over a noisy channel. Convolutional coding can be applied to a continuous input stream (which cannot be done with block codes), as well as blocks of data. In fact, a convolutional encoder can be viewed as a finite state machine. It generates a coded output data stream from an input data stream. It is usually composed of shift registers and a network of XOR (Exclusive-OR) gates as shown in Figure 2.1.

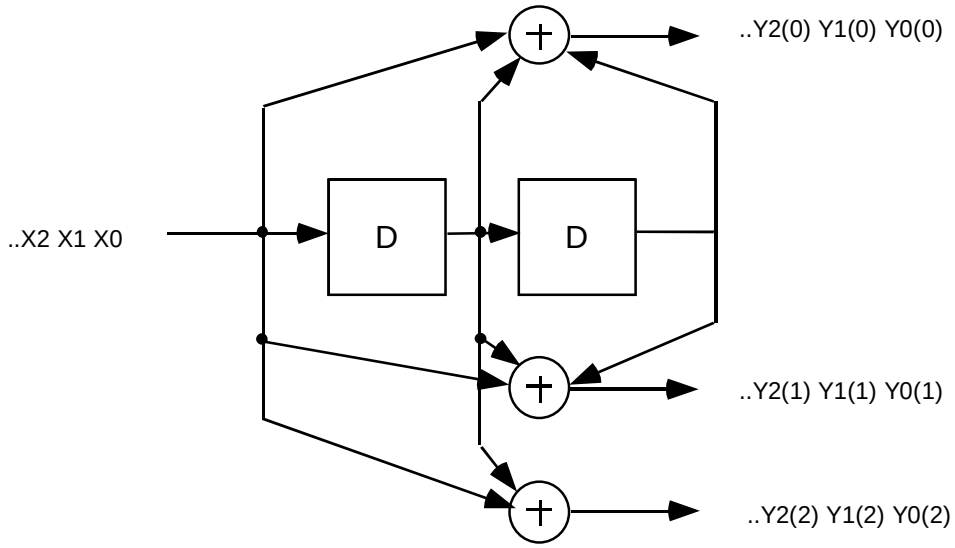

Figure 2.1 A rate-1/3 convolutional encoder [52]

The encoder in Figure 2.1 produces three bits of encoded information for each bit of input information, so it is called a rate 1/3 encoder. A convolutional encoder is generally characterized in (n, k, m) format, where

n is number of outputs of the encoder ;

k is number of inputs of the encoder ;

m is number of memory elements (flip-flops) of the longest shift register of the encoder. The rate of a (n,k,m) encoder is k/n. The encoder shown in the figure is a (3,1,2) encoder with rate 1/3. *In this thesis, we discuss decoding of convolutional codes generated by a (n,1,m) encoder with the rate 1/n.*

A convolutional encoder is a Mealy machine, where the output is a function of the current state and the current input. It consists of one or more shift registers and multiple XOR gates. The stream of information bits flows in to the shift register from one end and is shifted out at the other end. XOR gates are connected to some stages of the shift registers as well as to the current input to generate the output. There is no theoretical basis for the optimal location of the shift register stages to be connected to XOR gates. It is based on an empirical approach. The location of stages is determined by the interconnection function. The location of stages as well as the

number of memory elements determines the minimum Hamming distance. Minimum Hamming distance determines the maximal number of correctable bits. Interconnection functions for different rates and different number of memory elements and their minimum Hamming distances are available [52].

The operation of a convolutional encoder can be easily understood with the aid of a state diagram. Figure 2.2 represents the state diagram of the encoder shown in Figure 2.1. Figure 2.2 depicts state transitions and the corresponding encoded outputs. As there are two memory-

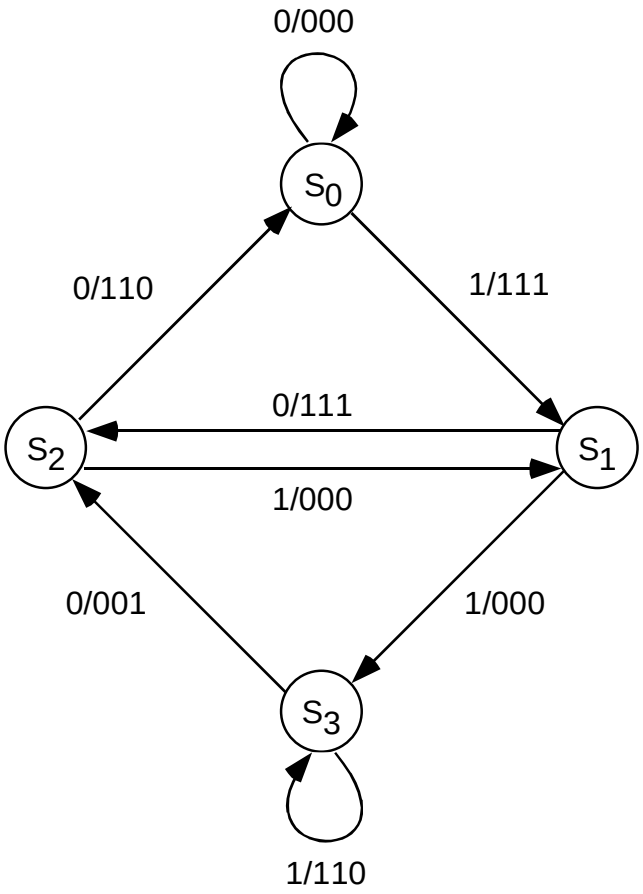

Figure 2.2 State diagram for encoder in Figure 2.1[52]

elements in the circuit, there are four possible states that the circuit can assume. These four states are represented as S_0 through S_3. Each state's information (i.e. the contents of flip-flops for the state) along with an input generates an encoded output code. For each state, there can be two

outgoing transitions; one corresponding to a ‘0’ input bit and the other corresponding to a ‘1’ input bit.

A trellis diagram is an extension of a state diagram that explicitly shows the passage of time. Figure 2.3 shows a trellis diagram for the encoder given in Figure 2.1. In the trellis diagram, nodes correspond to the states of the encoder. From an initial state (S_0) the trellis records the possible transitions to the next states for each possible input pattern. For the encoder in Figure 2.1, there are two encoded symbols corresponding to input bit ‘0’ and ‘1’. The Figure 2.3 shows the encoded symbol generated for each transition. At the stage *t=1* there are two states S_0 and S_1, and each state has two transitions corresponding to input bits ‘0’ and ‘1’. Hence the trellis grows up to the maximum number of states or nodes, which is decided by the number of memory elements in the encoder. After all the encoded symbols of the information bits are transmitted, the encoder is usually forced back into the initial state by applying a fixed input sequence called reset sequence. The fixed input sequence reduces the possible transitions. In this manner, the trellis shrinks until it reaches the initial state. The trellis diagram in Figure 2.3 is for an input length of five bits, in which the last two bits represent the reset sequence. It should be noted that, there is a unique path for every code word that begins and stops at the initial state.

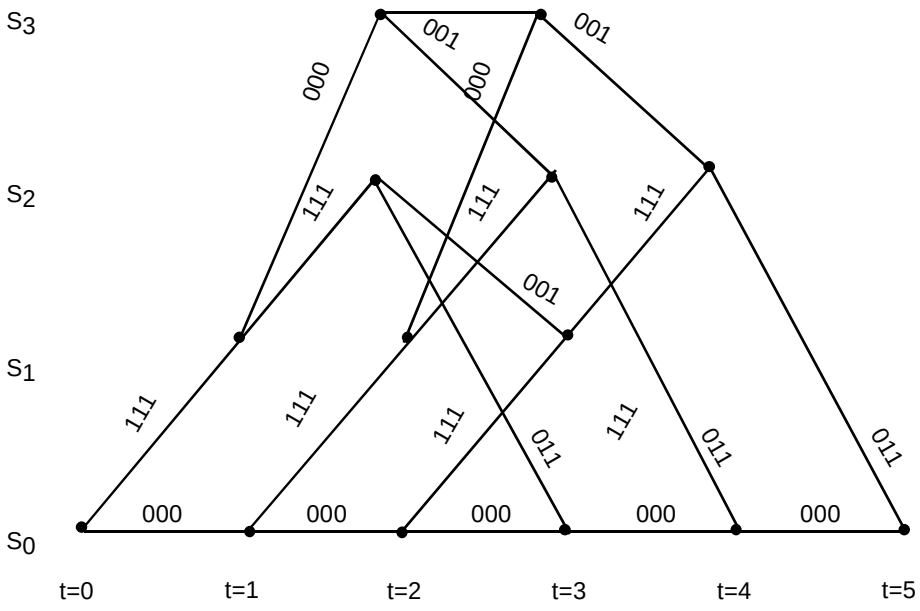

Figure 2.3 Trellis diagram for inputs of length three to the encoder in fig 2.1

2.3 Viterbi decoding algorithm

The Viterbi decoding algorithm is a decoding process for convolutional codes for a memory-less channel. Figure 2.4 depicts the normal flow of information over a noisy channel. For the purpose of error recovery, the encoder adds redundant information to the original information *i*, and the output *t* is transmitted through a channel. Input at receiver end (*r)* is the information with redundancy and possibly, noise. The receiver tries to extract the original information through a decoding algorithm and generates an estimate (*e*). A decoding algorithm that maximizes the probability *p(r|e)* is a **maximum likelihood (ML)** algorithm. An algorithm which maximizes the *p(e|r)* through the proper selection of the estimate (*e*) is called a **maximum a posteriori (MAP)** algorithm. The two algorithms have identical results when the source information *i* has a uniform distribution.

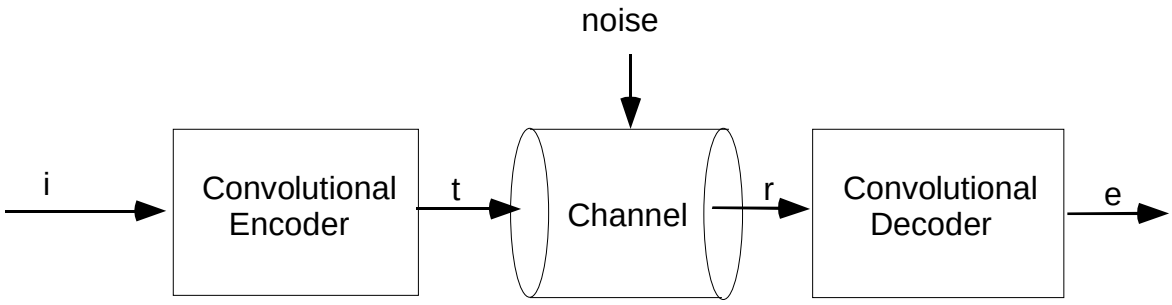

Figure 2.4 The convolutional decoding

Since the received signal is analog, it can be quantized into several levels. If the received signal is converted into two levels, either zero or one, it is called hard decision. If the input signal is quantized and processed for more than two levels, it is called soft decision. The soft decision captures more information in the input signal consequently performing better than the hard decision at the cost of a higher complexity. In this thesis, the ML algorithm with the hard decision has been employed.

The Viterbi algorithm based on the **ML** algorithm and the hard decision is illustrated in Figure 2.5. The trellis in the figure corresponds to the convolutional encoder given in Figure 2.1. The received code symbols are shown at the bottom of the trellis. The encoder encodes an input sequence (11010100) and generates the code word (111,000,001,001,111,001,111,110). This code word is transmitted over a noisy channel, and (101,100,001,011,111,101,111,110) is received at the other end. As mentioned earlier, the length of the trellis is equal to the length of the input sequence, which consists of the information bits followed by the reset sequence. The reset sequence, "00", forces the trellis into the initial state, so that the traceback can be started at the initial state.

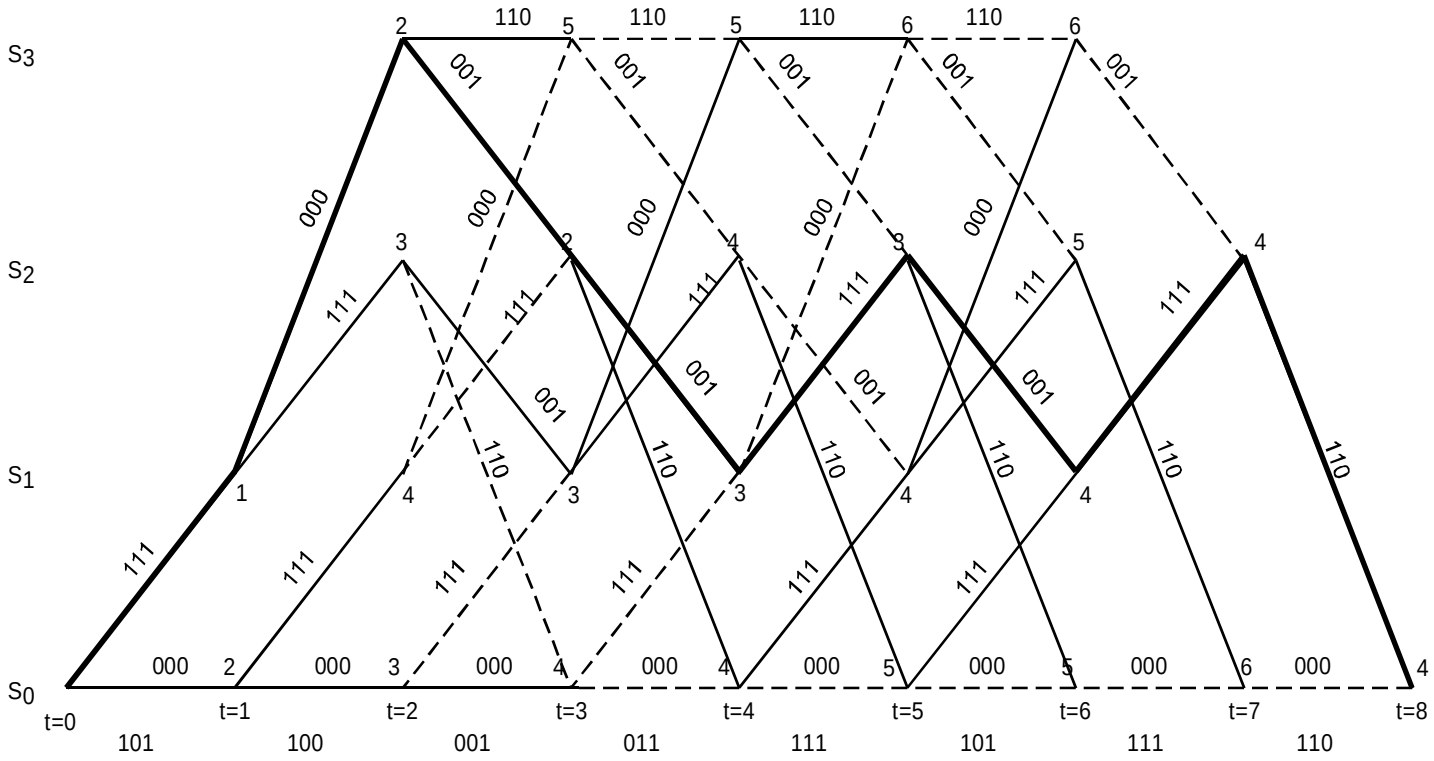

Figure 2.5 Hard-decision Viterbi decoding Example [52]

An **ML** path is found with the aid of a branch metric and a path metric. A branch metric is the Hamming distance between the estimate and the received code symbol. The branch metrics accumulated along a path form a path metric. A partial path metric at a state, often referred as

state metric, is the path metric for the path from the initial state to the given state. After the trellis grows to its maximal size, there are two incoming branches for each node. Between two branches, the branch with a smaller (in terms of Hamming distance) partial metric survives, and the other one is discarded. After surviving branches at all nodes in the trellis have been identified, there exists a unique path starting and ending at the same initial state in the trellis. The decoder generates an output sequence corresponding to the input sequence for this unique path. The procedure is explained below using the trellis diagram in Figure 2.5.

The path metric for state S_0 at time *t=0* is initialized to zero. At time *t=1* there is only one branch entering state S_0. This branch metric is the Hamming distance between the expected input "000" and the received input "101", which is two. The path metric of S_0 at time *t=1* is the sum of the old path metric of S_0 and the branch metric. Similarly, the path metric of S_1 at *t=1* is one. At *t=1* there is only one branch entering these nodes. The sole branch is the survivor branch. The same process repeats for *t=2.* At *t=3* there are two branches entering each node. For example, at state S_0, a branch with the partial path metric six (which is the sum of the path metric 3 of S_2 and the branch metric 3) enters to the state from S_2. The other branch with the partial path metric four also enters the state from S_0. Between the two branches, the branch from S_0 survives and the other one is discarded. Surviving branches are depicted in solid lines and discarded ones are in dotted lines in Figure 2.5.

Once the trellis is tagged with partial path metrics at each node, we perform a traceback to extract the decoded output sequence from the trellis. We start with state S_0 at time *t=8* and go backward in time. The sole survivor path leads to state S_2 at time *t=7*. From state S_2 at time *t=7,* we traceback to S_1 at time *t=6*. In this manner, a unique path shown in the bold line is identified. Note that each branch is associated with specific source input bit. For example, the branch from state S_2 at time *t=7* to node S_0 at time *t=8* corresponds to a bit '0' whose bit position is the seventh in the source input sequence. So while tracing back through the trellis, the decoded output sequence corresponding to these branches is generated.

2.4 Implementation of a Viterbi decoder

The major tasks in the Viterbi decoding process are as follows:

1. Quantization: Conversion of the analog inputs into digital.
2. Synchronization: Detection of the boundaries of frames and code symbols.
3. Branch metric computation.
4. State metric update: Update the state metric using the new branch metric.
5. Survivor path recording: Tag the surviving path at each node.
6. Output decision generation: Generation of the decoded output sequence based on the survivor path information.

Figure 2.6 shows the flow of the Viterbi decoding algorithm, which performs the above tasks in the specified order.

This section discusses the different parts of the Viterbi decoding process. Analog signals are quantized and converted into digital signals in the quantization block. The synchronization block detects the frame boundaries of code words and symbol boundaries. *We assume that a Viterbi decoder receives successive code symbols, in which the boundaries of the symbols and the frames have been identified.*

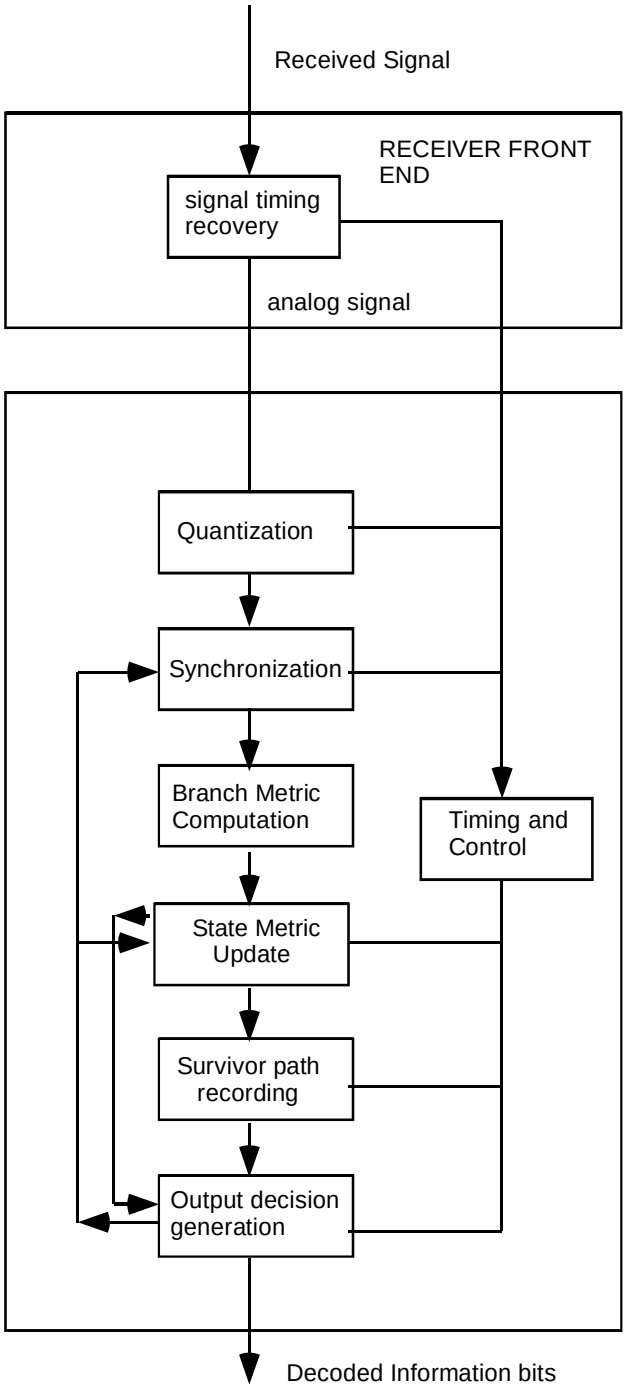

Figure 2.6 The flow in General Viterbi Decoder [52]

The branch metric computation block compares the received code symbol with the expected code symbol and counts the number of differing bits. An implementation of the block is shown in Figure 2.7.

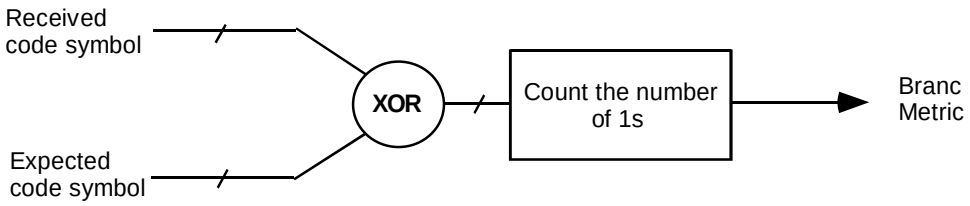

Figure 2.7 A branch metric computation block

The state metric update block selects the survivor path and updates the state metric. The trellis diagram for a rate 1/n convolutional encoder consists of butterfly structures. This structure contains a pair of origin and destination states, and four interconnecting branches as shown in Figure 2.8.

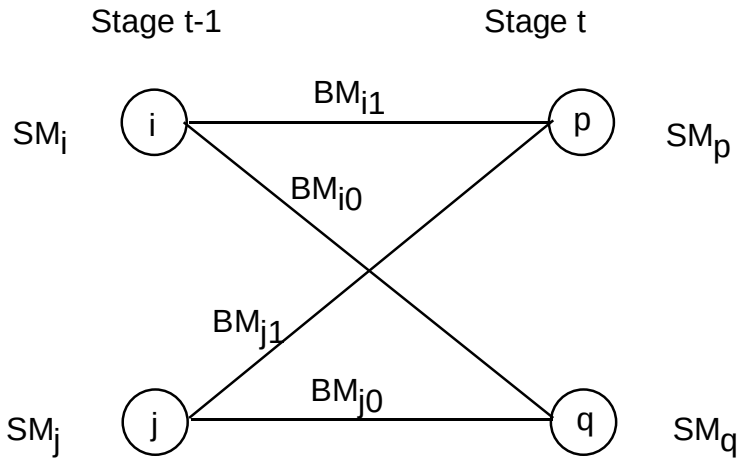

Figure 2.8 A butterfly structure for a convolutional encoder with rate 1/n

In the Figure 2.8, the upper (lower) branch from each state *i* or *j* is taken, when the corresponding source input bit is '1' ('0'). If the source input bit is '1' ('0'), the next state for both *i* or *j* is state *p (q)*. The following relations shown in figure are established for a (n,1,m) convolutional encoder [52].

Notatoin:

SM_x: state metric of a state x

BM_{xk}: branch metric from a state x under the source input k, where k ∈ {'0','1'}

For state i	For branch metric BM_{xx}	For state metric SM_x
$j= 2^{m-1}+i$	$BM_{i0}= n-BM_{i1}$	$SM_p= Min[(SM_i+BM_{i1}),(SM_j+BM_{j1})]$
$p = 2i+1$	$BM_{j0} = BM_{i1}$	$SM_q= Min[(SM_i+BM_{i0}),(SM_j+BM_{j0})]$
$q= 2i$	$BM_{j1} = BM_{i0}$	

Figure 2.9 The relationships of the states and branch metrics in a butterfly

It is important to note that state *p* is even and state *q* is odd. This implies that an odd (even) state is reached only if the source input bit is '0' ('1'). This property is utilized for the traceback, which is explained later. Another important point to be noted is that, it is possible to traceback from a state at a stage *t* to its previous state at the stage *t-1* provided the survivor branch of the state is the upper path or the lower path. If the survivor branch of an odd state *p* at stage *t* is the upper (lower) path, the previous state at stage *t-1* is state *i(j)*. Note that *i* is obtained as *(p-i)/2* and *j* is $2^{m-i}+i = 2^{m-i}+ (p-i)/2$. Similar results can be applied to an even state. In summary, if we record whether the survivor path is the upper path or the lower path, we can traceback from the final state to the initial state.

Each butterfly wing is usually implemented by a module called Add-Compare-Select (ACS) module. An ACS module for state *p* in Figure 2.8 is shown in Figure 2.10. The two adders compute the partial path metric of each branch, the comparator compares the two partial metrics, and the selector selects an appropriate branch. The new partial path metric updates the state metric of state *p*, and the survivor path-recording block records the survivor path.

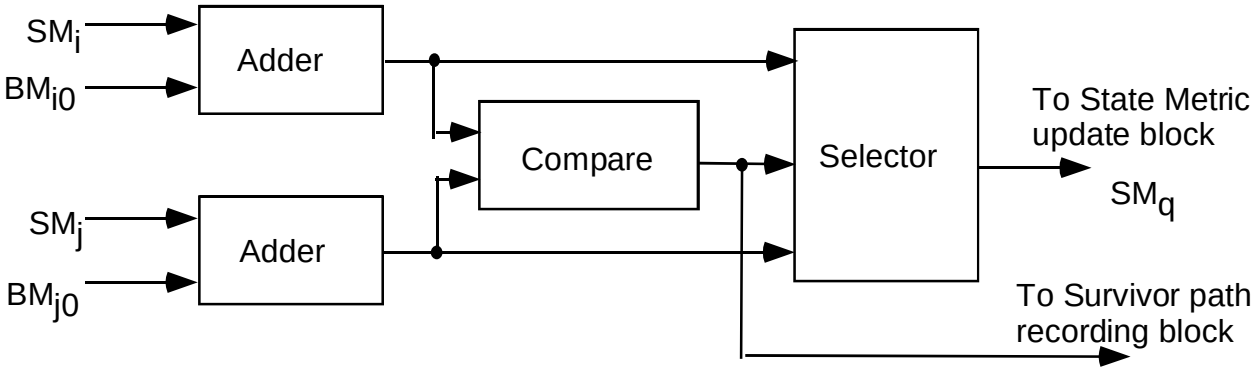

Figure 2.10 ACS (Add-Compare-Select) module

The number of necessary ACS module is equal to half the number of total states. Time sharing of some ACS modules is possible to save the hardware, but such sharing slows down the operation and dissipates more power. In this thesis we reckon replication of necessary ACS modules, which is more power efficient.

Two approaches are often used to record survivor branches, register-exchange and traceback [52]. The register-exchange approach assigns a register to each state. The register records the decoded output sequence along the path starting from the initial state to the final state, which is same as the initial state. Consider a trellis diagram shown in Figure 2.11. The register of state S_1 at $t=3$ contains 101. Note that the trellis follows along the bold path, and the decoded output sequence is 101. This approach eliminates the need to traceback, since the register of the final state contains the decoded output sequence. Hence, the approach may offer a high-speed operation, but it is not power efficient due to the need to copy all the registers in a stage to the next stage. We have investigated on the power efficiency of this approach.

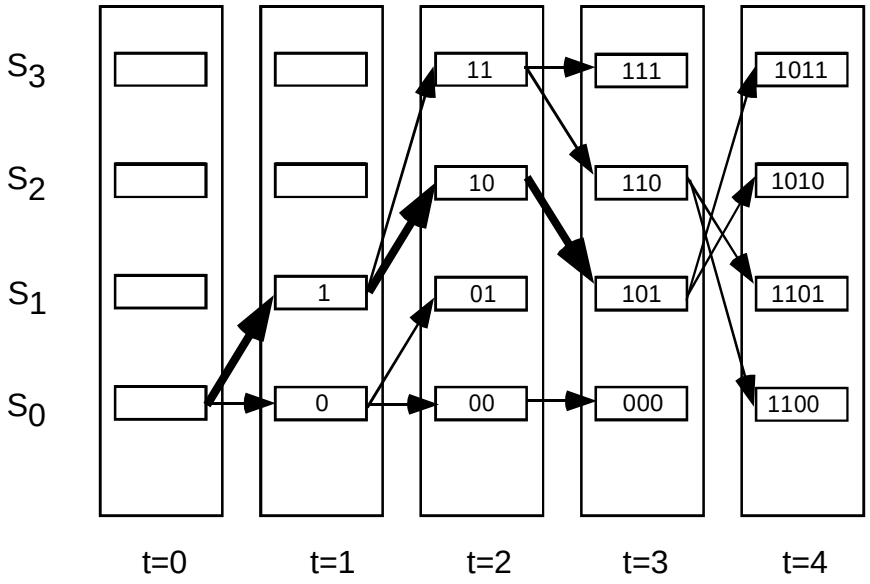

Figure 2.11 Register-exchange information generation method

The other approach called traceback records the survivor branch of each state. As explained earlier, it is possible to traceback the survivor path provided the survivor branch of each state is known. While following the survivor path, the decoded output bit is ‘0’ (‘1’) whenever it encounters an even (odd) state. A flip-flop is assigned to each state to store the survivor branch and the flip-flop records ‘1’ (‘0’) if the survivor branch is the upper (lower) path. Concatenation of decoded output bits in reverse order of time forms the decoded output sequence.

It is possible to form registers by collecting the flip-flops in the vertical direction or in the horizontal direction as shown in Figure 2.12. When a register is formed in vertical direction, it is referred to as “selective update” in this thesis. When a register is formed in horizontal direction, it is referred to as “shift update”.

In selective update, the survivor path information is filled from the left register to the right register as the time progresses. In contrast, survivor path information is applied to the least significant bits of all the registers in “shift update”. Then all the registers perform a shift left operation. Hence, each register in the shift update method fills in survivor path information from

the least significant bit toward the most significant bit. Figure 2.13 shows a selective update in the traceback approach.

The shift update is more complicated than the selective update. The shift update is described in [52], and the selective update is proposed by us to improve the shift update. In chapter 4, we show that the selective update is more efficient in power dissipation and requires less area than the shift update. Due to the need to traceback, the traceback approach is slower than the register-exchange approach.

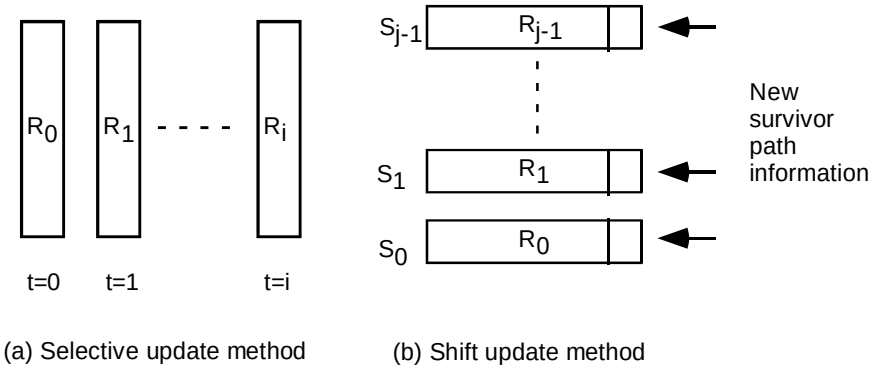

Figure 2.12 Two options for forming registers

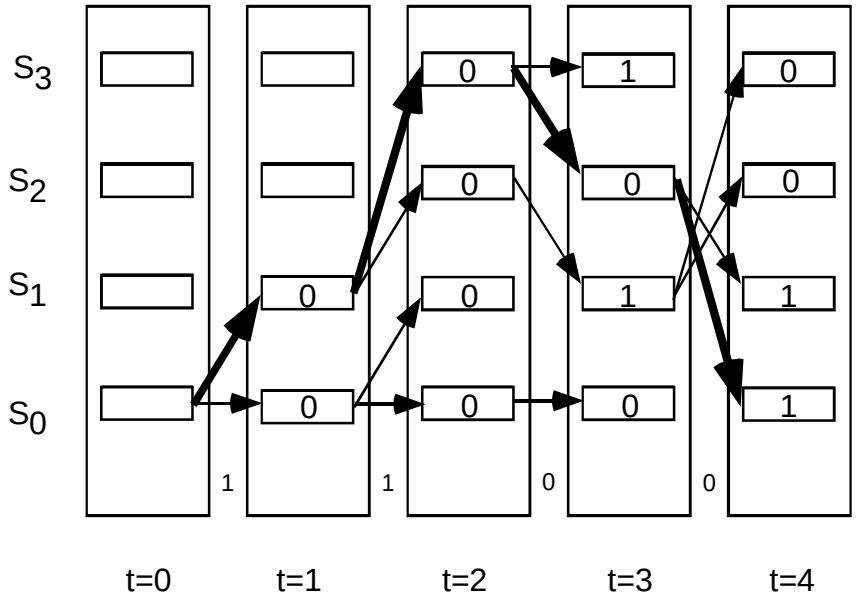

Figure 2.13 Selective update in the traceback approach

2.5 Low-power design techniques

Low-power design can be performed at the architecture level, the gate level and the switch level. At the architecture level, different architectures, such as parallel and pipelined architecture or transformations may be considered for low power dissipation. The reduction of switching activity at nodes of a circuit, which directly affects the power dissipation, is the major focus at the gate level. Parameter adjustment and transistor sizing can be applied at the switch level. Different transistor configurations in forming cells can also be applied at this level. For a standard cell approach, there is no control over internal circuitry (i.e. transistors) of a cell. Therefore low power design can be considered only at the architecture and the gate levels. In this thesis, we consider low power design techniques mainly at the gate level.

Full CMOS circuits dissipate most of its power during switching, called dynamic power dissipation. Dynamic power dissipation is responsible for usually over 85 to 90 percent of the total power dissipation [54]. Dynamic power dissipation of a full CMOS circuit is formulated as follows,

$$P = \alpha C_L V^2 f$$

here α is the switching activity, C_L is the parasitic capacitance, V is the supply voltage, and f is the clock frequency. Every time a gate changes its state (switches) it charges or discharges the parasitic capacitance. The amount of energy dissipated depends on the capacitance C_L and the source voltage V from which the capacitor is charged. The rate at which a gate switches in a circuit depends on the clock frequency. The switching activity α corresponds to the average percentage of gates, which switch for each clock cycle.

Given the formula for power dissipation, we can manipulate some parameters to reduce the power dissipation. The supply voltage and the clock frequency are determined at the system level, and they are beyond control of a circuit designer. The switching activity α and the parasitic capacitance C_L are affected by the circuit design. For the standard cell approach, it is possible to instruct some design tools, such as a place-and-router, to reduce the overall interconnect length and hence to reduce the parasitic capacitance C_L. However, a major reduction in power

dissipation can be achieved by reducing the switching activity α on which a designer has more control. A careful description of the circuit in a high-level hardware description language (such as VHDL) can yield a circuit with a lower switching activity. We summarize below general techniques that can be employed for low power dissipation under the standard cell design approach:

- *Elimination of redundant logic.*

A redundant logic, which does not contribute to the function of the circuit, dissipates power and should be eliminated. If a logic synthesis tool is used to synthesize a gate level circuit, elimination of redundant logic is performed during the logic synthesis.

- *Clock Gating.*

The clock gating is one of the most powerful low power design techniques. Some blocks of a circuit are used only during a certain period of time. The clock of the blocks can be disabled to eliminate unnecessary switching when the blocks are not in use. Figure 2.14 shows a clock gating method to disable a functional unit.

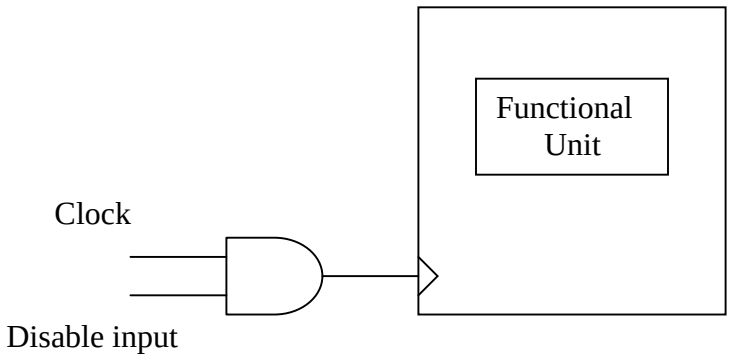

Figure 2.14 Clock gating scheme

Clock gating requires additional logic to generate enable signals. The additional logic dissipates power and may incur performance degradation. Therefore the clock gating should be employed only if the benefits are greater than the cost.

- *Toggle filtering.*

If signals arrive at the inputs of a combinational block at different times, the block may go through several intermediate transitions before it settles down. The intermediate transitions and consequently the dynamic power dissipation can be reduced, by blocking early signals until all input signals arrive.

Employment of appropriate low-power design techniques depends on the characteristic of the circuit and the operating environment. In our research, we consider the above three techniques for low-power design of a Viterbi decoder.

2.6 Review of previous work on low-power Viterbi decoder design

Viterbi algorithm and implementation of efficient Viterbi decoders have been investigated intensively for the past three decades [1]-[55]. Among them, we review three most relevant works for low-power Viterbi decoder designs.

Seki, Kubota, Mizoguchi and Kato [43] suggested a scarce state transition (SST) scheme to reduce the switching activity of a Viterbi decoder. The input is pre-decoded by a simple and hence, a power efficient decoder. The pre-decoded sequence, which is not optimal under a noisy channel, is reprocessed by a Viterbi decoder to improve performance. The authors showed that the pre-decoded sequence reduces the switching activity of the Viterbi decoder thereby reducing power dissipation. Lang, Chi and Cheng [33] applied the SST scheme to a turbo decoder and reduced power dissipation.

Kang and Wilson [27] suggested application of existing low-power design methodologies at different levels. At the architectural level, they suggested partition of major blocks and memory modules to reduce the power dissipation. They considered Grey coding for memory addressing, which incurs less switching compared to binary coding. In addition, they employed disabling of signals and of clock to reduce spurious transitions.

Garrett and Stan [23] suggested a low-power architecture of the soft-output Viterbi decoder for turbo codes. They proposed an orthogonal access memory structure, which enables parallel access of sequentially received data. Use of such a memory structure reduces the switching activity for read and write of survivor path information

Oh and Hwang [36] proposed a traceback scheme to cut down the switching activities incurred while tracing back. Their scheme is designed for a decoder, where the traceback starts before the end of the code word. The key idea is to reuse the information from the previous trace to shorten the traceback.

All the above works aim to reduce the switching activities of Viterbi decoders, which is an effective scheme for power reduction. Our methods investigated in this thesis also rely on the reduction of switching activities.

Chapter 3
Proposed Design

3.1 Overview

In this chapter, we describe the implementation of the Viterbi algorithm and propose a low-power design of Viterbi decoders. We also describe top-down design approach employed to implement Viterbi decoders with and without low-power consideration.

3.2 General Viterbi decoder

The major building blocks of a Viterbi decoder are shown in Figure 3.1. There are eight conceptual blocks of a Viterbi decoder, and the role of each block is described in detail below.

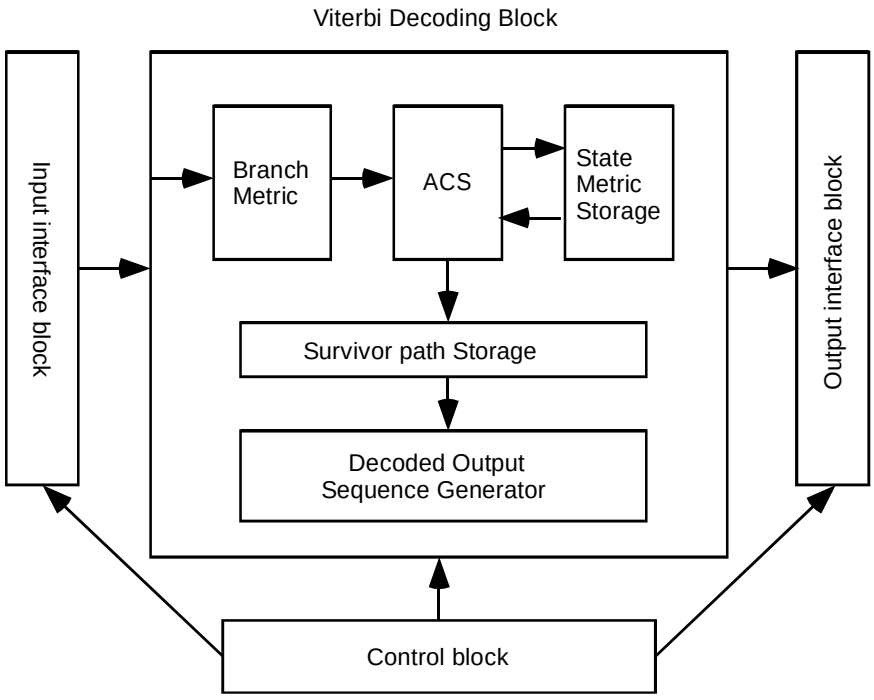

Figure 3.1 The block diagram of a general Viterbi Decoder

Input and output blocks: Input and output blocks provide the interface with the external components. In the case of radio communications, input received by the decoding block is usually serial, while the decoding block actually needs a parallel input. Serial to parallel conversion and vice versa are carried out by the input and output blocks.

Branch Metric: This block calculates the branch metric of each stage in the trellis. It also calculates the hamming distances (i.e. branch metric) between the received symbol and expected symbol.

State Metric Storage: The block stores the partial path metric of each state at the current stage.

ACS: The Add-Compare-Select block receives two branch metrics and the state metrics. An ACS module adds each incoming branch metric of the state to the corresponding state metric and compares the two results to select a smaller one. The state metric of the state is updated with the selected value, and the survivor path information is recorded in the survivor path storage module.

Survivor Path Storage: The survivor path storage block is necessary only for the traceback approach (explained later). The block records the survivor path of each state selected by the ACS module. It requires one bit of memory per state per stage to indicate whether the survivor path is the upper one or the lower one.

Output Generator: This block generates the decoded output sequence. In the traceback approach, the block incorporates combinational logic, which traces back along the survivor path and latches the path (equivalently the decoded output sequence) to a register.

3.3 Viterbi encoder

The size of a Viterbi decoder depends on the parameters L, M and m. Here, L is the number of code symbols in a code word, M is the total number of memory elements in the corresponding encoder and m is the maximal memory order, which is equal to the length of the longest input shift register in the encoder. The constraint of an encoder is given as (m+1).

Since the goal of our research is to design a low-power Viterbi decoder, we considered a small size Viterbi decoder for demonstrating the concept. The convolutional encoder, which corresponds to our Viterbi decoder considered in this thesis is shown in Figure 3.2.

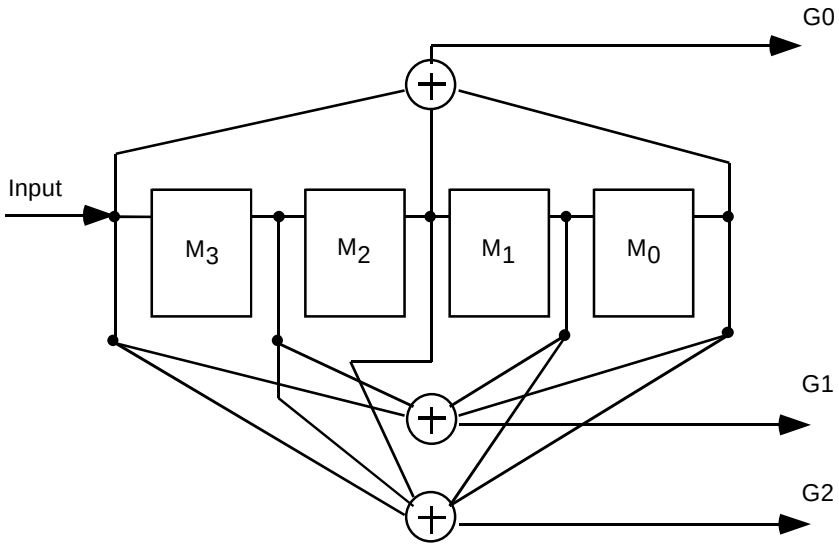

Figure 3.2 A convolutional encoder for the proposed Viterbi decoder

The encoder in Figure 3.2 is called a (3,1,4) encoder, meaning it has three output lines, one input line, and four memory elements in the shift register. As there is only one shift register in the encoder the total memory M and the maximal memory order m are four. We consider the length L of a code word to be 20, in which the last four code symbols reset the encoder into the

initial state of all ‘0’s. *All the Viterbi decoders described hereafter employs the encoder in Figure 3.2 with the code word length 20.*

The nodes in a trellis diagram correspond to the states of the convolutional encoder. Memory elements are labeled as $M_0 \ldots M_3$ in Figure 3.2. States in a trellis diagram are labeled in the binary format “$M_0M_1M_2M_3$”, in this thesis. For example, if $M_3M_2M_1M_0 = 1100$ for the encoder, the corresponding state is 0011, which is state three in the trellis diagram. The state number ranges from 0 to 15.

3.4 Implementation of a Viterbi decoder

The necessary blocks for the implementation of a Viterbi decoder often deviate from the block diagram in the Figure 3.1 depending on the design choices. Our Viterbi decoder consists of five blocks as shown in Figure 3.3. The survivor path storage block is necessary only for the traceback approach. We describe the implementation of each block below.

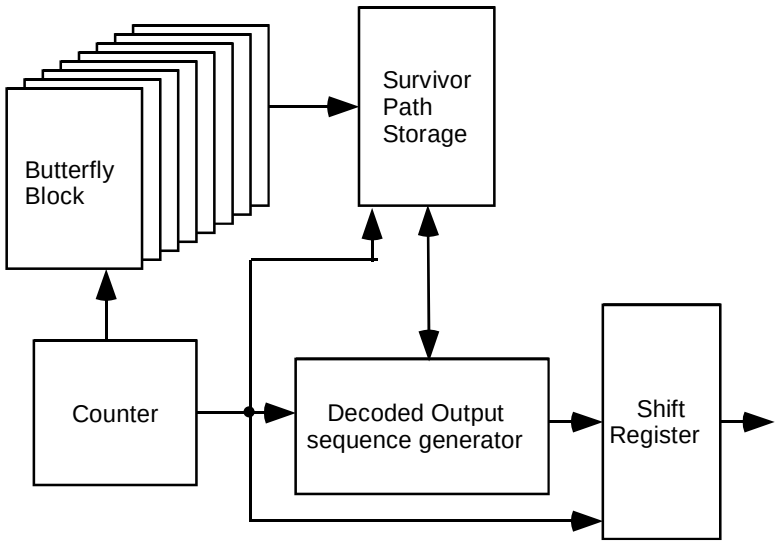

Figure 3.3 Block diagram of the proposed Viterbi decoder

3.4.1 Butterfly block

The Butterfly block is an integration of the branch metric block, the ACS block and the state metric storage block in Figure 3.1. There are eight butterflies in our Viterbi decoder, where each butterfly corresponds to two states as shown in Figure3.4.

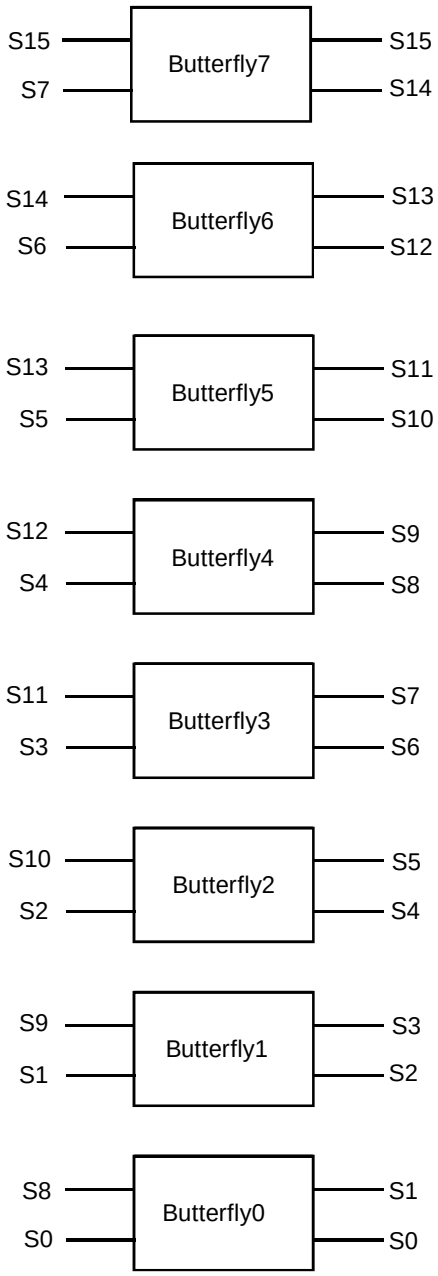

Figure 3.4 Butterfly blocks

There are two possible implementations of the butterfly block. One implementation stipulates the use of a single butterfly module with a separate fast clock. This butterfly is time-shared for all the states. The time sharing approach saves the area but dissipates more power due to a higher switching activity and the extra control circuit. In addition, two separate clock domains pose a problem in the scan design. The other approach is to employ eight butterfly modules where each module is dedicated to the corresponding states. In order to reduce the power dissipation at the cost of higher area, we adopted the latter approach, which is to stack eight butterfly modules.

We illustrate the implementation of the bottom butterfly module in Figure 3.4, where the input states are S_8 and S_0, and the output states are S_1 and S_0. The butterfly and its expected symbols are shown in Figure 3.5

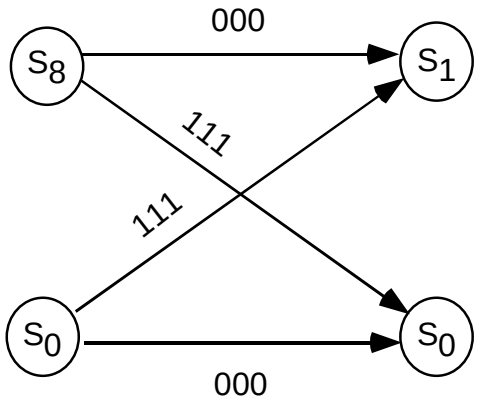

Figure 3.5 Implementation for bottom butterfly

The implementation of the two wings of a butterfly module is identical except their expected symbols. We show implementation of only the upper wing. This wing receives the partial path metrics from S_8 and S_0, adds each path metric with the branch metrics of the one of two top branches, and selects a branch with a smaller path metric. The new path metric replaces the partial path metric of S_1. The block diagram of the wing is shown in Figure 3.6.

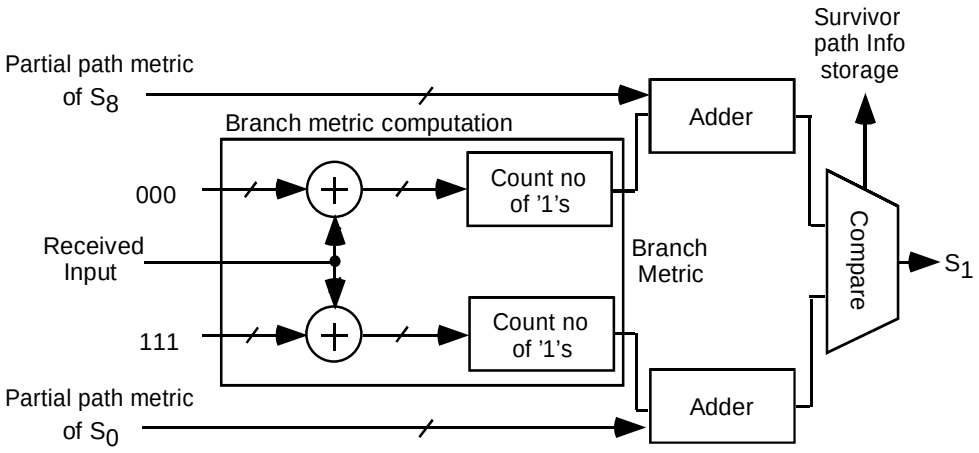

Figure 3.6 The block diagram for a wing of butterfly

Branch metric to be added with the path metric for state S_0 (S_8) is the Hamming distance between the expected code symbol "111" ("000") and the received input. The received input is XORed with the expected code symbol. The "Count no. of '1's" block generates a binary number equal to the number of '1's on its input. The branch metric is added to the partial path metric to calculate the new path metric. Two path metrics, the upper one and the lower one, are compared to select the survivor path, and the resultant metric of the selected path updates the path metric of state S_0 (S_8). The lower wing is identical to the upper wing except that the expected values differ.

3.4.2 Survivor path storage module

This block is necessary only for the traceback approach. Except for the head and the tail part of the trellis diagram there are two incoming branches for each state. Among the two incoming branches it is necessary to indicate which branch, either upper or lower one, survives. So only one bit of information is necessary for each state to flag the survivor path.

Only a subset of states (which are the last four stages in our case) receives two incoming branches for the tail part. In the head part, states receive only one incoming branch, which is always a lower branch.

If the above special cases are utilized for the head and tail parts, it is possible to reduce memory size for storage of survival paths. However, it increases the complexity of the design,

which may result in more hardware, therefore we decided to allocate the same size of registers for all stages as shown in Figure 3.7. In the figure, each register has 16 bits corresponding to 16 states in the trellis and each bit stores the survivor path of the corresponding state.

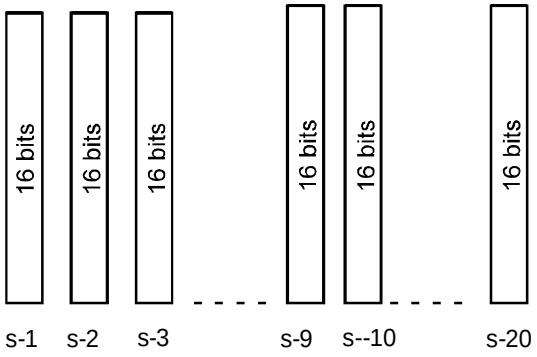

Figure 3.7 A register bank for the survivor path storage

The structure of the proposed survivor path storage module is shown in Figure 3.8. The five-bit counter keeps track of the current stage. The survivor path information of the 16 states, which is generated by the butterfly block, is passed to the register of the stage through the demultiplexer.

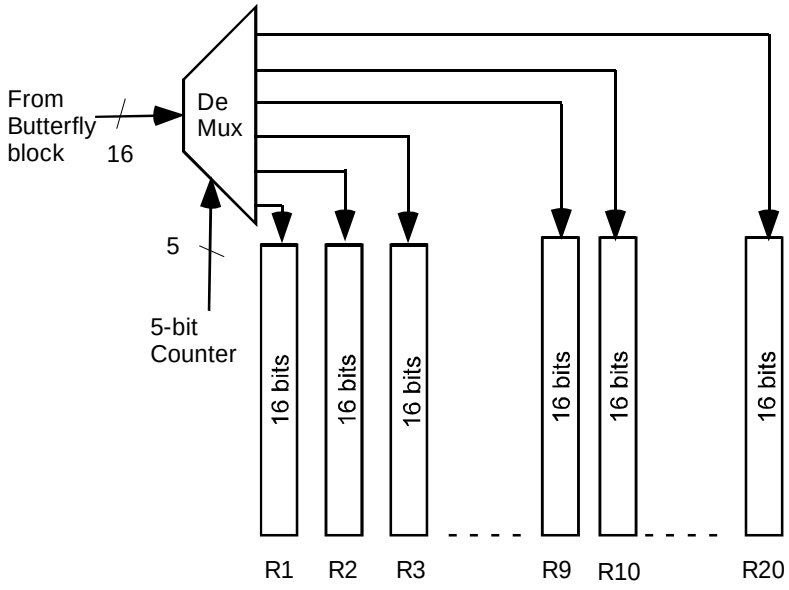

Figure 3.8 The structure of the proposed survivor path storage module

3.4.3 Decoded output sequence generation block

For the register-exchange approach, this block is a bank of registers, which holds the decoded output sequence. The decoded output sequence is the content of the register associated with state S_0. For the traceback approach, this block is a combinational logic, which performs traceback operation. The operation is explained in the following. In order to understand the traceback mechanism, let us consider the relationship between the states of a butterfly block of our Viterbi decoder. The states of a butterfly are as shown in Figure 3.4. Numbers inside the oval are the binary representation of the state. The label of an edge indicates the input applied to the encoder under that state.

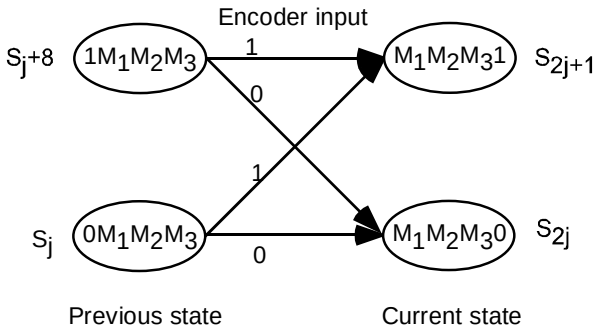

Figure 3.9 Relationship of states in a butterfly

As can be seen from the figure, if the current state is an odd state, such as S_1,S_3,S_5 and S_7 the input to the encoder must have been '1'. If the current state is even, the input must have been '0'. Suppose that we traceback the survivor path from the final state (S_0) to the initial state (S_0 again), the decoded output sequence is readily derived from the state numbers on the survivor path. From the current state, the previous state can also be readily obtained using the survivor path information. Suppose that the current state is $M_1M_2M_3$-, where '-' is either bit '0' or '1', the previous state is $1M_1M_2M_3$ if the survivor path is the upper one and is $0M_1M_2M_3$ if the survivor path is the lower one.

Based on the above two observations, we described the traceback operation in VHDL, so that a combinational logic block was inferred. The input of the combinational logic block is from

the survivor path storage block and the output is the decoded output sequence. The decoded output sequence is loaded into a parallel shift register, which outputs one bit of the decoded sequence on every clock cycle.

It is important to note that the traceback block can be activated only at the end of each coded word and deactivated for the rest of the period. This fact is utilized to reduce switching activities inside the module and subsequntly the power dissipation.

3.5 Proposed low-power design for traceback approach

Given the background about the implementation our Viterbi decoder, we propose low-power design of the Viterbi decoder for traceback approach. Two blocks, survivor path storage block and the decoded output sequence generator, are considered for low-power design. We also discuss a couple of design alternatives in the view of power dissipation. We do not consider redundancy elimination specifically, since it is performed in the optimization process during the logic synthesis.

3.5.1 Survivor path storage

The survivor path storage block holds the information on survivor paths. When the *i*th code symbol is received, the survivor path information is obtained and stored in the *i*th register. At this moment all other registers hold their contents, and hence their clocks can be gated to save power. Figure 3.10 shows a clock-gating scheme for this block.

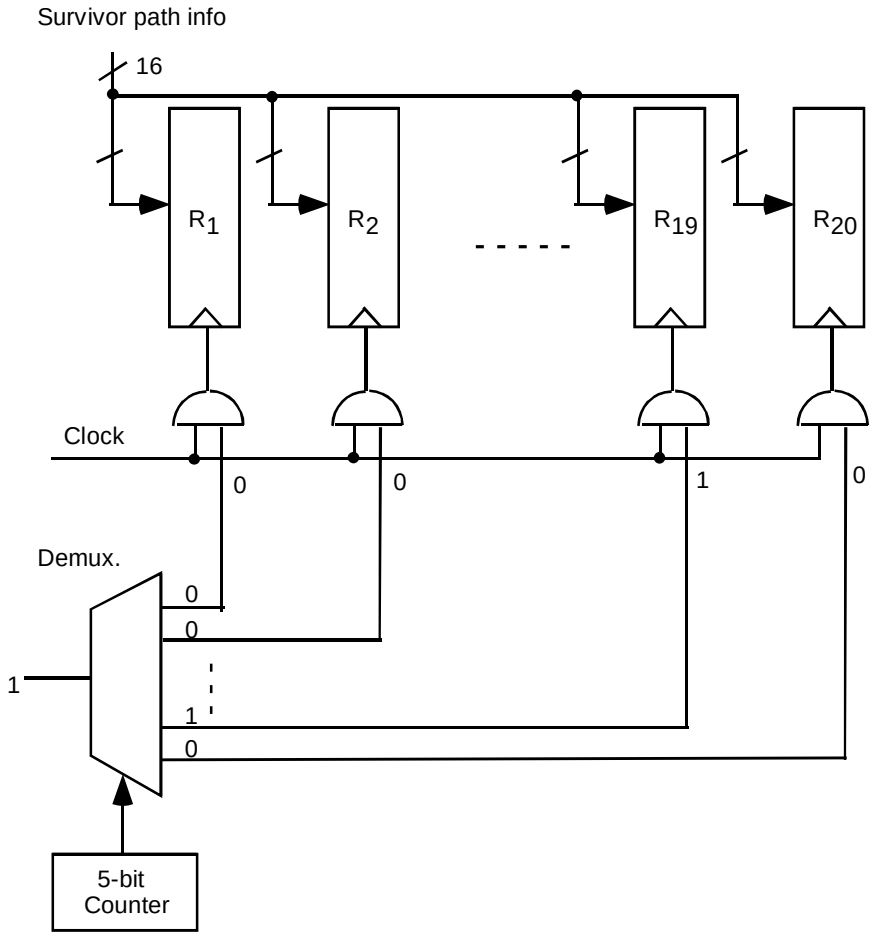

Figure 3.10 Clock gating in the Survivor path storage module

3.5.2 Traceback module

The traceback module traces back the survivor path after the entire code word has been received and generates the decoded output sequence. It is a combinational block and is active during only one clock cycle as shown below.

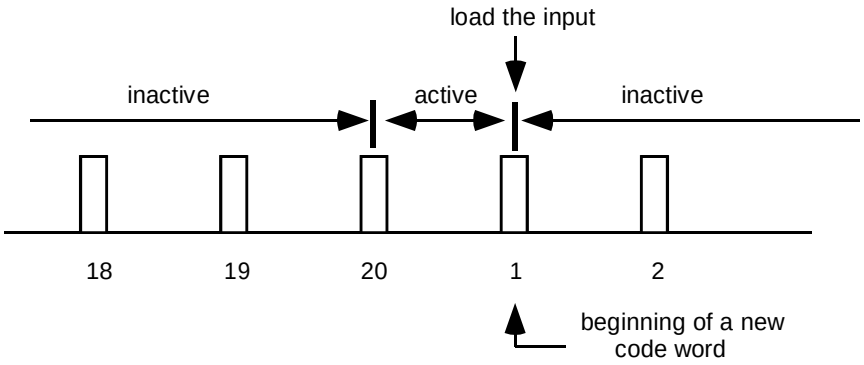

Figure 3.11 Activation of the traceback module

A block diagram of the module is shown in Figure 3.12. The input of the module is from the 20 registers containing the survivor path information. The output of the module, the decoded output sequence, is loaded into a shifter register at the first clock. The shift register shifts out the decoded output sequence one bit at a time during the following time frame.

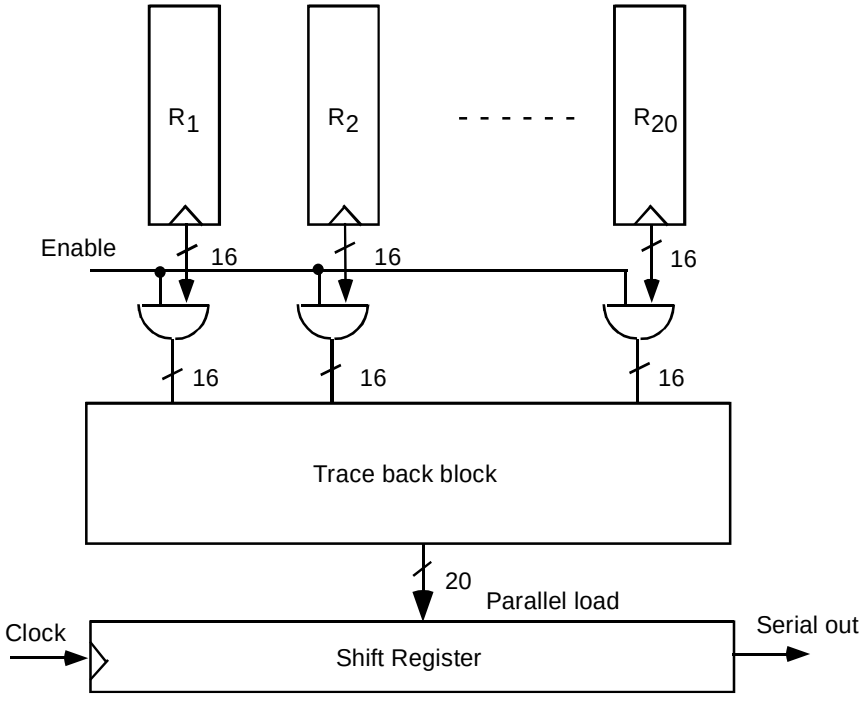

Figure 3.12 A block diagram of the traceback module

3.5.3 Traceback versus register-exchange approaches in power efficiency.

In the traceback approach, each register storing the survivor path information updates its content only once (when it receives the new survivor path information) during the entire period of a code word. In contrast, all the registers in the register-exchange approach update their contents for each code symbol. Hence, the switching activity of the registers in a traceback approach is much lower than that for the registers in a register-exchange approach. Hence, the registers in traceback approach would dissipate less power. Our experimental results presented in Chapter 4 confirm the speculation.

As explained earlier the registers and the traceback module are active only one clock period during the entire period of a code word in the traceback approach. So low power design techniques can be applied readily to the registers and the traceback module as proposed in this chapter. However, the same low-power techniques can not be applied to the register-exchange approach. Hence, the traceback approach is more desirable for applications in which power dissipation is critical.

3.5.4 Shift register versus multiplexer in power efficiency.

The decoded output sequence is loaded into a shift register in parallel and is shifted out serially, one bit at a time. The shift operation incurs high switching activities and hence may not be efficient in power. Instead of the shift register, a multiplexer can be used to perform the shift operation as shown in Figure 3.13.

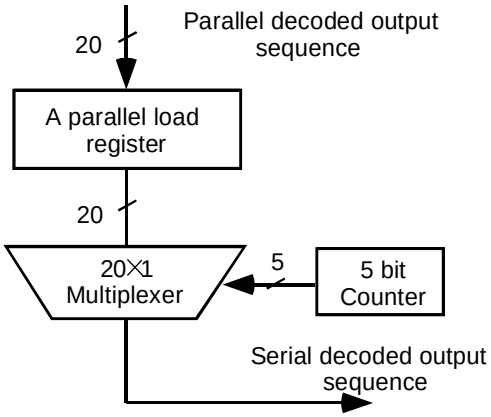

Figure 3.13 Multiplexer approach for shifting

A parallel load register loads the decoded output sequence. A multiplexer and a counter perform the shift operation to generate one bit at a time. The multiplexer based module eliminates the shift operation at the cost of more complex hardware. Due to the increased hardware complexity, the overall power saving for the multiplexer based approach is uncertain and is subject of the proposed research.

3.6 Design for testability for the proposed low-power design

Testability is an important issue for any design. In a scan design, scan cells replace memory elements, and the scan cells are put into a scan chain to form a shift register. The scan design enables one to access internal memory elements through a scan operation. In a full scan design all memory elements are put into a scan chain, where a subset of memory elements are put into the scan chain for a partial scan design. The two scan designs are a trade-off between testability and area overhead (and possibly performance). In this thesis, we considered a full scan design, which offers a better testability.

As discussed earlier, we employ a clock gating technique to deactivate modules when they are not in use for low power dissipation. An enable signal controls the clock gating as

shown in the Figure 3.10. However, all the clocks should be active to allow the scan operation during testing. Therefore, the addition of “Test mode” signal is necessary to override the enable signal during testing as shown in Figure 3.14. Whenever the circuit is put into the test mode the “Test mode” signal goes high and overrides the “Enable” signal.

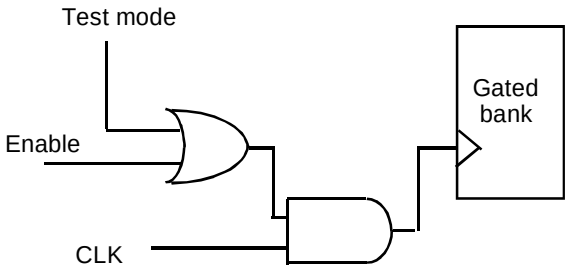

Figure 3.14 Gating circuit with overriding signal

3.7 Overall design flow.

The chart in the Figure 3.15 describes a standard cell based design flow, followed in this thesis. We describe major steps briefly in the following.

A design starts with the development of a behavioral VHDL description of the desired design. Synopsys simulation tools are used to verify the functionality as well as synthesizability of the design at the behavioral level. Design constraints, if any, and the operating conditions are also specified. In the logic synthesis process, RTL registers are specified for gating the clock. Once the clock gating has been finished and the design has been synthesized, Synopsys test compiler tools are used to insert a scan design. In this process all the sequential cells in the gate level design are replaced by scan cells in the library. Then a scan chain is formed. The scan chain can be formed automatically or can be specified in a specific order. Once a chain is assembled, the gate level design is represented in VHDL, and a functional simulation verifies its functionality. Then the design is saved in the Verilog format and is exported to the Cadence environment. An autolayout view is created for the gate-level design, which uses the abstract views of the standard cells.

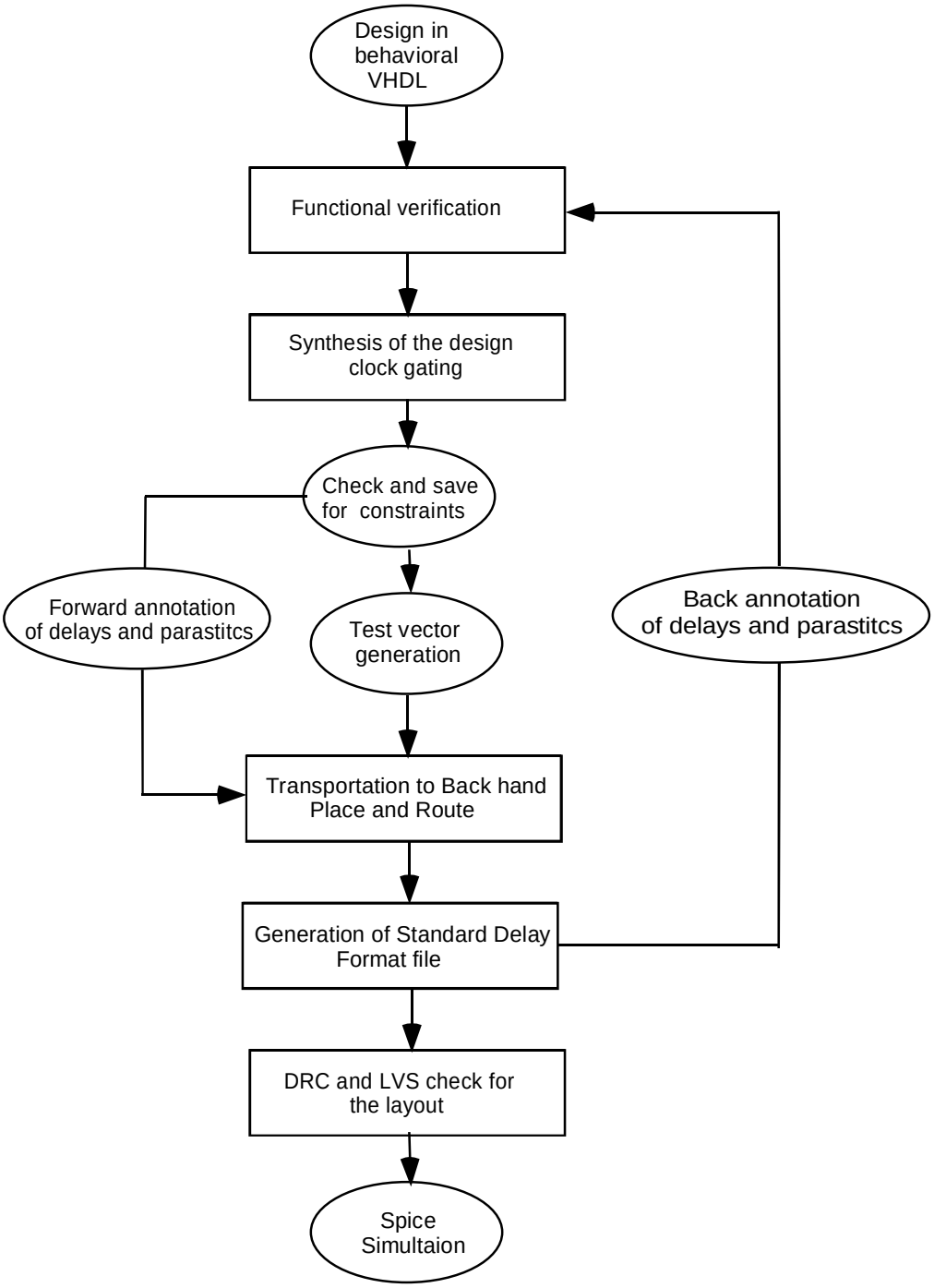

Figure 3.15 The design flow

After that the design is floorplanned and I/O pads are placed at the input and output ports. Then the design is placed and routed. The final layout is checked for DRC (Design Rule Check) and

LVS (Layout Versus Schematic). From the final layout timing and parasitic information is extracted and saved in SDF (Standard Delay Format) and SPEF (Standard Parasitic Exchange Format). These files are back annotated in the gate level design and the circuit is simulated again with the delay information. If the delays and constraints are not met, the gate level design is reoptimized and the process repeats. It is possible that from the gate level design the delay and constraint information is saved in SDF format and is forward annotated to the Floorplanner in the backhand tool. So the Floorplanner tries to adjust the layout to meet those constraints. Spice simulations may be necessary for the final design for more accurate analysis.

Chapter 4
Experimental results

4.1 Environment

In this chapter, we describe the environment for our experiments. Later we present experimental results for original and low-power Viterbi decoders. The technology used in our experiments is CMOS 0.25 μm with supply voltage 1.8V. The design kit used for our design supports six metal layers. Among several variations of Viterbi decoder and low-power design alternatives, our goal is to choose a design that would dissipate the least power. Therefore it is necessary to measure power dissipation of different implementations and compare them with each other.

Power dissipation can be measured at the gate level or at the switch level (extracted from the layout). The power measured at the switch level using a SPICE simulator is more accurate, but it takes longer processing time. The power measurement at the gate level is less accurate, but is much faster. Since our goal is to obtain a power efficient Viterbi decoder, the absolute accuracy of the measurement is not critical. Instead, the relative accuracy between two design alternatives suffices our needs. Hence, we opted to measure the power at the gate level. The measurement at the gate level has another advantage, we can choose a proper design at the gate level, which saves us from time consuming physical design process.

The design flow for power measurement at the gate level is shown in Figure 4.1. A logic synthesis tool of Synopsys, called Design Compiler, was used to obtain a gate level circuit from a RT level description. Design Compiler also provides an option to measure power dissipation at the RT level.

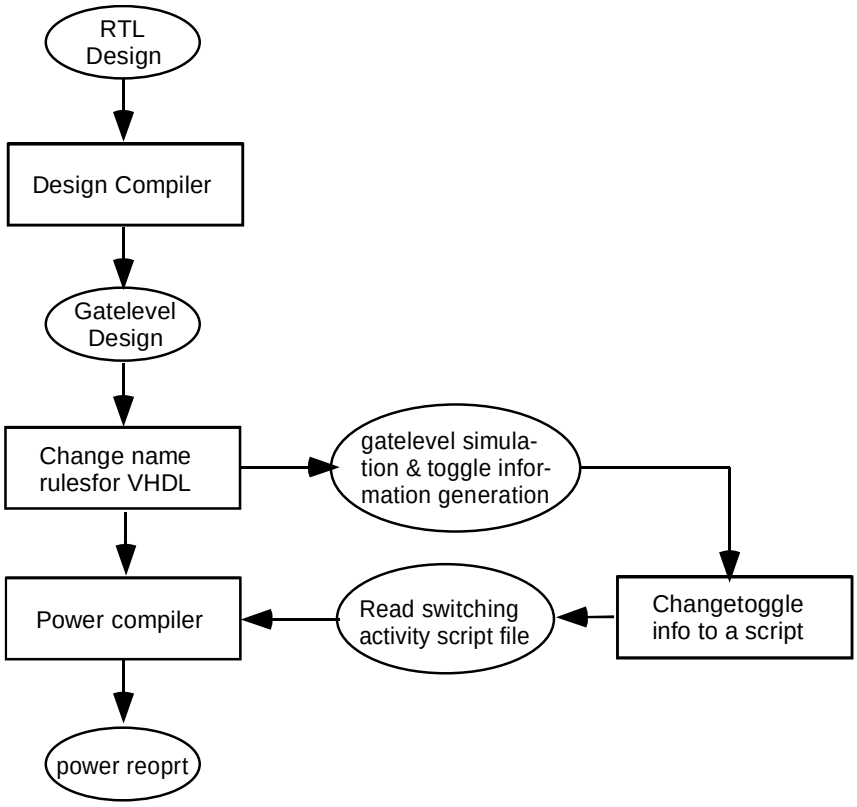

Figure 4.1 Methodology for Gate-level simulation approach. [Synopsys manual]

The power dissipation is the sum of the static and the dynamic power dissipation. The Design Compiler refers to the characterization of the technology library to estimate the static power dissipation. It computes the dynamic power dissipation using the formula $P = \alpha C_L V^2 f$, where α is the switching activity, C_L is the parasitic capacitance, V is the supply voltage, and f is the clock frequency. The capacitance and the supply voltage depend on the processing technology and are defined in the technology library.

There are two major approaches called probabilistic approach and simulation based approach, to measure the switching activity α. The probabilistic approach is faster, but incurs a larger error. As it turns out that the amount of error is unacceptable for our purpose, and we opted the simulation-based approach, where approximately 10,000 random input patterns were applied to

each design. The switching activity of each node was captured and annotated in the gate level design and then power dissipation was estimated.

The flow chart in Figure 4.1 shows the power measurement process in Synopsys environment. In the figure, the Design compiler of Synopsys reads an RTL design and generates a technology-dependent gatelevel design. After changing the naming rules for VHDL in the gate level design, the design can be sent to the VSS simulator. The VSS simulator applies test patterns (10,000 random patterns for our procedure) and performs logic simulation for the gate level design. The gate level simulator generates a toggle file containing the switching activitiy information at all nodes in the gate level design. Such a toggle file can be converted to a script file, which can annotate the switching acitivy into the acutal gate level design. The Design Compiler reads script file and annotates the switching acitvity. Power compiler performs a power analysis in the design using the annotated switching acitivity and generates accurate power dissipation resulsts.

4.2 Power dissipation in Viterbi decoders

To calibrate the efficiency of low power design alternatives, it is necessary to measure the power dissipation of original Viterbi decoders. As discussed in Chapter 3, two major approaches in generating decoded output sequence are:

1. Register-exchange approach
2. Traceback approach
    a) Shift update
    b) Selective update

The register-exchange approach is suitable for a fast operation, but it incurs more switching activities than the traceback approach. For the traceback approach, it is slower due to the need for the traceback, but it incurs less switching activities. Among the two methods, shift update and selective update, in the traceback approach, the selective update has less switching activity.

The area of major modules and the dynamic power dissipation of the three types of Viterbi decoders are given in Table 4.1. Refer to Figure 3.3 for major modules. (Static power dissipation is not available, since the library cells are not characterized for power dissipation.)

Table 4.1 Area and power dissipation of the three Viterbi decoders

MODULE	REGISTER-EXCHANGE	SHIFT UPDATE	SELECTIVE UPDATE
Butterfly Block	57528 μm^2	57528 μm^2	57528 μm^2
Decoded output sequence generator	110511 μm^2	0 μm^2	0 μm^2
Survivor path storage and traceback	0 μm^2	91737 μm^2	88155 μm^2
Shift register	4770 μm^2	4761 μm^2	4761 μm^2
Counter	1530 μm^2	1521 μm^2	1728 μm^2
Total area	176085 μm^2	157284 μm^2	154008 μm^2
Overall power dissipation	**1167.3** μw	**1528.5** μw	**1183.0** μw

The power dissipation of the three decoders is in the range of 1.1 mW to 1.5 mW, which is small due to the small size of the decoders. The power dissipation of the shift update approach is the highest among the three decoders experimented, dissipating almost 30% more power than the other approaches. This is an expected result as shift update approach incurs more switching activities.

The register-exchange approach results in the highest area overhead among the three decoders. The total size of registers for both approaches is the same, but the register-exchange approach results in more complex circuit for copying of registers.

From experiments, the selective update method and register-exchange method dissipate the same amount of power and that is less than that of shift update method, moroever the selective update method is the most efficient in area.

4.3 Low-power Viterbi decoders

As seen from the above experiment, the selective update method dissipates less power. Hence, we apply low power design techniques to the selective update method to further reduce the power dissipation. Three low power design techniques listed below are considered for our experiments.

1. Toggle Filtering on the decoded output sequence generator block.
2. Replacement of the shift register with multiplexer.
3. Clock gating on the survivor path storage block.

We apply one technique at a time in the above order. If the technique yields less power dissipation, it is accepted and is rejected otherwise.

4.3.1 Toggle filtering for output sequence generation module

Toggle filtering is the process of preventing unnecessary switching by making the input available to a module only when the module needs it. In the decoded output sequence generator of the Viterbi decoder block generates the sequence only when the last code symbol is received. Hence, the input to this module can be disabled for the rest of the period. Table 4.2 shows experimental results for the toggle filtering on the output decision module.

Table 4.2 Power Dissipation of a Viterbi Decoder with and without filtering the toggles

MODULE	SELECTIVE UPDATE METHOD WITHOUT TOGGLE FILTERING	SELECTIVE UPDATE METHOD WITH TOGGLE FILTERING
Butterfly Block	57528 μm^2	57528 μm^2
Survivor path storage and traceback	88155 μm^2	67257 μm^2
Shift register	4761 μm^2	4797 μm^2
Counter	1728 μm^2	1521 μm^2
Total area	154008 μm^2	132831 μm^2
Overall power dissipation	**1183.0 μw**	**776.7 μw**

As expected, the toggle-filtering scheme saves some power. The amount of power saving is 406 μw or 34%. Since the area is also reduced by 14%, the scheme of toggle filtering is worth employing.

4.3.2 Replacement of the shift register module with a multiplexer

The shift register, which is used to shift out a decoded output sequence serially, can be replaced by a multiplexer. (Refer to Figure 3.13) The shift register is 20 bits long, and the replacement needs a 20×1 multiplexer. The use of a multiplexer eliminates the need for shifting for a shift register at the cost of the power dissipation of the multiplexer. Table 4.3 shows the experimental results.

Table 4.3 Power Dissipation of a Viterbi Decoder with different implementations for the shift register block.

MODULE	SELECTIVE UPDATE METHOD WITH A SHIFT REGISTER	SELECTIVE UPDATE METHOD WITH THE MULTIPLEXER
Butterfly Block	57528 μm^2	57528 μm^2
Survivor path storage and traceback	67257 μm^2	67257 μm^2
Shift register	4797 μm^2	6408 μm^2
Counter	1521 μm^2	1521 μm^2
Total area	132831 μm^2	134442 μm^2
Overall power dissipation	**776.7 μw**	**778.1 μw**

As shown in table, a Viterbi decoder with a multiplexer module dissipates almost the same amount of power as that of a decoder with a shift register. However, the multiplexer module occupies almost double the area of a shift register. Hence, the multiplexer module is not employed for the proposed low-power Viterbi decoder design.

4.3.3 Clock gating

Clock gating is a commonly used low power scheme. The survivor path storage block, in which the registers update their values only once for each code word, is a good candidate for the clock gating. Experimental results on the clock gating of the survivor path storage block are given in Table 4.4.

Table 4.4 Power Dissipation of a Viterbi Decoder with and without clock gating for survivor path storage block

MODULE	SELECTIVE UPDATE/TOGGLE FILTERING WITHOUT CLOCK GATING	SELECTIVE UPDATE/TOGGLE FILTERING WITH CLOCK GATING
Butterfly Block	57528 μm^2	57528 μm^2
Survivor path storage and traceback	67257 μm^2	57780 μm^2
Shift register	4797 μm^2	4770 μm^2
Counter	1521 μm^2	1539 μm^2
Total area	132831 μm^2	123309 μm^2
Overall power dissipation	**776.7 μw**	**683.93 μw**

The gating clocking for the survivor module further reduces power dissipation by 13 %. It should be noted that the refernce circuit is a selective update with toggle filtering. The experimental result shows that the gated clock version of the decoder results in less area than the non clock-gated version. Since the clock gating introduces some control logic, in fact, the clock-gated design is expected to be larger in the area compared to the non clock-gated design. Analysis of the synthesizer tools reveals that whenever a register bank is not enabled all the time without clock gating, thus synthesizer infers a multiplexer to feed back the output of a flip-flop to the input. When such a register bank is gated, the synthesizer does not infer a multiplexer to result in an area reduction.

Figure 4.2 shows the power dissipation for the five different implementations of Viterbi decoder. The first three bars in the figure are the power dissipation of original Viterbi decoders. When both the toggle filtering and the clock gating is applied to a selective update Viterbi decoder in traceback approach, it saves about 55% (845 μw) of power compared with a shift update Viterbi decoder and about 42% (500 μw) compared with an original selective update Viterbi decoder.

In summary, the selective update traceback approach is the most power efficient approach. The toggle filtering method is the most effective technique for the low-power design of a Viterbi decoder.

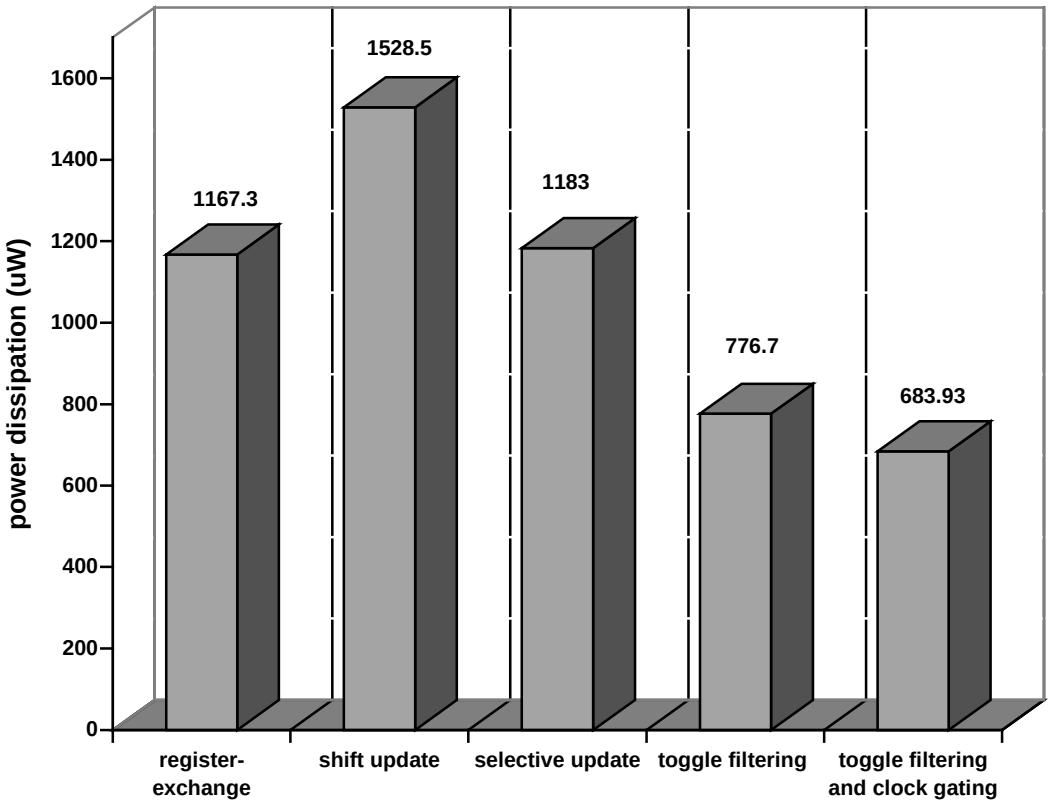

Figure 4.2 Power dissipation of five different implementations of a Viterbi decoder

Chapter 5

Summary

Convolutional coding is a coding scheme often employed in deep space communications and recently in digital wireless communications. Viterbi decoders are used to decode convolutional codes. Viterbi decoders employed in digital wireless communications are complex in its implementation and dissipate large power. We investigated a Viterbi decoder design for low-power dissipation in this thesis.

A Viterbi decoder consists of five major building blocks, input output interface block, branch metric computation block, Add-Compare-Select (ACS) block, survivor path storage block and the output sequence generator block. The I/O interface block provides necessary analog to digital conversion and synchronization. The branch metric computation block computes the branch metrics for each branch. The ACS block generates the survivor path information using the new branch metric and current state metric. The output sequence generator block estimates the original input sequence applied to the encoder.

There are two approaches for generation of decoded output sequence, register-exchange and traceback. In the register-exchange approach each state has a register to store the survivor path information starting from the initial state to the state. These registers are updated for each new code symbol received. So when a complete code word is received, decoded output sequence is readily available. In the traceback scheme, only the survivor path information for each state is stored for each code symbol. Once the complete code word is received, a traceback block extracts the decoded output sequence using the survivor path information. We considered two different methods for the traceback approach, shift update and selective update. In shift update method, the survivor path information is shifted in to the registers. In selective update method the survivor path information is routed by a multiplexer to appropriate registers. We investigated the power dissipation of the register-exchange and traceback approaches and the power dissipation of shift and selective update methods.

The focus of our research in the low-power design of Viterbi decoders at logic level is reduction of dynamic power dissipation in the standard cell design environment. We considered two methods, clock-gating and toggle-filtering, in our design. The clock-gating method was applied to the survivor path storage block and the toggle-filtering method to the traceback block. A control logic disables the clock when the registers are not in use, thus reducing the redundant clock switching at the registers. In toggle filtering, control logic blocks inputs until all of them are ready to be used, which reduces intermediate transitions.

The design flow that we followed is similar to the one used in industry. The behavior of a Viterbi decoder was described in VHDL, and then the description was modified to embed the low-power design. Then the design is synthesized to generate a gate level circuit. The technique of clock gating is applied with the help of synthesizer. To ease testing of the chip, a full scan design was applied to the gate-level design. The gate-level circuit is placed and routed using standard cells and a layout of the circuit was generated.

We measured the dynamic power dissipation of a circuit based on the annotated switching activity from gate level simulation. About 10,000 random patterns were simulated for the gate level design, and the switching activities of nodes are captured and annotated back in the gate level design. Static power dissipation for cells is not considered for the measurement. This method is a compromise between speed and accuracy, and it is sufficient for our purpose.

The selective update method with clock gating and the toggle filtering dissipates 683 μW and has 123309 μm^2 of area. The power saving is 55% compared with the shift update approach. The selective update with low-power techniques reduces 42% of power compared to the selective update approach without low-power techniques.

In summary, we proposed two methods to reduce the power dissipation in Viterbi decoders and demonstrated the effectiveness of the methods through experiments. The proposed methods can be applied to different technology, which is an important requirement to adapt rapidly developing VLSI technology. Finally, although the main goal of the research is to investigate low-power design of Viterbi decoders, we have established a standard cell design-flow at Virginia Tech as a by-product of the research. The establishment of the design flow paves a way for future research in this area.

Bibliography

1. "20 Mbps convolutional encoder Viterbi decoder STEL-2020," *Stanford Telecommunications, Santa Clara, CA,* Oct. 1989.
2. "The Q1650-3L Viterbi codec," *Qualcomm Inc., San Diego*, CA. 1991.
3. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of computer Algorithms*, Reading, MA Addison-Wesley, 1974.
4. ATSC Standard a/53(1995) "ATSC digital television standard", 1995.
5. P.J. Black and T.H.Meng, "A hardware efficient parallel Viterbi algorithm," in *Proc. ICASSP*, vol.2, pp. 893-896, 1990.
6. P.J. Black and T.H.Meng, " A 140-Mb/s, 32-state, radix-4 Viterbi decoder," *IEEE journal of Solid-state Circuits*, vol. 27, pp. 1877-1885, Dec. 1992.
7. P.J. Black and T.H. Meng," A unified approach to the Viterbi algorithm state metric update for shift register processes," *in Proc. ICASSP*, vol.5, pp. 629-632, Mar. 1992.
8. S. Benedetto, G. Montorsi, "Unveiling Turbo codes: Some results on parallel concatenated coding schemes," IEEE Transactions on Information Theory, Vol. 42 No. 2, p. 409-428, March 1996.
9. A. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473-484, Apr. 1992.
10. R. Cypher and C.B. Shung, "Generalized traceback techniques for survivor memory management in the Viterbi algorithm," in *Proc.* GLOBECOM, pp. 1318-1322, Dec. 1990.
11. J.E. Dunn,"A 50Mb/s multiplexed coding system for shuttle communications," *IEEE trans. Commun.*, vol. COM-26, pp. 1636-1638, Nov. 1978.
12. G Fettweis and H, Meyr, "High-speed parallel Viterbi decoding algorithm and VLSI-architecture," *IEEE communication Magazine*, vol. 29, pp. 46-55, May 1991.
13. G. Fettweis, H. Meyr, "A 100 Mbits/s Viterbi decoder chip: Novel architecture and its realization," *IEEE ICC'90,* 307.4, 463-467, April 1990.

14. G. Fettweis, H. Meyr, “A modular variable speed Viterbi decoding implementation for high data rates,” *North-Holland: Signal Processing IV, Proc. EUSIPCO’88*, 339-342, 1988.
15. G. Fettweis, H. Meyr,”A systeolic array Viterbi processor for high data rates,” *Int. Conf. On Systolic Arrays, Ireland*, 1989, Prentice Hall: ‘*Systolic Array Processors*’, 195-204,1989.
16. G. Fettweis, H. Meyr, “Cascaded feedforward architecture for parallel Viterbi decoding,” *IEEE ISCAS*, 978-81, *subm. Kluwer J. VLSI Sig. Proc.* 1990.
17. G. Fettweis and H. Meyr, “Feedforward architecture for parallel viterbi decoding,” *J. VLSI Signal Processing*, vol. 3, pp. 105-119, 1991.
18. G. Fettweis, H. Meyr, “High rate Viterbi processor: a systolic array solution,” *IEEE J. SAC*, Oct. 1990.
19. G. Fettweis, H. Dawid, and H. Meyr, “Minimized method for Viterbi decoding: 600 Mb/s per chip,” *in Proc. GLOBECOM 90*, vol. 3, pp. 1712-1716,Dec. 1990.
20. G. Fettweis, L. Thiele, H. Meyr, “Algorithm transformations for ulimited parallelism,” *Proc. IEEE ISCAS’90*, 1756-59; also L Thiele, G.Fettweis, “-,” *Electronics & Commun*. no.2, 83-91, 4/1990.
21. G.D. Forney, Jr., “The Viterbi algorithm,” *Proceedings of the IEEE*, vol. 61, pp. 268-278, Mar. 1973.
22. P.G.Glak and T.Kailath, “Locality connected VLSI architectures, for the Viterbi algorithm,” *IEEE J. Selected Areas Commun.*, vol.6, no.3, pp. 526-537, Apr. 1988.
23. D. Garrett and M. Stan, “ Low power architecture of the soft-output Viterbi algorithm,” *Electronic-Letters, Proceeding 98 for ISLEPD ’98*, p 262-267, 1998.
24. A.P.Hekstra,”An alternative to metric rescaling in Viterbi decoders,” *IEEE Trans. Commun.*, vol. 37, no.11 pp. 1220-22, Nov. 1989.
25. J. Heller and I. Jacobs, “Viterbi decoding for satellite and space communication,” *IEEE Transactions on Communication Technology*, vol. COM-19, pp. 835-848, Oct. 1971.
26. K. Kameda, “Audio broadcasting satellite service via communication satellites,” NHK R&D, (18), pp. 9-17 (in Japanese), February 1992.

27. I. Kang; A.N. Willson, Jr, " A low-power state-sequential Viterbi decoder for CDMA digital cellular applications," *Conference-Paper, ISCAS 96 (Cat. No.96CH35876). IEEE, New York, NY, USA*; 4 vol.(xlviii+692+801+612+845)pp. p.272-275 vol.4 1996;
28. I. Kang and A.N. Willson Jr," Low-power Viterbi decoder for CDMA mobile terminals, " *Conference-Paper; Journal-article, IEEE-Journal-of-Solid-State-Circuits.* Vol.33, no.3, p.473-82, March 1998.
29. S. Kato, M. Morikura, S. Kubota, H. Kazama and K. Enomoto, "A TDMA satellite communication system for ISDN services," Proc. Globecom'91, pp. 1533-1540, December 1991.
30. K. Kawazoe, S. Honda, S. Kubota and S. Kato, "Ultra-high-speed and universal coding-rate Viterbi decoder VLSIC," Proc. ICC'93, pp. 1434-1438, May 1993.
31. K. Koora, et. al, " From algorithms testing to ASIC input code of SOVA algorithm for Turbo codes," Proceedings of Turbo-code seminar, 1996.
32. S. Kubota, K. Ohtani and S. Kato, "A high-speed and high-coding-gain Viterbi decoder with low power consumption employing SST (scarce state transition) scheme," *Electron. Lett*., 22 (9), pp. 491-493,1986.
33. L. Lang; C.Y. Tsui and R.S. Cheng, "Low power soft output Viterbi decoder scheme for turbo code decoding," *Conference-Paper, ISCAS '97(Cat. No97CH35987). IEEE, New York, NY, USA*, 4 vol. Lxvi+2832 pp. p. 1369-1372 vol.2, 1997.
34. H.F. Lin and D.G. Messerschmitt, "Algorihtms and architectures for concurrent Viterbi decoding," in *Proc. ICC*, pp.836-840, 1989.
35. B.K. Min and N. Demassieux, "A versatile architecture for VLSI implementation of the Viterbi algorithm," in *Proc. ICASSP*, pp. 1101-1104, May 1991.
36. D. Oh and S. Hwang, "Design of a Viterbi decoder with low power using minimum-transition traceback scheme," *Electronic-Letters, IEE*, Vol.32, No.22, pp. 2198-2199, Oct. 1996.
37. D. Oh, Y. Kim, and S. Hwang "A VLSI architecture of the trellis decoder block for the digital HDTV grand alliance system," *IEEE Trans. Consum. Electron*, 42, (3) pp. 346-356, 1996.

38. J.K. Omura, "On the Viterbi decoding algorithm," *IEEE Transactions on Information Theory*, vol. IT-15, pp. 177-179, Jan 1969.
39. K.K. Parhi, D.G.Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters," *IEEE Tr. ASSP*, 1099-117, 7/89.
40. Qualcomm, Inc., "Proposed EIA/TIA interim standard wideband spread spectrum digital cellular system dual-mode mobile station – base station compatibility standard," April 21, 1992.
41. C.M. Rader, "Memory management in a Viterbi decoder," *IEEE Transactions on Communications*, vol. COM-29, pp. 1399-1401, Sept. 1981.
42. R. Scheweikert, A.J.Vinck,"Trellis coded modulation with high-speed low complexity decoding," *submitted to IEEE GLOBECOM* 1990.
43. K. Seki; S. Kubota; M. Mizoguchi and S. Kato, "Very low power consumption Viterbi decoder LSIC employing the SST (scarce state transition) scheme for multimedia mobile communications," *Electronics-Letters, IEE*, Vol.30, no.8, p.637-639, 14 April 1994.
44. C.B. Shung, P.H. Siegel, H.K. Thapar, and R. Karabed, "A 30-MHz trellis coded chip for partial response channels," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 1981-1987, Dec. 1991.
45. T. Trung, et. al, "A VLSI design for a traceback Viterbi decoder," IEEE Trans. On Communications, vol. 40 no.3, p 616-624, March 1992.
46. K.H. Tzou and J.G.Dunham, "Sliding block decoding of convolutioal codes," *IEEE trans. Commun.*, vol. COM-29, pp. 1401-1403, Sept. 1981.
47. J.D.Ullman, *Computational aspects of VLSI*, Rockville, Maryland: Computer Science Press, 1984.
48. G. Ungerbock, "Channel Coding with Mutlilevel/Phase Signals," *IEEE Tr Inf. Th.*, 55-67, 1/1982.
49. A.J. Viterbi, "Convolutional codes and their performance in communication systems," *IEEE Transactions on Communication Technology*, vol. COM-19, pp. 751-772, Oct. 1971.
50. A.J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260-269, Apr. 1967.

51. A.J. Viterbi and J.K. Omura, *Principles of Digital Communication and Coding*, New York McGraw-Hill, 1979.
52. S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, 1995.
53. R.W. Wood and D.A. Petersen,"Viterbi detection of Class IV partial response on a magnetic recording channel," *IEEE trans. Commun*. Vol. COM-34, pp. 454-461, May 1986.
54. G. K. Yeap, *Practical Low Power Digital VLSI Design*, Kluwer Academic Publishers, 1998.
55. T.C.Ying, M. Pedram, and A. Despain, "Exact and approximate methods for switching activity estimation in sequential logic circuits," Proc. 31st DAC, pp. 18-23, June 1994.

Appendix A. Software Documentation.

```
-- Module : Encoder.

-- Convolutional ( 3,1,4 ) Encoder

-- Inputs.
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. inp ( 1-bit ) : input information to be encoded.
-- Outputs :
-- 1. outpp ( 3-bit ) : encoded output

library IEEE;
use IEEE.std_logic_1164.all;

entity encoder is
      port(  clk,inp,res  :  in  std_logic  ;  outpp  :  out  std_logic_VECTOR(  2
downto 0 ) ) ;

end encoder ;

architecture str of encoder is
signal ff : std_logic_vector( 3 downto 0 ) ;
begin
process( clk, res )
begin

      if clk'EVENT and clk = '1'  then
            if res = '1' then

                  for i in 3 downto 0  loop
                        ff(i) <= '0' ;
                  end loop ;

            else
                  for i in 0 to 2 loop
                        ff(i) <= ff(i+1) ;
                  end loop ;

                  ff(3) <= inp ;
            end if ;
      end if ;

end process ;
            outpp(0) <= inp xor ff(2) xor ff(0)  ;
            outpp(1) <= inp xor ff(3) xor ff(1) xor ff(0) ;
            outpp(2) <= inp xor ff(3) xor ff(2) xor ff(1) xor ff(0) ;

 end   str ;
```

```
-- Module : Buffer.

-- The buffer module required just for tapping off the encoded output.
-- Inputs :
-- 1. inp ( 3-bit ) : encoded input
-- Outputs :
-- 1. opp ( 3-bit ) : encoded output

library IEEE;
use IEEE.std_logic_1164.all;

entity op is
      port( inp : in std_logic_VECTOR(2 downto 0);
                  opp : out std_logic_VECTOR(2 downto 0) );
   end op ;

  architecture simp of op is

  begin
      opp <= inp ;
  end simp ;
```

```
-- Module : noise


-- This is the module from which noise can be inserted in the encoded
-- input for testing purposes.

-- Inputs :
-- 1. nin ( 3-bit ) : encoded input
-- 2. nctrl ( 3-bit ) : noise bits

-- Output :
-- 1. nout  ( 3-bit ) : encoded output with noise


library IEEE;
use IEEE.std_logic_1164.all;

entity noise is
      port(       nin :in std_logic_VECTOR(2 downto 0);
                  nout : out std_logic_VECTOR(2 downto 0);
                  nctrl : in std_logic_VECTOR(2 downto 0)
           );
 end noise ;

 architecture xors of noise is

 begin


            nout(0) <= nin(0) xor nctrl(0) ;
            nout(1) <= nin(1) xor nctrl(1) ;
            nout(2) <= nin(2) xor nctrl(2) ;


 end   xors ;
```

```
-- Module : 0


-- Branch metric generator and Add-Compare-Select Module for the 0th
butterfly.
-- It stores the state metrics for state 0 and 8. It selects the survivor
path
-- for state 0 and state 1. It forwards decision to the survivor path storage
module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 8
-- 5. imetl ( 6-bits ) : state metric for state 0
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec0 ( 1-bit ) : survivor path decision for state 0
-- 2. dec1 ( 1-bit ) : survivor path decision for state 1
-- 3. ometu ( 6-bits ) : state metric for state 1
-- 4. ometl ( 6-bit ) : state metric for state 0

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

entity ACS_0 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec0, dec1 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )  ) ;
end ACS_0 ;

architecture spec of ACS_0 is
begin

      process( clk, rc ,res , count )
      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;


      begin

      if clk'event and clk = '1' then
            if res = '1' then
                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
                  inb := '0' ;
                  inc := '0' ;
```

```
            dec0 <= '0' ;
            dec1 <= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else

ina := rc(0)  ;
inb := rc(1)  ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina :=  not rc(0) ;
inb :=  not rc(1) ;
inc :=  not rc(2) ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100" )  or ( lower <= upper )  ) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec0 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec0 <= '1' ;
end if ;


lower := unsigned( imetl ) + unsigned( re ) ;


ina :=  rc(0);
inb :=  rc(1);
inc :=  rc(2);

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( (count <  "00100" )  or ( lower <= upper ) ) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec1 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
```

```
            dec1 <= '1' ;
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 1


-- Branch metric generator and Add-Compare-Select Module for the 1st
butterfly.
-- It stores the state metrics for state 1 and 9. It selects the survivor
path
-- for state 2 and state 3. It forwards decision to the survivor path storage
module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 9
-- 5. imetl ( 6-bits ) : state metric for state 1
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec2 ( 1-bit ) : survivor path decision for state 2
-- 2. dec3 ( 1-bit ) : survivor path decision for state 3
-- 3. ometu ( 6-bits ) : state metric for state 3
-- 4. ometl ( 6-bit ) : state metric for state 2

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

entity ACS_1 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec2, dec3 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )  ) ;
end ACS_1 ;

architecture spec of ACS_1 is

begin

      process( clk, rc ,res, count )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc  : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec2 <= '0' ;
            dec3<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else


ina := rc(0)  ;
inb := not rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;


ina := not rc(0)  ;
inb :=  rc(1)  ;
inc :=  rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100" )  or (lower <= upper) ) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec2 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec2 <= '1' ;
end if ;


lower := unsigned( imetl ) + unsigned( re ) ;


ina :=  rc(0)  ;
inb := not rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100" )  or ( lower <= upper ) ) then
```

```
            ometu <= conv_std_logic_vector( lower , 6) ;
            dec3 <= '0' ;
      else
            ometu <= conv_std_logic_vector( upper , 6) ;
            dec3 <= '1' ;
      end if ;
      end if ;
      end if ;


      end process ;

end spec;
```

```
-- Module : 2


-- Branch metric generator and Add-Compare-Select Module for the 2nd
butterfly.
-- It stores the state metrics for state 2 and 10. It selects the survivor
path
-- for state 4 and state 5. It forwards decision to the survivor path storage
module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 10
-- 5. imetl ( 6-bits ) : state metric for state 2
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec4 ( 1-bit ) : survivor path decision for state 4
-- 2. dec5 ( 1-bit ) : survivor path decision for state 5
-- 3. ometu ( 6-bits ) : state metric for state 5
-- 4. ometl ( 6-bit ) : state metric for state 4

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

entity ACS_2 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec4, dec5 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )) ;
end ACS_2 ;

architecture spec of ACS_2 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc  : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then
                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
                  inb := '0' ;
```

```
            inc := '0' ;
            dec4<= '0' ;
            dec5<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;
      else

ina := not rc(0)  ;
inb := rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina :=  rc(0)  ;
inb := not rc(1) ;
inc :=  rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100")  or ( lower <= upper ) ) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec4 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec4 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := not rc(0)  ;
inb := rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if(  ( count <  "00100" )  or ( lower <= upper )) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec5 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
      dec5 <= '1' ;
end if ;
```

```
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 3


-- Branch metric generator and Add-Compare-Select Module for the 3rd
butterfly.
-- It stores the state metrics for state 3 and 11. It selects the survivor
path
-- for state 6 and state 7. It forwards decision to the survivor path storage
module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 11
-- 5. imetl ( 6-bits ) : state metric for state 3
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec6 ( 1-bit ) : survivor path decision for state 6
-- 2. dec7 ( 1-bit ) : survivor path decision for state 7
-- 3. ometu ( 6-bits ) : state metric for state 7
-- 4. ometl ( 6-bit ) : state metric for state 6

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

entity ACS_3 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec6, dec7 : out std_logic;
      count : in std_logic_vector( 4 downto 0 )  ) ;
end ACS_3 ;

architecture spec of ACS_3 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable ina, inb, inc  : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec6<= '0' ;
            dec7<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else

ina := not rc(0)  ;
inb := not rc(1)  ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina :=  rc(0)  ;
inb :=  rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;

if((count <  "00100" )  or (  lower <= upper )) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec6 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec6 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := not rc(0)  ;
inb := not rc(1)  ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( (count <  "00100" )  or ( lower <= upper) ) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec7 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
      dec7 <= '1' ;
```

```
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 4


-- Branch metric generator and Add-Compare-Select Module for the 4th
butterfly.
-- It stores the state metrics for state 4 and 12. It selects the survivor
path
-- for state 8 and state 9. It forwards decision to the survivor path storage
module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 12
-- 5. imetl ( 6-bits ) : state metric for state 4
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec8 ( 1-bit ) : survivor path decision for state 8
-- 2. dec9 ( 1-bit ) : survivor path decision for state 9
-- 3. ometu ( 6-bits ) : state metric for state 9
-- 4. ometl ( 6-bit ) : state metric for state 8

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

entity ACS_4 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec8, dec9 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 ) ) ;
end ACS_4 ;

architecture spec of ACS_4 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec8<= '0' ;
            dec9<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else

ina := rc(0)  ;
inb := not rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := not  rc(0) ;
inb :=  rc(1) ;
inc :=  rc(2) ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100" )  or ( lower <= upper )) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec8 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec8 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := rc(0)  ;
inb := not rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( ( count <  "00100" )  or ( lower <= upper)  ) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec9 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
```

```
            dec9 <= '1' ;
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 5


-- Branch metric generator and Add-Compare-Select Module for the 5th
butterfly.
-- It stores the state metrics for state 5 and 13. It selects the survivor
path
-- for state 10 and state 11. It forwards decision to the survivor path
storage module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 13
-- 5. imetl ( 6-bits ) : state metric for state 5
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec10 ( 1-bit ) : survivor path decision for state 10
-- 2. dec11 ( 1-bit ) : survivor path decision for state 11
-- 3. ometu ( 6-bits ) : state metric for state 11
-- 4. ometl ( 6-bit ) : state metric for state 10

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

entity ACS_5 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec10,dec11 : out std_logic;
       count : in std_logic_vector( 4 downto 0 ) ) ;
end ACS_5 ;

architecture spec of ACS_5 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec10<= '0' ;
            dec11<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else

ina := rc(0)  ;
inb := rc(1)  ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina :=  not rc(0) ;
inb :=  not rc(1) ;
inc :=  not rc(2) ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( (count <  "00100" )  or ( lower <= upper) ) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec10 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec10 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := rc(0)  ;
inb := rc(1)  ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if(  (count <  "00100" )  or ( lower <= upper) ) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec11 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
```

```
            dec11 <= '1' ;
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 6


-- Branch metric generator and Add-Compare-Select Module for the 6th
butterfly.
-- It stores the state metrics for state 6 and 14. It selects the survivor
path
-- for state 12 and state 13. It forwards decision to the survivor path
storage module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 14
-- 5. imetl ( 6-bits ) : state metric for state 6
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec12 ( 1-bit ) : survivor path decision for state 12
-- 2. dec13 ( 1-bit ) : survivor path decision for state 13
-- 3. ometu ( 6-bits ) : state metric for state 13
-- 4. ometl ( 6-bit ) : state metric for state 12

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

entity ACS_6 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec12,dec13 : out std_logic;
      count : in std_logic_vector( 4 downto 0 )   ) ;
end ACS_6 ;

architecture spec of ACS_6 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec12<= '0' ;
            dec13<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;
      else

ina := not rc(0) ;
inb := not rc(1) ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;


ina :=  rc(0)  ;
inb :=  rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if(( count <  "00100")  or (  lower <= upper )) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec12 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec12 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := not rc(0) ;
inb := not rc(1) ;
inc := rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( (count <  "00100" )  or (  lower <= upper )) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec13 <= '0' ;
else
      ometu <= conv_std_logic_vector( upper , 6) ;
```

```
            dec13 <= '1' ;
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : 7


-- Branch metric generator and Add-Compare-Select Module for the 7th
butterfly.
-- It stores the state metrics for state 7 and 15. It selects the survivor
path
-- for state 14 and state 15. It forwards decision to the survivor path
storage module.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. rc ( 3-bits ) : input code symbol
-- 4. imetu ( 6-bits ) : state metric for state 15
-- 5. imetl ( 6-bits ) : state metric for state 7
-- 6. count ( 5-bits ) : system counter
-- Outputs :
-- 1. dec14 ( 1-bit ) : survivor path decision for state 14
-- 2. dec15 ( 1-bit ) : survivor path decision for state 15
-- 3. ometu ( 6-bits ) : state metric for state 15
-- 4. ometl ( 6-bit ) : state metric for state 14

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

entity ACS_7 is
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec14, dec15 : out std_logic ;
        count : in std_logic_vector( 4 downto 0 )  ) ;
end ACS_7 ;

architecture spec of ACS_7 is

begin

      process( clk, rc ,res )

      variable lower,upper  : unsigned( 5 downto 0 );
      variable  ina, inb, inc : std_logic ;
      variable re : std_logic_vector( 1 downto 0  ) ;
      begin

      if clk'event and clk = '1' then

            if res = '1' then

                  lower := "000000" ;
                  upper := "000000" ;
                  ina := '0' ;
```

```
            inb := '0' ;
            inc := '0' ;
            dec14<= '0' ;
            dec15<= '0' ;
            re := "00" ;
            ometu <= "000000" ;
            ometl <= "000000" ;

      else

ina := not rc(0)  ;
inb := rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

lower := unsigned( imetl ) + unsigned( re ) ;


ina :=  rc(0)  ;
inb := not rc(1)  ;
inc :=  rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if((count <  "00100" )  or (  lower <= upper )) then
      ometl <= conv_std_logic_vector( lower, 6 ) ;
      dec14 <= '0' ;
else
      ometl <= conv_std_logic_vector( upper , 6 ) ;
      dec14 <= '1' ;
end if ;

lower := unsigned( imetl ) + unsigned( re ) ;

ina := not rc(0)  ;
inb := rc(1)  ;
inc := not rc(2)  ;

re(0) := ina xor inb xor inc ;
re(1) := ( ina and inb ) or ( ina and inc ) or ( inb and inc ) ;

upper := unsigned( imetu ) + unsigned( re ) ;


if( (count <  "00100")  or ( lower <= upper ) ) then
      ometu <= conv_std_logic_vector( lower , 6) ;
      dec15 <= '0' ;
else
```

```
            ometu <= conv_std_logic_vector( upper , 6) ;
            dec15 <= '1' ;
      end if ;
      end if ;
      end if ;

      end process ;

end spec;
```

```
-- Module : fast_trace


--  Register-exchange estimated input sequence generator block. This block
gets
--  the survivor path information from ACS modules and carries out fast
regsiter-exchange
-- estimated input sequence is available as soon as the last code sybol in
the
-- code word is received.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. dec ( 16-bit ): survivor path information for code symbol
-- 4. count ( 5-bit ) : system counter input

-- Outputs :
-- 1. outp ( 20-bit ) : estimated input sequence

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.arraylib.all ;

entity fast_trace is
      port( clk, res : in std_logic ; dec : in std_logic_vector( 15 downto 0
) ;
      count   :   in   std_logic_vector(   4   downto   0   )   ;   outp   :   out
std_logic_vector( 19 downto 0 ) ) ;
end fast_trace ;

architecture low_power of fast_trace is

function   calc(  ins  :  arrayx16(  19  downto  0)  ;  j  :  integer  ;  dec  :
std_logic_vector( 15 downto 0 )
) return arrayx16 is
variable ret : arrayx16( 19 downto 0) ;
begin
      for i in 15 downto 0 loop

                  if ( dec(i) = '0' ) then
                        ret(i)(19  downto  (j+1)  )  :=  ins(  i/2  )(19  downto
(j+1) ) ;
                        ret(i)(j) := '0' ;
                  else
                        ret(i)(19 downto (j+1) ) := ins( i/2 + 8 )(19 downto
(j+1) ) ;
                        ret(i)(j) := '1' ;
                  end if ;
      end loop ;

return ret ;
end calc ;
```

```
      signal w : arrayx16( 19 downto 0) ;
begin
      process(res, count, dec, clk )



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

      if res = '1' then

            for i in 15 downto 0 loop
                  for j in 19 downto 0 loop
                        w(i)(j) <= '0' ;
                  end loop ;
            end loop ;


      else

      if( count = "00001" ) then
            for i in 15 downto 0 loop

                  if ( dec(i) = '0' ) then
                        w(i)(19) <= '0' ;
                  else
                        w(i)(19) <= '1' ;
                  end if ;
      end loop ;
      elsif( count = "00010" ) then
            w <= calc( w , 18 , dec ) ;
      elsif( count = "00011" ) then
            w <= calc( w , 17 , dec ) ;
      elsif( count = "00100" ) then
            w <= calc( w , 16 , dec ) ;
      elsif( count = "00101" ) then
            w <= calc( w , 15 , dec ) ;
      elsif( count = "00110" ) then
            w <= calc( w , 14 , dec ) ;
      elsif( count = "00111" ) then
            w <= calc( w , 13 , dec ) ;
      elsif( count = "01000" ) then
            w <= calc( w , 12 , dec ) ;
      elsif( count = "01001" ) then
            w <= calc( w , 11 , dec ) ;
      elsif( count = "01010" ) then
            w <= calc( w , 10, dec ) ;
      elsif( count = "01011" ) then
            w <= calc( w , 9, dec ) ;
      elsif( count = "01100" ) then
            w <= calc( w , 8 , dec ) ;
      elsif( count = "01101" ) then
            w <= calc( w , 7 , dec ) ;
```

```
      elsif( count = "01110" ) then
            w <= calc( w , 6 , dec ) ;
      elsif( count = "01111" ) then
            w <= calc( w , 5 , dec ) ;
      elsif( count = "10000" ) then
            w <= calc( w , 4 , dec ) ;
      elsif( count = "10001" ) then
            w <= calc( w , 3 , dec ) ;
      elsif( count = "10010" ) then
            w <= calc( w , 2 , dec ) ;
      elsif( count = "10011" ) then
            w <= calc( w , 1 , dec ) ;
      end if ;
      if( count = "00000" ) then
            w <= calc( w , 0 , dec ) ;


            end if ;
      end if ;
      end if ;
      end process ;
      outp <= w(0) ;

end low_power ;
```

```
-- Module : fast_trace


-- This is the Selective-update survivor path storage and estimated input
-- sequence generator along without toggle filtering.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. dec ( 16-bit ): survivor path information for code symbol
-- 4. count ( 5-bit ) : system counter input

-- Outputs :
-- 1. outp ( 20-bit ) : estimated input sequence

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.arraylib.all ;

entity fast_trace is
      port( clk, res : in std_logic ; dec : in std_logic_vector( 15 downto 0
) ;
      count   :   in   std_logic_vector(   4   downto   0   )   ;   outp   :   out
std_logic_vector( 19 downto 0 ) ) ;
end fast_trace ;

architecture low_power of fast_trace is
function find( datain : std_logic_VECTOR ; stt: std_logic_VECTOR( 3 downto 0
) ) return std_logic_VECTOR is
    variable vst :std_logic_VECTOR( 3 downto 0 ) ;
    begin
            for i in  3 downto 1 loop
                        vst(i-1) := stt(i) ;
                  end loop ;
            if ( datain(conv_integer( unsigned( stt ) ) )= '0' ) then
                  vst(3) := '0' ;
            else
                  vst(3) := '1' ;
            end if ;
    return vst ;
    end find ;



      signal w_1 : std_logic_vector( 15 downto 0) ;
      signal w_2 : std_logic_vector( 15 downto 0) ;
      signal w_3 : std_logic_vector( 15 downto 0) ;
      signal w_4 : std_logic_vector( 15 downto 0) ;
      signal w_5 : std_logic_vector( 15 downto 0) ;
      signal w_6 : std_logic_vector( 15 downto 0) ;
      signal w_7 : std_logic_vector( 15 downto 0) ;
      signal w_8 : std_logic_vector( 15 downto 0) ;
```

```
      signal w_9 : std_logic_vector( 15 downto 0) ;
      signal w_10 : std_logic_vector( 15 downto 0) ;
      signal w_11 : std_logic_vector( 15 downto 0) ;
      signal w_12 : std_logic_vector( 15 downto 0) ;
      signal w_13 : std_logic_vector( 15 downto 0) ;
      signal w_14 : std_logic_vector( 15 downto 0) ;
      signal w_15 : std_logic_vector( 15 downto 0) ;
      signal w_16 : std_logic_vector( 15 downto 0) ;
      signal w_17 : std_logic_vector( 15 downto 0) ;
      signal w_18 : std_logic_vector( 15 downto 0) ;
      signal w_19 : std_logic_vector( 15 downto 0) ;
      signal w_20 : std_logic_vector( 15 downto 0) ;
begin
      process(res, count, dec, clk )
      variable tmp : std_logic_vector( 3 downto 0 )  ;

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

      if ( res = '1' )   then

            w_1 <= "0000000000000000" ;
            w_2 <= "0000000000000000" ;
            w_3 <= "0000000000000000" ;
            w_4 <= "0000000000000000" ;
            w_5 <= "0000000000000000" ;
            w_6 <= "0000000000000000" ;
            w_7 <= "0000000000000000" ;
            w_8 <= "0000000000000000" ;
            w_9 <= "0000000000000000" ;
            w_10 <= "0000000000000000" ;
            w_11 <= "0000000000000000" ;
            w_12 <= "0000000000000000" ;
            w_13 <= "0000000000000000" ;
            w_14 <= "0000000000000000" ;
            w_15 <= "0000000000000000" ;
            w_16 <= "0000000000000000" ;
            w_17 <= "0000000000000000" ;
            w_18 <= "0000000000000000" ;
            w_19 <= "0000000000000000" ;
            w_20 <= "0000000000000000" ;
            outp <= "00000000000000000000" ;
      else

      if( count = "00001" ) then
            w_1 <= dec ;
      elsif( count = "00010" ) then
            w_2 <= dec ;
      elsif( count = "00011" ) then
            w_3 <= dec ;
      elsif( count = "00100" ) then
            w_4 <= dec ;
      elsif( count = "00101" ) then
            w_5 <= dec ;
```

```
elsif( count = "00110" ) then
      w_6 <= dec ;
elsif( count = "00111" ) then
      w_7 <= dec ;
elsif( count = "01000" ) then
      w_8 <= dec ;
elsif( count = "01001" ) then
      w_9 <= dec ;
elsif( count = "01010" ) then
      w_10 <= dec ;
elsif( count = "01011" ) then
      w_11 <= dec ;
elsif( count = "01100" ) then
      w_12 <= dec ;
elsif( count = "01101" ) then
      w_13 <= dec ;
elsif( count = "01110" ) then
      w_14 <= dec ;
elsif( count = "01111" ) then
      w_15 <= dec ;
elsif( count = "10000" ) then
      w_16 <= dec ;
elsif( count = "10001" ) then
      w_17 <= dec ;
elsif( count = "10010" ) then
      w_18 <= dec ;
elsif( count = "10011" ) then
      w_19 <= dec ;
end if ;

      w_20 <= dec ;
      tmp := "0000" ;
      outp(0) <= tmp(0) ;
      tmp := find( w_20 , tmp ) ;
      outp(1) <= tmp(0) ;
      tmp := find( w_19 , tmp ) ;
      outp(2) <= tmp(0) ;
      tmp := find( w_18 , tmp ) ;
      outp(3) <= tmp(0) ;
      tmp := find( w_17 , tmp ) ;
      outp(4) <= tmp(0) ;
      tmp := find( w_16 , tmp ) ;
      outp(5) <= tmp(0) ;
      tmp := find( w_15 , tmp ) ;
      outp(6) <= tmp(0) ;
      tmp := find( w_14 , tmp ) ;
      outp(7) <= tmp(0) ;
      tmp := find( w_13 , tmp ) ;
      outp(8) <= tmp(0) ;
      tmp := find( w_12 , tmp ) ;
      outp(9) <= tmp(0) ;
      tmp := find( w_11 , tmp ) ;
      outp(10) <= tmp(0) ;
      tmp := find( w_10 , tmp ) ;
```

```
            outp(11) <= tmp(0) ;
            tmp := find( w_9 , tmp ) ;
            outp(12) <= tmp(0) ;
            tmp := find( w_8 , tmp ) ;
            outp(13) <= tmp(0) ;
            tmp := find( w_7 , tmp ) ;
            outp(14) <= tmp(0) ;
            tmp := find( w_6 , tmp ) ;
            outp(15) <= tmp(0) ;
            tmp := find( w_5 , tmp ) ;
            outp(16) <= tmp(0) ;
            tmp := find( w_4 , tmp ) ;
            outp(17) <= tmp(0) ;
            tmp := find( w_3 ,tmp ) ;
            outp(18) <= tmp(0) ;
            tmp := find( w_2 , tmp ) ;
            outp(19) <= tmp(0) ;

      end if ;
      end if ;
      end process ;
end low_power ;
```

```
-- Module : fast_trace


-- This is the Shift-update survivor path storage and estimated input
-- sequence generator along with toggle filtering.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. dec ( 16-bit ): survivor path information for code symbol
-- 4. count ( 5-bit ) : system counter input

-- Outputs :
-- 1. outp ( 20-bit ) : estimated input sequence

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.arraylib.all ;

entity fast_trace is
      port( clk, res : in std_logic ; dec : in std_logic_vector( 15 downto 0
) ;
      count   :   in   std_logic_vector(   4   downto   0   )   ;   outp   :   out
std_logic_vector( 19 downto 0 ) ) ;
end fast_trace ;

architecture low_power of fast_trace is
function find( datain : std_logic_VECTOR ; stt: std_logic_VECTOR( 3 downto 0
) ) return std_logic_VECTOR is
    variable vst :std_logic_VECTOR( 3 downto 0 ) ;
    begin
            for i in  3 downto 1 loop
                        vst(i-1) := stt(i) ;
                  end loop ;
            if ( datain(conv_integer( unsigned( stt ) ) )= '0' ) then
                  vst(3) := '0' ;
            else
                  vst(3) := '1' ;
            end if ;
    return vst ;
    end find ;

begin
      process(res, count, dec, clk )


      variable tmp : std_logic_vector( 3 downto 0 )  ;

      variable w_1 : std_logic_vector( 15 downto 0) ;
      variable w_2 : std_logic_vector( 15 downto 0) ;
      variable w_3 : std_logic_vector( 15 downto 0) ;
      variable w_4 : std_logic_vector( 15 downto 0) ;
      variable w_5 : std_logic_vector( 15 downto 0) ;
```

```
variable w_6 : std_logic_vector( 15 downto 0) ;
variable w_7 : std_logic_vector( 15 downto 0) ;
variable w_8 : std_logic_vector( 15 downto 0) ;
variable w_9 : std_logic_vector( 15 downto 0) ;
variable w_10 : std_logic_vector( 15 downto 0) ;
variable w_11 : std_logic_vector( 15 downto 0) ;
variable w_12 : std_logic_vector( 15 downto 0) ;
variable w_13 : std_logic_vector( 15 downto 0) ;
variable w_14 : std_logic_vector( 15 downto 0) ;
variable w_15 : std_logic_vector( 15 downto 0) ;
variable w_16 : std_logic_vector( 15 downto 0) ;
variable w_17 : std_logic_vector( 15 downto 0) ;
variable w_18 : std_logic_vector( 15 downto 0) ;
variable w_19 : std_logic_vector( 15 downto 0) ;
variable w_20 : std_logic_vector( 15 downto 0) ;

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

if ( res = '1' )   then

      w_1 := "0000000000000000" ;
      w_2 := "0000000000000000" ;
      w_3 := "0000000000000000" ;
      w_4 := "0000000000000000" ;
      w_5 := "0000000000000000" ;
      w_6 := "0000000000000000" ;
      w_7 := "0000000000000000" ;
      w_8 := "0000000000000000" ;
      w_9 := "0000000000000000" ;
      w_10 := "0000000000000000" ;
      w_11 := "0000000000000000" ;
      w_12 := "0000000000000000" ;
      w_13 := "0000000000000000" ;
      w_14 := "0000000000000000" ;
      w_15 := "0000000000000000" ;
      w_16 := "0000000000000000" ;
      w_17 := "0000000000000000" ;
      w_18 := "0000000000000000" ;
      w_19 := "0000000000000000" ;
      w_20 := "0000000000000000" ;
      outp <= "00000000000000000000" ;
else

if( count >= "00001" ) and ( count <= "10011" ) then
      w_1 := w_2 ;
      w_2 := w_3 ;
      w_3 := w_4 ;
      w_4 := w_5 ;
      w_5 := w_6 ;
      w_6 := w_7 ;
      w_7 := w_8 ;
      w_8 := w_9 ;
      w_9 :=w_10  ;
```

```
      w_10 := w_11 ;
      w_11 := w_12 ;
      w_12 := w_13 ;
      w_13 := w_14 ;
      w_14 := w_15 ;
      w_15 := w_16 ;
      w_16 := w_17 ;
      w_17 := w_18 ;
      w_18 := w_19 ;
      w_19 := dec ;
end if ;
      w_20 := dec ;
      tmp := "0000" ;
      outp(0) <= tmp(0) ;
      tmp := find( w_20 , tmp ) ;
      outp(1) <= tmp(0) ;
      tmp := find( w_19 , tmp ) ;
      outp(2) <= tmp(0) ;
      tmp := find( w_18 , tmp ) ;
      outp(3) <= tmp(0) ;
      tmp := find( w_17 , tmp ) ;
      outp(4) <= tmp(0) ;
      tmp := find( w_16 , tmp ) ;
      outp(5) <= tmp(0) ;
      tmp := find( w_15 , tmp ) ;
      outp(6) <= tmp(0) ;
      tmp := find( w_14 , tmp ) ;
      outp(7) <= tmp(0) ;
      tmp := find( w_13 , tmp ) ;
      outp(8) <= tmp(0) ;
      tmp := find( w_12 , tmp ) ;
      outp(9) <= tmp(0) ;
      tmp := find( w_11 , tmp ) ;
      outp(10) <= tmp(0) ;
      tmp := find( w_10 , tmp ) ;
      outp(11) <= tmp(0) ;
      tmp := find( w_9 , tmp ) ;
      outp(12) <= tmp(0) ;
      tmp := find( w_8 , tmp ) ;
      outp(13) <= tmp(0) ;
      tmp := find( w_7 , tmp ) ;
      outp(14) <= tmp(0) ;
      tmp := find( w_6 , tmp ) ;
      outp(15) <= tmp(0) ;
      tmp := find( w_5 , tmp ) ;
      outp(16) <= tmp(0) ;
      tmp := find( w_4 , tmp ) ;
      outp(17) <= tmp(0) ;
      tmp := find( w_3 ,tmp ) ;
      outp(18) <= tmp(0) ;
      tmp := find( w_2 , tmp ) ;
      outp(19) <= tmp(0) ;

end if ;
```

```
      end if ;
      end process ;
end low_power ;
```

```
-- Module : fast_trace


-- This is the Selective-update survivor path storage and estimated input
-- sequence generator along with toggle filtering.

-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. dec ( 16-bit ): survivor path information for code symbol
-- 4. count ( 5-bit ) : system counter input

-- Outputs :
-- 1. outp ( 20-bit ) : estimated input sequence

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.arraylib.all ;

entity fast_trace is
      port( clk, res : in std_logic ; dec : in std_logic_vector( 15 downto 0
) ;
      count   :   in   std_logic_vector(   4   downto   0   )   ;   outp   :   out
std_logic_vector( 19 downto 0 ) ) ;
end fast_trace ;

architecture low_power of fast_trace is
function find( datain : std_logic_VECTOR ; stt: std_logic_VECTOR( 3 downto 0
) ) return std_logic_VECTOR is
    variable vst :std_logic_VECTOR( 3 downto 0 ) ;
    begin
            for i in  3 downto 1 loop
                        vst(i-1) := stt(i) ;
                  end loop ;
            if ( datain(conv_integer( unsigned( stt ) ) )= '0' ) then
                  vst(3) := '0' ;
            else
                  vst(3) := '1' ;
            end if ;
    return vst ;
    end find ;



      signal w_1 : std_logic_vector( 15 downto 0) ;
      signal w_2 : std_logic_vector( 15 downto 0) ;
      signal w_3 : std_logic_vector( 15 downto 0) ;
      signal w_4 : std_logic_vector( 15 downto 0) ;
      signal w_5 : std_logic_vector( 15 downto 0) ;
      signal w_6 : std_logic_vector( 15 downto 0) ;
      signal w_7 : std_logic_vector( 15 downto 0) ;
      signal w_8 : std_logic_vector( 15 downto 0) ;
```

```
      signal w_9 : std_logic_vector( 15 downto 0) ;
      signal w_10 : std_logic_vector( 15 downto 0) ;
      signal w_11 : std_logic_vector( 15 downto 0) ;
      signal w_12 : std_logic_vector( 15 downto 0) ;
      signal w_13 : std_logic_vector( 15 downto 0) ;
      signal w_14 : std_logic_vector( 15 downto 0) ;
      signal w_15 : std_logic_vector( 15 downto 0) ;
      signal w_16 : std_logic_vector( 15 downto 0) ;
      signal w_17 : std_logic_vector( 15 downto 0) ;
      signal w_18 : std_logic_vector( 15 downto 0) ;
      signal w_19 : std_logic_vector( 15 downto 0) ;
      signal w_20 : std_logic_vector( 15 downto 0) ;
begin
      process(res, count, dec, clk )
      variable tmp : std_logic_vector( 3 downto 0 )  ;

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

      if ( res = '1' )   then

            w_1 <= "0000000000000000" ;
            w_2 <= "0000000000000000" ;
            w_3 <= "0000000000000000" ;
            w_4 <= "0000000000000000" ;
            w_5 <= "0000000000000000" ;
            w_6 <= "0000000000000000" ;
            w_7 <= "0000000000000000" ;
            w_8 <= "0000000000000000" ;
            w_9 <= "0000000000000000" ;
            w_10 <= "0000000000000000" ;
            w_11 <= "0000000000000000" ;
            w_12 <= "0000000000000000" ;
            w_13 <= "0000000000000000" ;
            w_14 <= "0000000000000000" ;
            w_15 <= "0000000000000000" ;
            w_16 <= "0000000000000000" ;
            w_17 <= "0000000000000000" ;
            w_18 <= "0000000000000000" ;
            w_19 <= "0000000000000000" ;
            w_20 <= "0000000000000000" ;
            outp <= "00000000000000000000" ;
      else

      if( count = "00001" ) then
            w_1 <= dec ;
      elsif( count = "00010" ) then
            w_2 <= dec ;
      elsif( count = "00011" ) then
            w_3 <= dec ;
      elsif( count = "00100" ) then
            w_4 <= dec ;
      elsif( count = "00101" ) then
            w_5 <= dec ;
```

```
elsif( count = "00110" ) then
      w_6 <= dec ;
elsif( count = "00111" ) then
      w_7 <= dec ;
elsif( count = "01000" ) then
      w_8 <= dec ;
elsif( count = "01001" ) then
      w_9 <= dec ;
elsif( count = "01010" ) then
      w_10 <= dec ;
elsif( count = "01011" ) then
      w_11 <= dec ;
elsif( count = "01100" ) then
      w_12 <= dec ;
elsif( count = "01101" ) then
      w_13 <= dec ;
elsif( count = "01110" ) then
      w_14 <= dec ;
elsif( count = "01111" ) then
      w_15 <= dec ;
elsif( count = "10000" ) then
      w_16 <= dec ;
elsif( count = "10001" ) then
      w_17 <= dec ;
elsif( count = "10010" ) then
      w_18 <= dec ;
elsif( count = "10011" ) then
      w_19 <= dec ;
end if ;
if ( count = "00000" ) then
      w_20 <= dec ;
      tmp := "0000" ;
      outp(0) <= tmp(0) ;
      tmp := find( w_20 , tmp ) ;
      outp(1) <= tmp(0) ;
      tmp := find( w_19 , tmp ) ;
      outp(2) <= tmp(0) ;
      tmp := find( w_18 , tmp ) ;
      outp(3) <= tmp(0) ;
      tmp := find( w_17 , tmp ) ;
      outp(4) <= tmp(0) ;
      tmp := find( w_16 , tmp ) ;
      outp(5) <= tmp(0) ;
      tmp := find( w_15 , tmp ) ;
      outp(6) <= tmp(0) ;
      tmp := find( w_14 , tmp ) ;
      outp(7) <= tmp(0) ;
      tmp := find( w_13 , tmp ) ;
      outp(8) <= tmp(0) ;
      tmp := find( w_12 , tmp ) ;
      outp(9) <= tmp(0) ;
      tmp := find( w_11 , tmp ) ;
      outp(10) <= tmp(0) ;
      tmp := find( w_10 , tmp ) ;
```

```
            outp(11) <= tmp(0) ;
            tmp := find( w_9 , tmp ) ;
            outp(12) <= tmp(0) ;
            tmp := find( w_8 , tmp ) ;
            outp(13) <= tmp(0) ;
            tmp := find( w_7 , tmp ) ;
            outp(14) <= tmp(0) ;
            tmp := find( w_6 , tmp ) ;
            outp(15) <= tmp(0) ;
            tmp := find( w_5 , tmp ) ;
            outp(16) <= tmp(0) ;
            tmp := find( w_4 , tmp ) ;
            outp(17) <= tmp(0) ;
            tmp := find( w_3 ,tmp ) ;
            outp(18) <= tmp(0) ;
            tmp := find( w_2 , tmp ) ;
            outp(19) <= tmp(0) ;

            end if ;
      end if ;
      end if ;
      end process ;
end low_power ;
```

```
-- Module: SHIFTREG
--
-- 20-bit shift register with parallel load and synchronous reset
-- this module infers multiplexer for the shifting purposes
--
-- Inputs:
--    1. CLK (1 bit)
--    2. RES (1-bit) (active high synchronous reset)
--       3. LOADSHFT(1-bit) (High for parallel load/ Low for shift right
operation)
--    4. DATAIN (20-bit for parallel load) (LSB is output first serially)
-- Outputs:
--    1. SOUT(1-bit serial output)
--

library IEEE, synopsys;
use IEEE.std_logic_1164.all;
use synopsys.attributes.all ;

entity shiftreg is
   port (clk, res, loadshft: in std_logic;
         datain: in std_logic_VECTOR(19 downto 0);
         sout: out std_logic;
         count : in std_logic_vector( 4 downto 0)
        );
end shiftreg;

architecture alg of shiftreg is

  signal data: std_logic_VECTOR(19 downto 0);

begin

  shiftreg: process (clk)

             begin
               if clk'event and clk='1' then
                 if res = '1' then
                    for i in 19 downto 0 loop
                        data(i) <= '0' ;
                    end loop ;
                 elsif loadshft = '1' then
                   data <= datain;
                 end if;


   case count is  --synopsys infer_mux

        when   "00011" => sout <= data(19);
        when   "00100" => sout <= data(18);
        when   "00101" => sout <= data(17);
        when   "00110" => sout <= data(16);
        when   "00111" => sout <= data(15);
        when   "01000" => sout <= data(14) ;
```

```
        when   "01001" => sout <= data(13);
        when   "01010" => sout <= data(12);
        when   "01011" => sout <= data(11);
        when   "01100" => sout <= data(10) ;
        when   "01101" => sout <= data(9) ;
        when   "01110" => sout <= data(8);
        when   "01111" => sout <= data(7);
        when   "10000" => sout <= data(6);
        when   "10001" => sout <= data(5) ;
        when   "10010" => sout <= data(4);
        when   "10011" => sout <= data(3);
        when   "10100" => sout <= data(2) ;
        when   "00000" => sout <= data(1) ;

          when others => sout <= data(0)  ;
         end case ;
       end if ;
  end process;

end alg;
```

```
-- Module: SHIFTREG
--
-- 20-bit shift register with parallel load and synchronous reset
-- this module is a normal shift register.

-- Inputs:
--    1. CLK (1 bit)
--    2. RES (1-bit) (active high synchronous reset)
--       3. LOADSHFT(1-bit) (High for parallel load/ Low for shift right
operation)
--    4. DATAIN (20-bit for parallel load) (LSB is output first serially)
-- Outputs:
--    1. SOUT(1-bit serial output)
--

library IEEE;
use IEEE.std_logic_1164.all;

entity shiftreg is
   port (clk, res, loadshft: in std_logic;
         datain: in std_logic_VECTOR(19 downto 0);
         sout: out std_logic
        );
end shiftreg;

architecture alg of shiftreg is


  signal data: std_logic_VECTOR(19 downto 0);

begin

  shiftreg: process (clk)

             begin
               if clk'event and clk='1' then
                 if res = '1' then
                    for i in 19 downto 0 loop
                        data(i) <= '0' ;
                    end loop ;
                 elsif loadshft = '1' then
                   data <= datain;
                 else
                   data <= data(18 downto 0) & '0' ;
                 end if;
               end if;
             end process;

  sout <= data(19);

end alg;
```

```
-- Module: COUNTER
--
-- 5-bit counter with synchronous reset.
--
-- Inputs:
--    1. CLK (1 bit) system clock
--    2. RES (1-bit)  system reset
-- Outputs:
--    1. ZERO (1-bit) (High if count = "00001" )
--    2. COUNT (5-bit counter state)
--

library IEEE;
use IEEE.std_logic_1164.all;

entity counter is
   port (clk, res: in std_logic;
         zero: out std_logic;
         count: out std_logic_VECTOR(4 downto 0)
        );
end counter;

architecture alg of counter is

  signal cntstate: std_logic_VECTOR(4 downto 0);

begin

  shiftreg: process (clk)

             begin
               if clk'event and clk='1' then
                 if res = '1' then
                   cntstate <= "00000";
                 else
                    cntstate(0) <= not cntstate(0);
                   if cntstate(0) = '1' then
                     cntstate(1) <= not cntstate(1);
                     if cntstate(1) = '1' then
                       cntstate(2) <= not cntstate(2);
                       if cntstate(2) = '1' then
                         cntstate(3) <= not cntstate(3);
                           if cntstate(3) = '1' then
                             cntstate(4) <= not cntstate(4) ;
                         end if;
                       end if;
                     end if;
                   end if;
                 end if;if ( (cntstate(4))  and
          (cntstate(1)) and (cntstate(0)))  = '1' then
                   cntstate <= "00000" ;
                end if ;
               end if;
```

```
            end process;

  count <= cntstate;
  zero <= ( not cntstate(4))  and ( not cntstate(3) ) and ( not  cntstate(2))
and
          ( not cntstate(1)) and ( cntstate(0));

end alg;
```

```
-- Module : enc_dec


-- This is the top level module for the encoder-decoder. It connects
-- all submodules together.
-- Inputs :
-- 1. clk ( 1-bit ) : system clock
-- 2. res ( 1-bit ) : system reset
-- 3. ctrl ( 3-bit ) : noise bits
-- 4. inp ( 1-bit ) : information input
-- 5. sce ( 1-bit ) : Scan enable

-- Output :
-- 1. eop ( 3-bit ) : encoded information
-- 2. sout ( 1-bit ) : decoded information


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


entity enc_dec is
      port( clk, res: in std_logic;
      ctrl : in std_logic_VECTOR(2 downto 0);
      inp: in std_logic ;
      eop : out std_logic_VECTOR(2 downto 0);
      sout: out std_logic ;
      sce : in std_logic
        );
end enc_dec ;

architecture struct of enc_dec is

      component encoder
            port( clk,inp,res : in std_logic ; outpp : out std_logic_VECTOR(
2 downto 0 ) );
         end component ;
      component op
            port( inp : in std_logic_VECTOR(2 downto 0);
                  opp : out std_logic_VECTOR(2 downto 0) );
      end component ;
      component noise
            port( nin :in std_logic_VECTOR(2 downto 0);
                  nout : out std_logic_VECTOR(2 downto 0);
                  nctrl : in std_logic_VECTOR(2 downto 0)
             );
      end component ;

       component ACS_0
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
```

```
      dec0,dec1 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )  ) ;
            end  component ;

      component ACS_1
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec2,dec3 : out std_logic;
      count : in std_logic_vector( 4 downto 0 )  ) ;
            end  component;

      component ACS_2
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec4,dec5 : out std_logic;
      count : in std_logic_vector( 4 downto 0 ) ) ;
            end  component;

      component ACS_3
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec6,dec7 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )) ;
            end  component;

      component ACS_4
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec8,dec9 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )) ;
            end component ;

      component ACS_5
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
      dec10,dec11 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )) ;
            end component ;

      component ACS_6
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 ) ;
```

```
      dec12,dec13 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 )) ;
            end component ;

      component ACS_7
      port( clk, res : in std_logic ; rc : in  std_logic_vector( 2 downto 0 )
;
      imetu ,imetl  : in std_logic_vector( 5 downto 0) ;
      ometu, ometl : out std_logic_vector( 5 downto 0 );
      dec14,dec15 : out std_logic ;
      count : in std_logic_vector( 4 downto 0 ) ) ;
            end component ;



      component fast_trace
      port( clk, res : in std_logic ; dec : in std_logic_vector( 15 downto 0
) ;
      count   :   in   std_logic_vector(   4   downto   0   )   ;   outp   :   out
std_logic_vector( 19 downto 0 ) ) ;
            end component ;

      component shiftreg
        port (clk, res, loadshft: in std_logic;
          datain: in std_logic_VECTOR(19 downto 0);
          sout: out std_logic
--    count : in std_logic_vector( 4 downto 0 )
           );
        end component;

      component counter
        port (clk, res: in std_logic;
          zero: out std_logic;
          count: out std_logic_VECTOR(4 downto 0)
         );
       end component;

 signal zero : std_logic ;
 signal rc , tonoise:   std_logic_vector( 2 downto 0 ) ;
 signal    m0,m1,m2,m3,m4,m5,m6,m7,m8,m9,m10,m11,m12,m13,    m14,    m15    :
std_logic_vector( 5 downto 0 ) ;
 signal dec : std_logic_vector( 15 downto 0 ) ;
 signal count : std_logic_VECTOR(4 downto 0) ;
 signal outp : std_logic_vector( 19 downto 0 )  ;
 begin

 ee: encoder
      Port Map(
            clk => clk ,
            inp => inp ,
            res => res ,
            outpp => tonoise
            ) ;
 buf : op
```

```
      Port Map (
            inp => tonoise ,
            opp => eop
            );
 noise_gen : noise

      Port Map (

            nin => tonoise ,
            nout => rc ,
            nctrl => ctrl
            ) ;

 mm0 :  ACS_0

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m8,
              imetl => m0,
              ometu => m1,
              ometl => m0,
              dec0 => dec(0) ,
              dec1 => dec(1),
              count => count
             ) ;

  mm1 :  ACS_1

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m9,
              imetl => m1,
              ometu => m3,
              ometl => m2,
              dec2 => dec(2) ,
              dec3 => dec(3),
              count => count

              ) ;

mm2 :  ACS_2

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m10,
              imetl => m2,
              ometu => m5,
              ometl => m4,
              dec4 => dec(4) ,
              dec5 => dec(5),
              count => count
```

```
            ) ;

mm3 :  ACS_3

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m11,
              imetl => m3,
              ometu => m7,
              ometl => m6,
              dec6 => dec(6) ,
              dec7 => dec(7),
              count => count

              ) ;

mm4 :  ACS_4

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m12,
              imetl => m4,
              ometu => m9,
              ometl => m8,
              dec8 => dec(8) ,
              dec9 => dec(9) ,
              count => count

              ) ;

mm5 :  ACS_5

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m13,
              imetl => m5,
              ometu =>m11 ,
              ometl => m10,
              dec10 => dec(10) ,
              dec11 => dec(11),
              count => count

              ) ;

mm6 :  ACS_6

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m14,
```

```
              imetl => m6,
              ometu => m13,
              ometl => m12,
              dec12 => dec(12) ,
              dec13 => dec(13) ,
              count => count

              ) ;

mm7 :  ACS_7

      port map( clk => clk ,
              res => res ,
              rc => rc ,
              imetu => m15,
              imetl => m7,
              ometu => m15,
              ometl => m14,
              dec14 => dec(14) ,
              dec15 => dec(15) ,
              count => count

              ) ;


fast: fast_trace
      port map(
            clk => clk,
            res => res ,
            dec => dec ,
            count => count ,
            outp => outp
            ) ;


 shift: shiftreg
           port map(
             clk => clk,
             res => res,
             loadshft => zero,
             datain => outp,
             sout => sout --,
            -- count => count
           );

  countdat: counter
             port map(
               clk => clk,
               res => res,
               zero => zero,
               count => count
             );

  end struct ;
```

```
configuration CFG_L of enc_dec is

for struct
    for ee: encoder
    end for ;
    for  buf : op
    end for ;
    for noise_gen : noise
    end for ;
    for mm0 : ACS_0
    end for ;
    for mm1: ACS_1
    end for ;
    for mm2 : ACS_2
    end for ;
    for mm3 : ACS_3
    end for ;
    for mm4 : ACS_4
    end for ;
    for mm5 : ACS_5
    end for ;
    for mm6 : ACS_6
    end for ;
    for mm7 : ACS_7
    end for ;

    for fast: fast_trace
    end for;
    for shift: shiftreg
    end for ;
    for countdat : counter
    end for;

    end for ;

end CFG_L ;
```

```
-- Module tb_top.


-- Test bench for the encoder decoder block.

Library IEEE;
use IEEE.std_logic_1164.all;
use WORK.arraylib.all;


entity tb_top is
end ;

architecture bench of tb_top is


signal clk, res , sout , sce , inp : std_logic ;
signal ctrl, eop : std_logic_vector( 2 downto 0 ) ;

component enc_dec
      port( clk, res: in std_logic;
            ctrl : in std_logic_VECTOR(2 downto 0);
            inp: in std_logic ;
            eop : out std_logic_VECTOR(2 downto 0);
            sout: out std_logic ;
            sce : in std_logic
              );
end component ;

begin

      UUT : enc_dec

            Port Map (

                  clk  => clk ,
                  res => res,
                  ctrl => ctrl,
                  inp => inp ,
                  eop  => eop,
                  sout => sout,
                  sce => sce
                  ) ;


      osc : process

      begin
            clk <= '0' ;
            wait for 100 ns ;
            clk <= '1' ;
            wait for 100 ns ;
      end process ;
```

```
      process
      begin

            inp <= '0',
                  '1' after 200 ns,
                  '1' after 400 ns,
                  '0' after 600 ns,
                  '1' after 800 ns,
                  '0' after 1000 ns,
                  '1' after 1200 ns,
                  '0' after 1400 ns,
                  '1' after 1600 ns,
                  '1' after 1800 ns,

                  '0' after 2000 ns,
                  '0' after 2200 ns,
                  '0' after 2400 ns ,
                  '1' after 2600 ns,
                  '1' after 2800 ns,
                  '1' after 3000 ns ,
                  '0' after 3200 ns,
                  '1' after 4200 ns ,
                  '0' after 4600 ns ,
                  '1' after 5200 ns,
                  '0' after 6400 ns ,
                  '1' after 8600 ns ,
                  '0' after 9200 ns ,
                  '1' after 10400 ns ,
                  '0' after 11000 ns ;
             --   '1' after 13200 ns ,
             --   '0' after 14000 ns ;
             res <= '1' , '0' after 150 ns ;

             ctrl <= "000" ;
             sce <= '0' ;
             wait ;
      end process ;
end bench ;

configuration CFG_TB of tb_top is

      for bench
            for UUT : enc_dec
            end for ;
      end for ;
end ;
```

Vita

Samir Ranpara was born on 25th October 1974 and grew up in the Western part of India. In June 1991 he obtained his diploma in higher secondary school. He received his Bachelor of Engineering degree in Computer Engineering from Sardar Patel University, Gujarat in October 1995. After completing his undergraduate studies he worked as a Lecturer in Electrical and Computer Engineering department at Birla Vishwakarma Mahavidyalay, V.V.Nagar. In August 1997, he enrolled at Virginia Polytechnic Institute and State University to pursue his Masters in Computer Engineering. At Virginia Tech his area of concentration was Low-power chip design. His immediate plain is to begin work as a Component Designer with Intel Corporation, Portland, Oregon.