Low Power Register Exchange Viterbi Decoder for Wireless Applications

by

Dalia Abdel-Wahed Fouad El-Dib

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

Abstract

Since the invention of wireless telegraphy by Marconi in 1897, wireless technology has not only been enhanced, but also has become an integral part of our everyday lives. The first wireless mobile phone appeared around 1980. It was based on first generation analog technology that involved the use of Frequency Division Multiple Access (FDMA) techniques. Ten years later, second generation (2G) mobiles were dependent on Time Division Multiple Access (TDMA) techniques and Code Division Multiple Access (CDMA) techniques. Nowadays, third generation (3G) mobile systems depend on CDMA techniques to satisfy the need for faster, and more capacious data transmission in mobile wireless networks. Wideband CDMA (WCDMA) has become the major 3G air interface in the world. WCDMA employs convolutional encoding to encode voice and MPEG4 applications in the baseband transmitter at a maximum frequency of $2Mbps$. To decode convolutional codes, Andrew Viterbi invented the Viterbi Decoder (VD) in 1967. In 2G mobile terminals, the VD consumes approximately one third of the power consumption of a baseband mobile transceiver. Thus, in 3G mobile systems, it is essential to reduce the power consumption of the VD.

Conceptually, the Register Exchange (RE) method is simpler and faster than the Trace Back (TB) method for implementing the VD. However, in the RE method, each bit in the memory must be read and rewritten for each bit of information that is decoded. Therefore, the RE method is not appropriate for decoders with long constraint lengths. Although researchers have focused on implementing and optimizing the TB method, the RE method is focused on and modified in this thesis to reduce the RE method's power consumption.

This thesis proposes a novel modified RE method by adopting a *pointer* concept for implementing the survivor memory unit (SMU) of the VD. A pointer is assigned to each register or memory location. The contents of the pointer which points to one register is altered to point to a second register, instead of copying the contents of the first register to the second. When the pointer concept is applied to the RE's SMU implementation (modified RE), there is no need to copy the contents of the SMU and rewrite them, but one row of memory is still needed for each state of the VD. Thus, the VDs in CDMA systems require 256 rows of memory. Applying the pointer concept reduces the VD's power consumption by 20 percent as estimated by the VHDL synthesis tool and by the new power reduction estimation that is introduced in this work. The coding gain for the modified RE method is $2.6dB$ at an SNR of approximately 10^{-3}.

Furthermore, a novel zero-memory implementation for the modified RE method is proposed. If the initial state of the convolutional encoder is known, the entire

SMU of the modified RE VD is reduced to only one row. Because the decoded data is generated in the required order, even this row of memory is dispensable. The zero-memory architecture is called the MemoryLess Viterbi Decoder (MLVD), and reduces the power consumption by approximately 50 percent. A prototype of the MLVD with a one third convolutional code rate and a constraint length of nine is mapped into a Xilinx 2V6000 chip, operating at 25 MHz with a decoding throughput of more than $3Mbps$ and a latency of two data bits.

The other problem of the VD which is addressed in this thesis is the Add Compare Select Unit (ACSU) which is composed of 128 butterfly ACS modules. The ACSU's high parallelism has been previously solved by using a bit serial implementation. The 8-bit First Input First Output (FIFO) register, needed for the storage of each path metric (PM), is at the heart of the single bit serial ACS butterfly module. A new, simply controlled shift register is designed at the circuit level and integrated into the ACS module. A chip for the new module is also fabricated.

Acknowledgments

Foremost, all thanks are due to Almighty, the most merciful God. God blessed me and gave me the strength to finish this study while taking care of my family.

Prof. M.I. Elmasry, my thesis supervisor, has guided my work insightfully and enriched my research with his fruitful experience. Special thanks for trusting me and providing such a flexible working environment.

My parents, though not with me in Canada, have supported me with their love, care and prayers. Owing them my success, I will never be able to thank them enough. To them, I would like to express my sincere thanks.

My husband, Sherif Sadek, deserves a special acknowledgement. He has always encouraged me and stood by my side taking care of our little children (Hassan and Hussein). I would never have completed this work without his support. Also my dear and only brother, Mohamed El-Dib, has supported me unconditionally. Thanks to him and all my family members and friends.

Our VLSI system administrator, Phil Regier, has provided all of us with a very organized lab, and is always willing to help out with any technical problem. I really appreciate his dedication and experience, which helped me a lot throughout this research work.

CMC has provided me with a fabrication grant to fabricate my chip and provided technical support during the design, fabrication and test of the chip. Special thanks to Mariusz Jarosz and Hui Xu from CMC. Finally, I would like to acknowledge the financial support provided by the Egyptian government. OGSST and Professor Elmasry's NSERC Research Grants have supported my work for the last year.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	**Details**
1G	First Generation
2G	Second Generation
3G	Third Generation
ACS	Add Compare Select
ACSTOSM	Add Compare Select TO Survivor Memory
ACSU	Add Compare Select Unit
ADC	Analog to Digital Converter
AMPS	Advanced Mobile Phone Service
ASIC	Application Specific Integrated Circuits
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BM	Branch Metric
BMU	Branch Metric Unit
CDMA	Code Division Multiple Access
CMC	Canadian Microelectronics Corporation
CPL	Complementary Pass Gate Logic
CRC	Cyclic Redundancy Code
CS	Carry Save
CTIA	Cellular Telecommunications Industry Association
DSP	Digital Signal Processor
FDMA	Frequency Division Mutliple Access
GSM	Global System for Mobile Communications
IF	Intermediate Frequency
IPR	Intellectual Property Rights
ITU	International Telecommunication Union
LFSR	Linear Feedback Shift Register
LSB	Least Significant Bit

Abbreviation	**Details**
MLVD	MemoryLess Viterbi Decoder
MSB	Most Significant Bit
OVSF	Orthogonal Variable Spreading Factor
PCCC	Parallel Concatenated Convolutional Code
PDC	Personal Digital Cellular
PM	Path Metric
PMM	Path Metric Memory
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RE	Register Exchange
RF	Radio Frequency
RTL	Register-Transfer-Level
SMU	Survivor Memory Unit
TACS	Total Access Communication System
TB	Trace Back
TDMA	Time Division Multiple Access
TTA	Telecommunication Technology Association
UPR	Users' Performance Requirements
VA	Viterbi Algorithm
VD	Viterbi Decoder
VLSI	Very Large Scale Integrated
WCDMA	Wideband Code Division Multiple Access

Chapter 1

Introduction

From the advent of integrated circuits, circuit designers have been concerned with increasing circuit performance and minimizing the design area. Although low power microelectronics can be traced back to the invention of the transistor in 1947, it was an afterthought, not a design criterion. Today, a major creative challenge facing VLSI designers is to design products which consume minimal power on both the circuit level and the system level. This will significantly increase the battery life of portable devices, a necessity for the consumer who is *on the go* and requires a reliable and efficient wireless environment. Therefore, reducing the power consumption of 3G mobile terminals is the main objective of this thesis. The motivations for low power design are detailed in the next section.

1.1 Motivation for Low Power

The motivation for low power electronics has been derived from the following needs:

1. To extend the battery lifetime for portable devices such as laptops, cellular phones, electronic organizers, pacemakers, and hearing aids. This is the earliest and the most demanding motivation for low power design.

2. To increase the packing density which imposes severe restrictions on the power dissipation density in order to further enhance the speed of high performance systems. Low power reduces the costs that are associated with packaging, cooling, and fans, in addition to keeping a certain level of system reliability, which is threatened by excessive power dissipation. This is the most recent motivation for low power design.

3. To curb the increasing power consumption of desktop and deskside systems, where a competitive cost to performance ratio demands low power operations to reduce the power supply and cooling costs. In addition, the fact that 50 percent of office power is used by PCs suggests that low power design is essential for reducing the overall power budget. This is the broadest need for low power design.

A brief introduction to 3G mobile terminals is given in the next section.

1.2 Third Generation (3G) Mobile Systems

Since the introduction of cellular networks in the early 1980s, mobile technology has evolved very rapidly. In the 1990s 1G analog wireless systems were replaced by 2G digital technologies that delivered significant improvements in capacity, voice quality, and spectral efficiency. Most importantly, 2G digital technologies laid the foundation for the value added services, including data, which will continue to be enhanced into the future. Now, the 3G era that promises further improved network capacity provides high speed packet data, simultaneous data, and voice and realtime multimedia services.

The benefits of 3G to end users translate to the availability of a full suite of innovative services such as mobile access to intranets or the Internet, videoconferencing, and sending and receiving high quality images. 3G terminals and devices can provide the user interface for experiencing these enhanced services. CDMA2000 and WCDMA are two technologies that have been approved for 3G. Since the world's first commercial 3G network was based on WCDMA, the VD, which is part of the WCDMA mobile terminal, is investigated in this thesis to reduce the VD's power consumption.

1.3 Thesis Overview

Chapter 2 deals with the sources of power dissipation and the various low power design techniques. In Chapter 3, an overview of CDMA mobile systems and the application of the Viterbi Algorithm (VA) to decode the convolutional codes in such systems is presented. Chapter 4 introduces the pointer concept and its novel application to the RE implementation of a VD. Chapter 5 describes the fabricated chip which implements an ACS module, and Chapter 6 presents the ultra low power VD architecture, the MLVD. Finally, the conclusions and recommendations for future work are outlined in Chapter 7.

Chapter 2

Low Power Digital VLSI Design

2.1 Energy Consumption versus Power Consumption

Energy consumption is different from power consumption. If for example, for a CMOS gate we reduce its clock rate f, its power consumption will be reduced by the same proportion. However, its energy consumption will still be the same as shown in Figure 2.1 [1]. Only, the time required to complete the computation with low clock rate, will be increased. Therefore, after the computation is completed the battery will be just as dead as if the computation had been performed at high clock rate. Thus, for battery-operated devices low energy design is more important than low power design. However, the conventional term low power is used throughout the thesis to mean that low energy is the target [2].

2.2 Sources of Power Dissipation

In the 1980s, an increase in the power dissipation of Application Specific Integrated Circuits (ASICs) caused a worldwide shift from NPN bipolar and NMOS technologies to CMOS technology. CMOS significantly reduces the average power dissipation, but it is the demand for higher packing densities and higher operating frequencies in microelectronics that keeps low power design as the primary requirement. In order to minimize the power dissipation of a CMOS circuit, the main sources for the power dissipation of CMOS devices must be identified. There are three major power dissipation components within the CMOS inverter switching power, leakage power, and short circuit power [3].

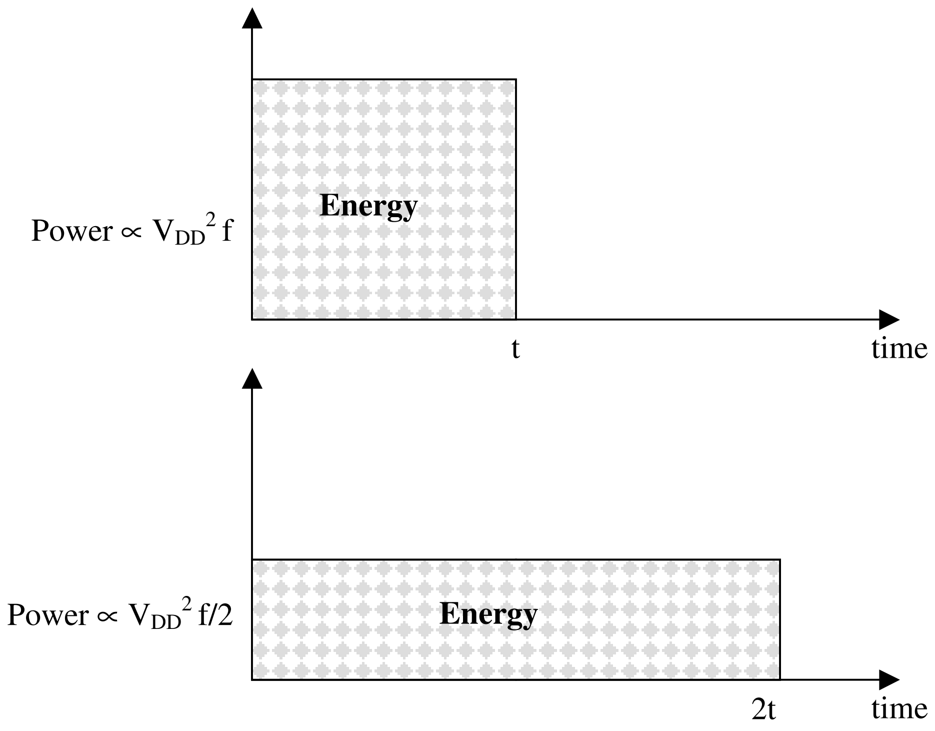


Figure 2.1: Energy consumption versus power consumption

2.2.1 Switching Power

The most dominant source of power dissipation in digital circuits is dynamic power dissipation. It describes the power required to charge and discharge the load capacitance, C_L, at the switching activity α, and is expressed as.

$$P_{dync} = \alpha C_L V_{DD} V_{swing} f. \tag{2.1}$$

Equation (2.1) shows that the dynamic power is decreased, by reducing the supply voltage, but this also reduces the speed. P_{dyn} is reduced also, by reducing C_L, but it is noted that scaling down the circuit does not necessarily reduce the capacitance by the same ratio. This occurs because the wiring capacitance does not scale well with submicron technologies. Thus, this source of power dissipation is usually reduced at the algorithmic, architectural, and Register-Transfer-Level (RTL) levels. A very important part of dynamic power is the glitching power, which is discussed as follows.

Glitching Power

In a static logic gate, the output or internal nodes can switch before the correct logical value is reached. Figure 2.2 depicts the logic transitions for a cascaded configuration. When the input is switched from 100 to 111, the output should remain high, but due to the delay associated with each gate, a glitch is added to the total switching of the circuit. The glitch is sometimes called a spurious transition.

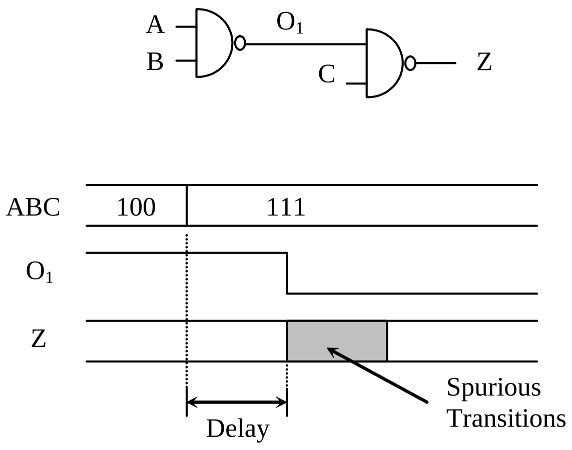


Figure 2.2: Glitching in static CMOS gates

If the power dissipation that is due to spurious transitions is important, then the logic of the circuit should be redesigned to avoid the cascaded implementation and to balance the delay paths, particularly on highly loaded nodes.

2.2.2 Short Circuit Power

Short circuit power is due to finite rise and fall times of the input signal. For a short period of time, there will be a conducive path open between V_{DD} and GND, because both NMOS and PMOS devices are ON. Such a path never exists in dynamic circuits, as precharge and evaluate transistors should never be ON simultaneously as this would lead to incorrect evaluation [4]. Short circuit power can be expressed as:

$$P_{sc} = I_{sc}V_{DD}, \tag{2.2}$$

where I_{sc} is the short circuit current. This power is limited by giving the output signal and the input signal equal rise and fall times. As a result, the P_{sc} is less than

20 percent of the total power. For submicron devices, where the V_{DD} approaches $(V_{Tn} + V_{Tp})$ or is less, the P_{sc} can be eliminated, because both devices cannot conduct simultaneously [2].

2.2.3 Leakage Power

The leakage power can be expressed as:

$$P_{leakage} = I_{leakage} V_{DD}, \tag{2.3}$$

where $I_{leakage}$ is the total leakage current in a CMOS circuit [3]. $I_{leakage}$ is caused by six short channel leakage mechanisms [5] the reverse bias pn junction leakage, subthreshold leakage, oxide tunneling current, gate current due to hot carrier injection, gate induced drain leakage, and the channel punchthrough current. For deep submicron devices with low supply voltages and low threshold voltages, the subthreshold leakage is the dominant leakage mechanism. As CMOS process advances to the sub-100nm regime, the gate oxide thickness of sub-20$\mathring{A}$ prevails in CMOS processes [3]. Gate leakage may become the dominant factor for sub-100nm generations unless new solutions emerge [6].

2.2.4 Static Power

Some CMOS circuit families such as the pseudo NMOS dissipate static power when the output is at logic 0 as illustrated in Figure 2.3 for a pseudo-NMOS inverter.

2.3 Low Power Design Techniques

Power dissipation is minimized at all design levels. At the circuit level, for example, the power is minimized by aggressively scaling down the supply voltage. It is also essential to minimize the switching activity. This can be accomplished at the logic, RTL, architectural, or algorithmic levels. The following is a brief discussion of some low power methodologies that are applied at various levels, including the system, algorithm, architecture, logic, circuit, and physical levels [2] [4].

2.3.1 System Level Optimization

Usually, low power design at the system level is realized by utilizing lower system clocks, in addition to using a high level integration. The high level integration

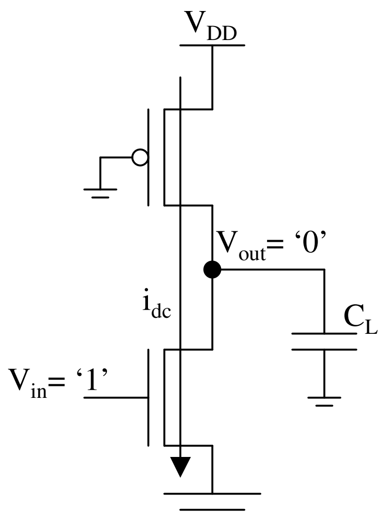


Figure 2.3: DC current in a Pseudo-NMOS inverter

includes the integration of off-chip memories and digital and analog peripherals. Other ways to reduce power at system level include using dynamic V_{DD} control according to system workload [3].

2.3.2 Algorithm Level Optimization

Secondly, at the algorithm level, the main concern is to minimize the number of operations, and thus, the number of hardware resources, or to minimize the switching activity by data coding.

Minimizing the Number of Operations

Minimizing the number of operations is crucial in minimizing power dissipation. However, different types of operations do not consume the same amount of power. The number of operations at the algorithm level can be decreased by data coding. The use of an appropriate coding technique for the signals, rather than a direct binary code, can reduce power by reducing the temporal bit transition activity. The following sections provide some examples of different data representations which are used to reduce power dissipation [4].

One Hot Coding: In regards to an inter-chip communication problem, where n-bit data words will be transmitted, a connection of $m = 2^n$ wires can be used. The n-bit data word is encoded for transmission by placing a 1 on the ith wire and 0 on the remaining m-1 wires. One hot encoding requires an exponentially increasing number of wires (2^n), but the coding does guarantee that precisely one $0 \rightarrow 1$ and one $1 \rightarrow 0$ bit transition occur when a different data word is sent.
Gray Coding: Gray coding, a coding style in which two successive numbers differ by one bit only, is most useful when the coding is sequential, and highly correlated data is transmitted over a bus. The number of transitions for the binary representation is twice the number of transitions for the gray coding. Table 2.1 displays the binary representation and gray code representation for the decimal numbers 0 to 15 [4].
Two's Complement vs. Sign Magnitude: In a sign magnitude representation, only one bit is allocated for the sign, and the others are designated for the magnitude. When the signals, transmitted over a bus, switch frequently around zero and do not utilize the entire bit-width, the sign magnitude representation reduces power significantly. This occurs because the sign bit representation prevents the sign extension of the most significant bits from switching each time, from positive to negative or vice versa, which is the case with the two's complement representation.

2.3.3 Architecture Level Optimization

Architecture level optimization is the third way to reduce power dissipation. At this level, a set of primitives such as adders, multipliers, ROMs and register files are adopted. Some techniques that have been used are examined in the following.

Parallel Processing

Usually, parallel processing trades area for reduced power dissipation. Parallelism can be applied to the same input to reduce the operating frequency, or parallelism can be applied to different inputs which are using the same computational resource. Both cases are explained next.
Parallel Processing for Reduced Operating Frequency: Typically, N processors can be parallelized by duplication with each processor running with a slower clock (by a factor N). Therefore, the power supply voltage can be aggressively decreased to meet a delay, which is almost equal to the reference delay, divided by N [2]. For the simple multiplier of Figure 2.4 [2], it is possible to maintain the

Table 2.1: Binary and gray code representations

Decimal Value	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

throughput while the power supply voltage is reduced by using the illustrated parallel architecture. This results in an estimated 63 percent power reduction, when the voltage drops from 5v to 2.9v. However, the area doubles [7].
Parallel Processing to Avoid Resource Sharing: One strategy for implementing a signal processing algorithm is the direct mapping approach, where there is a one-to-one correspondence between the operations in the signal flow graph and the operators in the final implementation. This strategy is also called a fully parallel implementation. In time-multiplexed architectures, the same functional hardware is used to execute multiple operations. Often, time-multiplexed architectures are employed, when there are area constraints or high throughput is not the goal. Figure 2.5 [8] demonstrates the parallel and time-multiplexed architectures of the simple addition (C=A+B). The time-multiplexed adder switches to a 50 percent extra capacitance, which results in a 50 percent increase in power dissipation.

In other words, the fully parallel architecture results in a 33 percent reduction in power dissipation due to the lower switching activity.

Pipelining

Usually, if a processor is pipelined with N stages of registers, then the delay between the pipeline stages is reduced by almost a factor of N. Thus, it is possible to maintain the operating frequency, while the supply voltage is aggressively scaled down. A pipelined architecture for a simple multiplier is given in Figure 2.6 [2]. Moreover, there is a 63 percent power reduction if the voltage drops from 5v to 2.9v. Consequently, the combination of pipelining and parallel processing results in large power savings.

Ordering of Input Signals

Switching activity can be reduced by optimizing the ordering of the operations in a design [2]. Figure 2.7 exemplifies the multiplication of the variable M with a constant, and the multiplication is decomposed into M+(M>>7)+(M>>8). The figure indicates two possible topologies. The second implementation switches 30 percent less capacitance than the first one, because the shifting to the right represents a scaling operation, which reduces the dynamic range of the signal.

2.3.4 Logic Level Optimization

The fourth way to reduce the power dissipation is optimization at the logic level.

Technology Mapping

Here, the reduction of the power dissipation during the logic synthesis is based on choosing a logic to minimize switching, positioning registers through retiming to reduce the glitching activity, and choosing gates from a library which reduces the switching.

Logic Level Power Down

For each logic function, there are several logic structures for implementing the function, but give different power results due to different switching activities and/or different operating times. For example, an adder has several topologies, including the ripple carry, carry lookahead, carry skip, carry select, and conditional sum, each with a different switching activity. For low power purposes only, the ripple carry is chosen. Also, the power dissipation can be reduced at the logic level at the expense of some additional control circuitry. For example, in a comparator with the inputs, A and B, an additional circuitry compares A[N-1] with B[N-1] (N is the number of bits). The result of this stage is used to decide if the comparison of A[N-1:0] and B[N-1:0] is necessary or not. The output of the most significant bit comparator provides a gated clock signal to the comparator of the remaining bits. The small overhead control circuitry provides significant power reductions [4].

2.3.5 Circuit Level Optimization

Circuit level optimization is the fifth way to reduce the power dissipation. Whereas scaling down of the power supply voltage V_{DD} is one of the most effective ways to achieve low power design, the threshold voltages of transistors also need to be scaled down to meet performance requirement. However the lowering of the transistor threshold voltage leads to the exponential growth of the subthreshold leakage current and the subthreshold leakage power dissipation [3]. In addition, power supply voltage reduction reduces the fault tolerance. A recent study [9] concludes that the use of minimally sized devices is the best option to save power and to increase fault tolerance.

Also, choosing the circuit topology affects the power dissipation. There are several options for the use of circuit topology to implement the various logic functions [2] [4]. Some of these options are discussed next.

Dynamic Logic vs. Static Logic

The choice of using static or dynamic logic depends on many criteria. Dynamic logic reduces the switching activity by reducing hazards, by eliminating short circuit dissipation, and by reducing parasitic node capacitances. Static logic has advantages because no precharge operation nor charge sharing exists.

Pass Transistor Logic vs. Conventional CMOS Logic

Since fewer transistors are required, pass gate logic is attractive. It reduces the physical capacitance and power dissipation. However, a Complementary Pass Gate

Logic (CPL) implementation has two fundamental problems. First, the threshold drop across single channel pass transistors requires an increase in the minimal supply voltage, V_{DD}. Secondly, because the high input voltage level at the regenerative inverters is not the V_{DD}, the PMOS device in the inverter is not fully turned OFF. As a result, direct path static power dissipation is significant. To solve these problems, reducing the threshold voltage has proven to be effective.

Synchronous vs. Asynchronous

The clock tree of synchronous circuits consumes significant power and is eliminated in asynchronous ones. Local handshake signals, which typically require less power than the clock tree are used instead. Thus, asynchronous circuits typically switch (and consume switching power) only when required or when their inputs change [10].

2.3.6 Physical Level Optimization

The last way to reduce power dissipation is at the physical level [2] [4].

Layout Optimization

The layout of a module can be rigid or flexible. A rigid module is restored as a layout in the design library and its dimensions and power dissipation are fixed. A flexible module does not have a predetermined implementation and is parameterizable with parameters such as shape and power dissipations for flexible implementations.

Place and Route

The placement and routing of standard cells, which are already laid out and well characterized, affects power dissipation. To minimize power, the modules should be placed so that the switching activity capacitance product is minimized and routed so that the high activity wires are kept short.

The low power design methodologies discussed in this chapter are very important for the design of mobile terminals. Because CDMA is the proposed digital communication technique for 3G wireless systems, CDMA wireless systems will be reviewed in Chapter 3.

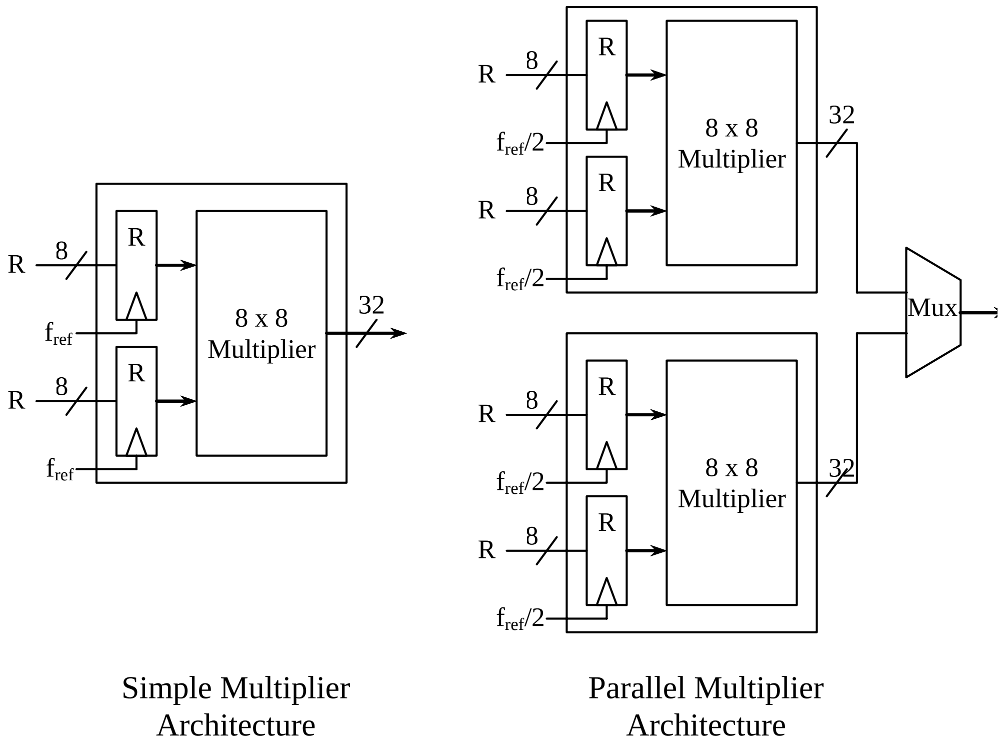

R
8
R
f_{ref}
8 x 8 Multiplier
32
Simple Multiplier Architecture
$f_{ref}/2$
Mux
Parallel Multiplier Architecture

Figure 2.4: Simple and parallel architectures of an 8 x 8 multiplier

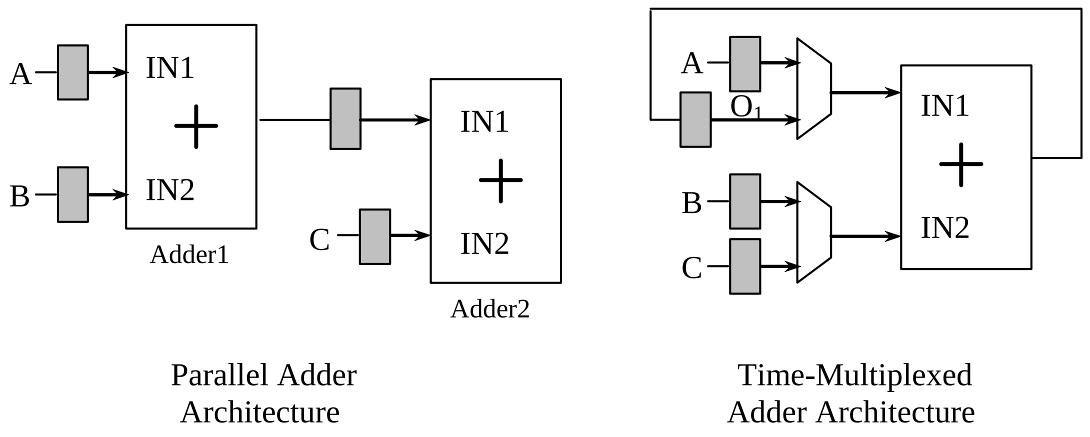

A
B
IN1
IN2
Adder1
C
IN1
IN2
Adder2
Parallel Adder Architecture
A
O_1
B
C
IN1
IN2
Time-Multiplexed Adder Architecture

Figure 2.5: Parallel and time-multiplexed architectures for a simple adder

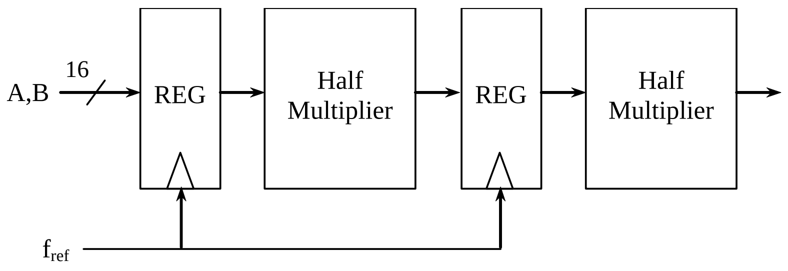


Figure 2.6: Pipelined architecture for the multiplier

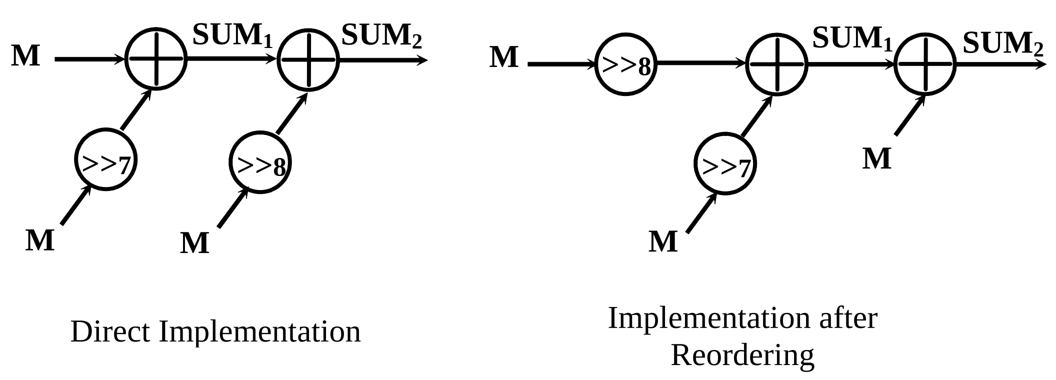


Figure 2.7: Reducing activity by reordering inputs

Chapter 3

CDMA Wireless Systems and Viterbi Decoders (VDs)

CDMA, a form of the spread spectrum technique, belongs to the family of digital communication techniques that have been used in military applications for many years. The core principle of spread spectrum is the use of noise-like carrier waves, and, as the name implies, bandwidths that are much wider than those that are required for simple point-to-point communication at the same data rate. Originally, there were two motivations for the development of CDMA: either to resist the efforts of the enemy to jam communications, or to hide the fact that communications were even taking place. The CDMA spread spectrum technology was introduced in the 2G of mobile systems, and then retained for the 3G of such systems.

This chapter outlines the background of the CDMA technique, including a brief history of the successive mobile generations. After a block diagram for the Wideband CDMA (WCDMA) modem is presented, a discussion of convolutional codes and the Viterbi decoding algorithm is presented. Finally, the different parts of the VD are outlined.

3.1 Multiple Access Comparison

There are three types of multiple access techniques used in communication systems: FDMA, TDMA, and CDMA. The following section discusses the different multiple access techniques used in the successive generations of mobile systems.

3.1.1 Frequency Division Multiple Access (FDMA)

In the FDMA standard analog cellular system, each user is assigned a discrete slice of the RF spectrum. FDMA permits only one user per channel so that the user can use the channel 100 percent of the time (Figure 3.1). Therefore, only the frequency dimension is adopted to define the channels. The principal disadvantage of FDMA is that it involves narrow band filters. They are not realizable in Very Large Scale Integrated (VLSI) circuits, which can lead to a high cost floor for terminals, even with a high production volume [11].

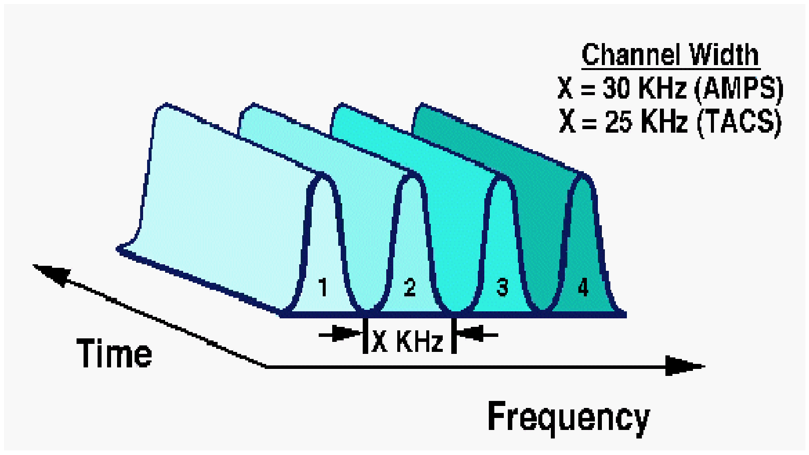


Figure 3.1: FDMA waveform

3.1.2 Time Division Multiple Access (TDMA)

The key point about TDMA is that the user is still assigned a discrete slice of RF spectrum, but multiple users now share the RF carrier on a time slot basis, alternately using the RF carrier. Frequency division is still employed, but the carriers are now subdivided into a number of time slots per carrier as seen in Figure 3.2. TDMA has the potential to be implemented in VLSI without narrow band filtering. However, particularly for mobile handsets on the reverse link, TDMA demands high peak power in the transmission mode which reduces battery life. A user is assigned a particular time slot in a carrier, and can send or receive information only at those times. This is true whether or not the other time slots are being used. Information flow is not continuous for any user; instead, it is sent and received in bursts, which are reassembled at the receiving end, and appear to provide a continuous flow because the process is so fast.

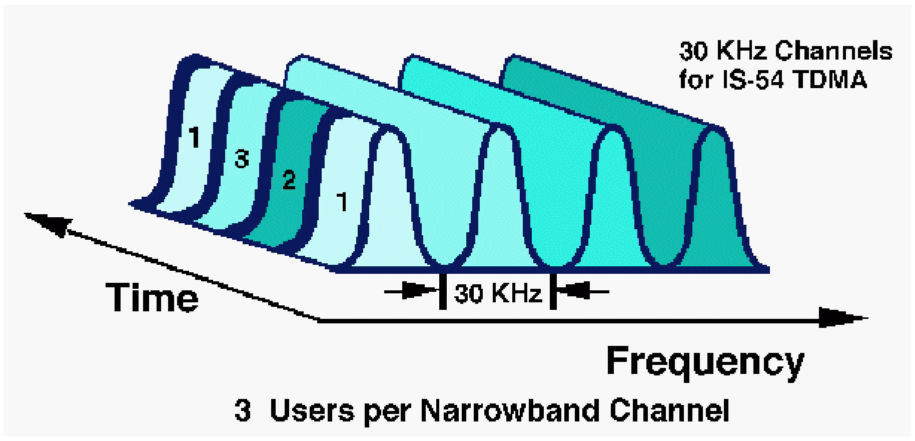

30 KHz Channels for IS-54 TDMA
1
3
2
1
Time
30 KHz
Frequency
3 Users per Narrowband Channel

Figure 3.2: TDMA waveform

3.1.3 Code Division Multiple Access (CDMA)

With CDMA, each signal consists of a different pseudo random binary sequence that modulates the carrier, spreading the spectrum of the waveform. A large number of CDMA signals share the same frequency spectrum, reflected in Figure 3.3. If CDMA is viewed in either the frequency or the time domain, the multiple access signals appear to be on top of each other. The signals are separated in the receivers by using a correlator which accepts signal energy only from the selected binary sequence, and despreads its spectrum. The other users' signals, whose codes do not match, are not despread in the bandwidth. As a result, the other signals contribute only to the noise and represent a self interference that is generated by the system. It is important to note that TDMA and CDMA use FDMA to divide the frequency band into smaller frequency channels which are then divided in a time or code division fashion

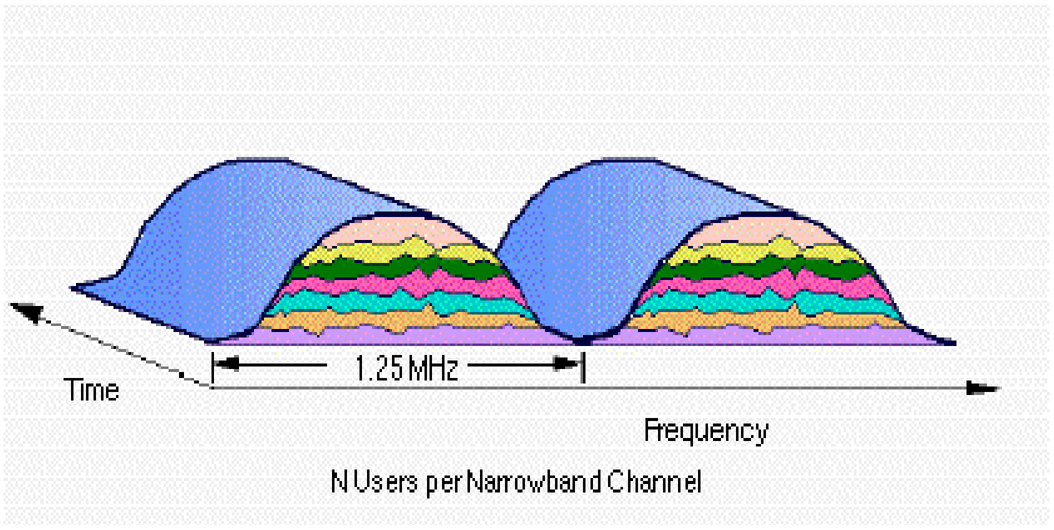


Figure 3.3: CDMA waveform

3.2 Brief History

Traditional analog cellular systems, representing the 1G cellular systems, adopt FDMA [12]. These systems are based on the Advanced Mobile Phone Service (AMPS) and Total Access Communication System (TACS) standards. In September 1988, the Cellular Telecommunications Industry Association (CTIA) released the Users' Performance Requirements (UPR) document, which specified the cellular carriers' requirements for the next (second) generation of cellular technology [13]. These requirements for digital technology include:

- A 10 fold increase over analog system capacity.
- A long life and adequate growth of 2G technology.
- An ability to introduce new features.
- Quality improvements.
- Privacy.
- Transition and compatibility with existing analog system.
- Availability and reasonable costs for dual mode radios and cells.

- Cellular open network architecture.

A common multiple access method for new digital cellular systems, representing the second generation cellular systems, is TDMA. Its digital standards include the North American Digital Cellular (known by its standard number, IS-54), the Global System for Mobile Communications (GSM), and the Personal Digital Cellular (PDC) [12].

In 1978, Cooper and Nettleton suggested the use of spread spectrum for cellular applications [13][14]. During the 1980s, Qualcomm investigated DS-CDMA techniques, which finally led to the commercialization of cellular spread spectrum communications in the form of the narrowband CDMA IS-95 standard in July 1993 [15]. In 1999, there were 250 million users in the global mobile communications market; 30 million of them were CDMA customers (12 percent) [11]. A recent analysis of wireless platform performance by TRAC found that CDMA outperforms other digital and analog technologies in every aspect, including signal quality, security, power consumption, and reliability. Also, TRAC found CDMA to be superior in signal security and voice quality over other digital air interface standards [11]. But by 2003, CDMA had a world market share of just 15 percent [16]. Emerging demands for higher rate data services and for better spectrum efficiency necessitated the development of 3G mobile radio systems. In the International Telecommunication Union (ITU), 3G networks are called IMT-2000 (UMTS in Europe) [15]. The objectives for the IMT-2000 air interface can be summarized as follows:

- Full coverage and mobility for 144 Kb/s, preferably 384 Kb/s.
- Limited coverage and mobility for 2Mb/s.
- High spectrum efficiency compared to that of existing systems.
- High flexibility to introduce new services.

During the 1990s, researchers focused on wideband CDMA technologies with a bandwidth of $5MHz$ or more. As a result, several trial systems have been built and tested. In Europe and Japan, WCDMA has been proposed to avoid IS-95 Intellectual Property Rights (IPR). In Korea, the Telecommunication Technology Association I and II (TTA I and TTA II) schemes have been adopted. In North America, cdma2000 uses a CDMA air-interface, based on the existing IS-95 standard, to provide wireline quality voice service, and high speed data services ranging from 144kbps for mobile users to 2Mbps for stationary ones. Cdma2000 is considered by the ITU to be 3G compatible, but this requires the implementation of a version of the technology, called cdma2000 3xRTT. Since only cdma2000 1xRTT, a narrowband version of the technology, has been implemented in practice, the cdma2000 systems fall short of the requirements for the 3G standards set by the ITU. The practical implementations of cdma2000 are comparable to those of GPRS and EDGE. Consequently, the 1xEV-DV, which is not commercially available, is really the 3G equivalent to WCDMA as it utilizes a new spectrum [16]. This important difference means that the world's first commercial 3G network was, in fact, based on WCDMA, which will be examined closer in the next section. Table 3.1 summarizes the evolutionary path of mobile generations.

Table 3.1: Evolutionary path of mobile generations

Specification	**2nd Generation**	**2.5 Generation**	**3rd Generation**
Data Rate	9.6kbps	115kbps 384kbps	2Mbps
Applications Drivers	Voice SMS	Internet (WAP) Data	Multimedia Interactivity
Standards	GSM CDMA(IS-95A)	GPRS/EDGE CDMA2000 1X	UMTS(WCDMA) CDMA2000 1xEV
Products	Phones	Smart Phones PDAs	Multimedia Laptops

3.3 WCDMA Transceiver

Figure 3.4 depicts a general block diagram [13] for a CDMA mobile transceiver. Traditionally, Radio Frequency (RF) functions and Intermediate Frequency (IF) functions have been realized by analog technology, and the baseband has been realizeable by digital technology [14]. Nowadays, the analog digital border is infringing on the IF section.

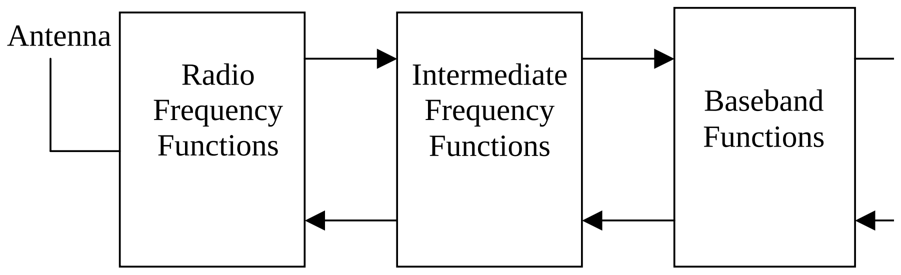


Figure 3.4: Block diagram of a wideband CDMA transceiver

3.3.1 WCDMA Transmitter

The digital architecture of a mobile transmitter that supports WCDMA is presented in Figure 3.5 [17]. To conform to the WCDMA standard, Cyclic Redundancy Code

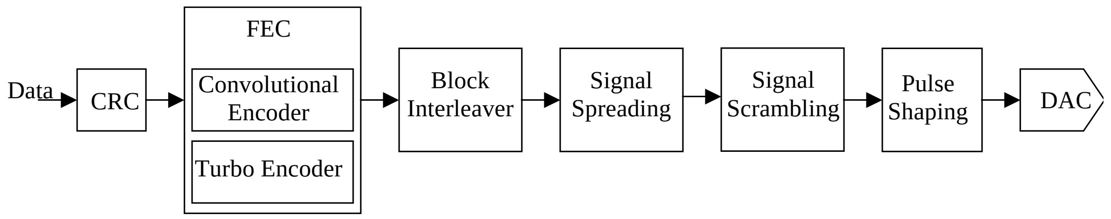


Figure 3.5: Block diagram of a WCDMA digital transmitter

(CRC) bits are added for error detection, and error correction bits are added for channel coding. The standard defines two encoding schemes to support a different quality of services. For voice and MPEG4 applications, the standard employs convolutional encoding. For data applications, the standard uses turbo encoding. Turbo encoding yields a relatively large encoding gain with a reasonable computational complexity. This encoding scheme is useful for data services that permit longer transmission delays. The data is then interleaved. The coded symbols are written into the interleaver, a memory array, by columns, and read out by rows in a predetermined order. Interleaving is a standard practice to combat signal degradation due to burst errors on the channel [15]. The data is then spread with a user or channel specific Orthogonal Variable Spreading Factor (OVSF) code to produce a data stream at a given chip-rate, where the cross correlation among all users is zero. The spread data stream is scrambled with Gold code so that the multipath signals can be uniquely identified and decoded by the receiver. To transmit a signal within the specified bandwidth, the data bits are shaped by using a pulse shaping filter. Next, the signal passes through carrier modulation and up-conversion to RF.

3.3.2 WCDMA Receiver

Figure 3.6 illustrates the general block diagram for a WCDMA receiver. Each chan-

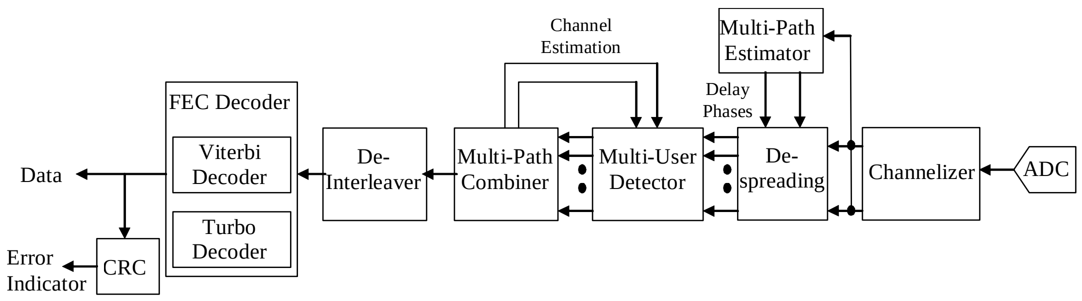


Figure 3.6: Block diagram of a digital WCDMA receiver

nelizer accesses the digital IF and translates a channel to the baseband. The radio environment of a wireless network system is multipath. To be effective, the system requires a despreader that can simultaneously despread the numerous multipaths of a single user, as well as multiple users (for a joint detection). A RAKE receiver, with its multiple fingers to despread the different multipaths, is employed for this function. Then, the block deinterleaver is responsible for performing the reverse action of the interleaver of the transmitter. There are two types of decoders that are used in the receiver. The VD is used to decode signals encoded by the convolutional encoder, and the turbo decoder is used with the turbo encoder. In the following sections, the convolutional encoder and its VD are detailed.

3.4 Convolutional Codes

Convolutional coding can be applied to a continuous input stream to generate a coded output data stream. Convolutional encoders involve the modulo-2 addition of selected taps of a serially time shifted delayed data sequence. The length of the delay is equal to K-1, where K is the constraint length. The exclusive OR gate performs a modulo-2 addition of the inputs. Typically, the encoder is composed of shift registers and a network of modulo-2 addition gates [18], evident in the example in Figure 3.7. Here, the encoder consists of a two-tap binary shift register,

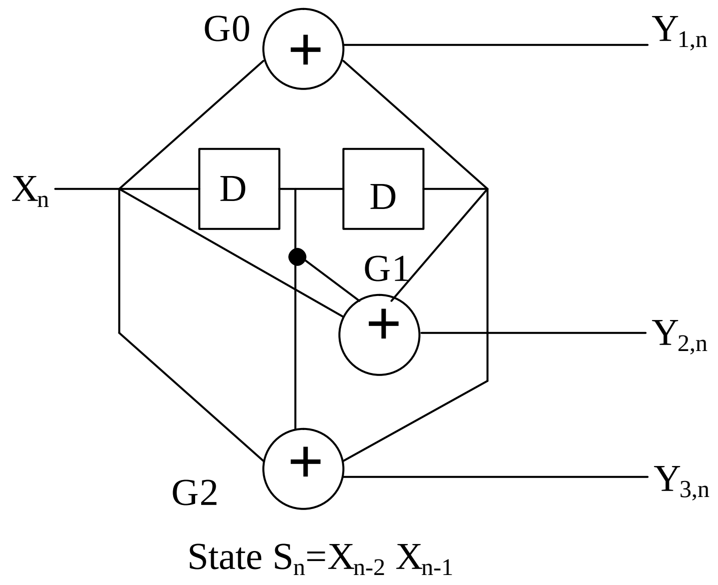


Figure 3.7: 4 state rate 1/3 convolutional encoder

and produces three bits of encoded information for each bit of input information. The encoder is a rate 1/3 convolutional encoder with a constraint length of K=3. Generally, the convolutional encoder contains K-1 shift registers. The output of the convolutional decoder is a function of the current state and the current input;

the generator functions for the encoder in Figure 3.7 are $G0$=101, $G1$=111, and $G2$=111. It is noted that there is no theoretical basis for the optimal location of the shift register stages to be connected to the modulo-2 addition. The operation of the convolutional encoder can be easily represented by the state transition diagram in Figure 3.8. Because of the two memory elements in the encoder, there are four

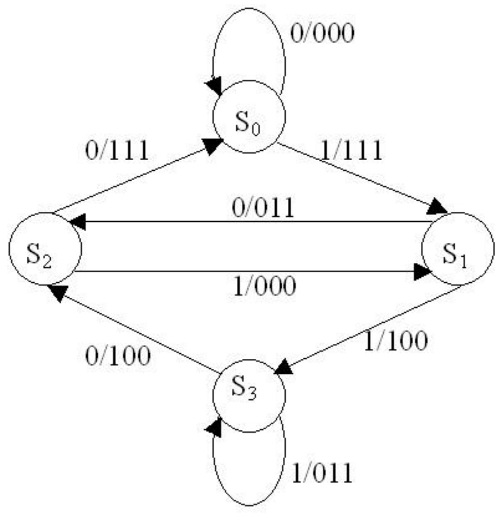


Figure 3.8: State diagram for the encoder shown in Figure 3.7

possible states, (S_0-S_3). Each input to a state generates an encoded output code, and causes a transition. For each state, there are two outgoing transitions; one corresponding to a 0 input bit, and the other corresponding to a 1 input bit. The evolution of the state diagram over time is described by the trellis diagram in Figure 3.9. Consequently, the trellis is a time-indexed version of the state diagram. Each node corresponds to a state at a given time index, and each branch corresponds to a state transition.

3.5 Viterbi Algorithm (VA)

In 1967, Viterbi developed the Viterbi Algorithm (VA) as a method to decode convolutional codes [19]. In order to make the VA a practical decoding technique, certain refinements were made on the basic algorithm [20]. A comprehensive tutorial on the VA is reported in [21]. The VA uses the trellis diagram to decode an input sequence, as demonstrated in Figure 3.10. To demonstrate the functionality of the

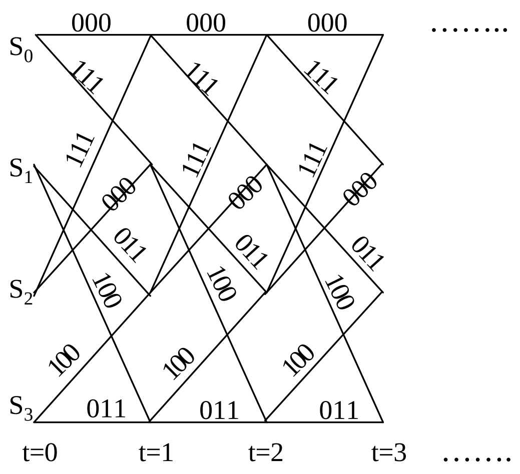


Figure 3.9: Trellis diagram for the encoder shown in Figure 3.7

VA, a sample input to the encoder of Figure 3.7 is tracked until the input is decoded. The encoder has an input sequence, (10110100100), and generates the code stream, (111, 011, 000, 100, 100, 000, 011, 111, 111, 011, 111). This stream is transmitted over a noisy channel, and the code stream, (111, 011, 00$\underline{1}$, 100, 100, 000, 011, 111, 11$\underline{0}$, 011, 111), for example, is received at the decoder. (The underlined bits are incorrect due to the noise encountered during transmission.) The VA, which uses a hard decision format, is exhibited in Figure 3.10. A node is assigned to each state for each time stage. The transition between two states is represented by a branch, which is assigned a weight, referred to as a branch metric (BM). The BM is a measure of the likelihood of the transition, given the noisy observations. The BM, in the simple hard decision example demonstrated in Figure 3.10, is simply the number of bits, in which the received and expected signals differ. The BMs that are accumulated along a path form a path metric (PM). For the two branches entering the same state, the branch with the smaller PM survives, and the other one is discarded. Then two methods can be used to extract the decoded bits: the trace back (TB) or the register exchange (RE). The calculation of the BMs and the PMs is required for both the TB and the RE methods. However, the nature of the decision bits, in addition to the methodology of extracting the decoded output bits, is different for each method.

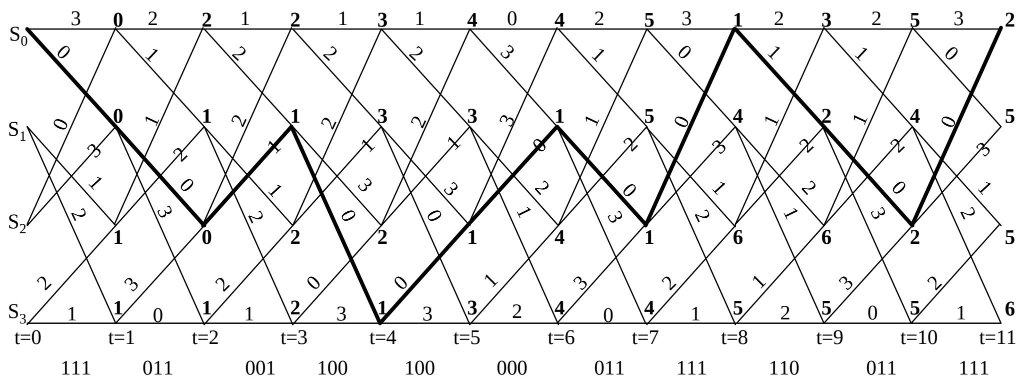


Figure 3.10: Trace Back (TB) Viterbi Decoding

3.5.1 Trace Back (TB) Method

At the last stage of the trellis diagram (see Figure 3.10), the TB method extracts the decoded bits, beginning from the state with the minimum PM, S_0. From this state and tracing backwards in time by following the survivor path, which originally contributed to the current PM, a unique path is identified. This is indicated by the bold line in Figure 3.10. While tracing back through the trellis, the decoded output sequence is generated in the reverse order.

3.5.2 Register Exchange (RE) Method

In the RE approach, a register is assigned to each state. The register records the decoded output sequence along the path from the initial state to the final state. This is depicted in Figure 3.11. At the last stage, the decoded output sequence is the one that is stored in the survivor path register, the register assigned to the state with the minimum PM. Since the RE method does not need tracing back, it is faster. However, the RE method does require the copying of all the registers at each stage. (The computations of the BMs and PMs are not indicated in Figure 3.11 for reasons of clarity.)

Next, the basic blocks of the hardware implementation of the VA, namely the VD are discussed.

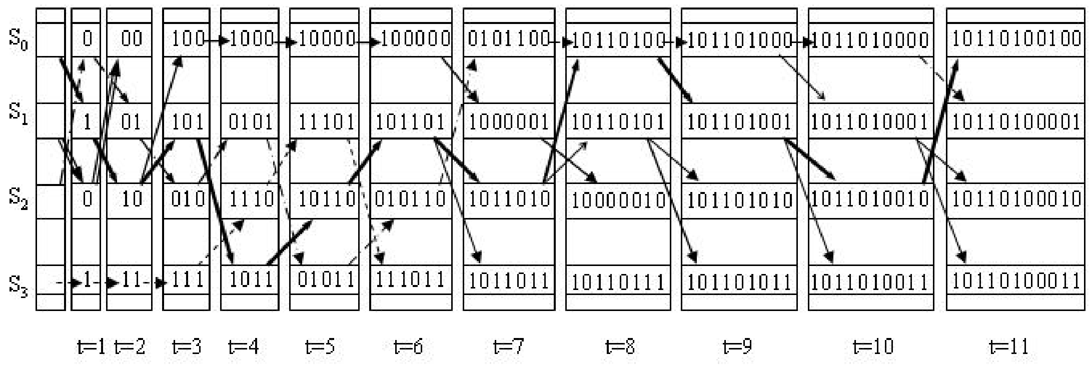


Figure 3.11: Register Exchange (RE) Method

3.6 Viterbi Decoder (VD)

The VD is composed of the three functional units in Figure 3.12:

1. The BM Unit (BMU) which calculates the BMs;
2. The Add Compare Select Unit (ACSU) which adds the BMs to the corresponding PMs, compares the new PMs, and then stores the selected PMs in the Path Metric Memory (PMM); at the same time, the ACSU stores the associated survivor path decisions in the Survivor Memory Unit (SMU);
3. The SMU which stores the survivor path decisions; then the TB mechanism is applied to the SMU.

Regarding the power dissipation of the VD, the SMU is the hottest spot in the VD due to the frequent memory accesses [22]. However, the ACSU is traditionally considered to be a bottleneck for the speed of the VD due to its intensive computations, but recently the memory access speed in the SMU has created even greater limitations. Therefore, this work focuses on modifying the SMU to improve both the speed and power dissipation of the VD. The new modified RE method is detailed in the next chapter.

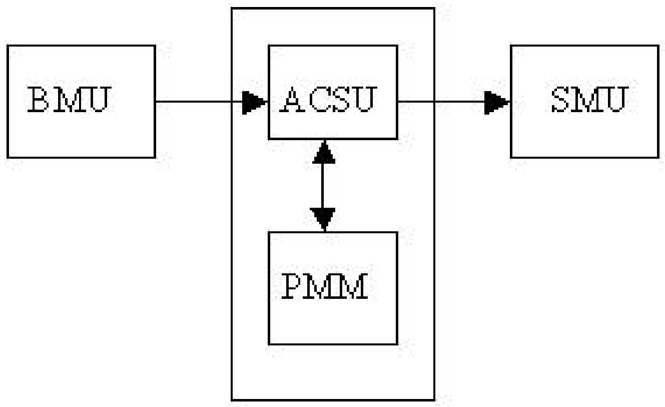


Figure 3.12: Simplified block diagram of the VD

Chapter 4

Modified Register Exchange (RE) Viterbi Decoder (VD)

Section 4.1 of this chapter summarizes the different implementations reported in the literature. Afterwards, the SMU of the modified RE design, proposed in this study, and its associated RTL design are detailed. All of the main concepts that are demonstrated in the figures have a small constraint length (K=3) VD. This simplifies the figures, because it is impossible to demonstrate the concepts by using a 256-state-VD (K=9) in a simple figure.

4.1 Surveying the State of the Art Developments

Usually, with the Viterbi Algorithm (VA), two methods are adopted to extract the decoded bits: the RE and the TB [23]. In the literature, the RE technique is acceptable for trellises with only a small number of states, whereas the TB approach is acceptable for trellises with a large number of states. Therefore, the TB method has been widely investigated and implemented. To attain high performance decoders, several architectures for the TB method have been reported [24][25][26][27][28][29][30][31][32][33][34][35][36]. An M-layered approach that combines the M-stages of the trellis into one stage has been proposed [24], and further developed by using radix-4 architectures [25][26]. Also, the block-based decoding approach [24] has been improved by designing the sliding block architecture [27]. Bit-serial approaches and operation reformulations have also been initiated [28][29]. An attempt to reduce the power consumption of the TB VD by reducing the paths being traced back was the minimum-transition TB scheme [30]. Lin increased the speed of the TB by saving the decisions in a permutation network [31].

Fettweis and Meyr have proposed a semi-ring algebraic solution [32]. All of these architectures utilize the TB method. Only a few attempts to combine the advantages of the TB and RE methods have been reported [33][34]. All of the previous design approaches were developed for low constraint length VDs (K=3 to K=7). The decoders in [35], [37], and [36] were designed for CDMA applications for which the constraint length must be K=9. Although Kang and Willson have introduced a very low-power TB VD [37], its speed is limited due to the use of sequential architectures for the ACS processing.

The decoder that was devised by Chang et al. has an even lower power consumption and achieves speeds in the range of Mb/s [36]. Because the authors' VD, as far as it is known, has the lowest power dissipation in its class, it is chosen as the VLSI benchmark in this paper and is referred to as the TB method.

This work introduces a new implementation which enables the RE method to be used for large constraint length VDs. In addition, the modified RE method is simpler, and has a lower power dissipation than that of the TB method.

A summary of the available implementations for the VD and its different parts in literature is given in Table 4.1, where L is the survivor path length ($5 * K$), D is the decoding depth, f_d is the data frequency, and T_d is the data frequency period. However, the details for commercially manufactured VDs are not published.

Table 4.1: Surveying the state of the art developments

Ref.	Const. ; Rate	Data Rate	Clock Freq	Memory *bits*	L *bits*	D *bits*	Power *mW*	Latency	Tech. μm	V_{DD} *volts*	Trans.
[36]	K=9; r=1/3	2$Mbps$	1-8 f_d	2*48*2^8	48	24	9.8	$(L+D)$	0.5	1.8	
[37]	K=9; r=1/2	14.4kbs	921.6kHz	$(L+D)$*2^8	>45	>1	0.24	$(L+D)$	0.8	1.65	65k
[27]	K=3; r=1/2	140MHz	12MHz		6		24		1.2	1.5	150k
[25]	K=6; r=1/2	140MHz	70MHz	3*32*2^5	32				1.2	5	146k
[38]	K=7; r=1/2		19MHz						2	4.75	50k

The first implementation in Table 4.1, designed for 3G WCDMA, achieved the target frequency of $2Mbps$. Also, the implementation has a minimum power dissipation at $2Mbps$; the second implementation has targeted 2G wireless CDMA terminals. Thus, a state sequential approach is used for implementing the ACSU, which is not feasible for the high frequency requirements of 3G systems. All the other implementations target VDs with low constraint lengths (K =9). This is not adopted by 3G wireless systems. It is obvious why the first implementation [36] will be the guide in the evaluation of the modified RE method, and the benchmark for this research. Although normalizing the power consumption with regards to the technology used is fairly useful in comparisons, normalization is eliminated in our case. The power reduction is realized in this work at algorithm level and hence a fair comparison is accomplished by comparing the number of operations.

In [36] a fully parallel architecture for implementing the ACSU was adopted which can also be applied to the proposed design. For the VD in [36], the TB method is utilized for reconstructing the decoded bits in the SMU, whereas the new design is based on the RE method. The following sections describe the SMU in both designs and provide a comparison of them.

4.2 Modified RE Method

In this research, a modification of the RE method which avoids the power expensive RE between two successive states is proposed. The RE method is based on successive RE operations between two origin states (i,j) and two destination states (p,q), using the butterfly unit as shown in Figure 4.1 [18].

The shifting of one origin state to the left, and appending the decoded bit (the bit that causes the transition) to the Least Significant Bit (LSB) of that origin state results in the associated destination state.

In this investigation, the proposed algorithm uses the pointer concept. Instead of moving the contents of the first register to a second register, the pointer to the first register is altered to point to the second register. In the VD, the pointer to a register is simply the current state of that register. For example, if ($PM_{t-1}^{(i)}$ $+BM_t^{(i,p)}$) is greater than ($PM_{t-1}^{(j)}+$ $BM_t^{(j,p)}$), then the path from j to p is the survivor path for p. The pointer j is shifted to the left, and the bit, which causes the survivor path transition from j to p (in this case 0), is appended to the LSB. Therefore, the pointer which had carried the value j now carries the value p. Then, the decoded bit is appended to the contents of the register whose pointer value is changed from j to p. It is noteworthy that the register has a fixed physical location;

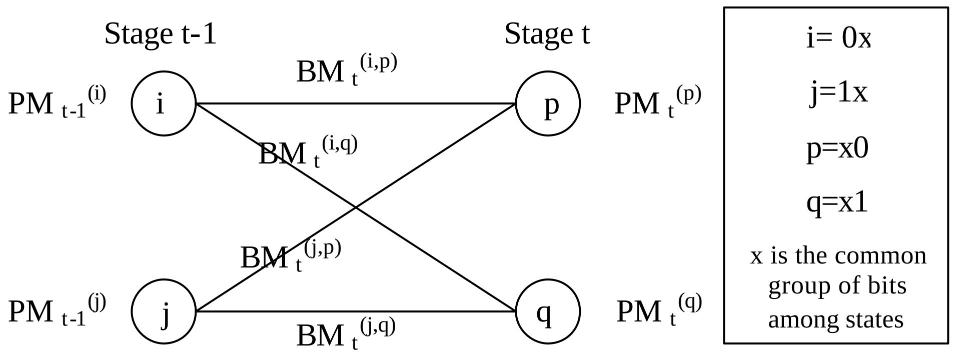


Figure 4.1: Butterfly structure of the ACS

only the value of its pointer changes, and a bit is appended to the corresponding register for each code word that is received. In the new method, the paths are handled from the perspective of the origin states, whereas with the TB method, the paths are handled from the perspective of the destination states. There can be two survivor paths, beginning from the same origin state and entering both destination states, as is the case in Figure 3.10 at time t=3, where the survivor paths for both S_0 and S_1 originate from S_2. This is not a problem for the TB method which monitors the paths from the destination states. But for the modified RE method, the following question arises: which value should the pointer S_2 take, S_0 or S_1?

In other words, which of the two paths originating from S_2 should be the survivor path, and how should the other path terminate? The answer is a new decision bit, called the termination bit, which the ACSU needs to produce for the modified RE architecture. For example, if both paths from state j are considered to be survivor paths for the destination states p and q, the BMs of both paths are compared. Then the pointer, which carries the value j, changes to the destination pointer, whose path has the smaller BM. The pointer of i changes to the other destination pointer, while the path from state i receives a termination high signal. This indicates that the path from state i is terminated, and the decision bits are no longer appended to the path's register. Consequently, only the pointer value of the terminated path will continue to be modified in order to prevent the presence of duplicated pointer values. The issuing of the additional termination bit for each state appears to overload the ACSU conventional operation, but this is not the case. According to the symmetric characteristics of the generator polynomials for the VD used in

CDMA applications ($G0$=557, $G1$=663, $G2$=711, K=9),

$$BM_t^{(i,p)} = BM_t^{(j,q)}, \tag{4.1}$$

$$BM_t^{(j,p)} = BM_t^{(i,q)}, \tag{4.2}$$

and

$$BM_t^{(i,p)} = -BM_t^{(i,q)}, \tag{4.3}$$

$$BM_t^{(j,p)} = -BM_t^{(j,q)}. \tag{4.4}$$

Thus,

$$BM_t^{(i,p)} = -BM_t^{(i,q)}, \tag{4.5}$$

and

$$BM_t^{(j,p)} = -BM_t^{(j,q)}. \tag{4.6}$$

Therefore, a comparison of the BMs, originating from the same origin state, is performed by checking the sign bit of only one BM. Figure 4.2 demonstrates the new RE method with terminating paths. By time t=11, three paths terminate while

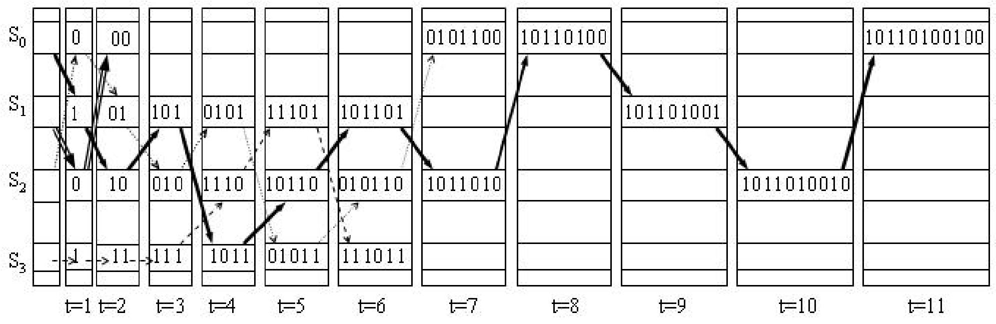


Figure 4.2: Register contents for the modified RE method

only one survives; this one will carry the decoded bits in the last stage. Figure 4.3 indicates the different values for the pointers and registers over time for the new implementation of the RE method. Before receiving the first codeword in a stream of length L codewords, all the PMs and termination flags (one termination flag for each row of memory) are reset. The pseudo code, representing the operations within a butterfly unit for a continuous uncontrolled input codestream, is presented in Appendix A.

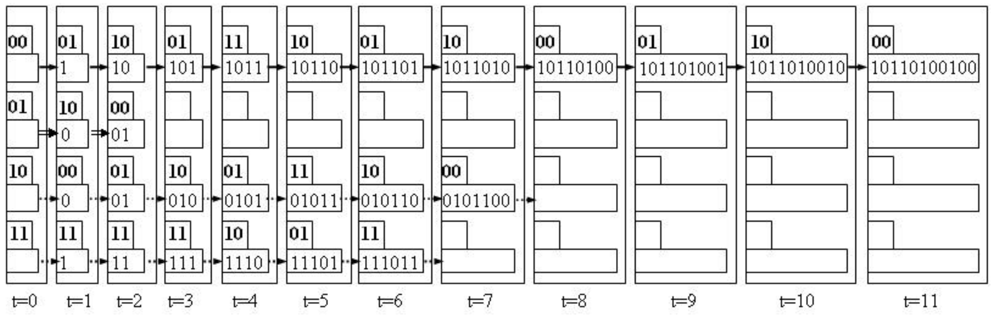


Figure 4.3: New RE approach with pointer implementation (the upper register carries the pointer and the lower register carries the decoded bits)

4.2.1 Performance Simulation Results

To test the performance of the proposed RE method, a JAVA software model is built. The JAVA model has the following inputs: the constraint length K, the rate r, the generator polynomials, the decoding length L, and the SNR E_b/N_0 (in db). Figure 4.4 plots the BER for the new VD in a channel with Additive White Gaussian Noise (AWGN). The simulated Java VD (K=9, r=1/3, $G0$=557, $G1$=663, $G2$=711, and L=48) uses a 3-bit soft decision. The plot in Figure 4.4 illustrates the BER versus the SNR (E_b/N_0) for a continuous uncontrolled input sequence. With an SNR of approximately 4.2 dB, the BER is 10^{-3}; the uncoded signal needs an SNR of 6.8 dB to reach the same BER of 10^{-3} [23]. Thus the Modified RE method has a coding gain of approximately 2.6dB at the BER of 10^{-3}. The equivalent TB VD with L=45, K=9 and r=1/2 has a coding gain of approximately 3.3dB [37]. Since the rate change from r=1/2 to r=1/3 causes a coding gain of approximately 0.4dB, the loss in coding gain for the modified RE method is around 1 dB for K=9, r=1/3 and L=48 if compared to the traditional TB method with the same parameters at a BER of 10^{-3}.

4.2.2 Other Suboptimal Decoders

In the literature, several alternative sequential and stack based techniques have been proposed to decode convolutional codes, but none of them are as computationally efficient as the VA. To reduce the computational complexity, several suboptimal variations of stack-based algorithms have been investigated, but they sacrifice performance, and require much more memory [39]. However, reference [40] states that technological change has rendered the traditional computation/memory

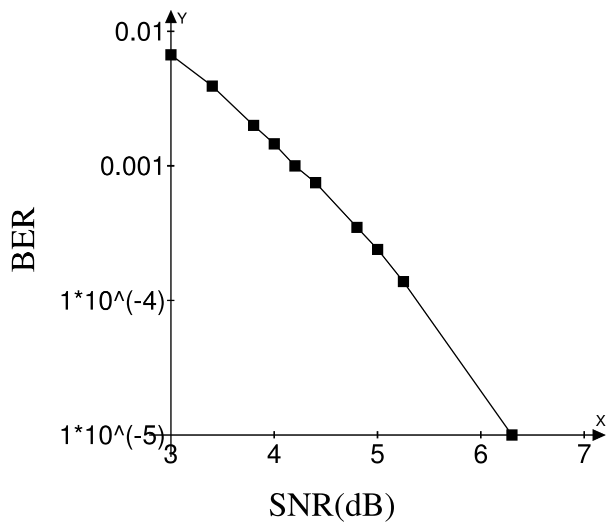


Figure 4.4: BER vs. SNR for the proposed modified RE method

trade highly suspect .

Adaptive Viterbi Algorithm (VA)

Some adaptive reduced-complexity VAs have been developed. The role of these algorithms is to adjust the number of preserved states, depending on the variation of channel noises, so that the loss of the correct path is prevented. The algorithm that was proposed in [41] collects several path metrics, and maps them into a path segment metric. Another adaptive VA has been developed by Chan and Haccoun [42]. The algorithm keeps only a certain amount of most-likely states, whose respective path metrics are below a given threshold. Henning and Chakrabarti have devised an adaptive T-algorithm, a variation of the VD that applies a threshold to the accumulated path metrics the VD uses to select only one path per state of a trellis stage for storage [43]. Adaptive methods can perform closely to the conventional VA with a smaller computational complexity, but none of these algorithms ensure the amount of reduction in the computational effort. In other words, the reduction in computational complexity is statistically dependent on the channel noise levels. These algorithms are also sensitive to the parameters of the model. They perform as well as conventional VDs with a significant reduction in the computational complexity only if the proper system parameters have been selected. In

[44], the VD was an extension of the work in [42] and it reduced the power dissipation of an equivalent systolic array VD, not a conventionally implemented one. The maximum weight basis decoding, suggested in [39], requires a low number of operations, but its power consumption has not been tested yet.

The design that is proposed in this work reduces the power dissipation by reducing the memory requirements and the memory activities. In addition, the new design allows higher frequencies for the SMU, because the read and write operations are performed at the data rate frequency. For a fair power reduction estimation, the new design is compared to a conventional VD decoder implementation, which has been considered to have the least power consumption in its class.

VHDL models are constructed for the SMUs of both the TB method and the modified RE method, and are briefly discussed in the next section.

4.3 VHDL Models for Comparison

To quantify the power dissipation reduction, a functioning RTL VHDL model is designed for the SMU of the new RE method, and for the SMU of the TB method.

4.3.1 TB Survivor Memory Unit (SMU) VHDL Model

Since a survivor path length of L=48 is used, the minimum survivor path memory size is 256 x 48 bits. The memory access ratio of the write and read operation is 1:3, instead of 1:48 [36]. The small ratio leads to a memory size of 256 x 96 bits, but achieves a much faster trace back by using a lower frequency for reading. Figure 4.5 illustrates the timing chart of the major operations in the TB VD. The clocks, CLK_s, CLK_w, and CLK_r, correspond to the operations of the input decision sampling, TB writing, and TB reading, respectively. The decode path length that is adopted is equal to 24. Consequently, after 48 TB read operations, 24 decoded bits are generated, consecutively, by the TB decode operations. Thus, a 24-stage LIFO buffer needs to be placed at the output to correct the output order.

The VHDL model for the TB SMU is composed of three main blocks which are portrayed in Figure 4.6. The TB_MEM block consists of 256 rows of memory, and each row is composed of 96 bits. The ROW_REG block is an 8-bit address decoder which contains the current row address being read during the TB operation, whether it is a TB read cycle or a TB decode cycle. For each read signal, the ROW_REG register is shifted once to the right, and the content of the last memory cell read is

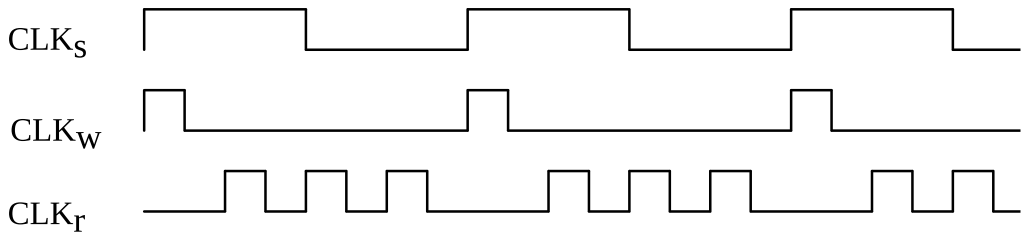


Figure 4.5: Timing chart of the main clocks for the TB VD

appended to the MSB of the register. The LIFO block is a 24-bit LIFO register, in which the decoded bits are stored serially, and read out in the reversed order.

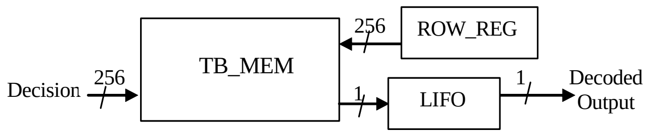


Figure 4.6: VHDL block diagram of the TB SMU architecture

4.3.2 Modified RE VHDL Model

To obtain a reasonable comparison between the TB and the RE implementations, the same survivor path length of L=48 is chosen. The VHDL model for the new RE SMU is also composed of three main blocks as signified in Figure 4.7. The ACS_to_SM block contains 256 pointers, each pointing to one row of memory. The block is responsible for routing the decision and termination bits to the appropriate row in the RE_MEM, and for changing the associated pointer to the new value. Instead of shifting 256 (8-bits) pointers for each incoming codeword, a circular pointer is used; that is, the MSB block, which is denoted in Figure 4.7. It is a circular counter which points to the current MSB of the pointers. Thus, the MSB block acts like an indirect pointer, a pointer to the pointer.

The MSB block has eight outputs that point to the eight bits of the pointers in parallel. The reset signal causes the eighth bit of the MSB block to be high which, in turn, causes the eighth bit of any pointer to be the MSB of that pointer.

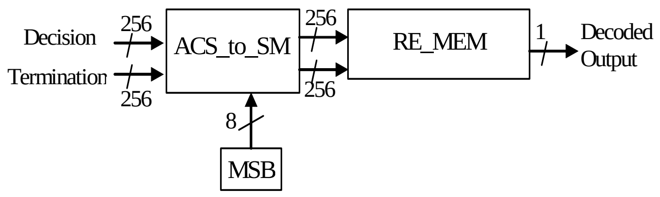


Figure 4.7: VHDL block diagram for the new RE SMU architecture

After accessing an incoming code word in the ACSU, a decision bit is written into the MSB of each pointer, and the output of the MSB block is shifted to the right such that only the seventh bit is high. For each pointer, this indicates that the seventh bit is now considered to be the MSB. In this case, pointer P should be read in the following order: $P_6P_5P_4P_3P_2P_1P_0P_7$, instead of the usual order: $P_7P_6P_5P_4P_3P_2P_1P_0$. This process continues with each incoming code word. The ACS_to_SM block design avoids the power dissipation that is associated with the register shifting of the pointers. The RE_MEM block in Figure 4.7 consists of 256 rows of memory, and each row is composed of 48 bits. Only one write_enable signal is required for each column, because the ACSU writes the decision bits into all the elements, one column at a time.

4.3.3 SMU Comparison

The RTL simulation-based power analysis, provided by Synopsys and based on a 0.35μm CMOS technology, is applied to both designs. The analysis shows a power reduction of 45 percent in the SMU of the modified RE method, and a power reduction of 23 percent for the entire VD. (The SMU consumes more than half of the power of the VD [21].) A simple power reduction estimation is presented in the next section.

The new RE implementation that is presented in this paper reduces the memory requirements of the SMU by 50 percent. Table 4.2 provides a comparison of the register and memory access operations that are completed by each method: the TB and the modified RE in the SMU. The operations in Table 4.2 are those that are required to decode 48 codewords.

W, r, and rg represent the power dissipation cost function for writing one bit into the memory block, reading one bit from the memory block, and writing one bit into a register, respectively. The reduction in the number of decision bits that are written into the memory for the new RE method is due to the termination concept. The power that is consumed by the memory cells during the precharge is a high percentage of a cell's power dissipation. Also, the power consumption of one memory cell is proportional to the number of memory cells in the memory block [45]. The memory block used in the designs is 256 x 48 bits (256 rows and 48 columns), and the size of one register is, approximately six times the size of one memory cell. Therefore, the following is assumed: $w{:}rg \sim$ 256 x 48:6 [46]. Since all the contents of one column of the memory block are written one at a time, the assumption is altered to the following: $w{:}rg \sim 48{:}6 = 8{:}1$. In addition, a memory cell uses less power for reading than for writing, because the reading can be performed by using a reduced swing on the bit lines. Consequently, it is assumed that $w{:}r \sim 2{:}1$ [46].

If the equivalent estimated value of w is substituted for r and rg in Table 4.2, the results in Table 4.3 are produced. According to Table 4.3, the estimated power reduction is 39 percent in the SMU (20 percent for the whole VD). This estimation yields acceptable results, if it is compared to the VHDL results.

The SMU is the hottest spot in the VD due to the frequent memory accesses [22]; therefore, it is modified to reduce the power dissipation. Regarding the speed of the VD, the ACSU is traditionally considered to be a bottleneck, because of the ACSU's intensive computations. Recently, the memory access speed in the SMU has created even greater limitations. The ACSU is composed of 128 ACS butterfly modules. One ACS butterfly module is designed at the circuit/gate level and fabricated. The ACS module inlcudes a new shifter and a low power 16-transistor full adder. The module is detailed in the next chapter.

Table 4.2: Comparison of the memory/register operations to decode 48 codewords

Operation	TB	New RE
Writing decision bits into the memory	256 x 48 12288w	(256 x 48)/2 6144w
Reading from the memory	48 x 3 144r	48 48r
Writing into the MSB of the pointers	–	256 x 48 12288rg
Writing termination bits into the termination registers	–	256 256rg
8-bit register shifting (shift right and append a bit to the LSB)	48 x 3 x 8 1152rg	–

Table 4.3: Estimated cost function for the memory/register operations to decode 48 codewords

Operation	TB	Modified RE
Writing decision bits into the memory	256 x 48 $12288w$	(256 x 48)/2 $6144w$
Reading from the memory	48 x $3r$ $72w$	$48r$ $24w$
Writing into the MSB of the pointers	–	256 x $48rg$ $1536w$
Writing the termination bits into the termination registers	–	$256rg$ $32w$
8-bit register shifting (shift to the right and append a bit to the LSB)	48 x 3 x $8rg$ $144w$	–
Total	$12504w$	$7736w$

Chapter 5

Add Compare Select Unit (ACSU) Chip

For WCDMA systems with K=9, there are 256 states (2^8=256) in the trellis. Therefore 512 Add and 256 Compare and Select operations need to be conducted for each decoded bit. A total number of 256/2= 128 ACS butterfly modules are needed.

5.1 Previous Designs for the ACSU

For a large constraint length VD, which requires a low decoding rate, a state sequential approach is normally used, where the same ASCU unit is serially used by a number of states [37]. To achieve higher speeds, state parallel approaches have been adopted. Earlier attempts to increase the speed of the ACS suggest a semi ring algebraic solution. It replaces the recursive ACS operations by an (n x n) matrix multiplication, where n is the number of states [32]. This method is not feasible for the large constraint length VD (K=9) that is employed in CDMA systems, because it requires a 256 x 256 matrix multiplication.

Because the absolute values of the path metrics are not relevant (only their relative values are), a Carry Save (CS) addition has been suggested [47]. In the CS addition, instead of loading each carry bit to the next full adder, the carry bit is combined with the sum bit for a new digit. This leads to a redundant representation with a larger number of bits, but has the advantage of eliminating the ripple in both the addition and minimum selections. Due to the increased complexity and increased number of bits, this technique is not adopted in the proposed ACSU design.

Lou has proven that five bits and eight bits are sufficient to store the branch metrics and path metrics [48]. To compare two path metrics, the computation of the MSB of the result for a straight forward two's complement subtraction of the two 8-bit path metrics is necessary. Thus, for a butterfly organization, the required operations are 1,024 8-bit additions, and 512 8-bit subtractions and selections. To reduce the power consumption of the ACSU, a reformulation of the ACS operations has been proposed in [29]. Another approach to reduce power consumption has been developed in [36] by applying a bit serial approach instead of a bit parallel approach, thus reducing the interconnect network between the ACSU and PMM. To improve the throughput of the ACSU, the M-layered approach for the ACSU indicates that M-stages of the trellis should be combined into one stage by dividing the received codeword stream into short blocks of codewords with a length of M [24]. Thus, instead of entering only one codeword to one stage of the ACS, a group of M codewords are entered to one M-layered ACSU. The memory and computational requirements increase drastically with the increased value of M. A special case for the M-layered method is the radix-4 (M=2) method in [25] which improves the throughput by a factor of 1.7, compared to the throughput of the conventional radix-2 VDs. The radix-4 implementation has been further improved in [27][26], but the relative power consumption of this technique has never been investigated. In [28], a technique, called the Decoupled radix-4, has combined the three techniques: the ACS operations reformulation, the Radix-4 method, and the bit serial approach.

Many researchers have explored the problem of interconnecting the various ACS modules [38] [49]. The interconnection problem results from the feedback that is inherent in the ACSU. The PMs that are calculated at one time stage are used in the next time stage to calculate the new PMs. Therefore, there are 256 connections among the 128 ACS modules in the parallel approach. If each connection is, in turn, composed of eight bits, the interconnection load is fairly large. As as a result, the bit serial approach significantly reduces the interconnection load. Based on the in-place updating of the path metrics [50], Chang has presented an efficient architecture for the VA which reduces the interconnection complexity of the ACSU, in addition to reducing the memory size and operations [51].

The ACSU architecture that is adopted in this research is presented in the next section.

5.2 Bit Serial ACS Architecture

For the modified RE method, the bit serial approach is used. The block diagram for one ACSU is displayed in Figure 5.1 The bit serial approach significantly reduces the

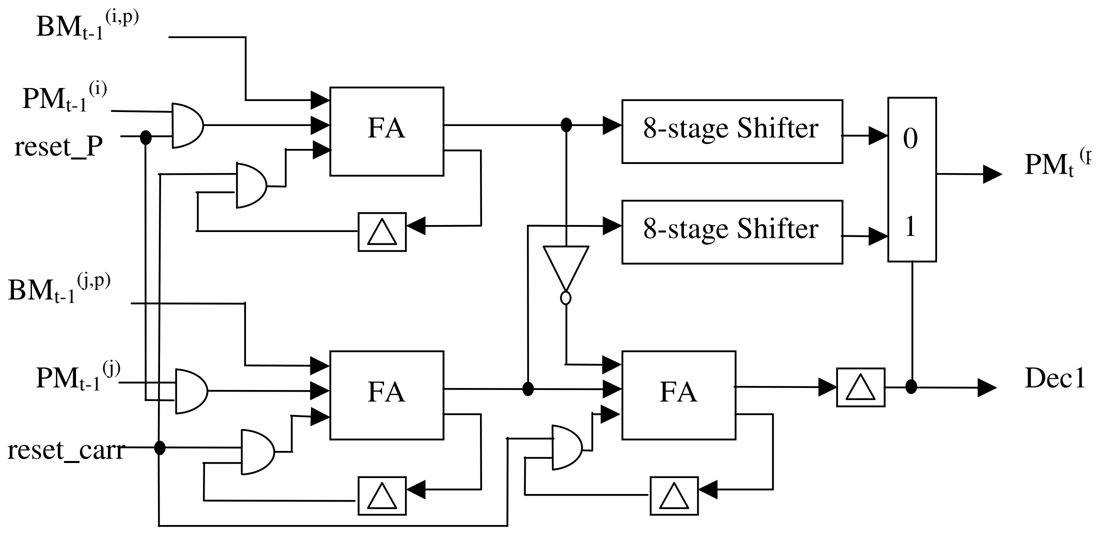


Figure 5.1: Bit serial ACS architecture

interconnection load, but does not allow for the reformulation of the ACS functions. Thus, the traditional addition and subtraction operations are performed in the bit serial ACSU. However, the bit serial approach necessitates two 8-bit registers for each PM to temporarily store the two calculated PM values which are compared in the ACSU so that the smaller one can be chosen as the new PM. In addition, extra registers are required to store the carry bits that are generated during the addition and subtraction operations in the ACSU. The registers' overload is reduced by using a ring-type FIFO register [36] [52]. According to the power estimates, a shift register based circuit consumes 35 percent more power than the ring-type FIFO [36]. But an 8-bit ring-type FIFO requires two 8-bit address pointers to control the FIFO functionality. In this work, a very simple shifter is introduced which requires only one control signal and is composed of eight master latches with a new design.

5.3 Simple Shift Register

Figure 5.2 is a diagram of the 8-bit shifter that is used in the ACSU. The transistor design of each master latch is represented in Figure 5.3.

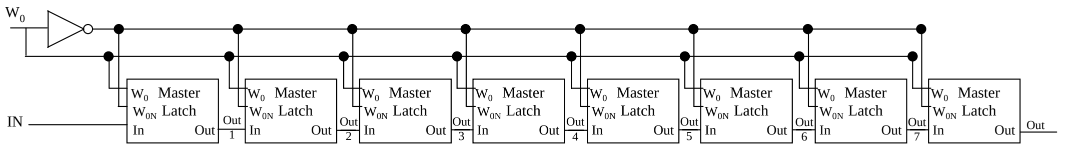


Figure 5.2: 8-bit shifter

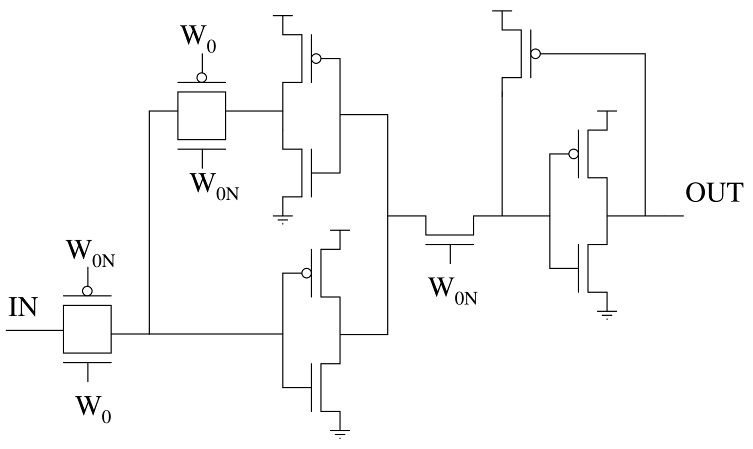


Figure 5.3: New master latch

The master latch is a slight modification of a regular transmission gate latch. The output stage is changed to ensure a proper shift operation with only one clock signal. The output is enabled by an NMOS pass transistor and buffered with an inverter. Since the output of the NMOS pass transistor has a high level of $(V_{DD} - V_T)$, the output buffer adopts a feedback PMOS device to restore the high level to the VDD [2]. For simulation purposes, the input and output signals are generated from buffers in order to emulate real conditions. The two control signals for controlling the shift operation are the inverse of each other. Since W_{ON} is generated by inverting the clock W_O, this simple shifter requires only one clock signal. The timing simulation for the 8-bit-shifter is shown in Figure 5.4. If minimum sized transistors with a width of 0.5μm (0.18 μm technology) are used, a maximum frequency of $125 MHz$ is achieved. The delay at this frequency is $0.6ns$.

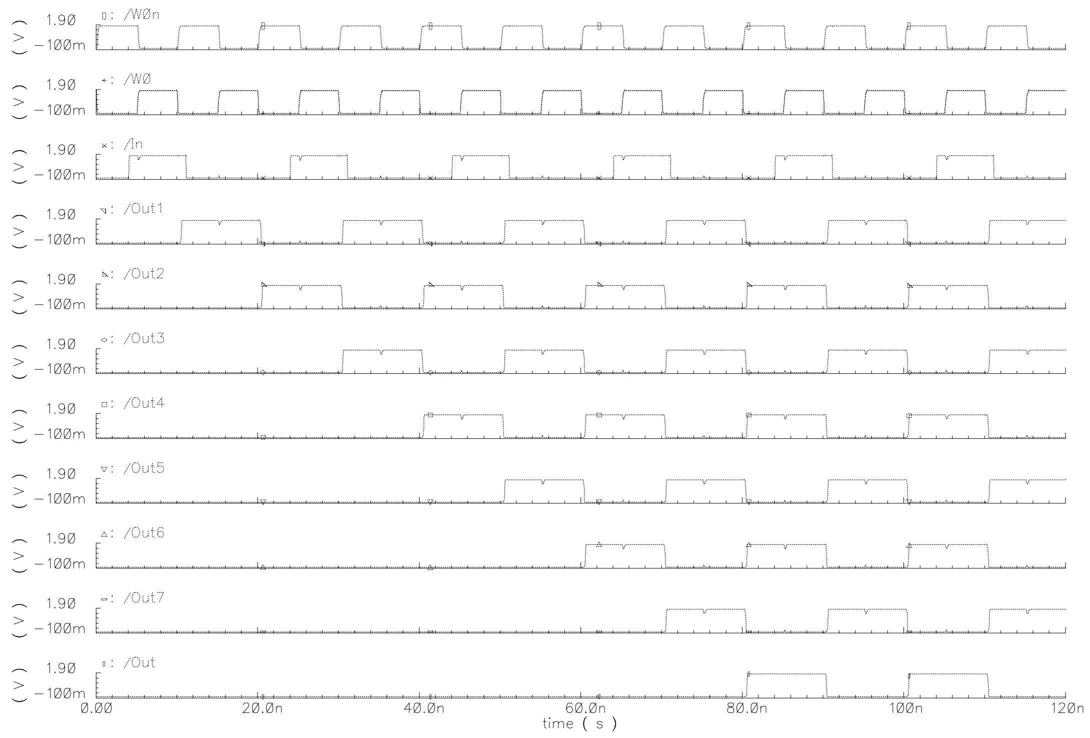


Figure 5.4: Timing simulations for the 8-bit shifter

5.3.1 Power Consumption

The shifter, in the ACSU design consumes more power than the ring-type FIFO. However, the simple shifter requires only one clock signal, whereas the ring-type FIFO requires eight control signals. These signals are anti phase and are routed to all 128 ACSU pairs. Since the routing presents a high hardware overhead, the simple shifter is used.

The ACSU is responsible for adding the BM to the corresponding PMs, and then comparing them so that the smaller PM can be chosen. Because the bit serial approach is adopted, extra registers are needed to save the carry bits.

5.4 Carry Propagation

The output of the carry's register, needs to be reset at the beginning of each cycle of the ACS operation. Only the output, not the content of the carry's register, must be reset. Resetting the content causes logical errors, because the FA itself is not synchronized. Consequently, correct functionality is maintained by providing the inputs at the appropriate time. A resetter, composed of only one transistor as shown in Figure 5.5, is used.

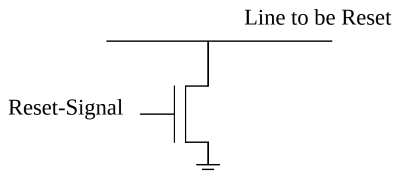


Figure 5.5: Simple resetter

The low power FA used in the ACSU is discussed in the next section.

5.5 Low Power Adder

The bit serial approach requires three 1-bit FAs for each branch of the butterfly ACSU. Thus, the VD needs (256*3=)768 1-bit FAs. Therefore, it is essential to use a low power FA design. Shams and Bayoumi have devised a novel high performance CMOS one-bit FA cell [53] [54], which outperforms other standard implementations. Another implementation, presented in [55], apparently has a lower power consumption, but due to the lack of detailed transistor sizing information and questionable loading effects, this cell is not used in the ACSU. The 16-transistor FA cell is presented in Figure 5.6.

5.6 Chip Specifications

One ACS butterfly module is fully designed and implemented on a chip. The chips fall into two general categories: core limited and pad limited. A core limited chip is one in which the size of the chip is dependent on the amount of logic it contains. The perimeter of the chip is more than sufficient to support the I/O, clock, power, and

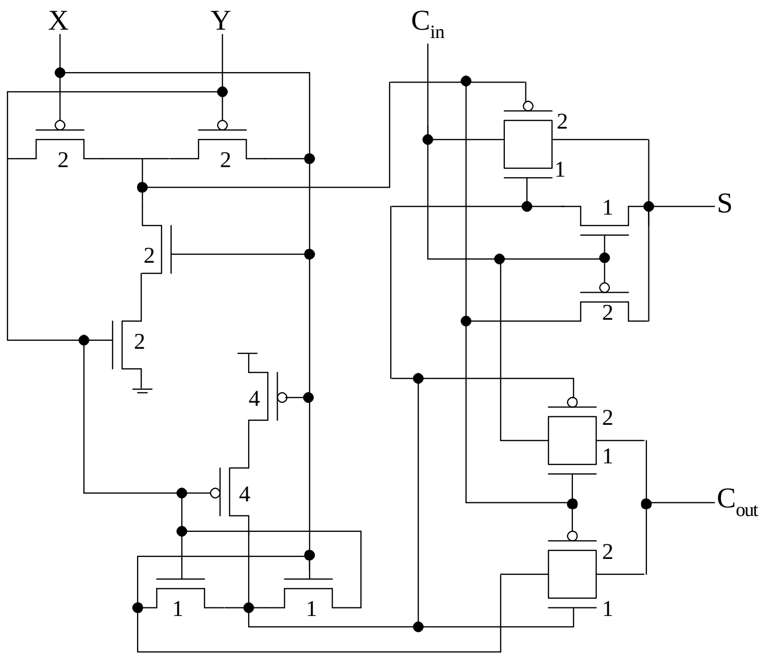


Figure 5.6: 16-transistor low power FA

ground bonding pads surrounding the chip. A pad limited chip's size is dictated by the bonding pads on the chip's perimeter. The pads are as close as possible which is consistent with the chip's design rules. Also, there can be a wasted, open area within the chip's core.

The fabricated ACS chip is pad limited and has a size of $1756\mu m$ x $1489\mu m$. The fabrication is run through Canadian Microelectronics Corporation (CMC) by using $0.18\mu m$ technology. Figure 5.7 shows the layout of the design, including the pads.

The chip operates at a 3.3v ring voltage and a 1.8v core voltage. A snapshot of the timing simulation for the designed chip with pads attached is depicted in Figure 5.8. The input signals are listed on the right side of the figure and the output signals are listed on the left side of the figure. There are two clock signals ($W0$, $W0n$), four data signals (BM_{ip}, BM_{jp}, PM_i, PM_j, and msb_BM_{jp}), and four control signals (*reset_comp*, *reset_PM*, *comp_enable*, and *comp_enable_n*). The function of the control signals is detailed in Table 5.1.

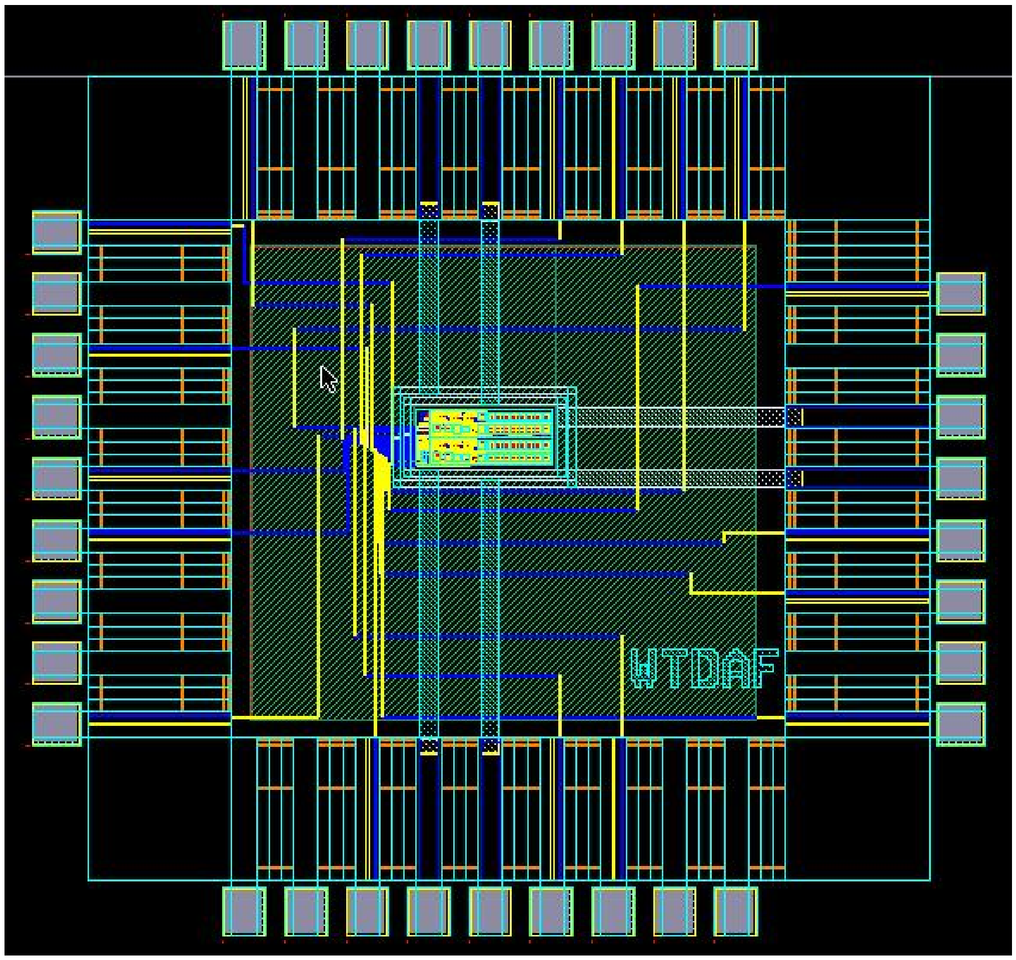


Figure 5.7: Layout of the fabricated chip

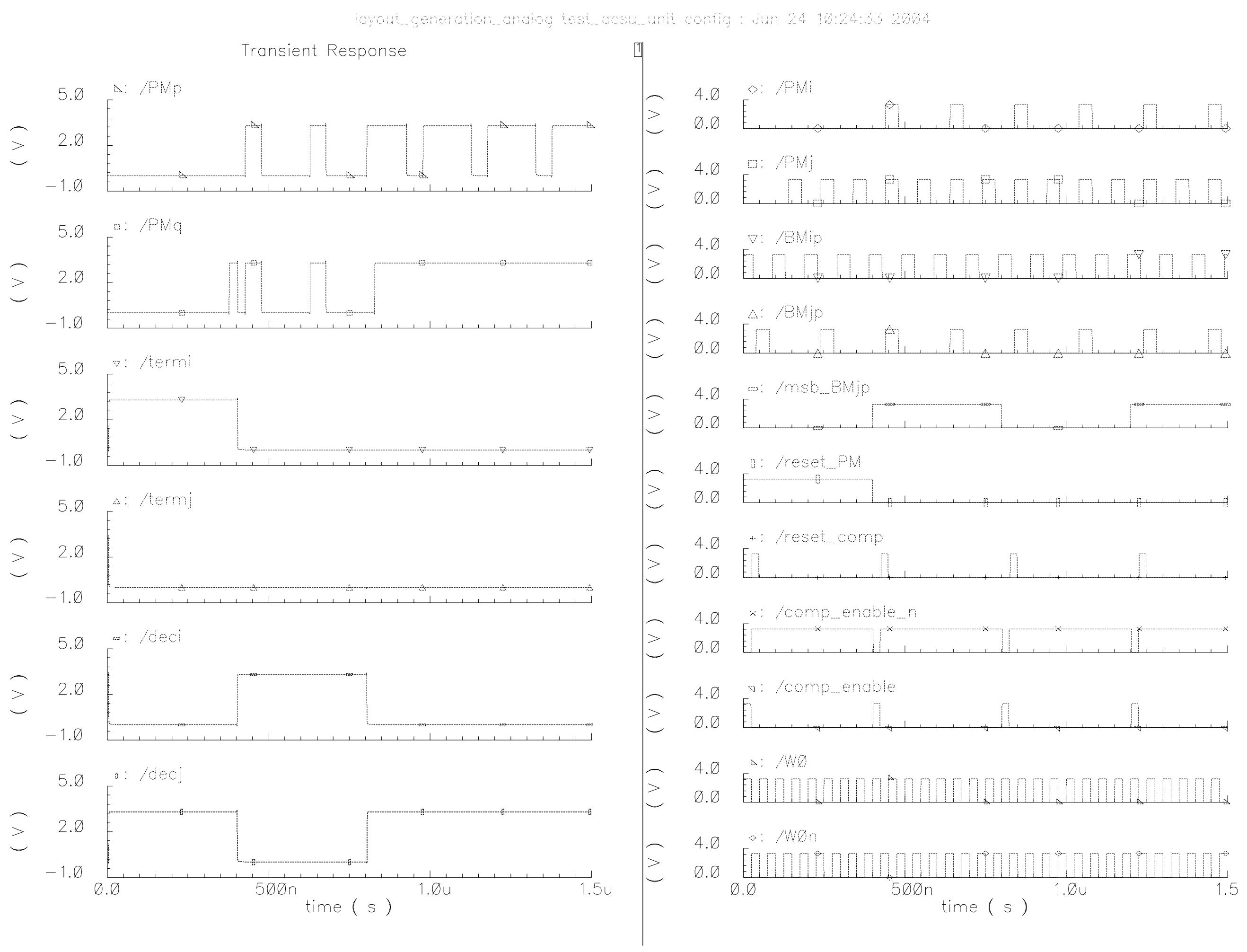


Figure 5.8: Timing simulation of the layout

Table 5.1: Function of the control signals

Control Signal	**Function**
reset_PM	to reset the PMs for the first cycle of the ACS module
reset_comp	to reset the carry signals at the proper time
comp_enable *comp_enable_n*	to control the latch that latches the subtraction's MSB which decides the smaller PM

Two packages are chosen for fabrication: the 40DIP and 44CQFP. The 40 pin DIP performs well at up to about 50 or 60 MHz, but the parasitic losses distort the signals if the chip operates at frequencies higher than 60 MHz. Therefore, 40DIP chips are only used for the basic operational checks. The 44CQFP package can operate at 160 MHz easily, and therefore is the best choice for the at-speed testing.

5.6.1 Chip Testing

The testing of the 40DIP chip was run on the IMS Tester at CMC. I sent the test vectors and they set up the testing equipment and the test settings and ran several tests. We shared the application online and I ran the final test. The snapshot of the final test is shown in Figure 5.9. All outputs were generated correctly at frequencies up to 100 MHz except one output.

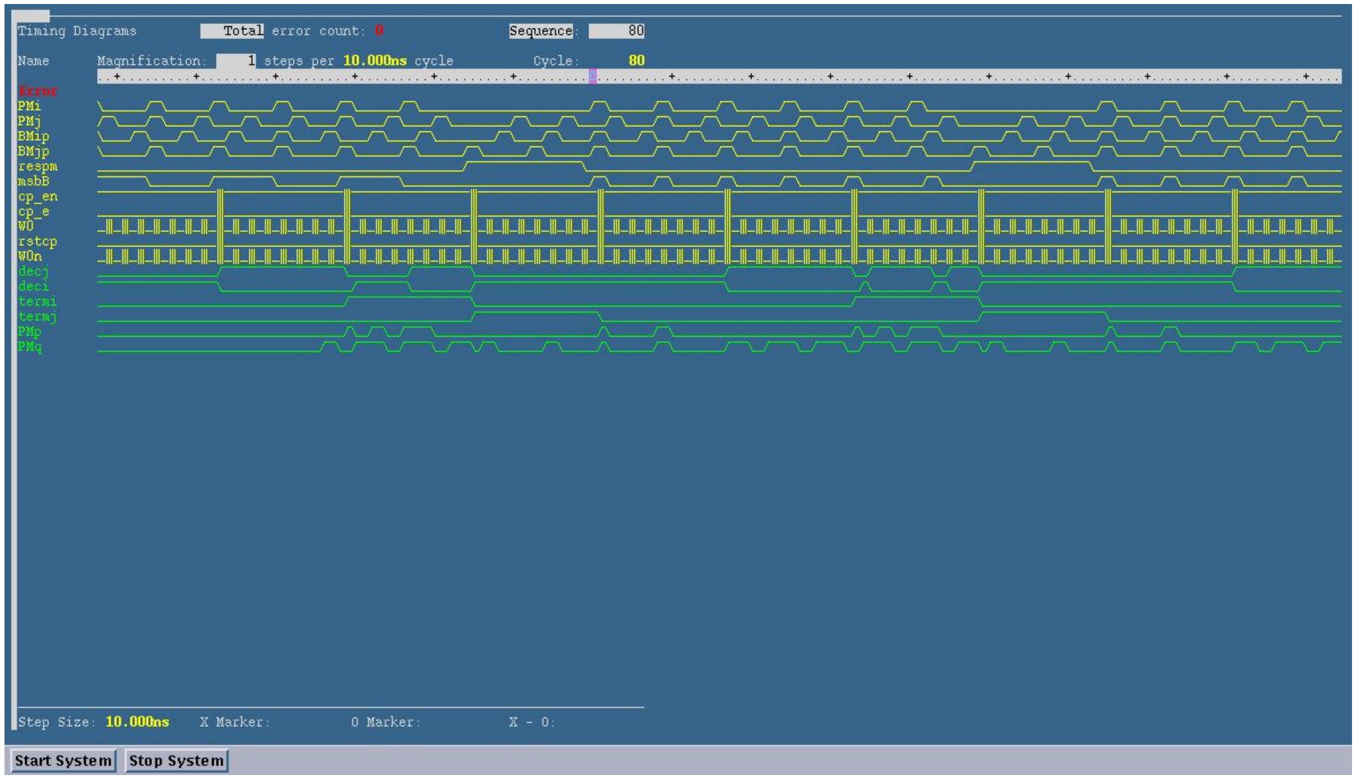

```
Timing Diagrams          Total error count: 0          Sequence: 80
Name    Magnification: 1 steps per 10.000ns cycle      Cycle: 80
Step Size: 10.000ns   X Marker:   0 Marker:   X - 0:
```

Figure 5.9: Testing simulations of the chip on the IMS shifter

The fabricated chip represents only one ACS butterfly module for the modified RE VD, and the VD is fully implemented on an FPGA after the VD's power consumption is further reduced. This is detailed in Chapter 6.

Chapter 6

Memoryless Viterbi Decoder (MLVD)

The modified RE VD uses a memory of 256 x 48. This is the minimum memory a traditional implementation of the VD requires to decode a convolutional code with $K = 9$ and $r = 1/3$. The modified RE architecture is further enhanced to reduce the required memory to zero. This is detailed in this chapter.

6.1 Architecture of the MLVD

The diagram of the modified RE approach, which was detailed in Chapter 4, is presented in Figure 6.1. This figure displays the successive values for the pointers and rows of memory over time. A closer look reveals that each row of memory is used to trace the decoded bits, if an initial state is assumed. The first row of memory decodes the data, if an initial state, S_0, is assumed. The last row records the decoded data, if an initial state, S_{255}, is assumed, and so on. At the end of the decoding process, the row which has the lowest PM is chosen to be the decoder output. If the initial state is known, are all these rows of memory necessary? Absolutely not. For example, if the initial state is zero, then only the first row of memory is needed. In other words, the storage of the decoded bits is necessary in order to choose only one row of memory at the end to represent the actual decoded bits. If the required row of memory is predetermined, then there is no need for the storage of the other rows.

Furthermore, there is no need for the storage of the row that is assigned to the predetermined initial state, because the RE approach generates the decoded bits in the correct order. The decoded bits are produced, and then read out from

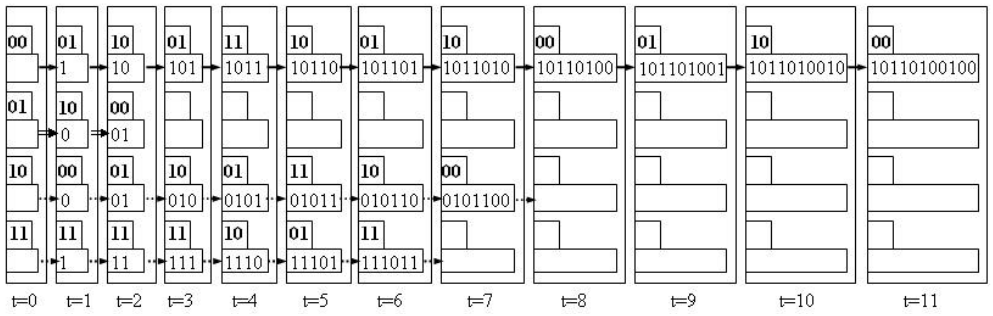


Figure 6.1: New RE approach with pointer implementation (the upper register carries the pointer and the lower register carries the decoded bits)

the decoder. Thus, a memory free Viterbi decoder can be implemented by solely resetting the encoder contents for each L bits that are encoded. There is no need to interrupt the data sequence nor to transmit a long sequence of zero data bits. The encoded data is continuous, but the contents of the encoder bits (eight bits for $K = 9$ convolutional encoder) are reset to zero for each L bits transmitted. The only overhead for such an implementation in a communication system is to synchronize between the transmitter and the receiver. The new VD implementation is called, the MemoryLess Viterbi Decoder (MLVD). Since the MLVD needs to track only one row, the MLVD requires only one pointer to track the current position of the decoder in the trellis in Figure 6.2.

The MLVD is designed in VHDL for the WCDMA applications. As mentioned in Chapter 3, the specifications for the VD for the reverse link (mobile to base) of WCDMA applications are listed in Table 6.1.

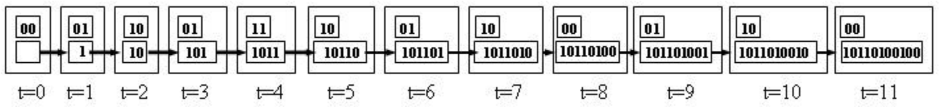


Figure 6.2: MLVD approach with pointer implementation (the upper box carries the pointer and the lower box carries the decoded bits)

The MLVD is an extra low power design for a VD with the only restriction of resetting the encoder register at each L of the encoded data bits and providing the necessary synchronization. The block diagram of the MLVD, designed in VHDL,

Table 6.1: VD specifications

Constraint Length	$K = 9$
Coding Rate	r=1/3
Generator Polynomials	$G0$= 557, $G1$=663 $G2$=711
Decision Level	3-bit Soft Decision
Path Metric	8-Modulo Arithmetic
Target Speed	$2Mbps$

is shown in Figure 6.3.

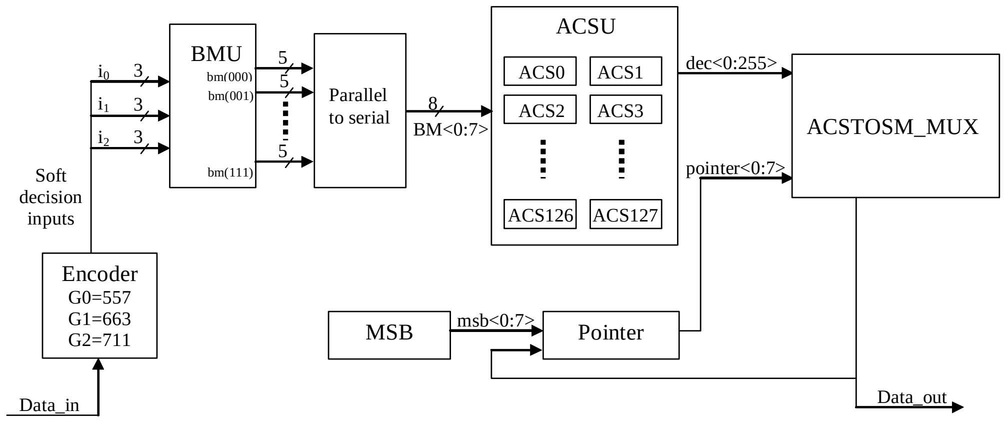


Figure 6.3: MLVD block diagram

In order to have a built-in self-test design, a Linear Feedback Shift Register (LFSR) and a comparator are added. The LFSR produces the random input for the encoder, whereas the comparator compares the delayed version of the LFSR with the output of the MLVD. An output signal, status, indicates the correct functionality of the design. The VHDL code for the MLVD is listed in Appendix A. The following is a discussion of the different parts of the MLVD design and their functionality.

6.1.1 Convolutional Encoder

The convolutional encoder that is implemented is that of the reverse link for WCDMA applications. It is a K=9 and r=1/3 convolutional encoder. Its input is generated by a LFSR which generates a random sequence of ones and zeros. Figure 6.4 shows the block diagram of the convolutional encoder. Three outputs are generated for each bit encoded. To implement a soft-decision VD, the output of the encoder is translated from (0,1) to (101, 011). 101 is the two's complement representation of the decimal number -3, and 011 is the representation of the decimal number 3. The output of the encoder is fed directly (without noise) into the first block of the VD, the BMU.

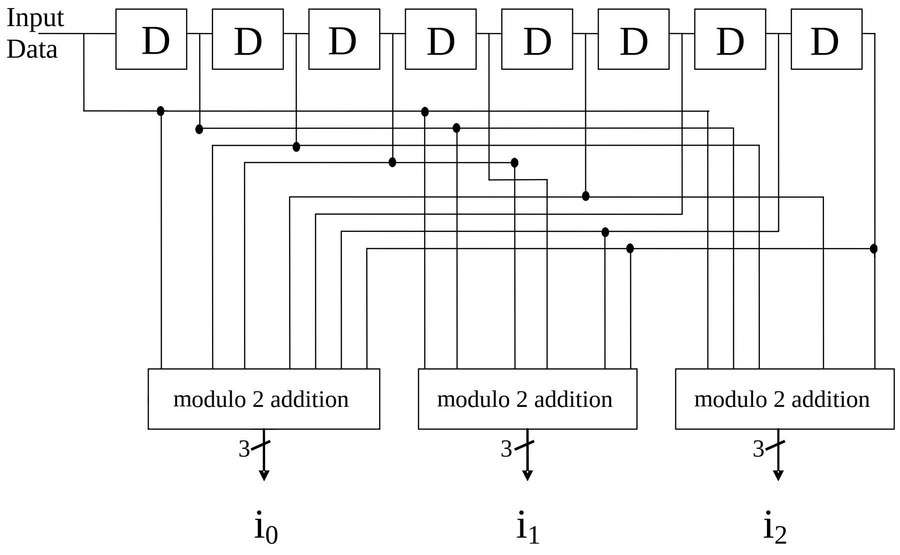


Figure 6.4: Convolutional encoder: $G0$=557, $G1$=663, and $G2$=711

6.1.2 Branch Metric Unit (BMU)

In the VA for decoding convolutional codes, the squared Euclidean distance is the optimum branch metric for decoding sequences that are transmitted in an AWGN channel [21]. Multiplication operations or look up tables are required for the VA to compute the squared distances to obtain the branch metrics. However, for binary convolutional codes, it is proven that linear distances (Hamming distances) can be used as the optimum branch metrics. This is true for convolutional codes, using anti-podal signaling [56], and for trellis codes with Quadrature Phase Shift Keying (QPSK), 8-Phase Shift Keying (8-PSK), 16 Quadrature Amplitude Modulation (16-QAM), or larger constellations [57].

For three 3-bit soft decision input bits, (i_0, i_1, i_2), each ranging from -3 to 3, eight 5-bit branch metrics are generated. The decision bits are represented in the two's complement representation. The BMU performs simple add and subtract operations on the decision bits to generate the output; for example, the branch metric for the state transition which produces the binary output (010) is i_0-i_1+i_2. The BMU performs the computations, as represented in Figure 6.5. The output of the BMU is still in a two's complement format. The bit serial format of the branch metrics is generated by the parallel to serial module at the output of the BMU, as shown in Figure 6.3. The bit serial format of the BMs is then fed into the ACSU.

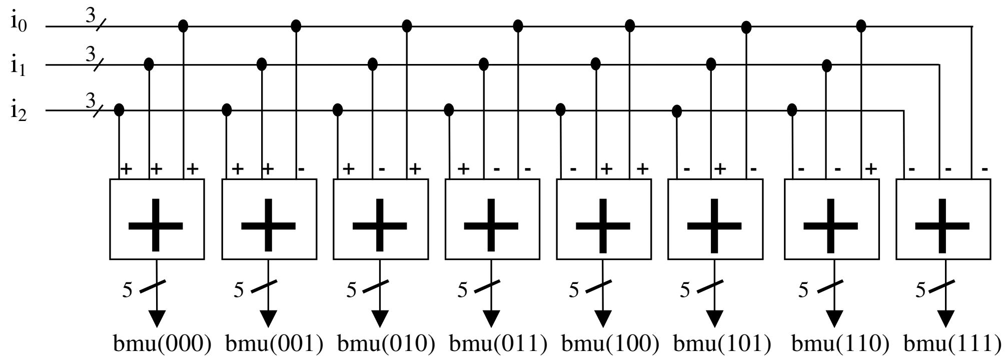


Figure 6.5: Branch Metric Unit (BMU) operations

6.1.3 Add Compare Select Unit(ACSU)

The ACSU is composed of 128 units; each is composed of an ACS butterfly module, as shown in Figure 6.6. The functionality of each ACS module is as described in Chapter 5, and the input PMs and BMs for the different ACSs are detailed in Appendix B. The routing among the different ACSU modules represents an overhead, which is left for the automatic optimization of the FPGA router.

6.1.4 Add Compare Select TO Survivor Memory (ACSTOSM)

The ACSTOSM is employed to route the decision of the appropriate ACS module to the output. The ACSTOSM is a 256 to one decoder. The select signal for this

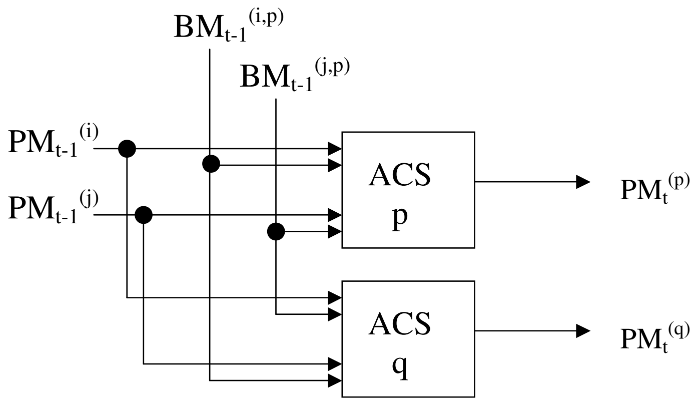


Figure 6.6: One ACS butterfly module

large decoder is the output of the pointer module. The output of the ACSTOSM module is already the decoded output sequence of the VD, but is fed back into the pointer module to update the current state in the decoding trellis.

6.1.5 Pointer

The pointer block contains the current state of the decoder (eight bits). For each bit decoded, the pointer content is updated. The exact position of the bit that will be updated is determined by the MSB block.

6.1.6 Most Significant Bit (MSB)

The MSB block has eight outputs that point to the eight bits of the pointer in parallel. At the reset, the eighth bit of the MSB block is enabled which, in turn, causes the eighth bit of the pointer to be the most significant bit. After accessing an incoming code word in the ACSU, a decision bit is written into the MSB of the pointer, and the output of the MSB block is shifted to the right so that only the seventh bit is enabled. For the pointer, this indicates that the seventh bit is now considered to be the most significant bit. In this case, pointer P should be read in the following order: $P_6P_5P_4P_3P_2P_1P_0P_7$, instead of the usual order: $P_7P_6P_5P_4P_3P_2P_1P_0$. This process continues with each incoming code word.

6.2 Power Consumption of the MLVD

To calculate the power reduction estimation, cost values for the MLVD operations are provided in Table 6.2. It is noted that many operations are no longer executed with the MLVD. The total in Table 6.2 shows that there is almost no power consumption that is associated with the SMU anymore. This occurs because all the SMU operations are related to the memory, and for the MLVD, no memory is required. Typically, the SMU consumes about 50 percent of the total VD power. Thus the MLVD reduces the power consumption of the VD by half. The MLVD has the same BER as the modified RE VD has. Table 6.3 demonstrates a quick comparison of the specification for the Traditional TB [36], the modified RE method [58], and the MLVD. To prepare the MLVD VHDL design that is implemented on the FPGA, the design is synthesized by using Synopsys tools. Then, the design is imported into Xilinx tools for the mapping and routing; then a VHDL file with timing information is re-simulated for the timing verification. The next two figures (Figure 6.7 and Figure 6.8) represent snapshots of the timing simulation for the MLVD.The last step is to generate the bit file that is downloaded on the FPGA chip.

Table 6.2: Estimated cost function to decode 48 codewords

Operation	TB	Modified RE	MLVD
Writing decision bits into the memory	256 x 48 $12288w$	$6144w$	–
Reading from the memory	48 x 3 r $72w$	$48r$ $24w$	–
Writing into the MSB of the pointers	–	256 x 48 rg $1536w$	$48rg$ $6w$
Writing the termination bits into the termination registers	–	$256rg$ $32w$	–
8-bit register shifting (shift to the right and append a bit to the LSB)	48 x 3 x $8rg$ $144w$	–	–
Total	$12504w$	$7736w$	$6w$

Table 6.3: Specifications comparison

Specification	TB	Modified RE	MLVD
Power Consumption	100%	80%	50%
Memory	2 x 48 x 256	48 x 256	–
Coding Gain	3.7dB	2.6dB	$\sim$2.6dB
Latency	$(L + D)$ x T_d	L x T_d	2 x T_d

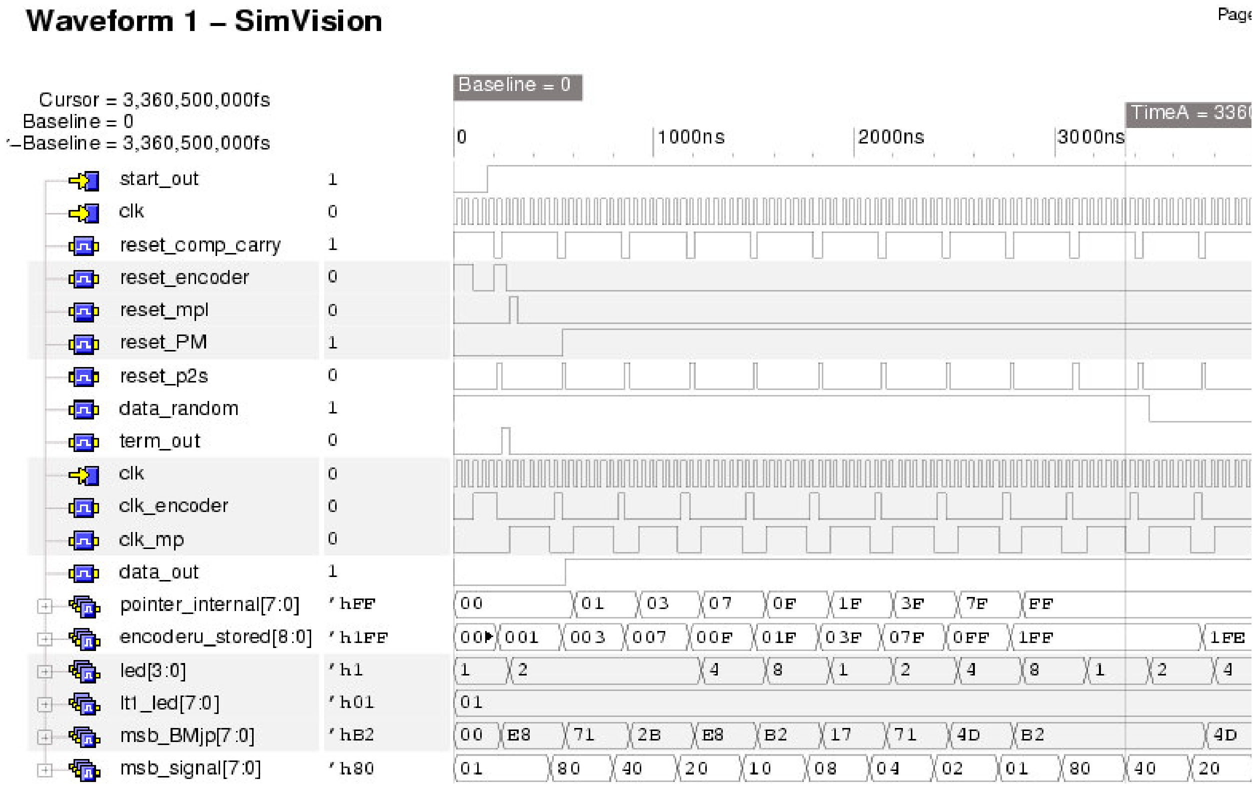
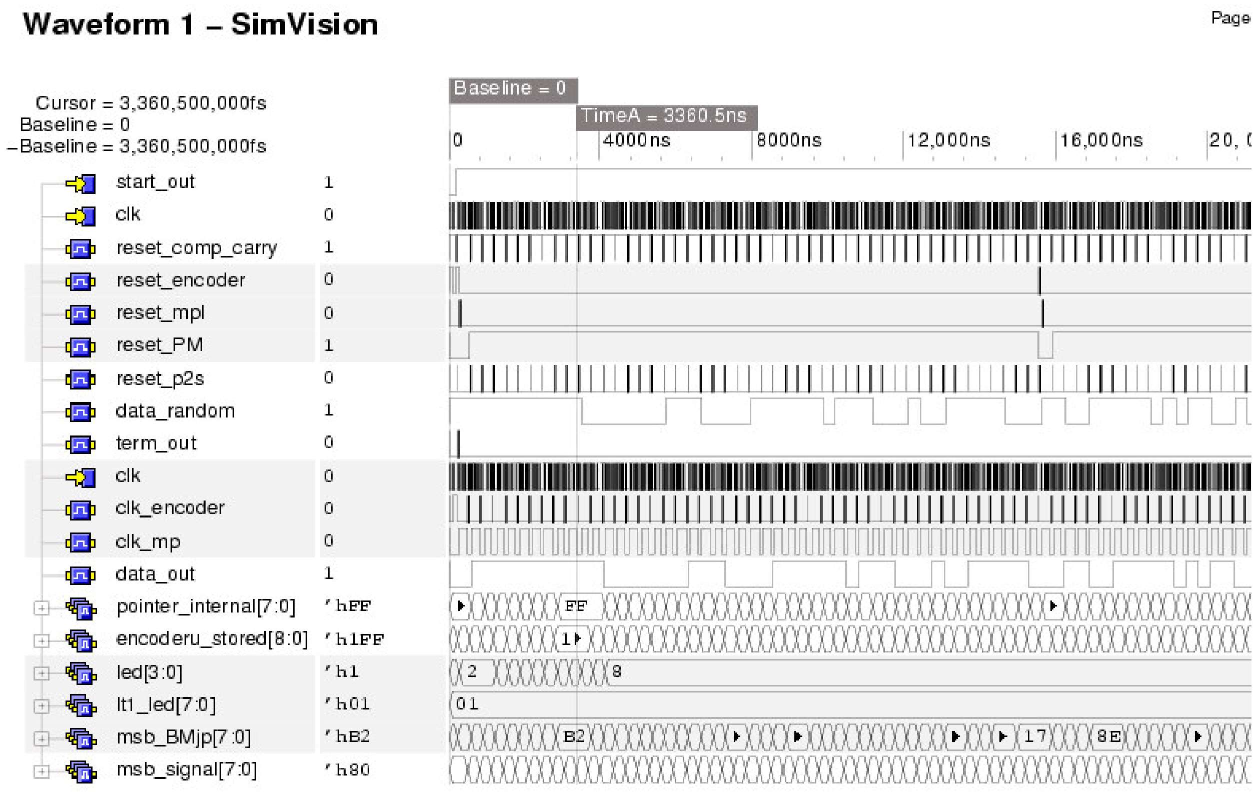


Figure 6.7: Snapshot of the timing simulation for the routed MLVD



Cursor = 3,360,500,000fs
Baseline = 0
–Baseline = 3,360,500,000fs

start_out	1
clk	0
reset_comp_carry	1
reset_encoder	0
reset_mpl	0
reset_PM	1
reset_p2s	0
data_random	1
term_out	0
clk	0
clk_encoder	0
clk_mp	0
data_out	1
pointer_internal[7:0]	'hFF
encoderu_stored[8:0]	'h1FF
led[3:0]	'h1
lt1_led[7:0]	'h01
msb_BMjp[7:0]	'hB2
msb_signal[7:0]	'h80

Figure 6.8: Snapshot of the timing simulation for the routed MLVD (zoom out)

6.3 Xilinx Implementation and Test Results

The MLVD is designed and implemented on a 2V6000 Xilinx chip. The chip is mounted on a rapid prototyping system provided to the University of Waterloo by CMC. The System includes the LT-XC2V6000 logic tile, which provides some leds to be connected to the output of the Xilinx Chip. These are used to flash serially in the case of the correct functionality. The MLVD consumes only 8 percent of the total slices of the 2V6000, comprising a total of 68,736 gates. The design consumes 16 I/O bits, and can be implemented on a much smaller FPGA, but was not available. Even the power consumption of the design on the FPGA is not very significant, because the main objective is to test the design feasibility and operability. The MLVD has a latency of two bits. The implemented design on the Xilinx FAPGA operates at 25 MHz with a decoding throughput of more than 3 $Mbps$. The detailed code is in Appendix A. Figure 6.9 shows the RPP system, mounted in a PC.

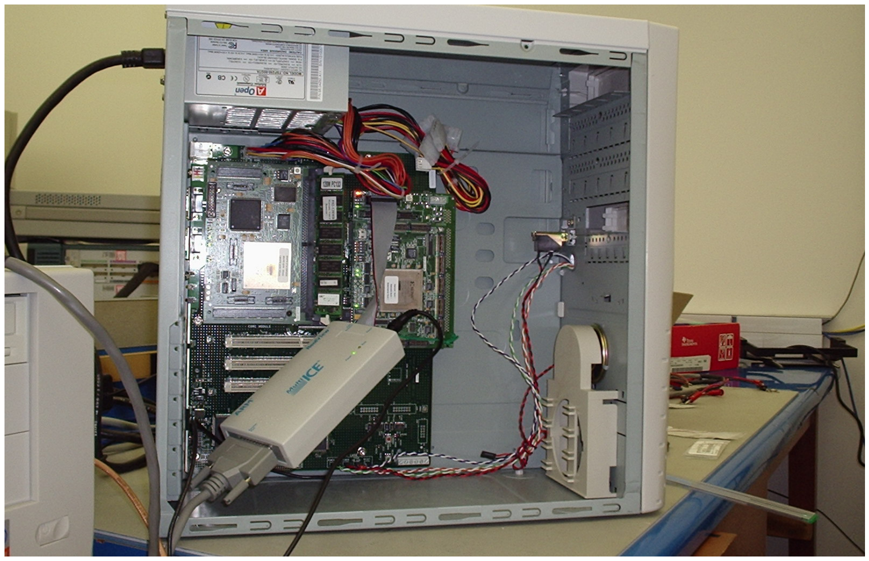


Figure 6.9: Rapid prototyping system

Chapter 7

Conclusions and Recommendations

Among the several architectures that are available to realize the VD, the RE method is conceptually the simplest, fastest, and most commonly used in VDs with only small values of K. The main drawback of the RE method is its power hungry implementation. In this work, the RE method even with large values of K (K=9) is modified to reduce the power consumption. The power reduction is as high as 50 percent, whereas the speed of the modified RE method is still higher than that of the TB method. An initial power reduction of 20 percent is realized by applying the pointer concept to the RE implementation. By reinforcing the initial state of the convolutional encoder and synchronizing the VD with the resetting procedure, a design, the MLVD, with the highest power reduction is realized (50 percent). The new MLVD is a memoryless high speed, low latency, and low power variation to the VD with an approximated BER of 10^{-5} and an SNR of 6.1 dB. The MLVD along with a convolutional encoder is implemented on a Xilinx 2V6000 chip to demonstrate both the design's functionality and feasible implementation.

For digital implementations, the Viterbi engine can be included in the core for some modern sigital signal processors (DSPs).For example, the wireless multimedia DSP chip in [59] supports the packed ACS byte instruction for Viterbi Decoding. The new MLVD in this thesis requires a slight modification in this wireless multimedia DSP and similar implementations in order to be executed by such DSPs.

This is true for 3G wireless technologies which are being produced as fast as expected because of the worldwide market downturn. Due to the low power and high speed challenges for the 4G wireless technologies, new developments are emerging. Analog decoders are proposed in the literature to replace digital decoders. So far, the implemented analog decoders have only a small number of states (as high as four) [60] [61]. These implementations assume that the VD is adjacent to the Analog to Digital Converter (ADC) in the receiver. This is not the case for the mobile terminal receiver. Thus, if the other components in the receiver, between the ADC and the VD, are also implemented in analog form, the ADC is shifted towards the decoders. Then, implementing an analog MLVD with 256 states is possible in the future.

Publications

[**1**] Dalia A. El-Dib and M.I. Elmasry, "Low-power register-exchange Viterbi decoder for high-speed wireless communications," *IEEE International Symposium on Circuits and Systems*, May 2002, pp. 737-740.

[**2**] Dalia A. El-Dib and M.I. Elmasry, "Modified Register-Exchange Viterbi Decoder for Low-Power Wireless Communications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 2, pp. 371- 378, February 2004.

[**3**] Dalia A. El-Dib and M.I. Elmasry, "Memoryless Viterbi Decoder: an extremely low power Viterbi Decoder," to be submitted in July 2004.

Bibliography

[1] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and B. Jacob, "A control-theoretic approach to dynamic voltage scheduling," *Proc., International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 255-266, 2003.

[2] A. Bellaouar and M. Elmasry, *Low Power Digital VLSI Design.* Kluwer Academic Publishers, 1995.

[3] S.-M. Kang, "Elements of low power design for integrated systems," *Proc. International Symposium on Low Power Design and Electronics*, pp. 205-210, 2003.

[4] A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design.* Kluwer Academic Publishers, 1995.

[5] K. Roy and et al., "Leakage control for deep-submicron circuits," *Proc. of SPIE, VLSI Circuits and Systems*, vol. 5117, pp. 135-146, 2003.

[6] T. Ghani and et al., "Scaling challenges and device design requirements for high performance sub-50nm gate length planar cmos transistors," *VLSI Technology, Digest of Technical Papers*, pp. 174-175, 2000.

[7] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, pp. 473-484, April 1992.

[8] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital cmos circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, April 1995.

[9] A. Maheshwari, W. Burleson, and R. Tessier, "Trading off transient fault tolerance and power consumption in deep submicron (dsm)vlsi circuits," *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, vol. 12, no. 3, pp. 299-311, March 2004.

[10] A. Branover, R. Kol, and R. Ginosar, "Asynchronous design by conversion: converting synchronous circuits into asynchronous ones," *Proc., Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, pp. 870-875, 2004.

[11] V. Garg, *IS-95 CDMA and Cdma2000-Cellular/PCS Systems Implementation.* Upper Saddle River, NJ: Prentice Hall, 1999.

[12] *CDMA technology and benefits - An introduction to the benefits of CDMA for wireless telephony*, Motorola Inc., March 1996.

[13] Q. Inc., *An overview of the application of code division multiple access (CDMA) to digital cellular systems and personal cellular networks*, May 1992.

[14] Cooper and R. Nettleton, "A spread-spectrum technique for high capacity mobile communications," *IEEE Trans. Veh. Tech.*, vol. 27, no. 4, pp. 264-275, 1978.

[15] R. Prasad and T. Ojanpera, "An overview of cdma evolution toward wideband cdma," *IEEE Communications Surveys*, vol. 1, no. 1, Fourth Quarter, 1998.

[16] *A History of Third Generation Mobile 3G*, Nokia Networks, March 2003.

[17] *Implementing a W-CDMA System with Altera Devices IP Functions*, Altera Corporation, September 2000.

[18] S. Ranpara, "On a viterbi decoder design for low power dissipation," Master's thesis, Virginia Polytechnic Institute and State University, 1999.

[19] A. Viterbi, "Error bounds for convolutional codes and asymptotically optimum decoding algorithm," *IEEE Transactions on Information theory*, vol. It-13, no. 2, pp. 260–269, April 1967.

[20] J. Heller and I. Jacobs, "Viterbi decoding for satellite and space communication," *IEEE Transactions on Communication Technology*, vol. COM–19, no. 5, pp. 835–848, Oct. 1971.

[21] G. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

[22] J. Ryu, S. Kim, J. Cho, H. Park, and Y. Y.H. Chang, "Lower power viterbi decoder architecture with a new clock-gating trace-back unit," *Proc. 6th International Conference on VLSI and CAD*, pp. 297-300, Oct. 1999.

[23] S. B. Wicker, *Error Control Systems for Digital Communication and Storage.* Prentice Hall, 1995.

[24] H.-D. Lin and D. Messerschmitt, “Algorithms and architectures for concurrent viterbi decoding,” *Proc. IEEE International Conference on Communications*, pp. 836-840, June 1989.

[25] P. Black and T.-Y. Meng, “A 140 mb/s 32-state radix-4 viterbi decoder,” *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp. 1877-1885, December 1992.

[26] A. Yeung and J. Rabaey, “A 210 mb/s radix-4 bit-level pipelined viterbi decoder,” *Digest of Technical Papers, Proc. International Solid-State Circuits Conference*, February 1995.

[27] P. Black and T.-Y. Meng, “A 1-gb/s, four-state, sliding block viterbi decoder,” *IEEE Journal of Solid-State Circuits*, vol. 32, no. 6, pp. 797-805, June 1997.

[28] K. Page and P. Chau, “Improved architectures for the add-compare-select operation in long constraint length viterbi decoding,” *IEEE Journal of Solid-State Circuits*, vol. 33, no. 1, pp. 151-155, January 1998.

[29] C.-Y. Tsui, R.-K. Cheng, and C. Ling, “Low power acs unit design for the viterbi decoder,” *Proc. IEEE International Symposium on Circuits and Systems*, pp. 137-140, 1999.

[30] D.-I. Oh and S.-Y. Hwang, “Design of a viterbi decoder with low power using minimum-transition traceback scheme,” *IEE Electronics Letters*, vol. 32, pp. 2198-2199, November 1996.

[31] M.-B. Lin, “New path history management circuits for viterbi decoders,” *IEEE Transactions on Communications*, vol. 48, no. 10, pp. 1605-1608, October 2000.

[32] G. Fettweis and H. Meyr, “Feedforward architectures for parallel viterbi decoding,” *Journal of VLSI Signal Processing*, vol. 3, pp. 105-119, 1991.

[33] S.-J. Jung, M.-H. Lee, and H.-J. Choi, “A new survivor memory management method in viterbi decoders: trace-delete method and its implementation,” *Proc. ICASSP*, pp. 3284-3286, 1996.

[34] P. Black and T.-Y. Meng, “Hybrid survivor path architectures for viterbi decoders,” *Proc. ICASSP*, pp. 433-436, 1993.

[35] J. H. et al, “”cdma mobile station modem asic”,” *IEEE Journal of Solid-State Circuits*, vol. 28, no. 3, pp. 253-260, March 1993.

[36] Y.-N. Chang, H. Suzuki, and K. Parhi, “A 2-mb/s 256-state 10-mw rate-1/3 viterbi decoder,” *IEEE Journal of Solid-State Circuits*, vol. 35, no. 6, pp. 826-834, June 2000.

[37] I. Kang and A. W. Jr, “Low-power viterbi decoder for cdma mobile terminals,” *IEEE Journal of Solid-State Circuits*, vol. 33, no. 3, pp. 473-482, March 1998.

[38] J. Sparso, H. Jorgensen, E. Paaske, S. Pedersen, and T. Rubner-Petersen, “An area-efficient topology for vlsi implementation of viterbi decoders and other shuffle-exchange type structures,” *IEEE Journal of Solid-State Circuits*, vol. 26, no. 2, pp. 90-97, February 1991.

[39] S. Das, E. Erkip, J. Cavallaro, and B. Aazhang, “Maximum weight basis decoding of convolutional codes,” *Proc. Global Telecommunication Conference*, pp. 835-841, 2000.

[40] G. Pottie, “Low latency sequential decoding,” *Proc. IEEE International Symopsium on Information Theory*, p. 499, 1997.

[41] C. Feldmann and J. Harris, “A constraint-length based modified viterbi algorithm with adaptive effort,” *IEEE Transactions on Communications*, vol. 47, no. 11, pp. 1611-1614, Nov. 1999.

[42] F. Chan and D. Haccoun, “Adaptive viterbi decoding of convolutional codes over memoryless channels,” *IEEE Transactions on Communications*, vol. 45, no. 11, pp. 1389-1400, Nov. 1997.

[43] R. Henning and C. Chakrabarti, “Low-power approach for decoding convolutional codes with adaptive viterbi algorithm approximations,” *Proc. IEEE International Symposium on Lower Power Electronics and Design*, pp. 68-71, August 2002.

[44] M. Guo, M. O. Ahmad, M. Swamy, and C. Wang, “A low-power systolic array-based adaptive viterbi decoder and its fpga implementation,” *Proc. IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 276-279, May 2003.

[45] D. Liu and C. Svensson, “Power consumption estimation in cmos vlsi chips,” *IEEE Journal of Solid-State Circuits*, vol. 29, no. 6, pp. 663-670, June 1994.

[46] C. Svensson, Private correspondence, Department of Physics and Measurement Technology, Linkping Univ., Sweden.

[47] G. Fettweis and H. Meyr, "High-rate viterbi processor: a systolic array solution," *IEEE Journal on Selected Areas in Communications*, pp. 1520-1534, Oct. 1990.

[48] H. Lou, "Viterbi decoder design for the is-95 cdma forward link," *Proc. Vehicular Technology Conference*, pp. 1346-1350, April 1996.

[49] B. Min and N. Demassieux, "A versatile architecture for vlsi implementation of the viterbi algorithm," *Proc. International Conference on Acoustics, Speech and Signal Processing*, pp. 1101-1104, May 1991.

[50] M. Biver, H. Kaeslin, and C. Tommasini, "In-place updating of path metrics in viterbi decoders," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 4, pp. 1158-1160, August 1989.

[51] Y.-N. Chang, "An efficient in-place vlsi architecture for viterbi algorithm," *Journal of VLSI Signal Processing*, vol. 33, no. 3, pp. 317, March 2003.

[52] C. Nicol, P. Larsson, K. Azadet, and J. O'Neill, "A low-power 128-tap digital adaptive equalizer for broadband modems," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 11, pp. 1777-1789, November 1997.

[53] A. Shams and M. Bayoumi, "A novel high-performance cmos 1-bit full-adder cell," *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 478-481, May 2000.

[54] A. Shams and M. Bayoumi, "A novel low-power building block cmos cell for adders," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 153-156, 1998.

[55] A. Fayed and M. Bayoumi, "A low power 10-transistor full adder cell for embedded architectures," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 226-229, 2001.

[56] H.-L. Lou, "Implementing the viterbi algorithm," *IEEE Signal Processing Magazine*, vol. 12, pp. 42-52, September 1995.

[57] H.-L. Lou, "Linear distances as branch metrics for viterbi decoding of trellis codes," *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 6, pp. 3267-3270, June 2000.

[58] D. A. El-Dib and M. I. Elmasry, "Modified register-exchange viterbi decoder for low-power wireless communications," *IEEE Transactions on Circuits and Systems I*, vol. 51, no. 2, pp. 371-378, February 2004.

[59] K. L. Heo, M. H. Sunwoo, and S. K. Oh, "Implementation of a wireless multimedia dsp chip for mobile applications," *IEEE Workshop on Signal Processing Systems*, pp. 51-56, August 2003.

[60] V. Gaudet and P. Gulak, "A 13,3-mb/s 0.35-μm cmos analog turbo decoder ic with a configurable interleaver," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 2010-2015, November 2003.

[61] A. Demosthenous and J. Taylor, "A 100-mb/s 2.8-v cmosp current-mode analog viterbi decoder," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 7, pp. 904-910, July 2002.

Appendix A

VHDL code for the Memoryless Viterbi Decoder

A.1 Lfsr.vhd

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

entity lfsr is
port (  clock : std_logic;
        reset : std_logic;
        data_out : out std_logic);
end lfsr;

-- The linear feedback shift register serves the generation
-- of random input to the encoder.

architecture rtl of lfsr is
signal lfsr_reg : std_logic_vector(9 downto 0);
begin
process (clock,reset)
    variable lfsr_tap : std_ulogic;
 begin
    if reset = '1' then
        lfsr_reg <= (others =>'1');
        data_out <= '1';
    else
      if clock'EVENT and clock='1' then
        lfsr_tap := lfsr_reg(6) xor lfsr_reg(9);
        lfsr_reg <= lfsr_reg(8 downto 0) & lfsr_tap;
        data_out <= lfsr_reg(9);
      end if;
    end if;
end process;
end rtl;
```

A.2 Encoder.vhd

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

entity encoder is
port(   data_in, reset, clk: in std_logic;
        i0,i1,i2: out std_logic_vector(2 downto 0)
    );
end encoder;

-- This is the code for K=9 r=1/3 encoder.
-- The output is converted into soft decision, such that:
-- 0 --> -3 (101)
-- 1 --> +3 (011)

architecture rtl of encoder is
    signal stored: std_logic_vector(0 to 8);
    signal code: std_logic_vector(0 to 2);
begin

process(reset,clk,stored,data_in)
 begin
 if clk='1' and clk'event then
    if reset ='1' then
        stored(0)<=data_in;
        stored(1 to 8)<="00000000";
     else
        stored<=data_in & stored(0 to 7);
     end if;
 end if;
end process;

code(0)<=stored(0) XOR stored(2) XOR stored(3) XOR stored(5) XOR
         stored(6) XOR  stored(7) XOR stored(8);
code(1)<=stored(0) XOR stored(1) XOR stored(3) XOR stored(4) XOR
         stored(7) XOR  stored(8);
code(2)<=stored(0) XOR stored(1) XOR stored(2) XOR stored(5) XOR
         stored(8);

process(code)
begin
-- the following is to present code in the range (-3 to 3)
 case code is
    when "000" => i0<="101"; i1<="101";i2<="101";
    when "001" => i0<="101"; i1<="101";i2<="011";
    when "010" => i0<="101"; i1<="011";i2<="101";
    when "011" => i0<="101"; i1<="011";i2<="011";
    when "100" => i0<="011"; i1<="101";i2<="101";
    when "101" => i0<="011"; i1<="101";i2<="011";
    when "110" => i0<="011"; i1<="011";i2<="101";
    when "111" => i0<="011"; i1<="011";i2<="011";
    when others => i0<="000"; i1<="000";i2<="000";
 end case;
end process;
end rtl;
```

A.3 Bmu.vhd

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

-- The BMU is operating at the bit rate. It receives soft-decision
-- decoded bits in the range {-3,3}.
-- The output is a signed 5-bit number (in 2's complement format)

entity bmu is
port(   i0, i1, i2: in std_logic_vector (2 downto 0);
        bmu000, bmu001, bmu010, bmu011, bmu100, bmu101,
        bmu110, bmu111: out std_logic_vector (4 downto 0)
     );
end bmu;

architecture rtl of bmu is
begin
process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu000<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu001<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu010<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu011<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu100<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu101<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu110<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu111<=temp000_1;
end process;
end rtl;
```

A.4 Parallel-to-serial.vhd

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

entity p2s is
port(   reset,clk: in std_logic;
        din: in std_logic_vector (4 downto 0);
        serout: out std_logic
        );
end p2s;

-- This is a parallel to serial converter for converting the
-- parallel output of the bmu to serial format.

architecture rtl of p2s is
-- the clk is working at 8 times the bit rate, while the reset
-- signal works at the bit rate only.
    signal def1: std_logic_vector(4 downto 0);
    signal start, signbit: std_logic;
begin

process (reset,din,clk)
    variable def2: std_logic_vector(4 downto 0);
 begin
 if reset ='1' then
   def1<=din;
```

```
    start<=din(0);
    signbit<=din(4);
  elsif clk ='1' and clk'event then
    def2:=signbit & def1(4 downto 1) ;
    start<=def1(1);
    def1<=def2;
 end if;
end process;
serout<=start;
end rtl;

library ieee;
use ieee.std_logic_1164.all;

entity p2s_block is
port(    reset,clk: in std_logic;
          bmu000, bmu001, bmu010, bmu011, bmu100, bmu101,
             bmu110, bmu111: in std_logic_vector (4 downto 0);
          BM: out std_logic_vector (0 to 7)
      );
end p2s_block;

-- This module contains eight units of the parallel to serial,
-- component which is responsible for converting the parallel
-- output of the BMU to a serial one.

architecture struct of p2s_block is
-- the clk is working at 8 times the bit rate, while the reset
-- signal works at the bit rate only.

component p2s
port(    reset,clk: in std_logic;
         din: in std_logic_vector (4 downto 0);
         serout: out std_logic
      );
end component;

begin
 p2s0: p2s port map (reset,clk, bmu000,BM(0));
 p2s1: p2s port map (reset,clk, bmu001,BM(1));
 p2s2: p2s port map (reset,clk, bmu010,BM(2));
 p2s3: p2s port map (reset,clk, bmu011,BM(3));
 p2s4: p2s port map (reset,clk, bmu100,BM(4));
 p2s5: p2s port map (reset,clk, bmu101,BM(5));
 p2s6: p2s port map (reset,clk, bmu110,BM(6));
 p2s7: p2s port map (reset,clk, bmu111,BM(7));
end struct;
```

A.5 Acsu.vhd

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

entity acsu is
port (PMi, BMip, PMj, BMjp, clk, reset_carry, clk_sub: in std_logic;
      PM, comp_out : out std_logic
       );
```

```
end acsu;

-- This module represents one unit of the add compare select operation.

architecture rtl of acsu is

signal resultc,resulta_internal,resulta_bar,cina: std_logic;
signal resultb_internal, cinb: std_logic;
signal cinc, selection,clk_data: std_logic;
signal resulta, resultb: std_logic_vector(0 to 7);
 begin
clk_data<= clk;

-- This process represents the first and second FIFO and the latches
-- for the carry of the three FAs and also the selection of the
-- smaller PM according to the MSB of the subtraction.

process (clk_data, resultc, resulta_bar, BMjp, PMj, resulta_bar, selection,
        cinb, cinc, clk_sub,reset_carry, clk, BMip, PMi, cina,
        resulta_internal, resultb_internal,resulta, resultb)

 begin
 if reset_carry='1' then
    cina<='0';
    cinb<='0';
    cinc<='0';
 elsif clk_data='0'and  clk_data'event then
    cina<= ((PMi AND BMip) OR (cina  AND PMi) OR (cina AND BMip)) ;
    cinb<= ((PMj AND BMjp) OR (cinb  AND PMj) OR (cinb  AND BMjp)) ;
    cinc<= ((resulta_bar AND resultb_internal) OR (cinc  AND resulta_bar) OR
            (cinc AND resultb_internal)) ;
 end if;
if clk='1' then
-- first FA
resulta_internal<=PMi XOR BMip XOR (cina);
resulta_bar<=not(PMi XOR BMip XOR (cina));


-- second FA
resultb_internal<=PMj XOR BMjp XOR (cinb  );


-- third FA, which acts as a subtractor
resultc <= resulta_bar XOR resultb_internal XOR (cinc  );

end if;
end process;

process (clk_data,resultc,resulta_bar,BMjp, PMj,resulta_bar,selection, cinb,
        cinc, clk_sub,reset_carry, clk, BMip, PMi, cina, resulta_internal,
        resultb_internal,resulta, resultb)
begin
if  clk='0' and clk'event  then


 resulta(0 to 7)<= resulta_internal&resulta(0 to 6);
 resultb(0 to 7)<= resultb_internal&resultb(0 to 6);

 if clk_sub='1' then
```

```
  selection<= not(resultc);
 -- comp_out<=not(resultc);
end if;
end if;

end process;
  comp_out<=selection;

PM<=resulta(7) when selection='1' else resultb(7);
end rtl;

library ieee;
use ieee.std_logic_1164.all;

entity acsu_pair is
port (PMi, BMip, PMj, BMjp, clk, reset_carry : in std_logic;
      msb_BMjp, comp_enable, reset_PM: in std_logic;
      PMp, PMq, termi, termj, deci, decj : out std_logic
      );
end acsu_pair;

-- This module represents one butterfly unit, which contains a pair
-- of the ACSU components.

architecture rtl of acsu_pair is
component acsu
port (PMi, BMip, PMj, BMjp, clk, reset_carry, clk_sub: in std_logic;
      PM, comp_out : out std_logic
      );
end component;
signal PMi_internal, PMj_internal,comp1, comp2: std_logic;
begin
acsu1: acsu port map(PMi_internal, BMip, PMj_internal, BMjp, clk, reset_carry,
            comp_enable, PMp, comp1);
acsu2: acsu port map(PMi_internal, BMjp, PMj_internal, BMip, clk, reset_carry,
            comp_enable, PMq, comp2);

process (PMi, PMj, reset_PM, clk)
begin
If (clk='1' and clk'event)  then
-- (PMi and PMj need to be reset with each reset of the decoder)
PMi_internal<=PMi AND reset_PM;
PMj_internal<=PMj AND reset_PM;
end if;
end process;
termi<=(not(comp1)) and (not(comp2));
termj<=comp1 and comp2;

deci<=(not(comp1) and comp2) or (msb_BMjp and (not(comp1)) and (not(comp2))) or
        ((msb_BMjp) and comp1 and comp2);
decj<=(not(comp2) and comp1) or (not(msb_BMjp) and (not(comp1))and(not(comp2)))
        or (not(msb_BMjp) and comp1 and comp2);
end rtl;

library ieee;
use ieee.std_logic_1164.all;

entity acsu_block is
port (BM, msb_BMjp: std_logic_vector(0 to 7);
      clk, reset_carry : in std_logic;
      comp_enable, reset_PM: in std_logic;
      term, dec : out std_logic_vector(0 to 255)
      );
end acsu_block;

-- This module represents the whole ACSU. It contains 126 units of the
-- acsu component.

architecture struct of acsu_block is
component acsu_pair
port (PMi, BMip, PMj, BMjp, clk, reset_carry : in std_logic;
      msb_BMjp, comp_enable, reset_PM: in std_logic;
      PMp, PMq, termi, termj, deci, decj : out std_logic
      );
end component;
signal PM: std_logic_vector(0 to 255);

begin
acsu_pair1  : acsu_pair port map
              (PM(0), BM(0), PM(128), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
               reset_PM, PM(0), PM(1), term(0), term(128), dec(0), dec(128));
acsu_pair2  : acsu_pair port map
              (PM(1), BM(3), PM(129), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
               reset_PM, PM(2), PM(3), term(1), term(129), dec(1), dec(129));
acsu_pair3  : acsu_pair port map
              (PM(2), BM(5), PM(130), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
               reset_PM, PM(4), PM(5), term(2), term(130), dec(2), dec(130));
acsu_pair4  : acsu_pair port map
              (PM(3), BM(6), PM(131), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
               reset_PM, PM(6), PM(7), term(3), term(131), dec(3), dec(131));
acsu_pair5  : acsu_pair port map
              (PM(4), BM(6), PM(132), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
               reset_PM, PM(8), PM(9), term(4), term(132), dec(4), dec(132));
acsu_pair6  : acsu_pair port map
              (PM(5), BM(5), PM(133), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
               reset_PM, PM(10), PM(11), term(5), term(133), dec(5), dec(133));
acsu_pair7  : acsu_pair port map
              (PM(6), BM(3), PM(134), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
               reset_PM, PM(12), PM(13), term(6), term(134), dec(6), dec(134));
acsu_pair8  : acsu_pair port map
              (PM(7), BM(0), PM(135), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
               reset_PM, PM(14), PM(15), term(7), term(135), dec(7), dec(135));
acsu_pair9  : acsu_pair port map
              (PM(8), BM(2), PM(136), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
               reset_PM, PM(16), PM(17), term(8), term(136), dec(8), dec(136));
acsu_pair10 : acsu_pair port map
              (PM(9), BM(1), PM(137), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
               reset_PM, PM(18), PM(19), term(9), term(137), dec(9), dec(137));
acsu_pair11 : acsu_pair port map
              (PM(10), BM(7), PM(138), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
               reset_PM, PM(20), PM(21), term(10), term(138), dec(10), dec(138));
acsu_pair12 : acsu_pair port map
              (PM(11), BM(4), PM(139), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
               reset_PM, PM(22), PM(23), term(11), term(139), dec(11), dec(139));
acsu_pair13 : acsu_pair port map
              (PM(12), BM(4), PM(140), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
               reset_PM, PM(24), PM(25), term(12), term(140), dec(12), dec(140));
acsu_pair14 : acsu_pair port map
```

```
                  (PM(13), BM(7), PM(141), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                   reset_PM, PM(26), PM(27), term(13), term(141), dec(13), dec(141));
    acsu_pair15 : acsu_pair port map
                  (PM(14), BM(1), PM(142), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                   reset_PM, PM(28), PM(29), term(14), term(142), dec(14), dec(142));
    acsu_pair16 : acsu_pair port map
                  (PM(15), BM(2), PM(143), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                   reset_PM, PM(30), PM(31), term(15), term(143), dec(15), dec(143));
    acsu_pair17 : acsu_pair port map
                  (PM(16), BM(5), PM(144), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
                   reset_PM, PM(32), PM(33), term(16), term(144), dec(16), dec(144));
    acsu_pair18 : acsu_pair port map
                  (PM(17), BM(6), PM(145), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
                   reset_PM, PM(34), PM(35), term(17), term(145), dec(17), dec(145));
    acsu_pair19 : acsu_pair port map
                  (PM(18), BM(0), PM(146), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
                   reset_PM, PM(36), PM(37), term(18), term(146), dec(18), dec(146));
    acsu_pair20 : acsu_pair port map
                  (PM(19), BM(3), PM(147), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
                   reset_PM, PM(38), PM(39), term(19), term(147), dec(19), dec(147));
    acsu_pair21 : acsu_pair port map
                  (PM(20), BM(3), PM(148), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
                   reset_PM, PM(40), PM(41), term(20), term(148), dec(20), dec(148));
    acsu_pair22 : acsu_pair port map
                  (PM(21), BM(0), PM(149), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
                   reset_PM, PM(42), PM(43), term(21), term(149), dec(21), dec(149));
    acsu_pair23 : acsu_pair port map
                  (PM(22), BM(6), PM(150), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
                   reset_PM, PM(44), PM(45), term(22), term(150), dec(22), dec(150));
    acsu_pair24 : acsu_pair port map
                  (PM(23), BM(5), PM(151), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
                   reset_PM, PM(46), PM(47), term(23), term(151), dec(23), dec(151));
    acsu_pair25 : acsu_pair port map
                  (PM(24), BM(7), PM(152), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                   reset_PM, PM(48), PM(49), term(24), term(152), dec(24), dec(152));
    acsu_pair26 : acsu_pair port map
                  (PM(25), BM(4), PM(153), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                   reset_PM, PM(50), PM(51), term(25), term(153), dec(25), dec(153));
    acsu_pair27 : acsu_pair port map
                  (PM(26), BM(2), PM(154), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                   reset_PM, PM(52), PM(53), term(26), term(154), dec(26), dec(154));
    acsu_pair28 : acsu_pair port map
                  (PM(27), BM(1), PM(155), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                   reset_PM, PM(54), PM(55), term(27), term(155), dec(27), dec(155));
    acsu_pair29 : acsu_pair port map
                  (PM(28), BM(1), PM(156), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                   reset_PM, PM(56), PM(57), term(28), term(156), dec(28), dec(156));
    acsu_pair30 : acsu_pair port map
                  (PM(29), BM(2), PM(157), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                   reset_PM, PM(58), PM(59), term(29), term(157), dec(29), dec(157));
    acsu_pair31 : acsu_pair port map
                  (PM(30), BM(4), PM(158), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                   reset_PM, PM(60), PM(61), term(30), term(158), dec(30), dec(158));
    acsu_pair32 : acsu_pair port map
                  (PM(31), BM(7), PM(159), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                   reset_PM, PM(62), PM(63), term(31), term(159), dec(31), dec(159));
    acsu_pair33 : acsu_pair port map
                  (PM(32), BM(4), PM(160), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                   reset_PM, PM(64), PM(65), term(32), term(160), dec(32), dec(160));
```

```
acsu_pair34 : acsu_pair port map
              (PM(33), BM(7), PM(161), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
               reset_PM, PM(66), PM(67), term(33), term(161), dec(33), dec(161));
acsu_pair35 : acsu_pair port map
              (PM(34), BM(1), PM(162), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
               reset_PM, PM(68), PM(69), term(34), term(162), dec(34), dec(162));
acsu_pair36 : acsu_pair port map
              (PM(35), BM(2), PM(163), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
               reset_PM, PM(70), PM(71), term(35), term(163), dec(35), dec(163));
acsu_pair37 : acsu_pair port map
              (PM(36), BM(2), PM(164), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
               reset_PM, PM(72), PM(73), term(36), term(164), dec(36), dec(164));
acsu_pair38 : acsu_pair port map
              (PM(37), BM(1), PM(165), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
               reset_PM, PM(74), PM(75), term(37), term(165), dec(37), dec(165));
acsu_pair39 : acsu_pair port map
              (PM(38), BM(7), PM(166), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
               reset_PM, PM(76), PM(77), term(38), term(166), dec(38), dec(166));
acsu_pair40 : acsu_pair port map
              (PM(39), BM(4), PM(167), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
               reset_PM, PM(78), PM(79), term(39), term(167), dec(39), dec(167));
acsu_pair41 : acsu_pair port map
              (PM(40), BM(6), PM(168), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
               reset_PM, PM(80), PM(81), term(40), term(168), dec(40), dec(168));
acsu_pair42 : acsu_pair port map
              (PM(41), BM(5), PM(169), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
               reset_PM, PM(82), PM(83), term(41), term(169), dec(41), dec(169));
acsu_pair43 : acsu_pair port map
              (PM(42), BM(3), PM(170), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
               reset_PM, PM(84), PM(85), term(42), term(170), dec(42), dec(170));
acsu_pair44 : acsu_pair port map
              (PM(43), BM(0), PM(171), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
               reset_PM, PM(86), PM(87), term(43), term(171), dec(43), dec(171));
acsu_pair45 : acsu_pair port map
              (PM(44), BM(0), PM(172), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
               reset_PM, PM(88), PM(89), term(44), term(172), dec(44), dec(172));
acsu_pair46 : acsu_pair port map
              (PM(45), BM(3), PM(173), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
               reset_PM, PM(90), PM(91), term(45), term(173), dec(45), dec(173));
acsu_pair47 : acsu_pair port map
              (PM(46), BM(5), PM(174), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
               reset_PM, PM(92), PM(93), term(46), term(174), dec(46), dec(174));
acsu_pair48 : acsu_pair port map
              (PM(47), BM(6), PM(175), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
               reset_PM, PM(94), PM(95), term(47), term(175), dec(47), dec(175));
acsu_pair49 : acsu_pair port map
              (PM(48), BM(1), PM(176), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
               reset_PM, PM(96), PM(97), term(48), term(176), dec(48), dec(176));
acsu_pair50 : acsu_pair port map
              (PM(49), BM(2), PM(177), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
               reset_PM, PM(98), PM(99), term(49), term(177), dec(49), dec(177));
acsu_pair51 : acsu_pair port map
              (PM(50), BM(4), PM(178), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
               reset_PM, PM(100), PM(101), term(50), term(178), dec(50), dec(178));
acsu_pair52 : acsu_pair port map
              (PM(51), BM(7), PM(179), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
               reset_PM, PM(102), PM(103), term(51), term(179), dec(51), dec(179));
acsu_pair53 : acsu_pair port map
              (PM(52), BM(7), PM(180), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
               reset_PM , PM(104) , PM(105) , term(52) , term(180) , dec(52) , dec(180));
acsu_pair54 : acsu_pair port map
              (PM(53) , BM(4) , PM(181) , BM(3) , clk , reset_carry , msb_BMjp(3) , comp_enable ,
               reset_PM , PM(106) , PM(107) , term(53) , term(181) , dec(53) , dec(181));
acsu_pair55 : acsu_pair port map
              (PM(54) , BM(2) , PM(182) , BM(5) , clk , reset_carry , msb_BMjp(5) , comp_enable ,
               reset_PM , PM(108) , PM(109) , term(54) , term(182) , dec(54) , dec(182));
acsu_pair56 : acsu_pair port map
              (PM(55) , BM(1) , PM(183) , BM(6) , clk , reset_carry , msb_BMjp(6) , comp_enable ,
               reset_PM , PM(110) , PM(111) , term(55) , term(183) , dec(55) , dec(183));
acsu_pair57 : acsu_pair port map
              (PM(56) , BM(3) , PM(184) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
               reset_PM , PM(112) , PM(113) , term(56) , term(184) , dec(56) , dec(184));
acsu_pair58 : acsu_pair port map
              (PM(57) , BM(0) , PM(185) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
               reset_PM , PM(114) , PM(115) , term(57) , term(185) , dec(57) , dec(185));
acsu_pair59 : acsu_pair port map
              (PM(58) , BM(6) , PM(186) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
               reset_PM , PM(116) , PM(117) , term(58) , term(186) , dec(58) , dec(186));
acsu_pair60 : acsu_pair port map
              (PM(59) , BM(5) , PM(187) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
               reset_PM , PM(118) , PM(119) , term(59) , term(187) , dec(59) , dec(187));
acsu_pair61 : acsu_pair port map
              (PM(60) , BM(5) , PM(188) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
               reset_PM , PM(120) , PM(121) , term(60) , term(188) , dec(60) , dec(188));
acsu_pair62 : acsu_pair port map
              (PM(61) , BM(6) , PM(189) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
               reset_PM , PM(122) , PM(123) , term(61) , term(189) , dec(61) , dec(189));
acsu_pair63 : acsu_pair port map
              (PM(62) , BM(0) , PM(190) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
               reset_PM , PM(124) , PM(125) , term(62) , term(190) , dec(62) , dec(190));
acsu_pair64 : acsu_pair port map
              (PM(63) , BM(3) , PM(191) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
               reset_PM , PM(126) , PM(127) , term(63) , term(191) , dec(63) , dec(191));
acsu_pair65 : acsu_pair port map
              (PM(64) , BM(6) , PM(192) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
               reset_PM , PM(128) , PM(129) , term(64) , term(192) , dec(64) , dec(192));
acsu_pair66 : acsu_pair port map
              (PM(65) , BM(5) , PM(193) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
               reset_PM , PM(130) , PM(131) , term(65) , term(193) , dec(65) , dec(193));
acsu_pair67 : acsu_pair port map
              (PM(66) , BM(3) , PM(194) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
               reset_PM , PM(132) , PM(133) , term(66) , term(194) , dec(66) , dec(194));
acsu_pair68 : acsu_pair port map
              (PM(67) , BM(0) , PM(195) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
               reset_PM , PM(134) , PM(135) , term(67) , term(195) , dec(67) , dec(195));
acsu_pair69 : acsu_pair port map
              (PM(68) , BM(0) , PM(196) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
               reset_PM , PM(136) , PM(137) , term(68) , term(196) , dec(68) , dec(196));
acsu_pair70 : acsu_pair port map
              (PM(69) , BM(3) , PM(197) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
               reset_PM , PM(138) , PM(139) , term(69) , term(197) , dec(69) , dec(197));
acsu_pair71 : acsu_pair port map
              (PM(70) , BM(5) , PM(198) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
               reset_PM , PM(140) , PM(141) , term(70) , term(198) , dec(70) , dec(198));
acsu_pair72 : acsu_pair port map
              (PM(71) , BM(6) , PM(199) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
               reset_PM , PM(142) , PM(143) , term(71) , term(199) , dec(71) , dec(199));
acsu_pair73 : acsu_pair port map
```

```
                (PM(72), BM(4), PM(200), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                 reset_PM, PM(144), PM(145), term(72), term(200), dec(72), dec(200));
acsu_pair74 : acsu_pair port map
                (PM(73), BM(7), PM(201), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                 reset_PM, PM(146), PM(147), term(73), term(201), dec(73), dec(201));
acsu_pair75 : acsu_pair port map
                (PM(74), BM(1), PM(202), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                 reset_PM, PM(148), PM(149), term(74), term(202), dec(74), dec(202));
acsu_pair76 : acsu_pair port map
                (PM(75), BM(2), PM(203), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                 reset_PM, PM(150), PM(151), term(75), term(203), dec(75), dec(203));
acsu_pair77 : acsu_pair port map
                (PM(76), BM(2), PM(204), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                 reset_PM, PM(152), PM(153), term(76), term(204), dec(76), dec(204));
acsu_pair78 : acsu_pair port map
                (PM(77), BM(1), PM(205), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                 reset_PM, PM(154), PM(155), term(77), term(205), dec(77), dec(205));
acsu_pair79 : acsu_pair port map
                (PM(78), BM(7), PM(206), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                 reset_PM, PM(156), PM(157), term(78), term(206), dec(78), dec(206));
acsu_pair80 : acsu_pair port map
                (PM(79), BM(4), PM(207), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                 reset_PM, PM(158), PM(159), term(79), term(207), dec(79), dec(207));
acsu_pair81 : acsu_pair port map
                (PM(80), BM(3), PM(208), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
                 reset_PM, PM(160), PM(161), term(80), term(208), dec(80), dec(208));
acsu_pair82 : acsu_pair port map
                (PM(81), BM(0), PM(209), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
                 reset_PM, PM(162), PM(163), term(81), term(209), dec(81), dec(209));
acsu_pair83 : acsu_pair port map
                (PM(82), BM(6), PM(210), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
                 reset_PM, PM(164), PM(165), term(82), term(210), dec(82), dec(210));
acsu_pair84 : acsu_pair port map
                (PM(83), BM(5), PM(211), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
                 reset_PM, PM(166), PM(167), term(83), term(211), dec(83), dec(211));
acsu_pair85 : acsu_pair port map
                (PM(84), BM(5), PM(212), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
                 reset_PM, PM(168), PM(169), term(84), term(212), dec(84), dec(212));
acsu_pair86 : acsu_pair port map
                (PM(85), BM(6), PM(213), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
                 reset_PM, PM(170), PM(171), term(85), term(213), dec(85), dec(213));
acsu_pair87 : acsu_pair port map
                (PM(86), BM(0), PM(214), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
                 reset_PM, PM(172), PM(173), term(86), term(214), dec(86), dec(214));
acsu_pair88 : acsu_pair port map
                (PM(87), BM(3), PM(215), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
                 reset_PM, PM(174), PM(175), term(87), term(215), dec(87), dec(215));
acsu_pair89 : acsu_pair port map
                (PM(88), BM(1), PM(216), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
                 reset_PM, PM(176), PM(177), term(88), term(216), dec(88), dec(216));
acsu_pair90 : acsu_pair port map
                (PM(89), BM(2), PM(217), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
                 reset_PM, PM(178), PM(179), term(89), term(217), dec(89), dec(217));
acsu_pair91 : acsu_pair port map
                (PM(90), BM(4), PM(218), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
                 reset_PM, PM(180), PM(181), term(90), term(218), dec(90), dec(218));
acsu_pair92 : acsu_pair port map
                (PM(91), BM(7), PM(219), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
                 reset_PM, PM(182), PM(183), term(91), term(219), dec(91), dec(219));
```

```
acsu_pair93 : acsu_pair port map
            (PM(92), BM(7), PM(220), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
             reset_PM, PM(184), PM(185), term(92), term(220), dec(92), dec(220));
acsu_pair94 : acsu_pair port map
            (PM(93), BM(4), PM(221), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
             reset_PM, PM(186), PM(187), term(93), term(221), dec(93), dec(221));
acsu_pair95 : acsu_pair port map
            (PM(94), BM(2), PM(222), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
             reset_PM, PM(188), PM(189), term(94), term(222), dec(94), dec(222));
acsu_pair96 : acsu_pair port map
            (PM(95), BM(1), PM(223), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
             reset_PM, PM(190), PM(191), term(95), term(223), dec(95), dec(223));
acsu_pair97 : acsu_pair port map
            (PM(96), BM(2), PM(224), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
             reset_PM, PM(192), PM(193), term(96), term(224), dec(96), dec(224));
acsu_pair98 : acsu_pair port map
            (PM(97), BM(1), PM(225), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
             reset_PM, PM(194), PM(195), term(97), term(225), dec(97), dec(225));
acsu_pair99 : acsu_pair port map
            (PM(98), BM(7), PM(226), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
             reset_PM, PM(196), PM(197), term(98), term(226), dec(98), dec(226));
acsu_pair100: acsu_pair port map
            (PM(99), BM(4), PM(227), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
             reset_PM, PM(198), PM(199), term(99), term(227), dec(99), dec(227));
acsu_pair101: acsu_pair port map
            (PM(100), BM(4), PM(228), BM(3), clk, reset_carry, msb_BMjp(3), comp_enable,
             reset_PM, PM(200), PM(201), term(100), term(228), dec(100), dec(228));
acsu_pair102: acsu_pair port map
            (PM(101), BM(7), PM(229), BM(0), clk, reset_carry, msb_BMjp(0), comp_enable,
             reset_PM, PM(202), PM(203), term(101), term(229), dec(101), dec(229));
acsu_pair103: acsu_pair port map
            (PM(102), BM(1), PM(230), BM(6), clk, reset_carry, msb_BMjp(6), comp_enable,
             reset_PM, PM(204), PM(205), term(102), term(230), dec(102), dec(230));
acsu_pair104: acsu_pair port map
            (PM(103), BM(2), PM(231), BM(5), clk, reset_carry, msb_BMjp(5), comp_enable,
             reset_PM, PM(206), PM(207), term(103), term(231), dec(103), dec(231));
acsu_pair105: acsu_pair port map
            (PM(104), BM(0), PM(232), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
             reset_PM, PM(208), PM(209), term(104), term(232), dec(104), dec(232));
acsu_pair106: acsu_pair port map
            (PM(105), BM(3), PM(233), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
             reset_PM, PM(210), PM(211), term(105), term(233), dec(105), dec(233));
acsu_pair107: acsu_pair port map
            (PM(106), BM(5), PM(234), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
             reset_PM, PM(212), PM(213), term(106), term(234), dec(106), dec(234));
acsu_pair108: acsu_pair port map
            (PM(107), BM(6), PM(235), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
             reset_PM, PM(214), PM(215), term(107), term(235), dec(107), dec(235));
acsu_pair109: acsu_pair port map
            (PM(108), BM(6), PM(236), BM(1), clk, reset_carry, msb_BMjp(1), comp_enable,
             reset_PM, PM(216), PM(217), term(108), term(236), dec(108), dec(236));
acsu_pair110: acsu_pair port map
            (PM(109), BM(5), PM(237), BM(2), clk, reset_carry, msb_BMjp(2), comp_enable,
             reset_PM, PM(218), PM(219), term(109), term(237), dec(109), dec(237));
acsu_pair111: acsu_pair port map
            (PM(110), BM(3), PM(238), BM(4), clk, reset_carry, msb_BMjp(4), comp_enable,
             reset_PM, PM(220), PM(221), term(110), term(238), dec(110), dec(238));
acsu_pair112: acsu_pair port map
            (PM(111), BM(0), PM(239), BM(7), clk, reset_carry, msb_BMjp(7), comp_enable,
             reset_PM , PM(222) , PM(223) , term(111) , term(239) , dec(111) , dec(239));
acsu_pair113 : acsu_pair port map
            (PM(112) , BM(7) , PM(240) , BM(0) , clk , reset_carry , msb_BMjp(0) , comp_enable ,
             reset_PM , PM(224) , PM(225) , term(112) , term(240) , dec(112) , dec(240));
acsu_pair114 : acsu_pair port map
            (PM(113) , BM(4) , PM(241) , BM(3) , clk , reset_carry , msb_BMjp(3) , comp_enable ,
             reset_PM , PM(226) , PM(227) , term(113) , term(241) , dec(113) , dec(241));
acsu_pair115 : acsu_pair port map
            (PM(114) , BM(2) , PM(242) , BM(5) , clk , reset_carry , msb_BMjp(5) , comp_enable ,
             reset_PM , PM(228) , PM(229) , term(114) , term(242) , dec(114) , dec(242));
acsu_pair116 : acsu_pair port map
            (PM(115) , BM(1) , PM(243) , BM(6) , clk , reset_carry , msb_BMjp(6) , comp_enable ,
             reset_PM , PM(230) , PM(231) , term(115) , term(243) , dec(115) , dec(243));
acsu_pair117 : acsu_pair port map
            (PM(116) , BM(1) , PM(244) , BM(6) , clk , reset_carry , msb_BMjp(6) , comp_enable ,
             reset_PM , PM(232) , PM(233) , term(116) , term(244) , dec(116) , dec(244));
acsu_pair118 : acsu_pair port map
            (PM(117) , BM(2) , PM(245) , BM(5) , clk , reset_carry , msb_BMjp(5) , comp_enable ,
             reset_PM , PM(234) , PM(235) , term(117) , term(245) , dec(117) , dec(245));
acsu_pair119 : acsu_pair port map
            (PM(118) , BM(4) , PM(246) , BM(3) , clk , reset_carry , msb_BMjp(3) , comp_enable ,
             reset_PM , PM(236) , PM(237) , term(118) , term(246) , dec(118) , dec(246));
acsu_pair120 : acsu_pair port map
            (PM(119) , BM(7) , PM(247) , BM(0) , clk , reset_carry , msb_BMjp(0) , comp_enable ,
             reset_PM , PM(238) , PM(239) , term(119) , term(247) , dec(119) , dec(247));
acsu_pair121 : acsu_pair port map
            (PM(120) , BM(5) , PM(248) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
             reset_PM , PM(240) , PM(241) , term(120) , term(248) , dec(120) , dec(248));
acsu_pair122 : acsu_pair port map
            (PM(121) , BM(6) , PM(249) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
             reset_PM , PM(242) , PM(243) , term(121) , term(249) , dec(121) , dec(249));
acsu_pair123 : acsu_pair port map
            (PM(122) , BM(0) , PM(250) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
             reset_PM , PM(244) , PM(245) , term(122) , term(250) , dec(122) , dec(250));
acsu_pair124 : acsu_pair port map
            (PM(123) , BM(3) , PM(251) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
             reset_PM , PM(246) , PM(247) , term(123) , term(251) , dec(123) , dec(251));
acsu_pair125 : acsu_pair port map
            (PM(124) , BM(3) , PM(252) , BM(4) , clk , reset_carry , msb_BMjp(4) , comp_enable ,
             reset_PM , PM(248) , PM(249) , term(124) , term(252) , dec(124) , dec(252));
acsu_pair126 : acsu_pair port map
            (PM(125) , BM(0) , PM(253) , BM(7) , clk , reset_carry , msb_BMjp(7) , comp_enable ,
             reset_PM , PM(250) , PM(251) , term(125) , term(253) , dec(125) , dec(253));
acsu_pair127 : acsu_pair port map
            (PM(126) , BM(6) , PM(254) , BM(1) , clk , reset_carry , msb_BMjp(1) , comp_enable ,
             reset_PM , PM(252) , PM(253) , term(126) , term(254) , dec(126) , dec(254));
acsu_pair128 : acsu_pair port map
            (PM(127) , BM(5) , PM(255) , BM(2) , clk , reset_carry , msb_BMjp(2) , comp_enable ,
             reset_PM , PM(254) , PM(255) , term(127) , term(255) , dec(127) , dec(255));

end struct;
```

A.6 Acstosm.vhd

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

```
entity acstosm_mux is
port(   clk,reset        : in std_logic;
        decision_in      : in std_logic_vector(0 to 255);
        termination_in   : in std_logic_vector(0 to 255);
        pointer          : in std_logic_vector( 7 downto 0);
        decision_out , termination_out: out std_logic
     );
end acstosm_mux;

-- The ACSTOSM module serves the routing of the desired decision bit
-- into the pointer for successive updating of the pointer and for
-- providing the decoded output.
-- It operates as a 256 to 1 decoder and the pointer is its select.

architecture comb of acstosm_mux is
      signal termination_register: std_logic;
begin

process(clk,reset,pointer, decision_in, termination_in,
        termination_register )
--variable termination_variable:std_logic_vector;
 begin
    if reset = '1' then
        termination_register <='0' ;
    elsif clk='0' and clk'event then
   --termination_variable:=termination_register;
        case pointer is
        when "00000000" =>       decision_out <= decision_in(0);
    termination_register <= termination_in(0) OR termination_register;
        when "00000001" =>       decision_out <= decision_in(1);
    termination_register <= termination_in(1) OR termination_register;
        when "00000010" =>       decision_out <= decision_in(2);
    termination_register <= termination_in(2) OR termination_register;
        when "00000011" =>       decision_out <= decision_in(3);
    termination_register <= termination_in(3) OR termination_register;
        when "00000100" =>       decision_out <= decision_in(4);
    termination_register <= termination_in(4) OR termination_register;
        when "00000101" =>       decision_out <= decision_in(5);
    termination_register <= termination_in(5) OR termination_register;
        when "00000110" =>       decision_out <= decision_in(6);
    termination_register <= termination_in(6) OR termination_register;
        when "00000111" =>       decision_out <= decision_in(7);
    termination_register <= termination_in(7) OR termination_register;
        when "00001000" =>       decision_out <= decision_in(8);
    termination_register <= termination_in(8) OR termination_register;
        when "00001001" =>       decision_out <= decision_in(9);
    termination_register <= termination_in(9) OR termination_register;
        when "00001010" =>       decision_out <= decision_in(10);
    termination_register <= termination_in(10) OR termination_register;
        when "00001011" =>       decision_out <= decision_in(11);
    termination_register <= termination_in(11) OR termination_register;
        when "00001100" =>       decision_out <= decision_in(12);
    termination_register <= termination_in(12) OR termination_register;
        when "00001101" =>       decision_out <= decision_in(13);
    termination_register <= termination_in(13) OR termination_register;
        when "00001110" =>       decision_out <= decision_in(14);
    termination_register <= termination_in(14) OR termination_register;
        when "00001111" =>       decision_out <= decision_in(15);
    termination_register <= termination_in(15) OR termination_register;
        when "00010000" =>      decision_out <= decision_in(16);
    termination_register <= termination_in(16) OR termination_register;
        when "00010001" =>      decision_out <= decision_in(17);
    termination_register <= termination_in(17) OR termination_register;
        when "00010010" =>      decision_out <= decision_in(18);
    termination_register <= termination_in(18) OR termination_register;
        when "00010011" =>      decision_out <= decision_in(19);
    termination_register <= termination_in(19) OR termination_register;
        when "00010100" =>      decision_out <= decision_in(20);
    termination_register <= termination_in(20) OR termination_register;
        when "00010101" =>      decision_out <= decision_in(21);
    termination_register <= termination_in(21) OR termination_register;
        when "00010110" =>      decision_out <= decision_in(22);
    termination_register <= termination_in(22) OR termination_register;
        when "00010111" =>      decision_out <= decision_in(23);
    termination_register <= termination_in(23) OR termination_register;
        when "00011000" =>      decision_out <= decision_in(24);
    termination_register <= termination_in(24) OR termination_register;
        when "00011001" =>      decision_out <= decision_in(25);
    termination_register <= termination_in(25) OR termination_register;
        when "00011010" =>      decision_out <= decision_in(26);
    termination_register <= termination_in(26) OR termination_register;
        when "00011011" =>      decision_out <= decision_in(27);
    termination_register <= termination_in(27) OR termination_register;
        when "00011100" =>      decision_out <= decision_in(28);
    termination_register <= termination_in(28) OR termination_register;
        when "00011101" =>      decision_out <= decision_in(29);
    termination_register <= termination_in(29) OR termination_register;
        when "00011110" =>      decision_out <= decision_in(30);
    termination_register <= termination_in(30) OR termination_register;
        when "00011111" =>      decision_out <= decision_in(31);
    termination_register <= termination_in(31) OR termination_register;
        when "00100000" =>      decision_out <= decision_in(32);
    termination_register <= termination_in(32) OR termination_register;
        when "00100001" =>      decision_out <= decision_in(33);
    termination_register <= termination_in(33) OR termination_register;
        when "00100010" =>      decision_out <= decision_in(34);
    termination_register <= termination_in(34) OR termination_register;
        when "00100011" =>      decision_out <= decision_in(35);
    termination_register <= termination_in(35) OR termination_register;
        when "00100100" =>      decision_out <= decision_in(36);
    termination_register <= termination_in(36) OR termination_register;
        when "00100101" =>      decision_out <= decision_in(37);
    termination_register <= termination_in(37) OR termination_register;
        when "00100110" =>      decision_out <= decision_in(38);
    termination_register <= termination_in(38) OR termination_register;
        when "00100111" =>      decision_out <= decision_in(39);
    termination_register <= termination_in(39) OR termination_register;
        when "00101000" =>      decision_out <= decision_in(40);
    termination_register <= termination_in(40) OR termination_register;
        when "00101001" =>      decision_out <= decision_in(41);
    termination_register <= termination_in(41) OR termination_register;
        when "00101010" =>      decision_out <= decision_in(42);
    termination_register <= termination_in(42) OR termination_register;
        when "00101011" =>      decision_out <= decision_in(43);
    termination_register <= termination_in(43) OR termination_register;
        when "00101100" =>      decision_out <= decision_in(44);
    termination_register <= termination_in(44) OR termination_register;
        when "00101101" =>      decision_out <= decision_in(45);
```

```
termination_register <= termination_in(45) OR termination_register;
    when "00101110" =>      decision_out <= decision_in(46);
termination_register <= termination_in(46) OR termination_register;
    when "00101111" =>      decision_out <= decision_in(47);
termination_register <= termination_in(47) OR termination_register;
    when "00110000" =>      decision_out <= decision_in(48);
termination_register <= termination_in(48) OR termination_register;
    when "00110001" =>      decision_out <= decision_in(49);
termination_register <= termination_in(49) OR termination_register;
    when "00110010" =>      decision_out <= decision_in(50);
termination_register <= termination_in(50) OR termination_register;
    when "00110011" =>      decision_out <= decision_in(51);
termination_register <= termination_in(51) OR termination_register;
    when "00110100" =>      decision_out <= decision_in(52);
termination_register <= termination_in(52) OR termination_register;
    when "00110101" =>      decision_out <= decision_in(53);
termination_register <= termination_in(53) OR termination_register;
    when "00110110" =>      decision_out <= decision_in(54);
termination_register <= termination_in(54) OR termination_register;
    when "00110111" =>      decision_out <= decision_in(55);
termination_register <= termination_in(55) OR termination_register;
    when "00111000" =>      decision_out <= decision_in(56);
termination_register <= termination_in(56) OR termination_register;
    when "00111001" =>      decision_out <= decision_in(57);
termination_register <= termination_in(57) OR termination_register;
    when "00111010" =>      decision_out <= decision_in(58);
termination_register <= termination_in(58) OR termination_register;
    when "00111011" =>      decision_out <= decision_in(59);
termination_register <= termination_in(59) OR termination_register;
    when "00111100" =>      decision_out <= decision_in(60);
termination_register <= termination_in(60) OR termination_register;
    when "00111101" =>      decision_out <= decision_in(61);
termination_register <= termination_in(61) OR termination_register;
    when "00111110" =>      decision_out <= decision_in(62);
termination_register <= termination_in(62) OR termination_register;
    when "00111111" =>      decision_out <= decision_in(63);
termination_register <= termination_in(63) OR termination_register;
    when "01000000" =>      decision_out <= decision_in(64)
     termination_register <= termination_in(64) OR termination_register;
    when "01000001" =>      decision_out <= decision_in(65);
termination_register <= termination_in(65) OR termination_register;
    when "01000010" =>      decision_out <= decision_in(66);
termination_register <= termination_in(66) OR termination_register;
    when "01000011" =>      decision_out <= decision_in(67);
termination_register <= termination_in(67) OR termination_register;
    when "01000100" =>      decision_out <= decision_in(68);
termination_register <= termination_in(68) OR termination_register;
    when "01000101" =>      decision_out <= decision_in(69);
termination_register <= termination_in(69) OR termination_register;
    when "01000110" =>      decision_out <= decision_in(70);
termination_register <= termination_in(70) OR termination_register;
    when "01000111" =>      decision_out <= decision_in(71);
termination_register <= termination_in(71) OR termination_register;
    when "01001000" =>      decision_out <= decision_in(72);
termination_register <= termination_in(72) OR termination_register;
    when "01001001" =>      decision_out <= decision_in(73);
termination_register <= termination_in(73) OR termination_register;
    when "01001010" =>      decision_out <= decision_in(74);
termination_register <= termination_in(74) OR termination_register;
```

```
    when "01001011" =>       decision_out <= decision_in(75);
termination_register <= termination_in(75) OR termination_register;
    when "01001100" =>       decision_out <= decision_in(76);
termination_register <= termination_in(76) OR termination_register;
    when "01001101" =>       decision_out <= decision_in(77);
termination_register <= termination_in(77) OR termination_register;
    when "01001110" =>       decision_out <= decision_in(78);
termination_register <= termination_in(78) OR termination_register;
    when "01001111" =>       decision_out <= decision_in(79);
termination_register <= termination_in(79) OR termination_register;
    when "01010000" =>       decision_out <= decision_in(80);
termination_register <= termination_in(80) OR termination_register;
    when "01010001" =>       decision_out <= decision_in(81);
termination_register <= termination_in(81) OR termination_register;
    when "01010010" =>       decision_out <= decision_in(82);
termination_register <= termination_in(82) OR termination_register;
    when "01010011" =>       decision_out <= decision_in(83);
termination_register <= termination_in(83) OR termination_register;
    when "01010100" =>       decision_out <= decision_in(84);
termination_register <= termination_in(84) OR termination_register;
    when "01010101" =>       decision_out <= decision_in(85);
termination_register <= termination_in(85) OR termination_register;
    when "01010110" =>       decision_out <= decision_in(86);
termination_register <= termination_in(86) OR termination_register;
    when "01010111" =>       decision_out <= decision_in(87);
termination_register <= termination_in(87) OR termination_register;
    when "01011000" =>       decision_out <= decision_in(88);
termination_register <= termination_in(88) OR termination_register;
    when "01011001" =>       decision_out <= decision_in(89);
termination_register <= termination_in(89) OR termination_register;
    when "01011010" =>       decision_out <= decision_in(90);
termination_register <= termination_in(90) OR termination_register;
    when "01011011" =>       decision_out <= decision_in(91);
termination_register <= termination_in(91) OR termination_register;
    when "01011100" =>       decision_out <= decision_in(92);
termination_register <= termination_in(92) OR termination_register;
    when "01011101" =>       decision_out <= decision_in(93);
termination_register <= termination_in(93) OR termination_register;
    when "01011110" =>       decision_out <= decision_in(94);
termination_register <= termination_in(94) OR termination_register;
    when "01011111" =>       decision_out <= decision_in(95);
termination_register <= termination_in(95) OR termination_register;
    when "01100000" =>       decision_out <= decision_in(96);
termination_register <= termination_in(96) OR termination_register;
    when "01100001" =>       decision_out <= decision_in(97);
termination_register <= termination_in(97) OR termination_register;
    when "01100010" =>       decision_out <= decision_in(98);
termination_register <= termination_in(98) OR termination_register;
    when "01100011" =>       decision_out <= decision_in(99);
termination_register <= termination_in(99) OR termination_register;
    when "01100100" =>       decision_out <= decision_in(100);
termination_register <= termination_in(100) OR termination_register;
    when "01100101" =>       decision_out <= decision_in(101);
termination_register <= termination_in(101) OR termination_register;
    when "01100110" =>       decision_out <= decision_in(102);
termination_register <= termination_in(102) OR termination_register;
    when "01100111" =>       decision_out <= decision_in(103);
termination_register <= termination_in(103) OR termination_register;
    when "01101000" =>       decision_out <= decision_in(104);
```

```
termination_register <= termination_in(104) OR termination_register;
    when "01101001" =>       decision_out <= decision_in(105);
termination_register <= termination_in(105) OR termination_register;
    when "01101010" =>       decision_out <= decision_in(106);
termination_register <= termination_in(106) OR termination_register;
    when "01101011" =>       decision_out <= decision_in(107);
termination_register <= termination_in(107) OR termination_register;
    when "01101100" =>       decision_out <= decision_in(108);
termination_register <= termination_in(108) OR termination_register;
    when "01101101" =>       decision_out <= decision_in(109);
termination_register <= termination_in(109) OR termination_register;
    when "01101110" =>       decision_out <= decision_in(110);
termination_register <= termination_in(110) OR termination_register;
    when "01101111" =>       decision_out <= decision_in(111);
termination_register <= termination_in(111) OR termination_register;
    when "01110000" =>       decision_out <= decision_in(112);
termination_register <= termination_in(112) OR termination_register;
    when "01110001" =>       decision_out <= decision_in(113);
termination_register <= termination_in(113) OR termination_register;
    when "01110010" =>       decision_out <= decision_in(114);
termination_register <= termination_in(114) OR termination_register;
    when "01110011" =>       decision_out <= decision_in(115);
termination_register <= termination_in(115) OR termination_register;
    when "01110100" =>       decision_out <= decision_in(116);
termination_register <= termination_in(116) OR termination_register;
    when "01110101" =>       decision_out <= decision_in(117);
termination_register <= termination_in(117) OR termination_register;
    when "01110110" =>       decision_out <= decision_in(118);
termination_register <= termination_in(118) OR termination_register;
    when "01110111" =>       decision_out <= decision_in(119);
termination_register <= termination_in(119) OR termination_register;
    when "01111000" =>       decision_out <= decision_in(120);
termination_register <= termination_in(120) OR termination_register;
    when "01111001" =>       decision_out <= decision_in(121);
termination_register <= termination_in(121) OR termination_register;
    when "01111010" =>       decision_out <= decision_in(122);
termination_register <= termination_in(122) OR termination_register;
    when "01111011" =>       decision_out <= decision_in(123);
termination_register <= termination_in(123) OR termination_register;
    when "01111100" =>       decision_out <= decision_in(124);
termination_register <= termination_in(124) OR termination_register;
    when "01111101" =>       decision_out <= decision_in(125);
termination_register <= termination_in(125) OR termination_register;
    when "01111110" =>       decision_out <= decision_in(126);
termination_register <= termination_in(126) OR termination_register;
    when "01111111" =>       decision_out <= decision_in(127);
termination_register <= termination_in(127) OR termination_register;
    when "10000000" =>       decision_out <= decision_in(128);
termination_register <= termination_in(128) OR termination_register;
    when "10000001" =>       decision_out <= decision_in(129);
termination_register <= termination_in(129) OR termination_register;
    when "10000010" =>       decision_out <= decision_in(130);
termination_register <= termination_in(130) OR termination_register;
    when "10000011" =>       decision_out <= decision_in(131);
termination_register <= termination_in(131) OR termination_register;
    when "10000100" =>       decision_out <= decision_in(132);
termination_register <= termination_in(132) OR termination_register;
    when "10000101" =>       decision_out <= decision_in(133);
termination_register <= termination_in(133) OR termination_register;
```

```
    when "10000110" =>      decision_out <= decision_in(134);
termination_register <= termination_in(134) OR termination_register;
    when "10000111" =>      decision_out <= decision_in(135);
termination_register <= termination_in(135) OR termination_register;
    when "10001000" =>      decision_out <= decision_in(136);
termination_register <= termination_in(136) OR termination_register;
    when "10001001" =>      decision_out <= decision_in(137);
termination_register <= termination_in(137) OR termination_register;
    when "10001010" =>      decision_out <= decision_in(138);
termination_register <= termination_in(138) OR termination_register;
    when "10001011" =>      decision_out <= decision_in(139);
termination_register <= termination_in(139) OR termination_register;
    when "10001100" =>      decision_out <= decision_in(140);
termination_register <= termination_in(140) OR termination_register;
    when "10001101" =>      decision_out <= decision_in(141);
termination_register <= termination_in(141) OR termination_register;
    when "10001110" =>      decision_out <= decision_in(142);
    termination_register <= termination_in(142) OR termination_register;
    when "10001111" =>      decision_out <= decision_in(143);
termination_register <= termination_in(143) OR termination_register;
    when "10010000" =>      decision_out <= decision_in(144);
termination_register <= termination_in(144) OR termination_register;
    when "10010001" =>      decision_out <= decision_in(145);
termination_register <= termination_in(145) OR termination_register;
    when "10010010" =>      decision_out <= decision_in(146);
termination_register <= termination_in(146) OR termination_register;
    when "10010011" =>      decision_out <= decision_in(147);
termination_register <= termination_in(147) OR termination_register;
    when "10010100" =>      decision_out <= decision_in(148);
termination_register <= termination_in(148) OR termination_register;
    when "10010101" =>      decision_out <= decision_in(149);
termination_register <= termination_in(149) OR termination_register;
    when "10010110" =>      decision_out <= decision_in(150);
termination_register <= termination_in(150) OR termination_register;
    when "10010111" =>      decision_out <= decision_in(151);
termination_register <= termination_in(151) OR termination_register;
    when "10011000" =>      decision_out <= decision_in(152);
termination_register <= termination_in(152) OR termination_register;
    when "10011001" =>      decision_out <= decision_in(153);
termination_register <= termination_in(153) OR termination_register;
    when "10011010" =>      decision_out <= decision_in(154);
termination_register <= termination_in(154) OR termination_register;
    when "10011011" =>      decision_out <= decision_in(155);
termination_register <= termination_in(155) OR termination_register;
    when "10011100" =>      decision_out <= decision_in(156);
termination_register <= termination_in(156) OR termination_register;
    when "10011101" =>      decision_out <= decision_in(157);
termination_register <= termination_in(157) OR termination_register;
    when "10011110" =>      decision_out <= decision_in(158);
termination_register <= termination_in(158) OR termination_register;
    when "10011111" =>      decision_out <= decision_in(159);
termination_register <= termination_in(159) OR termination_register;
    when "10100000" =>      decision_out <= decision_in(160);
termination_register <= termination_in(160) OR termination_register;
    when "10100001" =>      decision_out <= decision_in(161);
termination_register <= termination_in(161) OR termination_register;
    when "10100010" =>      decision_out <= decision_in(162);
termination_register <= termination_in(162) OR termination_register;
    when "10100011" =>      decision_out <= decision_in(163);
```

```
termination_register <= termination_in(163) OR termination_register;
    when "10100100" =>      decision_out <= decision_in(164);
termination_register <= termination_in(164) OR termination_register;
    when "10100101" =>      decision_out <= decision_in(165);
termination_register <= termination_in(165) OR termination_register;
    when "10100110" =>      decision_out <= decision_in(166);
termination_register <= termination_in(166) OR termination_register;
    when "10100111" =>      decision_out <= decision_in(167);
termination_register <= termination_in(167) OR termination_register;
    when "10101000" =>      decision_out <= decision_in(168);
termination_register <= termination_in(168) OR termination_register;
    when "10101001" =>      decision_out <= decision_in(169);
termination_register <= termination_in(169) OR termination_register;
    when "10101010" =>      decision_out <= decision_in(170);
termination_register <= termination_in(170) OR termination_register;
    when "10101011" =>      decision_out <= decision_in(171);
termination_register <= termination_in(171) OR termination_register;
    when "10101100" =>      decision_out <= decision_in(172);
termination_register <= termination_in(172) OR termination_register;
    when "10101101" =>      decision_out <= decision_in(173);
termination_register <= termination_in(173) OR termination_register;
    when "10101110" =>      decision_out <= decision_in(174);
termination_register <= termination_in(174) OR termination_register;
    when "10101111" =>      decision_out <= decision_in(175);
termination_register <= termination_in(175) OR termination_register;
    when "10110000" =>      decision_out <= decision_in(176);
termination_register <= termination_in(176) OR termination_register;
    when "10110001" =>      decision_out <= decision_in(177);
termination_register <= termination_in(177) OR termination_register;
    when "10110010" =>      decision_out <= decision_in(178);
termination_register <= termination_in(178) OR termination_register;
    when "10110011" =>      decision_out <= decision_in(179);
termination_register <= termination_in(179) OR termination_register;
    when "10110100" =>      decision_out <= decision_in(180);
termination_register <= termination_in(180) OR termination_register;
    when "10110101" =>      decision_out <= decision_in(181);
termination_register <= termination_in(181) OR termination_register;
    when "10110110" =>      decision_out <= decision_in(182);
termination_register <= termination_in(182) OR termination_register;
    when "10110111" =>      decision_out <= decision_in(183);
termination_register <= termination_in(183) OR termination_register;
    when "10111000" =>      decision_out <= decision_in(184);
termination_register <= termination_in(184) OR termination_register;
    when "10111001" =>      decision_out <= decision_in(185);
termination_register <= termination_in(185) OR termination_register;
    when "10111010" =>      decision_out <= decision_in(186);
termination_register <= termination_in(186) OR termination_register;
    when "10111011" =>      decision_out <= decision_in(187);
termination_register <= termination_in(187) OR termination_register;
    when "10111100" =>      decision_out <= decision_in(188);
termination_register <= termination_in(188) OR termination_register;
    when "10111101" =>      decision_out <= decision_in(189);
termination_register <= termination_in(189) OR termination_register;
    when "10111110" =>      decision_out <= decision_in(190);
termination_register <= termination_in(190) OR termination_register;
    when "10111111" =>      decision_out <= decision_in(191);
termination_register <= termination_in(191) OR termination_register;
    when "11000000" =>      decision_out <= decision_in(192);
termination_register <= termination_in(192) OR termination_register;
```

```
    when "11000001" =>       decision_out <= decision_in(193);
termination_register <= termination_in(193) OR termination_register;
    when "11000010" =>       decision_out <= decision_in(194);
termination_register <= termination_in(194) OR termination_register;
    when "11000011" =>       decision_out <= decision_in(195);
termination_register <= termination_in(195) OR termination_register;
    when "11000100" =>       decision_out <= decision_in(196);
termination_register <= termination_in(196) OR termination_register;
    when "11000101" =>       decision_out <= decision_in(197);
termination_register <= termination_in(197) OR termination_register;
    when "11000110" =>       decision_out <= decision_in(198);
termination_register <= termination_in(198) OR termination_register;
    when "11000111" =>       decision_out <= decision_in(199);
termination_register <= termination_in(199) OR termination_register;
    when "11001000" =>       decision_out <= decision_in(200);
termination_register <= termination_in(200) OR termination_register;
    when "11001001" =>       decision_out <= decision_in(201);
termination_register <= termination_in(201) OR termination_register;
    when "11001010" =>       decision_out <= decision_in(202);
termination_register <= termination_in(202) OR termination_register;
    when "11001011" =>       decision_out <= decision_in(203);
termination_register <= termination_in(203) OR termination_register;
    when "11001100" =>       decision_out <= decision_in(204);
termination_register <= termination_in(204) OR termination_register;
    when "11001101" =>       decision_out <= decision_in(205);
termination_register <= termination_in(205) OR termination_register;
    when "11001110" =>       decision_out <= decision_in(206);
termination_register <= termination_in(206) OR termination_register;
    when "11001111" =>       decision_out <= decision_in(207);
termination_register <= termination_in(207) OR termination_register;
    when "11010000" =>       decision_out <= decision_in(208);
termination_register <= termination_in(208) OR termination_register;
    when "11010001" =>       decision_out <= decision_in(209);
termination_register <= termination_in(209) OR termination_register;
    when "11010010" =>       decision_out <= decision_in(210);
termination_register <= termination_in(210) OR termination_register;
    when "11010011" =>       decision_out <= decision_in(211);
termination_register <= termination_in(211) OR termination_register;
    when "11010100" =>       decision_out <= decision_in(212);
termination_register <= termination_in(212) OR termination_register;
    when "11010101" =>       decision_out <= decision_in(213);
termination_register <= termination_in(213) OR termination_register;
    when "11010110" =>       decision_out <= decision_in(214);
termination_register <= termination_in(214) OR termination_register;
    when "11010111" =>       decision_out <= decision_in(215);
termination_register <= termination_in(215) OR termination_register;
    when "11011000" =>       decision_out <= decision_in(216);
termination_register <= termination_in(216) OR termination_register;
    when "11011001" =>       decision_out <= decision_in(217);
termination_register <= termination_in(217) OR termination_register;
    when "11011010" =>       decision_out <= decision_in(218);
termination_register <= termination_in(218) OR termination_register;
    when "11011011" =>       decision_out <= decision_in(219);
termination_register <= termination_in(219) OR termination_register;
    when "11011100" =>       decision_out <= decision_in(220);
termination_register <= termination_in(220) OR termination_register;
    when "11011101" =>       decision_out <= decision_in(221);
termination_register <= termination_in(221) OR termination_register;
    when "11011110" =>       decision_out <= decision_in(222);
```

```
termination_register <= termination_in(222) OR termination_register;
    when "11011111" =>      decision_out <= decision_in(223);
termination_register <= termination_in(223) OR termination_register;
    when "11100000" =>      decision_out <= decision_in(224);
termination_register <= termination_in(224) OR termination_register;
    when "11100001" =>      decision_out <= decision_in(225);
termination_register <= termination_in(225) OR termination_register;
    when "11100010" =>      decision_out <= decision_in(226);
termination_register <= termination_in(226) OR termination_register;
    when "11100011" =>      decision_out <= decision_in(227);
termination_register <= termination_in(227) OR termination_register;
    when "11100100" =>      decision_out <= decision_in(228);
termination_register <= termination_in(228) OR termination_register;
    when "11100101" =>      decision_out <= decision_in(229);
termination_register <= termination_in(229) OR termination_register;
    when "11100110" =>      decision_out <= decision_in(230);
termination_register <= termination_in(230) OR termination_register;
    when "11100111" =>      decision_out <= decision_in(231);
termination_register <= termination_in(231) OR termination_register;
    when "11101000" =>      decision_out <= decision_in(232);
termination_register <= termination_in(232) OR termination_register;
    when "11101001" =>      decision_out <= decision_in(233);
termination_register <= termination_in(233) OR termination_register;
    when "11101010" =>      decision_out <= decision_in(234);
termination_register <= termination_in(234) OR termination_register;
    when "11101011" =>      decision_out <= decision_in(235);
termination_register <= termination_in(235) OR termination_register;
    when "11101100" =>      decision_out <= decision_in(236);
termination_register <= termination_in(236) OR termination_register;
    when "11101101" =>      decision_out <= decision_in(237);
termination_register <= termination_in(237) OR termination_register;
    when "11101110" =>      decision_out <= decision_in(238);
termination_register <= termination_in(238) OR termination_register;
    when "11101111" =>      decision_out <= decision_in(239);
termination_register <= termination_in(239) OR termination_register;
    when "11110000" =>      decision_out <= decision_in(240);
termination_register <= termination_in(240) OR termination_register;
    when "11110001" =>      decision_out <= decision_in(241);
termination_register <= termination_in(241) OR termination_register;
    when "11110010" =>      decision_out <= decision_in(242);
termination_register <= termination_in(242) OR termination_register;
    when "11110011" =>      decision_out <= decision_in(243);
termination_register <= termination_in(243) OR termination_register;
    when "11110100" =>      decision_out <= decision_in(244);
termination_register <= termination_in(244) OR termination_register;
    when "11110101" =>      decision_out <= decision_in(245);
termination_register <= termination_in(245) OR termination_register;
    when "11110110" =>      decision_out <= decision_in(246);
termination_register <= termination_in(246) OR termination_register;
    when "11110111" =>      decision_out <= decision_in(247);
termination_register <= termination_in(247) OR termination_register;
    when "11111000" =>      decision_out <= decision_in(248);
termination_register <= termination_in(248) OR termination_register;
    when "11111001" =>      decision_out <= decision_in(249);
termination_register <= termination_in(249) OR termination_register;
    when "11111010" =>      decision_out <= decision_in(250);
termination_register <= termination_in(250) OR termination_register;
    when "11111011" =>      decision_out <= decision_in(251);
termination_register <= termination_in(251) OR termination_register;
```

```
        when "11111100" =>       decision_out <= decision_in(252);
    termination_register <= termination_in(252) OR termination_register;
        when "11111101" =>       decision_out <= decision_in(253);
    termination_register <= termination_in(253) OR termination_register;
        when "11111110" =>       decision_out <= decision_in(254);
    termination_register <= termination_in(254) OR termination_register;
        when "11111111" =>       decision_out <= decision_in(255);
    termination_register <= termination_in(255) OR termination_register;
        when others =>    decision_out <='0'; termination_register <='0';
    end case;
end if;
end process;
termination_out<=termination_register;
end comb;
```

A.7 Bmu.vhd

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

-- The BMU is operating at the bit rate. It receives soft-decision
-- decoded bits in the range {-3,3}.
-- The output is a signed 5-bit number (in 2's complement format)

entity bmu is
port(   i0, i1, i2: in std_logic_vector (2 downto 0);
        bmu000, bmu001, bmu010, bmu011, bmu100, bmu101,
        bmu110, bmu111: out std_logic_vector (4 downto 0)
     );
end bmu;

architecture rtl of bmu is
begin
process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu000<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu001<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu010<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu011<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu100<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))+ conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu101<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))+
                    conv_integer(signed(i2)),5);
    bmu110<=temp000_1;
end process;

process (i0,i2,i1)
 variable temp000_0,temp000_1: std_logic_vector( 4 downto 0);
  begin
    temp000_0:=conv_std_logic_vector(-conv_integer(signed(i0))- conv_integer(signed(i1)),5);
    temp000_1:=conv_std_logic_vector(conv_integer(signed(temp000_0))-
                    conv_integer(signed(i2)),5);
    bmu111<=temp000_1;
end process;
end rtl;
```

A.8 Pointer.vhd

```
 library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity pointer is
generic (reset_value: integer:=0);
    --0 is just the default value;
    port(   msb: in std_logic_vector (7 downto 0);
        clk , reset , new_bit: in std_logic;
        pointer_out: out std_logic_vector (7 downto 0));
end pointer;


-- The pointer points to the current state in the decoding trellis.
-- The most significant bit of the pointer is circular and is determined
-- by the output of the msb module.

architecture rtl of pointer is
    signal pointer_register: std_logic_vector(7 downto 0);
begin
   process(reset,clk,pointer_register,new_bit)
     begin
    if reset ='1' then
        pointer_register<=conv_std_logic_vector(reset_value,8);
        pointer_out<=conv_std_logic_vector(reset_value,8);
    elsif clk='1' and clk'event then
        case msb is
            when "10000000" =>  pointer_register(7)<=new_bit;
    pointer_out<=pointer_register(6 downto 0) & new_bit;
            when "01000000" =>  pointer_register(6) <= new_bit;
    pointer_out<=pointer_register(5 downto 0) & pointer_register(7)&new_bit;
            when "00100000" =>  pointer_register(5) <= new_bit;
    pointer_out<=pointer_register(4 downto 0) &pointer_register(7 downto 6)&new_bit;
            when "00010000" =>  pointer_register(4) <= new_bit;
    pointer_out<=pointer_register(3 downto 0) &pointer_register(7 downto 5)&new_bit;
            when "00001000" =>  pointer_register(3) <= new_bit;
    pointer_out<=pointer_register(2 downto 0) &pointer_register(7 downto 4)&new_bit;
            when "00000100" =>  pointer_register(2) <= new_bit;
    pointer_out<=pointer_register(1 downto 0) &pointer_register(7 downto 3)&new_bit;
            when "00000010" =>  pointer_register(1) <= new_bit;
    pointer_out<=pointer_register(0) & pointer_register(7 downto 2) & new_bit;
            when "00000001" =>  pointer_register(0) <= new_bit;
    pointer_out<=pointer_register(7 downto 1) & new_bit;
            when others=>       pointer_register(7) <=new_bit;
    pointer_out<=pointer_register;
        end case;
    end if;
end process;
end rtl;
```

A.9 Comparator.vhd

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

entity comparator is
port (data_random, clk, data_out_in: in std_logic;
      status: out std_logic);
end comparator;

-- The comparator compares the delayed version of the input
-- to the encoder with the output of the decoder.

architecture rtl of comparator is
signal data_random_delayed: std_logic_vector(11 downto 0);
```

```
  --input to encoder
signal data_out_delayed: std_logic_vector (9 downto 0);
  --output of the decoder
signal clk_internal: std_logic;
begin
process(data_random, clk, data_out_in)
 begin
 if clk='0' and clk'event then
    data_random_delayed(11 downto 0)<= data_random & data_random_delayed(11 downto 1);
    data_out_delayed(9 downto 0)<= data_out_in & data_out_delayed(9 downto 1);
    end if;
end process;

clk_internal<=clk;

process(data_random, clk_internal, data_out_in, data_random_delayed)
 begin
    if clk_internal='1' then
      if data_random_delayed(9 downto 0)=data_out_delayed(9 downto 0) then
        status <='1';
      else
        status <='0';
      end if;
    end if;
end process;
end rtl;
```

A.10 Ledcounter.vhd

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

entity ledcounter is
port (  clk, reset, status: in std_logic;
    led: out std_logic_vector(3 downto 0);
    lt1_led: out std_logic_vector(7 downto 0));
end ledcounter;

-- This module serves testing the functionality of the design.
-- If the output of the comparator (status) indicates high, then the
-- decoder is functioning correctly and the ledcount signal counts up.
-- When the status signal is low indicating a wrong output,
-- the badcount signal counts up.
-- Eight leds on the RPP systems were reserved for the correct
-- functionality and  four other leds are reserved for the incorrect
-- functionality indication.

architecture rtl of ledcounter is
 signal ledcount: std_logic_vector(25 downto 0);
 signal badcount: std_logic_vector(2 downto 0);
begin
process (reset, clk,status,ledcount)
 begin
 if reset='1' then
    ledcount <= (others => '0');
 elsif (clk'event and clk='1') then
```

```
     if status='1' then
         ledcount<=unsigned(ledcount) + 1;
    end if;
 end if;
end process;

process (reset, clk,status,ledcount)
 begin
 if reset='1' then
    badcount <= (others => '0');
 elsif (clk'event and clk='1') then
    if status='0' then
        badcount<=unsigned(badcount) + 1;
    end if;
 end if;
end process;

process (ledcount(25 downto 23))
-- Only the three most significant bits were used, such that the
-- normal eye can follow the flashing of the leds.
 begin
 case ledcount(25 downto 23) is
    when "000" =>   LT1_LED <= "00000001";
    when "001" =>   LT1_LED <= "00000010";
    when "010" =>   LT1_LED <= "00000100";
    when "011" =>   LT1_LED <= "00001000";
    when "100" =>   LT1_LED <= "00010000";
    when "101" =>   LT1_LED <= "00100000";
    when "110" =>   LT1_LED <= "01000000";
    when "111" =>   LT1_LED <= "10000000";
    when others =>  LT1_LED <= "11111111";
  end case;
end process;

process (badcount(2 downto 0))
 begin
 case badcount(2 downto 0) is
    when "000" =>  LED     <= "0001";
    when "001" =>  LED     <= "0010";
    when "010" =>  LED     <= "0100";
    when "011" =>  LED     <= "1000";
    when "100" =>  LED     <= "0001";
    when "101" =>  LED     <= "0010";
    when "110" =>  LED     <= "0100";
    when "111" =>  LED     <= "1000";
    when others => LED     <= "1111";
  end case;
end process;

end rtl;
```

A.11 Viterbi.vhd

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

-- This is the top viterbi decoder.
```

```
entity viterbi is
port (clk,start_out: in std_logic;
      data_out, term_out: out std_logic;
      led: out std_logic_vector(3 downto 0);
      lt1_led: out std_logic_vector(7 downto 0)
      );
end viterbi;

architecture struct of viterbi is

component comparator
port (data_random, clk, data_out_in: in std_logic;
      status: out std_logic);
end component;

component encoder
port(   data_in, reset, clk: in std_logic;
        i0,i1,i2: out std_logic_vector(2 downto 0)
     );
end component;

component controller
port (  clk_w0, start: in std_logic;
        clk_mp, clk_encoder: out std_logic;
        reset_encoder,reset_mpl: out std_logic;
        reset_PM,reset_p2s,reset_comp_carry: out std_logic
     );
end component;

component pointer
generic (reset_value: integer:=0);
    --0 is just the default value;
port(   msb: in std_logic_vector (7 downto 0);
        clk, reset, new_bit: in std_logic;
        pointer_out: out std_logic_vector (7 downto 0)
     );
end component;

component acstosm_mux
port(   clk, reset        : in std_logic;
        decision_in       : in std_logic_vector(0 to 255);
        termination_in    : in std_logic_vector(0 to 255);
        pointer           : in std_logic_vector( 7 downto 0);
        decision_out, termination_out: out std_logic
     );
end component;

component bmu
port(   i0, i1, i2: in std_logic_vector (2 downto 0);
        bmu000, bmu001, bmu010, bmu011, bmu100, bmu101, bmu110,
        bmu111: out std_logic_vector (4 downto 0));
end component;

component p2s_block
port(   reset,clk: in std_logic;
         bmu000, bmu001, bmu010, bmu011, bmu100, bmu101, bmu110,
         bmu111: in std_logic_vector (4 downto 0);
         BM: out std_logic_vector (0 to 7)
     );
```

```
end component;

component acsu_block
port(    BM, msb_BMjp: std_logic_vector(0 to 7);
         clk, reset_carry : in std_logic;
         comp_enable, reset_PM: in std_logic;
         term, dec : out std_logic_vector(0 to 255)
     );
end component;

component msb
port(    clk, reset: in std_logic;
         msb: out std_logic_vector(7 downto 0)
     );
end component;

component lfsr
port(    clock : std_logic;
         reset : std_logic;
         data_out : out std_logic
     );
end component;

component ledcounter
port(    clk, reset, status: in std_logic;
         led: out std_logic_vector(3 downto 0);
         lt1_led: out std_logic_vector(7 downto 0)
     );
end component;

signal reset_mpl, reset_PM, reset_p2s, reset_encoder,dec_internal: std_logic;
signal status: std_logic;
signal data_random,reset_comp_carry, clk_mp,clk_encoder: std_logic;
signal dec, term: std_logic_vector(0 to 255);
signal BM : std_logic_vector(0 to 7);
signal msb_BMjp: std_logic_vector (0 to 7);
signal msb_signal, pointer_internal: std_logic_vector(7 downto 0);
signal bmu000, bmu001, bmu010, bmu011, bmu100, bmu101, bmu110,
       bmu111: std_logic_vector(4 downto 0);
signal i0,i1,i2: std_logic_vector(2 downto 0);
signal start:std_logic;


begin
start<=not(start_out);
ledcounteru: ledcounter port map (clk_mp, start, status, led, lt1_led);
lfsru:lfsr port map (clk_mp, start, data_random);
comparatoru: comparator port map (data_random, clk_mp, dec_internal,status);
controlleru: controller port map (clk, start, clk_mp,clk_encoder,
    reset_encoder, reset_mpl, reset_PM, reset_p2s, reset_comp_carry);
encoderu: encoder port map (data_random, reset_encoder, reset_p2s, i0,i1,i2);
bmuu: bmu port map (     i0=>i0,
           i1=>i1,
           i2=>i2,
           bmu000=>bmu000,
           bmu001=>bmu001,
           bmu010=>bmu010,
           bmu011=>bmu011,
           bmu100=>bmu100,
            bmu101=>bmu101,
            bmu110=>bmu110,
            bmu111=>bmu111);
pointeru: pointer generic map(0)
    port map(msb_signal,clk_mp,reset_mpl,dec_internal,
    pointer_internal);
acstosmu: acstosm_mux port map (reset_comp_carry,reset_mpl, dec,term,pointer_internal,
    dec_internal,term_out);
p2s_blocku: p2s_block port map (reset_comp_carry,clk,bmu000, bmu001,
        bmu010, bmu011, bmu100, bmu101, bmu110, bmu111,BM);
acsu_blocku: acsu_block port map (BM=>BM,
        msb_BMjp=>msb_BMjp,
        clk=>clk, reset_carry=>reset_comp_carry,
        comp_enable=>clk_encoder,reset_PM=>reset_PM,
        term=>term,dec=>dec   );
process (bmu000, bmu001, bmu010, bmu011, bmu100, bmu101, bmu110, bmu111, reset_comp_carry,
        msb_BMjp, start)
begin
if start='1' then
    msb_BMjp<="00000000";
elsif reset_comp_carry='0' and reset_comp_carry'event then
    msb_BMjp(0)<=bmu000(4);
    msb_BMjp(1)<=bmu001(4);
    msb_BMjp(2)<=bmu010(4);
    msb_BMjp(3)<=bmu011(4);
    msb_BMjp(4)<=bmu100(4);
    msb_BMjp(5)<=bmu101(4);
    msb_BMjp(6)<=bmu110(4);
    msb_BMjp(7)<=bmu111(4);
end if;
end process;
msbu: msb port map (clk_mp, reset_mpl, msb_signal);
data_out<=dec_internal;
end struct;
```

Appendix B

Inputs of the ACSUs

Table B.1: ACSU modules connections with PMs and BMs

ACSU	PMi, PMj	BMi, BMj
0	000000000, 100000000	000, 111
1	010000000, 110000000	011, 100
2	001000000, 101000000	101, 010
3	011000000, 111000000	110, 001
4	000100000, 100100000	110, 001
5	010100000, 110100000	101, 010
6	001100000, 101100000	011, 100
7	011100000, 111100000	000, 111
8	000010000, 100010000	010, 101
9	010010000, 110010000	001, 110
10	001010000, 101010000	111, 000
11	011010000, 111010000	100, 011
12	000110000, 100110000	100, 011
13	010110000, 110110000	111, 000
14	001110000, 101110000	001, 110
15	011110000, 111110000	010, 101
16	000001000, 100001000	101, 010
17	010001000, 110001000	110, 001
18	001001000, 101001000	000, 111
19	011001000, 111001000	011, 100
20	000101000, 100101000	011, 100
21	010101000, 110101000	000, 111
22	001101000, 101101000	110, 001
23	011101000, 111101000	101, 010
24	000011000, 100011000	111, 000
25	010011000, 110011000	100, 011
26	001011000, 101011000	010, 101
27	011011000, 111011000	001, 110
28	000111000, 100111000	001, 110
29	010111000, 110111000	010, 101
30	001111000, 101111000	100, 011
31	011111000,111111000	111,000
32	000000100,100000100	100,011
33	010000100,110000100	111,000
34	001000100,101000100	001,110
35	011000100,111000100	010,101

Table B.1: ACSU modules connections with PMs and BMs (continued)

ACSU	PMi, PMj	BMi, BMj
36	000100100,100100100	010,101
37	010100100,110100100	001,110
38	001100100,101100100	111,000
39	011100100,111100100	100,011
40	000010100,100010100	110,001
41	010010100,110010100	101,010
42	001010100,101010100	011,100
43	011010100,111010100	000,111
44	000110100,100110100	000,111
45	010110100,110110100	011,100
46	001110100,101110100	101,010
47	011110100,111110100	110,001
48	000001100,100001100	001,110
49	010001100,110001100	010,101
50	001001100,101001100	100,011
51	011001100,111001100	111,000
52	000101100,100101100	111,000
53	010101100,110101100	100,011
54	001101100,101101100	010,101
55	011101100,111101100	001,110
56	000011100,100011100	011,100
57	010011100,110011100	000,111
58	001011100,101011100	110,001
59	011011100,111011100	101,010
60	000111100,100111100	101,010
61	010111100,110111100	110,001
62	001111100,101111100	000,111
63	011111100,111111100	011,100
64	000000010,100000010	110,001
65	010000010,110000010	101,010
66	001000010,101000010	011,100
67	011000010,111000010	000,111
68	000100010,100100010	000,111
69	010100010,110100010	011,100
70	001100010,101100010	101,010
71	011100010,111100010	110,001
72	000010010,100010010	100,011
73	010010010,110010010	111,000
74	001010010,101010010	001,110
75	011010010,111010010	010,101
76	000110010,100110010	010,101
77	010110010,110110010	001,110
78	001110010,101110010	111,000
79	011110010,111110010	100,011
80	000001010,100001010	011,100
81	010001010,110001010	000,111
82	001001010,101001010	110,001
83	011001010,111001010	101,010
84	000101010,100101010	101,010
85	010101010,110101010	110,001
86	001101010,101101010	000,111
87	011101010,111101010	011,100
88	000011010,100011010	001,110
89	010011010,110011010	010,101

Table B.1: ACSU modules connections with PMs and BMs (continued)

ACSU	PMi, PMj	BMi, BMj
90	001011010,101011010	100,011
91	011011010,111011010	111,000
92	000111010,100111010	111,000
93	010111010,110111010	100,011
94	001111010,101111010	010,101
95	011111010,111111010	001,110
96	000000110,100000110	010,101
97	010000110,110000110	001,110
98	001000110,101000110	111,000
99	011000110,111000110	100,011
100	000100110,100100110	100,011
101	010100110,110100110	111,000
102	001100110,101100110	001,110
103	011100110,111100110	010,101
104	000010110,100010110	000,111
105	010010110,110010110	011,100
106	001010110,101010110	101,010
107	011010110,111010110	110,001
108	000110110,100110110	110,001
109	010110110,110110110	101,010
110	001110110,101110110	011,100
111	011110110,111110110	000,111
112	000001110,100001110	111,000
113	010001110,110001110	100,011
114	001001110,101001110	010,101
115	011001110,111001110	001,110
116	000101110,100101110	001,110
117	010101110,110101110	010101
118	001101110,101101110	100,011
119	011101110,111101110	111,000
120	000011110,100011110	101,010
121	010011110,110011110	110,001
122	001011110,101011110	000,111
123	011011110,111011110	011,100
124	000111110,100111110	011,100
125	010111110,110111110	000,111
126	001111110,101111110	110,001
127	011111110,111111110	101,010